《面向对象程序设计》

课程设计报告

小组序号：16

组　　长：徐子健

组　　员：林显龙

组　　员：于新民

组　　员：钟新伟

指导教师：姚尧

中国地质大学(武汉)地理与信息工程学院

2024 年 8 月 18 日

目录

第1章　需求规格说明

1.1 系统描述

本次面向对象课程设计开发程序，要完成一个GIS数据管理与分析系统，包括数据管理与存储、矢量数据可视化与分析、栅格数据可视化与分析，矢量栅格三个模块。系统要设计ui界面，分析操作要有独立窗口。

该 GIS 程序，被我们命名为 MYGIS。

系统使用 C++语言编写，开发软件：Visual Studio2022，qt5.14.2，使用 GDAL 等第三方库完成实现 GIS 功能。

1.2 系统功能需求

系统应该具有以下功能：

1. 矢量和栅格数据的抽象，数据要能在系统内存中进行管理，不直接依赖文件路径。
2. 矢量数据要求能够能够在 ui 上使用 qt 绘图工具或相关第三方库绘图实现矢量数据的可视化。
3. 矢量数据能够进行空间分析，和统计分析等操作。
4. 栅格数据同样要求在 ui 上使用 qt 绘图工具或相关第三方库绘图实现栅格数据的可视化。
5. 栅格数据能够进行空间分析，统计分析等操作。
6. 一些附加功能：例如工程文件的保存和打开，超大栅格的读取等。

第2章 系统分析与设计

2.1 设计思想

2.1.1 存储结构

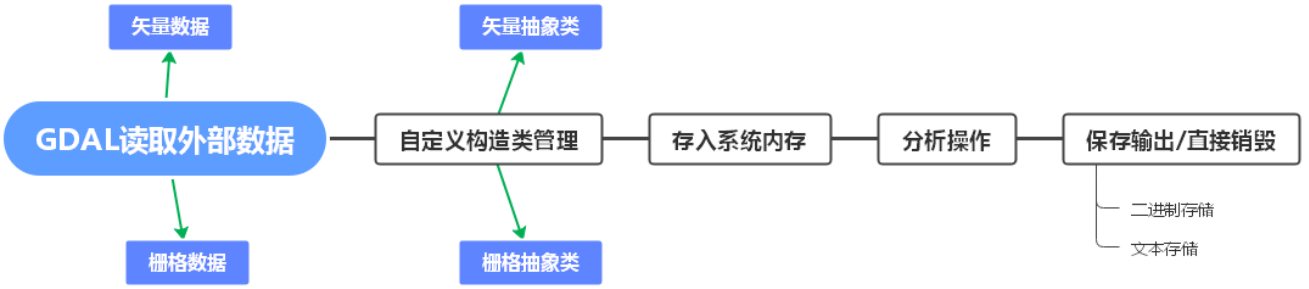

图 2.1-1 存储结构设计图

如图 2.1-2所示，系统存储结构主要在于内部内存的管理以及输出，这样做是为了保证源文件的的健康和安全，在后续分析调试时，只需要对系统内的数据直接进行操作就可以，不必依赖源文件的路径接口，同时在代码编写和操作调试中也不会直接影响到源文件的结构。

本系统并未实现数据库存储相关功能。程序内分析和修改后的图层数据保存主要支持矢量的文本存储和二进制存储！

2.1.2 系统设计思想

图 2.1-3 系统设计思想

如图图 2.1-4所示，本系统，主要有一个主窗口MYGIS，9个副窗口，系统提供文件的导入，文件数据在系统内存内进行操作，不会影响外部文件，本gis系统提供三个矢量分析功能：计算几何，叠加分析，包络矩形，三个栅格分析操作：假色彩组合显示栅格影像，栅格波段直方图绘制[1]，均衡化显示栅格数据。在系统内部进行分析操作后可以将分析后的矢量图层进行要素转shapefile操作进行二进制存储，或者使用文本存储，将文件保存为标准的WKT格式的CSV文件。

2.1.3 系统特色

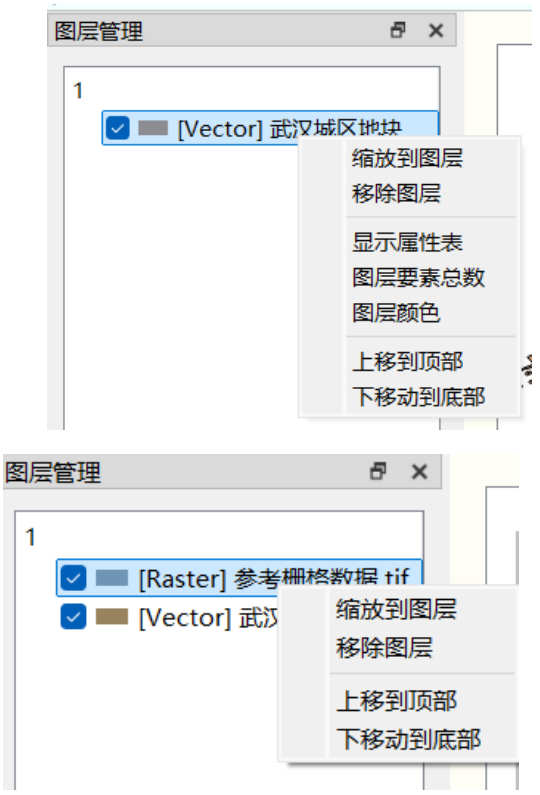

图 2.1-5 图层管理

图 2.1-6 底部实时显示鼠标经纬度和缩放

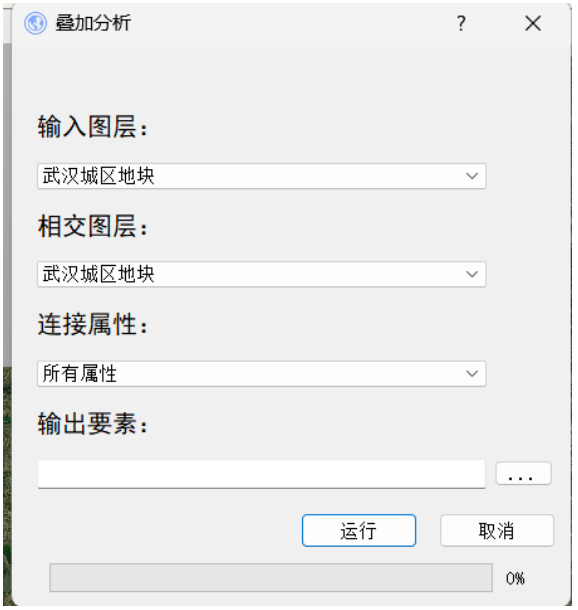

图 2.1-7 独立的分析窗口以及进度条

如以上几图所示，本系统设计主要表现在以下具体几点：

1.系统提供矢量和栅格图层管理，导入视图的图层可视化的展示在左侧图层管理栏，提供缩放，移除，图层顺序移动等基本操作。

2.本系统支持矢量和栅格同时管理，两种数据不进行单独的管理，同时存在于同一视图，彼此都能进行分析操作，互不影响。

3.本系统初始化了视图，视图内鼠标所在的经纬度和视图的缩放比例能实时在底部状态栏更新。

4.本系统对各种分析操作都提供单独窗口接入，每个窗口独立窗口，可以打开多个窗口不影响主程序。每个分析操作提供一个简易版本的进度条（该进度条为假进度条），进度条并未支持多线程，进度条是根据相关分析操作在主程序里被更新。

2.2 设计表示

2.2.1 矢量存储结构VectorLayerData

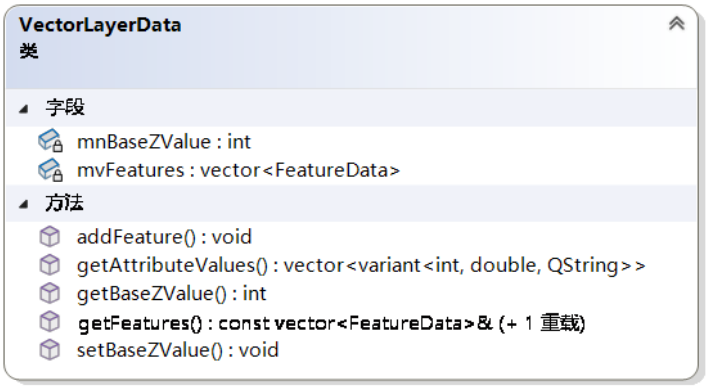

图 2.2-1VectorLayerData 类图

名称	类型	修饰符	摘要	隐藏
方法				
addFeature	void	public	添加一个要素到图层	
getAttributeValues	vector<variant<int, double, QString>	public	获取某一属性的所有值	
getBaseZValue	int	public	获取图层的基础 Z 值	
getFeatures	const vector<FeatureData>&	public	获取图层中的所有要素（只读）	
getFeatures	vector<FeatureData>&	public	获取图层中的所有要素（可修改）	
setBaseZValue	void	public	设置图层的基础 Z 值	

图 2.2-2VectorLayerData 类成员函数说明

图 2.2-3getAttributeValues 函数流程图

getAttributeValues函数伪代码：

输入： attributeName (QString)

1. 创建一个容器，用于存储属性值。
2. 遍历 mvFeatures 中的每个要素，获取要素的属性数据，查找属性表名对应的属性值。
3. 如果该属性值有效，将属性值添加到开始创建的容器中。

输出： 创建的容器，其包含所有找到的属性值

详细矢量存储结构在2.3部分说明。

2.2.2 栅格存储结构GDALRasterRead和RatserData

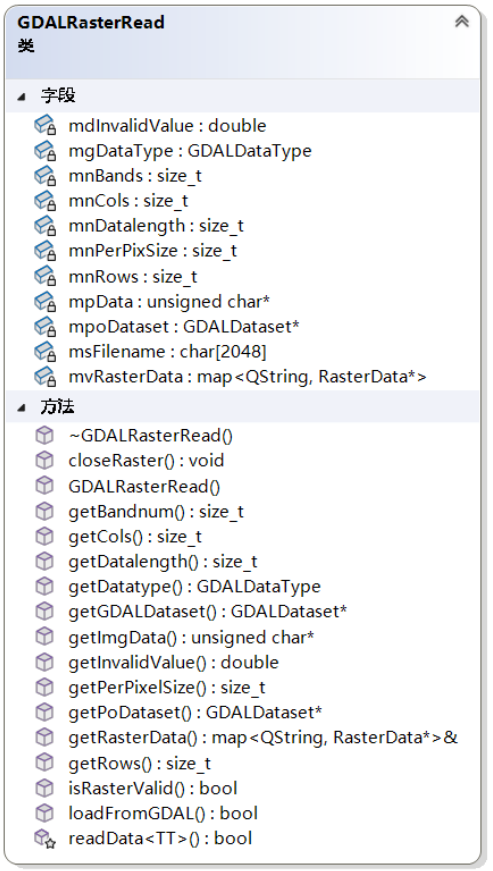

图 2.2-4GDALRasterRead 类图

名称	类型	修饰符	摘要	隐藏
方法				
~GDALRasterRead		public	析构函数	
closeRaster	void	public	关闭栅格数据集	
GDALRasterRead		public	构造函数	
getBandnum	size_t	public	获取栅格数据的波段数量	
getCols	size_t	public	获取栅格数据的列数	
getDatalength	size_t	public	获取栅格数据总字节数	
getDatatype	GDALDataType	public	获取栅格数据类型	
getGDALDataset	GDALDataset*	public	获取 GDAL 数据集指针	
getImgData	unsigned char*	public	获取栅格图像数据指针	
getInvalidValue	double	public	获取无效值	
getPerPixelSize	size_t	public	获取每像素字节数	
getPoDataset	GDALDataset*	public	获取 GDAL 数据集指针	
getRasterData	map<QString, RasterData*>&	public	获取栅格数据	
getRows	size_t	public	获取栅格数据的行数	
isRasterValid	bool	public	检查数据集是否有效	
loadFromGDAL	bool	public	从文件加载栅格数据	
readData<TT>	bool	protected	模板函数，读取不同类型的栅格数据	

图 2.2-5GDALRasterRead 类成员函数说明

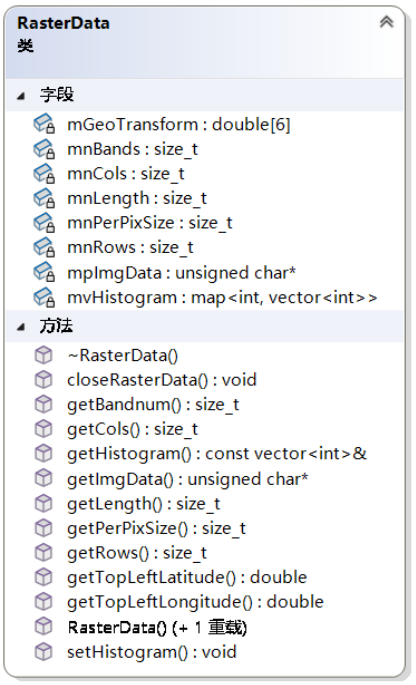

图 2.2-6RatserData 类图

~RasterData		public	析构函数：释放栅格数据内存
closeRasterData	void	public	释放栅格数据的内存
getBandnum	size_t	public	获取栅格数据的波段数
getCols	size_t	public	获取栅格数据的列数
getHistogram	const vector<int>&	public	获取直方图数据
getImgData	unsigned char*	public	获取栅格图像数据指针
getLength	size_t	public	获取栅格数据的总字节数
getPerPixSize	size_t	public	获取每像素的字节数
getRows	size_t	public	获取栅格数据的行数
getTopLeftLatitude	double	public	获取左上角纬度
getTopLeftLongitude	double	public	获取左上角经度
RasterData		public	构造函数：初始化栅格数据和相关参
RasterData		public	拷贝构造函数
setHistogram	void	public	设置和获取直方图数据

图 2.2-7RatserData 类成员函数说明

如上GDALRasterRead类图和RatserData类图所示，RasterData类作为一个接受栅格数据传递的类，没有特殊的函数方法或者其他结构，只接受传递过来的长宽高，波段等数据。

下面对GDALRasterRead中主要的两个loadFromGDAL和readData成员函数进行说明。

图 2.2-8loadFromGDAL 函数流程图

图 2.2-9readData 函数流程图

loadFromGDAL 函数伪代码：

输入:栅格文件的路径，图层名称。

1.关闭任何已打开的栅格数据集。
2.检查 GDAL 驱动是否已注册，如果未注册，则注册所有驱动。
3.尝试打开指定的栅格文件如果打开失败，则输出错误并返回失败。
4.获取栅格数据的基本信息：行数、列数、波段数、数据类型等。根据数据类型选择合适的读取方法。调用相应的数据读取函数。如果读取失败，则输出错误并返回失败。为图像数据分配内存并复制读取的数据。
5.获取地理变换参数。存储栅格数据，包括图像和直方图数据，关联到图层名称。返回成功，栅格数据成加载成功。

输出:返回成功（true），栅格数据加载成功。返回失败（false），加载失败，错误信息输出。

readData 函数伪代码：

输入:无输入，该函数为一个内部函数。

1.检查是否存在已打开的栅格数据集，如果不存在，返回失败。
2.计算栅格数据的总大小，并为其分配内存。
.3 读取栅格数据到内存中，使用 GDAL 的 RasterIO 方法读取数据。检查读取是否成功，如果读取失败，输出错误信息并返回失败。返回成功则表示数据读取完成。

输出:返回成功（true），数据读取成功。返回失败（false），读取失败，错误信息输出。

2.2.3 矢量和栅格绘图：GeometryItem和RasterItem

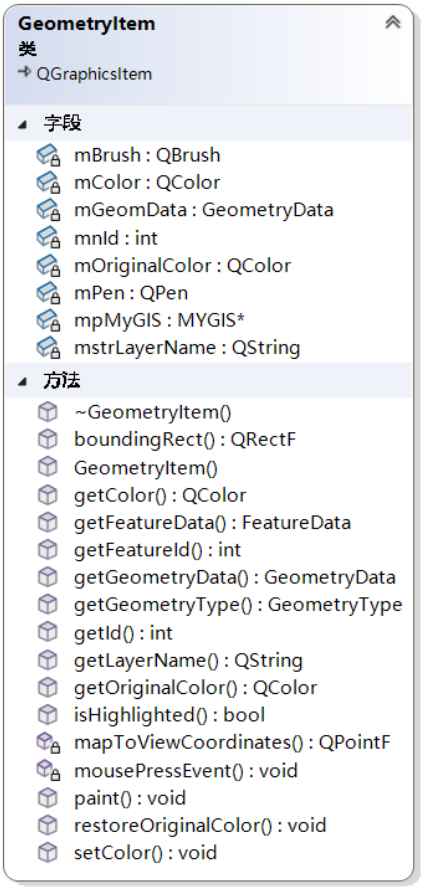

图 2.2-10GeometryItem 类图

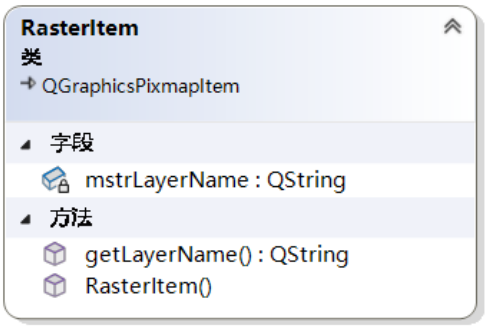

图 2.2-11RasterItem 类图

名称	类型	修饰符	摘要	隐藏
getGeometryData	GeometryData	public	获取几何数据	
getGeometryType	GeometryType	public	获取几何类型	
getId	int	public	获取几何要素的 ID	
getLayerName	QString	public	获取图层名称	
getOriginalColor	QColor	public	获取原始颜色	
isHighlighted	bool	public	判断当前要素是否被高亮（黄色表示	
mapToViewCoordinates	QPointF	private	将几何坐标映射到视图坐标系，翻转	
mousePressEvent	void	private	处理选中状态的逻辑	
paint	void	public	绘制函数，使用 QPainter 绘制几何	
restoreOriginalColor	void	public	恢复到原始颜色	
setColor	void	public	设置颜色，并更新绘制	

图 2.2-12GeometryItem 函数说明

名称	类型	修饰符	摘要	隐藏
方法				
getLayerName	QString	public	获取图层名称	
RasterItem		public	构造函数，初始化栅格项并设置图层	
字段				
mstrLayerName	QString	private	存储栅格图层的名称	

图 2.2-13RasterItem 函数说明

GeometryItem为矢量绘图的显示的item重载了QGraphicsItem，RasterItem为栅格绘图的显示的item重载了QGraphicsPixmapItem。矢量重载赋予了item。图层名，id，z值，矢量几何类型，图层颜色。栅格只赋予图层名。下面主要解释GeometryItem的boundingRect()和paint()两个主要的函数。

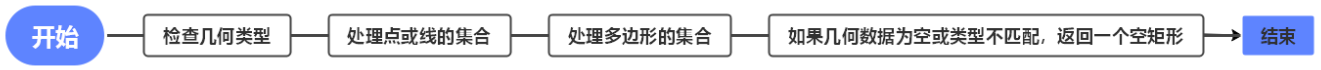

图 2.2-14boundingRect()流程图

图 2.2-15paint()流程图

boundingRect()函数的伪代码：

输入:无输入，该函数为一个内部函数，用于确定绘制区域

1.检查几何数据类型并初始化最小和最大 X、Y 坐标值。遍历所有点，更新最小和最大坐标值，将最小和最大坐标值转换为视图坐标。返回表示边界的矩形。
2.初始化一个空的边界矩形。遍历每个多边形，遍历多边形的每个点，更新最小和最大坐标值。
3.将这些点转换为视图坐标，形成边界矩形。将这个矩形与现有的边界矩形合并。返回总的边界矩形。
4. 如果几何数据为空或类型不匹配，返回空矩形。

输出：边界矩形范围。

paint()函数的伪代码：

输入:用于绘制图形的 QPainter 对象。

1.检查几何数据是否为空，如果几何数据为空，输出警告信息并停止绘制。
2.获取当前绘图环境的缩放因子，根据几何类型选择绘制方法。
3.根据对应的图像绘制相应的路径。
4.设置画笔和笔刷定义线条颜色和填充颜色。
5.使用 painter 绘制路径，在画布上显示几何形状。

输出：在画布上绘制几何形状（点、线、或多边形）。

2.2.4 主程序MYGIS

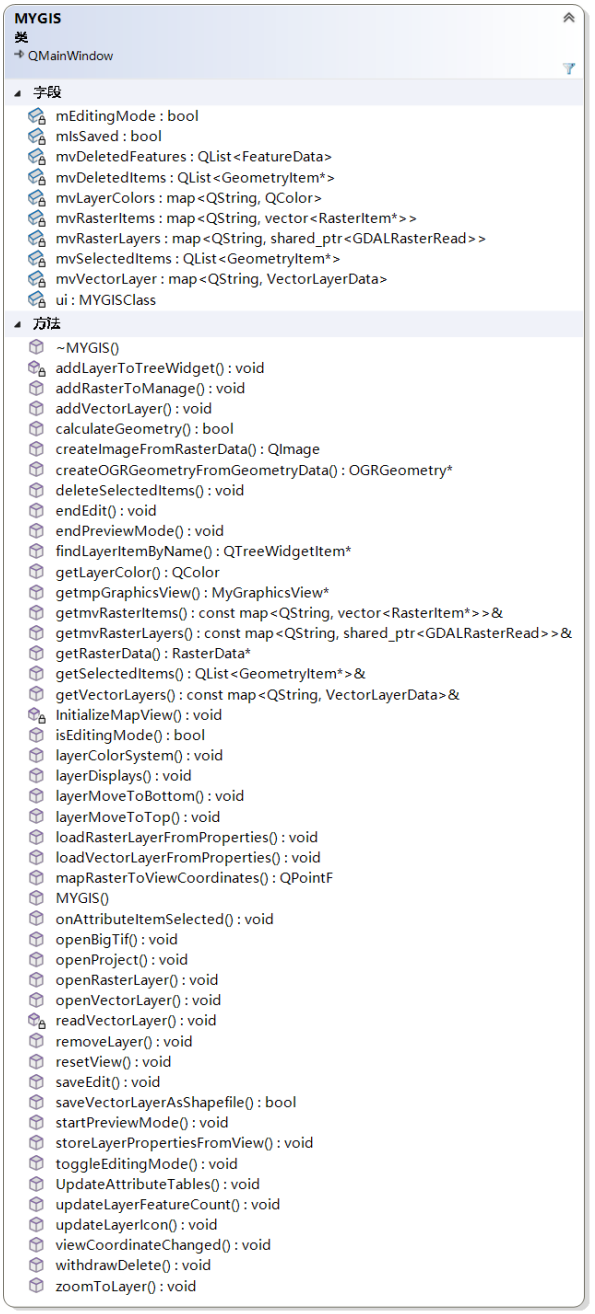

图 2.2-16MYGIS 类图

主程序MYGIS有关显示9个窗口的，槽函数和私有变量在类图中进行了影藏，窗口的初始化和show不必多加描述。

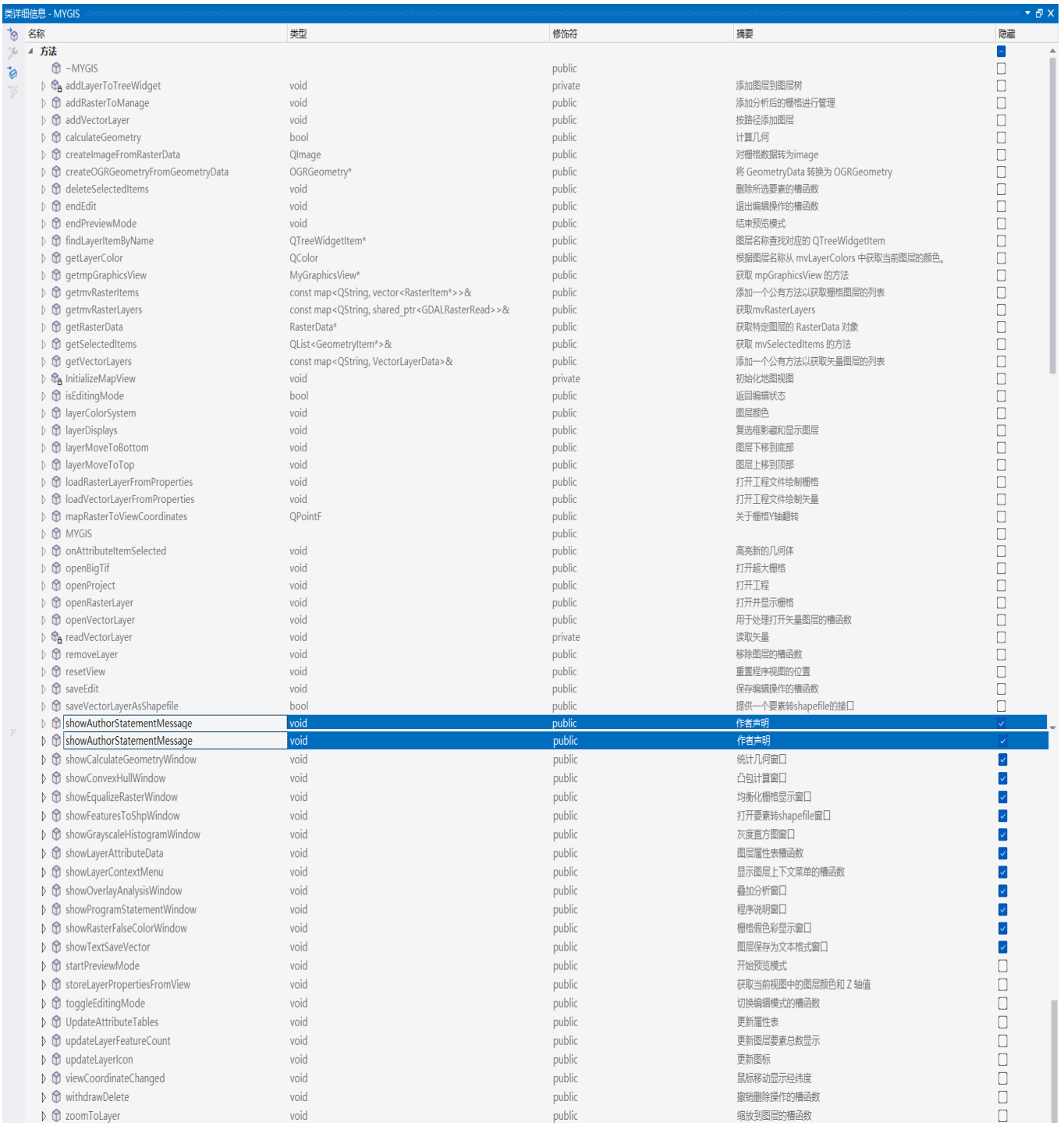

图 2.2-17MYGIS 成员函数说明

在MYGIS有两个三个重要的私有变量：mvVectorLayer（导入的矢量数据），mvRasterLayers（导入的栅格数据），mvRasterItems（绘制栅格的item数据）。其余的mv容器有三个是用于删除，撤回要素操作，并不重要。

mvRasterItems主要是为了和矢量的重载至QGraphicsItem的GeometryItem进行区分。

同时在MYGIS中还有几个主要且重要的成员函数或槽函数，我将一一介绍介绍他们。分别是：InitializeMapView()画布初始化，readVectorLayer()读入并存储矢量数据，openVectorLayer()打开并绘制矢量数据，openRasterLayer()栅格数据的导入,createImageFromRasterData()栅格数据的绘制，openBigTif()超大栅格。AddLayerToTreeWidget()导入图层加入图层管理。

图 2.2-18InitializeMapView()流程图

图 2.2-19readVectorLayer()流程图

图 2.2-20openVectorLayer()流程图

图 2.2-21openRasterLayer()流程图

图 2.2-22openBigTif()流程图

图 2.2-23AddLayerToTreeWidget()流程图

InitializeMapView()伪代码：

输入：无输入，为程序初始化画布所用。

1.定义地图的地理范围，设定为覆盖全球的经度和纬度范围，创建新的场景对象，并设置场景的边界为地图的地理范围。
2.将场景关联到视图，并调整视图以适应场景的比例使视图显示整个地理范围。
3.配置视图交互属性，禁用拖拽，设置缩放锚点为鼠标位置，启用鼠标追踪。

输出：视图现在能够正确显示地理范围，并响应用户交互。

readVectorLayer()伪代码：

输入:无输入，用于读入并将矢量存储到系统内存。

1.用户选择矢量数据文件。
2.遍历文件中的图层，遍历图层中的要素，读取每个要素的几何数据和属性数据。
3.基于文件类型或图层名称生成图层的显示名称，将图层数据存储到内部的数据结构中 mvVectorLayer，并记录文件路径和图层名称。

输出:存储并管理从矢量读取的矢量图层数据。

openVectorLayer()伪代码：

输入:无输入，槽函数用于绘制和显示矢量图层。

1.调用 readVectorLayer 函数读取矢量图层数据。
2.清空当前图层树中的所有矢量图层项。
3.遍历读取的矢量图层，为图层生成随机颜色。为图层中的每个要素创建几何图形项，并将其添加到场景中。
4.在图层树中添加新项显示新的矢量图层，更新视图以显示最新添加的图层。

输出:在视图和图层树中展示读取的矢量图层。

openRasterLayer()伪代码：

输入:无输入，槽函数用于读取并存储山栅格数据。

1.用户选择栅格文件。
2.加载所选栅格文件，存储栅格图层到抽象好的栅格存储结构 mvRasterItems，并在图层树中显示该图层。
3.检查栅格波段数量，如果波段数足够，显示图像。
4.创建图像并添加到场景中，调整视图以适应图像范围。

输出:加载并显示了栅格图层。

createImageFromRasterData()伪代码：

输入:rasterData（包含栅格数据的对象）。

1.获取栅格的行数、列数和波段数，创建图像对象，根据波段数选择灰度或 RGB 格式。
2.填充图像数据，如果是单波段，将数据填充为灰度图像；如果是多波段，假设为 RGB 格式，将数据填充为彩色图像。
3.返回创建的图像。

输出:返回一个由栅格数据创建的 QImage 对象。

openBigTif()伪代码：

输入:无输入，通过dilog选择文件。

1.用户选择一个超大栅格文件。
2.检查栅格波段数和地理变换信息，如果波段不足或地理变换获取失败，关闭文件并返回。
3.初始化处理参数，设置分块的块大小和相关变量。
4.分块读取栅格数据，按块读取栅格数据，将其转换为图像。创建图形项，将其添加到场景中管理。
5.更新进度条，在每个块处理完成后更新进度；如果用户取消，移除已绘制的图形项并退出。
6.将图层添加到管理结构和图层树中，关闭栅格数据集。

输出:成功加载并显示选择的超大栅格图像，或在失败或取消时清理内存。

AddLayerToTreeWidget()伪代码：

输入:layerName（图层的名称）。isRaster，布尔类型变量，确定图层是否为栅格图层。

1.生成完整的图层名称，基于图层类型（栅格或矢量）。检查图层是否已存在于树中：
2.遍历图层树，如果找到相同名称的图层，则退出。
3.创建新图层项，设置名称和默认复选框状态。
4.设置图层颜色和图标，根据图层类型设置颜色和图标。
5.将图层项添加到树的顶级节点，定义复选框，用于控制图层的显示和隐藏。

输出:在图层树中添加新的图层项，并默认设置为显示状态。

2.2.5 矢量分析：统计几何，包络矩形转面，叠加分析

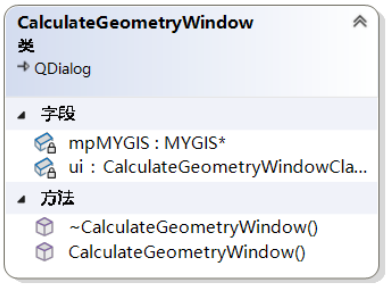

图 2.2-24CalculateGeometryWindow 类图

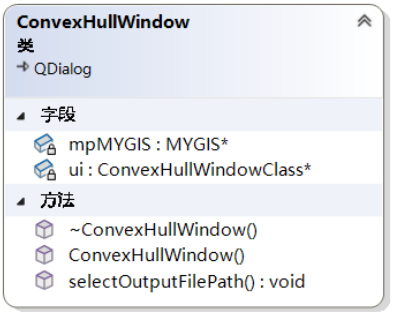

图 2.2-25ConvexHullWindow 类图

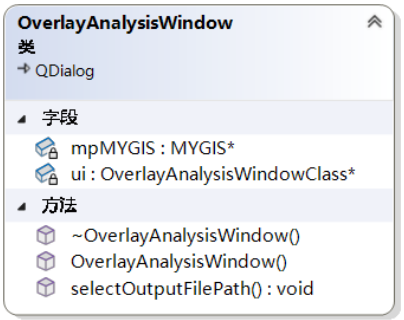

图 2.2-26OverlayAnalysisWindow

如图为三个矢量分析的窗口，如上三图可见，每个窗口都没有很复杂的结构，主要的分析功能都集成在运行按钮的槽函数，故直接介绍分析功能的伪代码。

计算几何函数 startcalculateGeometry() 函数伪代码：

输入:无输入，使用 UI 界面的用户选择的图层作为输入。（查找mvVectorLayer，获得图层名）

1.获取从 UI 界面获取选定的图层名称、线字段名称和面积字段名称。

2.调用主程序的几何计算方法：调用几何计算方法，并传递图层名称和字段名。
主程序几何计算方法伪代码：
2.1 输入:layerName，用户选择的图层名称。lineField，用于存储计算的线长度的字段名。areaField，用于存储计算的面积的字段名。
2.2 查找图层，根据图层名作为索引，如果图层不存在，显示警告并返回失败。
2.3 创建 OGR 几何对象，计算要素质心，确定 UTM 区号，将地理坐标系的要素转为投影坐标系，执行坐标转换。
2.4 根据几何类型计算几何属性，如果是线，计算长度并保存；如果是面，计算面积并保存；
2.5 释放内存，更新属性表。

输出:新建的字段被加到了相应所选的图层。

包络矩形转面的 startConvexHullCalculate()函数伪代码：
输入:无输入，通过 UI 界面的用户选择的矢量图层作为输入。（查找mvVectorLayer，获得图层名）

1.查找输入图层，根据图层名作为索引，如果图层未找到，显示错误并返回。
2.转换每个要素的几何数据 为 OGR 几何对象，并将这些几何对象添加到一个分组中。
3.计算每个几何对象的包围矩形，并将结果存储在另一个集合中。
4.创建输出 Shapefile，将包围矩形输出到 Shapefile，遍历包围矩形集合，将每个矩形作为要素添加到 Shapefile 中。
5.关闭 Shapefile，并释放资源，将生成的 Shapefile 添加到项目中。

输出:生成并保存了包含包围矩形的 Shapefile，将其添加到当前项目中。

叠加分析 startOverlayAnalysisCalculate()函数伪代码：
输入:无输入，通过 UI 界面的用户选择的矢量图层作为输入。（查找mvVectorLayer，获得图层名）

1.配置 GDAL ，设置支持中文路径和 Shapefile 编码，获取输入图层、分析图层和输出文件路径。
2..查找输入图层，根据图层名作为索引。
3.创建输出 Shapefile，在 Shapefile 中创建图层。
4.添加输入图层和分析图层的字段到新图层，计算两图层的几何交集并创建新要素，遍历输入图层的要素；遍历分析图层的要素；计算几何交集。
5.如果交集不为空，创建新要素并保留属性。
6.关闭 Shapefile，将生成的 Shapefile 添加到项目中。

输出:生成并保存了包含叠加分析结果的Shapefile，将其添加到当前项目中。

2.2.6 栅格分析：均衡化显示

因为每个分析窗口都很简单，每个窗口都没有很复杂的结构，主要的分析功能都集成在运行按钮的槽函

数，故直接介绍分析功能的伪代码。这里主要介绍最难的均衡化显示，假彩色直接根据mvRasterLayer里的波段数据更新并mvRasterItems然后重新绘制出来即可，灰度直方图使用mvRasterItems中灰度数据使用qcustomplot绘制即可。

均衡化显示 startEqualizeRaster()函数伪代码：（查找 mvRasterLayer，获得图层名）

输入:无输入，通过 UI 界面的用户选择的栅格图层作为输入。

1.获取并查找用户选择的栅格图层，获取该图层的信息。
2.初始化图像对象，对每个波段进行直方图均衡化。
3.计算累积分布函数 (CDF)，更新图像像素值。
4.生成均衡化图像，将其添加到项目中并更新视图。

输出:成功对所选栅格图层进行了直方图均衡化，并将处理后的图层添加到项目中并更新视图显示。

2.2.7 矢量保存：保存为文本

要素保存为 WKT 文本 saveLayerToCSV()函数伪代码：

输入: 无输入，通过 UI 选择的矢量图层作为输入和保存路径。（查找mvVectorLayer，获得图层名）

1. 查找输入图层，根据图层名作为索引。
2.打开文件以进行写入，使用 mvVectorLayer 里的几何和属性表数据。遍历图层的每个要素，将几何数据转换为 WKT 格式，并写入文件。
3.关闭写入文件。

输出: 将图层数据成功保存为 CSV 文件。

2.3 详细设计表示

2.3.1 内存数据结构

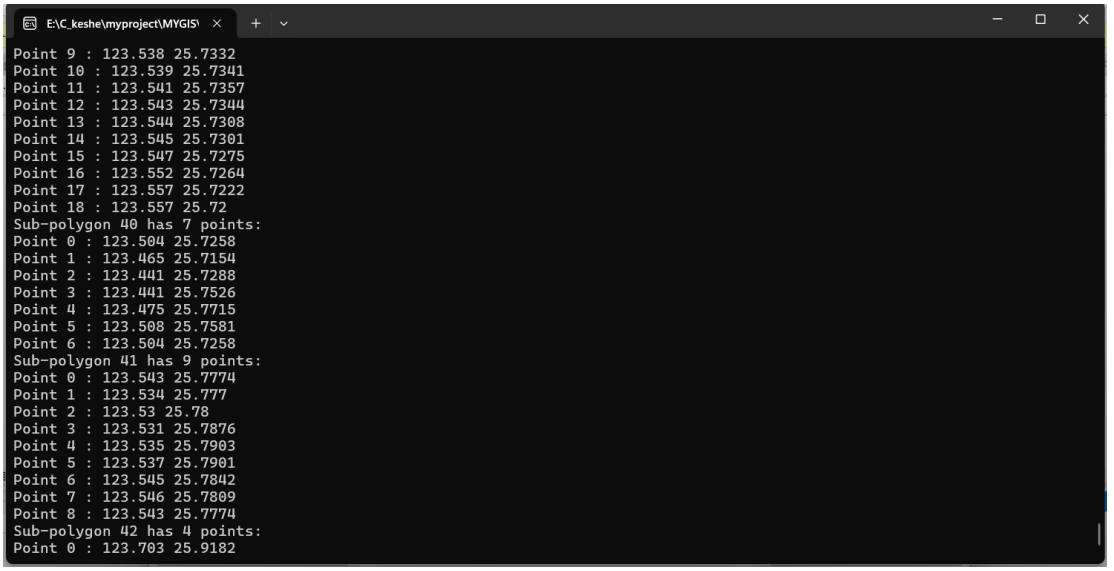

图 2.3-1GeometryData 类存储几何数据打印

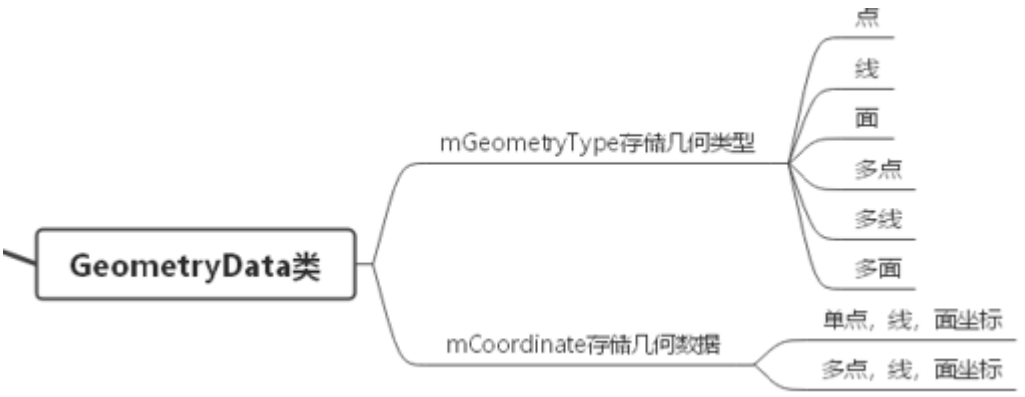

图 2.3-2GeometryData 类存储几何数据结构

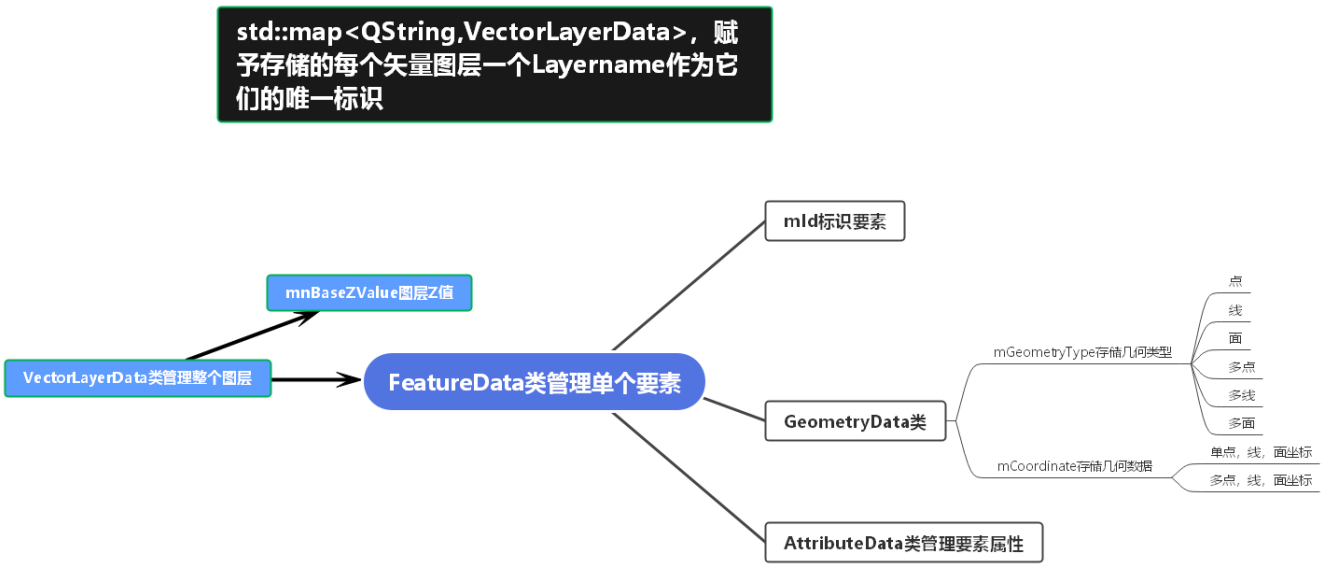

图 2.3-3 矢量存储整体结构

如图图 2.3-4所示，对于矢量的几何数据，我们存储方式是按照他的点来存储。

对于点存储为{{x1, y1}}，

对于线存储为{{x1, y1}, {x2, y2}, ...}。

对于多边形存储为 {{x1, y1}, {x2, y2}, ...}。

对于多多边形存储为{{{x1, y1}, {x2, y2}, ...}, {{x3, y3}, {x4, y4}, ...}}。

同时还根据读入时的geometry类型，对自定义的GeometryData类也赋予了相应的多点，多线，多面。

对于矢量的属性表AttributeData类，使用std::variant<int, double, QString>>，variant能让我们存储不同类型的属性数据。

FeatureData将管理AttributeData类和GeometryData类，它有一个ID作为要素的区分，也通过ID来联系视图里面item。最终VectorLayerData类存储所有的要素类。

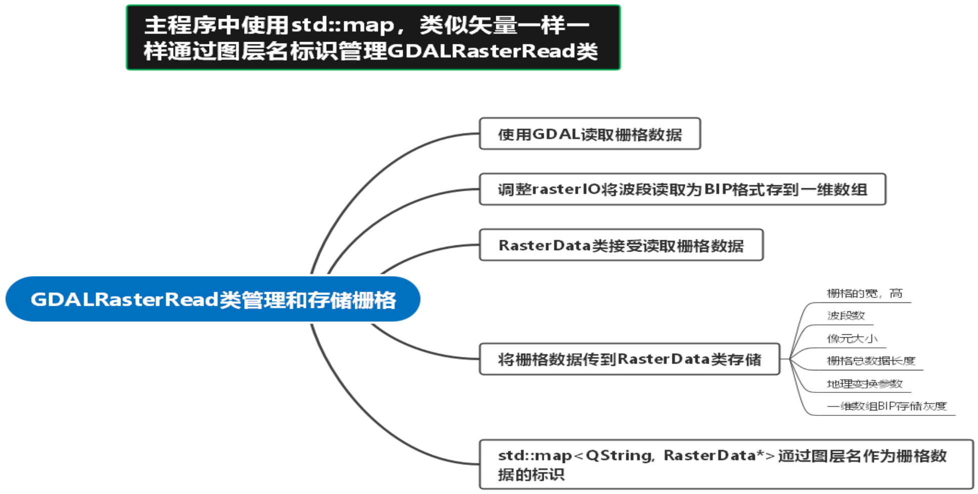

图 2.3-5 栅格存储结构

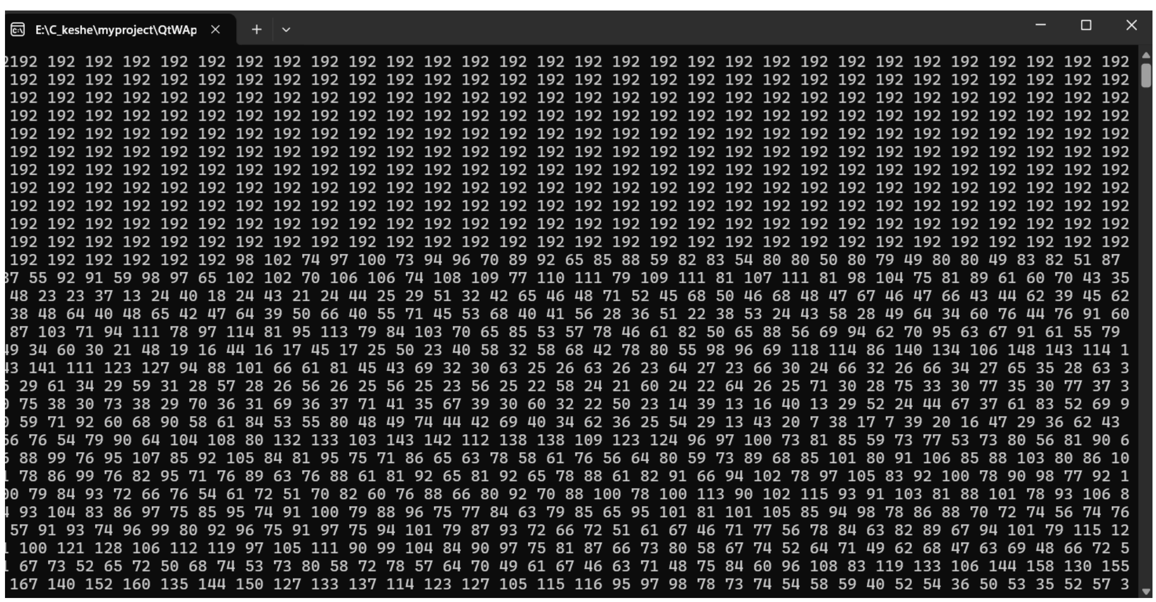

图 2.3-6debug 输出 BIP 格式栅格灰度

对于栅格数据，存储相对简单，灰度值数据我们只需要存在一个一维数组，按照BIP的格式，然后其他的宽高，波段数都可以通过复制传递到出去。

通过更改 rasterIO 的方式，设计 IO 的参数，设置每个像素直接的间隔，从而达到，逐个像素读取灰度值，而不是逐个波段读取灰度值。

2.3.2 外存文件格式

图 2.3-7 矢量数据二进制和文本保存

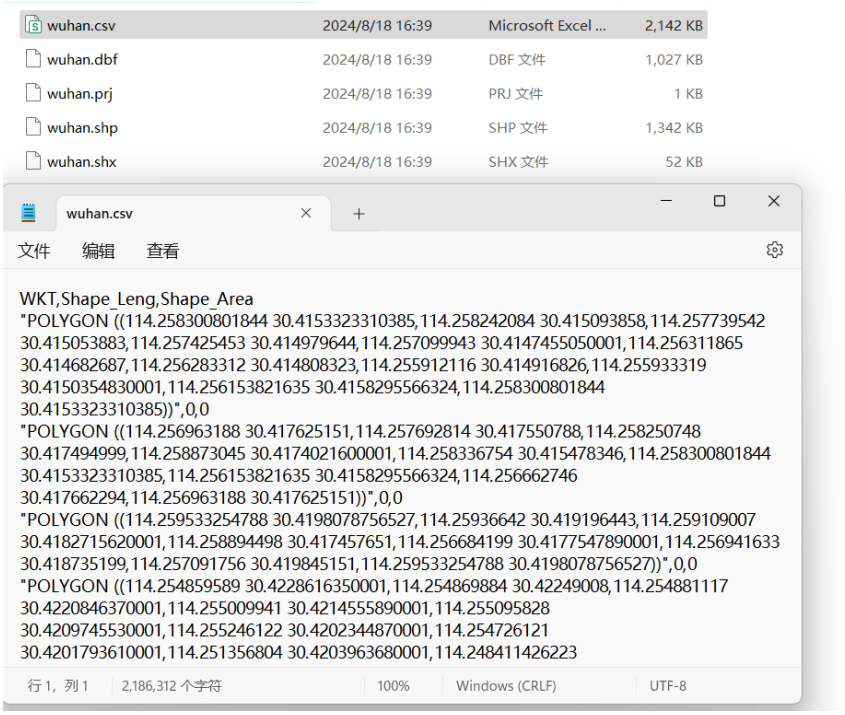

图 2.3-8 矢量外部存储

如上图所示，系统提供矢量保存为shapefile二进制格式，也支持保存为WKT文本格式。WKT文本格式，完全参照标准的WKT文件设计的存储，可以在其他arcgis软件打开，也可以在本系统打开。

图 2.3-9 工程 IO 操作

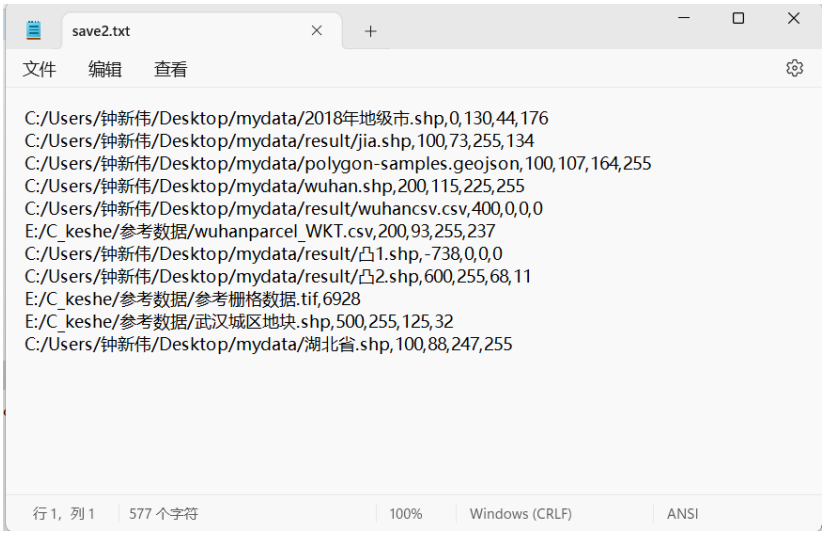

图 2.3-10 工程保存格式

如上两图所示，程序提供工程的保存和再次打开，读取。
实现思路：定义一个 ProjectManager 类，可以存储导入文件时的路径，和图层名。最后在选择保存工程时通过图层名，搜索当前视图中还存在的图层，获得他们的 z 值和图层颜色。将这些信息保存为一个 txt。
对于矢量存储为：path，z 值，rgb 图层颜色
对于栅格：path，z 值
当再次打开工程时，读取路径，设置 z 值，颜色，然后绘制出图层。

2.3.3 界面设计

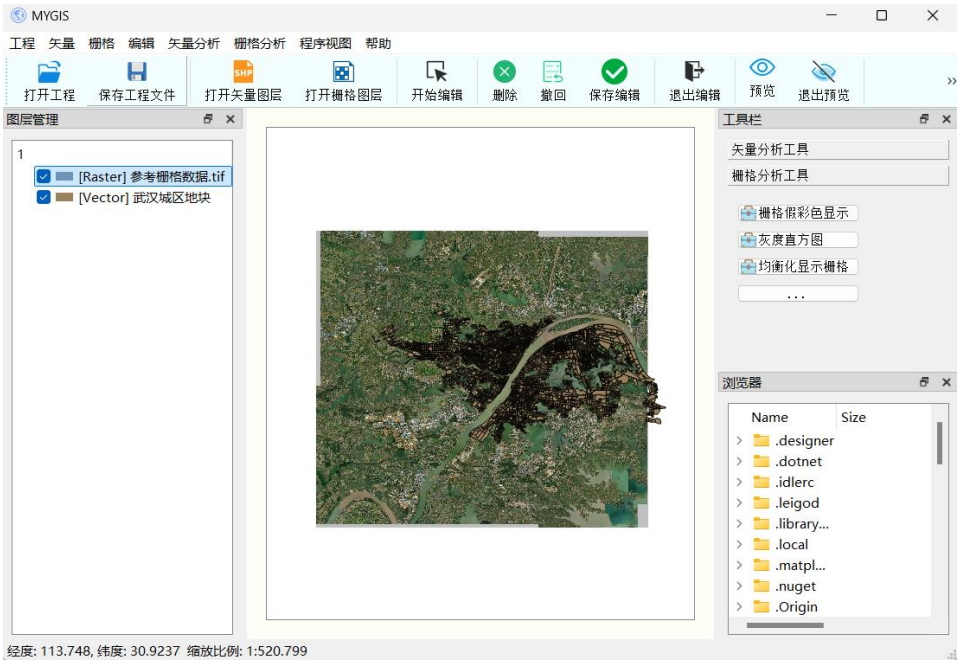

图 2.3-11 程序主 ui

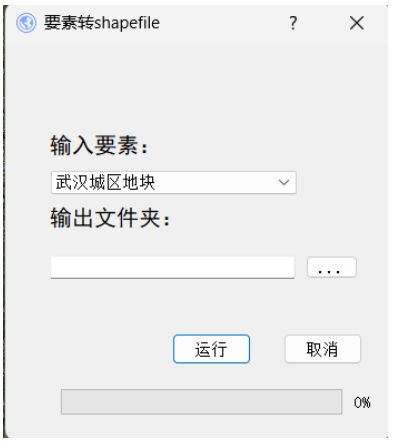

图 2.3-12 程序子 ui

如上两图所示，主程序ui分为最上部的菜单栏和工具栏，工具栏设置为text文本在Icon下方，左侧设计一个dockwidget，放置了QtreeWidget作为图层管理的列表。底部bar能够实时显示更新鼠标所在的经纬度，右侧又是两个dockwidget，一个放置工具栏toolbox和toolbutton，一个作为浏览器能够当前程序文件位置下的文件。

至于程序主 ui，一般都是一个 combox，能够根据 mvVectorLayer（导入的矢量数据）或者 mvRasterLayers，获取当前图层中的矢量数据。一个 lineEdit 作为保存的路径，一个 toolbutton 三个点表示选择保存路径，然后两个按钮，一个进度条（假的，根据函数运行进度 setValue）。

第3章 编码

3.1 矢量存储管理问题

在编程使用GDAL的过程中，我们发现GDAL对于矢量图层要素：几何和属性表的读取都是靠着指针来传递和进行，所以当我们直接对layer里面的feature进存储时，只能存一个指针，这个指针只有矢量数据集被打开时才能存在，一旦关闭数据集则该指针无效。

后面发现读取的矢量数据无法传递进内存，我们就想直接使用一个数据集管理多个矢量图层，数据集不关闭即可，但是这个想法更加荒谬，使用数据集路径进行访问，这样更加不完全，而且导致的内存泄露更多。

解决方案：抽象矢量数据，使用std::vector<std::vector<double>>，存储点，点集，最后再通过点来绘制经纬度。使用容器进行存储，就不再使用GDAL的指针，GDAL只辅助读，这样就避免了很多的指针为空问题。

3.2 投影问题

怎么让不同的矢量图层显示在相对位置，怎么让他们相对大小是正确的，怎么让他们在正确的经纬度。

在绘制矢量图层时，我们发现直接绘制的矢量图层，他们需要将经纬度转投为屏幕坐标系，然后才能在屏幕上显示，但是转为屏幕坐标系后，怎么确保他们相对大小正确，并且经纬度正确。成了问题。

解决方案：设置视图范围为地理坐标系（经度从 -180 到 180，纬度从 -90 到 90）QRectF geographicRect(-180, -90, 360, 180);直接使用经纬度来绘制，只需要确保经纬度小数点后面的精度，就无需再关注屏幕坐标问题。

3.3 图层管理问题

图层管理时，发现虽然矢量和栅格都添加到了图层管理树中，但是原先设计的图层管理只针对矢量。栅格和矢量无法区分操作。

解决方案：在将矢量或栅格图层添加进图层管理时，增加一个前缀[Vector]/[Raster]来进行区分，给与不同

类型数据，不同操作。

3.4 工程IO问题

怎么保存工程，最初的想法是：将系统内存里的mvVectorLayer（导入的矢量数据）全部存为WKT格式的文件到一个TXT中。后来发现这样存会非常慢而且没有效率。并且存储TIF时更是不可行

解决方案：在导入文件时存储文件的路径和图层名，在要保存时候获取当前mvVectorLayer，根据图层名为索引获取z轴高度和图层颜色，对于栅格也是同理。

第4章 结果分析

4.1 系统说明

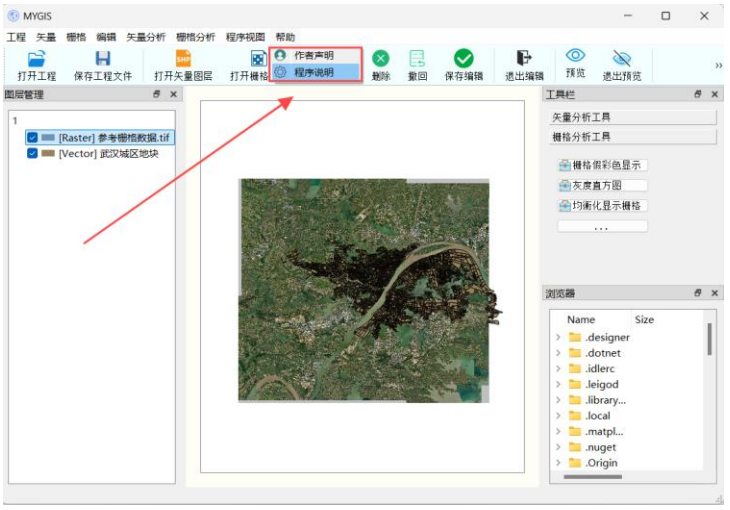

图 4.1-1 程序帮助

图 4.1-2 程序说明-1

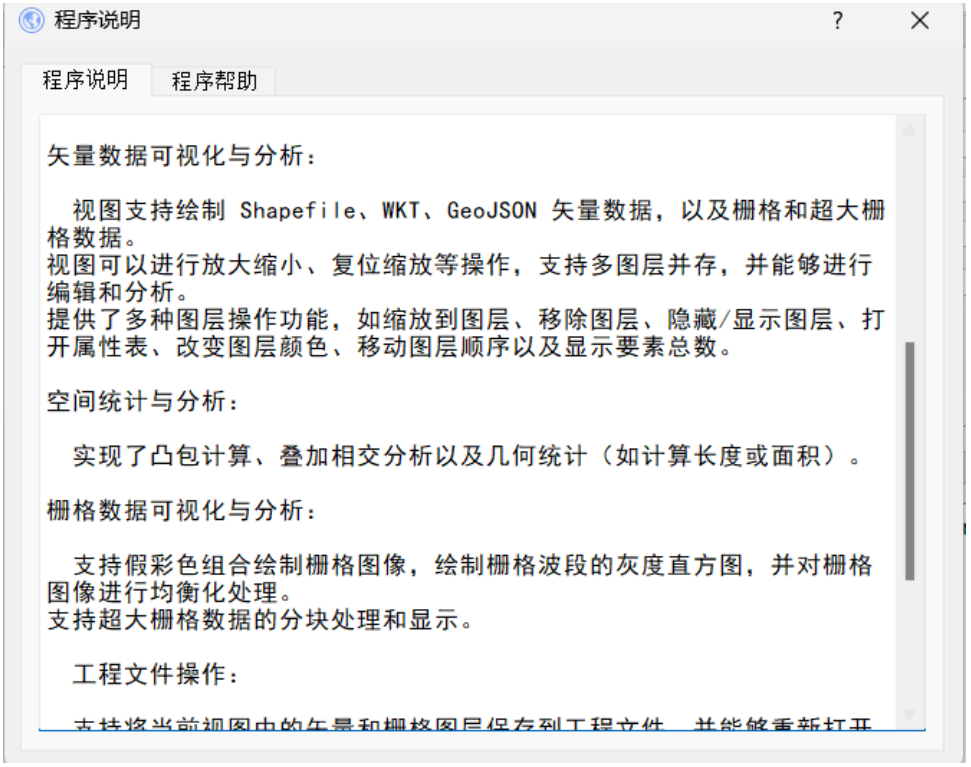

图 4.1-3 程序说明-2

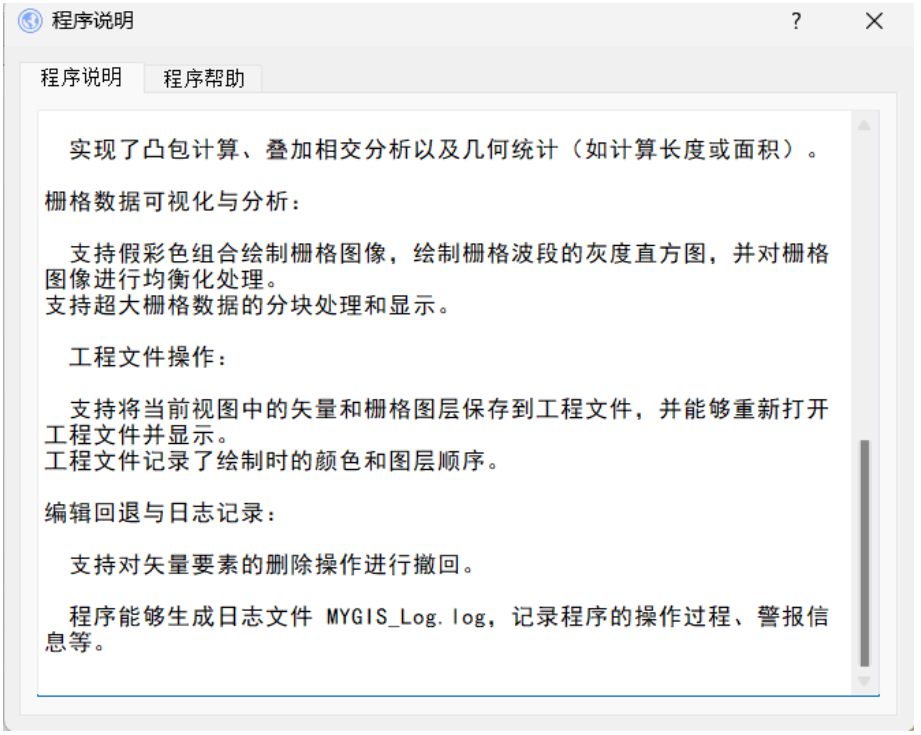

图 4.1-4 程序说明-3

图 4.1-5 程序帮助-1

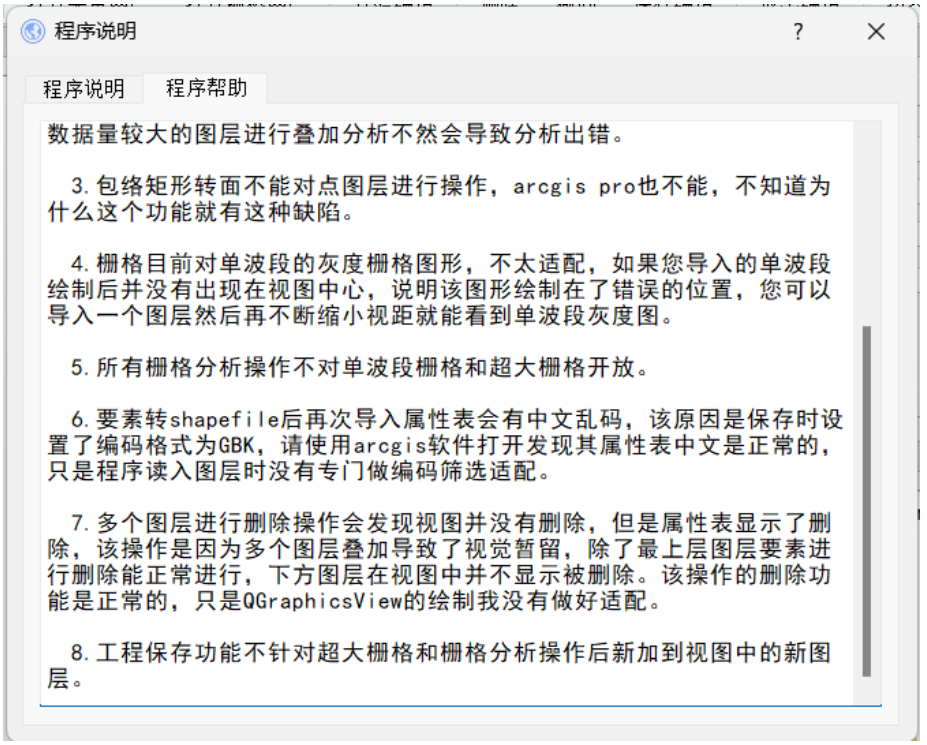

图 4.1-6 程序帮助-2

如上图所在，在程序中加了一个程序说明，能够进行程序的说明和帮助。

4.2 系统运行结果

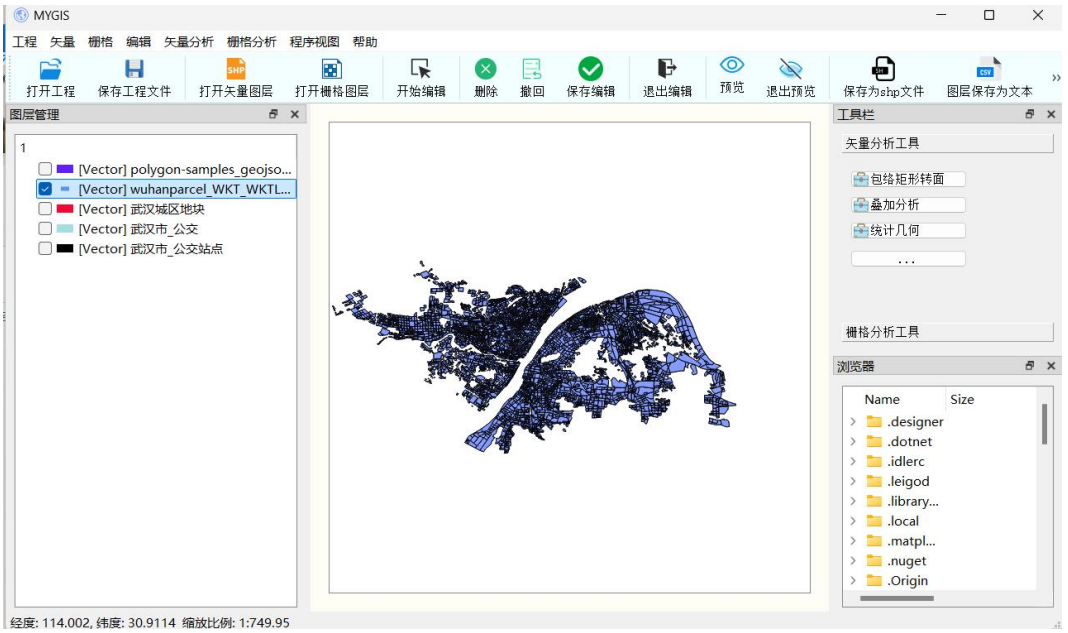

图 4.2-1 导入 WKT

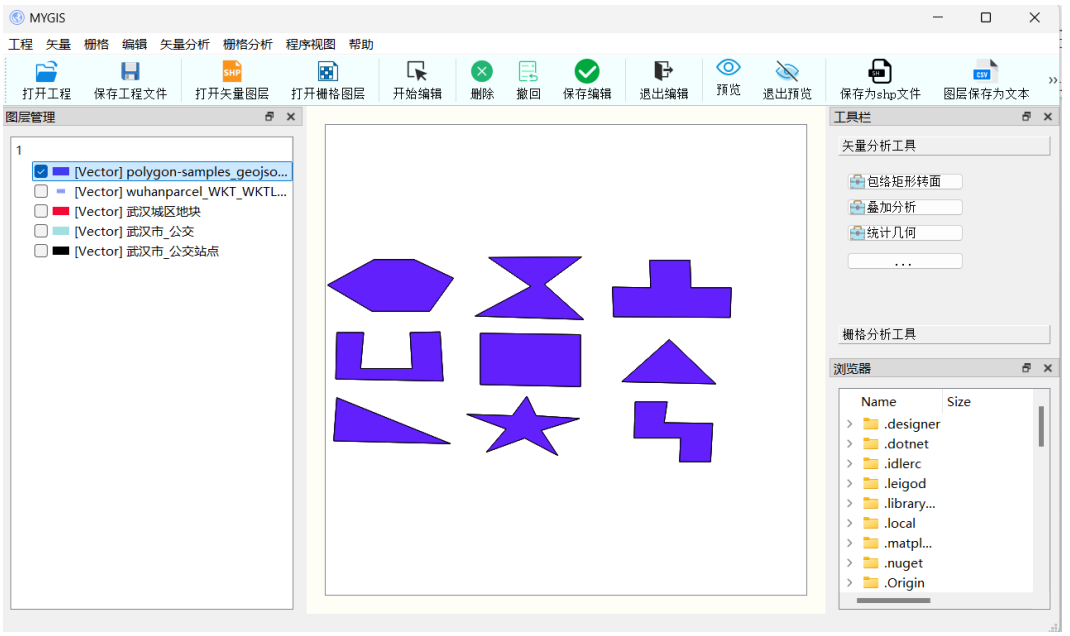

图 4.2-2 导入 geojson

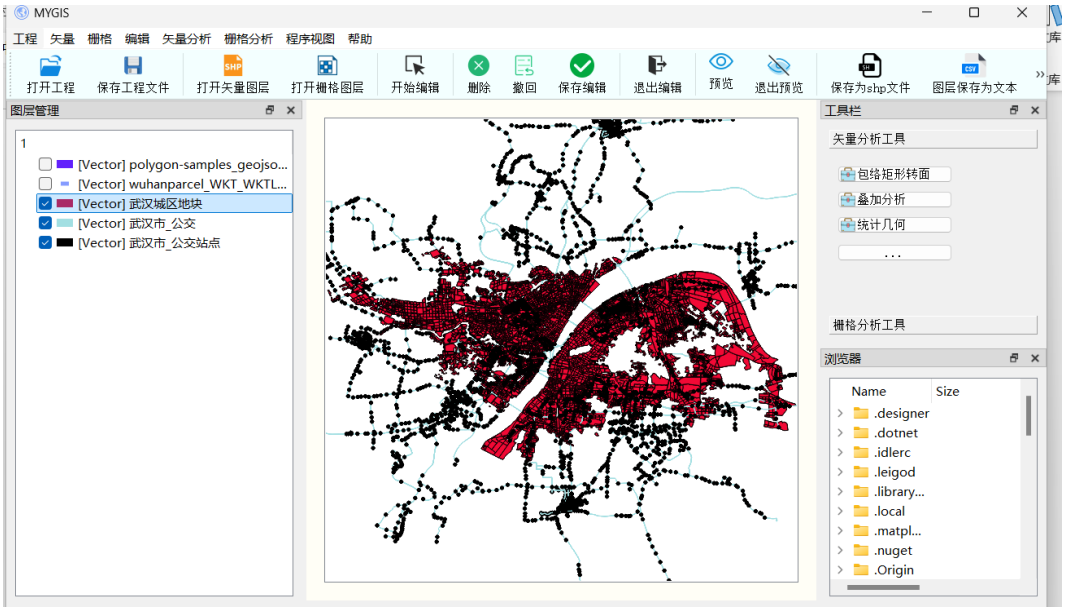

图 4.2-3 导入 shp

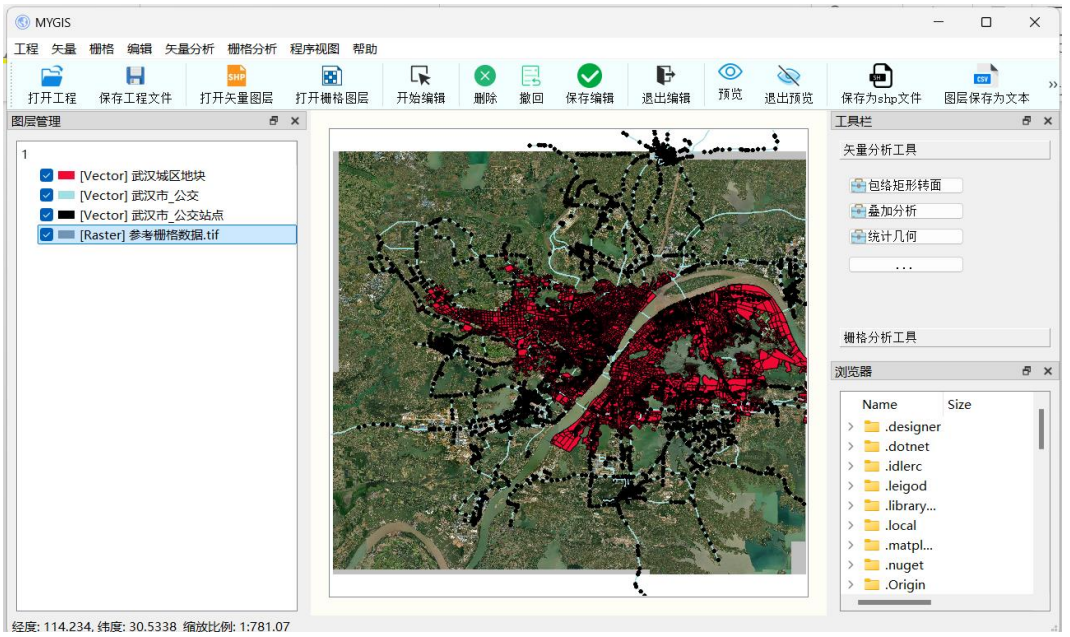

图 4.2-4 导入栅格

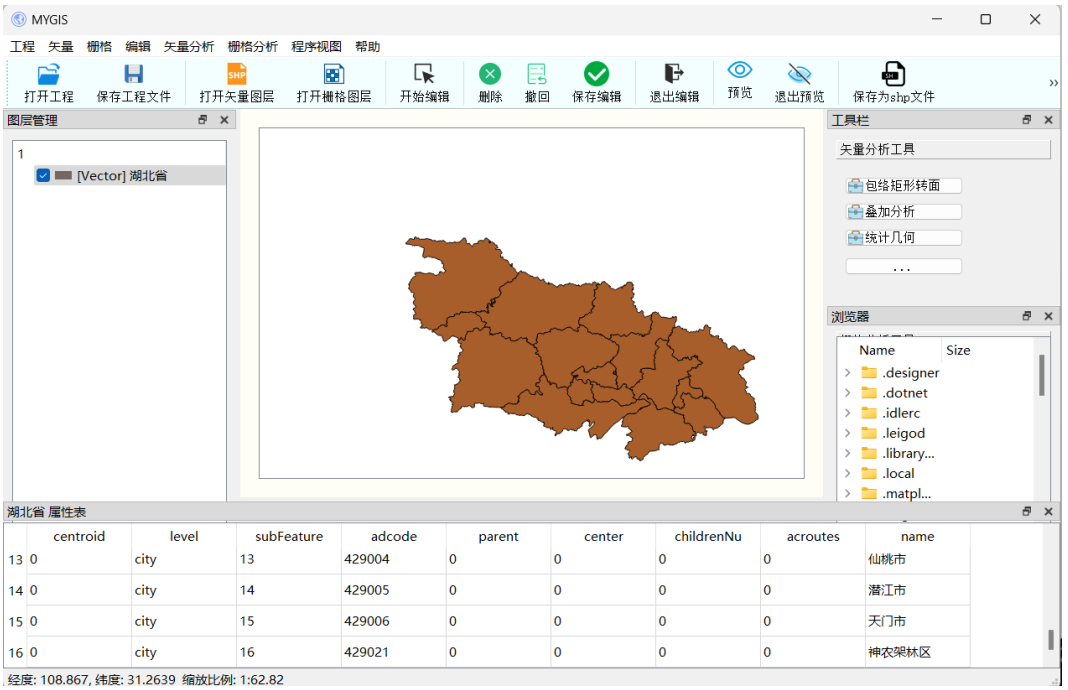

图 4.2-5 属性表和删除

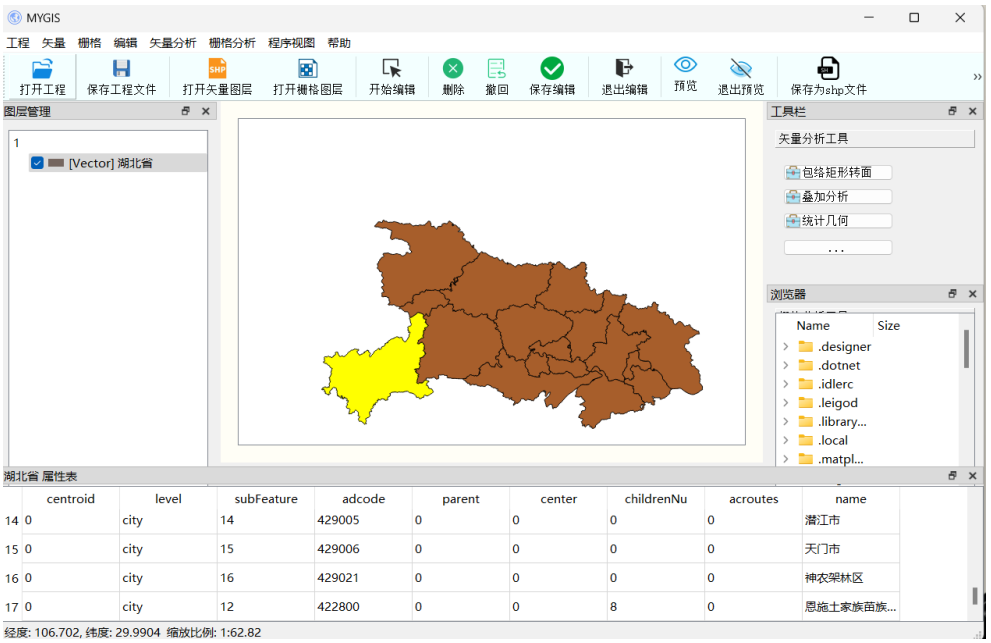

图 4.2-6 撤回

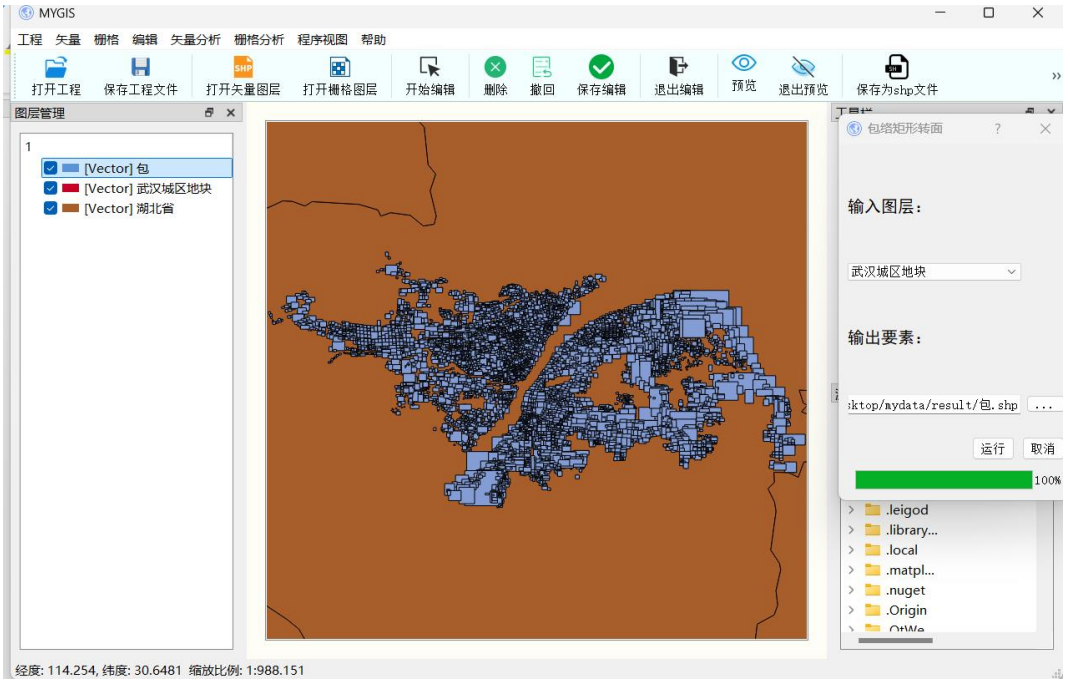

图 4.2-7 包络矩形

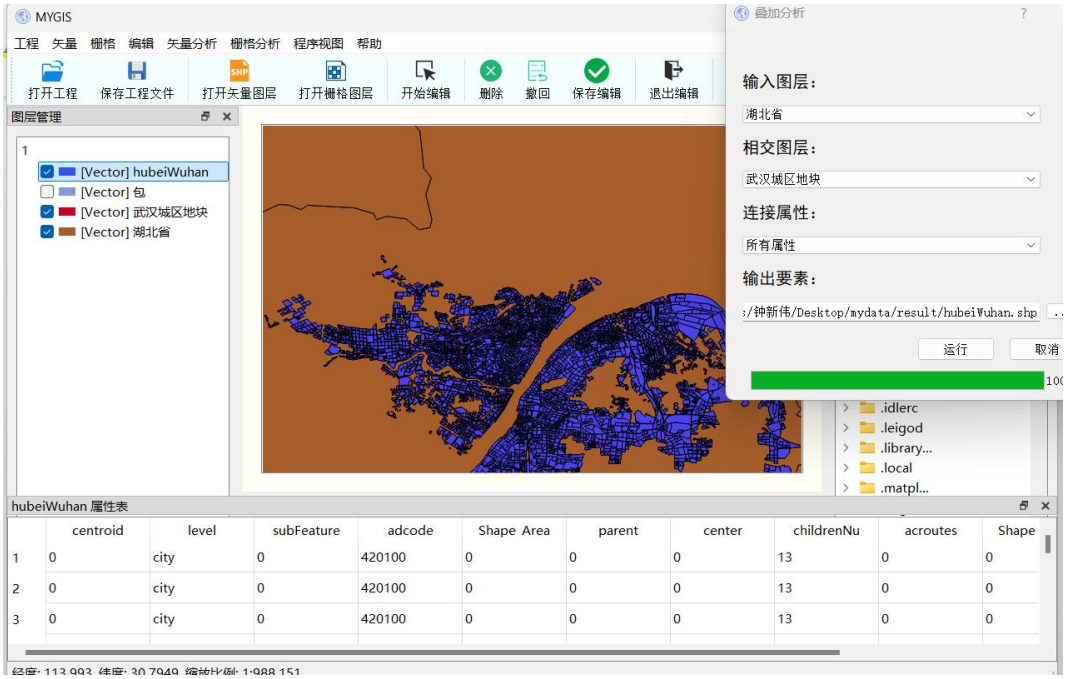

图 4.2-8 叠加分析

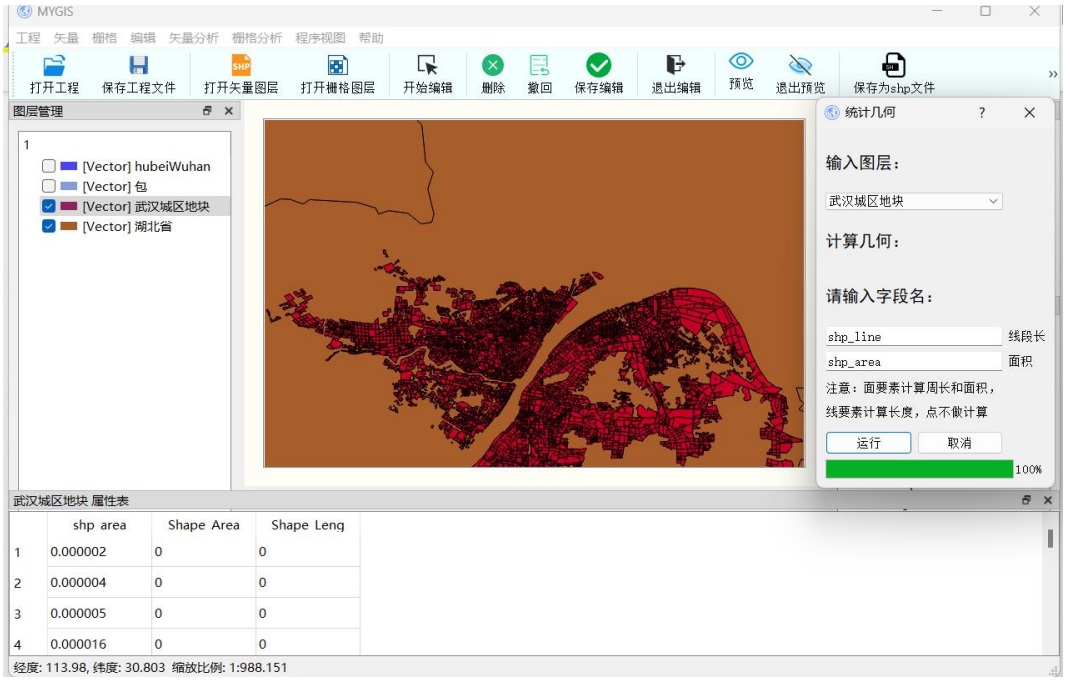

图 4.2-9 统计几何

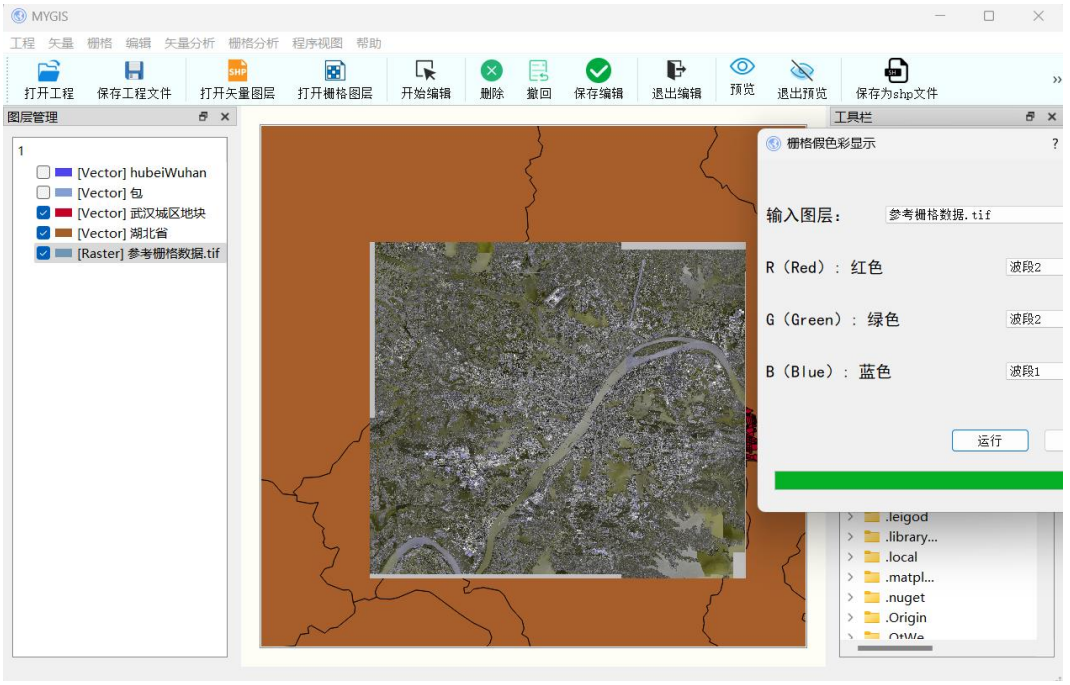

图 4.2-10 假色彩

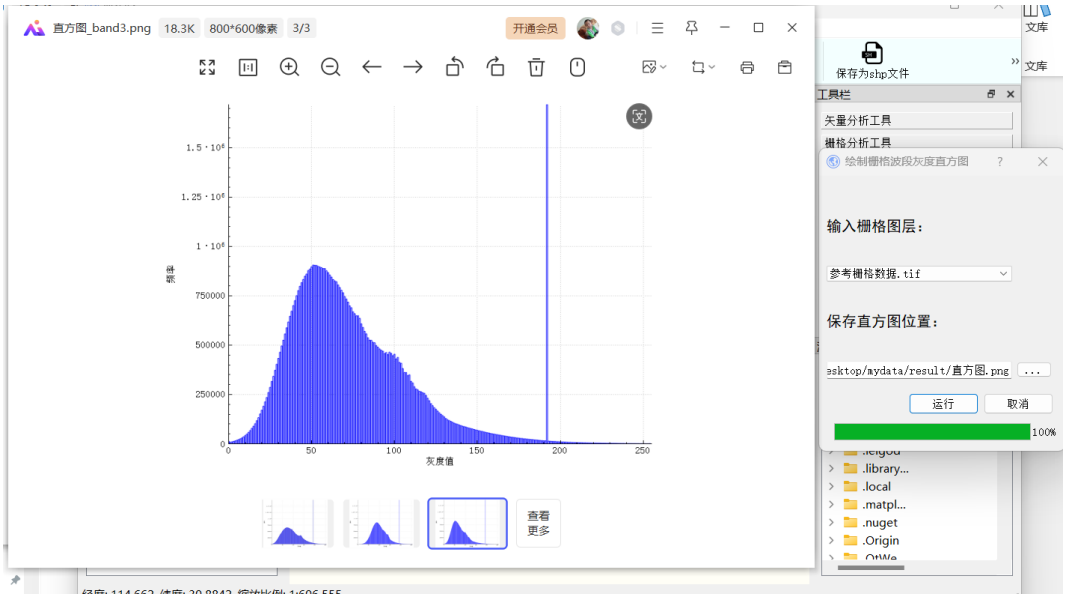

图 4.2-11 直方图

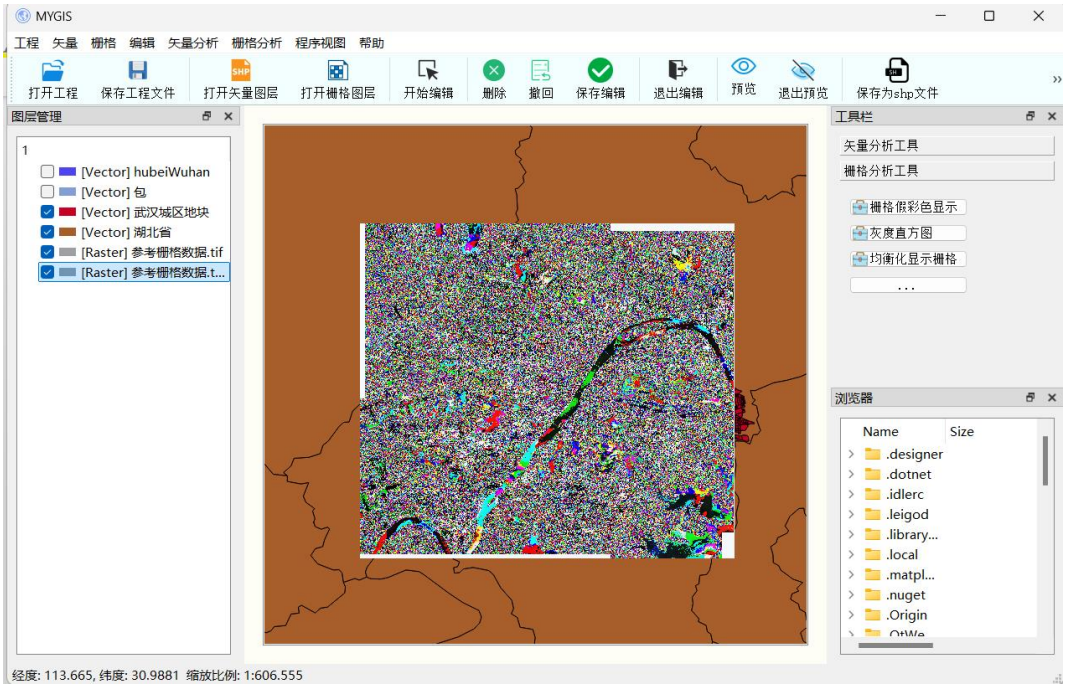

图 4.2-12 均衡化栅格

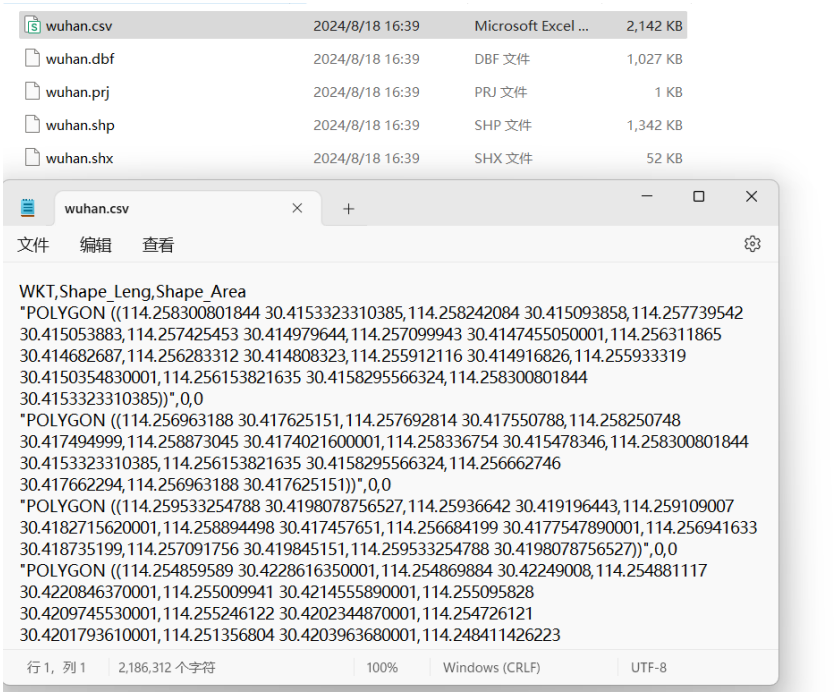

图 4.2-13 保存为 shp 和 wkt

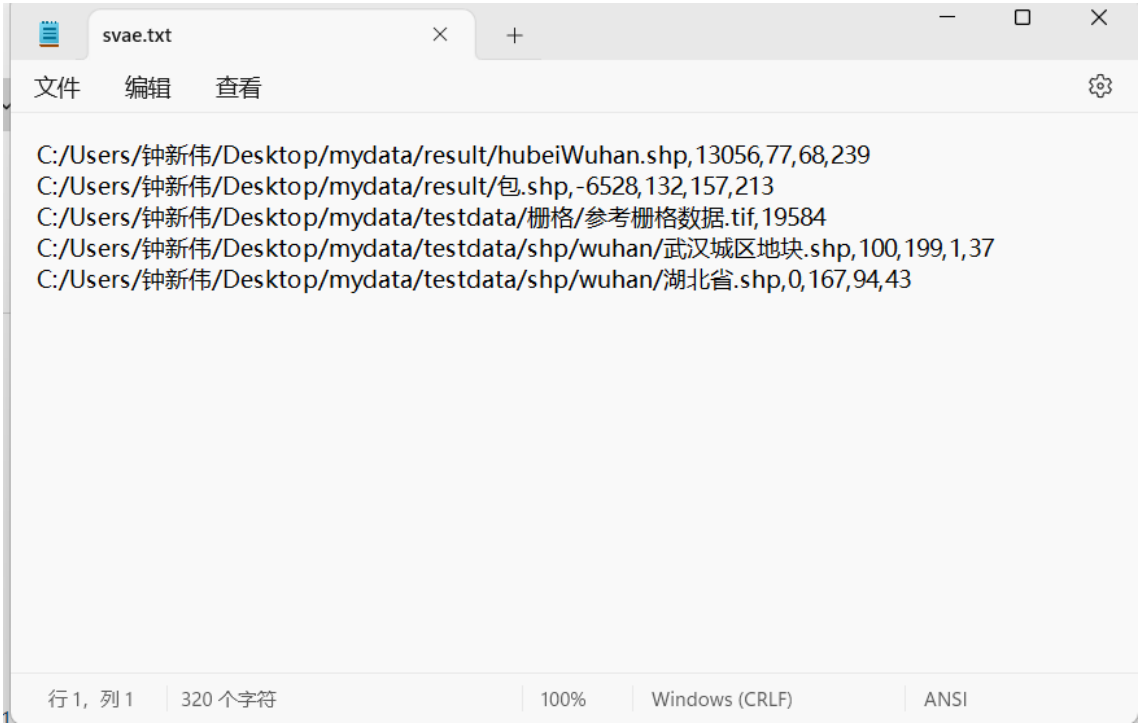

图 4.2-14 工程文件

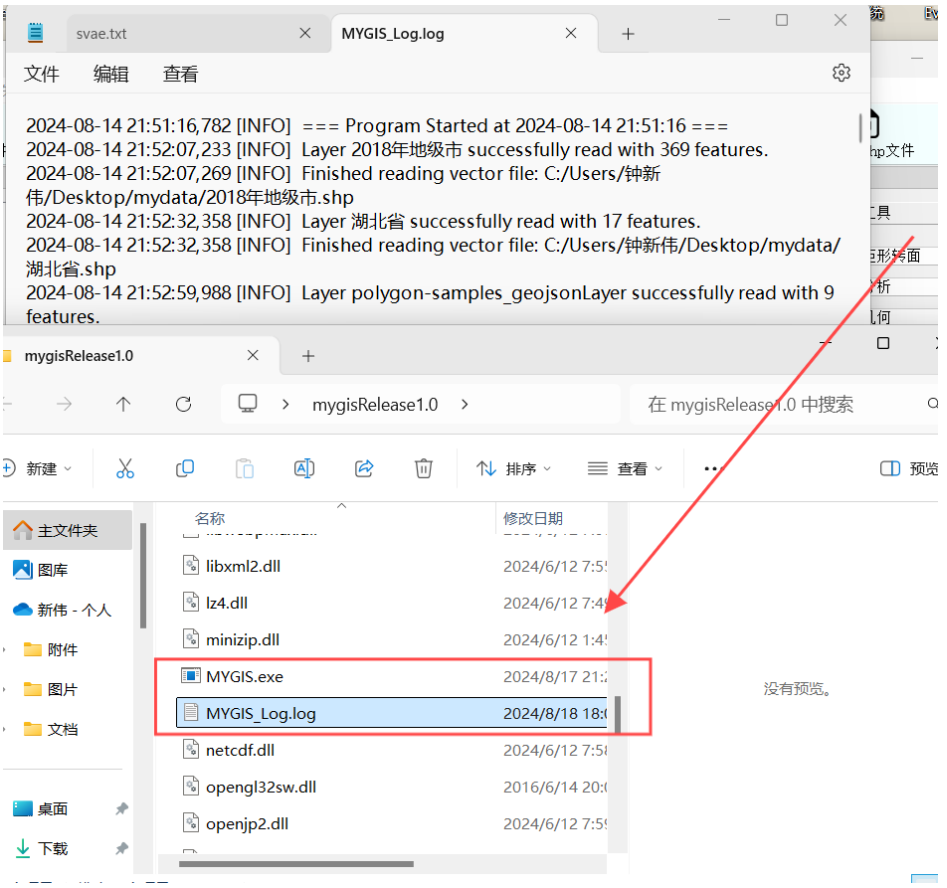

图 4.2-15 日志输出

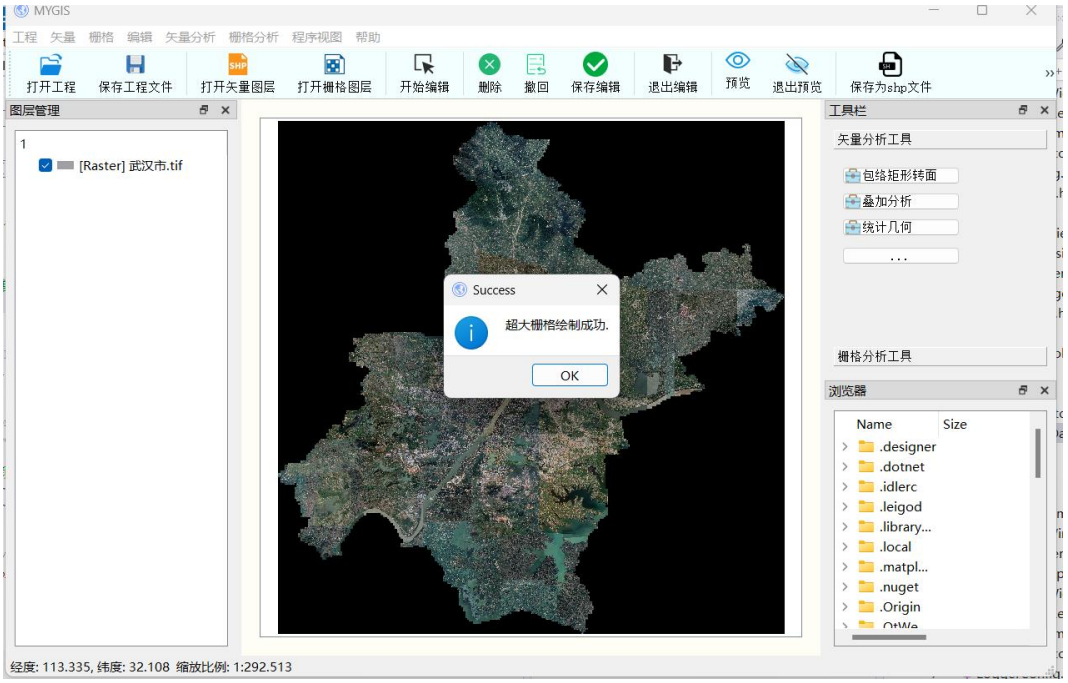

图 4.2-16 超大栅格

4.3 改进设想

关于程序：目前MYGIS程序还有很多不足，首先就是编码问题，指定保存编码格式的shapefile文件在本程序进行读取后中文会显示乱码，因为程序没有做编码选择器，在使用GDAL读取时应该对输入文件的编码进行判断。

还有就是投影问题，本程序因为没有设计投影变化，故而没有考虑相关投影问题，保存的shapefile文件都是默认WGS1984空间参考。

后续程序需要完善投影相关的操作。

第5章 小结

在本次课设中，我们学会了如何管理使用GDAL读取矢量栅格数据，如何自己定义存储类型管理数据，如何将数据可视化，如何对管理的数据进行分析操作。同时对于指针为空，内存泄露有了更多的体会和理解，指针的管理真的需要十万分注意，程序随时可能因为一个不起眼的指针没有正确销毁而不断报错卡死。

第6章 小组分工

徐子健：线下上机进行矢量读取和绘制（未完成）。

林显龙：线下上机简单完成了栅格的绘制（弃用，重写）。

于新民：做了一个TXT输出日志（弃用）。

钟新伟：线下上机完成矢量的简单读入和绘制，以及栅格的读取和抽象存储。之后重构整个系统。从头开始完成了矢量管理可视化，分析操作，栅格管理可视化，分析操作。

第7章 参考文献

[1] Qt：QCustomPlot使用教程_qt 4使用qcustomplot-CSDN博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/AwXr4
[2] 直方图实例详解（颜色直方图、灰度直方图）-CSDN博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/ZdgeY
[3] Visual Studio 2022 自动生成 C++代码类图_vs2022 生成类图-CSDN 博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/Kehc3
[4] VS2019 下打包 QT 项目的方法(包含第三方库)_vs +qt 打包-CSDN 博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/Bqgha
[5] GDAL库简介及函数说明-CSDN博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/YPviL
[6] 关于遥感中影像数据的组织方法 BIL/BSQ/BIP_bil 格式-CSDN 博客[EB/OL]. [2024-08-18]. http://t.csdnimg.cn/mU3RO

注：参考文献为China National Standard GB/T 7714-2015 格式

第8章 附录

矢量数据的读取和存入(部分代码)：

```
void MYGIS::readVectorLayer(){

// 打开矢量文件（包括 CSV 格式和 geojson）
GDALDataset* poDS = static_cast<GDALDataset*>(GDALOpenEx(fileName.toStdString().c_str(), GDAL_OF_VECTOR, NULL, NULL, NULL));
if (poDS == nullptr)
{
    QMessageBox::critical(this, tr("Error"), tr("Failed to open file."));
    return;
}

// 遍历所有图层并读取数据
for (int i = 0; i < poDS->GetLayerCount(); i++)
{
    OGRLayer* poLayer = poDS->GetLayer(i);
    if (poLayer == nullptr)
    {
        log4cpp::Category& logger = log4cpp::Category::getRoot();
        logger.warn("Warning: Layer %d is null!", i);
        continue;
```

```
}

// 创建一个新的 VectorLayerData 对象来存储当前图层的数据
VectorLayerData layerData;

// 读取要素
OGRFeature* poFeature;
poLayer->ResetReading(); // 重置读取，确保从图层的第一个要素开始
while ((poFeature = poLayer->GetNextFeature()) != nullptr) // 逐个读取图层中的每个要素
{
    // 获取要素的 FID 作为 ID
    int featureId = poFeature->GetFID();

    // 读取几何数据并判断其几何类型
    OGRGeometry* poGeometry = poFeature->GetGeometryRef();
    if (poGeometry != nullptr)
    {
        GeometryType geomType = GeometryType::Point;
        std::vector<std::vector<double>> coordinates;
        std::vector<std::vector<std::vector<double>>> multiPolygonCoordinates;

        if (wkbFlatten(poGeometry->getGeometryType()) == wkbPoint)
        {
            geomType = GeometryType::Point;
            OGRPoint* poPoint = static_cast<OGRPoint*>(poGeometry);
            coordinates.push_back({ poPoint->getX(), poPoint->getY() });
        }
        // 其他几何类型的处理逻辑...
    }

    // 创建 GeometryData 对象并存储几何数据
    GeometryData geomData = (geomType == GeometryType::MultiPolygon || geomType ==
GeometryType::MultiLineString) ?
                            GeometryData(geomType, multiPolygonCoordinates) :
                            GeometryData(geomType, coordinates);

    // 读取属性数据
    AttributeData attrData;
    for (int j = 0; j < poFeature->GetFieldCount(); j++)
    {
        OGRFieldDefn* poFieldDefn = poFeature->GetFieldDefnRef(j);
        const char* fieldName = poFieldDefn->GetNameRef();
        if (poFeature->IsFieldSet(j))
        {
            if (poFieldDefn->GetType() == OFTInteger)
```

```
            {
                int intValue = poFeature->GetFieldAsInteger(j);
                attrData.addAttribute(QString::fromUtf8(fieldName), intValue);
            }
            else if (poFieldDefn->GetType() == OFTReal)
            {
                double doubleValue = poFeature->GetFieldAsDouble(j);
                attrData.addAttribute(QString::fromUtf8(fieldName), doubleValue);
            }
            else if (poFieldDefn->GetType() == OFTString)
            {
                QString stringValue = QString::fromUtf8(poFeature->GetFieldAsString(j));
                attrData.addAttribute(QString::fromUtf8(fieldName), stringValue);
            }
        }
    }

    // 创建 FeatureData 对象并添加到图层数据中
    FeatureData featureData(featureId, geomData, attrData);
    layerData.addFeature(featureData);

    // 释放当前要素以避免内存泄漏
    OGRFeature::DestroyFeature(poFeature);
  }

  // 存储图层数据到 mvVectorLayer 中
  QString layerName = determineLayerName(extension, baseName, poLayer);
  mvVectorLayer[layerName] = layerData;

  // 存储文件路径和图层名称到 ProjectManager
  mpProjectManager->addLayer(fileName, layerName);

  logger.info("Layer %s successfully read with %d features.", layerName.toStdString().c_str(),
layerData.getFeatures().size());
}

// 关闭数据集
GDALClose(poDS);
logger.info("Finished reading vector file: %s", fileName.toStdString().c_str());
}
```

栅格数据的读入：

```
template<class TT> bool GDALRasterRead::readData()
{
    if (mpoDataset == nullptr)
        return false;

    // 计算栅格数据的总字节数并分配内存
    mnDatalength = mnRows * mnCols * mnBands * sizeof(TT);
    mpData = new unsigned char[(size_t)mnDatalength];

    // 使用 GDAL 的 RasterIO 方法读取栅格数据
    CPLErr _err = mpoDataset->RasterIO(GF_Read, 0, 0, mnCols, mnRows, mpData, mnCols, mnRows,
mgDataType, mnBands, nullptr,
        sizeof(unsigned char) * mnBands, sizeof(unsigned char) * mnBands * mnCols, sizeof(unsigned char));

    // 检查是否读取成功
    if (_err != CE_None)
    {
        std::cout << "GDALRasterRead::readData : raster io error!" << std::endl;
        return false;
    }

    return true;
}
```