# Summary For work (July 1-16)

Yubin Peng 11897169

## Part 1 Screen Area Screening Tool

Goal：For the original training data acquisition, it is based on manual fine-tuning script parameters, which is time-consuming, laborious and inefficient. Therefore, **a convenient image data acquisition tool is developed**. 我们原先训练数据采集方式，是基于手工微调脚本参数（采集图像区域的位置，长宽），费时费力、效率低下。因此，开发了一种方便的图像数据采集工具是必然且急需的。

### Result

Base on VS+Qt, I realize the tool called “**ScreeShot**”, which can intercepting image information of custom area and save them as pictures. 采集指定区域图像并保存为图片（jpg）。

**Functions:**

1. Implemented by keyboard keystrokes: 通过键盘实现

* Add/delete box 添加/删除截图框
* Change box type (positive/negative) 改变截图框类型
* Start/stop recording 开始/停止截图

1. Rectangular frame can be moved and stretched at will 截图框可移动和拉伸
2. Real-time updating of the location, length and width of the box 实时显示截图框左顶点位置和长宽
3. Customize the location, length and width of the box (confirm by pressing the Enter key) 自定义截图框的左顶点位置和长宽
4. Customize Folder Naming and Sampling Interval 自定义文件名和采样频率

**Presentation:**

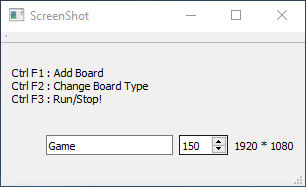


Fig.1 Set Game Name and Sampling Cycle

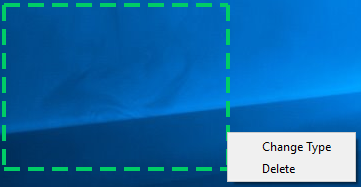


Fig.2 New Box(Dotted line) and Right-Click Menu



Fig.3 Real-time display of upper left vertex coordinates x, y, width and height of rectangular box



Fig.4 Capturing and saving image data in the Box(Solid line)

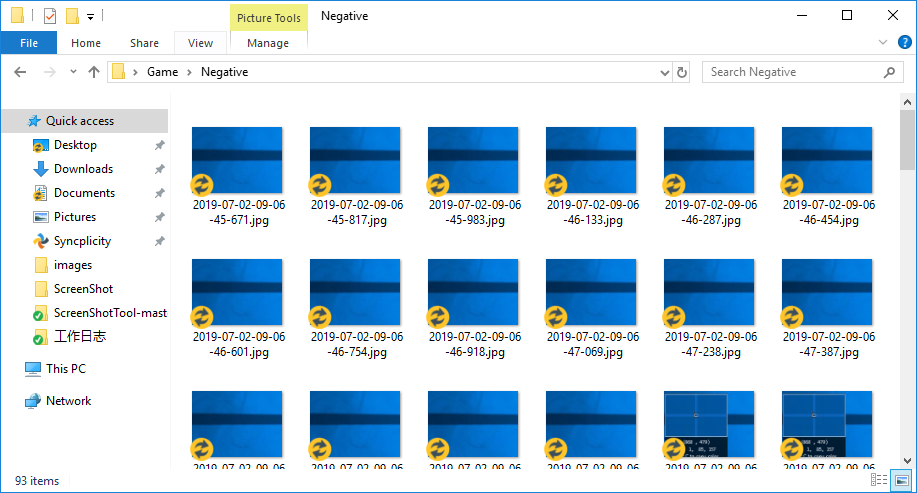


Fig.5 Image Preservation

### Learning

在此之前，我并未编写过类似的软件，这是一个新的开始。在开始编写脚本之前，我对开发的功能要求和可行性进行了分析。

#### 现有的类似软件分析：



Fig.6 EV录屏

现有的录屏软件都提供自定义区域录制和窗口置顶的功能。

#### QT截屏的可行性分析：

基于QT编写的截图软件如Snipaste功能受人推崇，而且在网上有很多基于QT的截图和录屏工程的实现。QT提供一个方法：**QScreen::grabWindow(WId window, int x = 0, int y = 0, int width = -1, int height = -1)**, 可实现自定义区域（左顶点位置、长宽）图像截取。基于此，使用QT是可行的。

#### 功能要求分析：

从实用性与具体运用场景出发，我列举了一下实现要求与相关技术，并一一实现。

* 截图框的实现 🡪 窗口去除标题栏、边框绘制（Css）
* 截图框透明，且不影响窗口下方其他软件的操作 🡪 窗口透明、鼠标穿透
* 截图框置于顶层 🡪 窗口置顶
* 截图框可移动、拉伸 🡪 鼠标事件、窗口拉伸与移动、信号触发机制
* 在截图框失去焦点时，仍对键盘事件有反应 🡪 全局热键、键盘事件
* 对特定区域图像获取 🡪 屏幕图像获取
* 持续的图像获取与保存 🡪 定时器、文件夹和文件操作、图像保存
* 截图框功能菜单 🡪 右键菜单（Menu\Action）
* 主程序最小化时，截图框不最小化 🡪 父子窗口继承机制

在我的设计中，截图框是独立的自定义控件，因此可以无限制的添加。满足于同时截取多个区域图像的要求。

### Suggestions

在机器学习、深度学习领域的研究工作中，准确、且大量的数据集是提高算法准确性的基础，也是最耗时间的。在我们的项目中，每个游戏不同模式都需要单独的数据集进行训练，所针对的位置不尽相同，而且为了提高准确率，对样本集的大小是有一定要求的。在这种情况下，开发一个便于图像数据采集的保存的工具是非常必要且明智的。

对于类似工具的开发，我有新的想法。目前模型在实际场景下的测试，光靠我们人眼对模型的准确率没有一个明确的判断。我们可以开发这样一个工具：

* **提供视频打标签功能**，在击杀/死亡等事件发生时，按下键盘特定键位，软件自动记录（文件名、事件序列[ 时间1，事件1；时间2，事件二；……]）并保存为数据文件。
* 对模型的输出进行记录，以该视频的实际事件序列进行对比（时间对比允许一定的容错率），实现**准确率和误判率的计算**

## Part 2 CsGo Server

### Result

Base on VS+Qt, I realize the tool called “CSGO\_Server”, which 基于Tcp协议实现监测用户在玩CSGO时的特殊事件，如击杀、死亡、换刀等。响应速度快，无延迟。

**Functions:**

1. Check the user's local environment for CSGO installation 检查游戏安装环境
2. Check CFG file and install it to user’s local directory 内置目标CFG文件
3. Real-time monitoring of user events and signal return 实时信息捕捉、分析和播报

**Presentation:**

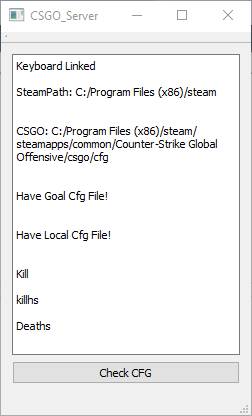


Fig.7 CSGO\_Server Show

如图，当软件开始运行，会检测键盘（DELUX）是否连接。点击“Check CFG”检查本地环境和CFG文件的配置情况。上图游戏事件发生依次为：普通击杀、爆头击杀、死亡。

### Learning

#### 用户本地环境检查，注册表信息查看

获取CSGO事件信息需要在电脑本地CSGO安装路径下放置一个CFG配置文件，那么检查用户电脑中是否安装CSGO是必要的步骤。



Fig.8 Check Process

在CSGO\_Server的开发中，是基于Qt现有的库函数实现的，在打包成外部库时需要添加Qt对应的依赖，这个将在后面的工作“Qt外部库打包”中解决。

如果用户有安装Stream，我可以通过**QSettings**从注册表中获取 Stream的安装路径，进而通过**QDir**检查用户是否安装CSGO。



Fig.9 SreamPath

#### 管理员权限的获取

要实现CSGO事件获取，需要在CSGO目录下放置特定的CFG文件，我通过**QFile**拷贝文件到指定位置。但如果用户将游戏安装在系统盘（C盘），会拷贝失败。经研究，我发现软件需要获取管理员权限。我是基于VS实现的Qt开发，经查询发现，在2017版VS可以很简单在设置里修改软件权限。

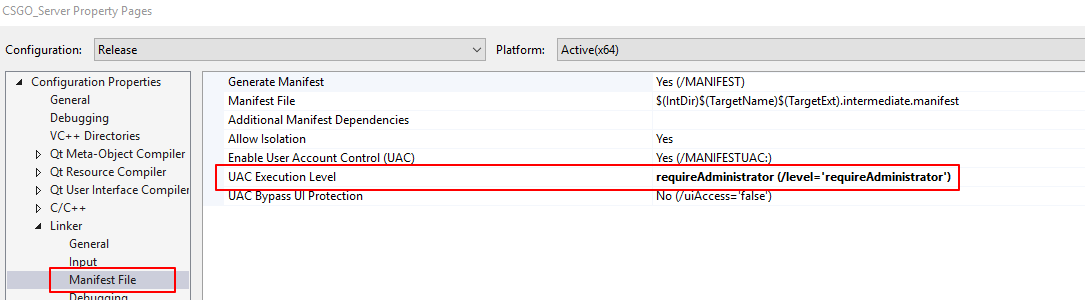


Fig.10 Change UAC Level

#### 对返回字符串的识别和信息提取

CSGO游戏是通过HTTP协议发送长字符串的方式来告知游戏事件的，我需要做的就是从中提取出关键信息，如steamid、游戏状态、击杀人数、死亡人数、武器状态等。

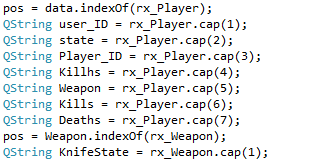
因为我们只需要实现简单的数据收发功能，所以搭建HTTP服务器不是必须的，而HTTP协议是基于TCP协议实现的。我采用Qt原有的库QTcpServer，对其接收信息函数等进行重写，实现满足项目需求的TCP服务器。

对于字符串的提取，我想到了正则表达式，经查询，Qt中有现成的库**QRegExp**，这是极好的消息。然而**QRegExp**的语法规则与常见的正则表达式不太一样，在多次尝试中，我发现以下规律。

* String: "steamid": "76561198316270523",
* RegExp: \"steamid\": \"([0-9]+)\",
* QRegExp: \\\"steamid\\\": \\\"([0-9]+)\\\",

对于转译，**QRegExp**需要多加’\\’。总的正则字符提取语句如下：

"\\\"steamid\\\": \\\"([0-9]+)\\\",([0-9a-z\\n\\t\\\"\\s\\S]+)\\\"player\\\": \\{[\\n\\t\\s\\S]+\\\"steamid\\\": \\\"([0-9]+)\\\",[0-9a-z\\n\\t\\\"\\s\\S]+\\\"round\_killhs\\\": ([0-9]+),[0-9a-z\\n\\t\\\"\\s\\S]+\\\"weapons\\\"([0-9a-z\\n\\t\\\"\\s\\S]+)\\\"kills\\\": ([0-9]+),\\n\\t\\t\\t\\\"assists\\\": [0-9]+,\\n\\t\\t\\t\\\"deaths\\\": ([0-9])+,"



以此实现对事件信息的提取。

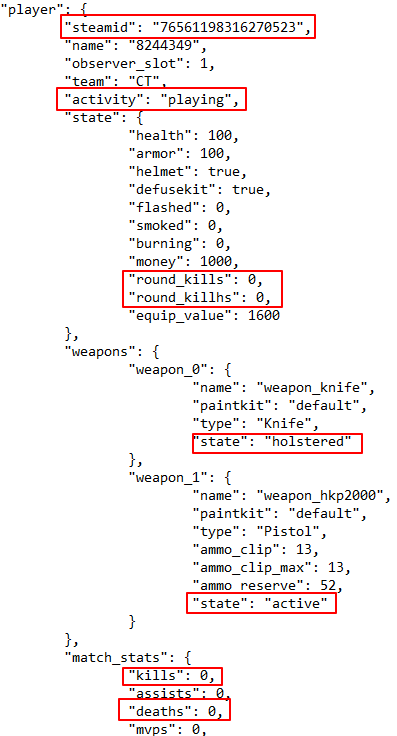
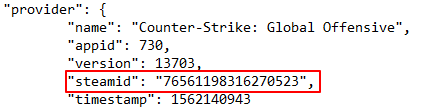


Fig.11 CSGO Message

#### 死亡事件后的他人信息误报

在用户（第一人称）死亡后，游戏视角会自动切换到第三人称，游戏会继续发送他人的游戏信息。因为我们是只针对用户本人游戏事件进行处理，所以需要进行信息过滤。



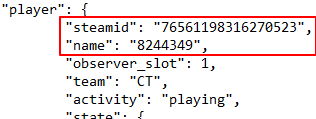


Fig.12 User Message

起初我针对”name”标签进行信息提取和判定，以第一次出现的信息提出的”name”作为用户名称（第一次出现时为第一人称，必然是该用户的），但是因为名称会出现中文、字母、数组、特定字符等，有数据误提取的可能，所以不采取。

在进一步的研究中，我发现，”provider”中提供的”steamed”是固定的，即为用户的steamid，不随游戏状态的改变而改变。而”player”中的”steamed”则与游戏状态相关，所以判断两个”steamed”是否相同可以作为游戏状态的判断，相同则为第一人称，即存活，不同则为第三人称，即死亡。

#### 解决数据包丢失的问题：HTTP 200 ok

在完成TCP服务器的基本功能后，我成功的接收到了CSGO游戏发出的消息并解析出事件信息，但是存在信息缺失和延迟问题。经研究我发现问题在于CSGO游戏是以HTTP协议发出信息，而我的服务器是TCP服务器，两者的差别在于HTTP服务器会在成功接收到数据包后发送”200 OK“数据包回复。因为我的TCP服务器并没有返回此数据报，所以游戏只能等timeout（5s）之后重新连接服务器，继续发送数据。于是我设置TCP服务器在每次收到信息之后都会直接返回以下数据报信息，成功实现无延迟的信息接收。



#### 多线程，防止阻塞

考虑到未来以此方式实现监测功能的游戏可能不止一种，我将TCP服务器修改成多线程模式，以防止消息阻塞。我采用Qt原有库**QThread**，对其**Run**函数进行重写。每当一个新的客户端连接进来，都会开启一个新的线程服务，在该客户端断开连接后退出该线程。

## Part 3 Middle Layer for Openvino Project

### Result

在我们Openvino项目中，中间层起到连接底层（models）和应用层（客户）的作用，中间层驱动底层并暴露APIs供应用层使用。

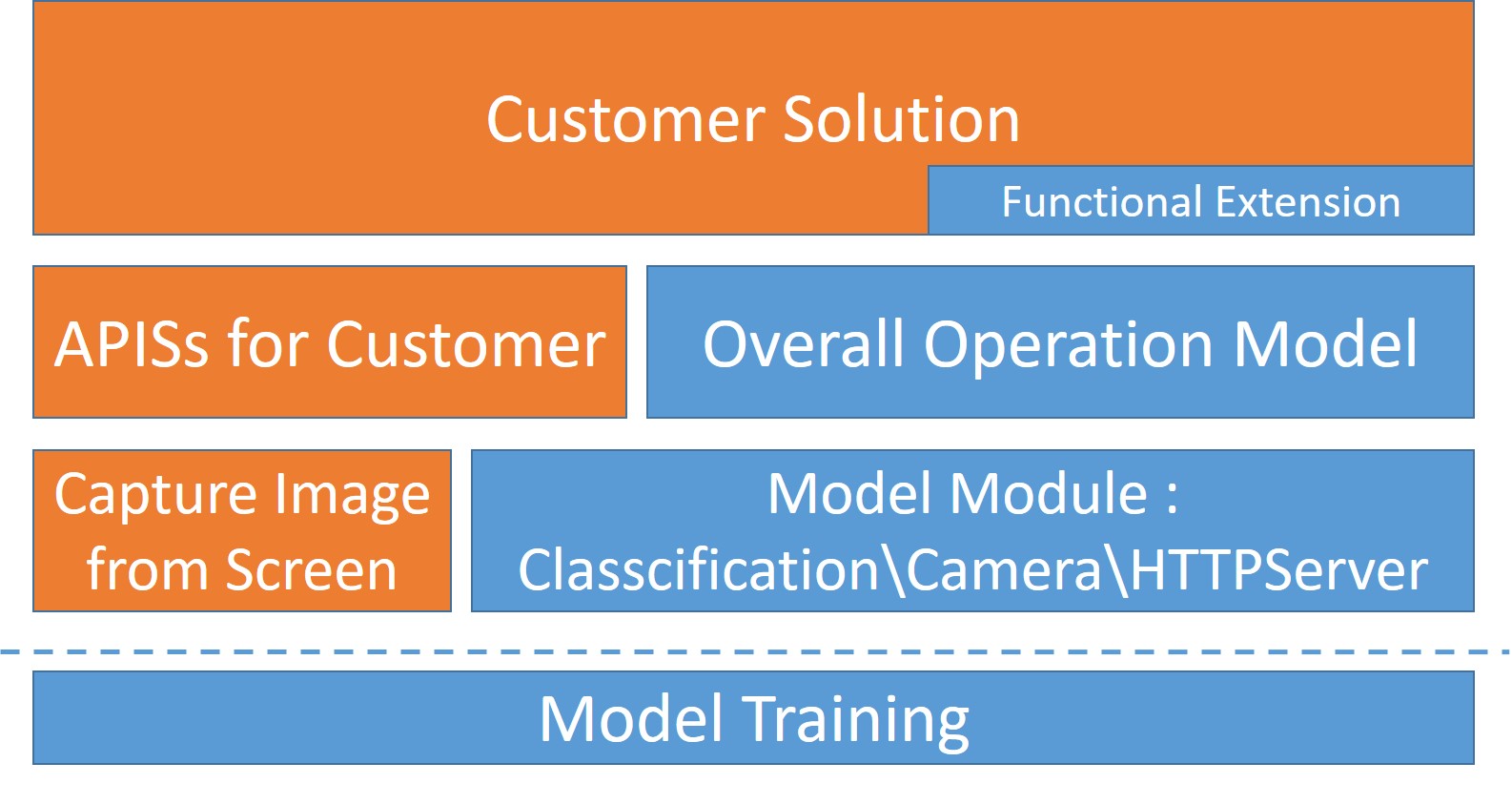


Fig.13 Layer Frame

在多次讨论后，我们定下以下几个基本的API实现：

* **Initial()**
* **MODsListReport**()
* **CfgMOD**()
* **RunMOD**()
* **StopMOD**()
* **Free**()

考虑到sdk的用户实用性，我对以上APIs进行补充：

* **Initial(**key**) :** Check key, Read XML Data
* **MODsListReport**(game) : Provide information about the patterns of the game
* **All\_MODsListReport**() : Provide information about all the patterns we support
* **CfgMOD**(modHandler, equipments): Setting for a single mode
* **CfgMOD**([modHandler], [equipments]): Setting for multiple modes
* **RunMOD**(modHandler) : Run the selected mode
* **StopMOD**(modHandler) : Stop the selected mode
* **Free**():Release memory

### Learning

#### 参数设置规范

考虑到框架对大量参数存储、修改和调用的需要，以及不同模型的参数配置。我采用XML文件（可扩展标记语言）作为一个小型数据库。设计架构如下：

<AICenter>

<Setting>

<Key> … </Key>

……

</Setting>

<GameList>

<Game Name="PBUG" Type="0" >

<Configure>

<EquipmentList>……</EquipmentList>

</Configure>

<Mod Name="PBUG\_KILL">

<Describe>PBUG\_KILL</Describe>

<Path\_XML>pbug\_kill.xml</Path\_XML>

<Path\_BIN>pbug\_kill.bin</Path\_BIN>

<Rect>

<X>100</X>

<Y>100</Y>

<W>100</W>

<H>100</H>

</Rect>

</Mod>

……

</Game>

……

</GameList>

</AICenter>

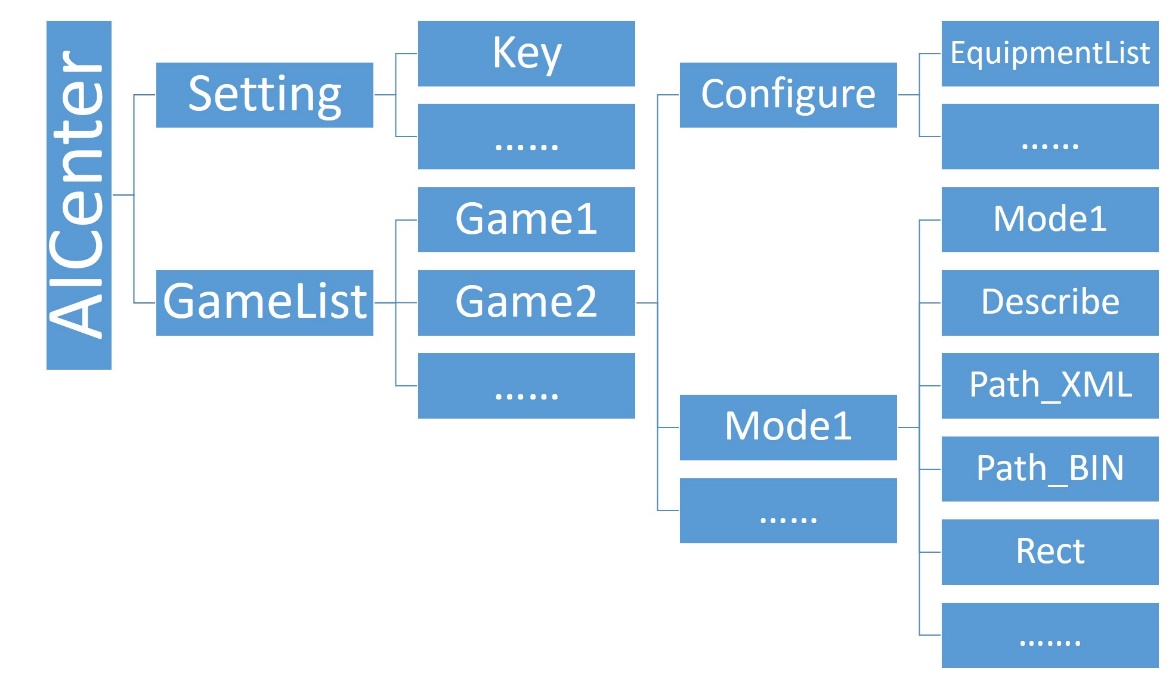


Fig.14 XML Frame

#### 游戏分类与数据存储

我发现目前同种游戏下不同模式的实现方式是相同的，存在3类实现方式，以游戏为单位：

1. **Classification: 0**
2. **Camera: 1**
3. **HttpServer : 2**

其中，监测用户头部活动（扭头）和用户离开等模型我归属为一个“游戏”下的多个模式，称为MOVEMENTS，属于Camera。标签记录格式如下：

<Game Name="MOVEMENTS" Type="1" >

<Configure>

<EquipmentList></EquipmentList>

</Configure>

<Mod Name="HEAD\_RIGHT">

<Describe>HEAD\_RIGHT</Describe>

</Mod>

……

</Game>

对于XML数据，我的设计是在初始化过程中进行读取，并用一个结构体保存它。关于在C++实现XML文件的读取，网上有很多相关开源库，在实用性和功能性的对比下，我选择tinyxml2工具，它使用更少的内存，更快，并使用更少的内存分配。

我发现不同类型的游戏的配置，和其模式需要的配置都是不同的，那么要怎么实现在一个Vector内高效简洁地保存不同类型的数据呢？我想到了软件设计模式中的**装饰者模型**，通过结构体继承的方式，实现多样化游戏和模式。游戏结构体为**AI\_GAME**，模式结构体为**AI\_MOD\_MES**。

**父类**

typedef struct \_MOD\_MES {

string Name;

AI\_MOD modInd;

string Describe;

bool ModChoose = false;

AI\_Equipment Equitment = AI\_Equipment::CPU;

}AI\_MOD\_MES;

typedef vector<AI\_MOD\_MES> AI\_MODMES\_LIST;

typedef struct \_GAME {

string Name;

AI\_MODE\_TYPE ModType;

GAME\_TYPE GameType;

int ModMinInd;

int ModMaxInd;

AI\_MODMES\_LIST ModMesList;

bool GameChoose = false;

}AI\_GAME;

**子类**

/\*\* CLASSIFICATION Struct \*/

typedef struct \_MOD\_MES\_CLASS : \_MOD\_MES {

string XML;

string BIN;

int Rect[4];

\_MOD\_MES\_CLASS(XMLElement \*Mod, int \*ModeInd) {

Name = Mod->Attribute("Name");

modInd = AI\_MOD(\*ModeInd);

Describe = Mod->FirstChildElement("Describe")->GetText();

XML = Mod->FirstChildElement("Path\_XML")->GetText();

BIN = Mod->FirstChildElement("Path\_BIN")->GetText();

XMLElement\* elmtRect = Mod->FirstChildElement("Rect");

Rect[0] = atoi(elmtRect->FirstChildElement("X")->GetText());

Rect[1] = atoi(elmtRect->FirstChildElement("Y")->GetText());

Rect[2] = atoi(elmtRect->FirstChildElement("W")->GetText());

Rect[3] = atoi(elmtRect->FirstChildElement("H")->GetText());

(\*ModeInd)++;

}

}AI\_MOD\_MES\_CLASS;

typedef struct \_GAME\_CLASS : AI\_GAME {

\_GAME\_CLASS(XMLElement \*Game, int \*GameInd, int \*ModeInd)

{

Name = Game->Attribute("Name");

ModType = AI\_MODE\_TYPE(atoi(Game->Attribute("Type")));

GameType = GAME\_TYPE(\*GameInd);

(\*GameInd)++;

ModMinInd = \*ModeInd;

//MOD

XMLElement\* elmtMod = Game->FirstChildElement("Mod");

while (elmtMod != NULL) {

ModMesList.push\_back(AI\_MOD\_MES\_CLASS(elmtMod, ModeInd));

elmtMod = elmtMod->NextSiblingElement();

}

ModMaxInd = (\*ModeInd)-1;

}

}AI\_GAME\_CLASS;

在XML框架和结构体的设计中，都满足于未来扩展加入更多的游戏。XML数据会保存在vertor<AI\_GAME>变量中。

#### 错误报错

考虑到用户在使用中间层API时，可能因为操作的不规范而造成错误，为了对中间层可能出现的不同错误进行区分性提示，我定义了一个枚举类型AI\_RESULT:

typedef enum

{

/\* no error \*/

AI\_CONFIGURE\_SUCCESS = 1, /\*\*< no error: The key matching is correct and the setup is successful\*/

AI\_ERR\_NONE = 0, /\*\*< NOT FAIL \*/

/\* reserved for unexpected errors \*/

AI\_ERR\_UNKNOWN = -1, /\*\*< unknown error. \*/

/\* error codes <0 \*/

AI\_ERR\_LACK\_XML = -2, /\*\*< Lack of XML files \*/

AI\_ERR\_KEY = -3, /\*\*< Key error \*/

AI\_ERR\_MODE\_ERRIND = -4, /\*\*< Model input errors \*/

AI\_ERR\_MODE\_CFGSet = -5, /\*\*< Model Not Set \*/

AI\_ERR\_EQUITMENT\_ERRIND = -6, /\*\*< Equipment input errors \*/

… …

} AI\_RESULT;

将根据后续开发过程中可能出现的错误继续扩充。

### Suggestions

对于中间层的实现，我的想法是立足用户的角度考虑APIs的实现，提供更多的功能选择，而且让客户操作起来方便、简单，是我们团队产出成熟化的一种展现。

未来我们可以把中间层接口定义为适应性更强，功能接口更为多元化的形式，有利于我们发展新客户和适应不同客户的要求。当然内部实现是不暴露给用户的。