

Ogre 的渲染系统 (Rendering System)

盛崇山

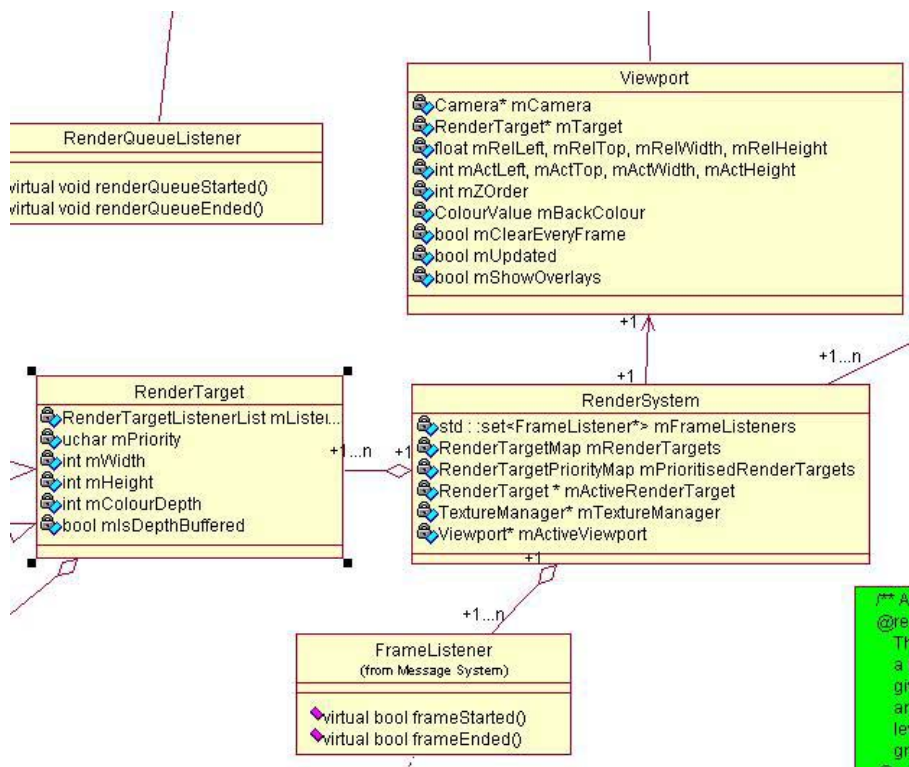
<http://antsam.blogone.net>

AntsamCGD@hotmail.com

渲染系统个人认为就是同图形学有关的一切系统；包括了 geometry system、material system、shader system。Rendering system 分为两部分：API 无关的部分和 API 相关部分。后者主要是只同 DirectX 和 OpenGL 相关的系统。前者则是两个的共同的特性。API 无关部分同 Pipeline Abstraction 有很大的关系，API 的部分功能就是设置 Pipeline (Pipeline configuration)，Pipeline 可以参考《Pipeline Design》。既然是 Render System 必然是有 Target 的，我们前面把 Scene Manager 比喻成舞台跟导演，那么 Render System 就是摄影机，而胶片就是 Render Target。先让我们看一下相关类的 UML 图，然后在详细这些类以及相互之间的关系。

一、Render System UML 图

Render System 相关类 UML Part one :



图一：Render System 相关类 UML Part one

图一中的类 View Port 包含了一个 Camera ,另外一个同 Render System 相关的类(未画出，图中只有一根聚合的关系线) Root。前面说过把 Render System 比喻成摄影机，现在我们进一步的细化：View Port 和 Camera 就是摄影机的摄像头，Render Target 就是

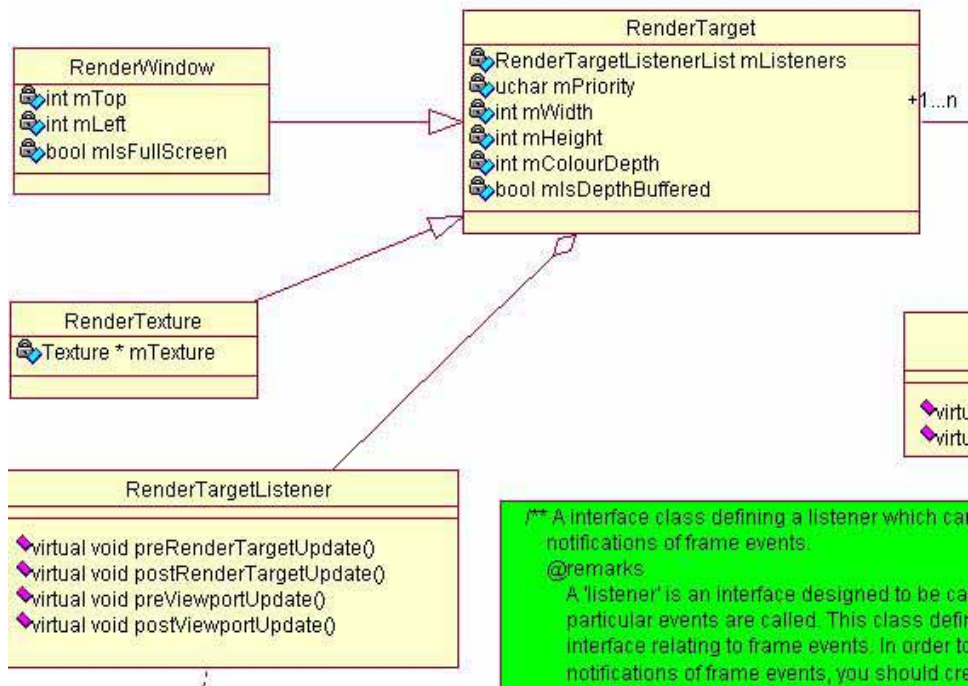
胶带，那么 Render System 就是一个光学系统，它把从摄像头进来的光进行处理变换，然后再把结果写到 Render Target 中，而 Root 就是各种各样的道具；那么，演员呢？演员就是游戏中的各种角色；电影剧本对应的是游戏剧本（或者称为策划）；化妆师对应的则是美工和建模人员；电影的音乐也越来越重要了，这就类似于游戏中的音乐。

Good (^_^)，现在让我们想象一下，我们在拍摄现场：

导演	Scene Manager
摄影机	Render System(including view port、camera、target)
灯光、道具、服装	Root(engine or Resource Manager)
演员	游戏中的角色 (mesh、 entity)
化妆师	美工和建模人员
剧本	游戏策划
电影音乐	游戏音乐

从上面可以看出电影公司和游戏制作公司非常相似，那么一个基本的游戏引擎应该至少包括前面的四项，在《场景组织分析》那篇文章中从功能上对导演和 Scene Manager 进行了类比，我们将在对 Render System 具体的分析后在进行类比。

Render Target 的 UML 图如下所示：



图二： Render System 相关类 UML 图 Part two

电影的胶片不同，Render Target 也有不同 (^_^)。在上面的两幅图中还有两个 listener 类应该注意一下，我们将在下面的详细分析中再做具体的分析。

二、详细分析

- Camera

首先我们看一下 Camera 的定义：

```
class _OgreExport Camera : public MovableObject
{
protected:
    // Camera orientation, quaternion style
    Quaternion mOrientation;

    // Camera position - default (0,0,0)
    Vector3 mPosition;

    // Stored versions of parent orientation / position
    mutable Quaternion mLastParentOrientation;
    mutable Vector3 mLastParentPosition;

    // Derived positions of parent orientation / position
    mutable Quaternion mDerivedOrientation;
    mutable Vector3 mDerivedPosition;

    // Camera y-direction field-of-view (default 45)
    Real mFOVy;
    // Far clip distance - default 10000
    Real mFarDist;
    // Near clip distance - default 100
    Real mNearDist;
    // x/y viewport ratio - default 1.3333
    Real mAspect;
    // Whether to yaw around a fixed axis.
    bool mYawFixed;
    // Fixed axis to yaw around
    Vector3 mYawFixedAxis;

    // The 6 main clipping planes
    mutable Plane mFrustumPlanes[6];

    // Orthographic or perspective?
    ProjectionType mProjType;
    // Rendering type
    SceneDetailLevel mSceneDetail;
```

Camera 的主要作用：Camera control、特定的拍摄手法（聚焦等^^，这主要是通过改变投影方式的）、Frustum Cull，上面的代码中用了三种不同的颜色来表示三种功能的数据结构。

Camera Control 用于控制摄影机的轨迹（例如：平移、旋转等，想象一下在拍电影的时候，摄影机是怎么移动的）；特定的拍摄手法主要是改变距离来改变成像的大小

(凸透镜成像原理^_^); Frustum Cull 用于剔除在摄影机范围之外的 Object, 减小渲染压力。

- View Port

代码定义如下:

```
class _OgreExport Viewport
{
public:
protected:
    Camera* mCamera;
    RenderTarget* mTarget;
    // Relative dimensions, irrespective of target dimensions (0..1)
    float mRelLeft, mRelTop, mRelWidth, mRelHeight;
    // Actual dimensions, based on target dimensions
    int mActLeft, mActTop, mActWidth, mActHeight;
```

View Port 作用有点难表述, 打个比方就是我们看的电影有的是宽银幕的, 而有的只是一般银幕。View Port 就类似于这个银幕的大小, 不同的 View Port 可以达到不同的效果。成员变量中还有 Camera 和 Render Target 两个变量, 暂时还没有弄清楚其作用, 可能是把 Camera 同 Render Target 一对一的对应起来吧。

- Render Target

Render Target 只是一个抽象类, 只包含了 Render Target 的基本信息, 代码定义如下所示:

```
class _OgreExport RenderTarget
{
public:
protected:
    int mWidth;
    int mHeight;
    int mColourDepth;
    bool mIsDepthBuffered;

    // Stats
    StatFlags mStatFlags;
    Timer* mTimer ;
    float mLastFPS;
    float mAvgFPS;
    float mBestFPS;
    float mWorstFPS;
    float mBestFrameTime ;
    float mWorstFrameTime ;
    unsigned int mTris;
    String mDebugText;
```

```

bool mActive;
typedef std::map<int, Viewport*, std::less<int> > ViewportList;
/// List of viewports, map on Z-order
ViewportList mViewportList;

typedef std::vector<RenderTargetListener*>
    RenderTargetListenerList;
RenderTargetListenerList mListeners;

```

代码中包括了三个基本信息：Render Target 的尺寸、渲染的 Fps (frame per second、Listener。下面看一下两个子类定义：Render Window 和 Render Texture。

```

class _OgreExport RenderTexture : public RenderTarget
{
protected:
    /// The texture that gets accesses by the rest of the API.
    Texture * mTexture;

class _OgreExport RenderWindow : public RenderTarget
{
protected:
    bool mIsFullScreen;
    int mLeft;
    int mTop;

```

上面两个类的定义中，不同的 target 有不同的特性，当纹理作为 target 的时候必须有一张纹理作为 target，所以包含了成员变量 (Texture m_Texture)。下面让我们来看一下更具体的 Render Target 的定义：

```

class D3D9RenderWindow : public RenderWindow
{
protected:
    HWND    mExternalHandle;    // External Win32 window handle
    HWND    mHWnd;              // Win32 Window handle
    HWND    mParentHWnd;        // Parent Win32 window handle
    bool    mActive;            // Is active i.e. visible
    bool    mReady;            // Is ready i.e. available for update
    bool    mClosed;

static LRESULT CALLBACK WndProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam );

```

```

// -----
// DirectX-specific
// -----

// Pointer to D3DDriver encapsulating Direct3D driver
D3D9Driver* mpD3DDriver;

// Pointer to the 3D device specific for this window
LPDIRECT3DDEVICE9  mpD3DDevice;
D3DPRESENT_PARAMETERS md3dpp;
LPDIRECT3DSURFACE9 mpRenderSurface;
LPDIRECT3DSURFACE9 mpRenderZBuffer;

```

到上面这个代码我们就比较熟悉了吧^^! 典型的 Win32 窗口。可能我们还有个疑问，怎么把 Object 渲染到 Window 上？

- Render System

现在我们可以看一下 Render System 的定义了：

```

class _OgreExport RenderSystem
{
protected:
    /** The render targets. */
    RenderTargetMap mRenderTargets;
    /** The render targets, ordered by priority. */
    RenderTargetPriorityMap mPrioritisedRenderTargets;
    /** The Active render target. */
    RenderTarget * mActiveRenderTarget;

    // Texture manager
    // A concrete class of this will be created and
    // made available under the TextureManager singleton,
    // managed by the RenderSystem
    TextureManager* mTextureManager;

    /// Used to store the capabilities of the graphics card
    RenderSystemCapabilities* mCapabilities;

    // Active viewport (dest for future rendering operations)
    Viewport* mActiveViewport;

    CullingMode mCullingMode;

    bool mVSync;

```

```

// Store record of texture unit settings for efficient alterations
Material::TextureLayer mTextureUnits[OGRE_MAX_TEXTURE_LAYERS];

size_t mFaceCount;
size_t mVertexCount;

/// Saved set of world matrices
Matrix4 mWorldMatrices[256];

```

上面的代码中有四块需要说明一下：第一块是把 Render System 和 Target 联系在一起，而第二块则是由于需要对 Texture 操作，第三块是必须设置的 ViewPort，最后一块模拟了 Matrix stack，这主要是由于 OpenGL 和 DirectX 中对于 Matrix 的实现方式的不同，而且用法也有区别。我们说过 Render System 的主要作用在于对 Pipeline 的配置（Pipeline configuration）。所以看 Render System 的成员函数更有意义。

1. _set 函数（主要用于内部设置）

```

◆ _setAlphaRejectSettings(CompareFunction func, unsigned short bias)
◆ _setAnisotropy(int maxAnisotropy)
◆ _setCullingMode(CullingMode mode)
◆ _setDepthBias(ushort bias)
◆ _setDepthBufferCheckEnabled(bool enabled = true)
◆ _setDepthBufferFunction(CompareFunction func = CM_GREATER_EQUAL)
◆ _setDepthBufferParams(bool depthTest = true, bool depthWrite = true)
◆ _setDepthBufferWriteEnabled(bool enabled = true)
◆ _setFog(FogMode mode = FOG_NONE, ColourValue colour, float density)
◆ _setProjectionMatrix(const Matrix4 &m)
◆ _setRasterisationMode(SceneDetailLevel level)
◆ _setSceneBlending(SceneBlendFactor sourceFactor, SceneBlendFactor destFactor)
◆ _setSurfaceParams(const ColourValue &ambient, const ColourValue &emissive)
◆ _setTexture(int unit, bool enabled, const String &texName)
◆ _setTextureAddressingMode(int unit, Material::TextureAddressingMode mode)
◆ _setTextureBlendMode(int unit, const LayerBlendMode &blendMode)
◆ _setTextureCoordCalculation(int unit, enum TexCoordCalculation mode)
◆ _setTextureCoordSet(int unit, int index)
◆ _setTextureLayerAnisotropy(int unit, int maxAnisotropy)
◆ _setTextureLayerFiltering(int unit, const TextureFilter &filter)
◆ _setTextureMatrix(int unit, const Matrix4 &xform)
◆ _setTextureUnitSettings(int texUnit, Material::TextureUnitSettings settings)
◆ _setViewMatrix(const Matrix4 &m)
◆ _setViewport(Viewport *vp)
◆ _setWorldMatrices(const Matrix4 *m, unsigned short count)
◆ _setWorldMatrix(const Matrix4 &m)

```

为什么需要一个内部的 set 函数呢？现在个人认为主要是由于 Engine 中对很多基本结构进行了包装，所以可以用能不设置函数进行一定的转化，例如 _setTexture，由于我们所有用的 Texture 类和 DirectX 中的 Texture

类不同，不能直接调用 DirectX 的 API，当然可以用引用成员变量的方法来进行设置，但是这么从外观上不好，而且也没有速度上的优势（暂时我只能给出这种解释^^）。还有一些 Matrix、Texture 属性、Depth Buffer 属性的设置。

2. Set 和 Get 函数

```
◆ getCapabilities()
◆ getConfigOptions()
◆ getErrorDescription(long errorNumber)
◆ getName()
◆ getRenderTarget(const String &name)
◆ getWaitForVerticalBlank()
◆ initialise(bool autoCreateWindow)
◆ reinitialise()
◆ removeFrameListener(FrameListener *oldListener)
◆ RenderSystem()
◆ ~RenderSystem()
◆ setAmbientLight(float r, float g, float b)
◆ setConfigOption(const String &name, const String &value)
◆ setLightingEnabled(bool enabled)
◆ setNormaliseNormals(bool normalise)
◆ setShadingType(ShadeOptions so)
◆ setStencilBufferDepthFailOperation(enum StencilOperation op)
◆ setStencilBufferFailOperation(enum StencilOperation op)
◆ setStencilBufferFunction(CompareFunction func)
◆ setStencilBufferMask(ulong mask)
◆ setStencilBufferParams(CompareFunction func = CMPF_ALWAYS_PASS)
◆ setStencilBufferPassOperation(enum StencilOperation op)
◆ setStencilBufferReferenceValue(ulong refValue)
◆ setStencilCheckEnabled(bool enabled)
◆ setTextureFiltering(TextureFilterOptions fo)
◆ setVertexBufferBinding(VertexBufferBinding *binding)
◆ setVertexDeclaration(VertexDeclaration *decl)
◆ setWaitForVerticalBlank(bool enabled)
◆ shutdown()
```

上面的图中包含了 Stencil Buffer、Light 的设置，最值得关注的是对于 Vertex Buffer 以及相关属性设置的函数。我们将在后面更具体的分析 Buffer。

现在我们基本上分析了 Render System 中于具体的 API 无关的部分，现在我们分析一下同 DirectX API 相关的部分，然后我们看看 D3D9RenderSystem 的定义：

```
class D3D9RenderSystem : public RenderSystem
{
private:
    这是同DirectX相关的

    /// Direct3D
```

```

LPDIRECT3D9          mpD3D;
/// Direct3D rendering device
LPDIRECT3DDEVICE9   mpD3DDevice;

.....
这是同Win32相关的
/// external window handle ;)
HWND mExternalHandle;
/// instance
HINSTANCE mhInstance;

这是同DirectX相关的
/// List of D3D drivers installed (video cards)
D3D9DriverList* mDriverList;
/// Currently active driver
D3D9Driver* mActiveD3DDriver;
/// Device caps.
D3DCAPS9 mCaps;

// Array of up to 8 lights, indexed as per API
// Note that a null value indicates a free slot
Light* mLights[MAX_LIGHTS];

D3D9DriverList* getDirect3DDrivers(void);
void refreshD3DSettings(void);

inline bool compareDecls( D3DVERTEXELEMENT9* pDecl1,
                          D3DVERTEXELEMENT9* pDecl2, int size );

```

这部分主要是由于DirectX中的用的是左手坐标系，而一般系统沿用的是OpenGL的标准，采用了右手坐标系，所以，在设置Matrix的时候需要进行转换。

```

// Matrix conversion
D3DXMATRIX makeD3DXMatrix( const Matrix4& mat );
Matrix4 convertD3DXMatrix( const D3DXMATRIX& mat );

```

```

void initInputDevices(void);
void processInputDevices(void);
void setD3D9Light( int index, Light* light );

```

这部分代码中直接用了DirectX的枚举常量，这种定义应该是非常原始的操作，尽量不要用（所以放在Private中）因为不具有通用性。

```

// state management methods, very primitive !!!

```

```

HRESULT __SetRenderState(D3DRENDERSTATETYPE state, DWORD value);
HRESULT __SetSamplerState(DWORD sampler,
                          D3DSAMPLERSTATETYPE type, DWORD value);
HRESULT __SetTextureStageState(DWORD stage,
                               D3DTEXTURESTAGESTATETYPE type, DWORD value);

```

```

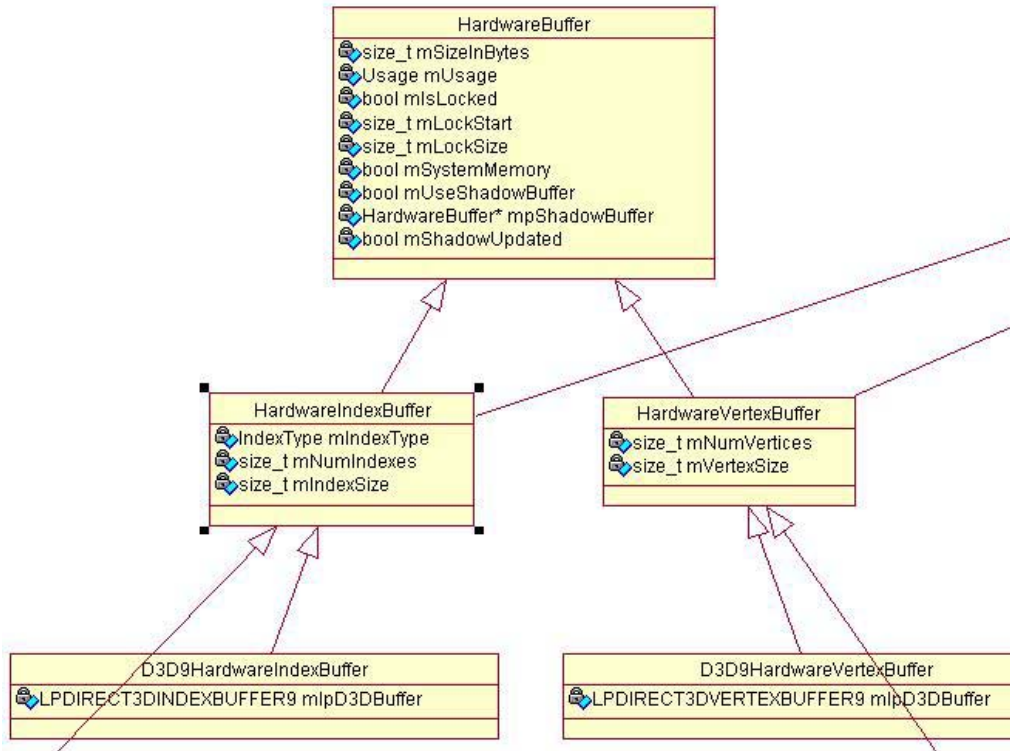
D3D9HardwareBufferManager* mHardwareBufferManager;
size_t mLastVertexSourceCount;

```

前面基本上把Render System分析的差不多了(^_^)，在Render System中还有一个于特定API相关的类非常重要，那就是Buffer，下面我们就是来分析Buffer，而且实现上主要以分析DirectX为主。

- Buffer

先看看同 Buffer 相关的类的 UML 图，这里主要分析 DirectX 的实现方式。



图三：Buffer 相关类 UML 图

几个问题：

- Buffer 主要用在哪里？
- Buffer 什么时候创建？
- Buffer 通过谁来创建？
- Buffer 怎么来管理？

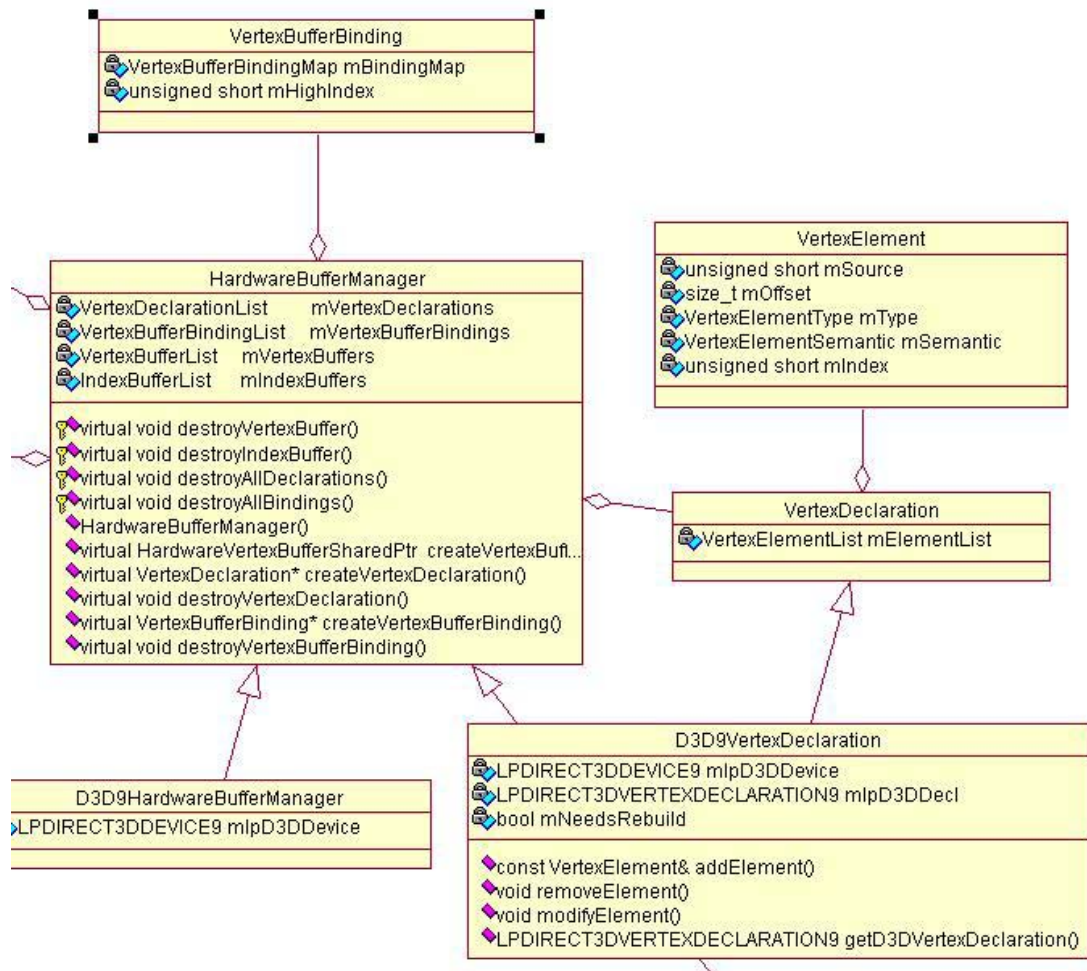
Buffer 目前主要用于存放 Mesh 的数据

Buffer 在构造函数中创建

主要通过 Hardware Manager 及其子类来创建，或者直接 new。

Buffer 通过 Hardware Manager 来管理，Hardware 是一个 Singleton。

现在分析一下 Hardware Manager，其相关类 UML 图如下所示：



上面的图中我们可以看出 Hardware Buffer Manager(在 DirectX 中 D3D9 Hardware Buffer Manager 对 Vertex Buffer 和 Index Buffer 进行管理，邮编的那几个类用于所名 Buffer 中的数据是以什么结构存放的，例如：Buffer 中是否包含 Color、Position、Normal、Texture Coordinate 等，什么顺序存放等都在 Declaration 中说明。

还有一个类必须注意的是 Vertex Buffer Binding，起作用我们可以直接引用代码中的说明：

```
/** Records the state of all the vertex buffer bindings required to provide a vertex declaration  
with the input data it needs for the vertex elements.
```

```
@remarks
```

```
Why do we have this binding list rather than just have VertexElement referring to the  
vertex buffers direct? Well, in the underlying APIs, binding the vertex buffers to an
```

index (or 'stream') is the way that vertex data is linked, so this structure better reflects the realities of that. In addition, by separating the vertex declaration from the list of vertex buffer bindings, it becomes possible to reuse bindings between declarations and vice versa, giving opportunities to reduce the state changes required to perform rendering.

@par

Like the other classes in this functional area, these binding maps should be created and destroyed using the HardwareBufferManager.

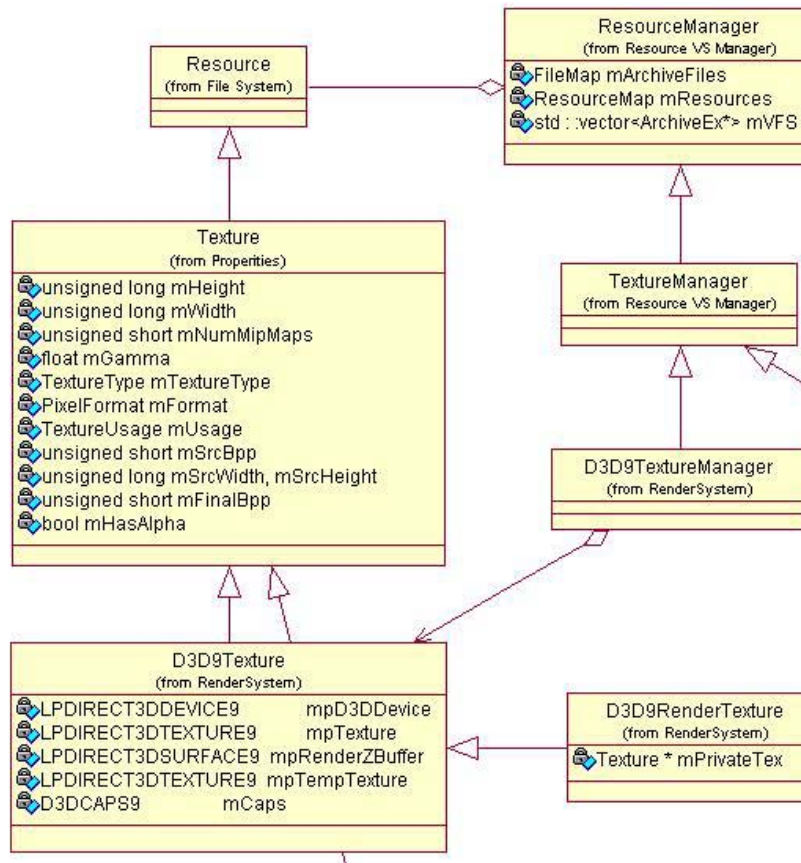
```
class _OgreExport VertexBufferBinding
{
public:
    /// Defines the vertex buffer bindings used as source for vertex declarations
    typedef std::map<unsigned short, HardwareVertexBufferSharedPtr>
        VertexBufferBindingMap;

protected:
    VertexBufferBindingMap mBindingMap;
    unsigned short mHighIndex;
```

除了 Buffer 与具体的 API 相关外，Texture 也是同 API 相关的。

- Texture

Texture 相关类 UML 图如下所示：



从 UML 图中我们可以看出 Texture 同 Buffer 在类结构上有非常多的相似性，从创建、管理上都一致，但是有一点是不同的，那就是 Texture 在构造函数中是不创建纹理接口的，只是初始化了纹理所需要的属性，在适当的时候才 load 纹理数据的，至于为什么，暂时未搞清楚。

OK，差不多分析好了，有了导演，有了摄影师，那我们还少一些基本的道具和工具，下次我们分析《数据文件结构》，到时候我们就可以着剧本拍电影了^_^。

三、Summary

暂时分析到这里，其实还可以分析的更细一些，不过现在没有必要。