获得vise3d引擎代码了，让我们看看这代码怎么下手阅读吧。

客户端：

框架入口

GameClient下的Game.cs是游戏的总入口，是一个单件，代表整个游戏，流程控制接口主要有:

Start(GameInit init)：游戏启动时调用，用来初始化游戏和引擎

Stop()：游戏关闭时调用，用来释放进程资源

Tick() ：游戏每格一定时间的更新调用

OnPause()：移动设备上切换到后台调用

OnResume()：移动设备从后台切换到前台调用

RenderTick()：每一帧的渲染调用

LogicTick()：每一帧的逻辑调用

初始化过程：

InitEngine(*CCore*.*Graphics*.*ViewTarget* vTarget, *CCore*.*Graphics*.*EDeviceType* deviceType)

在这个函数内会

1. 创建和初始化渲染环境（*CCore*.*Graphics*.*REnviroment*），里面包含了表征用于渲染的可见窗口的视口（*CCore*.*Graphics*.*View*）
2. 创建和初始化能够响应消息的界面根对象RootUIMsg
3. 启动一条异步线程用来处理逻辑（LogicTick()），主线程处理渲染（RenderTick()）

关于多线程渲染：

MainThreadTick()：入口函数

在这里，每一个帧，引擎会同时启动LogicTick()和RenderTick()，然后他们互相通过多线程事件处理当两个处理都完成后同步。同步后，再做一些引擎必须保证非多线程执行的每个Frame的善后收尾处理，比如交换各种渲染相关的前后缓冲。

IStage：

在Game单件中，我们放了一个CurrentStage，代表一个游戏表演舞台。比如登陆模块，我们可以单独做一个舞台，比如创建角色模块等，也可以派生IStage做一个舞台来写他的独有的逻辑处理，MainStage是最重要的游戏进行的舞台。通过Game的接口SetCurrentStage(IStage stage)来切换程序当前舞台。

在这个舞台，我们处理了诸如地图加载，运行管理等。

void Enter(Game game)

void Leave(Game game)

void Tick(Game game)

这是最重要的入口函数。

关于反射与数据绑定系统：

public static MainStage Instance

{

get

{

return *CCore*.*Support*.*ReflectionManager*.*Instance*.*GetClassObject*<MainStage>();

}

}

在MainStage中，他的单件处理是一个比较奇怪的代码，他不是直接new一个实例，而是通过反射管理器来获得一个实例，这个主要是方便日后的函数和数据自动绑定，结合[*CSUtility*.*Editor*.UIEditor\_BindingMethod]属性运用，我们可以在UI编辑器等环境，动态绑定UI事件响应，界面元素数据刷新等。

具体可以参阅cscommon工程的*CCore*.*Support*.*ReflectionManager*类了解细节。

网络与远程调用：

[*RPC*.*RPCClassAttribute*(typeof(GameRPC))]

public class GameRPC : *RPC*.*RPCObject；*

通过属性定制，我们可以让派生自RPC.RPCObject的类开始具备网络远程调用的能力。

在这个类，流程相关的函数有:

void OnRegServerConnected(*CSUtility*.*Net*.*TcpClient* pClient, byte[] pData, int nLength)

void OnGateServerConnected(*CSUtility*.*Net*.*TcpClient* pClient, byte[] pData, int nLength)

void OnGateServerDisConnected(*CSUtility*.*Net*.*TcpClient* pClient, byte[] pData, int nLength)

同步逻辑相关的函数，则请看有三种属性标识的函数，变量，索引器

[*RPC*.*RPCIndexObjectAttribute*(0, typeof(Guid), (int)*RPC*.*RPCExecuteLimitLevel*.*All*, false)]

[*RPC*.*RPCMethodAttribute*((int)*RPC*.*RPCExecuteLimitLevel*.*Player*, false)]

[*RPC*. RPCChildObjectAttribute (0, (int)*RPC*.*RPCExecuteLimitLevel*.*Player*, false)]

被这种标识的函数，变量，索引器，都可以被其他远程客户通过函数调用，调用代码风格大致如下：

*RPC*.*PackageWriter* pkg = new *RPC*.*PackageWriter*();

Server.H\_RPCRoot.smInstance.HGet\_RegServer(pkg).GetLowGateServer(pkg);

pkg.*WaitDoCommandWithTimeOut*(10.0F, *CCore*.*Engine*.*Instance*.Client.*RegSvrConnect*, *RPC*.*CommandTargetType*.*DefaultType*, new *System*.*Diagnostics*.*StackTrace*()).*OnFarCallFinished* = delegate (*RPC*.*PackageProxy* \_io, bool bTimeOut)

{

if (bTimeOut)

{

*UISystem*.*VMessageBox*.*Instance*.Show(Game.Instance.RootUIMsg.Root, "错误信息", "网络响应超时", *UISystem*.*VMessageBoxType*.*Ok*);

return;

}

*RPC*.*DataReader* dr;

\_io.Read(out dr);

string Ip;

*UInt16* port;

dr.Read(out Ip);

dr.Read(out port);

var ips = *System*.*Net*.*Dns*.*GetHostAddresses*(Ip);

Ip = ips[0].ToString();

*CCore*.*Engine*.*Instance*.Client.*RegSvrConnect*.Close();

*CCore*.*Engine*.*Instance*.Client.*GateSvrConnect*.Connect(Ip, port);

};

上面的例子就是客户端调用了Gate服务器上函数GetLowGateServer的示例

过程描述起来：

1. 新建一个数据包: *RPC*.*PackageWriter* pkg = new *RPC*.*PackageWriter*()
2. 客户端调用Gate服务器的接口Server.H\_RPCRoot.smInstance.HGet\_RegServer(pkg).GetLowGateServer(pkg)
3. 等待服务器执行结果返回，然后做处理：

pkg.*WaitDoCommandWithTimeOut*(10.0F, *CCore*.*Engine*.*Instance*.Client.*RegSvrConnect*, *RPC*.*CommandTargetType*.*DefaultType*, new *System*.*Diagnostics*.*StackTrace*()).*OnFarCallFinished* = delegate (*RPC*.*PackageProxy* \_io, bool bTimeOut)

这里返回后的参数从*RPC*.*PackageProxy* \_io中读取，而bTimeOut则告诉网络是否超时。

这里看起来是一个连续的过程，其实是一个非阻塞的异步行为，我们采用了匿名代理的语法糖，让逻辑变得自然易维护。