此教程版权归我所有，仅供个人学习使用，请勿转载。商业应用请同我联系。
此教程翻译了《GPU GEM2》中第三章的内容。由于本人水平有限，难免出错，不清楚的地方请大家以原著为准。也欢迎大家和我多多交流。
其中部分图片来自网络，尽量保证了和原书中插图一致。
特别感谢 mtt 重现了文章中的流程图^_^

翻译：clayman
Blog：http://blog.csdn.net/soilwork
clayman_joe@yahoo.com.cn

Inside Geometry Instancing

在交互式程序中，丰富用户体验的重要方法之一就是呈现一个充满大量各种有趣物体的世界。从数不清的草丛、树木到普通杂物：所有这些都能提高画面最终的效果，让用户保持“幻想状态（suspension of disbelief）”。只有用户相信并且融入了这个世界，才会对这个世界充满感情——这就是游戏开发的圣杯（Holy Grail）。

从渲染的观点来看，实现这种效果，无非就是渲染大量小物体，一般情况下，这些物体彼此都很类似，只在颜色、位置以及朝向上有细小的差别。举个例子，比如森林中所有树的几何形状都是很类似的，而在颜色和高度上有很大差别。对用户来说，由外形各异的树组成的森林才真实，才会相信它，从而丰富自己的游戏体验。

但是，使用当前的 GPU 和图形库渲染大量由少量多边形组成的小物体会带来很大的性能损失。诸如 Direct3D 和 OpenGL 之类的图形 API 都不是为了每帧渲染只有少数多边形的物体数千次而设计的。本文将讨论如何使用 Direct3D 把同一几何体渲染为大量独特的实体（instances）。下图是 Back & White 2 中，使用了这一技术的一个例子：

3.1 为何使用 Geometry Instancing （Why Geometry Instancing）

在 Direct3D 中，把三角形数据提交给 GPU 是一个相对很慢的操作。Wloka 2003 显示使用 Direct3D 在 1GHz 的 CPU 上，每秒只能渲染 10000 到 400000 批次（batches）。对于现代的 CPU，可以预测这个值大概在每秒 30000 到 120000 批次之间（对 FPS 为 30frame/sec 系统来说大概每帧 1000 到 4000 批次）。这太少了！这意味着如果我要渲染一片森林，每批次提交一颗树的数据，那么无论每棵树包含多少多边形，都将无法渲染 4000 棵树以上——因为 CPU 已经没有时间来处理其他任务了。这种情况当然是我们不想看到的。在应用程序中，我们希望最小化渲染状态和纹理的改变，同时，在 Direct3D 中使用一次方法调用，在同一批次中对同一三角形进行多次渲染。这样，就能减少 CPU 提交批次的时间，把 CPU 资源留给物理、AI 等其他系统。

3.2 定义（Definitions）

我们先来定义一系列与 geometry instancing 相关的概念。

3.2.1 几何包（Geometry Packet）

A geometry packet is a description of a packet of geometry to be instanced, a collection of vertices and indices。一个几何包可以使用顶点——包括他的位置、纹理坐标、法线、切线空间（tangent space）以及用于 skinning 的骨骼信息——以及顶点流中的索引信息来描述。这样的描述，可以直接映射为一个高效的提交几何体的方法。

几何包是对一个几何体在模型空间进行的抽象描述， 从而可以独立于当前的渲染环境。

下面是对几何包的一种可能的描述，它不但包含了几何体的信息，同时还包含了物体的边界球体信息：

```
struct GeometryPacker
{
    Primitive mPrimType;
    void* mVertice;
    unsigned int mVertexStride;

    unsigned short* mIndices;
    unsigned int mVertexCount;
    unsigned int mIndexCount;

    D3DXVECTOR3 mSphereCentre;
    float mSphereRadius;
}
```

3.2.2 实体属性（Instance Attribute）

对每个实体来说，典型的属性包括模型到世界的坐标变换矩阵，实体颜色以及由 animation player 提供的用于对几何包进行 skin 的骨骼。

```
struct InstanceAttributes
{
    D3DXMATRIX mModelMatrix;
    D3DCOLOR mInstanceColor;
    AnimationPlayer* mAnimationPlayer;
    unsigned int mLOD;
}
```

3.2.3 几何实体（Geometry Instance）

几何实体就是一个几何包与特定属性的集合。他直接联系到一个几何包以及一个将要用于渲染的实体属性，包含了将要提交给 GPU 的关于实体的完整描述。

```
struct GeometryInstance
{
    GeometryPacket* mGeometryPacket;
    InstanceAttributes mInstanceAttributes;
}
```

3.2.4 渲染及纹理环境（Render and Texture Context）

渲染环境指的是当前的 GPU 渲染状态（比如 alpha blending, testing states, active render target 等等）。纹理环境指的则是当前激活（active） 的纹理。通常使用类来对渲染状态和纹理状态进行模块化。

```
class RenderContext
```

```
{
    public:
    //begin the render context and make its render state active
    void Begin(void);
    //End the render context and restore previous render states if necessary
    void End(void);

    private:
    //Any description of the current render state and pixel and vertex shaders.
    //D3DX Effect framework is particularly useful
    ID3Deffect* mEffect;
    //Application-specific render states
    //….
};
class TextureContext
{
    public:
    //set current textures to the appropriate texture stages
    void Apply(void) const;

    private :
    Texture mDiffuseMap;
    Texture mLightMap;
    //……..
}
```

3.2.5 几何批次（Geometry Batch）

几何批次是一系列几何实体的集合，以及用来渲染这个集合的渲染状态和纹理环境。为了简化类的设计，通常直接映射为一次 DrawIndexedPrimitive()方法调用。以下是几何批次类的一个抽象接口：

```
class GeometryBatch
{
    public:
    //remove all instances form the geometry batch
    virtual void ClearInstances(void);
    //add an instance to the collection and return its ID. Return -1 if it can’t accept more instance.
    virtual int AddInstance(GeometryInstance* instance);
    //Commit all instances, to be called once before the render loop begins and after every change to the instances collection
    virtual unsigned int Commit(void) = 0;
    //Update the geometry batch, eventually prepare GPU-specific data ready to be submitted to the driver, fill vertex and
    //index buffers as necessary , to be called once per frame
    virtual void Update(void) = 0;
    //submit the batch to the driver, typically impemented eith a call to DrawIndexedPrimitive
    virtual void Render(void) const = 0;
```

```
    private:
    GeometryInstancesCollection mInstances;
}
```

3.3 实现（Implementation）

引擎的渲染器只能通过 GeometryBatch 的抽象接口来使用 geometry instancing，这样能很好隐藏具体的实体化（instancing）实现，同时，提供管理实体、更新数据、以及渲染批次的服务。这样引擎就能集中于分类（sorting）批次，从而最小化渲染和纹理状态的改变。同时，GeometryBatch 完成具体的实现，并且与 Direct3D 进行通信。

下面使用的伪代码实现了一个简单的渲染循环：

```
//Update phase
Foreach GeometryBatch in ActiveBatchesList
    GeometryBatch.Update();

//Render phase
Foreach RenderJContext
Begin
    RenderContext.BeginRendering();
    RenderContext.CommitStates();

    Foreach TextureContext
    Begin
        TextureContext.Apply();
        Foreach GeometryBatch in the texture context
            GeometryBatch.Render();
    End
End
```

为了能一次更新所有批次并且进行多次渲染，更新和渲染阶段应该分为独立的两部分：这种方法在渲染阴影贴图或者水面的反射以及折射时特别有用。这里我们将讨论 4 种 GeometryBatch 的实现，并且通过比较内存占用量、可控性来分析各种技术的性能特性。

这里是一个大概的摘要：

- **静态批次（static batching）**：执行 instance geometry 最快的方法。每个实体通过一次变换移动到世界坐标，附加上属性值，然后就提交给 GPU。静态批次很简单，但也是可控性最小的一种。
- **动态批次（Dynamic batching）**：执行 insance geometry 最慢的方法。每一帧里，每个经过变换，附加了属性的实体都以流的形式传入 GPU。动态批次可以完美的支持 skinning，也是可控性最强的。
- **Vertex constants instancing**：一种混合的实现方法。每个实体的几何信息都被复制多次，并且一次性把他们复制到 GPU 的缓存中。通过顶点常量，每一帧都重新设置实体属性，使用一个 vertex shader 完成 gemetry instancing。
- **Batching with Geometry Instancing API**。使用 DirectX 9 提供的 Geometry Instancing API，可以获得 GeForce 6 系列显卡完全的硬件支持，这是一种高效而又具有高度可控性的 geometry instancing 方法。与其他几种方法不同的是它不需要把几何包复制到 Direct3D 的顶点流中。

3.3.1 静态批次（Static Batching）

对静态批次来说，我们希望对所有实体进行一次变换之后，复制到一块静态顶点缓冲中。静态批次最大的优点就是高效，

同时几乎市场上所有的 GPU 都能支持这个特性。

为了实现静态批次，先创建一个用来填充经过变化后的几何体的顶点缓冲对象（当然也包括索引缓冲）。需要保证这个这个缓冲足够大，足以储存我们希望处理的所有实体。由于我们只对缓冲进行一次填充，并且不再做修改，因此，可以使用 Direct3D 中的 D3DUSAGE_WRITEONLY 标志，提示驱动程序把缓冲放到速度最快的可用显存中：

```
HRESULT res;
res = lpDevice -> CreateVertexBuffer( MAX_STATIC_BUFFER_SIZE, D3DUSAGE_WRITE, 0, D3DPOOL_MANAGED, &mStaticVertexStream, 0 );
ENGINE_ASSERT(SUCCEEDED(res));
```

根据应用程序的类型或者引擎的内存管理方式，可以选择使用 D3DPOOL_MANAGED 或 D3DPOOL_DEFAULT 标志来创建缓冲。

接下来实现 Commit()方法。它将把需要渲染的经过坐标变换的几何体数据填充到顶点和索引缓冲中。以下是 Commit 方法的伪代码实现：

```
Foreach GeometryInstance in Instances
Begin
    transform geometry in mGeometryPack to world space with instance mModelMatrix
    Apply other instnce attributes(like instace color)
    Copy transformed geometry to the Vertex Buffer
    Copy indices ( with the right offset) to the Index Buffer
    Advance current pointer to the Vertex Buffer
    Advance currect pointer to the Index Buffer
End
```

好了，接下来就只剩使用 DrawIndexedPrimitive()方法，提交这些准备好的数据了。Update()方法和 Render()方法的实现都很简单，这里不具体讨论。

静态批次是渲染大量实体最快的方法，它可以在一个批次中包含不同类型的几何包，但也有一些严重的限制：

- **大内存占用（Large memory footprint）**：根据几何包大小和希望渲染的实体数量，内存占用量可能会变的很大。对于大场景来说，应该预留出几何体所需的空间。Falling back to AGP memory is possible（注：这里应该指的是当显存不够用时，需要把数据分页存放到 AGP memory 中），但这会降低效率，因此，应该尽量避免。
- **不支持多种 LOD（No support for different level of detal）**：由于在提交数据时，所有实体都被一次性复制到顶点缓冲中，因此很难对每种环境都选择一个有效的 LOD 层次，同时，还会导致对多边形数量的预算不正确。可以使用一种半静态的方法来解决这个问题，把特定实体的所有 LOD 层次都放在顶点缓冲中，每一帧选择不同的索引值，来选择实体的正确 LOD。但这样会让实现看起来很笨拙，违反了我们使用这种方法最初的目的：简单并且高效。
- **No support for skinning**
- **不直接支持实体移动（No direct support for moving instances）**：由于效率的原因，实体的移动应该使用 vertex shader 逻辑和动态批次来实现。最终的解决方案其实就是 vertex constants instancing。

  接下来的一种方法将解除这些限制，以牺牲渲染速度换取可控性。

3.3.2 动态批次（DynamicBatching）

动态批次以降低渲染效率为代价，克服了静态批次方法的限制。动态批次最大的优点和静态批次一样，也能在不支持高级编程管道的 GPU 上使用。

首先使用 D3DUSAGE_DYNAMIC 和 D3DPOOL_DEFAULT 标志创建一块顶点缓冲（同样也包括相应的索引缓冲）。这些标志将保证缓冲处于最容易进行内存定位的地方，以满足我们动态更新的要求

```
HRESULT res;
res = lpDevice->CreateVertexBuffer(MAX_DYNAMIC_BUFFER_SIZE, D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY, 0 , D3DPOOL_DEFAULT, &mDynamicVertexStream, 0)
```

这里，选择正确的 MAX_DYNAMIC_BUFFER_SIZE 值是很重要的。有两种策略来选择这个值：

- 选择一个可以容纳每一帧里所有可能实体的足够大值。
- 选择一个足够大的值，以保证可以容纳一定量的实体。

第一种策略在一定程度上保证了更新和渲染批次的独立。更新批次意味着对动态缓冲中的所有数据进行数据流化（streaming）；而渲染则只是使用 DrawIndexedPrimitive()方法提交几何数据。当这种方法将会占用大量的图形内存（显存或者 AGP memory），同时，在最差的情况下，这种方法将变的不可靠，因为我们无法保证缓冲在整个应用程序生命期中都足够大。

第二种策略则需要在几何体信息数据流化和渲染之间进行交错：当动态缓冲被填满时，提交几何体进行渲染，同时丢弃缓冲中的数据，准备好填充更多将被数据流化的实体。为了优化性能，使用正确的标志是很重要的，换句话说就是，在每一批实体开始时都使用 D3DLOCK_DISCARD 标志锁定（locking）动态缓冲，此外，对每个将要数据流化的新实体都使用 D3DLOCK_WRITEONLY 标志。这个方法的缺点是每次当批次需要进行渲染时，都需要重新锁定缓冲，以数据流化几何体信息，比如实现阴影影射时。

应该根据应用程序的类型和具体要求来选择不同方法。这里，由于简单和清楚的原因，我们选择了第一种方法，但是也添加了一点点复杂度：动态批次天生支持 skinning，我们顺便对他进行了实现。

Update 方法与之前在 3.3.1 讨论的 Commit()方法很类似，但它需要在每一帧都执行。这里是伪代码的实现;

```
Foreach GeometryInstance in Instances
Begin
    Transform geometry in mGeometryPacket to world space with instance mModelMatrix
    if instance nedds skinning, request a set of bones from mAnimationPlayer and skin geometry
    Apply other instance attributes(like instance color)
    Copy transformd geometry to the Vertex Buffer
    Copy indices (with the right offset) to the Index Buffer
    Advance current pointer to the Vertex Buffer
    Advance current pointer to the Index Buffer
End
```

这种情况下，Render()方法只是简单的调用 DrawIndexedPrimitive()方法而已。

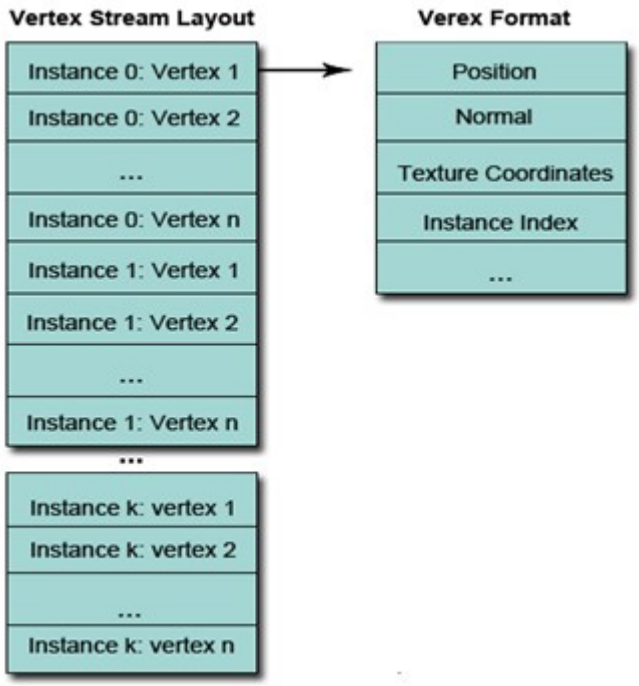
3.3.3 Vertex Constants Instancing

在 vertex constants instancing 方法中，我们利用顶点常量来储存实体属性。就渲染性能而言，顶点常量批次是非常快的，同时支持实体位置的移动，但这些特点都是以牺牲可控性为代价的。

以下是这种方法主要的限制：

- 根据常理数值的大小，每批次的实体数量是受限制的；通常对一次方法调用来说，批次中不会超过 50 到 100 个实体。但是，这足以满足减少 CPU 调用绘图函数的负载。
- 不支持 skinning；顶点常量全部用于储存实体属性了
- 需要支持 vertex shaders 的硬件

首先，需要准备一块静态的顶点缓冲（同样包括索引缓冲）来储存同一几何包的多个副本，每个副本都以模型坐标空间保存，并且对应批次中的一个实体。



必须更新最初的顶点格式，为每个顶点添加一个整数索引值。对每个实体来说，这个值将是一个常量，标志了特定几何包属于哪个实体。这和 palette skinning 有些类似，每个顶点都包含了一个索引，指向将会影响他的一个或多个骨骼。

更新之后的顶点格式如下：

```
Stuct InstanceVertex
{
    D3DVECTOR3    mPosition;
    //other properties……
    WORD      mInstanceIndex[4];    //Direct3D requires SHORT4
};
```

在所有实体数据都添加到几何批次之后，Commit()方法将按照正确的设计，准备好顶点缓冲。

接下来就是为每个需要渲染的实体加载属性。我们假设属性只包括描述实体位置和朝向的模型矩阵，以及实体颜色。

对于支持 DirectX9 系列的 GPU 来说，最多能使用 256 个顶点常量：我们使用其中的 200 个来保存实体属性。在我们所举的例子中，每个实体需要 4 个常量储存模型矩阵，1 个常量储存颜色，这样每个实体需要 5 个常量，因此每批次最多包含 40 个实体。

以下是 Update()方法。实际的实体将在 vertex shader 进行处理。

```
D3DVECTOR4  instancesData[MAX_NUMBER_OF_CONSTANTS];
unsigned int count = 0;
for(unsigned int i=0; i<GetInstancesCount(); ++i)
{
    //write model matrix
    instancesData[count++] = *(D3DXVECTOR4*) & mInstances[i].mModeMatrix.m11;
    instancesData[count++] = *(D3DXVECTOR4*) & mInstances[i].mModelMatrix.m21;
    instancesData[count++] = *(D3DXVECTOR4*) & mInstances[i].mModelMatrix.m31;
    instancesData[count++] = *(D3DXVECTOR4*) & mInstances[i].mModelMatrix.m41;
    //write instance color
    instaceData[count++] = ConverColorToVec4(mInstances[i].mColor);
}
```

```
lpDevice->SetVertexConstants(INSTANCES_DATA_FIRST_CONSTANT, instancesData, count);
```

下面是 vertex shader：

```
//vertex input declaration
struct vsInput
{
    float4 postion : POSITON;
    float3 normal : NORMAL;
    //other vertex data
    int4 instance_index : BLENDINDICES;
};

vsOutput VertexConstantsInstancingVS( in vsInput input)
{
    //get the instance index; the index is premultiplied by 5 to take account of the number of constants used by each instance
    int instanceIndex = ((int[4])(input.instance_index))[0];
    //access each row of the instance model matrix
    float4 m0 = InstanceData[instanceIndex + 0];
    float4 m1 = InstanceData[instanceIndex + 1];
    float4 m2 = InstanceData[instanceIndex + 2];
    float4 m3 = InstanceData[instanceIndex + 3];
    //construct the model matrix
    float4x4 modelMatrix = {m0, m1, m2, m3}
    //get the instance color
    float instanceColor = InstanceData[instanceIndex + 4];
    //transform input position and normal to world space with the instance model matrix
    float4 worldPostion = mul(input.position, modelMatrix);
    float3 worldNormal = mul(input.normal, modelMatrix;
    //output posion, normal and color
    output.position = mul(worldPostion, ViewProjectionMatrix);
    output.normal = mul(worldPostion,ViewProjectionMatrix);
    output.color = instanceColor;
    //output other vertex data
}
```

Render()方法设置观察和投影矩阵，并且调用一次 DrawIndexedPrimitive()方法提交所有实体。

实际代码中，可以把模型空间的旋转部分储存为一个四元数（quaternion）,从而节约 2 个常量，把最大实体数增加到 70 左右。之后，在 vertex shader 中重新构造矩阵，当然，这也增加了编码的复杂度和执行时间。

3.3.4 Batching with the Geometry Instancing API

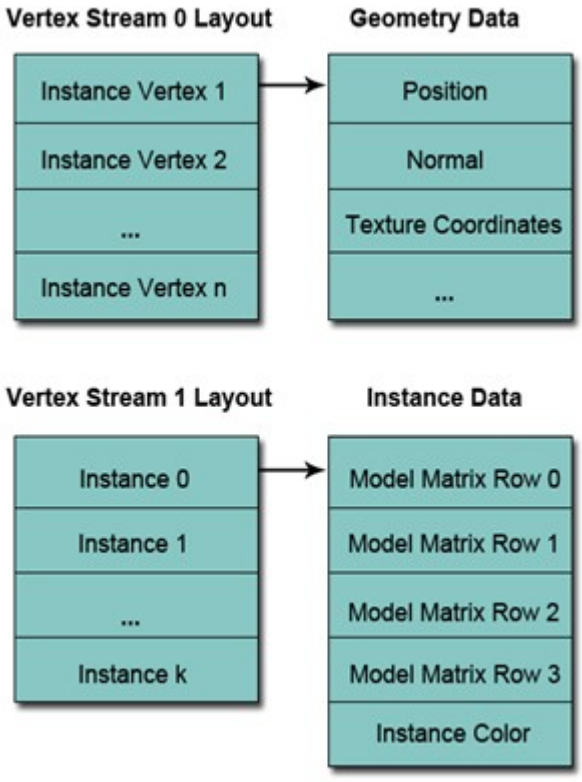
最后介绍的一种方法就是在 DirectX9 中引入的，完全可由 Geforce 6 系列 GPU 硬件实现的几何实体 API 批次。随着原来越多的硬件支持几何实体 API，这项技术将变的更加有趣，它只需要占用非常少的内存，另外也不需要太多 CPU 的干涉。它唯一的缺点就是只能处理来自同一几何包的实体。

DirectX9 提供了以下函数来访问几何实体 API：

```
HRESULT SetStreamSourceFreq( UINT StreamNumber, UINT FrequencyParameter);
```

StreamNumber 是目标数据流的索引，FrequencyParameter 表示每个顶点包含的实体数量。

我们首先创建 2 快顶点缓冲：一块静态缓冲，用来储存将被多次实体化的单一几何包；一块动态缓冲，用来储存实体数据。两个数据流如下图所示：



Commit()必须保证所有几何体都使用了同一几何包，并且把几何体的信息复制到静态缓冲中。

Update()只需简单的把所有实体属性复制到动态缓冲中。虽然它和动态批次中的 Update()方法很类似，但是却最小化了 CPU 的干涉和图形总线（AGP 或者 PCI－E）带宽。此外，我们可以分配一块足够大的顶点缓冲，来满足所有实体属性的需求，而不必担心显存消耗，因为每个实体属性只会占用整个几何包内存消耗的一小部分。

Render()方法使用正确流频率（stream frequency）设置好两个流，之后调用 DrawIndexedPrimitive()方法渲染同一批次中的所有实体，其代码如下：

```
unsigned int instancesCount = GetInstancesCount();
//set u stream source frequency for the first stream to render instancesCount instances
//D3DSTREAMSOURCE_INDEXEDDATA tell Direct3D we’ll use indexed geometry for instancing
lpDevice->SetStreamSourceFreq(0, D3DSTREAMSOURCE_INDEXEDDATA | instancesCount);
//set up first stream source with the vertex buffer containing geometry for the geometry packet
lpDevice->setStreamSource(0, mGeometryInstancingVB[0], 0, mGeometryPacketDeck);
//set up stream source frequency for the second stream; each set of instance attributes describes one instance to be rendered
lpDevice->SetstreamSouceFreq(1, D3DSTREAMSOURCE_INDEXEDDATA | 1);
// set up second stream source with the vertex buffer containing all instances’ attributes
pd3dDevice->SetStreamSource(1, mGeometryInstancingVB[0], 0, mInstancesDataVertexDecl);
```

GPU 通过虚拟复制（virtually duplicating）把顶点从第一个流打包到第二个流中。vertex shader 的输入参数包括顶点在模型空间下的位置，以及额外的用来把模型矩阵变换到世界空间下的实体属性。代码如下：

```
// vertex input declaration
struct vsInput
{
    //stream 0
```

```
        float4 position : POSITION;
        float3 normal    : NORMAL;
        //stream 1
        float4 model_matrix0  :   TEXCOORD0;
        float4 model_matrix1  :   TEXCOORD1;
        float4 model_matrix2  :   TEXCOORD2;
        float4 model_matrix3  :   TEXCOORD3;

        float4 instance_color   :   D3DCOLOR;
};

vsOutput geometryInstancingVS(in vsInput input)
{
        //construct the model matrix
        float4x4 modelMatrix =
        {
                input.model_matrix0,
                input.model_matrix1,
                input.model_matrix2,
                input.model_matrix3,
        }
        //transform inut position and normal to world space with the instance model matrix
        float4 worldPosition = mul(input.position, modelMatrix);
        float3 worldNormal = mul(input.normal,modelMatrix);
        //output positon, normal ,and color
        output.positon = mul(worldPostion,ViewProjectionMatrix);
        output.normal = mul(worldNormal,ViewProjectionMatrix);
        output.color = int.instance_color;
        //output other vertex data…..
}
```

由于最小化了 CPU 负载和内存占用，这种技术能高效的渲染同一几何体的大量副本，因此，也是游戏中理想的解决方案。当然，它的缺点在于需要硬件功能的支持，此外，也不能轻易实现 skinning。

如果需要实现 skinning，可以尝试把所有实体的所有骨骼信息储存为一张纹理，之后为相应的实体选择正确的骨骼，这需要用到 Shader Model3.0 中的顶点纹理访问功能。如果使用这种技术，那么访问顶点纹理带来的性能消耗是不确定的，应该实现进行测试。

3．4 结论

本文描述了几何实体的概念，并且描述了 4 中不同的技术，来达到高效渲染同一几何体多次的目的。每一种技术都有有点和缺点，没有哪种单一的方法能完美解决游戏场景中可能遇到的问题。应该根据应用程序的类型和渲染的物体种类来选择相应的方法。

一下是一些场景中建议使用的方法：

- 对于包含了同一几何体大量静态实体的室内场景，由于他们很少移动，静态批次是最好的选择。
- 包含了大量动画实体的户外场景，比如包含了数百战士的即时战略游戏，动态批次也许是最好的选择。

- 包含了大量蔬菜和树木的户外场景，通常需要对他们的属性进行修改（比如实现随风而动的效果），以及一些粒子系统，几何批次 API 也许就是最好的选择。

通常，同一应用程序会用到两个以上的方法。这种情况下，使用一个抽象的几何批次接口隐藏具体实现，能让引擎更容易进行模块化和管理。这样，对整个程序来说，几何实体化的实现工作也能减少很多。

（图中，静态的建筑使用了静态批次，而树则使用了几何实体 API）

PS：完整的demo大家可以参考NVIDIA SDK中的示例Instancing，也可以直接在这里下载。另外也可参考DirectX SDK中的示例Instancing。