## 概述

### 基本信息

1）软件信息

|  |  |
| --- | --- |
| 中文全称 | GIF动画序列核心编辑器 |
| 英文全称 | DrillGIFActionSequenceEditor |
| 标识号 | COAS |
| 编写时间 | 2021年1月 - 2021年3月 |
| 软件版本 | v1.10（编制文档时） |
| 软件类型 | 单机版应用软件 |

2）源码信息

|  |  |
| --- | --- |
| 代码文件 | 255个文件（编制文档时统计） |
| 代码行数 | 8224行（不精确统计，映射368个文件，其中包含了自动生成的头文件行数） |
| 编程语言 | C++ |
| 编程框架 | Qt 5.5 |
| 编程工具 | vs2013 |
| 其他说明 | 使用上述组合，能够将程序打包成32位和64位程序，32位程序能够在xp系统上运行。  （qt5.5是最高支持32位编译的版本，之后qt版本都不再支持32位，也就是说xp系统没法运行了。） |

### 作者提醒

这是一篇口语化的说明文档，非正式软件文档，理解难度较高。

|  |
| --- |
| 这篇文档很多细节会很难理解，因为都是想到哪写到哪。  因为很多点太难详细说明了，都是靠编程习惯和经验得出的 伪代码和知识碎片。 |

GIF动画序列核心编辑器 是一个封装好的软件成品。

|  |
| --- |
| 从用户、玩家的角度来看，软件成品是一个不可分割的独立物体；  但是从程序员的角度来看，软件成品是一个经过各种乱七八糟组合的怪物；  有经验的程序员都不会主动去了解别人的代码，因为那会释放出洪水猛兽；  所以如果你不幸打开了这个，要做好充足的心理准备。 |

该文档的主要目的是指路，具体技术细节不会特别详细说明。

|  |
| --- |
| 如果你想去探寻有价值的东西，作者我可以给你指路。  但是具体的技术细节，需要靠你自己去试。  一来，这些细节太难说清楚了，我每次认为的可行路线，都会被细枝末节折磨个半死，所以编程的第六感都是练出来的；  二来，这个编辑器的代码量，是 插件最大值编辑器 的8倍之多。我并没有那么多记忆力去记住每个细节，所以只能是哪里有印象，标注哪里。 |

程序中有许多 工具化、插件化 的对象，是可以复用的，后面章节会标出。

|  |
| --- |
| 举个例子，一个大机器被拆开后，最有价值的东西是芯片;  电路板、导线、外壳等都无关紧要，可以说都是围绕芯片功能而存在的；  芯片可以转移到其它机器上继续使用，而电路板、导线、外壳等，拆了就等同于废了。  所以，可复用的芯片，才是真正可以利用的价值。 |

C++的工具，比js插件难用。

|  |
| --- |
| js插件是8个引脚的小芯片；  C++工具是40个引脚的单片机，功能还没js灵活。  （但是，c++写起来会上瘾，不由自主地浪费大量时间完成一个小功能，目前原因不明） |

### 主要功能

|  |
| --- |
| 具体主要功能去看 ”关于GIF动画序列核心编辑器.docx” 帮助文档，这里不再赘述。 |

其实在v1.00版本完成时，作者我已经认定有了GIF生成和图片导入功能就足够了，这个软件已经固化了它的使命。

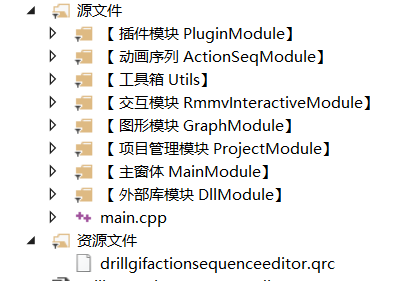
然而，令我惊讶的是，有群友提出了更有意思的功能，把多帧序列图片进行合成，变成一张大序列图片……

我震惊了，GIF生成不是那样用的啊……不过这个想法厉害了。

## 结构

### 模块简介

编辑器整体分为8个模块，如下图。



|  |  |  |
| --- | --- | --- |
| 模块 | 耦合度 | 备注 |
| 主窗体 | 极高 | 主窗体的ui界面。 |
| 动画序列模块 | 极高 | 主窗体的ui块。  包含动画序列的数据结构、编辑块、控件组成结构等。 |
| 插件模块 | 低 | 插件工具箱。  所有与 插件 相关的工具类、数据类、操作函数。  （操作函数不含任何rmmv交互，需要其他模块主动操作） |
| 交互模块 | 中 | 半定制的模块。  所有与rmmv窗口/rmmv文件的交互，都必须经过此模块。 |
| 图形模块 | 低 | 高级图形工具箱。  控件与QGraphView视图有较多的复杂操作。  使用方法比较复杂，部分强大的中间件，可能还需要继承才能使用。 |
| 项目管理模块 | 中 | 半定制的模块。  只需要修改部分宏定义，与字符串定义，就可以当工具类使用。不过，部分操作函数与主窗体密切相关，需要留意。 |
| 工具箱 | 低 | 一般工具箱，含有常用的工具类。  工具类都相对独立，详细去看看cpp中的使用方法。 |
| 外部库模块 | 低 | 扩展工具箱。  实际上也是工具类，只是为了防止导入lib库过多造成不必要的的项目臃肿而分离的。 |

|  |
| --- |
| **为什么要这样划分模块：**  一个机器设备，分为：芯片、元器件、电路板、导线。  最值钱的自然是芯片，最不值钱的当然是导线。  如果我们要从这个设备里搜索学习有价值的东西，那当然是芯片了。  拆开机器，最讨厌的当然导线，到处都是，连接复杂，而且一点也不值钱。  但我们也知道，只要我们拆了一根导线，这个机器就废了，正是因为这些导线，才能联通所有模块，实现完整的功能。  以此类推，任何一个软件产品都有极高耦合的模块，导线是必须的，不可避免。  我们为什么要看别人的代码？因为我们需要找到能为我们自己所用的东西。  我们如何评价别人的代码好坏？看他的代码能不能让我们快速找到我们要的”芯片”。 |

### UI分布图

主界面的ui分布块是拆散的，分别属于不同的模块中。

|  |
| --- |
| F:\rpg mv箱\OX0{`CGOI7PJA$J0P%79$Ew8.jpg  导航栏： [主窗体【MainModule】](#_主窗体【MainModule】)  动画序列块：[动画序列](#_动画序列)  （动画序列块用到了 可折叠选项卡 + 灵活分类树）  放映区： [放映区](#_放映区) |

|  |
| --- |
| C:\Users\lenovo\AppData\Roaming\Tencent\Users\1355126171\QQ\WinTemp\RichOle\G~}DFB9K%000}B(0A)H(X7A.png  单选表格：[单选表格](#_单选表格)  图片查看器-动图：[图片查看器-动图](#_图片查看器-动图) |

|  |
| --- |
| C:\Users\lenovo\AppData\Roaming\Tencent\Users\1355126171\QQ\WinTemp\RichOle\{51B8N8{}Z4Y~C8@UHTY0E2.png  单选表格：[单选表格](#_单选表格)  图片查看器-动图：[图片查看器-动图](#_图片查看器-动图)  快速表单： [集合组件-快速表单（widgetFastForm）](#_集合组件-快速表单（widgetFastForm）)  动画帧播放器： [动画帧播放器](#_动画帧播放器)  动画帧编辑块： [动画帧编辑块](#_动画帧编辑块) |

图片分割器 和 图片合成器 是单独的工具用窗口。

不属于软件的专属ui界面，所以这里不作拆解分析。

### 制作时间分布图

这里的统计并不完全，但有些模块实在太折磨人了，所以我还是都标记一下。

工程建立时间：2021-1-10

|  |  |  |  |
| --- | --- | --- | --- |
| 项 | 开始时间 | 所用时间 | 备注 |
| 单选表格 | 工程建立前 | 2天/维护2天 | 维护时间断断续续，非常长。 |
| 灵活分类树 | 2021-1-16 | 14天/维护3天 | 这个工具的子功能极多，看看头文件就能知道。  在那个时候几乎整4天全天写，右键菜单牵涉的功能太多，写不过来。 |
| Rmmv交互框架 | 2021-2-11 | 2天/维护1天 | 基于之前起草过插件编辑器工程，积累了两个星期的窗口交互代码，所以很快。 |
| 长度编辑器 | 2021-2-15 | 1天/维护3天 | 代码没动，只是划分结构，写说明文档居多 |
| GIF初探 | 2021-2-16 | 1整天 | 在理解xcimage的过程被虐，暂时放弃。 |
| 图片选择块 | 2021-2-18 | 2天/维护14天 | 图片选择块是动画帧编辑块的基类，虽然非常小心了，但是地基还是没打好，前后反复维护了多次。 |
| 动画帧编辑块 | 2021-2-19 | 2天/维护14天 | 右键菜单功能，能够瞬间将大量要用到的方法和函数牵出，能发现许多还没有考虑到的功能以及暂时未考虑到的大功能。  这是一个反复敲打的过程。 |
| 动画帧播放器 | 2021-2-21 | 2天/维护4天 | 为了与动画帧编辑块分离而建立的类，实际上只把计时器的功能分离出来了。其他功能都和编辑块紧密相连。 |
| 图片查看块-动图 | 2021-2-23 | 2天/维护7天 | 该类基于QgraphView，提供了鼠标缩放的功能。后来发现鼠标缩放功能太常用了，于是分离成了中间件。 |
| 动作元 | 2021-2-26 | 2天/维护4天 | 除了连接各ui的部件，还包括数据修改，json转换读取等结构。 |
| 工程模块 | 2021-2-27 | 1天 | 这部分模块是已经写好的，但是动画序列的存储数据结构，需要重新定义。 |
| GIF深入 | 2021-3-1 | 4天/维护2天 | 找到了gif.h。使用最基本的GIF生成，当时未能解决重影bug。 |
| Rmmv数据交互流程 | 2021-3-4 | 3天 | 这部分主要是为了拆解rmmv的数据结构，虽然原理简单（字符串拼接），但是代码部分比较繁杂。 |
| 放映区 | 2021-3-7 | 2天/维护1天 | 将前面的工具组合起来，放映区就成型了，我是这样想的。  然而，开始做的时候，才发现之前的工具需要强化、维护。比如单选表格的多选功能，图片查看器的变其他图功能。 |
| GIF强化 | 2021-3-11 | 3天/维护1天 | 使用了cximage库。1天写test工程，1天封装dll库，1天测试维护。  主要是群友反复询问重影问题，为什么不能导出透明问题，我头大了，就做吧。 |
| 图片合成器 | 2021-3-16 | 2天 | 发布之后，群友的新奇问题问到我了……居然还有这种想法。  （这个功能并不难做，只是要一套面板流程操作而已） |
| 图片切割器 | 2021-3-18 | 2天 | 做了这个，反向的也要做。 |
| 文档归纳 | 2021-3-20 | 14天 | 写到这里的时候，已经3月底了，而我发现仍然有好多东西没讲完。慢慢写吧。 |

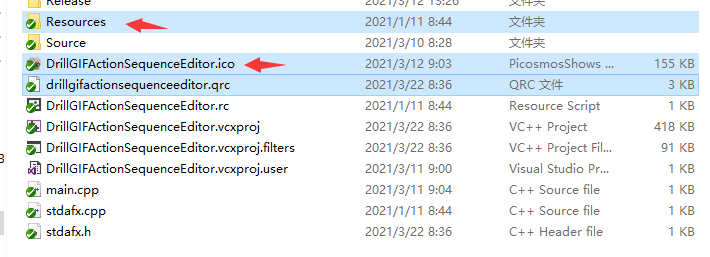
二月这一段时间，是春节；几乎全天，我都沉浸在这绵绵不断的思维泉涌之中；我依稀记得，我在当时并不高兴，因为预期一个星期能做完，结果用了近两个月。

最终，我还是承认了，C++是一门养生的语言，慢工才能出细活，不能急于求成。

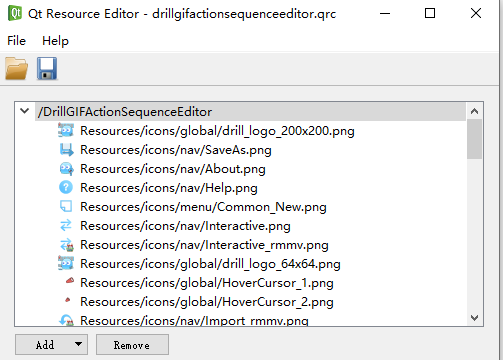
### 资源文件

所有qrc相关资源文件，都在Resources文件夹下。

（ico文件是exe的图标文件，不属于qrc资源文件）



其中放入的都是原创png图片，都符合drill图标的主题色（个人强迫症）



色卡如下：

白色：#ffffff 蓝深色：#449ed4

深灰：#525252 蓝中色：#5fc2ff

浅灰：#a6a6a6 蓝浅色：#8ad8ff

黑色：#000000

颜色在上述色卡之间徘徊（取色可以看出来），部分图标有渐变色。

另外：

图标完全打开一定有像素锯齿。

其实这个主题色，就是从我自己logo上衍生而来的。ヽ(\*。>Д<)o゜

### 流程

流程是程序内部无法改变的固定业务逻辑结构，如下图所示。

对应到”关于GIF动画序列编辑器.docx”中的 **操作示例** 章节。

**1) 主流程 - 导出文件**



**2) 主流程 - 配置到rmmv**



**3) 主流程说明**

从程序角度来说，直接进入动画帧编辑块，就能够实现导出GIF和导出序列大图。

但是该软件的主要功能是 动画序列编辑，所以固定流程必须走。

**4) 分支流程**

分支流程更加细碎化，这里不会全部列出。就部分的举例：

- 修改最大值：



- 查看rmmv中配置的动画序列：



很多流程都是经过功能块组合后产生的，所以需要尽可能保证流程中的功能块之间能够并列执行。

## 主窗体【MainModule】

**1）代码**

|  |  |
| --- | --- |
| 主窗体 | drillgifactionsequenceeditor.h  drillgifactionsequenceeditor.cpp  drillgifactionsequenceeditor.ui |
| 关于 | w\_SoftwareAbout.cpp  w\_SoftwareAbout.h  w\_SoftwareAbout.ui |
| 首页提示 | w\_SoftwareFirstTip.cpp  w\_SoftwareFirstTip.h  w\_SoftwareFirstTip.ui |

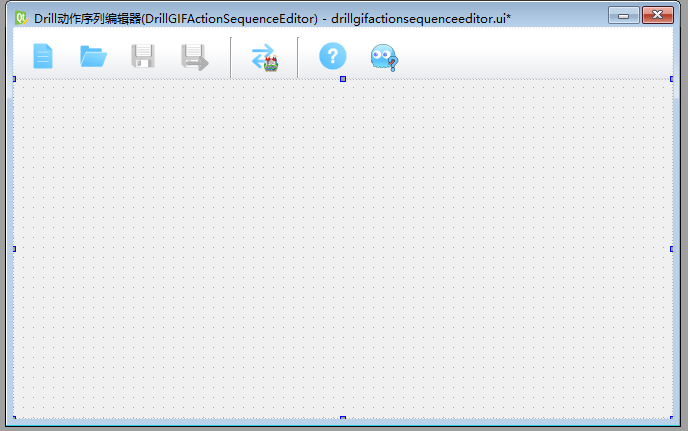
**2）界面**

主窗体就是一个原始面板，不含具体执行功能，划分成两个部分：

**内容区域：**受动画序列模块控制；

**导航栏：**提供功能按键、以及帮助信息。

打开主窗体的图像如下图：



需要说明的是，该软件的快捷键，不在主窗体中控制。

而是通过 动画序列控件块 自身的过滤器EventFilter覆写实现的。

**3）流程说明**

这里有不明bug：

在debug版本中，直接点击保存按钮，没有问题。

在release版本中，点击保存按钮，会闪退。

所以，为规避这个问题，按钮已被置灰处理，新建或打开文件才激活。

**4）与 项目管理模块 的交互**

代码中，有很多与项目交联的函数，实际上这些都只是调用项目模块的接口。

唯一控制写入的，是ui\_saveConfig() 窗口配置全局存储。

**5）窗口关闭事件**

关闭事件需要特别留意，因为在窗口关闭时，要确保 多线程 相关函数结束工作。

Qt会自动释放窗口占用的内存，但是无法释放其他线程。

QTimer定时触发器，就是使用多线程进行工作的，需要及时终止。

## 动画序列模块【ActionSeqModule】

注意，动画序列模块是集合了所有模块后的综合产物。高耦合。

最好先了解其它独立的低耦合模块之后，再来对此模块进行迭代。

### 数据结构

由于基于数据字典，这里的数据结构是多变的。

为了快速熟悉里了解，这里介绍几个关键的地方。

**1）代码**

|  |  |
| --- | --- |
| 核心复刻类 | Drill\_COAS\_Data.h  Drill\_COAS\_Data.cpp  Drill\_COAS\_Init.h  Drill\_COAS\_Init.cpp |
| 数据读取器 | s\_ActionSeqDataContainer.cpp  s\_ActionSeqDataContainer.h  s\_ActionSeqDataDefiner.cpp  s\_ActionSeqDataDefiner.h |
| 数据最大值编辑  （编辑窗口） | c\_ActionSeqLength.cpp  c\_ActionSeqLength.h  w\_ActionSeqLength.cpp  w\_ActionSeqLength.h  w\_ActionSeqLength.ui |

**2）原数据**

原数据的格式是固定的，存储在plugins.js中，或者xxxx.drillas工程文件中。

格式就是一个大字符串，字符串具有JSON格式，比如一个 动画序列类 的数据如下。

|  |
| --- |
| "{\"COAS\_type\":\"\",\"动作元-1\":\"{\\\"动作元优先级\\\":\\\"0\\\",\\\"动作元名称\\\":\\\"\\\",\\\"图像-模糊边缘\\\":\\\"false\\\",\\\"图像-色调值\\\":\\\"0\\\",\\\"备注\\\":\\\"\\\\\\\"\\\\\\\"\\\",\\\"帧间隔\\\":\\\"4\\\",\\\"帧间隔-明细表\\\":\\\"[]\\\",\\\"是否倒放\\\":\\\"false\\\",\\\"资源-动作元\\\":\\\"[]\\\"}\",\"动作元-10\":\"{}\",\"动作元-2\":\"{}\",\"动作元-3\":\"{}\",\"动作元-4\":\"{}\",\"动作元-5\":\"{}\",\"动作元-6\":\"{}\",\"动作元-7\":\"{}\",\"动作元-8\":\"{}\",\"动作元-9\":\"{}\",\"标签\":\"的萨达\",\"状态元-1\":\"{\\\"图像-模糊边缘\\\":\\\"false\\\",\\\"图像-色调值\\\":\\\"0\\\",\\\"备注\\\":\\\"\\\\\\\"\\\\\\\"\\\",\\\"帧间隔\\\":\\\"4\\\",\\\"帧间隔-明细表\\\":\\\"[]\\\",\\\"是否倒放\\\":\\\"false\\\",\\\"状态元优先级\\\":\\\"0\\\",\\\"状态元名称\\\":\\\"移动状态\\\",\\\"状态元权重\\\":\\\"40\\\",\\\"资源-状态元\\\":\\\"[]\\\"}\",\"状态元-10\":\"{}\",\"状态元-2\":\"{}\",\"状态元-3\":\"{}\",\"状态元-4\":\"{}\",\"状态元-5\":\"{}\",\"状态元-6\":\"{}\",\"状态元-7\":\"{}\",\"状态元-8\":\"{}\",\"状态元-9\":\"{}\",\"默认的状态元集合\":\"[\\\"是大多数\\\"]\"}" |

类关系如下：

|  |
| --- |
| 主容器： 动画序列列表  > 动画序列  > 动画序列属性  > 动作元列表  > 动作元属性  > 状态元列表  > 状态元属性 |

**2）变量定义**

简而言之，就是定义一个数据字典用于UI创建，用于识别js的json和c++的object。

插件Drill\_CoreOfActionSequence中

状态元 ~struct~DrillCOASState 注释

动作元 ~struct~DrillCOASAct 注释

对应了编辑器DrillGIFActionSequenceEditor中：

s\_ActionSeqDataDefiner.cpp（快速类定义器）

s\_ActionSeqDataDefiner.h

这里以状态元为例：

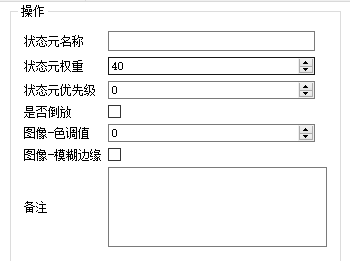
注释中，定义了状态元这一个大json的数据字典。

|  |
| --- |
| /\*~struct~DrillCOASState:  \*  \* @param 状态元名称  \*  \* @param 状态元权重  \*  \* @param 状态元优先级  \*  \* @param 资源-状态元  \*  \* @param 帧间隔  \*  \* @param 帧间隔-明细表  \*  \* @param 是否倒放  \*  \* @param 图像-色调值  \*  \* @param 图像-模糊边缘  \*  \* @param 备注  \*/ |

那么，编辑器中，也需要创建一个FastClass类来定义一个大json所带的数据字典。

|  |
| --- |
| /\*-----------------------------------  定义 - 状态元 数据表  \*/  C\_FastClass S\_ActionSeqDataDefiner::getTable\_State() {  C\_FastClass result\_class;  result\_class = C\_FastClass("状态元");  result\_class.addQString ("状态元名称", "");  result\_class.addInt ("状态元权重", "40");  result\_class.setInt\_UI\_ToQSpinBox ("状态元权重", 1, 999999, 1);  result\_class.addInt ("状态元优先级", "0");  result\_class.setInt\_UI\_ToQSpinBox ("状态元优先级", 0, 999999, 1);  // 资源-状态元 --> 动画帧编辑块  // 帧间隔 --> 动画帧编辑块  // 帧间隔列表 --> 动画帧编辑块  result\_class.addBoolean ("是否倒放", "false");  result\_class.addInt ("图像-色调值", "0");  result\_class.setInt\_UI\_ToQSpinBox ("图像-色调值", 0, 360, 1);  result\_class.addBoolean ("图像-模糊边缘", "false");  result\_class.addQString ("备注", "");  result\_class.setQString\_UI\_ToQPlainTextEdit ("备注",80,true);  return result\_class;  } |

根据数据字典，通过工具类可以定制特殊的UI，详细见 [集合组件-快速表单（widgetFastForm）](#_集合组件-快速表单（widgetFastForm）) ，输入输出都是一个 QJsonObject。



**3）变量获取**

简而言之，就是把plugins.js里面配好的数据，转成js能用的json，或者转存C++能用的qobject。

插件Drill\_CoreOfActionSequence中

变量获取 - 状态元 函数

变量获取 - 动作元 函数

对应了编辑器DrillGIFActionSequenceEditor中：

Drill\_COAS\_Init.cpp（复刻变量获取）

Drill\_COAS\_Init.h

这里继续以状态元为例：

由于数据被存到了 plugins.js里面了，取出来的时候，要进行进一步的json解析，于是就出现了下面的函数：

|  |
| --- |
| //==============================  // \* 变量获取 - 状态元  // （~struct~DrillCOASState）  //==============================  DrillUp.drill\_COAS\_initState = function( dataFrom ) {  var data = {};  data['name'] = String( dataFrom["状态元名称"] || "");  data['priority'] = Number( dataFrom["状态元优先级"] || 0);  data['proportion'] = Number( dataFrom["状态元权重"] || 40);  if( dataFrom["资源-状态元"] != "" &&  dataFrom["资源-状态元"] != undefined ){  data['gif\_src'] = JSON.parse( dataFrom["资源-状态元"] );  }else{  data['gif\_src'] = [];  }  if( dataFrom["帧间隔-明细表"] != "" &&  dataFrom["帧间隔-明细表"] != undefined ){  data['gif\_intervalTank'] = JSON.parse( dataFrom["帧间隔-明细表"] );  }else{  data['gif\_intervalTank'] = [];  }  data['gif\_src\_file'] = "img/Special\_\_actionSeq/";  data['gif\_interval'] = Number( dataFrom["帧间隔"] || 4);  data['gif\_back\_run'] = String( dataFrom["是否倒放"] || "false") == "true";  data['gif\_tint'] = Number( dataFrom["图像-色调值"] || 0);  data['gif\_smooth'] = String( dataFrom["图像-模糊边缘"] || "false") == "true";  var temp = String( dataFrom["备注"] || "" );  if( temp[0] == "\"" ){  temp = temp.substring(1,temp.length-1);  temp = temp.replace(/\\\\/g,"\\");  }  data['note'] = temp;    // > 帧间隔计算  data['gif\_intervalRealTank'] = [];  data['gif\_intervalRealTank\_total'] = 0;  for( var i=0; i < data['gif\_src'].length; i++ ){  var interval = data['gif\_interval'];  if( i < data['gif\_intervalTank'].length ){  interval = Number(data['gif\_intervalTank'][i]);  }  data['gif\_intervalRealTank'][i] = Number(interval);  data['gif\_intervalRealTank\_total'] += Number(interval);  }  return data;  } |

该类复刻到C++中，是这样的：

|  |
| --- |
| /\*-------------------------------------------------  变量获取 - 状态元（~struct~DrillCOASState）  \*/  QJsonObject Drill\_COAS\_Init::drill\_COAS\_initState(QJsonObject dataFrom){  QJsonObject data = QJsonObject();  data["name"] = dataFrom["状态元名称"].toString("");  data["priority"] = dataFrom["状态元优先级"].toInt(0);  data["proportion"] = dataFrom["状态元权重"].toInt(40);  QStringList gif\_src = TTool::\_JSON\_parse\_To\_QListQString\_(dataFrom["资源-状态元"].toString(""));  data["gif\_src"] = TTool::\_QJsonArray\_QStringToA\_(gif\_src);  QStringList gif\_intervalTank = TTool::\_JSON\_parse\_To\_QListQString\_(dataFrom["帧间隔-明细表"].toString(""));  QList<int> gif\_intervalTank\_int = TTool::\_QList\_QStringToInt\_(gif\_intervalTank);  data["gif\_intervalTank"] = TTool::\_QJsonArray\_IntToA\_(gif\_intervalTank\_int);  data["gif\_src\_file"] = "img/Special\_\_actionSeq/";  data["gif\_interval"] = dataFrom["帧间隔"].toInt(4);  data["gif\_back\_run"] = dataFrom["是否倒放"].toBool(false);  data["gif\_tint"] = dataFrom["图像-色调值"].toInt(0);  data["gif\_smooth"] = dataFrom["图像-模糊边缘"].toBool(false);  // > 帧间隔计算  QJsonArray gif\_intervalRealTank = QJsonArray();  int gif\_intervalRealTank\_total = 0;  for (int i = 0; i < gif\_src.count(); i++){  int interval = data["gif\_interval"].toInt();  if (i < gif\_intervalTank\_int.count()){  interval = gif\_intervalTank\_int[i];  }  gif\_intervalRealTank.append(interval);  gif\_intervalRealTank\_total += interval;  }  data["gif\_intervalRealTank"] = gif\_intervalRealTank;  data["gif\_intervalRealTank\_total"] = gif\_intervalRealTank\_total;  return data;  } |

整体流程上，传来的还是同一个json字符串数据，但由于语言不同，需要编写两套不同的数据处理函数。

### 动画序列

#### 动画序列（ActionSeq）

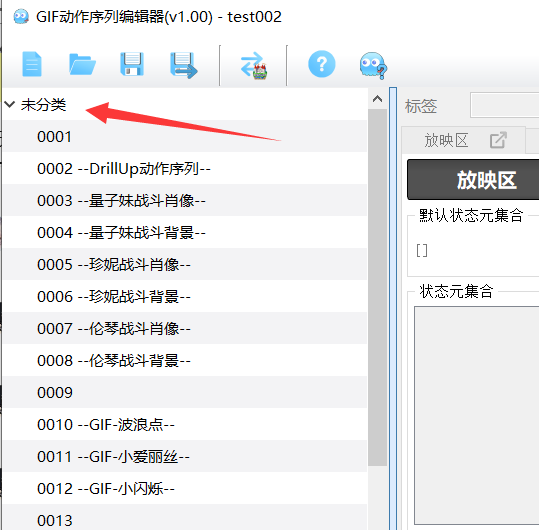
**1）代码**

|  |  |
| --- | --- |
| 动画序列块 | p\_ActionSeqPart.h  p\_ActionSeqPart.cpp  p\_ActionSeqPart.ui |

**2）界面**

动画序列的编辑块如下图，是左侧的树结构部分。

每个动画序列都是一个完整的对象，这个对象能够做出各种动作与状态。

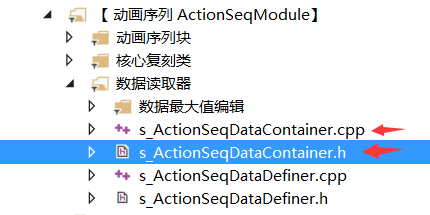


这个左侧树是基于工具类做出来的，每次改变树的选项，都将切换id对应的动画序列。

该工具类可以去看：[灵活分类树](#_灵活分类树) 。

存放动画序列的容器，可以去看数据类：S\_ActionSeqDataContainer

这个类与 插件模块、交互模块 都有调用关系，是一个牵线的中间类，耦合度非常高。



**3）定义**

**动画序列**：动画序列是一个完整的盒子，每个动画序列都包含成批量的GIF切片，即状态元 与 动作元。

动画序列、GIF动画序列、GIF动作序列 是同一个东西，只是称呼不同。



详细的定义说明，可以结合文档”关于GIF动画序列核心.docx”了解。

#### 状态元（State）

**1）代码**

|  |  |
| --- | --- |
| 状态元块 | p\_StatePart.h  p\_StatePart.cpp  p\_StatePart.ui |

**2）界面**

状态元的编辑块如下图。

由于配置的是持续循环的GIF切片，所以最好注意一下首帧和尾帧的连贯性。



**3）定义**

**状态元**：状态元本质上就是一个GIF切片，用于表示 持续执行的状态。

比如 小爱丽丝静止的GIF状态，小爱丽丝行走时的GIF状态。



状态元编辑块中每个状态元都是独立的切片单位，不含 组合切片 设置，组合情况需要去放映区看。

更多详细的定义说明，可以去看看”关于GIF动画序列核心.docx”。

|  |
| --- |
| //-----------------------------------  //----窗口  public:  bool m\_slotBlock\_source;  QList<QJsonObject> m\_stateDataList;  public:  //窗口 - 设置数据  // 【说明】：动作元只编辑数组元素，不改变数组长度。  void setData(QList<QJsonObject> stateDataList);  //窗口 - 取出数据  // 【说明】：外部接收后，还需要手动打包内容。  QList<QJsonObject> getData();  //窗口 - 本地数据 -> ui数据  void putDataToUi();  //窗口 - ui数据 -> 本地数据  void putUiToData(); |

状态元的数据始终为一个QJsonObject的形式打包传输。由于是多个状态元编辑，所以为QList<QJsonObject>。数据结构见前面章节：[数据结构](#_数据结构) 。

#### 动作元（Action）

**1）代码**

|  |  |
| --- | --- |
| 状态元块 | p\_ActionPart.h  p\_ActionPart.cpp  p\_ActionPart.ui |

**2）结构**

动作元和状态元的结构块是一模一样的，只是定义的类对象名称不一样。

这里不赘述。

#### 放映区（Playing）

**1）代码**

|  |  |
| --- | --- |
| 放映区块 | p\_PlayingPart.h  p\_PlayingPart.cpp  p\_PlayingPart.ui |
| 默认状态元集合  （编辑窗口） | w\_DefaultStateGroupEdit.h  w\_DefaultStateGroupEdit.cpp  w\_DefaultStateGroupEdit.ui |

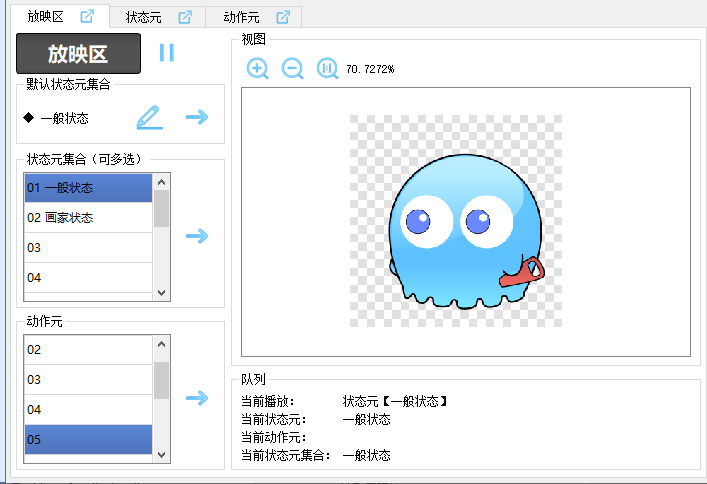
**2）界面**

放映区是在 状态块、动作元块 的基础上，垒起的耦合度最高的一个功能块类。

放映区的编辑块如下图。

放映区通过模拟输入名称指令序列的方式，对指定的动作进行播放控制。

播放的实际效果，与游戏中的图片播放一致。



只有一帧的状态元与动作元，播放效果并不明显；

如果你有多帧且复杂的gif动作，就能比较直观地理解 动画序列核心 结构。

具体可以看看后面章节 [配置行走图并播放](#_配置行走图并播放) 。

**3）定义**

**放映动画序列**：要操作这个动画序列盒子，需要从外部输入要播放的动作名、状态名、状态名列表。动画序列接收到输入后，按情况播放不同GIF图像。



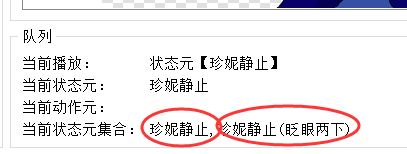
放映区的原理，就是相当于充当 子插件 调用动画序列，来达到动态变化的效果。

**4）默认状态元集合**：

点击开始播放后，放映视图会默认播放 状态元集合 的效果。



状态元集合中有多个状态元时，动画序列会在上一次状态元播放完毕后，会根据每个状态元的权重，随机抽取下一个状态元。



放映区播放的效果，与游戏中的图片实际播放效果一致。

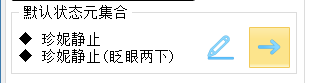
结合 放映区与状态元块 的设计，你可以灵活分配眨眼的动画时间，不需要反复打开rmmv的插件编辑器去调整参数。

**5）放映区的箭头**：

再次强调，动画序列是个黑盒子，子插件向盒子里输入动作名、状态名。状态名列表。



放映区的箭头，就是模拟名称的输入。你在放映区每点一次箭头，相当于在游戏中，调用了一次指令。（下图是子插件 图片-图片动画序列 中，执行插件指令的效果）





具体要去看子插件是如何设定的，有各种不同的方式进行调用指令。

### 快捷键

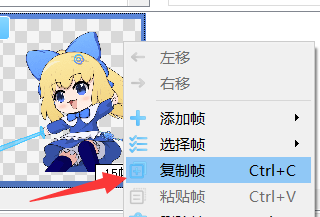
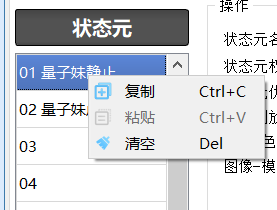
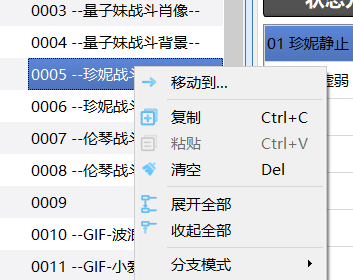
**1）相关的代码**

动画序列块、动作元块、状态元块 都有快捷键相关操作。

**2）复制粘贴**

动画序列栏、状态元编辑栏、动作元编辑栏、动画帧编辑栏，

都能够通过菜单右键，实现复制粘贴数据的功能。



将鼠标点击指定的区域，按Ctrl+C和Ctrl+V一样能够实现复制粘贴的效果，且它们之间复制的内容互不干扰。

|  |
| --- |
| //-----------------------------------  //----快捷键  protected:  QJsonObject m\_copyed\_data;  protected:  //操作 - 替换  void op\_replace(int index, QJsonObject state);  //操作 - 清空  void op\_clear(int index);  public:  //快捷键 - 事件  void keyPressEvent(QKeyEvent \*event);  //快捷键 - 复制  void shortcut\_copyData();  //快捷键 - 粘贴  void shortcut\_pasteData();  //快捷键 - 清空  void shortcut\_clearData(); |

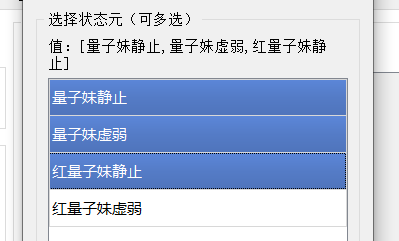
注意，上述操作只是控件内部控制的复制粘贴。没有使用到windows的剪贴板功能。也就是说，不能复制后，在别软件中粘贴出剪贴板的内容。

**3）多选编辑**

大部分控件都支持多选的功能。

按住Ctrl键，你可以选择多个项，按住shift也可以。





## 交互模块【RmmvInteractiveModule】

所有与rmmv窗口/rmmv文件的交互，都必须经过此模块。

（也就是说，要获取rmmv文件路径，都必须调用该类的函数。）

### 基本交互

考虑到后期都可能会用到同样的交互方法，

这里基本交互写成了一个半成品工具，专门针对并提供rmmv的基本交互功能。

**1）代码**

|  |  |
| --- | --- |
| 基本交互 | c\_RmmvProjectData.h  c\_RmmvProjectData.cpp  s\_RmmvDataContainer.h  s\_RmmvDataContainer.cpp |

**2）s\_RmmvDataContainer：**

该类类似于一个工具类，专门存储rmmv工程路径信息，并提供各项快捷函数。

在窗口初始化时，需要实例化。

> 提供了可存储到存档的rmmv工程数据容器。

> 提供一系列 工程文件 的获取功能，包括plugin文件夹的插件文件。

### 专用交互

**1）代码**

|  |  |
| --- | --- |
| 专用交互 | p\_RmmvCaller.h  p\_RmmvCaller.cpp  s\_RmmvCaller\_ActionSeq.h  s\_RmmvCaller\_ActionSeq.cpp |

**2）p\_RmmvCaller：**

该类需要被继承。（子类可以在保存rmmv后，执行自己的私有操作）

> 提供基本的 打开rmmv、保存rmmv、运行rmmv 功能。

> 交互通过C\_RmmvProjectData工程数据类。

因为功能涉及了窗口句柄的捕获，所以该交互功能做的非常简单纯粹，能做的就能做，不能做的，只能想别的办法了。

**3）S\_RmmvCaller\_ActionSeq：**

该类实质上就是一个流程管理器，连接 rmmv交互数据 与 操作面板 用的。

由于继承了 rmmvCaller，便可以进行打开/保存/运行rmmv操作。

该类会强制给工程更新插件、读取插件内容，这些都属于硬编码部分，与流程逻辑息息相关，这里不做多余分析了。

### 操作面板

**1）代码**

|  |  |
| --- | --- |
| 操作面板 | p\_RmmvOperateBoard.h  p\_RmmvOperateBoard.cpp  p\_RmmvOperateBoard.ui  w\_RmmvOperateBoard.h  w\_RmmvOperateBoard.cpp  w\_RmmvOperateBoard.ui |

**2）界面**

操作面板的结构如下：



**3）流程交互**

操作面板的耦合度很高，需要调用专用交互中的S\_RmmvCaller\_ActionSeq类。

简单来说，操作面板就是个UI，而专用交互才是数据控制函数。

## 插件模块【PluginModule】

该模块提供所有与 插件数据操作 相关的工具类。

### 最大值编辑器

最大值编辑器是一个打包好的完整工具，看说明调用函数即可。

具体数据定义细节，可以去看看 ”插件最大值编辑器-软件说明.docx”。

### 插件存储数据管理器

该管理器只提供操作 plugin.js 中的 插件存储数据 编辑结构。

**1）代码**

|  |  |
| --- | --- |
| 存储数据  （读取/写入） | c\_PluginData.h  c\_PluginData.cpp  s\_PluginDataContainer.h  s\_PluginDataContainer.cpp |

**2）结构**

存储数据结构如下：

|  |
| --- |
| {  ”name”: ” Drill\_CoreOfColor”,  ”description”: ” [v1.1] 系统 - 颜色核心”,  ”status”: true,  ”parameters”: {}  } |

读取后，将得到一个 QList<C\_PluginData\*> 类列表。

修改列表的内容，然后执行writePluginData，即可覆写回去。

不过要注意：

> 由于底层是QJsonDocument，所以写回去后顺序一定会乱。

> parameters参数的 {} 对象中的参数，不是 字符串object 。

举个例子：” {\”键\”:\”值\”} ”，这个值，就是一个 字符串object，需要转义（去掉斜杠”\”）才能当成正常的 {”键”: ”值”} 来使用。

## 图形模块【GraphModule】

这是一个巨大的模块。

只要与QGraphicsView相关的类，都会被放置到该模块中。

所幸现在编辑器与 游戏动态贴图 关系不大，所以内容不多。

### 模块说明

#### 接口调用模式

对象类要执行update有两种方式：

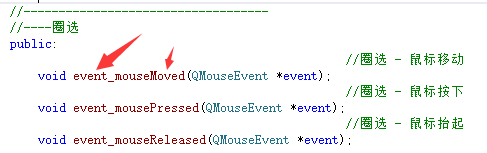
一种是自己继承update方法，然后执行自己的类方法。

另一种是提供自己update的方法让父类手动管理调用。

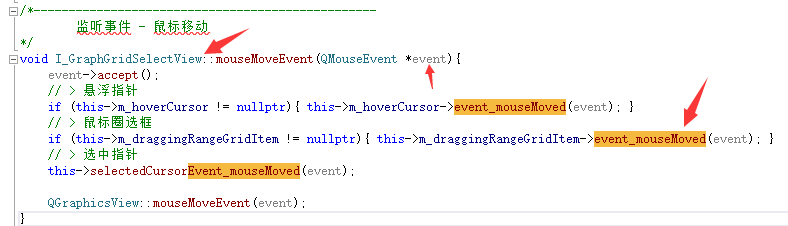
js中的sprite是使用了前者，scene调用window的ok和cancel事件是使用了后者。

c++的结构也一样，可以继承，也可被动调用。

图像模块中的贴图、中间件，都是采用了被动调用的方法，提供了event\_xxx的接口。



父类控件在实例化子类之后，还需要在自身event事件中，手动调用子类接口。

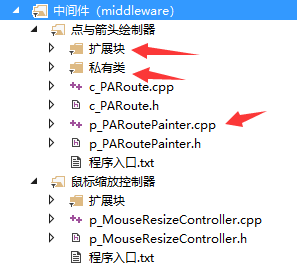


#### 基本定义

**贴图：**最小的组件，直接继承了QGraphItem，QGraphRectItem等类，并提供了相关事件响应接口。



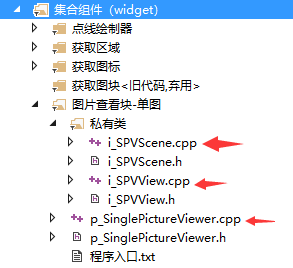
**中间件：**由多个贴图组合而成的功能性组件，其最大的特点是 依据功能 而组合，并且与其他功能相互独立，不会影响。但是，不能单独作为widget使用。



**集合组件：**该组件的特点是 具有完整的view，scene等私有继承类。

接口的类都是一个完整的widget块，可以和其他widget拼接成ui面板。

集合组件将中间件和贴图组合在一起，形成一个专有完整的图形控制功能，其中鼠标缩放块就是常用的中间件。



#### 结构类比

c++的qWidget块结构 和js的html网页结构十分相似。

视图、场景、贴图 如下表所示：

|  |  |  |
| --- | --- | --- |
|  | Qt框架 | Html5 + Pixi.js |
| 块 | QWidget | <div> |
| 视图 | QGraphicsView | <canvas> |
| 场景 | QGraphicsScene | Pixi.Stage、Scene\_Base |
| 贴图 | QGraphicsItem | Pixi.Sprite、Sprite\_Base |
| 资源图片 | QImage、QPixmap | Bitmap |

该工具类，每个类都要继承一次，然后层层调用。

就好比你在html5的div标签这一层，你要层层深入，先获取canvas，再获取Scene\_Base，最后获取到Sprite。通过在最外层的input标签，来控制最里面sprite的长宽属性

### 集合组件（widget）

#### 图片查看器-动图

**1) 结构**

用到的贴图：网格线、马赛克背景

用到的中间件： 鼠标缩放控制器

**2) 接口**

由于该查看器分别在 状态元、动作元、放映区 使用。

接口固定通过 图片资源（QFileInfo）进行交互。

|  |
| --- |
| //-----------------------------------  //----动画帧  public slots:  //动画帧 - 设置 图片资源  // 【说明1】：由于底层已固化，所以只能输入资源文件，QPixmap不支持。  // 【说明2】：如果你用索引切换帧，资源可以重复；  // 如果用资源名称切换帧，不建议重复。  void setSource(QList<QFileInfo> file\_list);  //动画帧 - 获取 图片资源  QList<QFileInfo> getSource();  //动画帧 - 清除资源  void clearSource();  //动画帧 - 切换帧（根据索引）  // 【说明】：索引如果越界，将不显示任何图像。  void setAnimFrame(int index);  //动画帧 - 切换帧（根据资源名称）  // 【说明】：没有资源会自动添加。添加时会去重。  void setAnimFile(QFileInfo file); |

状态元和动作元基于 索引编号，进行帧变化。

放映区基于 资源图片名，进行帧变化。

由于放映区随时可能会因为状态元修改而翻新，而放映区的资源图片是来不及不翻新的，所以接口函数具有 无资源时自动添加 的功能。

#### 图片查看器-单图

**1) 结构**

用到的贴图：网格线、马赛克背景

用到的中间件： 鼠标缩放控制器

**2) 接口**

单图的接口相对灵活，可用 QPixmap 和 QFileInfo 对象。

被 图片分割器 和 图片合成器 使用。

单图的特点，就是可以通过接口随时改变图像，没有动图的资源控制器功能。

|  |
| --- |
| //-----------------------------------  //----图片  public slots:  //图片 - 设置 图片资源  void setSource(QFileInfo file);  void setSource(QPixmap pixmap); |

#### 图片合成器

**1) 结构**

用到的贴图：网格线、马赛克背景

用到的中间件： 鼠标缩放控制器

用到的集合组件： 图片查看块-单图

**2）窗体说明**

该合成器的底层基于qimage，与pixi提供的bitmap的blt函数一样，从资源图片中选择一块矩形复制出来，然后建立一块新的固定高宽的pixmap，在这个pixmap中继续复制矩形。

|  |
| --- |
| inline void QPainter::drawPixmap(const QRect &targetRect, const QPixmap &pixmap, const QRect &sourceRect) |

不过需要注意的是，pixmap **不能** 在main函数中new，必须在主窗口创建完毕后点击按钮事件这种延迟new的方式才有效。

生成图片的原理其实几行代码就可以搞定，麻烦的是可视化。

不过反正作者我已经习惯了一大堆文件相互继承相互调用，那就做吧。

该控件与行编辑和列编辑的信号槽进行连接，可以实时进行变化预览。

|  |
| --- |
| //-----------------------------------  //----行列  public slots:  //行列 - 输入框变化  void rowChanged(int value);  void columnChanged(int value);  //行列 - 勾选变化（自适应）  void rowToggled(bool checked);  void columnToggled(bool checked);  //行列 - 下拉框变化  void comboxChanged(QString text);  protected:  //控件 - 计算行列  void autoFillNum(); |

#### 图片分割器

**1) 结构**

用到的贴图：网格线、马赛克背景

用到的中间件： 鼠标缩放控制器、方块阵列点选器

用到的集合组件： 图片查看块-单图

私有控件： P\_PPaViewer 切割帧选择器

**2）窗体说明**

图片分割器有两个QGraphView视图，一个是演示单图的，另一个是用来勾选帧的。

前者的网格划分与合成器的原理相似，后者用到了另一个中间件：方块阵列点选器。

两个视图分成了两个功能。

|  |
| --- |
| //-----------------------------------  //----流程  public slots:  //流程 - 进入流程1  void toFlow\_1();  //流程 - 进入流程2  void toFlow\_2(); |

数据控制的功能都在 子控件 和 集合组件 中实现了，具体可以看看 setData和getData函数，都是与 中间件/集合组件 的交互操作函数。

I\_PPaView 切割帧选择器中，将 鼠标缩放控制器、方块阵列点选器 两个中间件合并，以此实现了点选的功能。

|  |
| --- |
| //-----------------------------------  //----部件  private:  I\_PPaScene\* m\_scene;  P\_MouseResizeController\* m\_p\_MouseResizeController;  P\_MatrixBlockSelector\* m\_p\_MatrixBlockSelector;  public:  //部件 - 获取场景  I\_PPaScene\* getScene();  //部件 - 获取鼠标缩放控制器  P\_MouseResizeController\* getMouseResizeController();  //部件 - 方块阵列点选器  P\_MatrixBlockSelector\* getMatrixBlockSelector(); |

由于中间件 方块点选器 中用到了svg矢量图，所以工程需要额外导入qsvg库。由于基于视图的缩放功能非常必要，后期的svg会更多用于指示图标等功能。

### 贴图（item）

#### 马赛克背景

**1) 结构**

类型：单个贴图对象

继承于：QGraphicsPixmapItem

子容器：无

私有控件：无

**2) 功能**

提供一个直接生成背景的item对象。

可以调用相关接口函数，控制马赛克网格的密度。

|  |
| --- |
| //背景 - 设置背景（单色）  void setBackground\_oneColor(int width, int height, int block\_width, int block\_height, QColor color = QColor(255, 255, 255), int opacity = 30);  //背景 - 设置背景（双色）  void setBackground\_twoColor(int width, int height, int block\_width, int block\_height, QColor color1, QColor color2); |

#### 网格线

**1) 结构**

类型：单个容器管理对象

继承于：QObject

子容器：QList<QGraphicsLineItem\*>

私有控件：无

**2) 功能**

提供建立一系列网格线的功能。

|  |
| --- |
| public:  //网格线 - 建立网格  void rebuildGrid(int width, int height, int column, int row, QColor color);  //网格线 - 获取对象组  QList<QGraphicsLineItem\*> getItems();  //网格线 - 添加到场景  void addItemsToScene();  //网格线 - 清空对象  void clearAllItem(); |

### 中间件（middleware）

#### 鼠标缩放控制器

**1) 结构**

类型：单个容器管理对象

继承于：QObject

子容器：QList<QGraphicsLineItem\*>

私有控件：无

该中间件专门处理鼠标缩放功能，

相关接口可以去看看p\_MouseResizeController.h文件。

视图中，

你可以用鼠标直接拖拽其中的图片，移动视角。

你也可以使用 鼠标滚轮，进行放大缩小的视图缩放功能。



最大缩放比为500%，最小缩放比为5%。

#### 方块阵列点选器

## 项目管理模块【ProjectModule】

### 硬编码部分

项目管理模块是可复用的代码块，只是部分宏定义需要根据指定的工程进行定制调整。

打开模块后，留意下面的硬编码部分：

下列项目管理模块转移到新工程后可以直接使用，但硬编码部分需要修改。

**1）项目数据**

c\_ProjectData.h 文件中：

|  |
| --- |
| #define PROJECT\_SUFFIX "drillas" //项目文件后缀  #define PROJECT\_INSTANCE DrillGIFActionSequenceEditor::getInstance() //父窗口关联 |

c\_ProjectData.cpp 文件中：

|  |
| --- |
| QString C\_ProjectData::softname = "GIF动画序列编辑器"; //软件名  QString C\_ProjectData::version = "v1.12"; //软件版本  QString C\_ProjectData::version\_serial = ""; //软件版本号 |

**2）头文件引用**

修改了PROJECT\_INSTANCE之后，部分cpp文件会提示头文件引用的语法错误，修改为新的头文件引用即可。

**3）项目管理器**

s\_ProjectManager.cpp 文件中，清除数据、保存文件、读取文件部分：

注意，虽然可以通过 信号槽连接的方式 对项目管理器进行设置，但是这里的三个函数可能在部分项目下可能需要执行特殊操作。

|  |
| --- |
| /\* --------------------------------------------------------------  #项目 - 清除项目数据  \*/  void S\_ProjectManager::clearProject() { ... }  /\* --------------------------------------------------------------  #项目 - 保存（文件）  \*/  void S\_ProjectManager::createSaveFile() { ... }  /\* --------------------------------------------------------------  #项目 - 读取（文件）  \*/  void S\_ProjectManager::readSaveFile() { ... } |

另外，需要留意 历史记录 的全局存储位置。

|  |
| --- |
| /\* --------------------------------------------------------------  历史记录 - 保存记录（私有）  \*/  void S\_ProjectManager::saveHistory(){ ... }  /\* --------------------------------------------------------------  历史记录 - 读取记录（私有）  \*/  void S\_ProjectManager::loadHistory(){ ... } |

**4）窗口管理器**

窗口管理器部分可能也需要特殊操作，这里留了硬编码的接口：

|  |
| --- |
| /\*-------------------------------------------------  流程 - 新建（对话框）  \*/  void S\_ProjectWindowManager::newProject() {  bool is\_success = S\_ProjectManager::getInstance()->newProject();  // > 工程内容变化时  if (is\_success) {  //...  }  }  /\*-------------------------------------------------  流程 - 打开（对话框）  \*/  void S\_ProjectWindowManager::openProject() {  bool is\_success = S\_ProjectManager::getInstance()->openProject();  // > 工程内容变化时  if (is\_success) {  //...  }  } |

### 项目数据

**1）代码**

|  |  |
| --- | --- |
| 项目管理器 | s\_ProjectManager.h  s\_ProjectManager.cpp  s\_ProjectWindowManager.h  s\_ProjectWindowManager.cpp |
| 项目数据 | c\_ProjectData.h  c\_ProjectData.cpp  p\_ProjectCreate.h  p\_ProjectCreate.cpp  p\_ProjectCreate.ui  w\_ProjectCreate.h  w\_ProjectCreate.cpp  w\_ProjectCreate.ui |
| 扩展块 | w\_ProjectNote.h  w\_ProjectNote.cpp  w\_ProjectNote.ui |

**2）项目流程**

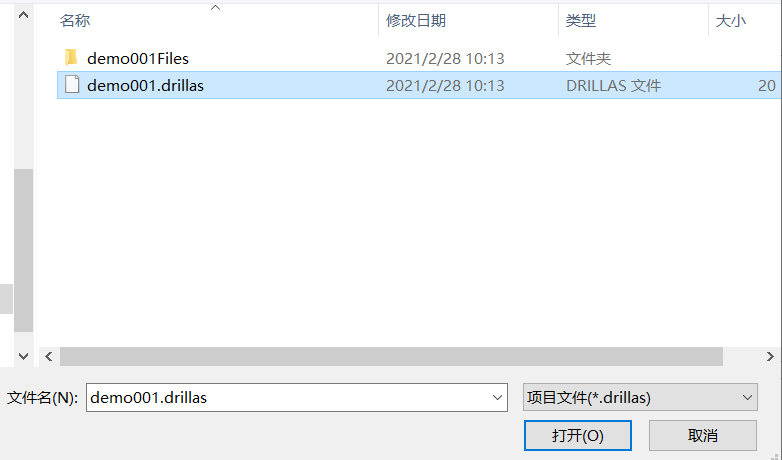
项目管理分为四种情况：新建、打开、保存、另存为。

具体流程在编辑器的帮助文档中有介绍。

**3）项目管理器功能**

**项目文件：**

保存项目时，默认建立一个项目文件夹 项目名Files，存储其他相关文件。工程文件 项目名.xxx 存储根目录，与项目文件夹同级。项目名支持"xxx.xx.xx.xx"的命名，不会出错。如下图：（”项目名.drillas” 与文件夹 “项目名Files” ）



**项目改动监听：**

如果项目中出现了任何值的改动提交，就会被记录下来，标题出现"\*"号。

此改动必须在其它地方主动调用 setDirty()才会被激活。

关闭文件时根据是否改动，提示"当前有未保存的修改，是否保存？"。

**非模态锁定：（暂未用到）**

创建非模态框时，执行项目的锁定方法，可以使得窗口在编辑期间，不能执行新建/打开/保存/另存为，也不能关闭窗口。

**最近打开的文件：（history）**

项目管理将记录最近打开的项目。新建、保存文件时，会显示使用过的文件夹路径。

**4）项目管理器的信号**

外部可以建立槽连接下列信号：

|  |
| --- |
| signals:  //全局 - 修改名称  void changeWindowTitle(QString title);  //项目 - 项目新建清空后（信号）  void newProjectFinished();  //项目 - 项目打开后（信号）  void openProjectFinished();  //项目 - 项目保存后（信号）  void saveProjectFinished(); |

### Temp文件夹

**1）代码**

|  |  |
| --- | --- |
| Temp文件夹 | s\_TempFileManager.h  s\_TempFileManager.cpp |

**2）结构**

默认c盘windows下是有一个temp文件夹以供放置缓存用文件的。但是，为了防止临时的文件被删除（一些文件需要读取后，再次使用），所以这里需要专门设置一个软件专门使用的文件夹空间，即temp文件夹。

文件夹基于工具箱的文件操作类p\_FileOperater，控制管理temp文件夹。包括：

TempToTemp内部文件复制；

ResourceToTemp从外部复制文件到内部；

generateTempFile内部生成文件；

生成Log日志文件；

文件整体保存转移到工程；

忽略特定的临时文件；

使用时注意接口的要求，尤其是 路径、文件名 的条件（这部分超级绕，从头到尾传的都是字符串）。

### 存储文件

**1）代码**

|  |  |
| --- | --- |
| 存储文件 | s\_StorageManager.h  s\_StorageManager.cpp  s\_StorageManagerTemplate.h  s\_StorageManagerTemplate.cpp |

**2）结构**

存储数据模板s\_StorageManagerTemplate是一个大的父类，使用时只要继承模板父类就可以了，数据会被集中在 项目打开/项目保存 的流程中执行；

|  |
| --- |
| //数据 - 获取存储的名称  virtual QString getSaveName() = 0;  //数据 - 清除当前管理器数据  virtual void clearAllData() = 0;  //数据 - 全部项目管理数据 -> QJsonObject  virtual QJsonObject getAllDataOfJsonObject() = 0;  //数据 - QJsonObject -> 全部项目管理数据  virtual void setAllDataFromJsonObject(QJsonObject obj) = 0; |

自动调用所有已经实例化后的子类，并自动操作子类的存储数据、读取数据功能，子类只要继承数据模板，复写相关类，然后再主窗体中getinstance就可以了，不需要修改父类。

### 存储文件-全局

**1）代码**

|  |  |
| --- | --- |
| 存储文件-全局 | s\_IniManager.h  s\_IniManager.cpp |

**2）说明**

该类基于QSettings文件的基本配置管理器，会在根目录生成一个ini文件。

此类没什么技术含量，直接使用Qsettings也可以，这个类是用于方便全局中随时使用而设定的。

## 工具箱【Utils】

工具箱中是一套完整的可拆解或组合的C++工具插件，这里对几个主要工具进行说明。

### 通用工具（common）

TTool.h和TTool.cpp是作者我常用的静态函数集合。

该集合更新很频繁，因为只要在写代码的时候，有一两个函数反反复复出现，我就会把这个函数扔到TTool里面。简化操作。

p\_FileOperater是一个文件操作的基类。

提供遍历复制/删除文件夹下全部文件的功能。

（c++的文件操作是所有程序员的噩梦，好记性不如烂笔头。一定要自己封装并总结出自己用的类）

### 管理器工具（manager）

#### GIF转换器

全局可用，提供基础的GIF读取与生成。（不含cximage）

生成GIF的底层是 gif.h （不能透明），拆解GIF的底层是QimageReader（透明GIF读取重影）。

#### 文本解析

常用类，打开长篇文章，使用正则，对每行进行 d\_indexOf ，找出关键行。

|  |
| --- |
| 关于文本解析，你最好根据自己的需求，造一个自用的正则表达式匹配的封装类。  因为现成的读取器都比较难懂，都是用的seeker光标，是增量读取的（读一行解析一行），之前尝试过在增量中获取数据，结果某一行要前面几行的数据……就手足无措了。  相比而言，全部读取后，再对每行作定量划分，容易理解代码原理和用法。  有时候就按行划分字符串都能弄死你，因为可能：存在空行、\r表示换行、\r\n表示换行、\n\r表示换行，这些复杂情况必须统一工具接口，把所有情况考虑到，才能实现完整读取数据。 |

#### 中文管理器

全局可用，提供中文转换、排序等功能。

这个类被 灵活分类树 使用，是“按名称排序”功能的底层。

### 集合组件-快速表单（widgetFastForm）

快速表单的功能是：

输入一大堆预置文本，然后转成数据表类；

系统根据数据表再转成ui输入框，并能从输入框中打包成实际数据存储。

快速类的定义，在.h头文件中已经有了非常详细的说明了，一般的数据结构，我都会定义一个s\_xxxxDataDefiner，专门存自定义的数据字典。如下图：

|  |
| --- |
| /\*-----------------------------------  定义 - 状态元 数据表  \*/  C\_FastClass S\_ActionSeqDataDefiner::getTable\_State() {  C\_FastClass result\_class;  result\_class = C\_FastClass("状态元");  result\_class.addQString ("状态元名称", "");  result\_class.addInt ("状态元权重", "40");  result\_class.setInt\_UI\_ToQSpinBox ("状态元权重", 1, 999999, 1);  result\_class.addInt ("状态元优先级", "0");  result\_class.setInt\_UI\_ToQSpinBox ("状态元优先级", 0, 999999, 1);  // 资源-状态元 --> 动画帧编辑块  // 帧间隔 --> 动画帧编辑块  // 帧间隔列表 --> 动画帧编辑块  result\_class.addBoolean ("是否倒放", "false");  result\_class.addInt ("图像-色调值", "0");  result\_class.setInt\_UI\_ToQSpinBox ("图像-色调值", 0, 360, 1);  result\_class.addBoolean ("图像-模糊边缘", "false");  return result\_class;  } |

将图中的C\_FastClass，交给C\_FastForm，便能生成ui表单。

（表单的结构与嵌套是作者我用网格比例固定好的）

这个数据字典的定义，都是用的不重复的中文名，重复的中文名表示强化这个参数的相关ui或值配置。

### 集合组件-表单（widgetForm）

#### 单选表格

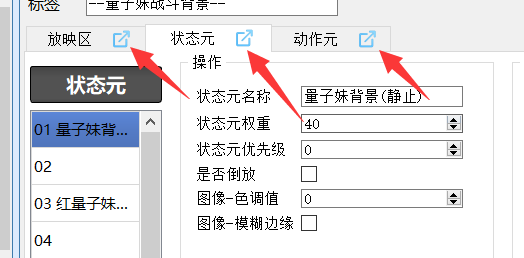
【待写完】

#### 可折叠选项卡

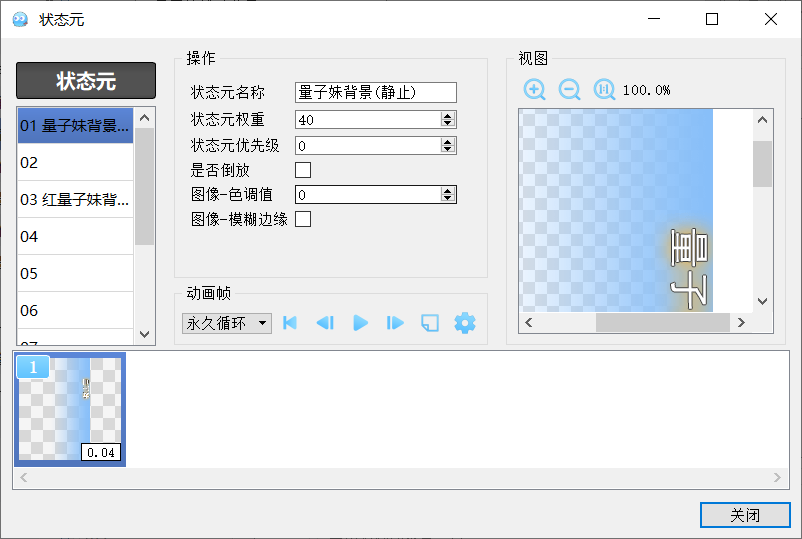
【待写完】

你可以点击 拖出 按钮，将指定的选项卡窗口化。

并且窗口可以最大化，撑开挤在一起的控件。



关闭窗口后，选项卡会恢复原位。



该功能就是为了针对rmmv极其反人类的设定：窗口不能改变大小，窗口全部都必须关掉才能编辑不能并行编辑 而设计的。

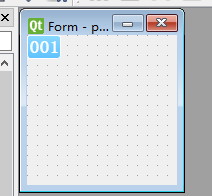
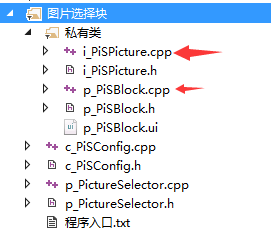
#### 图片选择块

动画帧编辑块的**基类**，显示一个列表，然后列出图片供选择。

只提供展示并选择的单一功能，不具备编辑，更不具备导入导出这些扩展功能。

这里继承了listwidget，每个item都是一个高度定制的图片ui控件，包括数字标签，马赛克背景，以及图片本身缩放的组合功能。

控件样式之类的设置，直接在ui里面修改stylesheet即可。



另一个经过反复修改的部分是 选中。

选中分为 单选和多选。

选择改变时，要发单选的信号和多选的信号以便接口对接。

同时，函数还需要提供手动选中某一项的功能，并且保证内部的信号与槽不会串一起。另外，单选和多选可以作为配置变量进行切换，方便作为多种控件功能使用。

|  |
| --- |
| signals:  //控件 - 选中选项变化（信号）  void currentIndexChanged(int index);  //控件 - 选中选项变化（多选时，信号）  void selectedIndexChanged\_Multi(QList<int> index); |

|  |
| --- |
| //-----------------------------------  //----选中  public slots:  //选中 - 设置选中  // 【参数】：数字，可以随意越界，越界后会选择最接近的一项。  // 【说明】：多选也可以用，将会保持只有一个被选中。  void selectIndex(int index);  //选中 - 设置选中（多选时）  // 【参数】：数字，可以随意越界，越界的数字没有效果。  // 【说明】：空数组可以取消所有选中。  void selectIndex\_Multi(QList<int> index\_list);  //选中 - 选中上一项  void selectLast();  //选中 - 选中下一项  void selectNext();  //选中 - 选中首项  void selectStart();  //选中 - 选中尾项  void selectEnd();  public:  //选中 - 获取选中数据  // 【说明】：-1，表示全部未选；多选时，表示第一个选中的索引。  int getSelectedIndex();  //选中 - 获取选中数据（多选时）  // 【说明】：注意，可能出现空数组情况。  QList<int> getSelectedIndex\_Multi(); |

上述为 选中 功能的常用接口与功能，如果你要写一个含选项的类，那么后期都会不可避免地要实现上面的全部函数。

### 集合组件-表单高级（widgetFormSenior）

#### 动画帧编辑块

1）QPixmap

这个类和 bitmap 功能极其相似，都具有缓存cache，反复调用不会增加系统内存占用。

缺点也比较明显，Qpixmap不能存数据，只能通过外部封装绑定。

由于Qpixmap与QfileInfo之间还有个Qimage类过渡，所以很多时候都要考虑封装还是不封装。

【待写完】

#### 动画帧播放器

【待写完】

#### 对象选择器（没用上）

该类在后期其他工具中会用到，这里就不解释了。

#### 灵活分类树

详细内容可见 灵活分类树.docx 和 灵活分类树.vsdx 的介绍。

目前有一般排序树和自定义分类排序树。

一般排序树不涉及额外内容修改，所以可以放心用，而自定义分类，需要考虑 名称外部修改、类型外部修改、id外部修改 三种情况，id被改就直接全部重刷，其他外部修改只刷新特定的id对象，树只能修改类型，且为内部修改。

【待写完】

## 外部库模块【DllModule】

### GIF相关

**1）格式说明**

GIF 格式指的是图像交换格式（Graphics Interchange Format，GIF）

该图像最初的目的，是为了无损【压缩】图片，并且保持动画性。

jpg只有rgb颜色数据，所以一定要涂上背景才能存。

png因为有rpga四种颜色数据，所以可以有透明数据。

GIF分为两种格式：一种是 GIF87a，另一种 GIF89a 。

这两种存储的颜色都【只有rgb红绿蓝三种颜色数据】，但是GIF89a加了一些扩展功能，能够让特定颜色变透明色。

也就是说，实际上gif的透明，就是一个欺骗色的映射，因为将其导出后，你会发现它实际上是有背景色的。

GIF变态处就在这里，让它实现完美透明难上加难，但是这家伙偏偏又能实现那么一点点。

在只有rgb数据的情况下，强行实现部分透明背景功能，就产生各种各样的问题，比如重影。

**2）局限性**

> 读取GIF文件 - QImageReader：

会出现重影问题。这个似乎是qt内部结构的bug。和矩阵块刷新有关，虽然翻出了源码，但是实在没有耐心理解里面的c++数组写法，因为里面的矩阵处理太复杂了，所以就放弃优化了。

> 读取GIF文件 - CXImage：

这是后来找到的一个解决方案，然而读到的基本都没问题。然而，有些屏幕截图压缩后的GIF，读出来会花屏。（一样的，没有耐心继续翻源码，直接用函数好了。）

我在探索过程中，逐渐发现：这种看似简单的问题如果仅仅是一个潜水坑，那么维护了那么长时间的开源代码作者为什么会注意不到？之前，我尝试踩了一下，然后我就掉进了深水的鸿沟里……

CXImage拆解GIF后，可以存储成 bmp、tga、tif、jpg等格式。不过那些我没有深入了。

> 写入GIF文件 - gif.h：

之前查到这个源码，还觉得这个头文件里面定义的 GifWriter\* 用起来有些不习惯。因为调取一个函数居然要传指针本身。后来，我用了 CXImage，我才知道，这种操作已经相当人性化了。

> 写入GIF文件 - CXImage：

CXImage去我写的dll工程里面可以看到。

有两个方法合成GIF：（GIF一定是8位索引色，这是文件结构规定的）

1. 直接将Qimage存成8位索引色png图片，然后让cximage去合成。

QImage可以转成QImage::Format\_Indexed8 变成 8位索引色。

但是生成的索引色后，每张图片的colorTable颜色表，**都不一样**。

【生成gif时，会直接拿第一张图片的颜色表对所有帧进行填涂，这就造成了索引颜色混乱，小爱丽丝变花的问题】

2. 正常将Qimge存成24位图片，然后让cximage合成，同时降位（图片失真）处理。

经过考虑，最后我选择了第二种方法，因为第一种生成的GIF 完全没法看 。

毕竟颜色表又是另一个巨坑。

### cximage

该编辑器的GIF生成与拆解部分，部分是基于 cximagecrt\_drill.dll 实现的。

<https://github.com/DrillUp/cximagecrt_drill>

调用的过程并不难，但是无奈原结构不支持Unicode码，所以只能先将文件转成英文ascii码，然后在对那个文件进行解析。解析完了再还原成中文。

## 其他说明

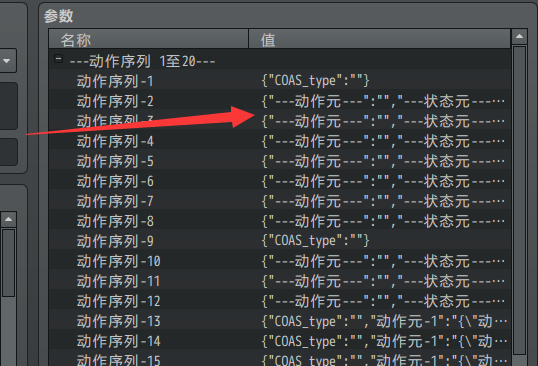
### 数据参数乱序

由于软件使用的底层是 qt5自带的QjsonDocument。

Qt5源码中，这种方式对json进行解析后，直接被放入hashmap哈希表里面，所以无论通过哪种方式，最后输出的顺序都是乱的。

之前有想过两个方法，一个是在QjsonObject里面额外存一个顺序列表；另一个是在参数名前添加前缀。但是这样做反而更复杂了，每一层都要加一次转义识别。所以放弃尝试解决此问题。

由于小工具的json生成底层与rmmv编辑器的json底层不一样，所以显示乱序属于正常现象，并不会影响游戏实际作用的效果。



双击点开参数编辑后，单条数据的 参数顺序 就会回来。

### 玄学Bug（未出现）

这里还是有必要说说，写代码几乎都会遇见玄学Bug，然而所幸的是，这个较大的工程中，我并没有遇到。

更多的是目前无法实现的功能，而不是无法解决的bug。

js插件中玄学bug经常会遇到一些，大多是指针紊乱造成的；

比如var aa = 10; 然后去执行if(aa == “10”)。

关于c++的玄学bug，大多都是 内存溢出、内存使用时 被清理造成的。

常见的有一个：

将指针(Xxx\*)强制存成(int)类型，然后再将int转回指针用。

这个bug在win7中不会出现，但是在win10里面会被当成无指向对象被清除，从而造成空指针异常。

### 关于软件文档

其实作者我是有想过：

将软件的全部结构整理一遍，然后写一套正式的软件文档，包含：软件需求分析、软件设计文档、软件测试报告、软件配置项管理等文档的。

但是后来试过之后，发现太复杂了。

专业的软件文档，不是单单介绍脚本的，还要介绍一群人开什么会，定下了什么需求，需求里面有什么说明事项，指标是什么。然后是软件工程计划，预期什么，完成了什么。接着是软件测试，软件性能指标比对，软件功能检查，然后软件验收……

如果是一群人开发软件，这样做……“似乎”有助于沟通吧。