课程 Project 报告

课程名称：	从 C/C++到 Haskell——函数式程序设计导引
PJ 名称 ：	博士的家·增强版
组员姓名：	任予琛　　高正祺　　刘诗玮
组员学号：	18212020031　18212020014　18212020026
所属院系：	微电子学院
所属专业：	微电子学与固体电子学
报告日期：	2018 年　12 月　28 日

目录

第1章 摘要

本文档为复旦大学微电子学院研究生课程“从C/C++到Haskell---函数式程序设计导引”期末project的说明。由18级科硕任予琛、高正祺、刘诗玮组成的小组使用函数式语言Haskell实现了一个界面精美的小游戏“Doctor's House - Enhanced Version”。根据要求，此游戏具有中等工作量，任予琛主要完成了整体游戏的逻辑代码；高正祺、刘诗玮主要完成了Haskell各种库的相关实现（安装、调试、使用）；游戏的资源准备由三人共同完成。**具体工作情况见下表：**

	任予琛	高正祺	刘诗玮
Nov.11 7pm-11pm	游戏调研与选择。设计游戏剧情、操作逻辑和代码框架。安装SDL2 库。	前期图形库调研，决定以 SDL2 库为基础实现游戏，实现屏幕打印出“Hello World”的图形界面。	环境配置，安装SDL2 库，解决Ubuntu16.04 下SDL2 环境搭建遇到的各种问题。
Nov.18 7pm-11pm	协助修改游戏 demo 版本的 bug。讨论确定本游戏框架和游戏资源。与高正祺在新框架下重写 demo。	实现绘制游戏开始界面的 demo 版本（未定义本游戏框架）。讨论确定游戏框架和游戏资源。与任予琛在新框架下重写 demo。	查阅 Hackage 上 SDL2 的 API 以及 Lazy Zoo 上 SDL2 的参考代码，实现 SDL2 显示图片基本流程以及多图像分块界面 demo，成功在窗口不同位置显示图片。
Nov.20 8pm-11pm	编写游戏引擎的核心函数。准备游戏资源。	查阅 Haskell 如何实现多文件编译。协助实现游戏的 main 函数，协助 debug。准备游戏资源。	准备游戏资源。
Nov.25 7pm-11pm	编写游戏引擎相关的辅助函数。第一次编译的 debug。继续准备剩余游戏资源。	根据已有的.hs 文件进行整合，第一次联合编译。继续准备剩余游戏资源。	调研文本显示，决定使用 sdl2-ttf 库，根据对伊的 C++参考代码，实现了 Haskell 下文本显示的 demo。继续准备剩余游戏资源。

Nov.27 8pm-11pm	游戏引擎代码收尾，将文本、音乐等代码合并到主函数中。	协助 debug（Haskell 中的类型错误）	调研背景音乐的添加，决定使用 sdl2-mixer 库实现背景音乐的播放。根据对应的 C++参考代码，完成 Haskell 下音频 demo。
Dec.1 7pm-11pm	调试通过可运行的第一版游戏，协助撰写游戏手册。	撰写游戏手册（游戏流程部分），协助调试第一版游戏。	撰写游戏手册（图片以及游戏布局部分），尝试根据 FFI 编写自己的 sdl2-ttf 库。
Dec.4 8pm-11pm	整理、简化、优化代码，整理代码文件和注释。	调研游戏中转场动画的实现。	撰写剩余游戏手册，调研游戏中转场动画的实现。
Dec.11 8pm-11pm	补充游戏的存档和读档功能，修改相关图片和触发域。	继续调研游戏中转场动画的实现。	继续调研游戏中转场动画的实现。
Dec.18 8pm-11pm	测试游戏，撰写本报告（游戏引擎代码等），以及演示所需的 PPT。	测试游戏，撰写本报告（SDL2 图形库等），整合报告格式。	测试游戏，撰写本报告（音乐、文本库等），以及演示所需的 PPT。
Dec.23 7pm-11pm	完善文档，准备 PPT 演示。	合并、完善文档，准备 PPT 演示。	完善文档，准备 PPT 演示。

第2章 需求分析

2.1 需求背景

本学期，笔者所修课程“从C/C++到Haskell---函数式程序设计导引”,从Haskell的角度讲述了函数式程序设计语言的特点。课程期末阶段，要求运用所学知识，完成一个中等工作量的project（设计游戏，阅读论文实现代码等），以期能够综合利用课程所学内容，熟练并掌握所学知识，增强对函数式语言的编写能力。

2.2 开发目的

依照课程要求，为加深笔者对函数式语言的理解，笔者开发了具有图形界面的小游戏“Doctor’s House”。望本游戏能够符合课程要求，望在过程中提升笔者的软件开发能力，望本游戏能够受到广大玩家的喜爱与支持。本游戏图形界面精美，可供各位游戏爱好者，特别是密室解密类游戏爱好者体验与试玩。笔者也希望本游戏能够为其他游戏软件开发者提供参考与灵感。

第3章 游戏设计

游戏有多种类别:动作类、设计类、卡牌类、装扮类等等。在前期规划时， 笔者讨论了多种设计思路:如卡牌游戏，策略游戏,以及我们达成共识的马里奥等等。但是在调研之后，我们发现马里奥的实现过于复杂，原因如下：（1）含有十分复杂的判断逻辑，例如碰撞检测、马里奥的运动计算；（2）地图的实现，由于马里奥是一个动态的游戏，如何显示地图也是一个很大的难题。最终，我们一致决定设计并实现一个中等难度的密室解密类的游戏，原因如下:

(1) 密室解密类游戏是一类热门游戏，趣味性强，且益智。

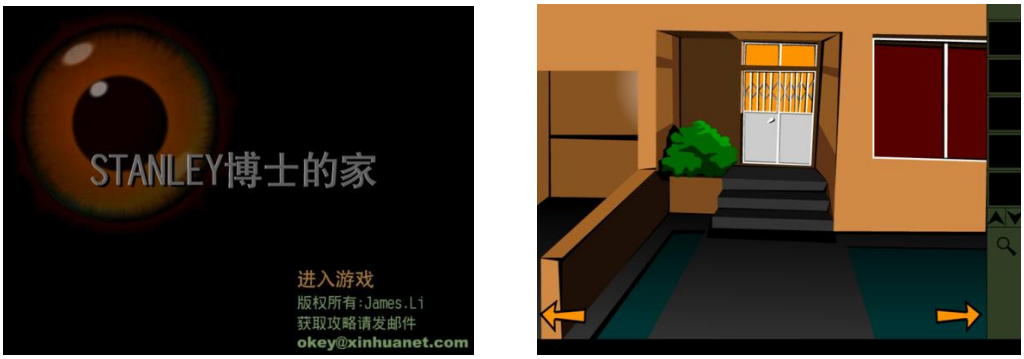
(2) 游戏操作简单，但动作丰富:仅需鼠标点击即可操作游戏，无需玩家记忆大量操作按键，容易上手;游戏中查看物品、使用物品、组合物品、切换场景、 触发事件等等动作效果是丰富充实的，多条剧情设计，能够带给玩家无穷的乐趣。

(3) 解密类游戏不像动作射击类游戏等包含大量动画场面，难度相比马里奥较为简单，但是相比“2048”等游戏较难，适合作为刚接触Haskell的新手期末project。

3.1 参考范例

近年来，比较经典的密室解密游戏有:“胡侦探”、“逃出 13 道黄门”、 “逃出深红色房间(中文版)”、“Stanley 博士的家 1、2”等。考虑到“胡侦探”的画面较为惊悚，而“逃出房间”等游戏的场景少且单一，因此我们决定参考“Stanley 博士的家”设计我们的密室解密游戏。

“Stanley 博士的家 1”，如图 1（a）所示，是李鹏(James Li)于 2005 年发布的一款基于 Flash 开发的解密游戏，游戏出台后受到热玩与广泛好评，目前已更新至第 3 版。游戏界面如图 1（b）所示，可分三个区域:左上角为游戏场景，鼠标点击可触发事件；右边为物品栏，可显示玩家所获得的全部物品，并进行查看(点击放大镜)；下面为对话提示栏，游戏过程中会适时为玩家进行提示和串联剧情。



(a) (b)

图 1.Stanley 博士的家 1 游戏示意

原游戏为单一剧情: Stanley 博士找 James 侦探来家叙事，但 James 侦探却在博士的家中发现了很多异样的事(如物品散乱，被人击晕，门锁破坏，狗被放出等)，甚至有人跌落天井。终于，James 侦探在二楼阳台发现了一个虚弱的老人。James 侦探以为老人就是博士，于是救起了他并帮他从房间暗格中取出了博士的秘密文件，但最终却发现原来老人正是袭击自己的人，而跌落的人才是 Stanley 博士!游戏在操作过程中除了基础的切换场景、使用和查看物品外，还能够组合物品，以及触发剧情动画。

游戏场景丰富但不琐碎是本游戏的一大特点。从房屋构造来看，博士的家不仅包含房屋外围，还包括一楼、夹层、二楼、暗格等等丰富的场景，可供玩家探索。每个房间中的触发事件具体而有逻辑，不像其他一些解谜游戏，物品隐藏的非常细微与琐碎。

综上，我们决定基于“Stanley 博士的家 1”设计我们的密室解密游戏。我们在“博士的家 •增强版”中，修改了原游戏的剧情，丰富了原游戏的场景和线索，去除了原游戏中冗余的动画。下面来详细介绍新游戏的设计内容。

3.2 剧情设计

原游戏仅含一条单一的剧情，虽结局出人意料但令玩家没有成就感(秘密文件最终还是被人偷走了)。因此，我们决定修改游戏剧情，增加两条剧情线路:一条为成功剧情，玩家保护了秘密文件，击倒了窃贼，并救出了博士;另一条为失败剧情，玩家找到了秘密文件，但遭窃贼偷袭，失去了秘密文件。游戏的开始剧情如图 2 所示:博士近日研发出了新成果，遭到窃贼的觊觎，你接到博士的信后，还没赶到博士的家中，博士的家就已经被窃贼光顾了。幸好，博士将秘密文件藏在了暗格中，窃贼将家翻了一个底朝天，也没能找到，而就在此时，你来到了博士的家。

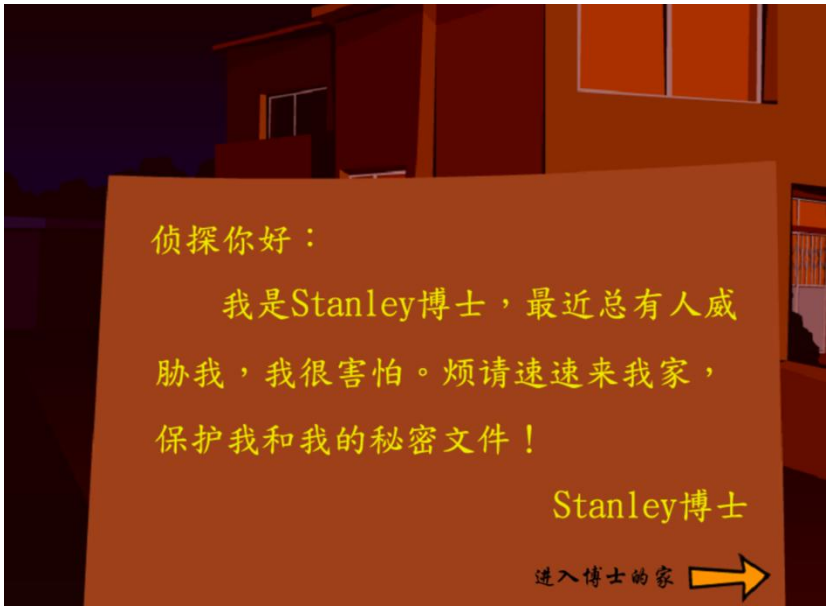


图 2.游戏开始界面

成功剧情：你到达博士的家后，叫门门不应，且发现房间钥匙等物品散落，狗被放出来了，博士也不见了。你感到事情已经变遭，身边隐藏着歹人。当你正盘算着怎么办时，你发现了博士留下的密码线索。你触发了暗格的隐藏密码，身后的窃贼被安保机制击倒。你关闭暗格，在二楼房间中救出了虚弱的博士，并打电话报了警。在你的帮助下，危机得以化解。

失败剧情：你在触发暗格密码后，获得了秘密文件。然后就在这时，藏匿在房中的窃贼将你击晕，抢走了你手中的秘密文件。清醒后，你为自己的失察懊悔不已！

3.3 界面设计

3.3.1 游戏开始与结束

游戏的开始和结束界面的背景图截取了原游戏中的画面，但按键和文字说明等均为笔者原创。

作为一个Flash小游戏，原游戏的开始界面仅有“进入游戏”这一个选项，虽然设计简洁，但损失了读取上次游戏进度等重要功能。我们所设计的新开始界面如图 3所示。可以看到，新界面多了“读取存档”和“退出游戏”两个新按钮。如果玩家点击“读取存档”，则可以立即从上次存档的游戏状态开始继续玩，避免从头开始探索；如果玩家点击“退出游戏”，则可以起到与直接点击窗口“叉号”一样的作用——关闭游戏界面。另外，开始界面的上方（含窗口）注明了游戏的名称，右下角注明了游戏的3位开发者。

图 3 游戏开始界面

如果你直接点击“新游戏”，那么你将从头开始游戏（这并不会抹杀已有的存档文件），你将先看到图 3所示的博士给你的说明初始剧情的信，点击右下角的箭头，你将正式进入游戏主界面，开始你的解谜之旅。

上节提到，我们所设计的新游戏包含双线剧情，因此相对应的也有两种游戏结束界面。如图 4所示，这两种结束界面分别意味着两个游戏结局，左图为成功结局，右图为失败结局。无论是哪一种结局，只要玩家能够达到结束界面，就说明他已经成功解决了本游戏的绝大部分关卡了，只不过在最后一步失败了而已。

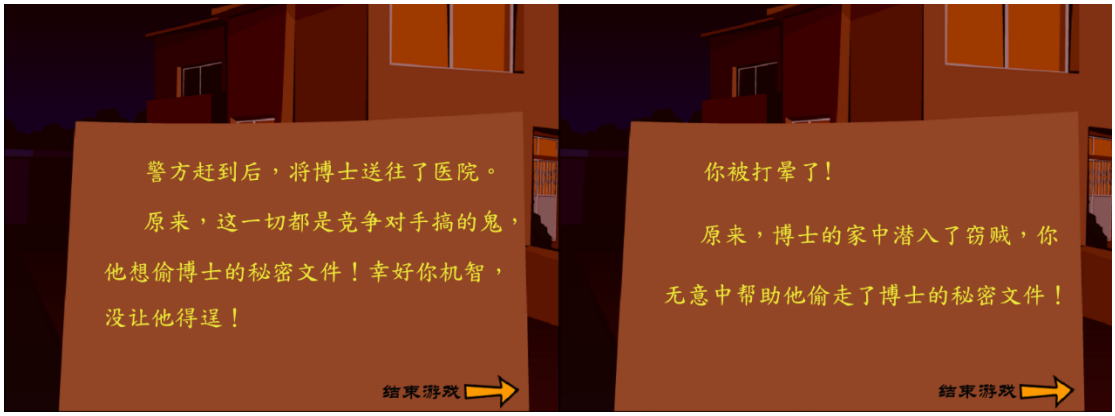


图 4 游戏结束界面（左图成功结局，右图失败结局）

对于图 4所示的结束界面，当玩家点击右下角的“结束游戏”箭头后，就会回到游戏的主菜单，即游戏的开始界面。

3.3.2 游戏场景

游戏场景的界面位置与原游戏一致，如图 5所示，占据窗口左上角的大部分位置。游戏场景采用第一人称视角，模拟玩家视野所关注到的地方，包括观察房间和观察物品。场景当中有箭头、门、密码锁等各种各样的触发域，玩家通过尝试点击场景中的触发域，以切换位置、转换视角或触发剧情，从而不断推进解谜剧情的发展。

本游戏的绝大多数游戏场景均截取自原游戏，少量游戏场景图片（如博士晕倒）等为笔者利用PS等软件绘制而成。

3.3.3 物品栏

物品栏的界面位置与原游戏一致，如图 5所示，占据窗口右边的矩形区域。物品栏的设计也与原游戏基本一致，从上到下依次是五个可显示的物品区域（可叠加显示黄色物品选框）、显示物品的上下翻页按钮、以及查看物品大图（并可进一步翻看物品）的放大镜。使用物品有两种方式：一种方式是点击任意一个物品区域，当出现黄色物品选框时，意味着玩家可以在场景中使用这个物品；另一种方式是点击放大镜，在场景中查看物品大图，以进一步展开、翻看物品细节。

本游戏的绝大多数物品均截取自原游戏，少量新物品（如万能钥匙）等为笔者利用PS等软件绘制而成。

图 5 游戏过程中的界面

3.3.4 存档与返回

在游戏过程中，玩家很有可能需要临时退出游戏。为了使玩家可以在下一次启动游戏时可以接着上次的继续玩，我们在物品栏的最下方新设置的两个按钮“存档”和“返回”。当玩家在游戏过程中点击“存档”按钮，当前的主游戏状态（不含查看物品等状态）会被记录在存档文件“record_[…].txt”中，同时游戏界面会提示你是否存档成功。接下来，玩家可以点击“返回”按钮回到游戏开始界面，或者直接关闭游戏窗口。

存档成功后，玩家就可以在下次进入游戏时，点击游戏开始界面中的“读取存档”，继续自己上次的解谜旅程。

3.3.5 对话提示

对话提示栏的界面位置与原游戏一致，如图 5所示，占据窗口下方的狭长矩形区域。当玩家尝试点击触发域（成功或失败）或查看物品时，对话提示栏会显示出相应的对话或提示信息，以提示玩家具体的物品信息或暗示所需的操作等。本游戏的全部对话提示文本均由笔者原创。

图 6 存档成功提示

3.4 游戏功能

不同的解谜游戏具有不同的功能设计。本游戏在借鉴原游戏操作的基础上，实现了下面几节所述的几个核心的游戏操作功能。

3.4.1 切换场景

通过切换游戏场景，可以模拟玩家的走动和视野，玩家要想切换场景有如下几种途径：

1) 点击场景中的箭头：我们在游戏场景中的边角处设置有明显的橘色箭头，方便玩家通过点击箭头以转向不同的房间位置。
2) 成功点击其他触发域：解谜游戏的精髓在于很多无明显标记[①]的触发域，如跳出窗户、上房顶、进入房间等。当玩家具备一定条件（如到达一定游戏阶段或选中有效物品）并点中相关触发域后，游戏场景将会自动进行切换。
3) 翻看物品：当玩家选中物品并点击放大镜后，其主视角场景将出现物品大图。物品大图并不仅仅是用来进行一次性查看的，玩家通过进一步点击物品可以切换到更多的物品细节场景，查看物品更多的隐藏信息。

3.4.2 触发事件

玩家在解密游戏过程中，结合自己的思考，可以触发多种事件。只有玩家不断的触发游戏事件，才能不断的接近游戏的结局。玩家触发事件的方式就是在满足一定的触发条件下，点击游戏场景中相应的触发域。这些触发条件包括：

1) 点中场景中特定的触发域范围。如果玩家不小心点在了触发域的周围，则游戏认为你没有点中触发域，不会触发相应的事件。
2) 达到一定游戏阶段。比如卧室中的电话，只有当博士在最后告知你要打电话时，你才能去卧室打电话；你提前点击打电话是没有任何效果的。
3) 选中所需物品或备用物品。例如，博士的家中有很多个房间，每个房间门都有和自己颜色匹配的房间钥匙，只有玩家拥有了对应的钥匙，并在物品栏中选中钥匙（出现黄色选框）后，才能打开房门。另外，当你最后获得了七彩的万能钥匙后，它可以作为房门的备用物品，打开任意一道房门。

当玩家触发事件成功（或失败）后，游戏会有相应的更新，这些更新包括但不限于：

1) 切换至新场景。
2) 更新本场景的触发域信息。例如，在你拾取了一个角落的钥匙后，“查看

[①] 无明显标记和无明显图片是两回事。有很多解谜游戏故意设置很细小的触发域，让玩家劳神费眼；而我们的游戏重点在于解谜，几乎没有设置特别细小的触发域。

这一角落”的触发域会被删除，你下次再想查看这个角落就不行了——因为这没有必要了。

3) 更新其他场景的触发域信息。例如，当你在狗舍把狗迷晕后，原切换到狗舍场景的两个触发域箭头和厨房门的信息就被更新了，当你再次点击它们时，你看到的就会是狗被迷晕后的场景，而不会是迷晕前的旧场景。

4) 增删物品。例如，当你拾取钥匙后，你就拥有了这把钥匙；当你使用肉后，肉这个物品就没了。

5) 组合物品。例如，盘子上放肉，肉上面撒酒，最后盘子、肉、酒这三个物品就组合在了一起，构成了一个物品——一盘浸有烈酒的肉。

6) 更新对话提示。

3.4.3 查看和使用物品

本游戏有关物品的功能存在两个区域。

最主要的区域就是右侧的物品栏，物品栏的最上方有五个物品显示区域，可以显示你所拥有的全部物品中的五个物品。如果你想要查看未显示的其他物品，可以通过点击上下翻页箭头以切换显示区域中的物品。如果你想对某物品进行进一步的查看和使用，你可以点击物品所在的显示区域，当该区域出现黄色的外边框时，代表这个物品被你选中。只有先选中某一物品，才能使用它进行相应的操作。

另一个物品相关的区域就是主场景的中心区域，如图 7所示，我们把它称为当前场景，当主场景和当前场景不同时，主场景作为背景，当前场景为前景，构成叠加的游戏界面。在你选中某一物品之后，你可以通过点击放大镜，从而调出该物品的大图，显示在主场景的中心，从而可查看其细节或进行针对物品的操作。

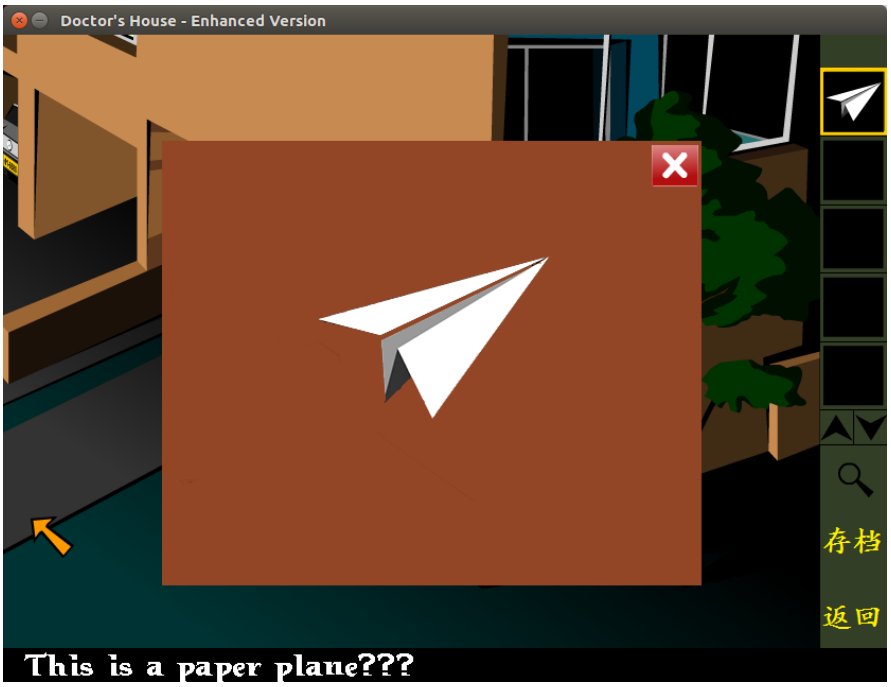


图 7 主场景的中心显示你所查看的物品大图

在当前场景为物品大图中，你所能进行的操作包括但不限于：

1) 翻看物品。通过点击物品大图中的物品，你可以进一步翻看其细节内容。
2) 组合物品。当你在查看盘子时，你可以把物品栏中的肉放在盘子上组合成新物品。
3) 关闭查看。你可以通过点击物品大图右上角的叉号，以关闭当前物品的查看。

另外，最常用的使用物品的方式就是在选中物品后，点击场景中相应的触发域以触发事件，这一点在上一节已经介绍过了。

3.4.4 对话和提示

在3.3.5节提到，当玩家尝试点击触发域（成功或失败）或查看物品时，对话提示栏会显示出相应的对话或提示信息，以提示玩家具体的物品信息或暗示所需的操作等。对话和提示的出现情景分为以下几种：

1) 成功触发事件：提示你所触发的事件信息，反映玩家的内心活动。
2) 触发事件失败：提示你所缺少的触发所需的物品或其他条件。
3) 点击人物：当你点击博士时，博士会与你对话互动。

3.4.5 背景音乐

我们为游戏精心挑选了符合原游戏氛围的背景音乐bgm.wav。我们设置本背景音乐随游戏启动，循环播放。

3.4.6 存档与读档

我们通过四个“record_[…].txt”存档文件，记录了玩家所存档的游戏进度。玩家在游戏过程中，可以随时点击右下方的“存档”按钮，将当前的游戏进度更新到存档文件中。我们在游戏的开始界面设置了“读取存档”按钮，玩家可以通过点击该按钮，从存档文件中加载上次记录的游戏进度。

由于游戏记录存储在存档文件中，而非保存在运行时参数中，因此玩家关闭游戏后并不会丢失先前存档的游戏进度。另外，玩家可以通过手动另存多份存档文件，以记录多种游戏进度。

另外，为了防止玩家通过修改存档文件以改写自己的游戏进度，我们初步采取了以下几种保护手段：

1) 隐藏源代码：我们将游戏源码编译、链接成为可执行文件后，可以隐藏游戏的设计细节，使玩家无法修改游戏，也难以读懂存档文件的含义。
2) 不记录游戏信息：存档文件仅记录玩家当前的状态，不记录游戏所期望的正确状态。以密码锁为例，存档文件中只记录玩家已经输过的密码，并不记录游戏所期望的正确密码，因此玩家也根本不知道应该修改密码为何值。

3.5 房屋布局

游戏房屋由以下四部分组成：

（1）房屋外围：游戏开始时玩家看到的是房门的外景。左转是厨房后门，包含狗舍；右转的两个场景均为房屋的外部，可看到房屋的各个轮廓。

（2）一楼：一楼包含卫生间，客厅和卧室，其中卫生间的门是不上锁的。卧室内部可进一步进入浴室， 卧室旁边是房屋的楼梯。

（3）夹层：楼梯上一小半可看到两个房门，一个是餐厅，餐厅外是天井；另一个是厨房，包含储藏食品的多个场景。

（4）房屋二楼：楼梯上到顶后进入房屋二楼，首先看到的是二楼的卫生间。进一步往里走，可以看到 4 个房间：杂物间是一个小房间；工作室包含阳台，玩家可以通过梯子进入到房顶；绿色的密室关押着博士；橘色的门后是二楼的卧

室，包含密室暗格。

3.6 游戏物品

游戏中，通过获得不同的物品可以解锁不同的游戏场景，获得不同的游戏线索。游戏通关所涉及到的全部物品如图 8 所示：

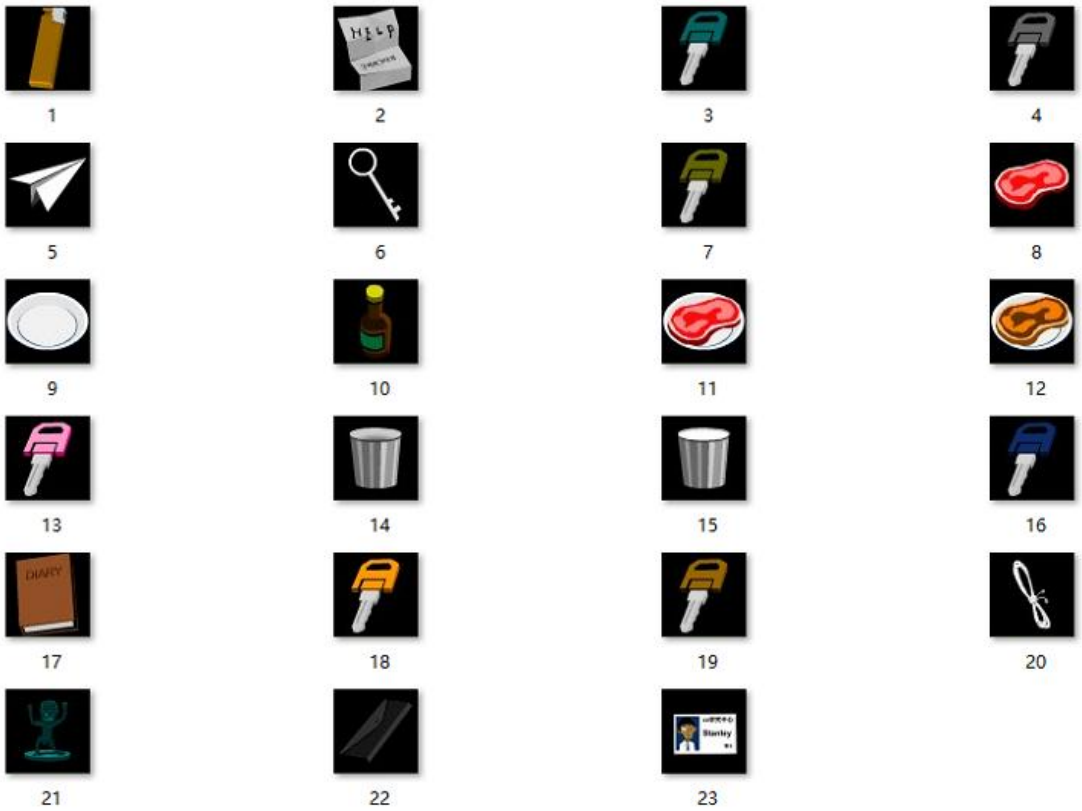


图 8 游戏物品

3.7 游戏流程

基于前述内容，整个游戏的通关流程设计如下：

（1）房子外围：取得铜像、纸飞机；打开纸飞机，获得房屋密码，进入房屋。

（2）一楼卫生间获得打火机，二楼卫生间获得客厅钥匙。

（3）在客厅获得一楼卧室钥匙，一楼卧室获得明信片、浴室钥匙；进入浴室，获得厨房钥匙。

（4）进厨房点蜡烛，组合盘子、肉、烈酒，迷晕房屋外围的狗，获得餐厅钥匙。

（5）进餐厅取水杯接水，到户外获得工作室钥匙。

（6）进工作室获得书及末页的铜像凹槽密码，左转阳台、右转屋顶得到杂

物间和二楼卧室钥匙。

（7）进杂货间取电话线，进二楼卧室的卫生间。

（8）取下镜子，输入密码出现铜像凹槽，插入铜像，出现钥匙插盘。

（9）关灯，依据彩虹光芒，在钥匙插盘上正确插入 7 把钥匙，出卫生间，看到暗格密码锁。

（10）查看明信片背面的密码，若输入正确密码则出现隐藏文件，游戏失败；若输入倒转的隐藏密码，则进入下一步。

（11）从倒地的窃贼身边取走万能钥匙，在二楼卧室旁边的房间中找到博士，给博士喝水。

（12）去一楼卧室给电话插上电话线，打电话报警，游戏成功。

第4章 游戏开发

4.1 Haskell 的库的使用

在本节中，我们将讲述本游戏设计中所涉及到的 Haskell 的库，其中包括 SDL2（游戏框架），SDL2-ttf（基于 SDL2 的文本库），SDL2-mixer（基于 SDL2 的音效库）。

4.1.1 SDL2 库

根据章节 3 中所述，这里我们要实现的游戏是一个静态界面的。在 Haskell 中实现 GUI 界面需要借助一些额外的包，我们首先要明确安装的包需要哪些功能,进而再进行选取合适的包进行安装。根据前述对游戏的描述，其最主要的功能是当点击窗口某处时，我们要判断鼠标点击的位置是否在矩形的触发域中，如果是的话,我们要触发事件(例如切换到下一场景,拾取到物品,使用物品等等)。综上,我们所需的包需要具有的功能总结如下:(1)在屏幕上显示一个窗口,(2)能够监控鼠标事件。获得鼠标位置,(3)能够在窗口画图，且能够将窗口分成不同的块，分块画图,(4)能够方便后续优化，例如加入转场动画、背景音乐、文本等。

经大量查阅资料，我们发现经常被用于游戏开发的有 SDL2（Haskell 中以 import SDL 导入）和 SDL（import Graphics.UI.SDL 导入）这两个库（注意这两个库极其容易弄混，请务必小心它们的名字!)。我们发现 Haskell 的库文档中对于这两个库的描述过于简略，纵使有每个函数的类型，但关于参数的解释太少，所以我们查阅了 github，找到了一些 demo。通过调试，我们成功跑通了 SDL2 的 demo；另外同时，在 github 上，我们发现 SDL2 可以用于实现马里奥这个游戏，所以我们推断 SDL2 这个库对于我们的游戏设计来说足够。为此我们最后选定使用 SDL2 这一库来进行游戏设计。其安装过程与其他包相同，使用命令：cabal install sdl2 即可;如果报错,请在运行 cabal 命令前,输入 sudo apt-get install libsdl2-dev。

下面我们结合代码，如图 9 所示，讲述 SDL2 的库的使用，一些特性以及实现时的一些 trick。在 main 函数的开头，我们调用 SDL.initialize 进行初始化，其参数是 SDL.InitiVideo，表示我们将基于视频进行初始化。第二步我们调用 SDL.createWindow 创建了一个窗口，它的第一个参数是一个字符串，表示窗口的名字，这里名字为“SDL Tutorial”;第三个参数是 SDL.defaultWindow 这样一个数据结构（注意这里 SDL.defaultWindow 是一个 data，它的 windowInitialSize 字段被 screenWidth 和 screenHeight 决定），表示我们将用一个宽为 screenWidth，高

为 screenHeight 的 defaultWindow 进行初始化。而后 showWindow 展示着一窗口。接着，getWindowSurface 得到 window 的 surface，以便于后面将图片画到 surface 上去。下一步作为 demo，我们只导入了两张图片即 first.bmp 和 second.bmp。

值得注意是为 xout 和 xout2 赋值这两行着实花费了我们不少时间。主要原因是 getDataFileName 这一函数。图 10 展示了我们在 github 上找到的 SDL2 的 demo 代码的片段。我们在写代码时参考这一片段时，我们发现 getDataFileName 这一函数时来自于一个叫做 Path_sdl2 的包，然而我们仔细查阅了整个 demo 的源代码也没找到名字为 Path_sdl2 的文件，我们随之以为这是一个外部依赖包，尝试了 cabal install Paths_sdl2，然而发现根本没有这一个包。在尝试了前述两种方式后均失败后，我们查阅了资料，了解到在用 cabal 这个命令执行一个名字为 x.hs 的脚本时，会自动产生一个 Paths_x.hs 的脚本，里面会定义好 getDataFileName 这一函数。在我们同样尝试使用 cabal 命令来运行脚本时，这一路径图仍然一直加载错误。最后我们想到另一种办法，就是自己写一个名为 getDataFileName 的函数。为此我们要了解 getDataFileName 的作用。

```
main :: IO ()
main = do
  SDL.initialize [SDL.InitVideo]
  window <- SDL.createWindow "SDL Tutorial" SDL.defaultWindow { SDL.windowInitialSize = V2 screenWidth screenHeight }
  SDL.showWindow window
  screenSurface <- SDL.getWindowSurface window

  xOut <- getDataFileName "figs/first.bmp" >>= SDL.loadBMP
  xOut2 <- getDataFileName "figs/second.bmp" >>= SDL.loadBMP
  SDL.surfaceBlit xOut Nothing screenSurface Nothing
  let
    loop mouse buttons = do
      --events  <- SDL.pollEvents
      events <- map SDL.eventPayload <$> ((<$>) (\a -> a:[]) SDL.waitEvent)
      mousePos <- SDL.getAbsoluteMouseLocation

      let current_state = getAny $ foldMap (\case
                SDL.MouseButtonEvent e -> Any $ SDL.mouseButtonEventMotion e == SDL.Pressed
                otherwise -> Any False) events

      let quit = SDL.QuitEvent `elem` events
      let mouse' = Mouse {posM = mousePos, state = current_state}
      let res = map (handleEvent mouse') buttons
      if (head res) == True
         then SDL.surfaceBlit xOut2 Nothing screenSurface Nothing
         else return $  Just (SDL.Rectangle (P (V2 0 0)) (V2 0 0))
      SDL.updateWindowSurface window
      unless (quit || (last res)) (loop mouse' buttons)

  loop (Mouse {posM = (P (V2 0 0)), state = False}) [button1,button2]

  SDL.freeSurface xOut
  SDL.destroyWindow window
  SDL.quit
```

图 9.SDL2 的 demo 代码片段

我们仔细阅读了代码，demo 中这一行所做的事情是给定一个 bmp 图片地址，我们通过 getDataFileName 以及 SDL.loadbmp 后存到 helloWorld 这个变量中。通过 monad 符号<-, >>=，以及 SDL.loadBMP 的类型，我们推断出 getDataFileName 的类型应该为类似于“字符串”到“IO 字符串”。通过反复尝试，我们最后写出了 getDataFileName 的函数类型以及函数体如所示。这里 FilePath 实际上就是 [Char]，是一个 Haskell 已经预定义好的数据类型。这一 getDataFileName 着实花费了我们不少时间。

```haskell
import Paths_sdl2 (getDataFileName)
screenWidth, screenHeight :: CInt
(screenWidth, screenHeight) = (640, 480)
main :: IO ()
main = do
  SDL.initialize [SDL.InitVideo]
  window <- SDL.createWindow "SDL Tutorial" SDL.defaultWindow { SDL.windowInitialSize = V2 screenWidth screenHeight }
  SDL.showWindow window
  screenSurface <- SDL.getWindowSurface window

  helloWorld <- getDataFileName "examples/lazyfoo/hello_world.bmp" >>= SDL.loadBMP
```

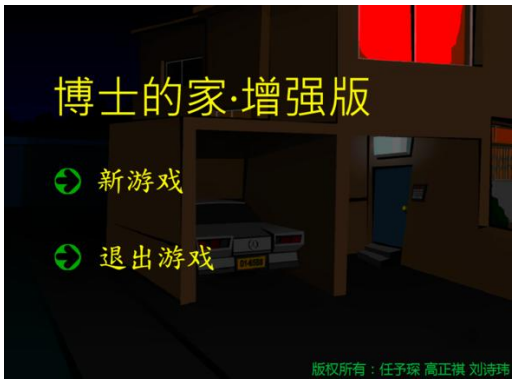
图 10.github 上 SDL2 的 demo

```haskell
getDataFileName :: FilePath -> IO FilePath
getDataFileName = return
```

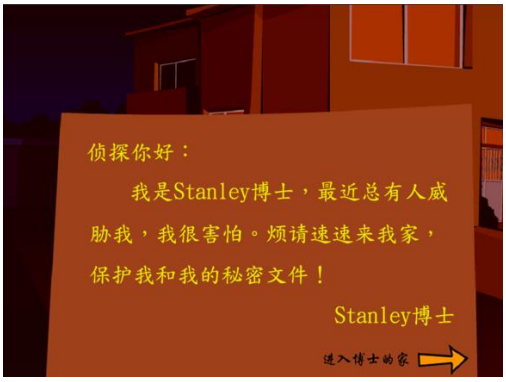
图 11. getDataFileName 的函数体

在完成了这一函数后，我们回到图 9 的代码片段中，我们导入了 first.bmp 和 second.bmp 后，我们先在窗口上显示出 first.bmp。而后在代码中，我们定义了一个函数 loop，loop 有两个参数 Mouse 和 Button（事实上这里我们实现的冗余了，loop 可以不需要参数的，但是这作为 demo 已经足够）。在 loop 的函数体中，我们通过函数 eventpayLoad 拿到事件存放到 events 中，而后我们得到鼠标的位置存放到 mousePos 中，得到鼠标状态（是否按下）存放到 current_state 中。然后我们先判断是否当前事件时是退出，这一消息由变量 quit 表示，变量 quit 是一个 bool 型，True 表示是退出事件，False 表示不是退出事件。而后我们用 handleEvent 来判断鼠标位置和所有按钮 buttons 之间的关系，这可以通过函数 map 来实现，即我们通过 map，遍历 buttons 这个列表，逐一判断当前鼠标和按钮的关系，这一结果也为一个列表，我们存放到变量 res 中。在这一 demo 中，res 只有两个元素，因为在 first.bmp 中只有两个按钮，如图 12 所示。一个是“新游戏”一个是“退出游戏”。接下来我们进行判断：如果 res 第一个元素是 True（表明我们点击了新游戏），则我们在 screensurface 上面显示 second.bmp。值得注意的是 if 语句必须要有 then 和 else，而且 then 和 else 中语句类型要相同，我们参考 then 的返回类型，写出了实际上什么也没做的 else 语句。最后我们更新 surface，将 second.bmp 真正显示到窗口上。最后，如果没有退出（退出的判定是：为退出事件即点击窗口左上角的关闭，或者点击第二个按钮即“退出游戏”按钮），我们递归调用 loop。

在完成了 loop 的定义后，我们在 main 函数中，调用 loop 函数，进入循环。最后在 main 函数末尾，我们释放所有资源。



(a)



(b)

图 12.利用 SDL2 实现的游戏 demo 示意。我们首先在窗口显示图（a），当我们点击图（a）中的“新游戏”时，会跳转到图片（b），点击“退出游戏”会结束。

4.1.2 SDL2-mixer 库

在 Hackage 的 SDL2 库中已经包含了播放音乐的相关函数。但是，SDL 原有的函数只支持播放.wav 格式的音乐，且不区分音效与背景音乐。因此我们使用了 SDL 更为强大的音频扩展库 SDL2_MIXER。

支持 Haskell 的 SDL_MIXER 库源自：https://github.com/tempname11/sdl2-mixer。SDL_MIXER 将音频分为两类：音效(foreground music)，背景音乐(background music)。

背景音乐：每次只能播放一首背景音乐，直到到达循环次数或强制关闭。若在播放背景音乐的时候播放另一首背景音乐，前者的所有资源会先被释放，然后创建相关资源播放新的背景资源。若同时播放多首背景音乐，只有最后加载的背景音乐才会被播放。

音效：可以同时播放多个音效。音效可以和背景音乐同时存在。如在游戏中背景音乐可以一直存在，而触发某一动作后会另有特殊音效。

背景音乐：播放流程见图 13。

```
SDL.initialize [SDL.InitAudio]                                          → 初始SDL_MIXER
Mix.openAudio 44100 Mix.AUDIO_S16LSB 2 4096                             → 打开播放设备
Mix.allocateChannels 16                                                 → 设置混音通道(mix channel)数
music <- withCString "./bgm.mp3" $ \cstr -> Mix.loadMUS cstr            → 加载音频文件
temp <- Mix.playMusic music (-1)                                        → 播放音频
assert $ temp /= -1                                                     → 是否正常播放
threadDelay 100000                                                      → 等待音频播放完成
```

图 13 背景音乐播放流程

Mix.openAudio frequency, format, channels, chunksize。Frequency 表示输出采样频率，普通 CD 的频率为 44.1KHz。SDL2_MIXER 建议设计游戏时使用 22.05KHz，因为较高的采样频率会造成更高的 CPU 占用率。Format 为输出音频数据在缓冲中的存储格式。Channels 为输出声道数，1 为单声道，2 为双声道。Chunksize 为音频数据缓冲区大小，若设置过小，音乐的某些片段在播放时会被

跳过，若设置过大，游戏开始后一段时间才有背景音乐播放。

Mix.allocateChannels channelsNum。ChannelsNum 为播放音乐使用的通道数(Number of mixed channels)，这里通道与 Mix.openaudio 中的输出声道是两个概念。但并没有查到声道(channel)与通道(mixed channel)的区别。需要注意的是，在 Mix.openAudio 在打开播放设备时，已经默认使用 8 个通道播放音频。

Mix.playMusic music loops。Music 为加载的音频文件。Loops 表示循环播放次数，-1 表示一直重复循环。

threadDelay 是手动施加的一段延时。这一段延时是必须的，若不加这段延时，该参考代码在运行后会很快结束，听不到任何声音。在 Mix.playMusic 播放音乐后，音乐播放在后台完成。此时程序仍会继续运行，当程序运行完成后，所有资源会被释放，声音便不能再播放。

音效：播放流程见图 14。

```
SDL.initialize [SDL.InitAudio]                                  → 初始SDL_MIXER
Mix.openAudio 44100 Mix.AUDIO_S16LSB 2 4096                     → 打开播放设备
Mix.allocateChannels 16                                         → 设置混音通道(mix channel)数
sound <- withCString "./bgm.wav" $ \cstr -> Mix.loadWav cstr    → 加载音频文件
channel <- Mix.playChannel (-1) sound 0                         → 播放音频
assert $ channel /= -1                                          → 是否正常播放
threadDelay 100000                                              → 等待音频播放完成
```

图 14 音效播放流程

Mix.loadWav。加载.wav 格式的音效文件。实际上该函数也可以用于加载其他格式的音频文件，如.mp3 等。

Mix.playChannel channel_index, sound, loop。Channel_index 为播放音效所用通道索引，-1 表示自动选择第一个空闲的通道（索引最小的空闲通道）。Sound 为加载的音效数据。Loop 为播放次数，0 为播放一次。我们可以为每一个音效通道设置不同的效果，如音量、开始、暂停、结束等。

SDL2_MIXER 背景音乐播放与主程序其他代码、多个音效之间线程的调度对于使用者是透明的。

本游戏使用的 SDL2_MIXER 库通过调用 C++ API 来支持 Haskell 语言。在 Hackage 查阅其他作者编写的 SDL2_MIXER 库，发现方法大同小异。这里特别介绍一下如何在 Haskell 中调用其他语言编写好的函数。

Haskell 使用外部函数接口(Foreign Function Interface FFI)来与其他语言交互。Haskell 既可以调用其他语言的函数，也可以被其他语言调用。在使用 FFI 时，需要在文件开始加：

{-# LANGUAGE ForeignFunctionInterface #-}。

在调用其他语言的函数时，只需要将函数原型转换为 Haskell 中等价的原型。图 15 为使用 ccall 调用 C/C++函数的示例。

```
foreign import ccall "SDL_mixer.h Mix_LoadMUS" loadMus' :: CString -> IO (Ptr Music)
```

图 15 FFI 示例

需要注意的是，在原型转换的时候，外部语言函数的数据类型在 Haskell 中往往不支持。此时需要导入 Foreign 模块。经常使用的 Foreign 模块支持的类型总结在表 1。

模块	类型
Foreign.C.Types	CInt ， CDouble ， CUChar
Foreign.C.String	CString
Foreign.C.Ptr	指针

表 1 Foreign 模块支持的 C 语言类型

FFI 之所以可以支持外部语言函数调用，是因为任何语言被编译为机械码后，函数变成为机械码中的一个标志。调用函数时只需要把参数放到内存中正确的位置，函数只需要在相应内存位置上读数据即可。

4.1.3 SDL2-TTF 库

Hackage 的 SDL2 库不支持打印文本文字，因此我们使用了 SDL2 的扩展库 SDL2_TTF 以支持文字显示。游戏所用 SDL_TTF 库源自：https://github.com/Haskell-game/sdl2-ttf。文本显示流程见图 6。

```
white :: SDL.Font.Color                                                    ——→ 字体颜色
white = SDL.V4 255 255 255 0

Font.initialize                                                            ——→ 初始TTF
font <- Font.load "films.Hellbound.ttf" 30                                 ——→ 加载字库，设置字号
text <- Font.solid font red "There is the dog"                             ——→ 加载文本
Font.free font                                                             ——→ 释放字库
SDL.surfaceBlit text Nothing screenSurface (Just (P (V2 20 575)))          ——→ 文本显示
```

图 16 文本显示流程

Font.load text.ttf size。载入预先下载的.ttf 字库，并设置字体大小。

Font.solid font color texture。Solid 指字符为实心。载入的文本内容此时已经被转换为图片形式。

SDL.surfaceBlit pic pic_pos win win_pos。Pic 为将要显示的图片。pic_pos 为 Pic 上的矩形区域，若无空则指整张图片。Win 为显示窗口。Win_pos 为显示窗口显示图片的位置，只需要左上角坐标(xmin, ymin)即可。使用前述的 FFI，我们实现了 SDL_TTF 库基本函数的 Haskell 版本。具体见 TTF.hs。

4.2 游戏资源的准备

原游戏显示界面可划分为图 17 各区域。需要注意的是查看物品区域、对话框区域、收集物品排列顺序、主场景区域是可以任意组合的。因此不能截取整个游戏界面按游戏顺序依次显示。因为我们很难遍历所有可能存在的显示组合。其次，即使可以遍历，游戏素材存储也需占用大量的空间。

我们将游戏场景中各独立区域依次截图存储，作为本游戏设计时的素材。游戏素材的获取与整理是一项工作量巨大又枯燥的工作，为了减少手动截取图片所用时间，我们使用 OpenCV 编写了几个图片批处理的脚本，总结在表 2 中。

图 17 原游戏显示区域划分

函数名	函数功能
screenshot	鼠标控制截图区域并保存图片；打印图片左上角坐标并保存
rename	某些图片漏截后，重新修改文件夹下所有图片名称，保证截取顺序不变
resize	统一手动截取的图片大小

表 2 游戏素材批处理函数

即使有这些自动化脚本存在，由于各游戏素材之间存在逻辑关系同时为了满足数据结构定义，仍花费了大量时间初始化各类数据。最终保存在 Diag.hs，Images.hs，Image_pos.hs，Items.hs，Triggers.hs 中。

4.3 游戏资源的 Data 数据

游戏在运行过程中，需要有一系列的Data数据记录当前游戏的进度和状态。

由于Haskell没有全局变量，因此游戏的运行过程就是不断循环调用游戏引擎中的多层次函数，入口参数是上述Data数据，返回值是更新后的Data数据（有些Data数据是恒定值，不会被更新）。这一过程在图 18中显示地很清晰。

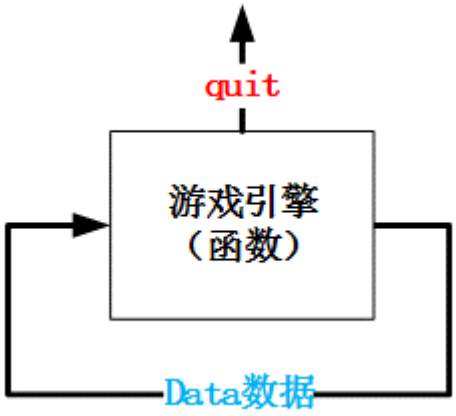


图 18 游戏运行流程简图

因此，在介绍具体的游戏引擎函数设计代码之前，我将先在本章介绍Data数据的具体定义。

4.3.1 界面：Interface

“data Interface”为“界面”类型，它记录了当前游戏状态和玩家状态，具体定义如图 19所示。

```
data Interface = Interface {  dialogue          :: Int
                            , possessed_items   :: [Int]
                            , selected_item     :: Int
                            , showed_first_item :: Int
                            , main_scene        :: Int
                            , current_scene     :: Int
                           }
```

图 19 定义 data Interface

其中：

- dialogue记录当前的对话栏文字索引值；
- possessed_items记录玩家当前拥有的全部物品索引值；
- selected_item记录玩家当前选中的某一物品在possessed_items中的索引值；
- showed_first_item记录物品栏五个显示区域中第一个区域所显示的物品在possessed_items中的索引值；

- main_scene记录游戏主场景的索引值，current_scene记录游戏当前场景的索引值。

由于各游戏资源已经分别存储在相应的列表中了，因此Interface中的各字段只需要记录相应游戏资源在其列表中的**索引值**即可——故大量字段均为**Int类型**。例如，记所有物品的资源列表为all_items，则玩家当前所选中的物品实际为all_items !! (possessed_items !! selected_item)。另外，如果索引值为“-1”，则代表无相应的资源。

由于五个物品显示区域是顺序排列的，因此只需要记录第一个区域所显示的物品showed_first_item，即可推断出后续4个（如果有的话）需要显示的物品。

游戏场景分为主场景main_scene和当前场景current_scene。主场景代表玩家所处的房间位置，即“背景场景”；当前场景代表玩家的视野，即“目光场景”。当玩家在房间里走动时，背景场景和目光场景是一致的；而当玩家查看物品时，背景场景不变，而目光场景变更为所查看物品的大图。这样设置的目的是，当玩家退出当前场景后，游戏引擎可以立即知道需要返回到哪个主场景中。

综上我们可以看到，Interface所记录的信息与游戏界面需要显示的图像信息密切相关。后续会看到，游戏引擎的绘图函数就是根据Interface所记录的信息依次绘制游戏窗口界面的。

data Interface类型定义在“Data_define.hs”文件中，其初始值init_interface和相关的界面参数定义在“Interface.hs”文件中。

4.3.2 场景：Scene

“data Scene”为“场景”类型，它记录了某一游戏场景所包含的信息，具体定义如图 20所示。

```
data Scene = Scene {  background :: Int
                   , triggers   :: [Int]
                   }
```

图 20 定义 data Scene

“场景”即为3.3.2节所介绍的游戏场景。在图 20的定义中：background记录场景图片的索引值，triggers记录场景中所包含的全部触发域的索引值。

本游戏共包含93个场景，它们全部初始化记录在变量“init_scenes”中。在游戏运行过程中，部分场景的信息可能会有更新和修改，实时的场景信息变量通常命名为“all_scenes”。

data Scene类型定义在“Data_define.hs”文件中，其初始值init_scenes和相关的

界面参数定义在“Scenes.hs”文件中。

4.3.3 触发域：Trigger

“data Trigger”为“触发域”类型，它记录了某一触发域的本体信息和更新信息，具体定义如图 21所示。

```
data Trigger = Trigger {  picture :: Int
                        , trigger_vertex :: (Point V2 CInt)
                        , trigger_size :: V2 CInt
                        , trigger_type :: Triggertype
                        , needed_item :: Int
                        , backup_item :: Int
                        , trigger_data :: Int

                        , relation_trigger :: Int
                        , relation_trigger_new_scene :: Int
                        , relation_trigger_new_picture :: Int
                        , triggerself_new_picture :: Int
                        , fail_dialogue :: Int
                        , new_dialogue :: Int
                        , new_scene :: Int
                        , add_item :: Int
                        , delete_item :: Int
                        , selected_item_delete :: Bool
                        , change_main_scene :: Bool
                        , relation_scene :: Int
                        , relation_scene_trigger_delete :: Int
                        , relation_scene_trigger_add :: Int
                        , triggerself_delete :: Bool
                       }  deriving (Eq)
```

图 21 定义 data Trigger

“触发域”在3.4.2节有所介绍，当玩家在符合一定条件的前提下点击场景中的触发域后，游戏引擎会根据触发域中的更新信息更新游戏进度、玩家状态和游戏资源，从而推动解谜游戏的发展。在图 21的定义中，空行之前为触发域的本体信息，记录触发域的区域、图片和类型等基本信息；空行之后为触发域的更新信息，记录触发域被成功或失败触发后游戏进度、玩家状态和游戏资源的更新操作。

触发域的本体信息包括：

- picture记录触发域本身的小图片的索引值；
- trigger_vertex和trigger_size分别代表触发区域的左上顶点坐标和长宽，从而划定触发域的矩形范围；
- trigger_type记录触发域的类型；
- needed_item和backup_item分别代表触发所需的物品索引值和备用物品索引值；

- trigger_data代表触发域所蕴含的数字信息。

其中，触发域的类型“data Triggertype”记录了多种不同的触发域类型，以使游戏引擎进行有针对性的处理。这些触发类型包括：

- 房门密码锁按钮“HOUSE_PASSWORD_BUTTON”
- 铜像密码锁按钮“BRONZE_PASSWORD_BUTTON”
- 保险箱高级密码锁按钮“ADVANCED_PASSWORD_BUTTON”
- 七彩钥匙插盘中的钥匙孔“KEY_HOLE”
- 主菜单返回按钮“MENU”
- 游戏结束按钮“END_GAME”
- 其他普通的触发域“ORDINARY”

trigger_data记录了触发域所蕴含的数字信息，信息含义与上述触发域的类型相关。例如，密码锁按钮的trigger_data记录该按钮所代表的1位十进制数字（0~9），插盘钥匙孔的trigger_data记录该插孔在插盘中的序号（0~6）。

触发域的更新信息包括：

- relation_trigger记录某一相关触发域的索引值，relation_trigger_new_scene更新相关触发域的new_scene值，relation_trigger_new_picture更新相关触发域的picture值；
- triggerself_new_picture更新触发域自己的picture值；
- fail_dialogue记录触发失败后所显示的对话提示索引值，new_dialogue录触发成功后所显示的对话提示索引值；
- new_scene记录触发成功后玩家所到的新场景，change_main_scene记录触发成功后是否将主场景更新为当前场景；
- add_item记录触发成功后玩家所获得的新物品，delete_item记录触发成功后玩家所失去的物品，selected_item_delete记录触发成功后是否删除玩家所使用的物品；
- relation_scene记录某一相关场景的索引值，relation_scene_trigger_delete删除相关场景中所包含的触发域，relation_scene_trigger_add向相关场景中添加触发域；
- triggerself_delete记录触发成功后本触发域是否消失。

本游戏共包含188个触发域，它们全部初始化记录在变量“init_triggers”中。在游戏运行过程中，部分触发域的信息可能会有更新和修改，实时的触发域信息

变量通常命名为“all_triggers”。

data Trigger类型和data Triggertype类型定义在“Data_define.hs”文件中，触发域列表的初始值init_triggers定义在“Triggers.hs”文件中。

4.3.4 物品：Item

“data Item”为“物品”类型，它记录了某一游戏物品所包含的信息，具体定义如图 22所示。

```
data Item = Item {  item_picture :: Int
                  , item_scene :: Int
                  , trigger_picture :: Int
                 }
```

图 22 定义 data Item

“物品”即为玩家可拾取、使用并显示在物品栏中的游戏物品。在图 22的定义中：item_picture记录游戏物品在物品栏中所显示的小图的索引值，item_scene记录查看游戏物品时的（当前）场景的索引值，trigger_picture记录游戏物品被放置在场景中所呈现的图片的索引值。

以某一颜色的钥匙物品为例：它在物品栏的显示区域中的图片索引值为item_picture；当用户点击放大镜查看这一钥匙时，当前场景切换到item_scene，主场景不变，进入钥匙大图的观察视角；当用户把钥匙插入七彩插盘中后，插孔上显示的钥匙图的索引值为trigger_picture。

本游戏共包含22个物品，它们全部记录在常量“all_items”中。在游戏运行过程中，全部物品的信息均不会改变，因此all_items不会作为函数参数在游戏引擎函数中被频繁更新。

data Item类型定义在“Data_define.hs”文件中，其列表值all_items定义在“Items.hs”文件中。

4.3.5 普通密码锁：Password_Lock

“data Password_Lock”为“普通密码锁”类型，它记录了某一普通密码锁所包含的信息，具体定义如图 23所示。

```
data Password_Lock = Password_Lock { correct_password :: [Int]
                                   , input_password   :: [Int]
                                   }
```

图 23 定义 data Password_Lock

“普通密码锁”包含游戏过程中的房门密码锁（见图 5）和铜像密码锁，这两个密码锁仅包含输入正确密码一种开锁可能。在图 23的定义中：correct_password逆序记录各位正确密码，input_password逆序记录玩家输入的各位密码。

之所以密码的各个位是逆序记录的，是因为这样在玩家输入一位密码时，该密码可以直接通过“:”操作符附加在input_password列表的头部。input_password的密码位数和correct_password的密码位数保持一致，用户每向input_password中输入一位密码，input_password就会丢弃其尾部的密码。因此，只有当用户连续输入正确的密码列表后，input_password == correct_password，密码锁才会被判为“打开”。

本游戏共包含2个普通密码锁，它们分别记录在变量house_password_lock（房屋密码锁）和bronze_password_lock（铜像密码锁）中。在游戏运行过程中，密码锁的input_password可能会有更新，但correct_password保持不变。

data Password_Lock类型定义在“Data_define.hs”文件中，其两个变量house_password_lock和bronze_password_lock定义在“Locks.hs”文件中。

4.3.6 高级密码锁：Advanced_Password_Lock

“data Advanced_Password_Lock”为“高级密码锁”类型，它记录了高级密码锁所包含的信息，具体定义如图 24所示。

```
data Advanced_Password_Lock = Advanced_Password_Lock { ad_correct_password :: [Int]
                                                     , ad_hidden_password  :: [Int]
                                                     , ad_input_password   :: [Int]
                                                     }
```

图 24 定义 data Password_Lock

“高级密码锁”即为游戏最后阶段的保险箱密码锁，保险箱密码锁除了包含输入正确密码这种开锁可能外，还包含输入隐藏密码这种开锁可能。在图 24的定义中：ad_correct_password逆序记录各位正确密码，ad_hidden_password逆序记录各位隐藏密码，ad_input_password逆序记录玩家输入的各位密码。

用户完整输入正确密码或隐藏密码后，游戏剧情都会有相应的变化，只不过会导致不同的结局。

本游戏唯一的高级密码锁记录在变量file_password_lock中。在游戏运行过程中，密码锁的ad_input_password可能会有更新，但ad_correct_password和ad_hidden_password保持不变。

data Advanced_Password_Lock类型定义在“Data_define.hs”文件中，其唯一变量file_password_lock定义在“Locks.hs”文件中。

4.3.7 钥匙插盘：Keys_Lock

“data Keys_Lock”为“钥匙插盘”类型，它记录了游戏中七彩钥匙插盘所包含的信息，具体定义如图 25所示。

```
data Keys_Lock = Keys_Lock { correct_keys :: [Int]
                           , current_keys :: [Int]
                           }
```

图 25 定义 data Keys_Lock

“钥匙插盘”即为游戏中包含7个钥匙插孔的钥匙插盘。在图 25的定义中：correct_keys依次记录7个钥匙插孔所应该插入的钥匙物品的索引值，current_keys依次记录7个钥匙插孔当前插入的钥匙物品的索引值。

玩家可以选择往钥匙插盘的各钥匙插孔中插入任一一种颜色的钥匙，也可以随时从中拔出任意一把钥匙换个插孔重新插入。只有当“correct_keys == current_keys”后才会触发下一步的游戏剧情。

本游戏唯一的钥匙插盘记录在变量rainbow_lock中。在游戏运行过程中，钥匙插盘的current_keys可能会有更新，但correct_keys保持不变。

data Keys_Lock类型定义在“Data_define.hs”文件中，其唯一变量rainbow_lock定义在“Locks.hs”文件中。

图 26 七彩钥匙插盘（插入了 3 把钥匙）

4.3.8 对话提示

游戏的对话提示为字符串String类型，本游戏全部的对话提示记录在常量字符串列表all_dialogues中。all_dialogues定义在“Diag.hs”文件中，在游戏运行过程中它可以被索引，但不会被修改。

4.3.9 图片

游戏运行过程中的大量场景、物品栏、物品等均以图片形式呈现在窗口中，这些图片也是预先确定的。图片有3方面的信息：图片文件路径、图片在窗口中的显示位置和图片的大小。由于图片的大小已经记录在BMP图片文件中，因此在代码中只需要记录图片的文件路径和其在窗口中的显示位置即可。

全部图片的文件路径记录在常量all_images_file中，all_images_file定义在“Images.hs”文件中，在游戏运行过程中它可以被索引，但不会被修改。

全部图片在窗口中的显示位置用图片的左上角坐标来记录，所有图片显示在窗口中的左上角坐标点记录在常量all_images_pos中，all_images_pos定义在“Images_pos.hs”文件中，在游戏运行过程中它可以被索引，但不会被修改。

4.3.10 小结

经过前面几节的介绍，我们发现在大量的游戏资源中，有些资源在运行过程中不断变化（如界面、密码锁等），有些资源在运行过程中恒定不变（如对话、图片等）。因此，本章首图 18所显示的循环更新的Data数据仅包含界面、场景列表、触发域列表、密码锁和钥匙插盘这几种Data数据；物品列表、对话提示列表和图片列表则作为恒定的游戏参数。

4.4 核心函数

本节介绍游戏源码中支持游戏运行的几个重要函数及其相关的子函数。

4.4.1 主函数：main

本游戏的主函数main的类型为“IO ()”，在main函数的开头，我们初始化SDL图像，音乐，文本；而后进入游戏整体逻辑。

4.4.2 绘制游戏界面：draw_interface

上节提到，由于需要使用SDL2库相关的对象和函数，draw_interface函数以

“let”语句的方式定义在主函数的代码段中。draw_interface函数的定义如下图所示，可以看到，draw_interface函数的入口参数有interface、all_scenes和all_triggers，其内部执行“do”语句块所标志的IO操作。

```
let
  draw_interface interface all_scenes all_triggers = do
```

图 27 draw_interface 函数定义

具体来说，draw_interface函数内部有如下几段代码，它们以空行分割，分别执行下列操作：

1) 绘制物品栏和对话提示栏的背景图；
2) 绘制物品选框（如果有物品被选中的话）；
3) 绘制物品栏中的物品小图（如有有物品的话）；
4) 绘制对话提示文本；
5) 绘制主场景及其触发域；
6) 绘制当前场景及其触发域（如果不同于主场景的话）
7) 更新窗口

其中，往窗口上绘制图片用到了SDL2的库函数SDL.surfaceBlit，最后更新窗口用到了SDL2的库函数SDL.updateWindowSurface，这两个函数在前述章节已经介绍过了。另外，“return $ Just (SDL.Rectangle (P (V2 0 0)) (V2 0 0))”语句并不绘制任何图像，只是作为冗余代码，完整填充if语句的then或else分支。

draw_interface函数的完整代码详见“main.hs”文件的main函数中，其部分代码段的含义介绍如下。

4.4.2.1 绘制物品选框和物品小图

当选中某一物品时，该物品的显示区域会出现黄色的外边框，其绘制方式是：先画外边框的图，再在其上叠加画物品的图。因此，draw_interface函数要先通过if语句判断是否要画外边框，如果需要的话就先画外边框，再在之后的代码段中绘制具体的物品小图。

由于只有有限的5个物品显示区域，因此最多只能画5个物品小图。画物品小图的函数为loop_draw_items，它通过判断showed_first_item的值和possessed_items列表中的物品个数，以确定5个物品显示区域中是否要绘制哪些物品的小图。

4.4.2.2 绘制对话提示文本

在游戏界面的对话提示栏显示文本的方式是：先将文本用SDL.Font.solid函数生成SDL2库的图片格式，然后再用SDL.surfaceBlit函数绘制在对话提示栏。

4.4.2.3 绘制场景和其中的触发域

由于当前场景要显示在主场景的前面，因此当当前场景不同于主场景时，要先绘制主场景，再在其上叠加绘制当前场景。

绘制当前场景较为简单，因为当前场景中的全部图片和图片显示位置已经在all_images_file和all_images_pos中定义好了。只需要先通过get_scene_images函数获取当前场景和其中全部触发域的图片，然后在循环调用loop_draw_cur函数一个个绘制图片即可。

然而，主场景中包含钥匙插盘的绘制。由于各钥匙孔被插入的钥匙不同，因此在绘制过程中需要实时确定钥匙孔上的钥匙图片。故绘制主场景的方式是：先绘制主场景的背景大图，然后依次绘制各触发域的小图；如果触发域的类型非钥匙孔，则直接按照all_images_file和all_images_pos中的定义绘制；如果触发域的类型为钥匙孔，则需要根据其被插入的钥匙而实时选择所绘制的图片，图片的显示位置即为钥匙孔触发域的位置。

4.4.3 游戏引擎：loop_run

loop_run函数即为游戏引擎顶层循环调用的函数，它和draw_interface函数类似的以“let”语句的方式定义在主函数的代码段中。loop_run函数的定义如图 28所示，可以看到，loop_run函数的入口参数即为前述总结的在游戏运行中需要实时更新的Data数据interface、all_scenes和all_triggers等，其内部执行“do”语句块所标志的IO操作。

```
let
  loop_run interface all_scenes all_triggers house_password_lock bronze_password_lock file_password_lock rainbow_lock = do
    event <- map SDL.eventPayload <$> ((<$>) (\a -> a:[]) SDL.waitEvent)  --get [one event]
    mousePos <- SDL.getAbsoluteMouseLocation  --get mouse position

    let whether_click = getAny $ foldMap (\case
              SDL.MouseButtonEvent e -> Any $ SDL.mouseButtonEventMotion e == SDL.Pressed
              otherwise -> Any False) event
    let quit = SDL.QuitEvent `elem` event

    if whether_click
```

图 28 loop_run 函数定义及头部代码

具体来说，loop_run函数内部有如图 28所示的三段代码，它们以空行分割，功能分别是：

1) 调用SDL2库函数，获取当前事件“head event”和鼠标位置mousePos；

2) 获取初步的事件和鼠标信息——是否是鼠标点击操作？是否为“退出”事件？

3) 如果是鼠标点击事件，则选择特定的if-then-else分支进行处理。

如果“if whether_click”被判断为真，则需要根据鼠标点击的位置和当前的游戏状态在之后大量的if-then-else分支中选取特定的操作，这些分支如下：

1) 点击物品栏：包括选中物品（五个物品显示区域之一）、上下翻页和点击放大镜，这部分操作在4.4.4节具体介绍。

2) 点击“存档”按钮：在游戏开始后，玩家点击存档按钮将会记录当前的游戏进度和玩家状态，这部分操作在4.4.6节具体介绍。

3) 点击“读取存档”按钮：在游戏开始界面中，玩家点击读取存档按钮将会读取上次记录的游戏进度和玩家状态，这部分操作在4.4.7节具体介绍。

4) 返回主菜单：在游戏开始后，玩家可以随时点击“返回”按钮回到游戏的开始界面。一旦玩家选择返回主菜单，则游戏引擎将先绘制主菜单的图片，然后丢弃玩家之前的游戏进度和状态，采用其初始化值继续运行游戏引擎。

5) 点击触发域：如果鼠标点击事件非上述任何一种情况的话，则认为玩家可能点击了场景中的触发域，这部分操作在4.4.5节具体介绍。

在上述分支中，如图 29所示，代码基本为特征性的3行：

1) 第一行为“let”语句，调用相应的操作函数，处理事件，接收更新后的游戏资源变量；

2) 第二行调用draw_interface函数，绘制新的界面；

3) 第三行循环调用loop_run函数，用新的参数继续运行游戏引擎。

```
else do let (user_quit, new_interface, new_all_scenes, new_all_triggers, ne
         draw_interface new_interface new_all_scenes new_all_triggers
         unless (quit || user_quit) (loop_run new_interface new_all_scenes r
```

图 29 loop_run 函数中 if 分支的特征代码

loop_run函数的完整代码详见“main.hs”文件的main函数中。

4.4.4 点击物品栏：event_

在4.4.3节提到，玩家点击物品栏的事件处理占据loop_run函数的多个if分支，每个分支都是特征的3行代码，代码之间的区别在于第一行游戏资源参数的更新函数，这些点击物品栏后的游戏资源更新函数定义如下：

1) 点击放大镜“event_magnifier”：在游戏运行过程中，玩家点击放大镜区域可以查看物品大图，即主场景不变，当前场景更新为该物品大图的场景；另外，查看物品时也会有相应的物品说明信息。因此，“event_magnifier”函数仅会修改interface界面变量。

2) 物品上下翻页“event_turnpageup/down”：玩家点击物品栏的上下翻页键，且有多余的物品可以显示的话，“event_turnpageup/down”函数会更新interface中showed_first_item的值。向上翻页则showed_first_item减1，向下翻页则showed_first_item加1。

3) 选中某一物品“event_item1/2/3/4/5”：玩家点击某一物品显示区域，且该区域显示有物品的话，“event_item1/2/3/4/5”函数会更新interface中的selected_item为该区域的物品编号。

上述8个与点击物品栏相关的interface更新函数详见“Item_functions.hs”代码文件。

4.4.5 点击场景中的触发域：try_triggers 与 trigger_event

在4.4.3节提到，当玩家点击其他区域时，游戏引擎会猜测玩家尝试点击当前场景中的触发域，因而会调用try_triggers函数判断玩家是否点中了某触发域，以及该触发域是否被成功触发。如果玩家成功触发了某触发域，游戏引擎会进一步调用trigger_event函数实施相应的触发事件、更新游戏资源。

本节所属的两个核心函数try_triggers与trigger_event详见“Trigger_functions.hs”代码文件，其中所用到的一些辅助的小函数详见“Auxiliary_functions.hs”代码文件。

4.4.5.1 判断是否触发事件：try_triggers

try_triggers的入口参数有两类，第一类为鼠标位置mousePos，第二类为当前的游戏Data数据interface、all_scenes和all_triggers等7个；返回值除了更新后的游戏Data数据外，还包含用户是否结束游戏的Bool类型值user_quit。

try_triggers共有3个处理分支：

1) 如果成功触发的触发域列表my_trigger为空，且失败触发的触发域列表fail_trigger也为空：这说明用户并没有点中任何触发域。try_triggers函数调用update_dialogue函数将interface数据中的对话提示更新为空（-1），然后原样返回输入的游戏资源数据。

2) 如果失败触发的触发域列表fail_trigger不为空：这说明用户点中了某触发域，但由于缺少条件而没能触发该触发域。try_triggers函数调用update_dialogue函数将interface数据中的对话提示更新为该触发域的失败提示“fail_dialogue”，然后原样返回输入的游戏资源数据。

3) 最后，如果成功触发的触发域列表my_trigger不为空：这说明用户成功触发了某触发域，try_triggers函数调用trigger_event函数触发相应的事件。

对于已有的触发域，其是否被成功触发有两个条件：条件一是调用within_tri_pos函数，判断鼠标是否落在触发域内；条件二是调用have_useful_item函数，判断用户是否选中（使用）触发所需的物品。

4.4.5.2 触发事件：trigger_event

如果try_triggers函数判断用户成功触发了某触发域，则会调用trigger_event函数处理相应的触发事件。由于触发域的类型和操作较多，因此trigger_event函数的代码段相对庞大一些。

trigger_event函数的入口参数不再需要鼠标位置，而需要被成功触发的触发域的索引值以及其他游戏资源数据；trigger_event函数的返回值与try_triggers函数的返回值类型匹配。

trigger_event函数的分支操作数量与触发域的类型“Triggertype”数量基本一致，即trigger_event函数是根据触发域的类型来区别处理不同的触发事件的。其中较特殊的是，为了避免玩家在进出房门时重复输入密码，“HOUSE_PASSWORD_BUTTON”类型的房屋密码锁按钮占据两个操作分支：当房门还没有被打开时，一个分支正常处理密码按钮的输入事件；如果房门已经被打开，则另一个分支原样返回输入的游戏资源，不再改动房屋密码锁的信息。

```
trigger_event :: Int -> Interface -> [Scene] -> [Trigger] -> Password_Lock -> Password_Lock ->
trigger_event my_tri_index interface all_scenes all_triggers house_password_lock bronze_passwo
  | (trigger_type my_tri == HOUSE_PASSWORD_BUTTON) && (judge_open house_password_lock) = (Fals
  | ((trigger_type my_tri) == HOUSE_PASSWORD_BUTTON) && (not (judge_open house_password_lock))
  | (trigger_type my_tri) == BRONZE_PASSWORD_BUTTON = (False, bronze_new_interface, all_scenes
  | (trigger_type my_tri) == ADVANCED_PASSWORD_BUTTON = (False, advanced_new_interface, all_sc
  | (trigger_type my_tri) == KEY_HOLE = (False, rainbow_new_interface, all_scenes, rainbow_new
  | (trigger_type my_tri) == MENU = (False, init_interface, init_scenes, init_triggers, house_
  | (trigger_type my_tri) == END_GAME = (True, interface, all_scenes, all_triggers, house_pass
  | otherwise = (False, ordinary_new_interface, ordinary_new_all_scenes, ordinary_new_all_trig
      where my_tri = all_triggers !! my_tri_index
```

图 30 trigger_event 函数定义及头部代码

如图 30所示，trigger_event函数的核心功能就是根据不同类型触发域的信息返回更新后的游戏资源Data数据，每一种情况下新数据的更新方式/变量定义在where语句块中。具体来说，有下列游戏资源的更新：

1) 【HOUSE_PASSWORD_BUTTON】在房屋密码锁未开的情况下输入密码：首先，trigger_event函数会更新房屋密码锁的密码输入信息为“new_house_password_lock”；另外，如果房门被打开的话，还需要更新触发域列表信息为“house_new_all_triggers”，使用户可以进入房间。

2) 【BRONZE_PASSWORD_BUTTON】点击铜像密码锁按钮：由于铜像密码锁被打开后就会消失，因此不需要像房屋密码锁一样多一个分支处理密码锁打开后的操作。和点击房屋密码锁类似的，trigger_event函数也会先更新铜像密码锁的密码输入信息为“new_bronze_password_lock”，如果密码锁被打开的话，还需要更新界面和触发域列表为“bronze_new_interface”和“bronze_new_all_triggers”，以打开铜像暗格，允许用户使用七彩钥匙插盘。

3) 【ADVANCED_PASSWORD_BUTTON】点击文件/保险箱密码锁按钮：首先，trigger_event函数会更新文件密码锁的密码输入信息为“new_file_password_lock”；另外，如果用户触发了隐藏密码或正确密码后，还需要更新界面信息为“advanced_new_interface”，推动下一步的游戏剧情。

4) 【KEY_HOLE】点击钥匙插盘中的某一插孔：首先，trigger_event函数会更新钥匙插盘的插入情况为“new_rainbow_lock”；另外，如果用户成功将七彩钥匙插入到了对应的插孔位置，则需要进一步更新界面和触发域列表为“rainbow_new_interface”和“rainbow_new_all_triggers”，以出现文件保险箱。对于本事件来说，上述各更新参数通过两个if分支进行两种不同的操作：如果该插孔已经插有钥匙（pull_out_key_item >= 0），则认为用户在拔出钥匙，被拔出的钥匙增添到用户所拥有的物品列表中；否则，如果用户选中了某一钥匙（elem rainbow_lock_item [3, 6, 7, 8, 11, 12, 16]），则认为用户在插入钥匙，被插入的钥匙记录在钥匙插盘中，同时从用户所拥有的物品列表中删除该钥匙。

5) 【MENU】返回主菜单/游戏开始界面：当用户到达解谜游戏的终点后，结局界面的右下角有“结束游戏”的触发域箭头，用户点击该触发域将会回到游戏主菜单。对于本触发域，trigger_event函数会直接返回游戏资源的初始化值（init_interface、init_scenes……），将游戏状态和玩家状态归零。

6) 【END_GAME】退出游戏：当用户点击主菜单中的“退出游戏”触发域后，trigger_event函数会将返回值的user_quit标志位置为True；游戏引擎

loop_run接收到user_quit信息后，会退出游戏，关闭游戏窗口。

7) 【ORDINARY】点击其他普通触发域：当用户点击其他大量的普通触发域时，trigger_event函数会根据前面介绍的Trigger类型的全部更新信息参数将游戏资源分别更新为ordinary_new_interface、ordinary_new_all_scenes和ordinary_new_all_triggers。

4.4.6 存档：record_

在4.4.3节提到，玩家在游戏过程中点击存档按钮的事件处理占据loop_run函数的一个if分支。然而，由于存档操作涉及到文件的写操作，因此其分支操作，和前述特征性的3行代码有所不同。

```
else if (within_ra_pos mousePos ra_record) && (not (elem (main_scene interface) [87, 88, 89, 90]))
    then do r1 <- record_interface interface
            r2 <- record_all_scenes all_scenes
            r3 <- record_all_triggers all_triggers
            r4 <- record_all_locks house_password_lock bronze_password_lock file_password_lock ra
            if r1 && r2 && r3 && r4
            then do let new_interface = interface {current_scene = 91}
                    draw_interface new_interface all_scenes all_triggers
                    loop_run new_interface all_scenes all_triggers house_password_lock bronze_pas
            else do let new_interface = interface {current_scene = 92}
                    draw_interface new_interface all_scenes all_triggers
                    loop_run new_interface all_scenes all_triggers house_password_lock bronze_pas
```

图 31 游戏引擎中的存档代码段分支

如图 31所示，游戏引擎首先调用4个文件写操作相关的“record_”函数，（尝试）将游戏的动态资源记录在4个“record_…txt”文本中，然后判断文件写操作是否成功（r1 && r2 && r3 && r4）。如果成功，则显示“存档成功”的界面，继续运行游戏；如果失败，则显示“存档失败”的界面，继续运行游戏。

文件写操作相关的“record_”函数有两种类型。第一种类型针对interface、密码锁和钥匙插盘这几种数据量较小的单一数据写入，它们的数据量是固定的几个，因此只需要一级“record_”函数写入相应文件即可。以“record_interface”函数为例，如图 32所示，首先用openFile函数打开文件，并将文件句柄传给handle，然后调用hIsOpen函数判断文件是否已经打开。如果文件已经打开，则配合使用hPutStrLn函数和show函数将interface数据中的各个信息逐行写入"record_interface_data.txt"文件中，最后调用hClose函数关闭文件；如果文件没有打开，就返回错误信息。

```haskell
record_interface :: Interface -> IO (Bool)
record_interface interface = do handle <- openFile "record_interface_data.txt" WriteMode
                                h_is_open <- hIsOpen handle
                                if h_is_open
                                then do hPutStrLn handle (show (dialogue interface))
                                        hPutStrLn handle (show (possessed_items interface))
                                        hPutStrLn handle (show (selected_item interface))
                                        hPutStrLn handle (show (showed_first_item interface))
                                        hPutStrLn handle (show (main_scene interface))
                                        hClose handle
                                        return (True)
                                else return (False)
```

图 32 record_interface 函数存档 interface

另一种“record_”函数类型针对场景和触发域这两种数据量较大的数据列表写入，由于数据列表存储有若干个独立元素，因此需要借助“loop_record_”函数依次将列表中的各元素写入文件。以存档场景列表为例，如图 33所示，“record_all_scenes”函数和前述“record_interface”函数的代码比较类似，只不过将原来写入文件的操作改为调用“loop_record_all_scenes”函数循环实施。“loop_record_all_scenes”函数的循环操作方式是：判断当前输入的场景列表中是否还有元素，如果有的话，则将头部元素取出来，将其全部信息逐行写入"record_all_scenes.txt"文件，然后用尾列表继续调用“loop_record_all_scenes”函数。

```haskell
record_all_scenes :: [Scene] -> IO (Bool)
record_all_scenes all_scenes = do handle <- openFile "record_all_scenes.txt" WriteMode
                                  h_is_open <- hIsOpen handle
                                  if h_is_open
                                  then do loop_record_all_scenes handle all_scenes
                                          hClose handle
                                          return (True)
                                  else return (False)

loop_record_all_scenes :: Handle -> [Scene] -> IO ()
loop_record_all_scenes handle all_scenes = do if length all_scenes > 0
                                              then do let cur_scene = head all_scenes
                                                      hPutStrLn handle (show (background cur_scene))
                                                      hPutStrLn handle (show (triggers cur_scene))
                                                      loop_record_all_scenes handle (tail all_scenes)
                                              else return ()
```

图 33 存档场景列表所需的两个函数

另外需要注意的是，游戏资源的存档信息是经过筛选的，只在文件中存储可能会发生变化的数据，一些常值尤其是像密码锁的正确密码这种关键信息是不会记录在存档文件中的。

上述与存档相关的“_record_”函数详见“Record.hs”代码文件。

4.4.7 读档：reload_

和上节所述的存档功能类似的，玩家在主菜单中点击“读取存档”按钮的事件处理占据loop_run函数的一个if分支，其具体操作是先调用与文件读相关的“reload_”函数将上次存档的游戏进度和玩家状态更新到当前的游戏资源中，然

后继续运行游戏引擎。

文件读操作相关的“reload_”函数也有两种类型。第一种类型针对interface、密码锁和钥匙插盘这几种数据量较小的单一数据读取，它们的数据量是固定的几个，因此只需要一级“reload_”函数读取相应文件即可。另一种“reload_”函数类型针对场景和触发域这两种数据量较大的数据列表读取，由于数据列表包含若干个独立元素，因此需要借助“loop_record_”函数从文件中依次取出数据对列表中的元素进行一一赋值。

我们以读档界面的函数“reload_interface”为例，展示文件读取的功能。如图34所示，首先用openFile函数打开文件，并将文件句柄传给handle，然后调用hIsOpen函数判断文件是否已经打开。如果文件已经打开，则调用hGetLine函数将“record_interface_data.txt”文件中的数据逐行读取到变量“content_”中，然后调用hClose函数关闭文件，并利用“content_”数据构建新的interface并返回；如果文件没有打开，则不更新interface。

```
reload_interface :: Interface -> IO Interface
reload_interface interface = do handle <- openFile "record_interface_data.txt" ReadMode
                                h_is_open <- hIsOpen handle
                                if h_is_open
                                then do content_dialogue <- hGetLine handle
                                        content_possessed_items <- hGetLine handle
                                        content_selected_item <- hGetLine handle
                                        content_showed_first_item <- hGetLine handle
                                        content_main_scene <- hGetLine handle
                                        hClose handle
                                        return Interface { dialogue = str2int content_dialogue
                                                         , possessed_items = str2intlist content_possessed_items
                                                         , selected_item = str2int content_selected_item
                                                         , showed_first_item = str2int content_showed_first_item
                                                         , main_scene = str2int content_main_scene
                                                         , current_scene = str2int content_main_scene }
                                else return interface
```

图 34 reload_interface 函数读档 interface

其中，文件读写数据只涉及到两种Data类型，一种是Int，另一种是[Int]。由于从文件中读到的是字符串类型，因此需要分别借助函数str2int和str2intlist将字符串分别转换为上述两种Data类型。

上述与读档相关的“_reload_”函数详见“Reload.hs”代码文件，一些辅助函数详见“Auxiliary_functions.hs”代码文件。

4.5 Haskell 特色代码

本节，我将从本游戏的源代码中挑选一些具有Haskell特色的函数或代码段进行分析与解读。

4.5.1 列表推导：获取当前场景的全部触发域

在“Auxiliary_functions.hs”文件的“get_scene_images”函数中，获取当前场

景的全部触发域“cur_triggers”用到了列表推导的写法：

```
cur_triggers = [all_triggers !! index | index <- (triggers cur_scene)]
```

上行代码中，“index”代表当前场景所包含的全部触发域对应的索引值，然后被用来在全部触发域“all_triggers”中索引特定的几个触发域，最终返回相应触发域的列表“cur_triggers”。

4.5.2 高阶函数：获取当前场景的全部触发域图片索引值

在“Auxiliary_functions.hs”文件的“get_scene_images”函数中，获取当前场景的全部触发域图片索引值“cur_triggers_images”用到了“filter”和“map”两个高阶函数：

```
cur_triggers_images = filter (>= 0) (map picture cur_triggers)
```

上行代码中，“map”函数使得触发域列表“cur_triggers”退化为图片索引值“picture”的列表；“filter”从索引值列表中筛选出非负数（有效的图片索引值）。

4.5.3 列表操作：改变列表中特定序号的元素

由于Haskell不存储变量，因此要想改变列表中的某一元素，其方法是输入旧列表、返回新列表。“Auxiliary_functions.hs”文件的“change_list_x”用来实现这一操作：

```
change_list_x :: [a] -> Int -> a -> [a]
change_list_x ls n new = (take n ls) ++ (new : (drop (n+1) ls))
```

“change_list_x”函数的功能是将列表“ls”中的第“n”号元素更新为“new”，其实现方式是：用“take”函数取出第“n”号元素之前的子列表，用“drop”函数取出第“n”号元素之后的子列表，中间的元素替换为“new”，然后进行三者的拼接操作。

4.5.4 λ表达式：将字符串中的逗号替换为空格

在“Auxiliary_functions.hs”文件的“str2intlist”函数中，“filt_core_str”字符串是将“core_str”字符串中的逗号替换为空格后得到的新字符串：

```
filt_core_str = map (\ch -> if ch == ',' then ' ' else ch) core_str
```

由于将逗号替换为空格的元素操作非常简单，因此直接用λ表达式实现了该函数功能，避免再定义琐碎的小函数。

4.5.5 条件分支与缩进：将字符转换为数字

Haskell语言对缩进的要求非常严格，特别是do语句块和条件分支语句。在我们的代码中，有用到两种条件分支写法，一种是比较常规的if-then-else分支，另一种是被称为“卫兵”的条件分支写法：

```
char2int :: Char -> Int
char2int ch
  |ch == '1' = 1
  |ch == '2' = 2
  |ch == '3' = 3
  |ch == '4' = 4
  |ch == '5' = 5
  |ch == '6' = 6
  |ch == '7' = 7
  |ch == '8' = 8
  |ch == '9' = 9
  |otherwise = 0
```

“Auxiliary_functions.hs”文件的“char2int”函数负责将一个数字字符转换为数字类型，该函数采用卫兵的条件列举方式使得代码非常规整。

4.5.6 递归处理列表中的各个元素：存档场景列表

Haskell的函数由于没有循环语句，因此当需要处理全部列表元素时，可以采用递归的方式，每次递归对一个元素执行操作。例如4.4.6节介绍的对场景列表的存档函数“loop_record_all_scenes”，它就是通过递归调用自身的方式实现了场景列表全部元素的文件写操作。

4.5.7 where：处理触发事件

Haskell提供了一种定义局部变量和函数的方式，即“where”关键字所声明的代码段。这在4.4.5.2节中所介绍的“trigger_event”函数中得到了很好的体现。“trigger_event”函数的定义方式就是用简短的语句定义返回值的“名称”，然后再用大量的where语句“解释”各返回值的含义。

当然，由于这种写法使得“trigger_event”函数体过于庞大，因此在今后后期优化时，可以按照触发域类型将其拆分为多个函数进行定义。

第5章 展望与总结

由于时间关系，本游戏还有很多未来及实现的功能，我们期望在后期有时间的话进行补充和学习：

1) 视频库的使用，为游戏增添更多的动画剧情和转场效果。
2) 改进资源初始化的赋值方式：我们在初始化资源时，是根据Data类型的数据定义进行一个个赋值的，我们期望能设计出可读性更高的游戏资源的初始化赋值方式。
3) 升级存档和读档功能：由于存档和读档功能是在最后才添加的功能，还有可扩充的功能。例如设置多份存档供玩家进行选择。
4) 进一步优化代码。

在本项目中我们学会了Haskell中SDL2等库的使用，在撰写代码时遇到了各种各样的bug，然而由于Haskell的社区不是很活跃，竟然在stackoverflow上有很多问题都不能得到解决！这着实考验了我们的debug能力，然而在这样的过程中，我们更是发现了Haskell强类型的好处，有很多bug我们都是通过报出的类型错误信息进行推断最后成功调试通过。我们也希望Haskell的社区能编写像python的tensorflow那样友好的文库档说明，以便于更多的人了解、使用Haskell。

任予琛、高正祺、刘诗玮

2018.12.28

于复旦大学微电子学院

参考资料

[1] sdl2[OL]. *http://hackage.haskell.org/package/sdl2*.

[2] Haskell SDL2 Examples[OL]. *https://github.com/palf/haskell-sdl2-examples*.

[3] sdl2-ttf[OL]. *https://github.com/haskell-game/sdl2-ttf*.

[4] sdl2-mix[OL]. *https://github.com/tempname11/sdl2-mixer*.