
SORA
HOSHIZORA

Hoshizora简单教程&说明文档

Author: Griffin Bu

Rev 3
Release Date: 6/9/2011

Sora的初始化

Sora的初始化包括以下几步

1.1. 包含头文件以及设置包含路径

在你的主cpp文件里包含”Hoshizora.h”, 这个头文件讲包含Sora核心头文件以及一些辅助头文件

设置包含路径主要针对Sora主文件夹以及需要使用的Plugins文件夹, 在Windows下需要注意设置Library的寻找路径, 因为VC++不会自动链接工程内的lib文件.

Sora和Plugins这两个文件夹必须被设置为头文件搜寻目录, 否则Sora将无法编译. 特别注意一些使用了第三方库的插件, 例如SoraGUIChan, SoraFreetypeFont等需要将自身文件夹也加为头文件搜寻目录.

1.2. 注册需要使用的核心组件

组件注册主要通过SoraCore类的register*系列函数完成.

SoraCore是整个Sora的最核心类, Sora核心的基础功能都依靠他来完成.

SoraCore是一个Singleton类, 你可以通过SoraCore::Instance()来获取他的实例, 或者直接使用sora命名空间内的sora::SORA定义来简化表示

当拥有的SoraCore实例之后就可以开始注册组件了, Sora自带的组件都存在于根目录的Plugins文件夹下, 你可以根据你的平台和项目需要自由设置

Sora一共有8个可以注册的核心组件, API为

registerRenderSystem 注册使用的渲染器

* registerResourceManager 添加一个资源管理器

registerInput 注册输入管理器

registerFontManager 注册字体管理器

registerSoundSystem 注册声音管理器

* registerMiscTool 注册各个平台的操纵系统相关功能实现类

* registerPluginManager 注册插件管理器

* registerTimer 注册计时器

注意registerPluginManager, registerMiscTool和registerTimer默认情况下会由SoraCore根据当前平台自动注册, 所以无需手动再注册一次, Sora自带的各个平台对应的misctool和timer存在于Sora/Defaults目录下. 当然你也可以注册你自己的对应实现.

在这些API里面, 要最简化启动Sora只需要registerRenderSystem即可, 但是会缺少很多核心功能. 如果renderSystem也缺乏, 那么Sora将无法启动.

1.2.1 registerRenderSystem

这个函数注册Sora将使用的渲染器, 渲染器需要继承自SoraRenderSystem, 这个基类定义了渲染器需要实现的接口, 包括窗口创建, 贴图加载和渲染等一系列功能.

Sora核心已经包括了三个针对不同平台的渲染器, 包括

SoraHGERenderer(SoraDXRenderer) 针对Windows

SoraOGLRenderer 针对Mac OS X和Linux

SoraOGLESRenderer 针对iOS以及其余可能的移动平台, 目前针对iOS有部分特化内容

通常你只需要包含他们的头文件然后通过registerRenderSystem注册即可, 例如sora::SORA->registerRenderSystem(new sora::SoraOGLRenderer)

1.2.2 registerResourceManager

这个函数注册Sora使用的资源管理器. 注意资源管理器在Sora核心内的表示并非唯一, 而是一个链表. 所以你可以注册复数个资源管理器. Sora核心已经自带并且自动注册了针对本地硬盘资源的管理器SoraFolderResourceManager.

资源管理器需要继承自SoraResourceManager, 基类定义了一系列子类需要实现的接口.

Sora核心内和ResourceManager相关的函数有

loadResourcePack
attachResourcePack
getResourceFile
readResourceFile
getResourceFileSize
freeResourceFile
enumFilesInFolder

对这些函数的调用都将转发到注册过的ResourceManager内, 依照注册顺序, 取第一个正确的返回值.

注意SoraFolderResourceManager总是存在于ResourceManager链表的尾部, 所以优先级总是最低. 其余ResourceManager的优先级按照注册顺序来.

核心的Plugins实现了一个针对zip压缩包的SoraZipResourceManager, 要使用只需注册即可.

1.2.3 registerInput

注册Sora使用的外设输入管理器. 子类继承自SoraInput.

Sora核心所有的外设输入函数都依赖于此, 如果没有注册Sora可以启动但是无法使用外设输入功能.

1.2.4 registerFontManager

注册Sora使用的字体管理器, 如果要使Sora能够渲染文字这个组件则必须被注册. 子类派生自SoraFontManager和SoraFont. 其中SoraFontManager用于管理字体, SoraFont则是Font的具体实现.

Sora核心的Plugins里实现了基于freetype最新版的SoraFreetyperFont和基于苹果Texture2D的SoraiOSFontManager, 分别针对桌面平台和iOS平台.

1.2.5 registerSoundSystem

注册Sora使用的声音管理器, 子类派生自SoraSoundSystem. 这个组件能让Sora拥有播放声音程度的能力.

Sora核心的Plugins里实现了基于FMOD的SoraFMODSoundSystem和基于Audiere的SoraAudiereSoundSystem. SoraAudiereSoundSystem只能运行于Windows和Linux(需要手动编译Audiere), 而FMOD则是各个平台都可以.

1.2.6 registerMiscTool

注册一个各个操纵系统平台相关功能的实现类, 例如MessageBox, 打开文件对话框等等. Sora核心已经自带了针对各个平台的实现并且会自动注册. 子类派生自SoraMiscTool.

1.2.7 registerPluginManager

注册插件管理器, SoraCore使用此组件管理所有SoraPlugin的子类. 子类派生自SoraPluginManager

1.2.8 registerTimer

注册Sora核心使用的计时器. 为各个平台提供FPS管理设置等功能. Sora核心已经实现针对Windows, Mac OS X的高精度, 低占用计时器和用标准Time库实现的通用计时器.

1.3. 创建窗口

当注册完Sora核心将要使用的组件之后, 你就可以创建窗口了. 主窗口创建通过SoraCore::createWindow完成.

Sora主窗口依靠SoraWindowInfoBase基类描述, 这个基类描述了窗口大小, 是否使用鼠标等窗口属性以及窗口的主update和render函数. 你的窗口类必须派生自这个类然后实现他的接口.

例如class myWindow: public sora::SoraWindowInfoBase {

```
    ...;
}
```

然后通过sora::SORA->createWindow(new myWindow);

注意你不可以手动删除你的窗口结构, 否则将导致崩溃. Sora核心会自动管理他的内存.

这个函数将会返回一个针对你的窗口的HANDLE, 这个handle是为了向以前版本兼容而存在的. 你可以通过判断这个handle是否为0来判断主窗口创建是否成功.

在Sora运行过程中, 你随时可以通过SoraCore的getMainWindowHandle和getMainWindow来获取你的主窗口. 其中getMainWindowHandle的语义会依据平台而不同, 例如Windows下会返回Windows的HWND结构而其余平台则和createWindow的返回值相同. 具体信息请查阅API文档. getMainWindow则会返回一个SoraWindowInfoBase指针, 你可以通过STL的dynamic_cast功能来转换到你的主窗口类.

1.4. 开始

当主窗口创建成功之后, 你就可以通过SoraCore::start()来开始运行Sora了. 此时Sora核心将完成组件检测等功能, 然后进入窗口主循环, 每帧轮流调用主窗口类的update和render函数. Sora正式开始运行.

Sora的主窗口

2.1 主窗口基础

Sora的主窗口都继承自SoraWindowInfoBase, 需要实现的接口有

定义窗口的宽高, 例如800*600

- getWindowWidth()
- getWindowHeight()

定义窗口的初始位置, (0, 0)为屏幕中央

- getWindowPosX()
- getWindowPosY()

定义窗口的名字(标题)

- getWindowName()

定义窗口的id(核心暂时无用)

- getWindowId()

初始化, 在Sora核心Start的时候被调用

- init();

是否窗口化, 如果返回false则会创建一个全屏的窗口

- isWindowed()

是否隐藏鼠标, 如果隐藏鼠标则无法使用鼠标输入

- hideMouse()

帧回调函数, 每帧调用

- renderFunc()
- updateFunc()

基于此, 我们可以创建一个简单的窗口

```
class mainWindow: public sora::SoraWindowInfoBase {
      public:
            bool updateFunc() { return false; }
            bool renderFunc() { return false; }
     void init()     { }

     int32 getWindowWidth() { return 1024; }
     int32 getWindowHeight() { return 768; }

     int32 getWindowPosX() { return 0; }
     int32 getWindowPosY() { return 0; }

     SoraString getWindowName() { return "Reflection"; }
     SoraString getWindowId() { return "MainWindow"; }

            bool isWindowSubWindow() { return
false; }
            bool isWindowed() { return true; }
            bool hideMouse() { return false; }
     };
```

这里创建了一个简单的主窗口类, 他的大小(分辨率)为1024*768.

有了这个类, 我们就可以调用SoraCore的createWindow函数来注册Sora要使用的主窗口了

sora::SORA->createWindow(new mainWindow);

sora::SORA->start();

2.2 开始渲染

Sora每一帧的渲染在开始前我们都必须情况上一帧的内容和通知Sora我们开始一帧的渲染了, 这个功能可以通过调用SoraCore的beginScene函数来达成.

同时在渲染的最末, 我们必须通知Sora这一帧的渲染结束, 把缓冲区的内容刷新上屏幕好让用户看到, 这个可以通过调用SoraCore的endScene达成

所以一般来说, 一个主窗口的renderFunc总是以这两个函数开头和结尾, 例如

```
bool mainWindow::render() {
    sora::SORA->beginScene();

    // do some rendering stuff

    sora::SORA->endScene();
}
```

2.3 深入beginScene

从上边的例子我们看到beginScene是没有带参数的, 但是实际上这个函数可以接收两个参数, 一个是clear color, 一个是render target.

2.3.1 ClearColor

这个无符号32位整数参数定义了刷新后的背景颜色, 默认值是黑色. 例如我们在刚才那个例子中这样调用beginScene函数

sora::SORA->beginScene(0xFF0000FF)

那么我们将得到一个红色的背景

* 32位整数RGBA颜色的表示

一般我们用一个无符号的32位整数来表示RGBA颜色, 每个分量为8位, 取值0-255, 所以颜色0xFF0000FF的R = FF = 255, G = 0, B = 0, Alpha = FF = 255

同时在SoraColor.h里定义了一些方便分离和设置RGBA分量的宏, 包括

CSETA

CSETR

CSETG

CSETB

CGETA

CGETR

CGETG

CGETB

例如

CGETA(0xFFFFFFFF)就会返回255

而CSETA(0x00FFFFFF, AA)就会返回0XAAFFFFFF

2.3.2 RenderTarget

这个参数定义了Sora的渲染目标. 默认情况下, Sora会直接把需要渲染的内容渲染到屏幕, 但是如果我们需要把内容渲染到一张贴图上的话就需要用到这个函数了. 这个参数也是一个无符号32位整数, 代表一个RenderTarget的句柄(handle). RenderTarget可以通过如下函数创建

SoraCore::createRenderTarget(int32 width, int32 height, bool zbuffer);

这个函数接收3个参数, 前两个参数代表我们要创建的贴图target的宽和高, 0代表使用主窗口的宽或者高, 后一个参数代表是否使用zbuffer缓冲, 也就是渲染内容到这个target上时使用zbuffer. 关于zbuffer的更多内容参见ZBuffer一节.

RenderTarget在使用完后可以通过

SoraCore::freeTarget(ulong32 target)

来释放他占用的内存

在我们在一个target上渲染完成我们需要渲染的东西后, 我们同样需要调用一次sora::SORA->endScene()来强制清空缓冲区. 然后我们就可以通过

SoraCore::getTargetTexture(ulong32 target)

来获取这个RenderTarget的贴图. 在拥有了贴图之后我们就可以把他渲染到屏幕了. 下边是一个在一个target上画一个大小为窗口大小的box, 然后在屏幕上渲染这个target的例子, 关于SoraSprite的内容你可以先不用担心, 暂时只用知道一个Sprite代表一个可渲染的物件就行了.

```
void mainWindow::init() {
     myTarget = sora::SORA->createTarget(0, 0, true);
     mySprite = new sora::SoraSprite(0);
}

bool mainWindow::render() {
     sora::SORA->beginScene(0, myTarget);
     sora::SORA->renderBox(0.f,
               0.f,
               getWindowWidth(),
               getWindowHeight(),
                    0xFFFFFFFF,
               0.f);
     sora::SORA->endScene();

     sora::SORA->beginScene();
     mySprite->setTexture(sora::SORA->getTargetTexture(myTarget));
     mySprite->render();
     sora::SORA->endScene();
}
```

2.4 帧循环

从主窗口我们可以看到一帧的循环分为两个函数, update和render, 我们一般在update函数里进行物件的update, 例如运动, 然后在render里渲染, 以使游戏的逻辑更加清晰.

2.5 FPS以及时间

SoraCore内有一系列时间相关的函数, 他们由SoraTimer实现, 包括

setFPS(int32 fps)　设置fps
getFPS()　获取当前fps
getDelta()　获取当前帧间隔时间
getTime()　获取运行时间
getFrameCount()　获取总运行帧数
setTimeScale(scale)　设置时间流逝比例, 这个函数影响getDelta的值
getTimeScale()　获取当前时间流逝比例
getCurrentSystemTime()　获取当前系统时间

2.6 总结

做完以上步骤, 我们已经完成了第一步, 创建一个窗口, 下面我们将在上面渲染一些东西来开始创作.

本节完整的初始化代码如下

```
#include “Hoshizora.h”
#include “Plugins/SoraOGLRenderer/SoraOGLRenderer.h”
#include “Plugins/SoraOGLRenderer/SoraOGLInput.h”
#include “Plugins/SoraFMODSoundSystem/SoraFMODSoundSystem.h”
#include “Plugins/SoraHGERenderer/SoraHGERenderer.h”
#include “Plugins/SoraHGERenderer/SoraHGEInput.h”

int main(int argc, char* argv[]) {
     sora::SoraCore* sora = sora::SoraCore::Instance();
     sora->registerRenderSystem(new sora::SoraOGLRenderer);
     sora->registerInput(new sora::SoraOGLInput);
      sora->registerSoundSystem(new
sora::SoraFMODSoundSystem);

     sora->createWindow(new mainWindow);
     sora->start();

     return 0;
}
```

注: 在Windows下OGLRenderer也可以运行, 但是如果你需要使用基于DirectX的Renderer, 逆序要将sora->registerRenderSystem(new sora::SoraOGLRenderer)改为sora->registerRenderSystem(new sora::SoraHGERenderer), 相应的input也要改为sora->registerInput(new sora::SoraHGEInput)

Ex: 跨平台定义

在SoraPlatform.h内包含了关于平台的一些宏定义, Sora核心会根据编译器相关内容判断当前平台, 所以你可以通过那些宏定义来得知当前的平台, 比如在2.6的例子中, 我们可以通过简单的修改让代码在Windows平台下使用HGERenderer而OS X下使用OGLRenderer

```
#include “Hoshizora.h”
#include “Plugins/SoraOGLRenderer/SoraOGLRenderer.h”
#include “Plugins/SoraOGLRenderer/SoraOGLInput.h”
#include “Plugins/SoraFMODSoundSystem/SoraFMODSoundSystem.h”
#include “Plugins/SoraHGERenderer/SoraHGERenderer.h”
#include “Plugins/SoraHGERenderer/SoraHGEInput.h”

int main(int argc, char* argv[]) {
     sora::SoraCore* sora = sora::SoraCore::Instance();

#if defined(OS_OSX)
     sora->registerRenderSystem(new sora::SoraOGLRenderer);
     sora->registerInput(new sora::SoraOGLInput);
#elif defined(OS_WIN32)
     sora->registerRenderSystem(new sora::SoraHGERenderer);
     sora->registerInput(new sora::SoraHGEInput);
#else
     return 0;
#endif

     sora->registerSoundSystem(new sora::SoraFMODSoundSystem);

     sora->createWindow(new mainWindow);
     sora->start();

     return 0;
}
```

目前关于平台的宏定义有
OS_OSX
OS_WIN32
OS_LINUX
OS_IOS
这四种

同时SoraPlatform内还包括了很多数据类型的定义, 方便32/64 bit之间的编译, 他们包括

uint8, uint16, uint32, ulong32, uint64, int8, int16, int32, long32, int64

其中需要特别注意的是ulong32, 虽然名字是ulong32但是实际上长度会依据平台而不同, 如果需要32位整数请用int32/uint32, 但是如果需要指针handle请用ulong32, 因为指针长度会变化.

Texture和Sprite 贴图和精灵

3.1 Texture

Texture代表是的一张可渲染的贴图, 但是他本身不能被直接渲染, 需要借助于Sprite.

Texture可以从一张图片创建, 例如loli.png, 也可以指定宽高创建一张空的贴图, 这两个功能通过SoraCore内的

createTexture(const SoraWString& path)

createTextureWH(int32 width, int32 height)

来完成

* 字符串的表示

在Sora里有两种String命名, SoraWString和SoraString, 分别代表std::string和std::wstring.

这两个函数返回创建的Texture的Handle, 类型是ulong32或者sora::HSORATEXTURE, 两个表示等同

3.2 Sprite

有了Texture以后, 我们就可以创建Sprite了, Sprite是Sora内的最基本渲染单元. 类在Sora/SoraSprite.h内申明. 他有两个构造函数.

```
SoraSprite(HSORATEXTURE tex);
SoraSprite(HSORATEXTURE tex, float32 x, float32 y, float32 width, float32 height);
```

3.2.1 渲染

SoraSprite拥有三个渲染函数, 分别为

```
   void render();
   void render(float32 x, float32 y);
   void render4V(float32 x1, float32 y1, float32 x2, float32 y2, float32 x3, float32 y3, float32 x4, float32 y4);
```

分别是渲染到设置的位置, 渲染到指定位置, 渲染到一个矩形内

3.2.2 Sprite变换

```
void setScale(float32 h, float32 v);
float32 getVScale() const;
float32 getHScale() const;
void setRotation(float32 r);
float32 getRotation() const;
void setRotationZ(float32 rz);
float32 getRotationZ() const;
void setFlip(bool hflag, bool vflag, bool bFlipCenter=true);
bool getHFlip() const;
bool getVFlip() const;
```

上面的函数能设置Sprite的缩放, 旋转角度, 翻转等等属性.

3.2.3 深入SoraSprite

3.2.3.1 贴图数据

在Sora里, 我们还可以直接获取到贴图的原始颜色数据, SoraCore内有

```
uint32* textureLock(HSORATEXTURE);
void textureUnlock(HSORATEXTURE);
```

两个函数

其中textureLock返回目标Texture的原始颜色数据指针, 每个像素为一个32位无符号整数

而textureUnlock则是把textureLock返回的数据写入texture, 所以我们可以通过这个两个函数直接对texture的颜色进行处理, 例如.≤÷CPU运算的各类图形效果

而我们也可以直接通过SoraSprite做到这些事情, SoraSprite内的相应函数为

```
uint32* getPixelData() const;
void unlockPixelData();
```

在Sora/SoraGraphicEffect.h内定义了一些最基础的图形算法, 包括高斯模糊, 灰度图, 反色和Alpha混合. 不过需要注意的是由于他们都是CPU运算, 所以效率很慢, 不建议在帧update的时候调用他们, 那样将导致明显的lag.

3.2.3.2 ImageEffect

在Sora/SoraImageEffect.h内定义了一系列的图形变换效果, 包括

- SoraImageEffectFade 淡入/淡出
- SoraImageEffectScale 放大/缩小
- SoraImageEffectTransitions 坐标变换
- SoraImageEffectColorTransitions 颜色变换
- SoraImageEffectRotation 旋转

除此之外还有个特殊的ImageEffect类 SoraImageEffectList, 这个类本身没有效果, 但是他可以添加别的ImageEffect来创建一个ImageEffect动画队列, 例如先淡入, 然后放大2倍, 之后旋转一圈.

利用这些ImageEffect, 我们可以创造漂亮的图形变换动画效果, 每个Effect都有三种播放模式

- IMAGE_EFFECT_ONCE 播放一次
- IMAGE_EFFECT_REPEAT 重复播放
- IMAGE_EFFECT_PINGPONG 往返播放

而SoraSprite拥有addEffect接口, 你可以对一个Sprite直接加上某个或者多个图形变换效果, 甚至图形变换效果队列.

值得注意的是SoraImageEffect的内存是由SoraSprite管理的, 所以每次addEffect的时候必须用new的方式进行

例如mySprite->addEffect(new sora::SoraImageEffectFade(0.f, 1.f, 1.f, sora::IMAGE_EFFECT_ONCE);

还有需要注意的一点是如果sprite内有effect, 那么在update的时候必须调用mySprite->update(delta_time)函数来update各个effect, 否则effect不会被执行.

* 深入ImageEffect

通常情况下所有ImageEffect的变换都是线性的, 而我们会忽略所有ImageEffect构造函数的最后一个参数. 实际上他们的构造函数的最后一个参数就可以让我们自定义Effect的变换

最后一个参数的类型为CoreTransformer<CoreTransform>*

这是一个提供针对CoreTransform变换的Object, 在CoreTransformer.h内定义.

这个基类定义了唯一的一个接口

T slerp(T& start, T& end, float32 time)

三个参数分别为变换开始值, 结束值, 以及当前时间比例, 取值0.0 - 1.0, ImageEffect通过这个函数获取Effect在任意时刻的变换值

默认情况下, 所有ImageEffect会使用CoreLinearTransformer, 他的返回值很简单,

return start + (end - start) * time;

提供了一個线性的变化.

基于此, 我们可以通过继承自CoreTransformer来创造自定义的变换规则, 例如弹簧或者波浪.

需要注意的是传进ImageEffect的Transformer不会被自动释放, 你需要手动管理他们的内存. 因为大部分情况下Transformer都是通用的(除非类本身包含其余属性)

3.2.3.3 Shader

SoraSprite也拥有添加shader的接口

```
void attachShader(SoraShader*);
void detachShader(SoraShader*);
SoraShader* attachShader(const SoraWString& shaderPath, const
     SoraString& entry, SORA_SHADER_TYPE type);
bool hasShader() const;
void clearShader();
```

其中SoraShader是Sora核心内关于Shader的基类, 实现依赖于SoraRenderSystem. 核心插件内在桌面平台上采用Nvidia的跨平台Shader技术Cg, 在移动平台上则是glsl(需要OpenGLES 2.0+)

Shader相关的具体内容参见Shader一节

3.2.3.4 顶点着色

SoraSprite的setColor和setZ函数都带有两个参数, 第一个为要设置的值, 第二个则是指定的顶点, 默认为-1代表所有顶点. 一个Sprite有4个顶点, 所以第二个参数取值为-1或者0-3.

针对不同的顶点设置不同的颜色将导致sprite的颜色会在不同顶点之间依赖顶点颜色平滑变化

3.2.3.5 渲染模式

SoraSprite的setBlendMode函数能设置Sprite使用的渲染模式, 默认为BLEND_DEFAULT, 这个值由3个值组成, 分别为

BLEND_COLORMUL
BLEND_ALPHABLEND
BLEND_NOZWRITE

设置不同的渲染模式将导致渲染器针对这个sprite的渲染做不同的处理, 如果要开启zbuffer, 必须要设置BlendMode包含BLEND_ZWRITE, BLEND_DEFAULT_Z在BLEND_DEFAULT的基础上做了这个处理

3.2.3.6 自定义顶点渲染

有些时候, 我们希望以自定义的顶点渲染一张贴图, 这时候就可以使用SoraSprite的renderWithVertices函数, 函数原型为

SoraSprite::renderWithVertices(SoraVertex* vertex, uint32 vertexSize, int32 drawMode)

vertex参数是我们定义的顶点们
vertexSize是vertex的数量
drawMode是顶点描画模式, 有

SORA_LINE 每两个顶点为一条直线
SORA_TRIANGLES 每三个顶点为一个三角形
SORA_TRIANGLES_FAN 每两个顶点和第一个顶点组成三角形
SORA_TRIANGLES_STRIP 每个顶点和他下面的两个顶点组成三角形
SORA_QUAD 每4个顶点组成矩形

同时SoraCore内拥有

renderWithVertices(HSORATEXTURE, blendMode, vertex, vertexSize, drawMode)

你也可以通过这个函数直接描画一张贴图

Sora的顶点由SoraVertex描述, 结构体包含

x x坐标
y y坐标
z z坐标
col 顶点颜色
tx 贴图x坐标
ty 贴图y坐标

其中x, y即为屏幕位置, z为深度
tx和ty为贴图坐标, 取值0-1, 注意是以完整的贴图大小为基数计算的

例如

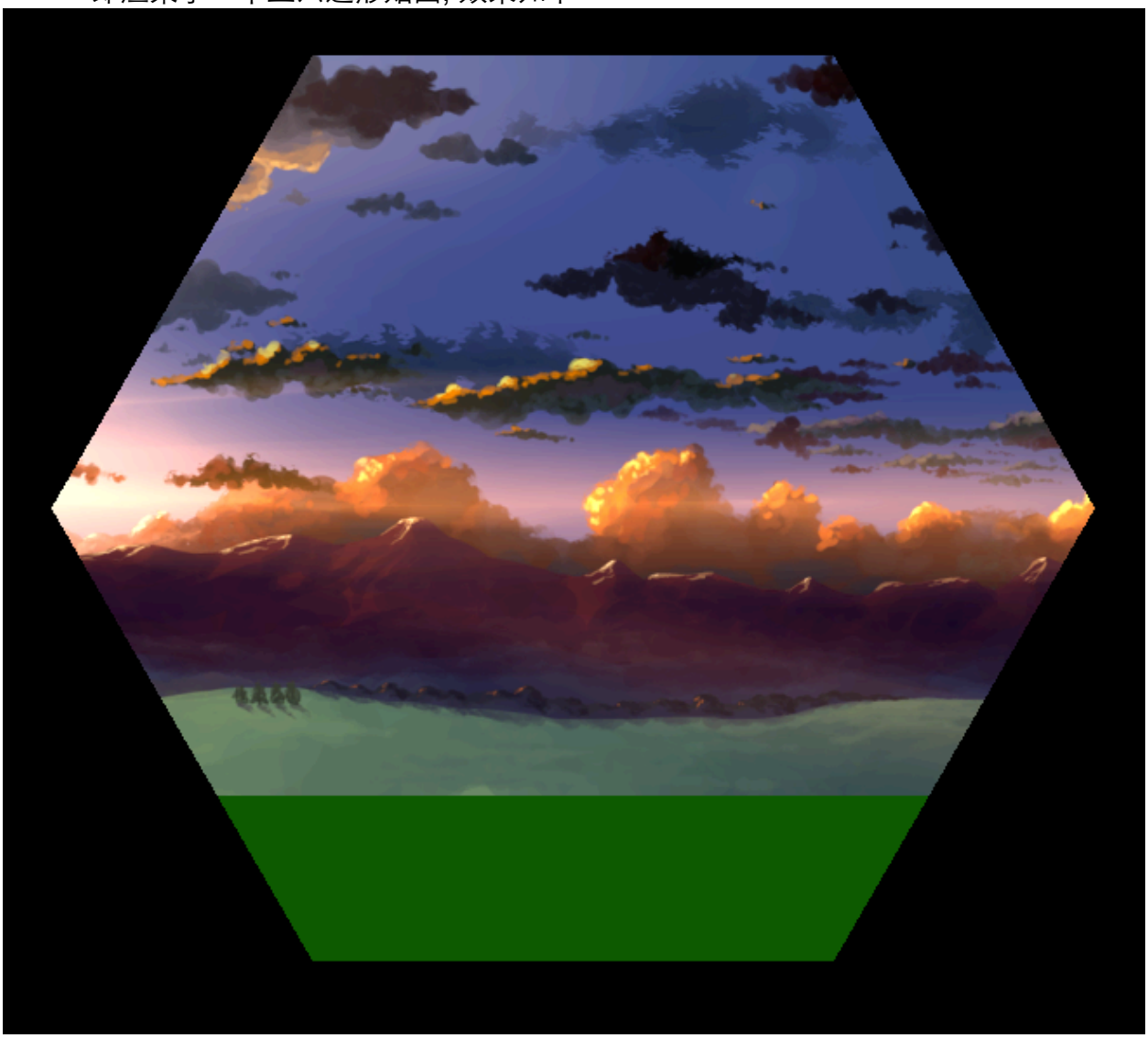
```
float32 px = 400.f, py = 300.f;
for(int i=0; i<6; ++i) {
    vert[i].col = 0xFFFFFFFF;
    vert[i].z = 0.f;

    vert[i].x = px + 300*cosf(sora::DGR_RAD(i*60));
    vert[i].y = py + 300*sinf(sora::DGR_RAD(i*60));

    vert[i].tx = (pSpr->getSpriteWidth() / 2 + 300*cosf(sora::DGR_RAD(i*60))) / pSpr->getTextureWidth(false);
    vert[i].ty = (pSpr->getSpriteHeight() / 2 + 300*sinf(sora::DGR_RAD(i*60))) / pSpr->getTextureHeight(false);
}

pSpr->renderWithVertices(vert, 6, SORA_TRIANGLES_FAN);
```

即渲染了一个正六边形贴图, 效果如下



3.3 Shader

3.3.1 基础

Sora核心通过SoraShader和SoraShaderContext类对Shader提供了原生的支持. 他们的定义位于Sora/SoraShader.h.

Shader(着色器)应用于GPU编程, 可大幅度提高图形处理效率. Sora内的shader分为两类

sora::FRAGMENT_SHADER 片段着色器, 用于调整每个像素的颜色

sora::VERTEX_SHADER 顶点着色器, 用于调整顶点位置, 颜色, 纹理坐标等属性.

由于DirectX和OpenGL对于Shader的定义不同, 为了方便跨平台, Sora在桌面平台上采用了Nvidia的跨平台shader语言 - Cg. 这样在桌面上就不用为了不同的平台而编写不同的shader. (当然在移动平台上(如果是OpenGLES 2.0以上)还是需要重写一遍glsl).

虽然有了跨平台的shader语言, Shader的实现还是依赖于渲染器的. 所以在Sora的Plugins里, SoraHGERenderer和SoraOGLRenderer实际上都依赖于SoraShader插件, 这个插件实现的Cg的opengl和dx部分, 同时在Plugins/cg文件夹内有cg在各个平台的库和头文件.

利用Shader, 我们可以在几乎不影响CPU效率的同时进行复杂的图形操纵, 例如动态高斯模糊, 动态阴影等.

3.3.2 SoraShader和SoraShaderContext

在Sora核心, SoraShader用于表示一个具体的Shader文件, 而SoraShaderContext则用于管理SoraShader并且应用SoraShader.

通常我们在渲染一张贴图前用attachShaderContext应用一个context, 这样在这张贴图渲染的时候, 这个ShaderContext包含的SoraShader的代码都会传送到GPU得到应用, 在贴图渲染完毕后, 我们用detachShaderContext函数移除ShaderContext以不影响下一张被渲染的贴图.

3.3.3 使用

在SoraCore核心内, 有如下4个函数

```
SoraShaderContext* SORACALL createShaderContext();
void SORACALL attachShaderContext(SoraShaderContext* context);
void SORACALL detachShaderContext();
SoraShader* SORACALL createShader(const SoraWString& file, const SoraString& entry, SORA_SHADER_TYPE type);
```

其中createShaderContext创建一个新的ShaderContext

attachShaderContext和detachShaderContext则是应用/移除一个ShaderContext

createShader则是方便快速创建一个shader, 实际上你可以通过SoraShaderContext->createShader或者attachShader达到同样的目的. 当然这两个函数有所不同, createShader只是单纯的利用shader引擎创建一个shader, 并不会添加到ShaderContext自身, 而attachShader则是创建并且添加一个shader到自身.

所有的attachShader都是接收三个参数, 第一个是shader file, 依赖于shader引擎文件写法可能不同. 第二个是shader的入口函数(主函数). 第三个则是Shader的类型, sora::FRAGMENT_SHADER或者sora::VERTEX_SHADER.

同时在SoraSprite内也有一些方便在Sprite上使用Shader的函数. 他们是

```
void attachShader(SoraShader*);
void detachShader(SoraShader*);
SoraShader* attachShader(const SoraWString& shaderPath, const SoraString& entry, SORA_SHADER_TYPE type);
bool hasShader() const;
void clearShader();
```

其中attachShader和detachShader分别是向Sprite内部的ShaderContext添加/移除一个Shader, 而attachShader的另一个版本则是从文件添加一个Shader. hasShader返回这个sprite是否应用了shader, clearShader则是清除这个sprite拥有的Shader. 如果一个Sprite拥有Shader, 那么当他被渲染时, 他会自动attach和detach自身的ShaderContext上核心渲染器, 而不用手动操作.

3.3.4 简单的Cg Fragment Shader写法

以下是一个简单的Cg Fragment Shader的例子

```
struct output {
        float4 color : COLOR;
};

uniform float4 hlColor;

output highlight(float2 texCoord: TEXCOORD0,
                 uniform sampler2D decals: TEX0)
{
        output OUT;
        float4 texcolor = tex2D(decals, texCoord);
            float4 ratecolor = texcolor * hlColor;

        OUT.color = texcolor * 0.2 + ratecolor * 0.8;

        return OUT;
}
```

语法和C语言类似, 其中float4代表4个float组成的数组, 一般用于表示颜色, 你可以用rgba直接访问4个分量, 同理还有float3.xyz, float2.xy等变量类型. highlight是这个shader的主函数, 两个参数分别是当前贴图位置和贴图的sampler. 这个函数返回在当前贴图位置的贴图的颜色.

你可以用tex2D(sampler, coord)来获取贴图在这一点本身的颜色, 然后进行处理, 最后返回.

在这个例子里, 就是把贴图颜色和hlColor进行混合然后返回. hlColor是一个uniform类型的变量, 表示这个变量的初始值是从Shader代码外部设置的. 当变量不是uniform变量的时候, 你可以用类似C语言的初始值设置方式来设置初始值, 例如float4 green = float4(0, 1, 0, 1); 注意在Cg的Fragment Shader内, 颜色的取值都是0.0 - 1.0, texCoord贴图位置的取值也是0.0 - 1.0.

3.3.5 Shader变量的设置

SoraShader拥有如下5个和变量相关的函数

setParameterfv(name, val, size)

设置一个float类型的变量数组, name是变量名字, val是要设置的值的数组, size是数组大小, 例如4在Cg内表示float4.

setParameteriv(name, val, size)
setParameterfv的integer版

getParameterfv(name, val, size)
获取一个float类型的变量数组, name是变量名字, val是缓冲, size是数组大小. 请确保缓冲区的大小大于或者等于size

getParameteriv(name, val, size)
getParameterfv的integer版

setTexture(decalName, tex)
设置一个贴图变量, decalName是变量名字, 例如"decal2", tex是HSORATEXTURE, SoraTexture的Handle, 你必须保证这张Texture有效.在Cg的Fragment Shader内, 你可以通过
uniform sampler2D decal2: TEX1变量来获取这张贴图

有了这些函数, 我们就能在一定程序上操纵Shader内的变量了, 例如在刚才那个例子里, 我们先创建一个shader
SoraShader* myShader = mySprite->attachShader("highlight.fs", "highlight", sora::FRAGMENT_SHADER);

然后我们可以通过
float hlcolor[4] = {1.0, 0.0, 1.0, 1.0};
myShader->setParameterfv("hlColor", hlcolor, 4);
来设置混合颜色的值.

3.3.6 一些Reference
Nvidia CG References & Tutorial
http://www.evernote.com/shard/s45/sh/7a7dc357-59f2-4dc0-83e4-a5fd547d3902/b1c2001a6fccfd61b05fcfe45418107a
HLSL Pixel Shader Effects Tutorial
http://blogs.microsoft.co.il/blogs/tamir/archive/2008/06/17/hlsl-pixel-shader-effects-tutorial.aspx
提供了几个简单的Pixel Shader(Fragment Shader)效果的示例代码, 虽说用的是HLSL, 但是转换一下也不是难事.

Ex: 资源的管理

在调用和资源相关的函数时, SoraCore都会试图从SoraResourceManager内获取资源内容的指针, 所以资源加载严格依赖于SoraResourceManager和load过的资源包

SoraCore内关于资源有如下函数

loadResourcePack(file)　打开一个资源包, 默认视为Folder, 如果有注册过的SoraResourceManager则会试图使用他们打开这个文件

attachResourcePack(pack)　加载一个打开的资源包, 只有执行了这步资源包才算正式被核心加载进去

detachResourcePack()　卸载一个加载了的资源包

getResourceFile(file, ref size)　读取一个资源文件, 返回数据指针, size为资源大小的引用, 如果失败则返回NULL, 注意这个函数不管资源多大都会一次性读取

readResourceFile　读取一个资源文件, size为指定的大小

getResourceFileSize　获取资源文件大小

freeResourceFile　释放资源占用的内存, 注意所有getResourceFile或者readResourceFile返回的指针在使用完之后都必须要通过这个函数施放, 否则将导致问题或者内存泄露

enumFilesInFolder　枚举一个文件夹(资源包文件夹)内的文件, 依赖于ResourceManager的实现

在Sora/SoraFileUtility.h内也定义了一些针对文件方面的函数, 详见API文档

*** 同时除非有特殊说明, 所有Sora插件的文件获取都是基于SoraCore的资源相关API, 所以不用担心资源封装解包等问题. 只要设置好资源相关内容, 只要文件存在且路径正确, 不管是在硬盘上还是在一个压缩包内, 自带的Sora插件都能获取到资源文件的内容. 同时我也建议你在编写自己的插件的时候尽量支持内存读取以方便和SoraCore的资源管理整合.

Event Callback 事件回调

4.1 事件基础类 SoraEvent和SoraEventHandler

4.1.1 注册Event处理函数

在Sora核心内, SoraEvent是所有事件的基类, 位于Sora/SoraEvent.h. 同时SoraEventHandler是所有事件响应器的基类.

在Sora核心, 几乎所有的基础类都是SoraEventHandler的子类, 包括SoraWindowInfoBase, SoraObject等. 所以当你创建了一个主窗口的时候它就已经拥有了处理Event的能力.

SoraEvent的分发是依赖于C++的RTTI机制的, 每个种类的Event都必须拥有自己的处理函数.

在Sora核心, 有如下三种基础的Event事件

SoraKeyEvent	输入事件, 依赖于SoraInput
SoraPlaybackEvent	回放事件, 依赖于SoundSystem或者MoviePlayer
SoraTimerEvent	计时器事件, 依赖于SoraEventManager

那么怎么让一个类能处理Event呢

1. 注册Event处理函数, 通过基类的registerEventFunc函数, 这个函数的原型为

registerEventFunc(T*, void (T::*memFn)(SoraEvent* event))

其中第一个参数为EventHandler本身, 第二个就是EventHandle函数了.

例如, 我们要是我们的主窗口能够处理SoraKeyEvent的话, 只需要在类里面加上

void myOnKeyEvent(const sora::SoraKeyEvent* kev);

函数, 然后在类初始化的时候加上

registerEventFunc(this, &mainWindow::myOnKeyEvent);

即可, 这样别的类就可以通过

mainWindow->handleEvent(KEY_EVENT)

来使mainWindow自动调用myOnKeyEvent函数了.

其余所有类型的Event都是一样的处理方式. 只要我们拥有Event和EventHandler指针, 就可以通过

EventHandler->handleEvent(event)方式来让EventHandler调用自己注册过的某个类型的Event的的处理函数.

4.1.1.2 Consume Event

当你不希望一个Event被继续传递或者希望告知会处理同一个Event的其余组件这个Event已经被使用时, 你应该手动Consume这个Event

SoraEvent::consume函数将设置这个SoraEvent的consumed状态为true,

你可以通过

SoraEvent::isConsumed()函数来获取某个Event是否已经被Consume.

同时你在接收到一个Event之后总是应该先检查他是否已经被Consume以保证自己是第一个处理者, 避免重复处理.

4.1.2 Event的分发

Sora核心内拥有SoraEventManger类, 定义于Sora/SoraEventManager.h. 这个类也是一个Singleton类, 你也可以通过sora::SORA_EVENT_MANAEGER来访问.

SoraEventManager能使你更加方便的处理Event分发, 而不同管EventHandler的具体类型. 这个类的函数如下

registerEvent(eventName, handler, event);
unregisterEvent(eventName);
unregisterEventHandlerFromEvent(eventName, handler);
这三个函数是用来注册EventHandler到EventManager的
registerEvent接收三个参数, 第一个是这个EventHandler群组的名字, 例如”Windows”. 第二个参数是EventHandler本身, 例如mainWindow, 第三个参数则是event, 在这里可以为NULL, 因为Event可能是动态生成的. 使用相同的名字多次调用这个函数则是把多个EventHandler加入到同一个群组内.
第二个和第三个函数则是反注册某个Event群组或者从Event群组里面反注册一个Handler.
在注册完之后我们可以通过
sendMessage(eventName, params, receiver);
sendMessage(eventName, ev);
来分发Event.
第一个sendMessage函数接收三个参数. 第一个是EventHandler群组的名字, 第二个是参数, 多用于脚本, 第三个则是指定某一个handler, 可以忽略, 默认为0, 代表向eventName群组的所有EventHandler发布registerEvent时注册的消息, 如果注册时的Event为0, 则这个函数什么都不会做.
第二个sendMessage则更加灵活. 第一个参数同样是EventHandler群组的名字, 而第二个参数则是一个SoraEvent指针. 你可以通过这个函数向任意群组内的EventHandler发送任意消息.

除了处理基本的EventHandler注册以及Event分发, SoraEventManager还可以处理Timer以及InputEvent.

4.1.3 TimerEvent 计时器事件
4.1.3.1 注册
SoraEventManager内和Timer相关的函数有如下几个
createTimerEvent(handler, time, repeat)
registerTimerEvent(handler, ev, time, repeat)
unregisterEvent(handle)
第一个函数是创建一个timer. 第一个参数是EventHandler, 第二个参数是时间, 第三个参数则是是否重复
第二个函数是注册一个自定义的TimerEvent, 比createTimerEvent函数多了一个参数, 表示自己的TimerEvent, 例如通过继承加上一些自定义属性的TimerEvent, 让Timer拥有更高的灵活性.
第三个函数则是反注册一个handler, 相当于删除一个timer
所有TimerEvent是从注册开始, 依赖于SoraTimer的DeltaTime来计算的. 当时间大于等于注册的时间时, SoraEventManager则会给注册的TimerEvent发送SoraTimerEvent或者自定义的TimerEvent. 所以确保你的类已经注册过TimerEvent处理函数, 确保能够正确收到事件.
下边是一个注册每秒调用一次的Timer的例子

```
// 函数本身
void mainWindow::onTimerEvent(const sora::SoraTimerEvent* tev) {
     // do something s
}

// 初始化
registerEventFunc(this, &mainWindow::onTimerEvent);
sora::SORA_EVENT_MANAGER->createTimerEvent(this, 1.f, true);
```

4.1.3.2 SoraTimerEvent

SoraTimerEvent定义在Sora/SoraTimerEvent.h内, 拥有如下几个函数

getTime()　返回间隔时间(Timer时间)

getTotalTime()　返回自Timer被注册开始的总时间

4.1.4 InputEvent 输入事件

4.1.4.1 注册

SoraEventManger内同样可以注册SoraKeyEvent的handler. 通过如下两个函数

registerInputEventHandler(handler);

unregisterInputEventHandler(handler);

第一个函数注册一个 SoraKeyEvent 的接受者, 第二个函数则是反注册一个接受者.

同时还有个特殊的函数publishInputEvent(SoraKeyEvent* ev)

这个是给SoraInput的子类留的, 因为InputEvent需要他们在接收到输入事件的时候动态创建然后分发. 你也可以通过这个函数模拟输入(如果你的程序内的输入都是依靠InputEvent的的话)

要让类拥有处理SoraKeyEvent的能力和处理SoraTimerEvent类似. 下边是一个给mainWindow加上inputEvent处理的例子.

```
// 函数
void mainWindow::onKeyEvent(const sora::SoraKeyEvent* kev) {
     if(kev->type == SORA_INPUT_KEYDOWN) {
     if(kev->key == SORA_KEY_1 ) {
            // 按下了"1"键
     }
     }
}

// 初始化
registerEventFunc(this, &mainWindow::onKeyEvent);
sora::SORA_EVENT_MANAGER->registerInputEventHandler(this);
```

4.1.4.2 SoraKeyEvent

SoraKeyEvent定义在Sora/SoraKeyInfo.h内. 拥有如下成员/函数

isKeyDown	是否是按键按下事件
isKeyUp	是否是按键抬起事件
getKey	获取按下/抬起的键
isKeyPressed(key)	是否是某个键按下的事件
isKeyUp(key)	是否是某个键抬起的事件

isShiftFlag	shift键是否被按下
isCtrlFlag	ctrl键是否被按下
isAltFlag	alt键是否被按下

type	事件类型(按下/抬起)
key	事件按键
flags	SORA_INPUT_FLAG_*标记
chr	按键的ASCII码
wheel	鼠标滚轮位置
x, y	鼠标位置(鼠标事件)

通过SoraKeyEvent你可以获取到关于输入的一切信息, 然后处理.

4.1.4.3

如果你不希望InputEvent事件在某个处理之后继续向下传递, 那么你应该手动Consume这个Event.

当一个InputEvent被Comsume, 表明他已经完全处理完毕, 不用再传递或者储存到当前帧的InputEvent栈.

4.1.5 SoraPlaybackEvent

SoraPlaybackEvent是一类特殊的Event, 他并不通过SoraEventManager分发, 而是在SoraMusicFile或者SoraMoviePlayer内部使用, 供音乐/电影的播放状态改变使用. 你也可以通过继承SoraPlaybackEventHandler来创建属于自己的PlaybackEventHandler. 通常来说你不需要理会.

插件与FrameListener

在Sora核心, 除了主窗口每帧的回调之外, 还有两类方法监听每帧的事件, 他们是Plugin插件和FrameListener帧监听器.

5.1 Plugins 插件

顾名思义, Plugin就是Sora插件的基础类, 定义在SoraPlugin.h, 由SoraPluginManager管理. 所有的Plugin必须拥有自己独特的名字方便管理. 在SoraCore中有

registerPlugin
unistallPlugin
getPlugin

函数来注册/删除/获取某个插件.

这里的插件并不同于由SoraCore的register*系列函数注册的核心组件, 那些组件并不能由以上函数注册或者获取.

所有注册的插件每帧都会被回调update(deltaTime)函数, 来update自身. 比如一个SoundSystem必须通过这个函数每帧update自己的状态(当然SoundSystem的update是由核心组件的update完成的).

SoraPlugin基类定义了所有Plugin的接口, 包括

install	注册时调用
initialise	初始化时调用(注册之后)
shutdown	被移除时调用
unistall	移除完毕时调用
update	每帧调用
* getName	返回一个独特的插件名字, 例如”PhysicalWorld”

Sora的插件里面, 基于Box2D的SoraBox2D插件内的SoraPhysicalWorld的物理世界update就是通过Plugin机制完成的.

5.2 FrameListener 帧监听器

与Plugins不同, 帧监听器并不依靠于唯一的名字来辨别, 他们的注册和删除只能依靠指针

在SoraCore内有

addFrameListener
delFrameListener

函数来处理FrameListener的添加和删除

FrameListener同样每帧都会被回调固定的函数, 但是和Plugin不同的是FrameListener每帧被回调两次, 一次在帧开始, 主窗口update之前, 一次在帧末尾, 主窗口渲染之后, 用于处理对于回调位置要求比较高的情况.

FrameListener的基类是SoraFrameListener, 定义在Sora/SoraFrameListener.h. 子类可以实现的接口有

onFrameStart	帧开始
onFrameEnd	帧结束

6 Depth Buffer和Z Sort

这部分和Sora核心关系不大, 而是和图形渲染相关. 通常情况下, 如果打开DepthBuffer, 渲染带有透明像素的贴图的时候必须特别小心, 必须从最后渲染到最前按顺序渲染, 不然可能会由贴图被裁剪掉, 颜色完全不会被写入.

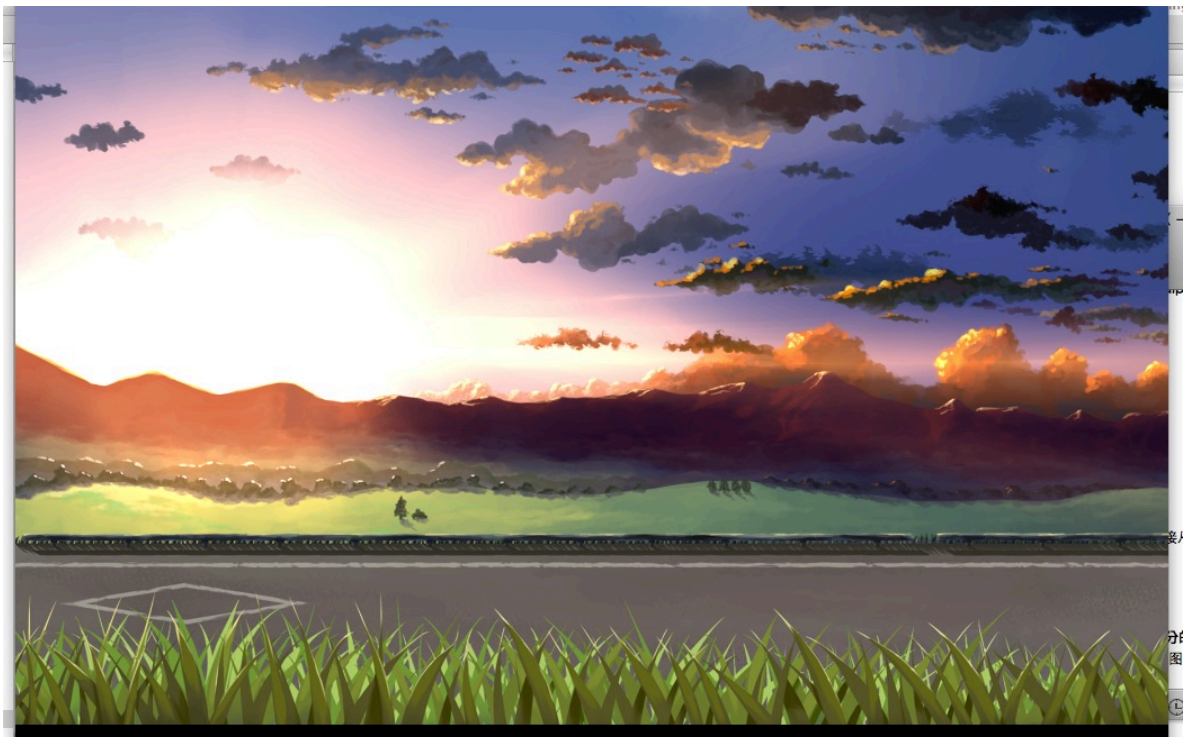
但是在很多情况下, 我们并不能用代码顺序确定贴图的渲染顺序, 而是希望通过z排序来进行. 这里就产生了矛盾. SoraCore内的beginZBufferSort和endZBufferSort函数就是为了这种情况而生的.

在SoraSprite渲染的时候, 你可以通过设置BlendMode内含有BLEND_ZWRITE来打开depthbuffer渲染. 同时通过setZ函数设置sprite的z深度值, 取值范围0.0 - 1.0

如果你有几张带有透明像素的贴图而你有希望他们能按照z的大小排序, 你可以在开始他们的渲染之前调用SoraCore::beginZBufferSort函数, 然后在渲染完毕之后, endScene之前调用endZBufferSort函数. 这样这几张贴图无论代码顺序如何, 都会按照他们的z值从小到大排序然后渲染.

当然这两个函数也有限制, 就是因为考虑到内存和效率, z值只能精确到0.001. 不过大部分情况下都够用了.

例如我们的背景由三个部分组成, 背景 - 路 - 草, 他们的渲染是依靠于z深度的. 如果我们代码也按照 背景 - 路 - 草 的顺序渲染, 那么场景不会有任何问题.



如果我们代码按照 路 - 草 - 背景 的顺序渲染, 则场景会出线不正确的透明像素裁剪.

但是如果我们在渲染代码之前和之后加上beginZBufferSort和endZBufferSort, 则无论代码顺序如何, 我们总能够依靠z得到正确的结果. (下图背景用了下shader)

基础组件和辅助功能

7.1 stringid

在Sora核心, stringId是一个字符串的唯一id, 基于crc32 hash, 用于快速名字比较.

stringid可以通过

str2id(string/wstring)

宏来获取stringId. 默认情况下, 所有被stringId话的string都会被cache方便重新获取, 如果你不需要cache, 则可以通过

str2idnc(string/wstirng)

宏来获取

对于cache过的字符串, 可以通过宏

id2str

id2strw

来获取对应的string或者wstring

7.2 AutoListElement

这个类是Sora基类之一, 所有改类的某一类型的子类都会在构造的时候自动被加入到一个链表, 你可以通过任意一个子类的members成员来获取到这个链表.

例如

myClass::Members myMembers = myClass::members;

7.3 Serializable

这个基类是SoraObject的基类之一, 表示一个物体是否可以被序列化和提供同一的序列化接口

7.4 SoraAutoPtr

这个类是Sora核心的智能指针类, 该类的行为类似于指针但是提供了引用计数特性以保证物体在被多个智能指针共享的时候的安全.

7.5 StringConvert

SoraStringConv.h定义了string和wstring之类的转换函数ws2s和s2ws, 为了效率和效果, 在各个平台的实现并不一致.

7.6 SoraFileUtitliy

这个静态类提供了一些和文件相关的常用函数

fileExists	判断某个路径的文件是否存在
getAbsolutePath	获取某个文件的绝对路径
getWrittablePath	获取可以写入文件的路径(在移动平台上表现不同)
getApplicationPath	获取程序自身所在的路径
getFontPath	获取操纵系统字体文件夹所在位置
getFileName	获取某个完整路径的文件名
isFont	判断某个文件是否为字体

7.7 打开文件与保存文件对话框

SoraCore提供了

fileOpenDiaglog(filter, path)

fileSaveDialog(filter, path, ext)

函数, 用于打开一个打开(保存)文件的对话框. 此对话框的实现依赖于操纵系统. 他们的参数都可以忽略. 其中第一个和第二个参数分别表述过滤器和默认路径. 而fileSaveDialog的第三个参数表示默认的保存扩展名(在用户没有输入的情况下, windows下有效)

注意不同平台下, 这两个函数的表现并不相同, filter的写法也不相同. 在OS X下, filter是用";"分割的扩展名, 例如txt;doc, 而在windows下, 则比较复杂, 书写规则为

"描述\0扩展名\0\0"

例如

"魔兽争霸3地图文件(*.w3x, *.w3m);*.w3x;*.w3m\0\0"

7.8 占用的内存

SoraCore内提供了getEngineMemoryUsage函数来获取当前引擎消耗的内存, linux下需要libproc库的支持.

7.9 RGBA颜色

SoraColorRGBA类提供了操纵RGBA颜色相关的功能. 例如SoraColorRGBA::getHWColor就可以把自身转换为32位无符号整数描述的颜色. SoraColorRGBA内部的颜色表示为4个浮点数, 范围0.0 - 1.0

7.10 EnvValues

定义于SoraEnvValues.h内的SoraEnvValues类提供了一个全局的数值保存机制. 你可以通过SET_ENV或者SET_ENV_(TYPE)宏来保存一个数值, 通过GET_ENV或者GET_ENV_(TYPE)宏来获取一个数值. 支持bool, int, float, stringId和void*指针. 同时Sora内部也会用此机制储存一些相关数值, 例如SoraCore会设置CORE_SCREEN_WIDTH和CORE_SCREEN_HEIGHT为主窗口的大小. 所以在创建主窗口后你也可以通过GET_ENV_INT("CORE_SCREEN_WIDTH(HEIGHT)")来获取主窗口大小.

7.11 Exception 异常机制

Sora核心在大部分情况下都不会抛出异常, 而是通过不正确的返回值通知用户失败.

Sora的异常类是SoraException, 定义在SoraException.h, 你可以通过SoraException::what函数来获取具体异常发生的信息, 函数, 代码文件和代码行数描述. 如果要抛出一个异常, 则可以使用SORA_EXCEPTION(mssg)宏. 例如throw SORA_EXCEPTION("A exception happened : (");

7.12 向量, 矩形和四元數

Sora核心的向量和矩形使用的是hgeVector和hgeRect, 你也可以通過SoraVector或者SoraRect访问到. 同时vector3.h内定义了vector3类, 在quaternion.h内定义了quaternion四元数类. rect4v.h内的rect4v则是个任意矩形, 可以通过hgeRect旋转生成.

7.13 ini, json和xml

Sora核心内已经包含了ini, json和xml三种常用的数据格式文件的解析库, 分别是SoraINIFile, JsonCpp和TinyXML, 你可以通过Sora/SoraINIFile/, Sora/json/, sora/tinyXML/之内的头文件访问到他们

7.14 lexer

Sora核心内还包含了一个词法解析器，支持自定义operators. 位于Sora/lexer，类名为llexer

7.15 SoraCanvas

SoraCanvas是针对RenderTarget的一个封装，内置SoraSprite并且提供添加Shader，ImageEffect等功能. 使用上只需在渲染之前用beginRender取代beginScene，在渲染完成之后调用finishRender取代endScene就可以了.

然后可以用render和update函数渲染和update自身.

getCanvasSprite可以返回Canvas的SoraSprite

而添加shader和ImageEffect相关的函数和SoraSprite的使用规则一样.

7.16 SoraPNGOps

7.16.1 SoraPNGOptimizer

SoraPNGOps插件内的SoraPNGOptimizer类提供了png贴图对于浮点坐标的优化功能，可以使贴图在旋转或者缩放时拥有更好的显示效果. 用于素材处理.

他有如下几个函数

optimizePNGTex

优化一张贴图，接收HSORATEXTURE作为参数，贴图数据会被改变

optimizePNGData

优化一个贴图的颜色数据，接收数据指针，宽，高以及真实宽

optimizePNGFromAndWriteToFile

优化一个PNG文件并且写入另一个文件

optimizeTexAndWriteToFile

优化一张贴图并且写入文件

writeTexToFile

直接把一张贴图写入文件

7.16.2 PNGWritter

这个头文件提供了基于libpng的png写入函数

```
static bool writePNGWithPitch(FILE* fp, uint32* bits, int32 width, int32 height, int32 pitch, bool setbgr=false, bool needAlpha=true)
```

其中fp为要写入的文件的指针，bits为数据指针，width和height为宽高，pitch为真实宽，setbgr是是否翻转Endian表示(rgb->bgr)，needAlpha是是否使用alpha通道.

7.16.3 SoraCompressedTexture

这是一类特殊的Texture，用于压缩不同的Texture到同一张大的贴图上来节省空间和方便资源管理. 用于素材处理.

他拥有如下函数

addSprite(SoraSprite*, std::string& descriptor)

往texture添加一个sprite, 只有texturerect内的数据会被写入, descriptor为描述字符串, 用于在读取的时候获取Sprite

delSprite(descriptor)

移除一个已添加的sprite
getSprite(descriptor)
获取基于新tex的sprite, 必须compress或者load过后才有效, 内存由CompressedTexture管理
compress()
压缩已经添加的sprite到贴图
compressAndWriteToFile(file)
压缩并且写入文件名为file的png图像, 同时会生成一个同名的.pngdes文件, 用于描述各个sprite在贴图的位置
loadCompressedFile(file)
加载一个pngdes文件, 会依靠文件信息重建添加过的sprite, 之后可以用getSprite(descriptor)来获取各个sprite

7.16.4 RectPlacement

这个是SoraCompressedTexture的基础, 用于寻找最优化的Rect不重合放置方案.

7.17 SoraSprietAnimation

这是一个为SoraSprite提供逐帧动画的插件, 类名为SoraSpriteAnimation, 以SoraSprite为基类. SoraSpriteAnimation的定义文件有两种, 一种是人工编写的, 一种是压缩过后供引擎使用的. 一般来说, 要定义一个Animation我们先要手动编写这个Animation的属性, 然后通过SoraAnimationEditor查看编辑调整这个Animation, 然后保存为二进制文件. 然后在引擎内, 我们可以通过UNPACK_ANIMATION或者UNPACK_ANIMATION_IN_MEM宏解压二进制文件, 生成一个SoraSpriteAnimation, 然后就可以像使用SoraSprite一样, 使用SpriteAnimation了. 同时SoraSpriteAnimation添加了以下函数.

play()
播放默认动画
playEx(name, loop, queue)
播放一个名字为name的动画, loop是是否循环播放, queue是是否排到动画序列, 如果是false, 则目标动画会立即开始播放, 否则将依照队列顺序播放
pause()
暂停动画播放
resume()
恢复动画播放
setDefaultAnimation(name)
设置默认动画
getAnimationSize
获取动画总数
getCurrAnimationIndex
获取当前动画的Index
isPlaying
动画是否在播放
getAnimationName(index)
获取Index为index的动画的名字

getCurrentAnimationName()
获取当前正在播放的的动画的名字
setAnchor(anchor)
设置sprite的中心点，有五个可用enum参数
ANCHOR_UPPER_LEFT 左上
ANCHOR_UPPER_RIGHT 右上
ANCHOR_LOWER_RIGHT 左下
ANCHOR_LOWER_LEFT 右下
ANCHOR_MIDDLE 中心

注意SoraSpriteAnimation的播放依赖于update函数传进来的dt，你可以通过加大或者减小dt的方式来动态控制播放速率.

7.18 物理和SoraPhysicalObject

Sora的Plugins里提供了基于Box2D的物理类，其中SoraPhysicalWorld是对于b2World的封装，而SoraPhysicalWorld是针对b2Body的封装. Box2D的具体使用请参照Box2D的文档.

SoraPhysicalWorld是一个Singleton类，在构造的时候就会注册自身到SoraCore的Plugin List内，所以你不用手动每帧去调用box2D的iterateWorld函数. 同时SoraPhysicalWorld内还有一些帮助快速转换真实坐标和box2D坐标(通过Scale)的函数以及快速生成一些常见形状，如box, circle, edge, polygen的函数.

SoraPhysicalObject则封装了b2Body，继承自SoraObject，每帧他的位置都会被自动设置为b2Body的位置，简化简单的处理.

7.19 SoraSoundManager

SoraSoundManager插件提供了SoraMusicFile和SoraSoundEffectFile的管理功能. 分为SoraBGMManager和SoraSoundEffectManager.

7.19.1 SoraBGMManager

这个类是为了统一管理背景音乐而存在，拥有如下函数

play(bgmPath, addToQueue)
播放一个bgm文件，bgmPath为文件路径，addToQueue为是否加入bgm队列(循环bgm etc)，如果不加入队列，则bgm会立刻开始播放

stop(stopAtEnd)
停止当前bgm，stopAtEnd为是否播放完再停止

playBGS(bgsPath, bgsid, looptime, volumeScale, bgmVolumeScale)
播放一个背景声音文件，bgsPath为文件路径，id为指定的用户指定的id(可以同时播放n个背景声音文件)，looptime为循环次数，volumeScale为这个bgs的音量所占当前背景音乐音量的比例，bgmVolumeScale是在播放这个bgs的时候调整bgm的音量为当前音量的百分比的比例

注意这个函数返回bgsid, 如果用户指定的id没有被占用, 则返回的id和参数id相同, 否则返回的id将是最近的一个可用id

adjustBGSVolume(bgsid, volume)
调整id为bgsid的bgs的音量为volume

stopBGS(bgsid)
停止id为bgsid的bgs

pause()
暂停背景音乐播放

resume()
恢复背景音乐播放

freeALLBGMs()
释放所有的bgm, 释放内存

toNextBGM()
跳到bgmQueue的下一首音乐

toPrevBGM()
跳到bgmQueue的前一首音乐

setVolume(volume)
设置音量, 取值0.0 - 100.0

getVolume()
获取音量

setPitch(pitch)
设置pitch, 取值0.5 - 2.0

getPitch()
获取pitch

setPan(float32 pan)
设置pan, 取值 -1.0 - 1.0

getPan()
获取pan

setFadeTime(fadeInTime, fadeOutTime)
设置bgm在切换时的淡入淡出时间. 在bgm队列中的bgm切换时前一首将以fadeOutTime淡出, 后一首将以fadeInTime淡入

getFadeInTime()
获取淡入时间

getFadeOutTime()
获取淡出时间

enableRandomBGMQueuePlay(flag)
如果flag为true, 则每次toNextBGM, toPrevBGM, 或者bgm队列中bgm切换时会切换到bgm队列中的随机一首bgm

isRandomBGMQueuePlayEnabled()
是否允许随机bgm播放

7.19.2 SoraSoundEffectManager

这个类是为了方便管理SoundEffect, 有如下函数

load(effectPath, eid)
加载一个effect, 这个函数返回effect id, effectPath为SoundEffect路径, sid为用户指定的id, 如果id已经存在, 则返回一个最近可用的id

unload(effectid)
卸载一个id为effectid的effect

get(effectId)
获取一个id为effectId的effect的SoraSoundEffectFile的指针

play(effectid)
播放一个id为effectid的effect

stop(effectid)
停止播放一个id为effectid的effect

playDirect(effectName)
直接播放一个路径为effectName的effect, SoraSoundEffectManager会加载并且缓冲他, 下次调用playDirect的时候会先试图从缓冲中读取

setVolume(vol)
设置全局音量

getVolume()
获取全局音量

7.20 SoraParticleSystem

SoraParticleSystem是Sora的粒子插件, 你可以通过SoraParticleEditor来编辑粒子, 然后使用SoraParticleSystem来播放.

7.20.1 Effector

你可以通过继承SoraParticleEffector来对粒子施加影响. 每帧每个粒子都会回调Effector的Effect函数.

7.20.2 3D粒子旋转

SoraParticleSystem支持3d旋转, 按z缩放. 你可以通过rotate(roll, pitch, yaw)来进行3D旋转.

7.21 libvlc和Movie播放

SoraMoviePlayer是Sora播放视频的基类. Sora的Plugins内的libvlc插件提供了基于libvlc的跨平台视频播放插件. 你可以通过MoviePlayer的frameChanged函数查看视频是否改变了当前帧, 通过getPixelData函数获取当前视频帧的原始颜色信息, 然后通过SoraSprite的getPixelData和unlockPixelData把获取的视频帧信息写入贴图, 然后渲染. 在获取完数据之后, 你需要手动调用moviePlayer的setFinish函数来标记moviePlayer的frameChanged标记为false. 需要注意的是openMedia函数的format参数, 因为颜色格式的不同, 在opengl下, 这个参数需要是RGBA(默认值), 而在hge+dx的环境下, 需要是RV32. 否则getPixelData将返回对于当前渲染器不正确的颜色数据. 下面是一个播放视频的例子

```
if(moviePlayer->openMedia(L"./AMV_Scenario.mp4")) {
    pSpr = new sora::SoraSprite(sora::SORA->createTextureWH(moviePlayer->getWidth(), moviePlayer->getHeight()));
    moviePlayer->play();
};

 if(moviePlayer->isPlaying()) {
     if(moviePlayer->frameChanged()) {
         ulong32* data = pSpr->getPixelData();
         if(data) {
             void* movieData = moviePlayer->getPixelData();
             if(movieData) {
                 if(moviePlayer->getWidth() != 0) {
                     memcpy(data, movieData, moviePlayer->getWidth()*moviePlayer->getHeight()*4);
                 }
             }
         }
         pSpr->unlockPixelData();
         moviePlayer->setFinish();
     }
 }
```

RTTI Reflection RTTI反射

8.1 RTTI反射基础

RTTI(Runtime type identification)运行时类型识别是C++的机制之一. 例如你可以通过dynamic_cast来转换父类指针到子类指针, 这里面就用到了RTTI, 因为转换不一定正确, 必须使用附加信息来判断要转换的目标是否真正为父类指针的子类. 又如你在运行时可以通过typeid来获取一个类的独一无二的名字(依赖于编译器实现).

但是不同于C#, Java等语言, C++并没有RTTI反射机制, 也就是我们在运行时可以通过字符串访问到类的函数或者成员变量. 我们也没法在运行时知道一个类究竟拥有一些什么函数和什么成员变量或者单纯的通过类名来构造一个类. 例如

```
myClass = ClassManager.newClass("myClass");
```

但是在有些时候, 游戏引擎又需要RTTI反射机制, 尤其提现在脚本上. 例如我们做一个编辑器的时候需要显示一个Object的所有属性并且允许修改. 如果这个类是一个C++类且没有RTTI机制, 我们就只能hardcode了.

Sora核心内实现了一个C++的类成员反射机制, 位于Sora/RTTI文件夹内. 在需要的时候, 你可以通过一些宏来描述一个类, 以在运行时通过字符串来获取这个类的一些信息, 例如成员变量, 成员函数, 父类等.

SoraRTTI是与核心相对独立的实现, Sora核心部分暂时尚未整合RTTI反射机制.

8.2 描述一个Class

SoraRTTI的头文件在Sora/SoraRTTI/SoraRTTI.h内, 里面包含了其余的头文件和定义了用于描述和注册一个Class的一些宏.

要使一个Class拥有RTTI描述, 你首先要在Class内部描述用DESCRIBLE_CLASS宏描述Class, 然后在全局空间用REGISTER_CLASS注册Class到SoraRTTIClassManager.

8.2.1 DESCRIBE_CLASS

以下是一个描述一个叫做myClass的类的例子.

```
class myClass {
public:
    myClass();
    myClass(int i);
    myClass(float j);
    myClass(int i, float j);

    int geti() const {return mi;}
    int getj() const {return mj;}

    void seti(int i) { mi = i; }
    void setj(float j) { mj = j; }

    void print(int i) { std::cout<<i; }
    void print(float j) { std::cout <<j; }

    DESCRIBE_CLASS(myClass,

                   (RTTI_DESCRIBED_FIELD(mi, int, sora::RTTI_FLAG_PRIVATE),
                    RTTI_FIELD(mj, float, sora::RTTI_FLAG_PRIVATE)),

                   (RTTI_METHOD(geti, sora::RTTI_FLAG_PUBLIC),
                    RTTI_METHOD(getj, sora::RTTI_FLAG_PUBLIC),
                    RTTI_METHOD(seti, sora::RTTI_FLAG_PUBLIC),
                    RTTI_METHOD(setj, sora::RTTI_FLAG_PUBLIC),
                    RTTI_OVERLOAD_METHOD_1(print, void, int, sora::RTTI_FLAG_PUBLIC),
                    RTTI_OVERLOAD_METHOD_1(print, void, float, sora::RTTI_FLAG_PUBLIC)),

                   RTTI_NO_BASE_CLASS,

                   (RTTI_DEFAULT_CONSTRUCTOR(),
                    RTTI_CONSTRUCTOR_1(int),
                    RTTI_CONSTRUCTOR_1(float),
                    RTTI_CONSTRUCTOR_2(int, float)));

    DESCRIBE_FIELD(mi, int);

private:
    int mi;
    float mj;
};
```

如代码所示，这个myClass拥有四个构造函数，两个成员变量，一套set/get函数和重载的print函数。要用RTTI描述这个Class我们只需要使用DESCRIBE_CLASS宏，这个宏接收五个参数，分别是

类名
成员变量链表
成员函数链表
基类链表
构造函数链表

多个成员变量/成员函数/基类/构造函数用”,” operator链接.
除了类名以外，其余的描述都必须通过对应的宏来进行.

8.2.1.1 RTTI_FIELD, RTTI_DESCRIBED_FIELD RTTI_NO_FIELD

这个宏用于描述一个成员变量，但是通过这个宏描述的变量不可以动态获取或者设置。如果要允许动态设置成员变量，则必须使用RTTI_DESCRIBED_FIELD，但是RTTI_DESCRIBED_FIELD宏必须搭配DESCRIBE_FIELD宏使用。DESCRIBE_FIELD宏也必须在类定义域内，接收两个参数，一个是变量，一个是变量类型。而

RTTI_FIELD以及RTTI_DESCRIBED_FILED宏都接收3个参数，变量，变量类型和RTTI作用域flag，例如RTTI_FLAG_PUBLIC. 如果不需要描述成员变量，则用RTTI_NO_FIELD占位.

相应的RTTI_PTR, RTTI_DESCRIBED_PTR, RTTI_ARRAY, RTTI_DESCRIBE_ARRAY分别针对指针类型的成员变量和数组类型的成员变量.

8.2.1.2 RTTI_METHOD RTTI_OVERLOAD_METHOD RTTI_NO_METHOD

这两个宏用于描述一个类成员函数，第一个用于描述普通函数，接收两个参数，一个是函数，一个是flag，含义和RTTI_FIELD的flag一样.

RTTI_OVERLOAD_METHOD_(param number)则是用于描述重载函数. 例如RTTI_OVERLOAD_METHOD_1则是描述拥有一个参数的重载函数. 这个宏接收的参数数目为3+函数参数数目，例如RTTI_OVERLOAD_METHOD_1接收4个参数. 第一个和第二个参数接收函数和返回值类型，参数3到(3+函数参数数目-1)为各个参数的类型，最后一个依旧是flag. 如果不需要描述Method则用RTTI_NO_METHOD占位.

8.2.1.3 RTTI_NO_BASE_CLASS RTTI_BASE_CLASS

如果类没有父类或者不需要描述父类，则用RTTI_NO_BASE_CLASS占位. 如果有父类，则用RTTI_BASE_CLASS宏. 宏接收2个参数，一个是父类，一个是flag.

例如RTTI_BASE_CLASS(myBaseClass, sora::RTTI_FLAG_PUBLIC).

多个父类之间同样通过”,” operator链接.

8.2.1.4 RTTI_DEFAULT_CONSTRUCTOR RTTI_CONSTRUCTOR RTTI_NO_CONSTRUCTOR

这三类宏用于描述类的构造函数，RTTI_DEFAULT_CONSTRUCTOR表示类拥有默认构造函数，RTTI_CONSTRUCTOR_(constructor param number)描述拥有n个参数的构造函数，宏接收n个参数，分别为构造函数各个参数的类型. 而RTTI_NO_CONSTRUCTOR则用来占位.

8.2.2 REGISTER_CLASS

这个宏是把描述了的类注册到RTTIClassManager，必须存在于全局空间. 接收三个参数，一个是类名，第二个是namespace名，第三个则是flag.

8.3 获取Class信息

描述和注册Class之后，我们就可以通过SoraRTTIClassManager来获取Class的Descriptor类或者构造任意注册过的类了.

SoraRTTIClassManager拥有下边两个运行时函数

```
SoraRTTIClassDescriptor* findClass(const std::string& className);

void* constructClass(const std::string& className, void** params, unsigned int paramSize, const char* signature);
```

findClass返回一个名字为className的ClassDescriptor.

constructClass从className的构造函数构造一个新类，第二个参数是函数参数，第三个参数是函数参数数目，第四个则是应对重载函数的签名，可以为空.

* RTTI函数调用的参数传递

SoraRTTI的类函数调用参数采用void**传递，这是一个参数数组，例如

```
void** params;
params[0] = &myInt;
params[1] = &myFloat;
```

则是一个有两个参数的参数数组，第一个参数为一个int，第二个参数为一个float.

* RTTI重载函数的签名

对于重载的构造函数或者成员函数，我们需要一个特别的字符串签名来辨别他们. 这个签名为

```
(paramtype1, paramtype2, paramtype3, ...)
```

例如在上面的myClass例子里，如果我们要用myClass(int)构造函数构造myClass，则需要传递”(int)”字符串给constructClass的第四个参数. 注意第四个参数只有在有相同参数数目的函数重载的时候才有用. 对于参数数目不同的函数SoraRTTI会依据参数数目区别. 如果是用构造默认函数，则参数数组也可以为NULL.

8.3.1 SoraRTTIClassDescriptor

这个类是RTTIClass的描述类，每个由SoraRTTI描述的类皆有一个.

```
std::string getDecl();
virtual std::string getName();

std::string getNamespace() const;
std::string getDescription() const;

void* constructClass(void** params, unsigned int paramsize, const char* signature=NULL);

METHOD_LIST&        getMethods();
FIELD_LIST&         getFields();
BASE_CLASS_LIST&    getBaseClasses();
CONSTRUCTOR_LIST&   getConstructors();

SoraRTTIMethodDescriptor*       findMethod(const std::string& name);
SoraRTTIFieldDescriptor*        findField(const std::string& name);
SoraRTTIClassDescriptor*        findBaseClass(const std::string& name);
SoraRTTIConstructorDescriptor*  findConstructor(const std::string& name);
```

SoraRTTIClassDescriptor拥有如上运行时函数.

getDecl　返回类的定义，输出例如

```
class myClass {
// more class infos
};
```

信息

getName　返回类名

getNamespace　返回命名空间

getDescriptor　返回类描述(自定义)

constructClass 和SoraRTTIClassManager的含义相同

getMethods　返回类的成员函数链表

getFields　返回类的成员变量链表

getBaseClasses 返回类的基类链表
getConstructors 返回类的构造函数链表
findMethod   返回某个特定的成员函数
findField 返回某个特定的函数成员
findBaseClass   返回某个特定的基类
findConstructor 返回某个特定的构造函数

8.3.2 SoraRTTIMethodDescriptor

这个是SoraRTTI的成员函数描述类.

```
std::string getDecl() const;
std::string getName() const;

SoraRTTIType* invoke(void* obj, void* params[]);
SoraRTTIType* invoke(void* obj, const std::string& params);

SoraRTTIClassDescriptor* getClass() const;
```

getDecl   返回函数定义, 例如void geti(int)
getName   返回函数名字, 例如geti
invoke   调用函数, obj为类的实例, params和参照RTTI参数传递
getClass 返回函数所属的类

8.3.3 SoraRTTIFieldDescriptor

这个是SoraRTTI的成员变量描述类

```
void set(void* obj, SoraRTTIType* var);
SoraRTTIType* get(void* obj);

int getFlags() const;

std::string getName() const;
std::string getDecl() const;

SoraRTTIType* getType() const;

SoraRTTIClassDescriptor* getDescriptorClass() const;
```

set 设置变量的值, 必须是用DESCRIBE_FIELD宏描述过的变量
get 获取变量的值, 必须是用DESCRIBE_FIELD宏描述过的变量
getFlags 获取变量的flag
getName   获取变量名字, 例如mi
getDecl   获取变量定义, 例如int mi;
getType   获取变量类型
getDescriptorClass 获取变量所属的类

其中Set和Get函数需要是用DESCRIBE_FIELD描述过的变量, 同时用SoraRTTIVar设置/获取. SoraRTTIVar是SoraRTTIType的子类之一, 是一个模板类, 所以参数类型上是SoraRTTIType. 同时可以采用sora::SoraRTTIFieldGet模板函数把SoraRTTIVar转换为某个类型. 这个函数接收两个参数, 一个是SoraRTTIFieldDescriptor, 一个是类实例. 例如

```
sora::SoraRTTIFieldDescriptor* iDes = fooDes->findField("i");

if(iDes != NULL) {
    try {
        int i = sora::SoraRTTIFieldGet<int>(iDes, &foo);
        std::cout<<"Foo::i = "<<i<<std::endl;
    } catch(sora::SoraRTTIException& excp) {
        std::cout<<excp.get()<<std::endl;
    }
}
```

同时还有SoraRTTIFieldSet模板函数用于方便设置类变量. 这个函数接收3个参数, 第一个为SoraRTTIFieldDescriptor, 第二个为类实例, 第三个为变量类型.

* SoraRTTIException

因为在RTTI内, 变量类型都是通过SoraRTTIVar<T>来转换, 所以当转换失败的时候, SoraRTTI会抛出SoraRTTIException异常.

what函数则是返回具体信息字符串.

8.3.4 SoraRTTIVar

这是SoraRTTI用于表示变量的类, 相比void*保留了一定的类型信息, 相对更加安全. 可以用set或者get来设置/获取复制的储存的值.

8.3.5 SoraRTTIConstructorDescriptor

这个是类构造函数的描述类

```
void* operator()(void** params);

int getParamNum() const;

std::string getSignature() const;
```

operator()	调用构造函数
getParamNum	获取构造函数参数数目
getSignature	获取构造函数的签名

8.3.6 RTTI函数的调用

例如

```
sora::SoraRTTIMethodDescriptor* fooFn = fooDes->findMethod("bar");
if(fooFn != NULL) {
    int retVal;
    void* params[1];
    int param = 5;
    params[0] = &param;
    retVal = sora::ReturnValToVar<int>(fooFn->invoke(&foo, params));
    std::cout<<"calling Foo::bar with i=5, @retval = "<<retVal<<std::endl;
}
```

就调用了foo的bar函数, invoke函数返回的同样是SoraRTTIVar, 可以通过sora::ReturnValToVar模板函数把返回值转换为某个类型.

Guichan和GUI

这部分不会详细介绍Guichan的使用, 详细的Guichan使用文档请参见Guichan的文档.

Sora的GUI基于Guichan的修改版, 位于Plugins/SoraGUIChan. 由于原版Guichan缺少一些机制, Sora更改了Guichan的核心部分, 并且在外围也加上了一些支持机制.

9.1 SoraGUIChan的初始化: gcnInitializer

为了方便SoraGUIChan的初始化, SoraGUIChan定义了gcnInitializer类, 位于SoraGUIChan/guichansetup.h内. 这个类提供了和Guichan相关的一些常用功能.

initGUIChan(font, size)
初始化Guichan, 向Guichan注册SoraGUIChan的各个组件, 接收两个参数, 第一个是font, 第二个是fontsize, 可以为NULL, 但是guichan将无法使用globalfont渲染文字.

createTop()
在initGUIChan的最后会被自动调用, 创建guichan的top widget, 无需手动调用(调用了也没效果)

getTop()
获取guichan的top widget

findWidet(sid)
查找Widget id为sid的Widget, 依赖于top widget

addWidget(widget, parentId)
往guichan内加入一个widget, parentId为他的父控件id, 可以为0. 在parentId为0或者为"top"的时候, widget会设置top widget为自己的父控件, 否则设置findWidget(parentId)的结果为自己的父控件

removeWidget(widget)
从guichan内移除一个widget, 依赖于widget的父控件

getWidgetAt(x, y, widgetId)
获取位于(x,y)坐标的widget, widgetId为指定获取某个id为WidgetId在(x,y)处的子控件. 可以忽略.

gcnLogic
执行guichan的logic操作

gcnDraw
描绘guichan

getGui

返回指向guichan的Gui类的指针. (gcnInitializer所依赖的)

9.2 扩展的Guichan核心机制

SoraGUIChan内的guichan基于guichan 0.8.2, 扩展了modifier和sound支持.

9.2.1 Sound和SoundLoader

Sound在扩展的Guichan内代表一个声音, 而SoundLoader代表Sound加载器, 在所有的Sound内都有一个静态的全局SoundLoader指针存在. 你可以通过gcn::Gui::registerSoundLoader来注册这个全局loader, 用gcn::Gui::removeSoundLoader来反注册这个全局loader.

但是我并没有给gcn::Widget加上声音支持, 因为各个控件对声音的需要差别太多并且内部控件实际使用情况很少.

9.2.2 Modifier

Modifier是gcn::Widget的一类新成员, 你可以通过gcn::Widget::addModifier来添加一个modifier, 通过gcn::Widget::removeModifier来移除一个modifier.

Modifier基类定义了isFinished接口, 在这个接口的返回值为true的时候, widget会自动删除这个modifier, 所以必须保证addModifier总是有效并且对于每个widget都是独一无二的.

如果存在Modifier, Widget会在每次logic的时候调用所有Modifier的update(Widget* widget)函数, 让modifier修改自身状态. 当一个modifier完成自身的使命之后, 可以设置mFinished为true, 让widget删除自身.

9.3 xmlgui和jsongui

9.3.1 基础

SoraGUIChan也提供了xmlgui和jsongui定义文件的解析功能. 其中xmlgui基于XMLGuichan(http://code.google.com/p/xmlguichan/), jsongui则是Sora的一部分. 两种格式功能和含义等同, 但是写法不一样.

xmlgui位于SoraGUIChan/xmlgui.h

jsongui位于SoraGUIChan/jsongui.h

xmlgui必须以一个主Window或者Container作为总Container/Window, 例如

```
<container name="SoraWindow" width="1024" height="768" x="0" y="0" opaque="false" foregroundColor="0xAAAAAA">
        // some widgets
</container>
```

而jsongui则不一定, 可以以一个widget数组表示, widget数组的名字必须为”widgets”, 例如

```
“widgets”: [
     { “button”: {
     // button properties
     }
```

```
        },
        { “listbox”: {
        // listbox properties
        }

        }
     ]
```

在jsongui里, 所有Container/Window的子控件必须以”widgets”数组的形式作为container/window的属性之一, 而在xmlgui里则是直接作为container/window的子节点.

而Widget的写法都是 WidgetName + properties, 例如在xml里

```
<button x=”1” y=”2” name=”open” />
```

在json里

```
“button”: {
     x = 1,
     y = 2,
     name = “open”
}
```

9.3.2 属性

无论在jsongui还是xmlgui内, widget都拥有如下通用属性

x	x坐标
y	y坐标
width	宽
height	高
basecolor	基础颜色
foregroundcolor	前景颜色
backgroundcolor	背景颜色
framesize	边框宽
font	字体
visible	是否默认可见
focusable	是否可以获取焦点
enabled	是否默认开启
tabin	是否允许tab键移动焦点到自身
tabout	是否允许在按下tab键的时候移动焦点到下一个
eventId	widget的ActionEventId
parent	父控件, 在此field不为空的时候, 这个控件的父控件不一定是xml/json定义文件内从属的父控件
name	控件独一无二的名字, 用于xmlgui或者jsongui储存和获取
reponser	SoraGUIResponser名字, 参见GUIReponser一节

* 颜色依赖于Widget的实现, 在不同Widget内可能拥有不同的含义

并且依照widget类型的不同, 不同的widget拥有自己独特的属性.

控件	属性
container	opaque(透明)
label	caption align(对齐方式, “left/center/right”
imagebutton	image(图片路径)
button	caption align(对齐方式, “left/center/right”
checkbox	caption marked(是否默认被标记)
radiobutton	caption group(组名)
icon	image(图片路径)
textbox	editable(是否可以被编辑) text opaque
textfield	text
slider	start(开始值) end(结束值) value(默认值) markerLength(滑块长度) stepLength(单步长度) orientation(方向, “horizontal/vertical”)
window	caption tabbing movable(是否可以移动) titleBarHeight(标题栏高度)
scrollArea	hPolicy(纵向滚动规则, “ALWAYS/NEVER”) vPolicy(横向滚动规则, “ALWAYS/NEVER”) vScrollAmount(横向滚动幅度) hScrollAmount(纵向滚动幅度) content(内含Widget的name)

控件	属性
dropdown	所有名字为li的子节点皆为dropdown的一栏(xml), “items”数组为dropdown的条目(json) selected(被选中的一栏)
listbox	所有名字为li的子节点皆为listbox的一栏(xml), “items”数组为listbox的条目 selected(被选中的一栏)

9.3.3 写法的一些区别

虽然属性名称是通用的，但是在xmlgui和jsongui内属性的写法不一定相同．例如listbox和dropdown在xml内，是用所有名字为li的子节点表示自己拥有的条目，而在json内则是一个items数组，例如在xml内

```
<listbox /* some properties >
     <li name=”1”/>
     <li name=”2”/>
</listbox>
```

而在json内则是

```
“listbox”: {
     // some propertise
     “items”: [
     “1”, “2”
     ]
}
```

总在来说，表示可能有复数属性的属性时，xml总是使用子节点，而json总是使用特定命名的数组．

9.3.4 GUIResponser

在xmlgui和jsongui内，还有一类特殊的属性，responser和responsetype，这两个属性属于Sora的扩展，方便用定义文件定义widget的各类reponser．要使用这个特性，有两件事要做．首先在代码内书写Reponser类并且注册．

GUIResponser的基类是SoraGUIResponer，定义在SoraGUIChan/SoraGUIResponser内．他继承了所有widget可能的reponser的所有接口，通常你只用实现其中的几个就可以．例如想接收按钮被点击的事件就只需实现action接口．

在实现了responser类之后，就可以通过SoraGUIResponserMap注册responser，SoraGUIResponserMap同样是一个Singleton类，你可以通过SoraGUIResponserMap::Instance获取到他的实例．例如

```
SoraGUIResponserMap::Instance()->registerResponser
("SpeedPanel", new SpeedPanelResponser);
```

这就注册了一个名叫SpeedPanel的responser, 然后在xml或者json定义文件内你的widget就可以标记responser = “SpeedPanel”来表示这个widget将发送自身事件到这个responser. 而responsetype则是需要发送的事件的类型, 有如下六类

action
death
key
mouse
focus
* seletion

其中selection类型只对于listbox或者dropbox才有效, 不同的responsetype有不同的回调函数, 你需要根据你的需求来在responser内实现相应的函数. 例如

```
<button /*some props*/ responser=”SpeedPanel” responsetype=”action”/>
```

或者

```
“button”: {
     // some props
     responser = “SpeedPanel”,
     responsetype = “action”
}
```

当Widget接收到事件同时他有处理这一事件的responser的时候, 他就会调用responser的相应函数, 此时在responser内你可以通过getID()函数来获取发生事件的widget的id(xmlgui或者jsongui内的name属性), 通过getSource()来获取发生事件的widget本身. 例如

```
class SpeedPanelResponser: public SoraGUIResponser {
    void action() {
        PSLIDER p = (gcn::Slider*)getSource();
        if(getID().compare("MinSpeed") == 0) {
            peffect->pheader.fMinSpeed = p->getValue();
        }
        else if(getID().compare("MaxSpeed") == 0) {
            peffect->pheader.fMaxSpeed = p->getValue();
        }
        else if(getID().compare("MinLinearAccel") == 0) {
            peffect->pheader.fMinLinearAcc = p->getValue();
        }
        else if(getID().compare("MaxLinearAccel") == 0) {
            peffect->pheader.fMaxLinearAcc = p->getValue();
        }
        else if(getID().compare("MinTrigAccel") == 0) {
            peffect->pheader.fMinTrigAcc = p->getValue();
        }
        else if(getID().compare("MaxTrigAccel") == 0) {
            peffect->pheader.fMaxTrigAcc = p->getValue();
        }
    }
};
```

这个responser就响应5个slider的事件, 获取slider的value然后设置particlesystem的相应属性.

9.3.5 Modifier

在xmlgui或者jsongui内还有一个特殊的属性, modifier, 定义了被扩展的guichan核心机制之一 – modifier.

modifier属性必须作为xml控件的子结点或者json控件的”modifiers”数组, 写法在xml内为

```
<widget /* some props */ >
     <modifier name=”name” object=”widgetname”/>
</widget>
```

在json内为

```
“modifiers”: [
     {
     name = “name”,
     object = “widgetname”
     }
]
```

目前可用的modifier只有LabelSliderLinker一个, name为SliderLinker, 只在Label下有效. 作用是把一个Slider的Value和一个Label的内容链接, 让Label动态反映Slider的Value. object属性为要链接的slider的name, 由于解析顺序, label必须在目标slider之下.

例如

```
<slider name=”mySlider” /* some other props />
<label /* some props */ >
     <modifier name=”SliderLinker” object=”mySlider”/>
</label>
```

9.3.6 新控件的解析

xmlgui和jsongui都支持注册新控件的解析函数，在xml内这个函数的原型为

```
typedef void (*parseFunc)(TiXmlElement* element,
gcn::Widget* widget, XmlGui* pcaller);
```

在json内则为

```
typedef void (*JsonGuiParseFunc)(const Json::Value& val,
gcn::Widget* parent, JsonGui* pCaller);
```

他们都通过registerParseFunc(widgetname, parseFunc)的方式注册，优先级比内置控件低. 当xmlgui和jsongui找不到关于一个控件的名字(widgetName)的内置解析函数的时候，就会试图查找这些注册的外部函数，如果有符合的，则以当前解析状态调用这些函数. 对于第一个参数，xmlgui给的是一个xml节点，而jsongui则是给的一个json object，都代表当前的widget. 第二个参数则是按照解析状态当前控件的父控件，如果没有则为NULL，第三个参数则是xmlgui或者jsongui本身，函数在解析完控件的时候，可以通过xmlgui或者jsongui添加这个控件到xmlgui或者jsongui方便储存和获取.

9.4 gcnExtend插件

Sora的Plugins内还提供了gcnExtend插件，提供了一些常用的guichan控件扩展，包括

gcnImageButton2　　完全基于Image的Button
gcnDraggableIcon　　可以拖拽的Icon
gcnDraggableImageButton　　可以拖拽的ImageButton
gcnDraggableImageButtonIncubator　　在点击的时候会生成新的Widget的的控件. (拖过拖拽生成新的Widget)
gcnBackgoundContainer　　可以设置背景的Container
gcnSelectableContainer　　可以用鼠标选取子Widget的Container
等

Lua导出

10.1 基础

Sora的Plugins内提供了SoraLua插件, 提供了SoraLuaObject, SoraLuaExport和各类wrapper函数. SoraLua基于修改版的LuaPlus(添加std::string和std::wstring的支持).

10.2 SoraLuaObject

SoraLuaObject是针对LuaPlus的LuaState封装, 可以直接从文件或者LuaState构造, 可以load或者do script/string/buffer. 可以方便的获取各种类型的lua变量. 同时在调用SoraLuaObject的update和render函数时会自动调用响应Lua代码内的update和render function.

10.3 SoraLuaExport

这个静态类用于导出部分Sora核心组件到Lua. SoraLuaObject在构造的时候会自动导入常量, SoraCore, Sprites和Font.

10.4 更多的导出

除了会自动导出的部分, 部分Sora插件内也提供了Lua导出函数. 例如SoraSoundManager插件就提供了exportSoundManager函数导出SoraBGMManager和SoraSoundEffectManager. 或者SoraGUIChan的guichanWrapper提供了guilib的导出.

10.5 导出组件的使用

Sora的lua导出分为两类. 一类是基于LuaClass直接导出的Singleton类, 例如SoraCore, SoraBGMManager等. 这类类你可以通过名字直接调用他们的函数. 例如

```
SoraCore:messageBox(“kak”, “lal”, MB_OK);
```

还有一类是通过wrapper函数导出的lib, 例如spritelib, fontlib, guilib. 具体可用函数请参见API文档.

10.6 LuaFunctionHelper

SoraLua/SoraLuaFunctionHelper定义了一些快速调用LuaPlus的LuaFunction的方法. 统一以callLuaFunc模板函数命名. 第一个和第二个参数都是LuaState示例和函数名字, 剩下的参数依赖于参数数量而不同.

例如

```
callLuaFunc<int, float>(myState, “test”, 1, 1.0);
```

Debug

11.1 SoraConsole

在Sora/cmd/SoraConsole.h内定义了SoraConsole这个内置的Singleton Console, 有两个tab, 一个是cmd line, 用于执行调试命令等. 一个是log, 用于显示SoraInternalLogger内的log, 并可以保存到SoraLog.log文件.

Console的默认呼出快捷键是SORA_KEY_GRAVE, 用户可以通过SoraConsole::setActiveKey(key)来设置呼出的快捷键.

Console的位置, 配色等信息都是可以调整的, 具体请参见API文档.

用户可以通过tab键在Console的两个tab之间切换.

11.1.1 Cmd

SoraConsole内部并不处理命令. 需要处理Cmd调试命令的类必须是SoraEventHandler然后通过SoraConsole的registerCmdHandler(handler, cmd)来注册自己需要处理的命令. 例如SoraCore就调用了

```
registerCmdHandler(this, “set”);
registerCmdHandler(this, “exit”);
```

等. 来处理核心的一些命令.

当有命令被输入且回车被按下时, SoraConsole会搜寻当前命令(第一个空格之前的字符串)对应的Handler, 然后构造SoraConsoleEvent传递给Handler处理.

Handler可以从SoraConsoleEvent的getCmd()函数获取当前Cmd, getParams()函数获取参数, 并且可以用setResult()设置命令返回值以显示在console.

Cmd的参数Params是以一个单独的字符串表示的, 用户需要手动分割, 例如使用SoraStringConv.h里的deliStr.

返回值需要用户手动设置, 如果有返回值被设置了, 那么返回值会以绿色显示在执行了的命令的下方.

当命令历史超过Console的高度时, 用户可以通过上下键在滚动历史记录. 同时输入clear命令可以清空历史记录.

关于Event回调的更多信息请参考第4章Event事件回调.

以下是目前可用的Cmd列表

set window.width (width)	设置主窗口的宽度
set window.height (height)	设置主窗口的高度
set window.title (title)	设置主窗口的标题
exit	退出Sora引擎
clear	清空Console记录

11.1.2 Log

这个tab会以不同的颜色显示SoraInternalLogger内的log, 用户可以通过上下键来滚动记录.

11.2 Profiler

Profiler则是一个记录代码运行时间的helper. 你可以通过AUTO_PROFILE(name)宏来开始一个profile, 然后通过SoraGlobalProfiler的getProfile(name)来获取一个特定的profile或者printProfile(name)来向logger里输出这次Profile的信息. 注意profile是依赖于构造和析构函数记录时间. 所以你可以通过{ }包含代码块的方式来profile某一块代码的运行时间. 例如

```
// code part 1
{
    AUTO_PROFILE("myProfile");
    // code part 2
}
// code part 3
```

这样myProfile这个profile就是记录的code part 2的运行时间

11.3 Debug, DebugPtr

在Sora/Debug/SoraInternalLogger.h内定义Debug和DebugPtr两个宏, 其中Debug单纯代表SoraInternalLogger, 可以使用printf函数. 而DebugPtr则是代表SoraInternalLogger的实例, 可以使用log, get等其余函数.

11.3.1 log

Sora的debug函数接收两个参数, 一个是debug信息本身, 一个是log level, log level决定了这条信息在SoraConsole内显示的颜色.

LOG_LEVEL_NORMAL	白色
LOG_LEVEL_WARNING	黄色
LOG_LEVEL_NOTICE	蓝色
LOG_LEVEL_ERROR	红色

同时SoraCore内也有log, logf, logw等log接口. 一般来说用户通过SoraCore的log函数来向InternalLogger输入log.

SoraInternalLog内还定义了vamssg函数, 用于格式化一个字符串为std::string, 例如

vamssg("time: %d", 10)将返回"time: 10", 利用这个函数可以方便的log需要格式化的字符串.

12. Localize 本地化

Sora核心内的SoraLocalizer类提供了基本的本地化支持.

12.1 Locale文件写法

一个locale文件的格式如下

```
@LOCALE_NAME
IDENT = “STRING”
“FILE” = “LOCALIZED_FILE”
```

文件开头第一行必须是@加上locale的名字, 例如chn
同一个locale可以分为几个文件定义.

ident是一个标识符, 后面跟着的是本地化的字符串
“FILE”标识一个字符串文件名, 后面跟着的是本地化的字符串文件名, 用于本地化资源文件

例如,

```
@chn
hello = “你好”
“hello.png” = “hello_chn.png”
```

就定义了chn这个locale里的两个本地化内容, 同时我们可能有

```
@eng
hello = “hello”
“hello.png” = “hello_eng.png”
```

就定义了eng这个locale的本地化内容.

12.2 加载

我们可以通过SoraLocalizer的addLocaleConf函数来加载一个Locale文件.
每加载一个locale都会导致当前locale被设置为加载的locale文件的locale

12.3 本地化字符串

在加载完locale文件之后, 我们就可以通过GET_LOCAL_STR(ident)宏来获取本地化字符串, 通过setLocale函数来设置当前locale, 例如在上面的例子里, 如果我们执行代码:

```
SoraLocalizer::Instance()->setLocale(“chn”);
SoraWString myHello = GET_LOCAL_STR(“hello”);
```

则myHello的值为”你好”

基于此, 在实际工程中, 建议所以涉及到本地化字符串的地方都使用GET_LOCAL_STR宏替代, 当指定ident不存在时, 会返回ident本身, 所以不用担心无效字符串问题

12.4 本地化资源文件

本地化之后的资源文件名可以通过GET_LOCAL_RESOURCE(name)来获取, 例如在上面的例子里

```
SoraWString myHelloPic = GET_LOCAL_RESOURCE(“hello.png”)
```

在当前locale为chn的情况下会被设置为hello_chn.png, 而如果locale是eng则会被设置为hello_eng.png

更新记录:
6.7
提出第7章的Debug内容, 结合新的SoraConsole相关内容组成第11章.
RTTI增加一些说明.
调整了下排版.

6.9
添加Event的Consume和inputEvent的Consume相关内容
添加第12章: 本地化