相关主题 RECOMMEND ARTICLE
- 胶囊和凸多边形的动态碰撞检测
- 经验技巧：分享两条Delphi开发经验
- A* Algorithm
- Java程序最容易犯的21种错误实例分析
- 《Breaking The Walls》算法的第一印象和空间分割杂论
- 高效网游服务器实现探讨（一）
- 向量几何在游戏编程中的使用6
- 游戏中的渲染器架构

MORE

推荐文章 RECOMMEND ARTICLE
- 游戏音乐制作案例之《战火 红色警戒》音效制作揭秘
- 英雄连Online 原画
- 游戏音乐制作案例之《乱武天下》
- 游戏音乐制作案例之《诛仙》
- 《鹿鼎记》最新原画
- MIDP2.1规范的新特性
- 3D游戏编程入门经典(6)
- Introduction to 3d game engine design using directx-9 and c#(10)

MORE

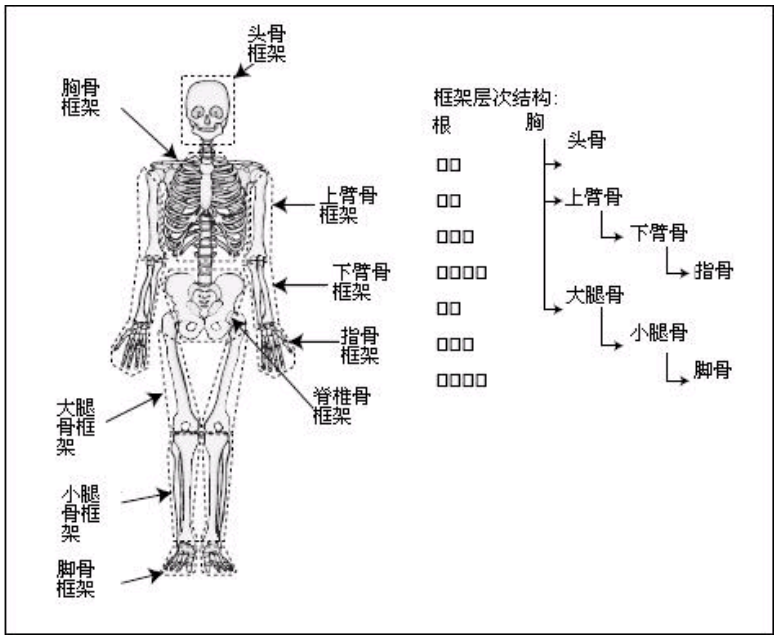
热门文章 HOT ARTICLE
- [电子书下载]游戏设计 — 原理与实践
- [电子书下载]网络游戏开发
- 游戏设计全过程
- [电子书下载]游戏设计技术
- [电子书下载]游戏设计理论
- CS游戏人物模型制作教程
- CG人物插画基本流程
- [转贴]MAX高级人头教程

MORE

您的位置： 编程技巧

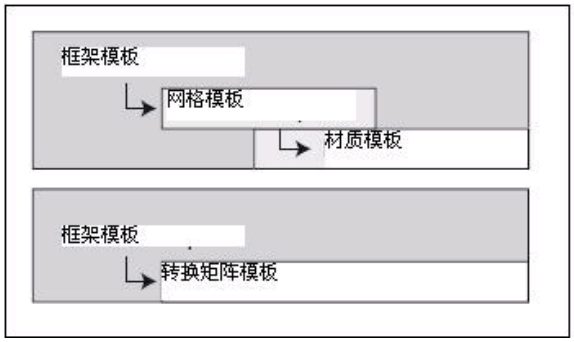
文章标题	游戏中如何加载人物,场景模型		
来源：	[ogdev]	浏览：	[1102]

网格的处理:
在最底层,Direct3D只能处理多边形,不能处理网格.D3DX给Direct3D系统增添了一些用来处理网格的对象.
在最底层,网格可能是由成千上万个顶点组成的,维护起来相当烦琐.幸好,Direct3D有一个自带的文件格式用来存储于网格相关的数据,包括:顶点,表面,法线向量和纹理.这个文件格式就是 .X 文件.
.X文件简介:
.X文件是微软开发的用来描述3D模型的数据文件.它是由许多模板组成的,并且是可以扩展的,这就意味着我们可以用它来存储所有的3D模型.
一个.X文件可以是文本形式的--为了让我们更方便的维护它,或者是二进制文件--为了使得文件更小,也为了更好的保护我们的重要数据..X文件的格式是基于模板的,这使得它看起来有点象C语言中的结构体.
模板解析:
我们用模板来存储一组数据,一般来说, .X文件中的模板是用来存储一些关于网格的信息.比如说,有的模板用来定义顶点,多边形,纹理映射,法线向量等.大多数的模板被包含在其它的网格里,组成了一个模板层次结构:比如说,一个用来定义顶点法线向量的模板可以被包含在一个网格模板中.同样,网格模板可以被包含在一个框架模板中(用来引用一些具体的模板).
我们可以像实例化C语言中结构体那样来实例化一个模板.同样,这里也需要有个"模板引用",这个功能可以让我们只定义一个模板,但在多处使用(与C语言中的变量声明类似).比如说,我们定义了一个网格,在所有用到它的地方我们不是一遍又一遍地声明同一网格,而是只声明它的一个引用即可.
D3DX使得网格的使用更加简单,我们不用关心里面的每一个模板,只需要关心里面的网格模板和框架模板.
使用框架层次:
我们使用框架模板来来把一些相关的网格组合起来,使得管理更加简单.我们可以只创建一个网格对象而在多个框架模板中引用它.这就使得我们实现了数据的重用.
比如说,我们有一个台球的模型网格,因为我们总共需要15个台球,所以我们就来创建15个模板,每个模板中都包含一个对原始台球网格引用.然后我们把每个模板都"固定"在球桌的范围内(通过使用框架转换矩阵模板),使得每个台球实例在框架也即球桌的范围内移动.从本质上讲,我们只是从一个网格模型创建了15个实例而已.
除了可以使用框架来创建一个网格模型的多个实例外, 我们还可以用它们来创建一个分层次的框架结构.一个分层次的框架结构定义了一个场景或者是一组网格.当整个框架移动时,里面所有的框架都会相应移动.
比如说,我们可以把人的骨架想象成一个分层次的框架(如下图所示):



框架层次的的最上方是胸,从胸往下,我们可以用一块块骨头连接起来,胸连着脊,脊连着腿,腿连着脚.沿着这个顺序,所用的骨骼都按一定顺序排列好.
这样我们就有了一个"根"框架---胸骨框架.根框架没有父框架,意思是说它位于整个框架层次的最上面,不从属于任何一个框架.被连接到其它框架的框架被称作"子框架".
当一个框架移动时,它的所有子节点都要跟着移动.比如说, 你要移动上臂,你的下臂跟手都要移动,这我们都很容易理解.然而,如果我们移动我们的手,则只有手会移动.因为它没有子节点.
每个框架都有自己的朝向,在.X文件中的术语叫做框架转换.对于在一个框架层次中转换的情况,对一个较高层次的框架进行转换时,它的所有子节点都要跟着转换.
框架层次对于网格的高级应用和动画技术是非常重要的,在后面的学习中我们将发现这一点.
另外一个使用框架层次的原因是可以将一些特定的场景分离开,然后只对这一小部分场景进行改变,这在游戏中是经常需要的.比如说,我们有一个框架来显示房子, 另外一个框架来显示门,我们可以通过只修改房子的那一框架而不必改变显示房子的那个框架.

.X文件剖析:
剖析一个.X文件对于我们来说是非常重要的,因为我们需要知道文件中的一些重要数据.我们使用IDirectXFile等一组对象来实现对.X文件的剖析---通过枚举其中的模板来实现.
剖析.X文件并没有你想象的那么难,通过扫描整个框架层次,寻找我们要用的模板.最困难的部分是模板是可以嵌套的,所以在枚举的过程我们除了遇到模板外,还有可能遇到模板引用(还记得吗?),如果枚举到的是模板引用,我们要找到其原始数据.下图为大家展示了模板之间的嵌套关系:



下面的这些代码用来打开和分析.X文件.尽管有一些晦涩,不妨先看看:

```
BOOL ParseXFile(char *Filename)
{
IDirectXFile *pDXFile = NULL;
IDirectXFileEnumObject *pDXEnum = NULL;
IDirectXFileData *pDXData = NULL;
// Create the .X file object
if(FAILED(DirectXFileCreate(&pDXFile)))
return FALSE;
// Register the templates in use
// Use the standard retained mode templates from Direct3D
if(FAILED(pDXFile->RegisterTemplates((LPVOID) D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES))) {
pDXFile->Release();
return FALSE;
}
// Create an enumeration object
if(FAILED(pDXFile->CreateEnumObject((LPVOID)Filename, DXFILELOAD_FROMFILE, &pDXEnum))) {
pDXFile->Release();
return FALSE;
}
// Enumerate all top-level templates
while(SUCCEEDED(pDXEnum->GetNextDataObject(&pDXData))) {
ParseXFileData(pDXData);
ReleaseCOM(pDXData);
}
// Release objects
ReleaseCOM(pDXEnum);
ReleaseCOM(pDXFile);
// Return a success
return TRUE;
}
void ParseXFileData(IDirectXFileData *pData)
{
IDirectXFileObject *pSubObj = NULL;
IDirectXFileData *pSubData = NULL;
IDirectXFileDataReference *pDataRef = NULL;
const GUID *pType = NULL;
char *pName = NULL;
DWORD dwSize;
char *pBuffer;
// Get the template type
if(FAILED(pData->GetType(&pType)))
return;
// Get the template name (if any)
if(FAILED(pData->GetName(NULL, &dwSize)))
return;
if(dwSize) {
if((pName = new char[dwSize]) != NULL)
pData->GetName(pName, &dwSize);
}
// Give template a default name if none found
if(pName == NULL) {
if((pName = new char[9]) == NULL)
return;
strcpy(pName, “Template”);
}
// See what the template was and deal with it
// This is where you’ll jump in with your own code
// Scan for embedded templates
while(SUCCEEDED(pData->GetNextObject(&pSubObj))) {
// Process embedded references
if(SUCCEEDED(pSubObj->QueryInterface( IID_IDirectXFileDataReference, (void**)&pDataRef))) {
if(SUCCEEDED(pDataRef->Resolve(&pSubData))) {
ParseXFileData(pSubData);
ReleaseCOM(pSubData);
}
ReleaseCOM(pDataRef);
}
// Process non-referenced embedded templates
if(SUCCEEDED(pSubObj->QueryInterface( IID_IDirectXFileData, (void**)&pSubData))) {
ParseXFileData(pSubData);
```

```
ReleaseCOM(pSubData);
}
ReleaseCOM(pSubObj);
}
// Release name buffer
delete pName;
}
```

ParseXFile()跟ParseXFileData()这两个函数共同完成了.X文件中每个模板的分析.ParseXFile()打开一个.X文件,并且枚举处在模板层次中最高层的模板,每当找到一个模板，就交给ParseXFileData()来处理.

ParseXFileData()处理模板数据.首先,它得到模板类型和模板实例名称.然后就处理模板数据，再就处理子节点的数据，直到所有的数据都处理完毕.
用D3DX处理网格:
D3DX是DirectX自带的一组函数库,使得我们可以站在巨人的肩膀上,同时也使得对网格的处理更加简单.我们主要处理两种网格：Standard 类型和skinned 网格.Standard网格就是普通网格,只有纹理使得它看起来比较好看，再也没有其它修饰,而skinned网格则不同,它是可以"动"的,也就是说运行时它的一部分是会动的.为了达到"动"的目的,我们必须把网格的顶点都连接到一组"骨骼"上,只要"骨骼"动时,顶点就会相应的变化其位置.下面先让我们来了一个这两种类型的网格都会使用到的对象 ID3DXBuffer.

ID3DXBuffer 概述:
一个网格中的数据量有时候是惊人的，特别是如果网格里面还包含有动画方面的信息,那么该如何管理这些重要的信息呢?
D3DX利用ID3DXBuffer对象来保存和检索数据,网格中关于顶点,纹理,材质的信息都被保存到ID3DXBuffer中.
ID3DXBuffer只有两个成员函数,这使得对它的使用非常清晰,第一个函数:

```
void *ID3DXBuffer::GetBufferPointer();
```

这个函数是返回装载数据缓冲区的指针,这个函数返回的是一个void类型的指针,我们可以根据自己的需要进行转换.
第二个函数是:

```
DWORD ID3DXBuffer::GetBufferSize()
```

这个函数能够返回数据缓冲区的大小.

下面让我们来看看如何来创建ID3DXBuffer对象:

```
HRESULT D3DXCreateBuffer(
DWORD NumBytes, //数据缓冲区的大小
ID3DXBuffer **ppvBuffer); //ID3DXBuffer对象指针
```

来看一个具体的例子:

```
ID3DXBuffer *pBuffer;
// 创建缓冲区
if(SUCCEEDED(D3DXCreateBuffer(1024, &pBuffer)))
{
// 返回缓冲区指针
char *pPtr = pBuffer->GetBufferPoint();
// 清空缓冲区
memset(pPtr, 0, pBuffer->GetBufferSize());
// 释放缓冲区
pBuffer->Release();
}
```

Standard网格:
前面我们简述了Standard网格,Standard网格是最简单的网格,所以这也是我们学习的最好起点.使用D3DX来处理Standard网格就更加简单.一会我们就会发现用D3DX来处理一个网格仅仅需要几行代码就可以胜任.首先来看如何来装载网格:

```
HRESULT D3DXLoadMeshFromX(
LPCTSTR pFilename, //要装载的网格文件名称
DWORD Options, //装载方式,稍后介绍
LPDIRECT3DDEVICE9 pD3DDevice, //预先定义的设备对象
LPD3DXBUFFER * ppAdjacency, //用于存放面法线的缓冲区对象
LPD3DXBUFFER * ppMaterials, //用于存放材质的缓冲区对象
LPD3DXBUFFER * ppEffectInstances, //用于存放关于特效的缓冲区
DWORD * pNumMaterials, //材质的个数
LPD3DXMESH * ppMesh //所要创建的ID3DXMesh对象
);
```

有了上面的说明我们就大致了解了如何使用这个函数，给出所要加载的网格文件名称,和相应的ID3DXBuffer,ID3DXMesh对象,还有用来记录材质个数的DWORD变量,就完成了网格的加载.
如果我们所要加载的.X文件中含有多个网格模型,它将把所有的网格模型都组合成一个网格,下面来看一段代码:

```
// g_pD3DDevice 已经定义了的设备对象
ID3DXBuffer *pD3DXMaterials;
DWORD g_dwNumMaterials;
ID3DXMesh *g_pD3DXMesh;
if(FAILED(D3DXLoadMeshFromX(“mesh.x”, D3DXMESH_SYSTEMMEM, g_pD3DDevice, NULL, &pD3DXMaterials,
&g_dwNumMaterials,
&g_pD3DXMesh)))
{
// 错误处理
}
```

当成功加载了网格之后,下面的代码来分析关于材质和纹理的数据信息:

```
D3DXMATERIAL *pMaterials = NULL;
D3DMATERIAL8 *g_pMaterialList = NULL;
IDirect3DTexture8 **g_pTextureList;
// 获得材质链表指针
pMaterials = (D3DXMATERIAL*)pD3DXMaterials->GetBufferPointer();
if(pMaterials != NULL) {
// 申请用来存放材质数据的缓冲区
g_pMaterialList = new D3DMATERIAL8[dwNumMaterials];
// 申请用来存放纹理数据的缓冲区
g_pTextureList = new IDirect3DTexture8[dwNumMaterials];
// 纹理数据的保存
for(DWORD i=0;ig_pMaterialList[i] = pMaterials[i].MatD3D;
// 设置材质的属性,以决定如何跟光交互
g_pMaterialList[i].Ambient = g_pMaterialList[i].Diffuse;
// 保存纹理数据数据
```

```
if(FAILED(D3DXCreateTextureFromFileA(g_pD3DDevice, g_pMaterials[i]->pTextureFilename, &g_pTextureList[i])))
g_pTextureList[i] = NULL;
}
//释放材质缓冲区
pD3DXMaterials->Release();
} else {
//如果没有材质的话，就创建默认的材质
g_dwNumMaterials = 1;
//设置材质属性
g_pMaterialList = new D3DMATERIAL8[1];
g_pMaterialList[i].Diffuse.r = 1.0f;
g_pMaterialList[i].Diffuse.g = 1.0f;
g_pMaterialList[i].Diffuse.b = 1.0f;
g_pMaterialList[i].Diffuse.a = 1.0f;
g_pMaterialList[i].Ambient = g_pMaterialList[i].Diffuse;
// 创建空的纹理
g_pTextureList = new IDirect3DTexture8[1];
g_pTextureList[0] = NULL;
}
```

执行完上面的代码后,我们就得到了用于渲染的材质,纹理数据,接下来的工作就是来渲染我们的网格.

网格的渲染:

网格渲染其实也很简单,ID3DXMesh有一个专门负责渲染的函数:DrawSubset,它用来渲染一些网格的子集,说白了也就是网格的一部分,一个网格可能由于纹理的变化而分成多个部分,就是通过来渲染这一些小区域来完成整个网格的渲染,关键是我们自己要明白是如何划分的,下面的图可以帮助我们理解: