信息与软件工程学院

企业实习总结报告

学　　号：2021091202022

姓　　名：刘治学

专业方向：软件工程（互联网+）

企业名称：成都中科合迅科技有限公司

实习岗位名称：实习生

企业指导教师：刘鹏江

院内指导教师：王伟东

摘要

关键词：

ABSTRACT

Keywords:

1 实习概况

1.1 实习公司情况和岗位职责

1.1.1 实习公司概况

本人实习单位，成都中科合迅科技有限公司，成立于 2013 年，总部位于成都市高新区科园南路 1 号海特国际广场 4 号楼 7-8 层，注册资本 2766 万元，现有员工 300 余人。是专业从事自主安全可控军用软件研发和网络空间军事装备研制的高科技企业。公司在北京、上海、武汉、西安设有分支机构，全国有超过 50 家的军队、军工科研客户。

公司的口号是“铸造军工品质，磨砺国产匠心”。国家目前的军工行业现状分析主要有三点。第一，一带一路、走出国门，必须建设能打胜仗的全球化现代军队，国家军队体制改革孕育重大行业发展机会。第二，现有科研体系无法满足国家强国强军梦想“军民融合、民参军”，给民营科技企业提供历史发展机遇；西方主要发达国家军民通用技术已超过 80%，军事专用技术已不到 15%。第三，我国国防军费 2017 年突破万亿大关，武器装备建设有望持续加速，战略空军、远洋海军、国防信息化成为军队建设的重中之重。

公司就在这样的背景下成立与发展，至今已过十年。公司的发展阶段大致可分三个阶段。第一个阶段，2013 年到 2017 年，是基础能力建设阶段，主要成绩有军工二级保密资格，GJB9001C-2017 质量管理体系，国家级高新技术产业等。第二个阶段，2017 年到 2020 年，是核心能力聚焦阶段，主要成绩有第二批国家级专精特新“小巨人”，四川省瞪羚企业，连续四年成都市新经济百强企业等。第三个阶段，2020 年至今，是核心业务产品化阶段，主要成绩有四川省雁阵企业——省发改委定向培育，四川省首批 30 家“新经济示范”四川省计算机学会科学技术一等奖等 8 个省、部级奖项等。公司还获得了很多其他的资质与荣誉。

1.1.2 岗位职责

公司的核心业务可以简单概括为“一个核心产品，两个基本能力，两个数字化方向”。具体见图 1-1：

图 1-1 公司核心业务

我在实习期间就参与在合迅智灵产品的研发中。这也是公司的核心产品。合迅智灵产品完成了在国产化软硬件环境下多平台的全覆盖，支持桌面、移动、嵌入式等多种平台，满足了军工科研院所适用的更高要求，并与银河麒麟、元心 OS、锐华 Reworks 等公司签订了战略合作协议，目前合迅智灵已进入国防科工局国产化工具软件推广应用目录。

公司合迅智灵产品的功能架构见图 1-2。进入公司以后我所在项目组一直在进行基础平台的研发工作。这一部分的具体工作将在具体任务部分进行阐述。

图 1-2 合迅智灵产品功能架构

1.2 项目的背景、意义以及国内外研究现状

LarkSDK 是一款通用 C++开发框架。框架的最终目的是简化应用程序的开发。为达此目的，LarkSDK 也提供了一整套语义直观明晰、设计模块化的 C++类库，为用户封装了操作系统平台与硬件差异、提供了开箱即用的基础编程工具。

和 STL、Boost 相比，LarkSDK 可以用于构建图形界面。

和 MFC、GTK 相比，LarkSDK 具备跨平台能力，同一套源码可以同时工作在 Linux 和 Windows 之下。

当然提到跨平台的 C++图形开发框架，Qt 在业界仍然是占据统治地位的产品。Qt 从上个世纪九十年代开始发展，直到现在都是业界的一面大旗，也为 LarkSDK 的研发提供了很多参考。Qt 所提供的功能已经非常完备，绝大多数从事 C++软件开发的工程师基本上都以 Qt 为第一选择。

但是可能由于战线过长或者历史遗留等问题，在研发 LarkSDK 的过程中，我们发现 Qt 中部分功能的设计也许并不是最优解，甚至部分功能的设计相当糟糕。同时加上国际形势和国产生态的需求，LarkSDK 应运而生。相比 Qt，LarkSDK 更轻量、更简单、更年轻，同时更懂国产生态。我们并不奢望取代 Qt，我们仅仅是为业界提供另一个选择。这条路并不容易，但我们坚信这是一条正确的路，难而正确的路。

随着自主可控 IT 行业重点部分建设日趋完善，国家现已正式步入产业链改革深水区。其中软件开发平台是 IT 行业的血液，它可以支持各种底层的芯片和操作系统，将下层硬件和操作系统细节封装起来，对上层应用软件提供统一、易用、简洁的开发接口与工具，是打通芯片、操作系统与应用软件的关键，更贯穿了应用软件的设计、开发、测试与维护全生命周期。

目前，国内在软件开发平台领域的缺失，导致国内软件行业在开发软件时不得不大量使用国外厂商的开发平台，从而遭遇信息安全存在隐患、国产操作系统环境适配性差、维护服务没有支持、版权使用存在风险等重要问题，没有国产软件开发平台支持，软件开发环节效率低、软件开发成果不稳定也成为国内软件行业普遍存在的问题。国产 C++软件开发平台的目的，是打造通用跨平台的国产化基础开发套件，提供对应用程序的关键共性运行逻辑的支持，进而构建完整的国产化软件集成开发环境，以构建完整的基于 C++语言的国产应用程序开发生态。这也是我们需要努力做一个属于国人自己的 LarkSDK 产品的真正原因。

LarkSDK 在设计之初就是致力于跨平台的，它封装了不同操作平台的特性和底层调用，隐藏了各个操作系统平台的不同之处，让用户可以专注于业务开发而无

需过多关心平台差异；同时，LarkSDK 也并非单纯的浅层 UI 库，它覆盖了从底层的操作系统事件监听与分发，到通用的跨平台用户组件的外观与行为定制；它既提供各种各样的底层代码工具，如常见元素容器和加解密算法，以提升 C++的开发效率和体验，又能够通过自带的用户界面框架直接构建图形用户程序，未来还将提供界面编辑器及自动化测试工具等，从而完成应用软件开发的全生命周期闭环。

和 Qt Framework 一样，LarkSDK 是一款单纯而完整的 C++开发框架。万丈高楼平地起，除必要的基础轮子外，LarkSDK 并不依赖任何其他框架而存在。

由于一些原因，目前国内的软件行业语境下，对于诸如语言、框架、工具、平台等技术概念，存在一定程度上的混淆使用。虽然不太愿意这么表达，但是在国内 LarkSDK 确实是唯一的一款通用基础开发框架，技术上并没有真正意义的竞品。这也是我们所看到的绝大部分需要使用 C++开发应用软件的场合都是 Qt 的原因。

而合迅智灵是一款全国产的、自主可控的、拥有完全自主知识产权的基础软件开发平台。LarkSDK 则是平台为应用软件开发者提供的底层软件开发框架。我们致力于成为 Qt 之外的第二选择。

不过值得一提的是，站在市场层面解决实际需求的视角下，确实存在一些产品，和 LarkSDK 存在一定的功能覆盖，例如：

1、统信 DTK

DTK(Development ToolKit)是统信基于 Qt 开发的一整套简单且实用的通用开发框架，处于统信 UOS 操作系统的核心位置。其提供丰富的开发接口与支持工具，满足日常图形应用、业务应用、系统定制应用的开发需求，提供 30 余个预定义组件，如统信 UOS 浏览器、音乐、邮件等 40 余款 UOS 应用均使用 DTK 开发。

其本质上是一个在 Qt 的基础之上构建的扩展组件库。利用 Qt 框架与操作系统底层对接，借助 Qt 的能力实现各种具体的用户组件。用户本质上还是在使用 Qt 开发。其具备的跨平台能力本质上也是 Qt 本身的能力。

2、华为 ArkUI

ArkUI 是一套用于构建图形用户界面的声明式 UI 开发框架。它使用极简的 UI 信息语法，提供丰富的 UI 组件及包含实时界面预览工具在内的集成开发环境等。提供基于 ArkTS 开发语言的应用程序接口，支持各种 HarmonyOS 设备。

严格说 ArkUI 和 LarkSDK 在技术上并无关联，其本质是一套专门用于鸿蒙 HarmonyOS 生态的开发工具链的一部分，完整的鸿蒙 HarmonyOS 生态包含 ArkTS(基于 TypeScript 的开发语言)、ArkUI (基于 ArkTS 的一套界面组件语言)、

ArkCompiler (用于处理 ArkTS 的编译工具)，以及 DevEco Studio(基于 VSCode 构建的集成开发环境)，构成完整的生态工具链。也即是说，ArkUI 是生态专有生态的开发工具，和直接面向操作系统底层的通用开发工具 LarkSDK 并不处于一条技术路线上。

3、致远电子 AWTK

AWTK 全称为 Toolkit AnyWhere，是 ZLG 倾心打造的一套基于 C 语言开发的 GUI 框架。旨在为用户提供一个功能强大、高效可靠、简单易用、可轻松做出炫酷效果的 GUI 引擎，支持跨平台同步开发，一次编程，到处编译，跨平台使用。

然而 AWTK 本质上也是一套基于 SDL(opens new window)构建的 GUI 库。其主要能力，如图形渲染、跨平台与底层交互等，均由 SDL 框架提供。

总的来讲，LarkSDK 产品的意义是重大的，我个人也很幸运能参与到项目的研发当中。

1.3 实习目标和具体任务

1.3.1 实习目标

对于实习目标，我将其分为公司目标和个人目标。对于公司目标，我希望严格按照公司的安排，全身心投入完成合迅智灵基础平台的研发。对于个人目标，我希望能在实习期间，能跟随前辈的脚步，保质保量完成好安排到的工作，同时能够在实习期间学到更多知识和技术，以学习为主，以功利为辅，提升培养自己的技术和能力。

1.3.2 具体任务

1、LarkSDK

前面提到，我目前参与在合迅智灵基础平台的产品研发当中，下面我对该基础平台的业务进行相关介绍。

合迅智灵基础平台，全称叫合迅智灵国产化基础开发套件，英文名叫LarkSDK。LarkSDK 是合迅智灵产品 LarkStudio5 的核心部件，是一套跨平台的C++基础开发库。

LarkSDK 对标 Qt。Qt 是一款历史悠久，发展稳定的跨平台 C++基础开发库。在全世界被广泛使用，但美中不足的是，该产品并不是国人开发。为顺应军工国产化的大趋势，公司在 2020 年起进入合迅智灵产品的研发，其中 LarkSDK 的部分作为基础平台，又是重中之重。

LarkSDK 分为以下三个主要子模块和三个扩展模块。

三个主要子模块：

(1) LarkSDK-Core (larkcore)：核心类库，包含对象模型、事件机制、线程管理和主程序框架等。

(2) LarkSDK-Util (larkutil)：实用跨平台工具类库，包含常用的数据结构和算法。

(3) LarkSDK-GUI (larkgui)：图形绘制类库，包含跨平台图形绘制接口、基础窗体和常用组件，以及多绘图引擎支持支持等。

三个扩展模块：

(4) LarkSDK-DB (larkdb)：数据库支持模块。

(5) LarkSDK-Network (larknetwork)：网络编程支持模块。

(6) LarkSDK-XML (larkxml)：XML 支持模块。

公司与本校本院某实验室在 2021 年末达成了合作，校方开始了 LarkSDK 部分的初步编写。到 23 年末为止已经经过一期和二期的阶段，已初步完成开发。但由

于学校学生对实际工程的接触较少以及理论知识不牢固，编写的代码是存在很多问题的。在我进入公司以前，公司的前辈们就针对部分问题进行了优化重构，但是整体来看仍难以达到标准。

因此在进入公司以后，我的第一份任务就是进行代码走查。何为代码走查？简单来讲就是读别人的代码发现问题，但是这其中的工作量和难度也是不小的。领导对我们的要求是对于该部分涉及到的内容，包括涉及到的其他代码，都要阅读和理解，然后汇报。虽然枯燥甚至有些难，但我确实学到了很多知识和技巧。我的老总曾经说过："提升技术的最好办法就是阅读代码。"我觉得说的有理，阅读别人的代码，首先是理解别人的思路，然后延申思考，思考他这里为什么写得好，写得不好，想想如果是自己该如何构思，如何下笔，这对一个问题从 0 到 1 的剖析是非常有帮助的，而我在学习的过程中，不仅要学习知识和技术，更重要的是培养工程上的思维和方法。

以下列出实习过程中的代码走查记录表 1-1：

表 1-1 代码走查表

迭代时间	走查代码
1.15 - 1.26	LStack、LQueue、LByteArray
1.29 - 2.8	LObject、LApplication、LSignal
2.19 - 3.1	线程管理、线程数据、互斥锁、读写锁部分
3.4 - 3.15	LPen、LBrush、LLinearGradient、LMenu、LMenuItem、LMenuItemSeparator
4.7 - 4.19	LarkXML 模块

既然做了代码走查，那么必然需要记录，后续优化重构。代码走查的问题是发现问题，记录问题，而最终的解决依赖于后续的代码优化重构。在实习期间中，我完成了线程部分代码、LPen、LBrush 等的优化重构，完成了栈 LStack 和队列 LQueue 容器、字符串列表（LStringList）、关联性容器（LHash、LMap、LSet）、日期时间数据类型（LDateTime、LDate、LTime）以及三元组容器（LTrio）等工具类代码的优化重构。同时我自主研究 QDir 和 QFileInfo 的设计，去掉设计不合理的地方，设计出了我们自己的 LFileSystemPath 和 LFileSystemEntry，与上面的结构分别对应。在组内成员的共同努力下，util 部分目前已经基本全部处理完毕。经过测试，目前功能稳定可用，具体见图 1-3。

模块	功能	C++ 类		质量情况评估	建议措施	具体问题与修改意见	优先级	处理人	处理状态	处理完成时间
LarkSDK-Util	序列容器	字符串列表	LStringList	勉强可用	代码重构	1. 后续继承LVector并重新实现相关功能 2. 直接添加lstring.h头文件，不使用前置声明 3. 继承LVector后调用父类方法实现拷贝操作运算符等 4. 继承LVector后再实现sort排序 5. 4.30 更新：删除所有无用的接口和除 = 号以外的运算符重载，只保留构造函数、析构函数、= 号运算符，以及 join 函数 6. 后续实现 LVariant 数据类型支持	中	刘当学	回归测试	2024Q2
	关联容器	哈希容器	LHash<Key, T>	不可用	重新实现	以 std::unordered_map 重构	中	刘当学	回归测试	2024Q2
		集合容器	LSet<T>	不可用	重新实现	以 std::unordered_set 重构	中	刘当学	回归测试	2024Q2
		字典容器	LMap<Key, T>	不可用	重新实现	以 std::map 重构	中	刘当学	回归测试	2024Q2
	日期时间数据类型	LDateTime LDate LTime		勉强可用	优化修改	1. 规范代码，合理使用内联和非内联 2. 目前存在操作修正，无需对无效时间抛异常 3. 多个接口中多次使用了将LDateTime转换为std::tm且代码量较大，提供私有接口直接调用即可 4. 删除isNull()和isValid()接口 5. 操作运算符重载！=、<、<=、==、>、>=的实现直接对私有数据进行比较即可，且在实现<=、>=时使用操作运算符重载>、<、=即可 6. time_t表示时间戳相关接口参数传递和返回值修改为int，部分转换时间戳接口保留time_t 7. LDate默认构造函数初始时间修改为0 8. 设置时间的相关接口判断改为三目运算符实现 9. LDate的fromString(const LString &lstring)接口需要对传入的错误格式进行处理 10. LDate的操作运算符<不使用std::tie实现，直接比较即可 11. 直接使用操作修正时间设置即可，不需要isValid判断且不需要返回bool	中	刘当学	回归测试	2024Q2

图 1-3 util 模块优化重构工作记录表

第三，课题调研。绝大多数的功能都是在产品不断的发展和迭代中，因为需求而应运而生的。面对一个新的需求或者问题，需要进行大量的调研才能解决问题。这不仅是工作的过程，更是一个学习和进步的过程。在实习期间，我很幸运参与了一些课题的完整调研。我从中学到了很多知识和技术，也为公司的产品贡献了自己的力量。具体见表 1-2。

表 1-2 课题调研表

迭代时间	调研课题
4.7 - 4.19	标准库 string 的 sso 优化对 LVector 插入影响的探究
4.22 - 4.30	LDir 和 LFileInfo 的语义和设计
5.13 - 5.24	Qt Graphics View Framework 预研
5.27 - 6.7	一些关于空间数据结构的简单研究与实现
6.17 - 6.28	在 X11 下使用 cairo 引擎绘制图形
7.1 - 7.12	使用 Woboq CodeBrowser 搭建源代码网站

2、LarkTestKit

LarkSDK 作为一个跨平台的 C++基础开发库。我们是采用 googleTest 开源框架进行测试的，为了做到国产化适配，故提出了自动化测试框架 LarkTestKit 的需求。该部分源代码交由校方某同学负责，提交了初版，我的任务是审核该部分代码，与校方同学和测试沟通，并设计编写测试用例。巧合的是，负责 LarkTestKit 的学长在四月中旬也来到公司也进行线下实习。因而在中期的工作中，我协助他一起完成 LarkTestKit 的工作，一起讨论 LarkTestKit 中的部分设计思路、实现方案等。目前 LarkTestKit 已经测试完毕，初步测试结果良好，功能稳定可用。

2 复杂工程问题和解决方案

本部分将针对上述具体任务当中的某个环节、某个步骤遇到的工程问题出发，分析问题的来龙去脉，并设计合理的解决方案应对，总结于此。有部分问题在早期的报告中已阐述过，这里不再赘述。

2.1 Qt Graphics View Framework 预研

该架构涉及到的最主要的三个类是 QGraphicsScene、QGraphicsView 和 QGraphicsItem。

2.1.1 整体流程

1. 绘制流程（QGraphicsItem->QGraphicsScene->QGraphicsView）

QGraphicsItem 当中保存了自身的“场景坐标”供 QGraphicsScene 进行管理。在绘制时，由 QGraphicsView 对象调用渲染方法，根据自身所设置的可视化相关属性，基于“视图坐标”确定将要绘制的 QGraphicsScene 当中有哪一部分“场景坐标”内的图元需要渲染，随后通过 QGraphicsScene 提供的方法，将属于这部分“场景坐标”内的图元(也就是 QGraphicsItem)全部找出，并渲染这些图元到可视化 viewport 中。

2. 事件流转（QGraphicsView->QGraphicsScene->QGraphicsItem）

由 QGraphicsView 绘制出的 viewport 是与用户直接交互的对象，用户发起的 UI 事件都由 QGraphicsView 首先接收，它在接收到事件以后对其中的部分参数进行适当的处理(如鼠标事件的坐标进行转换)，随后将事件转发给 QGraphicsScene 对象，由 QGraphicsScene 确定事件发送到哪个具体的图元(如鼠标事件发送到符合坐标位置的图元，键盘事件发送到当前焦点所在的图元)，图元在接收到事件以后作自行处理。

3. QGraphicsScene 的查询图元的 BSP 树相关算法

在实际的工程场景中，极有可能会出现一个 QGraphicsScene 管理非常多个 QGraphicsItem，例如几百万个，当用户与 QGraphicsView 交互的时候，需要经过 QGraphicsScene 将本事件传递到某个或者某些具体的图元，如果挨个遍历每个图元，并且判断是否触发事件，那么耗费的代价就太大了。我们希望做到的效率是几百万条数据在几毫秒以内能够确定目标图元，因此需要上相应的算法。

4. 图元实现优先级以及顺序和事件实现优先级以及顺序

QGraphicsItem 本质上是一个抽象父类，为子类规定了各种需要覆写的方法，

子类通过重写这些方法能够很好的被 QGraphicsScene 所管理，最终呈现出来。因此不仅支持官方提供的矩形、椭圆、文本框等，还能处理自定义类型的图元。

在现阶段，先不考虑纯虚的抽象类，先把 QGraphicsItem 当成矩形框来做，先和 QGraphicsScene 把流程跑通，后续在扩展的时候也能非常快就完成工作。

事件处理是一个大块。经讨论，事件处理的优先级是鼠标事件大于键盘事件大于拖拽事件。因此第一步实现的时候，先实现最基本的 handleMousePressEvent，当这个流程通了以后，后续的所有操作都是一模一样的了。

5. 整体实现顺序

先把 QGraphicsView 剔出来，QGraphicsView 实际上只负责可视化的绘制，真正做处理的操作在 QGraphicsScene 这一层，因此管理类 QGraphicsScene 和图元类 QGraphicsItem 是第一步需要实现的。

目前的工作计划经商讨以后，决定先调研 QGraphicsScene 和 QGraphicsItem 的源码，明确如何以最小的代价实现我们目前需要的功能，包括事件转发、绘制逻辑、坐标系统等，这样明确需求和设计以后再实现初版，搭好整体架构。

2.2.2 重要功能总结

1、QGraphicsScene

QGraphicsScene 类提供了一个管理大量 QGraphicsItem 的容器。

QGraphicsScene 类只管理所有的图元，若想要可视化场景，需要配合 QGraphicsView。

QGraphicsItem 作为 QGraphicsScene 类的图元，自然有很多接口都是围绕着 QGraphicsItem 来的，例如：

(1) 添加图元：除了最基本的 addItem()，还有更多针对特殊类型图元的 add*() 函数。

```
void addItem(QGraphicsItem *item);
QGraphicsEllipseItem *addEllipse(const QRectF &rect, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsLineItem *addLine(const QLineF &line, const QPen &pen = QPen());
QGraphicsPathItem *addPath(const QPainterPath &path, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsPixmapItem *addPixmap(const QPixmap &pixmap);
QGraphicsPolygonItem *addPolygon(const QPolygonF &polygon, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsRectItem *addRect(const QRectF &rect, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsTextItem *addText(const QString &text, const QFont &font = QFont());
QGraphicsSimpleTextItem *addSimpleText(const QString &text, const QFont &font = QFont());
QGraphicsProxyWidget *addWidget(QWidget *widget, Qt::WindowFlags wFlags = Qt::WindowFlags());
inline QGraphicsEllipseItem *addEllipse(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())
{ return addEllipse(rect: QRectF(x, y, w, h), pen, brush); }
inline QGraphicsLineItem *addLine(qreal x1, qreal y1, qreal x2, qreal y2, const QPen &pen = QPen())
{ return addLine(line: QLineF(x1, y1, x2, y2), pen); }
inline QGraphicsRectItem *addRect(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())
{ return addRect(rect: QRectF(x, y, w, h), pen, brush); }
```

图 2-1 addItem() 系列函数

(2) 查找图元：在大规模数据的情况下保证在极短的时间内查询到对应的图元，Qt 使用的是索引算法，使用的是 BSP（二进制空间分区）树。能够在极短的时间内在极大的数据规模中确定某个图元的位置，这也是 QGraphicsScene 的最大优势之一。

(3) 移除图元：removeItem()。

(4) 管理图元状态，处理图元选择和焦点处理等。

- setSelectionArea()：可以传递一个形状范围来选择图元项目
- clearSelection()：清除当前选择
- selectedItems()：获取被选择的图元的列表
- setFocusItem()、setFocus()：设置图元获取焦点
- focusItem()：获取当前焦点

(5) 接受事件：用户通过 QGraphicsView 触发事件，并通过 QGraphicsScene 将事件转发到对应的图元：例如鼠标按下、移动、释放和双击事件，鼠标悬停（LarkSDK 目前好像没有）、滚轮事件，键盘输入焦点和按键事件，拖拽事件等。

```
protected:
    bool event(QEvent *event) override;
    bool eventFilter(QObject *watched, QEvent *event) override;
    virtual void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
    virtual void dragEnterEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dragMoveEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dragLeaveEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dropEvent(QGraphicsSceneDragDropEvent *event);
    virtual void focusInEvent(QFocusEvent *event);
    virtual void focusOutEvent(QFocusEvent *event);
    virtual void helpEvent(QGraphicsSceneHelpEvent *event);
    virtual void keyPressEvent(QKeyEvent *event);
    virtual void keyReleaseEvent(QKeyEvent *event);
    virtual void mousePressEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
    virtual void wheelEvent(QGraphicsSceneWheelEvent *event);
    virtual void inputMethodEvent(QInputMethodEvent *event);
```

图 2-2 事件转发系列函数

QGraphicsScene 另一个重要的功能就是转发 QGraphicsView 的事件，通过一系列操作，例如通过鼠标点击的坐标计算出到底是选中了哪个图元，键盘事件对应的哪些图元具有焦点等，能够将这些事件转发给对应的图元，最后进入真正的事件循环进行处理。

2、QGraphicsView

QGraphicsView 类真正提供可视化 QGraphicsScene 的内容的功能，它在一个可滚动的 viewport 之内将一个 QGraphicsScene 中的内容实现可视化。

QGraphicsView 主要功能包括但不限于：

(1) 设置可视化操作的属性：QGraphicsView 中提供了大量可设置的属性用以指示在实现可视化操作时的各种具体事项，如 RenderHints 提供参数初始化用于绘制的 QPainter，Alignment 提供当前视图中所绘制的场景的对齐方式。

```
    QPainter::RenderHints renderHints() const;
    void setRenderHint(QPainter::RenderHint hint, bool enabled = true);
    void setRenderHints(QPainter::RenderHints hints);

    Qt::Alignment alignment() const;
    void setAlignment(Qt::Alignment alignment);

    ViewportAnchor transformationAnchor() const;
    void setTransformationAnchor(ViewportAnchor anchor);

    ViewportAnchor resizeAnchor() const;
    void setResizeAnchor(ViewportAnchor anchor);

    ViewportUpdateMode viewportUpdateMode() const;
    void setViewportUpdateMode(ViewportUpdateMode mode);

    OptimizationFlags optimizationFlags() const;
    void setOptimizationFlag(OptimizationFlag flag, bool enabled = true);
    void setOptimizationFlags(OptimizationFlags flags);

    DragMode dragMode() const;
    void setDragMode(DragMode mode);

#if QT_CONFIG(rubberband)
    Qt::ItemSelectionMode rubberBandSelectionMode() const;
    void setRubberBandSelectionMode(Qt::ItemSelectionMode mode);
    QRect rubberBandRect() const;
#endif

    CacheMode cacheMode() const;
    void setCacheMode(CacheMode mode);
    void resetCachedContent();

    bool isInteractive() const;
    void setInteractive(bool allowed);
```

图 2-3 设置可视化操作的接口

(2) 对场景（Scene）进行可视化与视觉效果调整：QGraphicsView 对象的成员方法 render 对场景进行可视化的绘制呈现在 viewport 中，并提供了一系列方法对 viewport 整体的视觉效果进行调整，如 centerOn 方法将滚动 viewport 中的内容以确保场景坐标 pos 在视图居中，fitInView 方法将缩放并滚动 viewport 中的内容使得场景内的矩形区域 rect 铺满当前 viewport。

```
void centerOn(const QPointF &pos);
inline void centerOn(qreal x, qreal y);
void centerOn(const QGraphicsItem *item);
void ensureVisible(const QRectF &rect, int xmargin = 50, int ymargin = 50);
inline void ensureVisible(qreal x, qreal y, qreal w, qreal h, int xmargin = 50, int ymargin = 50);
void ensureVisible(const QGraphicsItem *item, int xmargin = 50, int ymargin = 50);
void fitInView(const QRectF &rect, Qt::AspectRatioMode aspectRadioMode = Qt::IgnoreAspectRatio);
inline void fitInView(qreal x, qreal y, qreal w, qreal h,
                      Qt::AspectRatioMode aspectRadioMode = Qt::IgnoreAspectRatio);
void fitInView(const QGraphicsItem *item,
               Qt::AspectRatioMode aspectRadioMode = Qt::IgnoreAspectRatio);

void render(QPainter *painter, const QRectF &target = QRectF(), const QRect &source = QRect(),
            Qt::AspectRatioMode aspectRatioMode = Qt::KeepAspectRatio);
```

图 2-4 进行可视化与视觉效果调整的接口

(3) 管理“场景（Scene）坐标”与“视图（View）坐标”之间的数学关系，并提供方法对视图内容施行各种坐标变换。QGraphicsScene 对象当中的各个图元有其在 QGraphicsScene 中的坐标即“场景坐标”，它们代表了各个图元在 QGraphicsScene 中的位置信息；而 QGraphicsView 对各个图元进行绘制以及调整变换时则是通过由自身管理的“视图坐标”，它们代表了各个要绘制的图形在 viewport 中的位置信息。QGraphicsView 可以由用户设置“场景坐标”向“视图坐标”变换的方式，对 viewport 实现旋转、伸缩等坐标变换，同时由于 QGraphicsView 的绘图使用“视图坐标”，因此这个过程不会干扰图元自身的“场景坐标”。此外还提供 mapToScene/mapFromScene 方法供用户调用实现这两种坐标之间的数学换算。

(4) 接受鼠标和键盘的事件，并通过处理传递给 QGraphicsScene 对象，进而通过索引算法转发给对应的图元。

3、QGraphicsItem

QGraphicsItem 是所有图元的基类，可以派生出各种典型的形状（例如矩形、椭圆、文本等）和自定义的形状。

QGraphicsItem 主要功能包括但不限于：

(1) 接受 QGraphicsScene 传递的事件：进行事件处理，例如鼠标按下、移动、释放和双击事件，鼠标悬停、滚轮事件，键盘输入焦点和按键事件，拖拽事件等。

```
protected:
    bool sceneEvent(QEvent *event) override;
    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event) override;
    void contextMenuEvent(QGraphicsSceneContextMenuEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;
    void focusInEvent(QFocusEvent *event) override;
    void focusOutEvent(QFocusEvent *event) override;
    void dragEnterEvent(QGraphicsSceneDragDropEvent *event) override;
    void dragLeaveEvent(QGraphicsSceneDragDropEvent *event) override;
    void dragMoveEvent(QGraphicsSceneDragDropEvent *event) override;
    void dropEvent(QGraphicsSceneDragDropEvent *event) override;
    void inputMethodEvent(QInputMethodEvent *event) override;
    void hoverEnterEvent(QGraphicsSceneHoverEvent *event) override;
    void hoverMoveEvent(QGraphicsSceneHoverEvent *event) override;
    void hoverLeaveEvent(QGraphicsSceneHoverEvent *event) override;
```

图 2-5 QGraphicsScene 传递的事件的方法

(2) 坐标系统

每个 Item 都有自己的本地坐标系，一般以自身的中心为(0, 0)，自身的方向作为基准方向建立，多个 Item 的情况就如图。因此需要通过某些机制将不同 Item 的坐标联系在一起。

为了统一方便的管理，引入 parent-child 的关系。每个对象的变换都依赖于其父对象的坐标。子对象的 pos()接口返回的是其在父对象坐标系统中的坐标。子对象的坐标处理是首先通过父对象不断向上传递，最终得到一个真实的坐标。同理，父对象的坐标变换也会同理影响到子对象的真实位置（批量处理），但是注意子对象存储的坐标没有变化，这样就非常好维护了。

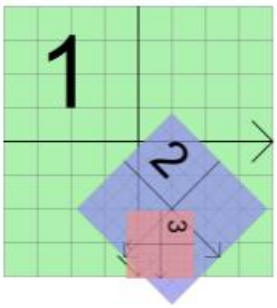

图 2-6 坐标变换示意图

(3) 坐标变换

QGraphicsItem 除了提供基本位置 pos()以外，还支持坐标变换，例如 setRotation()旋转，setScale()缩放等。

同理，父对象的变换会影响子对象，例如父对象顺时针转 90 度，子对象会跟着一起转 90 度。

(4) 提供分组功能（后续考虑）

通过前面的坐标系统可以知道，每个 QGraphicsItem 都有一个父对象，也可以有一系列子对象，这是类似于对象树机制的在构造的时候确定的关系。 当然也可以手动分组。QGraphicsItemGroup 是一个特殊的派生类，该类记录了一系列的图元为一个组，在该组的所有图元通过 Group 调用的时候移动、事件处理等都会进行批量处理。

(5) 提供编写自定义图元的接口（后续考虑）

创建一个 QGraphicsItem 的子类，然后覆写两个纯虚函数 boundingRect()返回该图元项目所绘制区域的估计值，paint()实现实际绘图。

boundingRect()返回的是一个 QRectF 类型，将该图元的外部估计边界定义为矩形，这个方法也可以为不同 Item 的范围做大致的估算，可以被其他的方法所调用，省去一些暴力查找的过程。当然对于真正的矩形 boundingRect()可以返回精确的范围，对于其他的曲线或者不规则的形状只能做大致的范围。

(6) 碰撞检测（后续考虑）

通过 shape()函数和 collidesWith()这两个虚拟函数，可以支持碰撞检测。shape()函数返回一个局部的坐标 QPainterPath。目前没有细节调研 QPainterPath，只知道 QPainterPath 记录了绘图的路径，比如 2D 图形的形状是由某些直线、曲线等构成的，通过这个能够确定图形的形状。

QGraphicsItem 会根据默认的 shape()函数自动处理碰撞检测，实现合理的效果，比如在碰撞区域应该如何进行绘制。如果用户想要定义自己的碰撞检测，可以通过 collidesWith()实现。

(7) 图元顺序（后续考虑）

难免会发生两个图元的范围出现重叠的情况。合理处理顺序决定了鼠标点击的时候哪些场景会接受鼠标事件。一个比较合理的想法是子对象位于父对象的顶部，而同级对象之间按照定义的顺序进行堆叠。例如添加对象 A、B，那么对象 B 位于 A 的顶部。这是比较符合自然逻辑的，Qt 也是这样做的。

Qt 提供了一些可以更改项目的排序方式的接口。例如可以在一个图元项目上调用 setZValue()，以将其显式堆叠在其他同级项目之上或之下。项的默认 Z 值为 0，具有相同 Z 值的项按插入顺序堆叠。还可以设置 ItemStacksBehindParent 标志以将子项堆叠在其父项之后。

2.2.3 2D BSP 树在 QGraphicsScene 中的应用

BSP 树构造一个 n 维空间到凸子空间的分层细分（a BSP tree is a heirarchical subdivisions of n dimensional space into convex subspaces）。每个节点都有一个前叶子节点和后叶子节点。从根节点开始，所有后续插入都由当前节点的超平面划分。在二维空间中，超平面是一条线。 在 3 维空间中，超平面是一个平面。BSP 树的最终目标是让超平面的分布情况满足“每个叶节点都在父节点超平面的前面或后面”（The end goal of a BSP tree if for the hyperplanes of the leaf nodes to be trivially "behind" or "infront" of the parent hyperplane.）。

BSP 树对于与静态图像的显示进行实时交互非常有用。 在渲染静态场景之前，需要计算 BSP 树。 可以非常快地（线性时间）遍历 BSP 树，以去除隐藏的表面和投射阴影。 通过一些工作，可以修改 BSP 树以处理场景中的动态事件。

下面是在对象空间构建 BSP 树的过程：

(1) 首先，确定要划分的世界区域以及其中包含的所有多边形。为了方便讨论，我们将使用一个二维世界。

(2) 创建一个根节点 L，该节点本身对应一个分区超平面(在二维世界中，分区超平面就是直线)；同时这个节点维护一个多边形列表，在节点刚刚创建时，多边形列表中保存着它对应的超平面所划分的目标区域(对 L 而言，它对应的直线所划分的目标区域就是整个二维世界)当中的所有多边形。

(3) 使用 L 对世界进行分区，假设分为了两个区域 A 和 B。在根节点上创建两个叶子节点分别对应 A 和 B，并将世界中的所有多边形移动到 A 或 B 的多边形列表中。遵循以下规则：

对于世界中的每个多边形：

- 如果该多边形完全位于 A 区域，请将该多边形移动到 A 区域的节点列表中。
- 如果该多边形完全位于 B 区域，请将该多边形移动到 B 区域的节点列表中。
- 如果该多边形与 L 相交，则将其拆分为两个多边形，并将它们移动到 A 和 B 的相应多边形列表中。这种情况下，算法必须找到多边形与分割线 L 的交点，以确定多边形的哪一部分位于哪个区域。
- 如果该多边形与 L 重合(也就是说，恰好是一条位于 L 上的线段)，请保持其仍位于节点 L 处的多边形列表中。

在上述步骤完成后，节点 A 和 B 的多边形列表已经生成，但是 A 和 B 尚未完成创建分区超平面并划分的步骤，也就是说，对于当前的节点 A，它的状态与步骤 1 中的“整个二维世界”一致(区域确定，包含的多边形确定)，只不过区域大小有

差别，B 同理。

(4) 在 A 和 B 上继续划分其所对应的二维区域，并将上述算法递归地应用于 A 和 B 的多边形列表。

最终的切分效果图如图 2-7 所示：

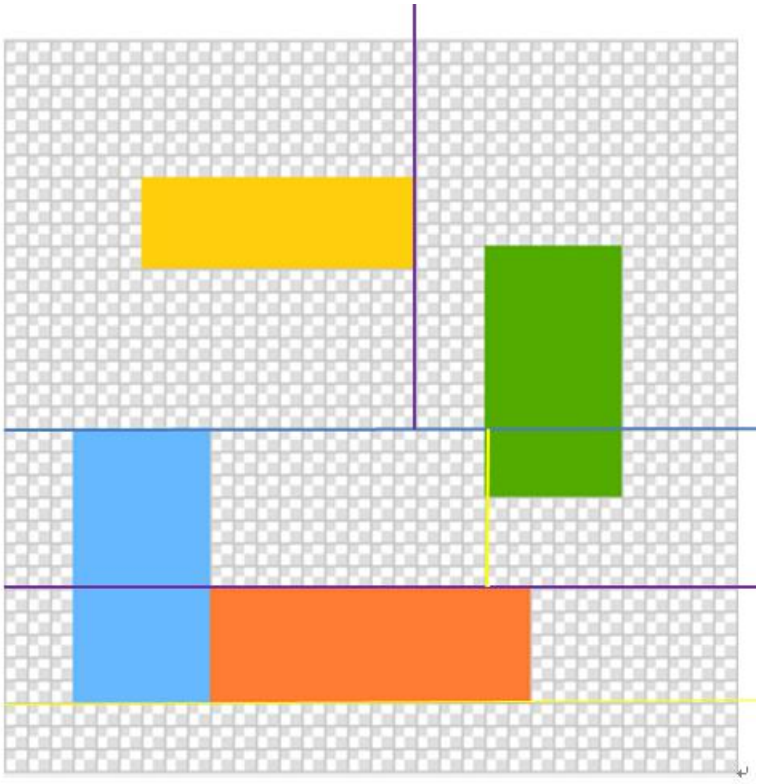

图 2-7 切分效果图

2.2 一些关于空间数据结构的简单研究与实现

2.2.1 前言

首先，我们来了解一下计算机中的“视图”“图形”等这些概念。我们日常的语境当中在说“视图”、“照片”、“图形”等等这些和图像有关的概念时，一般都认为它单纯指代一个具有视觉效果的画面——比如高考卷子上立体几何的那个题，你会把它对应到“图形”的概念上；你相册里面那张让你看了就来气的跟前对象的合照，你会把它对应到“照片”的概念上，等等等等。不过在计算机中，“视图”“图像”这些概念的涵义还包含了其他的方面，对于一个图像，计算机所要关注的不止是它呈现的视觉效果，还要使用合适的数字格式存储图像，并按照用户的要求对图像进行分析、处理和加工。我们可以把计算机中的“视图场景”看作一个“虚拟的视图场景”：它在概念上更多地指代的是对图像的数字格式存储以及处理加工的相关操作，不妨回想一下我们前面叙述“空间数据结构”的时候所说的“保存图元信息”“定位查找元素”这些关键词，就对应的是这样的概念，很好理解，数字格式便是图像在计算机中的存在形式。这也是前面为什么要用“虚拟”一词，在计算机中说到“图像”“视图”等等时，我们不应直接以日常的语境将它理解成一幅纯粹的“视觉画面”，而是要关注到计算机中实际存储并操作的对象——“数字格式的图像”。我们下面的内容当中也都基于上述概念，讨论的是“数字格式的图像”。

假如我们现在要设计一个视图场景(Scene)，在其中包含很多图元对象(Item)。就像这样:

图 2-8 视图场景实例

上面就是一个场景，其中包含了大量的线条与色块等等，这些所有的图形要素都统称为"图元"。而场景就负责管理并绘制图元、接收并向对应的图元转发 UI 事件。我们现在假设场景是静态场景，也就是场景中的图元在初始化以后就不会有任何改变。

现在我们想让这个场景当中的图元响应用户的 UI 操作，比如说，用鼠标点击场景中的某个位置，场景就需要找到有哪根线条覆盖到了那个位置，然后让那个线条按照某种规则动起来。虽然我们在实现这个逻辑的时候是多么地希望能有一个“点信息表”可以根据鼠标点下的位置直接检索出对应的点上有哪些图元覆盖到这样的相关信息，但场景保存的是“各个图元”相关的信息成一个“图元信息表”，而不太可能把“每个点”相关的信息保存成“点信息表”——如果想用“点”来衡量一整个视图场景的大小，那真是让人吐血，想想你那 8K 超清的大屏幕上算出来有多少个点吧，如果想要保存“每个点被哪些图元覆盖到”这样的信息，需要一个什么规模的二维数组(.....)。所以我们知道用户的鼠标落在哪个“点”上之后，要做的是去“图元信息表”寻找“哪些图元覆盖了这个点”，毕竟，场景保存的是“各个图元”的“图元信息表”，不会丧心病狂到去把“每个点”的信息保存成一个“点信息表”。

你可能会说，这多简单，对图元信息表中的每个图元遍历一次，看看哪个图元覆盖了那个位置呗。我们一般管这种做法叫暴力搜索，当然这不是说它完全错误，一个方法只要能达到目的那它就不是绝对的错误；但是必须考虑的事情是，宇宙是有限的，连一个葛立恒数大小的物理概念在我们的宇宙中都找不到，那就不能在讨论一个事情的时候抛开它的时空限制、可用资源限制这些事实层面的东西不谈。比如说，现在场景当中有一千万个图元(不要惊讶，这种情况当然有可能出现，比如说卫星地图或者高品质游戏等一些高精绘图场景，或者你哪天想起来要给你朋友展示一下属于码农的浪漫于是把参数调成一千万)，那用户每点一下就要暴力搜索数秒，如果这景象出现在一个游戏里那它早已被市场淘汰。但是前面我们又说过，“遍历图元”这样的操作逃不掉的，那么有什么能提高效率的方法呢？

计算机图形学对此问题早有研究，并提出了“空间数据结构”(Spatical Data)的概念。很多时候我们需要能够方便地在空间中定位和查找元素的数据结构来处理物体，这称为空间数据结构。空间数据结构将空间划分为多个层次多个区域，并在保存图元信息时使用对应的数据结构记录每个划分出的区域中完全或部分包含的图元并保存，这样就更方便定位和查找空间中的元素，被广泛用在图形学场景中用来加快运算。例如，现在场景中有一千万个图元，原本在遍历的时候需要对这一千万个图元都遍历，但是现在有一种空间数据结构将整个场景不重不漏地划分

成了 1000 个区域，那现在平均每个区域就只有一万个图元了，而用户点到的那个位置一定只落在某个区域当中(谁让它这么没出息只是一个点呢)，所以现在只需要遍历一万个左右的图元了。当然这只是一个比较理想化的模型，实际操作起来还会遇到其他的细节问题，比如说如果有很多图元它就是比你能划出来的区域更大该怎么办呢。而下文所要探讨的，就是均匀网格、BSP 这两种空间数据结构在二维空间下的情形以及实现时的各种细节问题。

2.2.2 均匀网格的概念

均匀网格(Grid)是一种空间数据结构，它使用了一个最为简单朴素的做法，就是将一个空间均匀地划分为大小相等的网格。在二维空间中，均匀网格在一个平面区域内使用等距的直线将其划分为大小相等的网格子区域。把空间划分成均匀网格，使用数组记录每个网格中包含的图元，就形成了一个简单的空间数据结构。下图展示了一个二维均匀网格的示例：

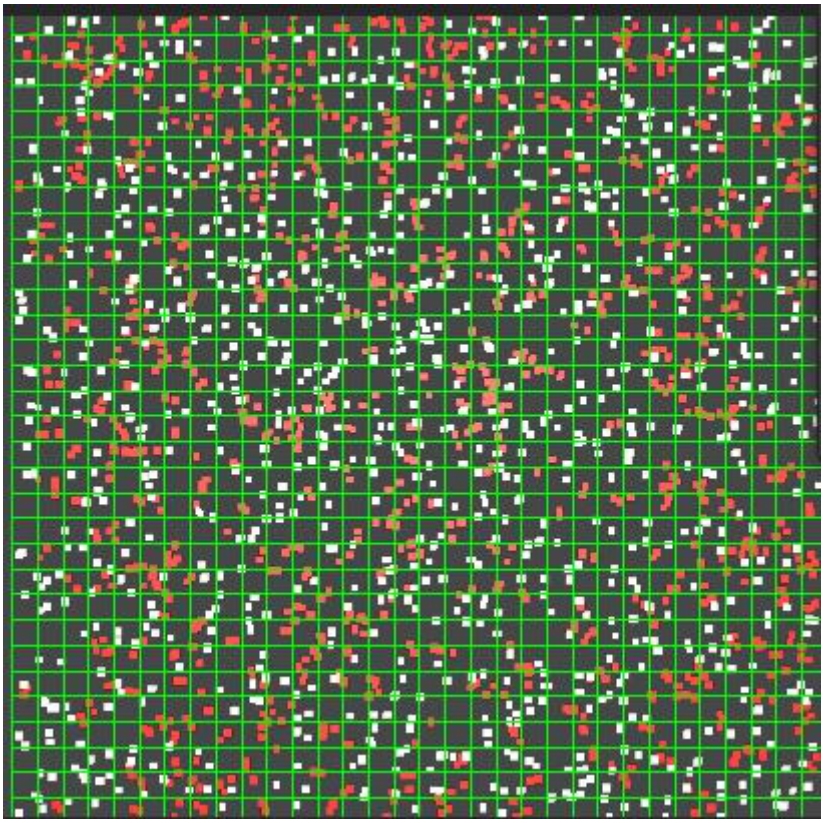

图 2-9 二维均匀网格的实例

2.2.3 BSP 的概念

BSP (Binary Space Partitioning，空间二叉划分)也是一种空间数据结构，它可以对一个二维或三维空间进行划分，本文探讨的是二维空间的情形。每次将空间

划分为两个部分并对每个划分出的子空间递归重复这个过程，然后使用树结构将空间组合起来，最后得到的就是一棵二叉树，这棵二叉树的每个叶节点对应一块区域，每个非叶节点对应一次划分所用的分割线。这个道理其实很容易想到，因为在进行划分的过程中，所有非叶节点都经历了派生出子节点的过程，也就是说它们经历了“从叶节点变成了非叶节点”的过程，这个过程自然会让这个节点的角色发生改变：当一个节点还没有派生子节点时，它在这棵(尚未完成)的树上是一个叶节点，代表了一个区域；而当它派生出子节点后，它自己就变成了非叶节点，那么它现在就代表了一条分割线，其实很好理解，从一个原本的叶节点“派生子节点”对应的操作就是对原本节点代表的区域进行划分，在划分后这个节点就代表了分割线，而它派生的子节点现在就代表分出的两个子区域，当然，子节点也可以继续对自己进行上面这个操作。

一种简单的 BSP 划分方式是“轴向划分”，它每次划分都简单地在每个子空间的对称轴上进行划分，对于矩形场景而言，对称轴就是每个子空间水平或垂直的中线轴，而且这样划分出的子空间同样是矩形，对这些子空间也依然从对称轴上进行划分。通常来说，轴向划分最终形成一棵满二叉树：所有非叶节点的度都是 2，所有叶节点都在同一层次上，也就是说每一个同级的子区域都会进行划分，直到达到所要求的树深度。我们在划分区域时一般都采取“横竖交替”的策略(很好理解，有谁会一直竖着画呢？双缝干涉实验？)，如果一个节点是按照水平中线轴划分，那么它所派生的子节点再划分的时候就是按照垂直中线轴划分。

我们会发现，轴向划分最终得到的空间数据结构与网格无异。实际上，BSP 划分时可以使用任意位置、任意方向的分割线，轴向划分是 BSP 最简单的划分方式，而在很多实际应用当中，都会基于图元的形状大小以及位置等信息而使用更加灵活的分割线，以使得尽可能多的图元都只落在一个子区域当中，能够提升运算的性能。

2.2.4 简单的研究与实现

1、QT 的 BSPTree

我们先来看有名的图形界面框架 QT。QT 中的视图场景 QGraphicsScene 使用的是轴向划分 BSP：

```
1.  void QGraphicsSceneBspTree::initialize(const QRectF &rect, int
    depth, int index)
2.  {
3.      Node *node = &nodes[index];
```

```
4.    if (index == 0) {
5.      node->type = Node::Horizontal;
6.      node->offset = rect.center().y();
7.    }
8.    if (depth) {
9.      Node::Type type;
10.     QRectF rect1, rect2;
11.     qreal offset1, offset2;
12.     if (node->type == Node::Horizontal) {
13.       type = Node::Vertical;
14.       rect1.setRect(ax: rect.left(), ay: rect.top(), aaw: rect.w
    idth(), aah: rect.height() / 2);
15.       rect2.setRect(ax: rect1.left(), ay: rect1.bottom(), aaw: r
    ect1.width(), aah: rect.height() - rect1.height());
16.       offset1 = rect1.center().x();
17.       offset2 = rect2.center().x();
18.     } else {
19.       type = Node::Horizontal;
20.       rect1.setRect(ax: rect.left(), ay: rect.top(), aaw: rect.w
    idth() / 2, aah: rect.height());
21.       rect2.setRect(ax: rect1.right(), ay: rect1.top(), aaw: rec
    t.width() - rect1.width(), aah: rect1.height());
22.       offset1 = rect1.center().y();
23.       offset2 = rect2.center().y();
24.     }
25.     int childIndex = firstChildIndex(index);
26.     Node *child = &nodes[childIndex];
27.     child->offset = offset1;
28.     child->type = type;
29.     child = &nodes[childIndex + 1];
30.     child->offset = offset2;
31.     child->type = type;
32.     initialize(rect: rect1, depth: depth - 1, index: childIndex);
33.     initialize(rect: rect2, depth: depth - 1, index: childIndex +
    1);
34.   } else {
35.     node->type = Node::Leaf;
36.     node->leafIndex = leafCnt++;
37.   }
38. }
```

nodes 数组与 index 参数用于保存二叉树的信息，如何保存二叉树不是本文探讨的重点。看看这个二叉树是如何生成的：

由用户指定初始的 rect 和 depth，rect 参数是矩形对象用于指定 BSP 所要划分的区域，depth 参数用于指定二叉树的深度。二叉树节点 Node 为如下的数据结构:

```
1.  struct Node
2.  {
3.    enum Type { Horizontal, Vertical, Leaf };
4.    union {
5.      qreal offset;
6.      int leafIndex;
7.    };
8.    Type type;
9.  };
```

这里使用了一个非常巧妙的设计：非叶节点对应的是分割线，但是我们并不需要保存“一条线”下来，由于使用轴向划分的 BSP，因此一条分割线只需要它的分割方向(水平/竖直)和它距离场景原点坐标轴的距离就可以确定位置，所以非叶节点保存的是 type(Horizental/Vertical)以及当前分割线与 type 对应坐标轴之间的距离 offset。叶节点的 type 是 Leaf，它对应一个区域，区域的信息同样不保存在 Node 本身当中，而是将每个区域中所包含的图元的信息保存在另一个单独的数组 leaves 中，叶节点的 Node 保存一个 leafIndex 作为 leaves 数组的下标，这样叶节点对应的区域就是 leaves[leafIndex]。由于一个节点只可能是叶节点或非叶节点之一，所以使用 union 联合 offset 和 leafIndex。

生成二叉树的过程递归调用 initialize 函数，递归调用的参数 rect 是本次调用后切出的子区域，depth 每次递归都减 1。在函数中判断 depth，当 depth 为 0 时表示已经到达目标深度，则当前的节点为叶节点，type 设为 Leaf 且记录 leafIndex；否则当前的节点为非叶节点，并且根据当前节点的 type 和 rect 计算出下层节点(child)在递归调用时的参数：下层节点的 type 应与当前节点的相反，下层节点的 rect 是当前分割线所分出的两个子区域。

2、BSP

这里的 BSP 就是在吸收 QT 的经验以后自己实现的 BSP 结构了。相关思路同上，不再赘述。

BSP 的数据结构为 Node 所串联成的二叉树。Node 的结构如下：

```
1.  /**
2.   * @enum SplitType
```

```
3.   * @brief 枚举区域分割的类型。
4.   */
5.  enum SplitType
6.  {
7.    Horizontal = 0, ///< 水平分割
8.    Vertical,    ///< 竖直分割
9.    Leaf      ///< 叶子节点
10. };
11. /**
12.  * @class Node
13.  * @brief 树的节点，对应分割形成的某块区域。
14.  */
15. struct Node
16. {
17.
18.   /**
19.    * @brief 默认构造。
20.    */
21.   Node() = default;
22.   /**
23.    * @brief 析构函数，处理本节点以及子节点的释放。
24.    */
25.   ~Node();
26.   /**
27.    * @brief 该节点的分割类型。
28.    */
29.   SplitType m_splitType;
30.   /**
31.    * @details offset 和 leafIndex 分别对应非叶子节点和叶子节点的数据信息，对于同一个节点这两条数据不可能共存。因此为了节省内存，采用 union
32.    */
33.   union
34.   {
35.     /**
36.      * @brief 分割线的横坐标或纵坐标的偏移量，是横坐标还是纵坐标取决于分割类型。
37.      */
38.     int m_offset;
39.     /**
40.      * @brief 节点的下标，在外部的 leaves 数组中使用。
41.      */
```

```
    int m_leafIndex;
  };
  /**
   * @brief 左子节点指针。
   */
  struct Node *m_left = nullptr;
  /**
   * @brief 右子节点指针。
   */
  struct Node *m_right = nullptr;
};
Node::~Node()
{
  if (!m_left && !m_right) return;

  delete m_left;
  m_left = nullptr;

  delete m_right;
  m_right = nullptr;
}
```

BSP 包含的数据成员如下：

```
/**
 * @brief 整棵 BSP 树的根节点。
 * @todo 后续考虑自己实现简单的对象树机制，不使用智能指针
 */
Node *m_root = nullptr;

/**
 * @brief 存储每个叶子节点中的 PicItem（图元）列表。
 * @details 树构建成功以后，所有的 PicItem 都存储在叶子节点的区域中，为了方便获取，将数据提取到整棵树的数据结构中，叶子节点中存储下标方便访问。注意，每个 "Vector<PicItem *>"对应一个节点，因此 m_leaves 实际上以一维的方式组织各个节点。
 */
Vector<Vector<PicItem *>> m_leaves;

/**
 * @brief 树的深度，对应分割的次数。
 */
```

```
16. int m_depth = 0;
17.
18. /**
19.  * @brief 整棵树作用的 2D 平面范围。
20.  */
21. Rect m_region;
```

由用户传入矩形区域 region 与树的深度 depth，程序传入初始根节点并递归调用 init 函数创建各个子节点。

注：本代码中 Rect 对象的 x1()和 y1()返回矩形对象左上角的坐标，x2()和 y2()返回矩形对象右下角的坐标，下同。

```
1.  void init(Node *node, const Rect &region, int depth)
2.  {
3.    // depth > 0，继续向下分割
4.    if (depth > 0)
5.    {
6.      // 为了统一命名，使用 left/right 对应逻辑上的 左/右 子节点
7.      // 水平 Horizontal：left 为上半边，right 为下半边
8.      // 垂直 Vertical：left 为左半边，right 为右半边
9.      int offsetLeft = 0, offsetRight = 0;
10.     Rect leftRect, rightRect;
11.     SplitType newSplit;
12.
13.     if (SplitType::Horizontal == node->m_splitType)
14.     {
15.       // 当前节点为水平分割 Horizontal，则子节点为 Vertical，left 为上半边，right 为下半边
16.       newSplit = SplitType::Vertical;
17.       leftRect = Rect(region.x1(), region.y1(), region.width(), region.height() / 2);
18.       rightRect = Rect(leftRect.x1(), leftRect.y2(), region.width(), region.height() / 2);
19.       offsetLeft = leftRect.x1() + leftRect.width() / 2;
20.       offsetRight = rightRect.x1() + rightRect.width() / 2;
21.     }
22.     else
23.     {
24.       // 当前节点为垂直分割 Vertical，则子节点为 Horizontal，left 为左半边，right 为右半边
25.       newSplit = SplitType::Horizontal;
```

```
26.       leftRect = Rect(region.x1(), region.y1(), region.width() /
    2, region.height());
27.       rightRect = Rect(leftRect.x2(), leftRect.y1(), region.wid
    th() / 2, region.height());
28.       offsetLeft = leftRect.y1() + leftRect.height() / 2;
29.       offsetRight = rightRect.y1() + rightRect.height() / 2;
30.     }
31.
32.     node->m_left = new Node;
33.     node->m_left->m_splitType = newSplit;
34.     node->m_left->m_offset = offsetLeft;
35.
36.     node->m_right = new Node;
37.     node->m_right->m_splitType = newSplit;
38.     node->m_right->m_offset = offsetRight;
39.
40.     init(node->m_left, leftRect, depth - 1);
41.     init(node->m_right, rightRect, depth - 1);
42.   }
43.   // 遇到叶子节点
44.   else
45.   {
46.     node->m_splitType = SplitType::Leaf;
47.     node->m_leafIndex = m_leaves.size();
48.     m_leaves.append(Vector<PicItem *>());
49.   }
50. }
```

查询是空间数据结构应当提供的基本用途，根据用户传入的区域查询当前空间数据结构中该传入区域命中的子区域(在 BSP 中，是叶节点)；在查询的基础上，用户或程序可以对命中的区域或对应区域中的图元执行操作，这个操作可以是在区域中增删图元、对区域中指定特征的图元进行进一步查询，等等等等。

```
1.  /**
2.   * @brief 定义回调函数类型，用于对叶节点执行操作
3.   */
4.  using Visitor = std::function<void(LList<LCanvasItem *> &)>;
5.
6.  // 以 addItem 为例
7.  void addItem(LCanvasItem *item)
8.  {
```

```
    auto func = [&item](LList<LCanvasItem *> &items)
    {
      items.append(item);
    };

    update(func, m_root, item->boundingRect());
}

/**
 * @brief 根据所给的区域查询命中的叶子节点，并执行指定的操作。
 * @param visitor 函数对象。用于对查找到的叶子节点执行操作
 * @param node 根节点
 * @param rect 目标区域矩形
 */
void update(const Visitor &visitor, Node *node, const Rect &rect)
{
    if (m_leaves.isEmpty()) return;

    switch (node->m_splitType)
    {
      case SplitType::Leaf:
        visitor(m_leaves[node->m_leafIndex]);
        break;

      case SplitType::Vertical:
      {
        if (rect.x1() < node->m_offset)
        {
          update(visitor, node->m_left, rect);

          if (rect.x2() >= node->m_offset) update(visitor, node->m_right, rect);
        }
        else
        {
          update(visitor, node->m_right, rect);
        }

        break;
      }

```

```
    case SplitType::Horizontal:
    {
      if (rect.y1() < node->m_offset)
      {
        update(visitor, node->m_left, rect);

        if (rect.y2() >= node->m_offset) update(visitor, node->m_right, rect);
      }
      else
      {
        update(visitor, node->m_right, rect);
      }

      break;
    }
  }
}
```

3、Grid

均匀网格的数据成员如下：

```
/**
 * @brief 存储每个网格中的 PicItem 列表。
 * @details 树构建成功以后，所有的 PicItem 都存储在网格的区域中，为了方便获取，将数据提取到整个网格的数据结构中，通过数学计算得出下标方便访问。注意，每个"Vector<PicItem *>"对应一个网格，因此 m_grids 实际上以"一维数组排列二维网格"的方式组织各个网格。
 */
Vector<Vector<PicItem *>> m_grids;

/**
 * @brief 网格分割出的每边的区间个数。
 */
int m_sections = 0;

/**
 * @brief 整个网格作用的 2D 平面范围。
 */
Rect m_region;
```

与 BSP 不同的是，BSP 有代表节点 Node 的“单元结构”，而均匀网格当中

没有设置“代表网格单元结构”Grid。这是基于两种空间数据结构的基本用途“查询”逻辑上的不同：BSP 树本身的结构和内容就决定了在查询时必须与非叶节点的分割线比较才能找到最终位于叶节点的子区域，树结构的特性就是不提供直接访问任何特定叶节点的快速途径，访问树中的指定节点必须从根节点开始，无法避开各个 Node 之间的逻辑结构(即使用数组保存它也是如此)，也因此在 BSP 树中需要保存每条分割线的信息；而 Grid 本身的结构与数组对应，并且均匀划分使其在查询时能够通过数学计算的方式直接获知子区域的数组下标，因此就不需要专门去保存每个网格本身的信息，只需要将每个网格对应的图元列表保存为“图元列表的数组”作为查询时访问的目标。

均匀网格的成员函数通过用户传入的目标区域与分割线数量进行初始化：

```
1.  /**
2.   * @brief 带参构造。
3.   * @param region 需要作用的区域
4.   * @param splitNum 网格的分割线数量(经纬两个方向分割线数量相同)
5.   */
6.  Grid::Grid(const Rect &region, int splitNum) : m_region(region),
    m_sections(splitNum + 1), m_grids(Vector<Vector<PicItem *>>((spl
    itNum + 1) * (splitNum + 1))) {}
```

均匀网格的 m_grids 在初始化时规定好大小也就是网格数量，不过此时并不存储任何内容，因为初始化时还没有图元。与 BSP 不同的是，BSP 中 init 函数进行了划分区域、创建各个节点等操作，而均匀网格没有这一步骤。这也是因为 Grid 不需要保存每个网格的信息。

根据用户传入的区域查询当前空间数据结构中该传入区域命中的子区域(在均匀网格中，是数组下标)；在查询的基础上，用户或程序可以对命中的区域或对应区域中的图元执行操作，这个操作可以是在区域中增删图元、对区域中指定特征的图元进行进一步查询，等等等等。

```
1.  void update(const Visitor &visitor, const Rect &rect)
2.  {
3.      int x1Index = rect.x1() < m_region.x1() ? 0 : rect.x1() * m_secti
    ons / m_region.width();
4.      int y1Index = rect.y1() < m_region.y1() ? 0 : rect.y1() * m_secti
    ons / m_region.height();
5.      int x2Index = rect.x2() >= m_region.x2() ? (m_sections - 1) : rec
    t.x2() * m_sections / m_region.width();
6.      int y2Index = rect.y2() >= m_region.y2() ? (m_sections - 1) : rec
    t.y2() * m_sections / m_region.height();
```

```
7.
8.    for (int y = y1Index; y <= y2Index; ++y)
9.    {
10.     for (int x = x1Index; x <= x2Index; ++x)
11.     {
12.       visitor(m_grids[y * m_sections + x]);
13.     }
14.   }
15. }
```

2.3 在 X11 下使用 cairo 引擎绘制图形

2.4 使用 Woboq CodeBrowser 搭建源代码网站

3 知识技能学习情况

3.1 开发环境和工具

请详细写出实习任务涉及的开发环境和工具介绍等。

3.2 预备知识

请详细写出完成实习任务需要的预备知识等。

3.3 新知识点学习和掌握情况

请详细写出完成实习任务需要学习的新知识及要求。

4 工程协作交流情况

（阐述在实习执行过程中，针对特定的目标或问题，与工程项目组成员，包括与其他学科的成员合作并开展工作的情况

5 工程计划管控与执行情况

（阐述项目实施过程中遇到的突发或异常情况时如何采取措施，保证项目按时实施。）

6 职业素养与工程伦理的学习与培养

（描述对软件工程系统的质量、环境、职业健康、安全和服务意识的学习和认识，对职业道德和规范的理解和遵守。）

7 对软件工程实践以及软件工程领域发展的认识

（描述软件工程实践对环境和社会可持续发展的影响。）

8 结束语

8.1 课题完成情况、有待进一步解决的问题及方向

8.2 本人对于企业实习的收获及体会

参考文献

[1] 王浩刚，聂在平.三维矢量散射积分方程中奇异性分析[J]. 电子学报，1999, 27(12): 68-71

[2] X. F. Liu, B. Z. Wang, W. Shao. A marching-on-in-order scheme for exact attenuation constant extraction of lossy transmission lines[C]. China-Japan Joint Microwave Conference Proceedings, Chengdu, 2006, 527-529

[3] 竺可桢.物理学[M]. 北京：科学出版社，1973, 56-60

致谢

总结报告最基本要求是 15000 字以上。