Lecture11 让画面动起来

> http://opengameart.org 开源游戏艺术资产

1. Unity Shader 中的时间内置变量

动画效果往往是把时间添加到一些变量的计算中

Unity Shader 提供了一系列关于时间的内置变量，使得方便在 Shader 中访问这些时间变量

名称	类型	描述
_Time	float4	t 是自场景加载开始所经过的时间，4 个分量的值分别为 (t/20, t, 2t, 3t)
_SinTime	float4	t 是时间的正弦值，4 个分量的值分别为 (t/8. t/4, t/2, t)
_CosTime	float4	t 是时间的余弦值，4 个分量的值分别为 (t/8. t/4, t/2, t)
unity_DeltaTime	float4	dt 是时间增量，4 个分量的值分别是 (dt, 1/dt, smoothDt, 1/smoothDt)

2. 纹理动画

序列帧动画

依次播放一系列关键帧图像，当播放速度达到一定数值时，看起来就像一个连续的动画

- 优点：灵活性很强
- 缺点：美术工程量较大

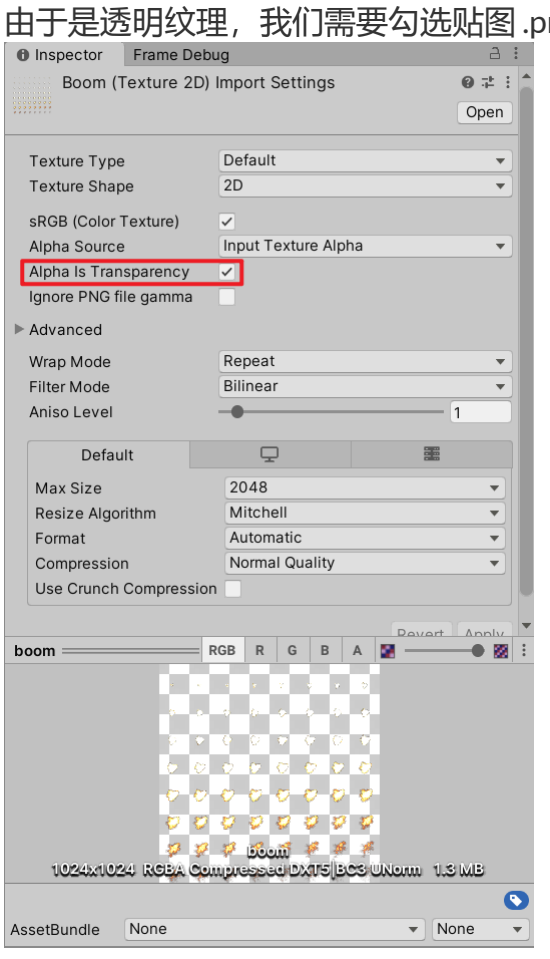
实践 - 关键帧 Shader

序列帧图像通常是透明纹理，需要设置 Pass 的状态

```
Shader "Unity Shaders Book/Chapter 11/Image Sequence Animation" {
   Properties {
      _Color ("Color Tint", Color) = (1, 1, 1, 1)
      _MainTex ("Image Sequence", 2D) = "white" {}
       _HorizontalAmount ("Horizontal Amount", Float) = 4 // 水平方向关键帧数量
       _VerticalAmount ("Vertical Amount", Float) = 4 // 竖直方向关键帧数量
       _Speed ("Speed", Range(1, 100)) = 30 // 控制序列帧动画的播放速度
   }
   SubShader {
       // 序列帧图像通常是透明纹理
       // 半透明图像的标配设置
      Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}

      Pass {
         Tags { "LightMode"="ForwardBase" }

         ZWrite Off
         Blend SrcAlpha OneMinusSrcAlpha

         CGPROGRAM

         #pragma vertex vert
         #pragma fragment frag

         #include "UnityCG.cginc"

         fixed4 _Color;
         sampler2D _MainTex;
         float4 _MainTex_ST;
         float _HorizontalAmount;
         float _VerticalAmount;
         float _Speed;

         struct a2v {
             float4 vertex : POSITION;
             float2 texcoord : TEXCOORD0;
```

```
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        v2f vert (a2v v) {
           v2f o;
           o.pos = UnityObjectToClipPos(v.vertex);
           o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
           return o;
        }

        fixed4 frag (v2f i) : SV_Target {
           float time = floor(_Time.y * _Speed);
            // 计算了关键帧所在行列索引数
           float row = floor(time / _HorizontalAmount);
           float column = time - row * _HorizontalAmount;

//           half2 uv = float2(i.uv.x /_HorizontalAmount, i.uv.y / _VerticalAmount);
//           uv.x += column / _HorizontalAmount;
//           uv.y -= row / _VerticalAmount;
           half2 uv = i.uv + half2(column, -row);
           uv.x /=  _HorizontalAmount;
           uv.y /= _VerticalAmount;

           fixed4 c = tex2D(_MainTex, uv);
           c.rgb *= _Color;

           return c;
        }

        ENDCG
      }
   }
   FallBack "Transparent/VertexLit"
}
```

- 由于是透明纹理，我们需要勾选贴图.png中的`Alpha Is Transparency`属性



滚动的背景

很多 2D 游戏都是使用不断滚动的背景来模拟游戏角色在场景中穿梭，这些背景包含了多个 layer 来模拟视差效果

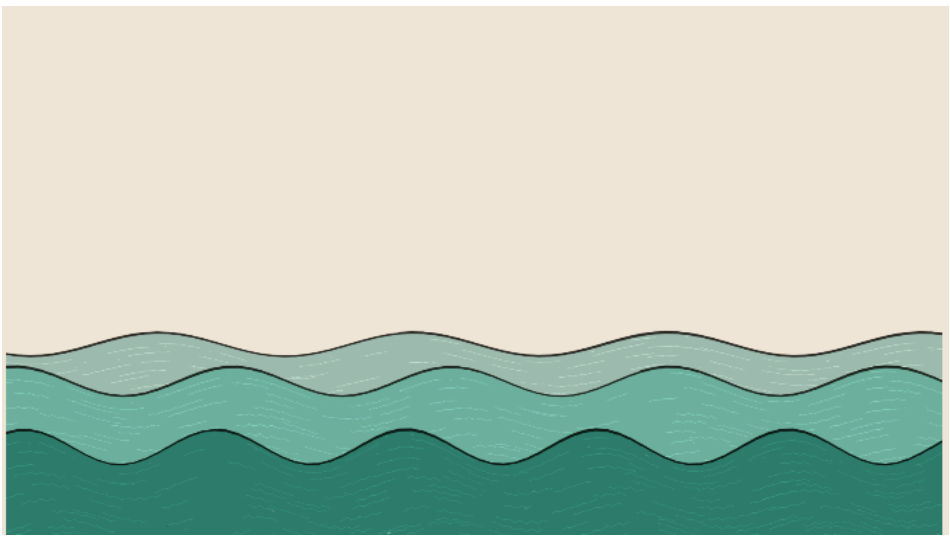
```
Shader "Unity Shaders Book/Chapter 11/Scrolling Background" {
    Properties {
        _MainTex ("Base Layer (RGB)", 2D) = "white" {} // 第一层（较远）
        _DetailTex ("2nd Layer (RGB)", 2D) = "white" {} // 第二层（较近）
        _ScrollX ("Base layer Scroll Speed", Float) = 1.0 // 一层滚动速度
        _Scroll2X ("2nd layer Scroll Speed", Float) = 1.0 // 二层滚动速度
        _Multiplier ("Layer Multiplier", Float) = 1 // 控制纹理的整体亮度
    }
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry"}

        Pass {
            Tags { "LightMode"="ForwardBase" }

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            sampler2D _MainTex;
            sampler2D _DetailTex;
            float4 _MainTex_ST;
            float4 _DetailTex_ST;
            float _ScrollX;
            float _Scroll2X;
            float _Multiplier;

            struct a2v {
                float4 vertex : POSITION;
                float4 texcoord : TEXCOORD0;
            };

            struct v2f {
                float4 pos : SV_POSITION;
                float4 uv : TEXCOORD0;
            };

            v2f vert (a2v v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);

                // 计算两层背景的纹理坐标+使用_Time.y变量在水平方向对纹理坐标进行偏移
                o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex) + frac(float2(_ScrollX, 0.0) *
_Time.y);
                o.uv.zw = TRANSFORM_TEX(v.texcoord, _DetailTex) + frac(float2(_Scroll2X, 0.0)
* _Time.y);

                return o;
            }

            fixed4 frag (v2f i) : SV_Target {
                fixed4 firstLayer = tex2D(_MainTex, i.uv.xy);
                fixed4 secondLayer = tex2D(_DetailTex, i.uv.zw);

                // 使用第二层纹理的透明通道来混合两张纹理
                // 这里的插值是一个很好的图层叠加trick
                fixed4 c = lerp(firstLayer, secondLayer, secondLayer.a);
```

```
            c.rgb *= _Multiplier;

            return c;
        }

        ENDCG
    }
  }
  FallBack "VertexLit"
}
```

3. 顶点动画

顶点动画常常用于模拟飘动的旗帜、湍流的小溪等效果

流动的小溪



河流的模拟是顶点动画最常用的效果之一，正弦函数等可以模拟水流的波动效果

```
Shader "Unity Shaders Book/Chapter 11/Water" {
  Properties {
    _MainTex ("Main Tex", 2D) = "white" {} // 河流纹理
    _Color ("Color Tint", Color) = (1, 1, 1, 1) // 整体颜色
    _Magnitude ("Distortion Magnitude", Float) = 1 // 水流波动幅度
    _Frequency ("Distortion Frequency", Float) = 1 // 波动频率
    _InvWaveLength ("Distortion Inverse Wave Length", Float) = 10 // 波长的倒数
    _Speed ("Speed", Float) = 0.5 // 河流纹理移动速度
  }
  SubShader {
    // Need to disable batching because of the vertex animation
    // 一些SubShader在使用Unity的批处理功能时会出现问题，可以通过DisableBatching标签来指明是否对该SubShader使用批处理
    // 这些特殊处理的Shader指包含了模型空间顶点动画的Shader
    Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" "DisableBatching"="True"}

    Pass {
      Tags { "LightMode"="ForwardBase" }

      ZWrite Off
      Blend SrcAlpha OneMinusSrcAlpha
      Cull Off // 关闭剔除功能，让水流的每个面都能显示

      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag

      #include "UnityCG.cginc"

      sampler2D _MainTex;
      float4 _MainTex_ST;
      fixed4 _Color;
      float _Magnitude;
```

```
        float _Frequency;
        float _InvWaveLength;
        float _Speed;

        struct a2v {
            float4 vertex : POSITION;
            float4 texcoord : TEXCOORD0;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        v2f vert(a2v v) {
            v2f o;

            float4 offset;
            offset.yzw = float3(0.0, 0.0, 0.0);
            offset.x = sin(_Frequency * _Time.y + v.vertex.z * _InvWaveLength) *
_Magnitude; // 模型空间x方向上的偏移
            o.pos = UnityObjectToClipPos(v.vertex + offset);

            o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
            o.uv +=  float2(0.0, _Time.y * _Speed);

            return o;
        }

        fixed4 frag(v2f i) : SV_Target {
            fixed4 c = tex2D(_MainTex, i.uv);
            c.rgb *= _Color.rgb;

            return c;
        }

        ENDCG
    }
  }
  FallBack "Transparent/VertexLit"
}
```

广告牌 Billboarding

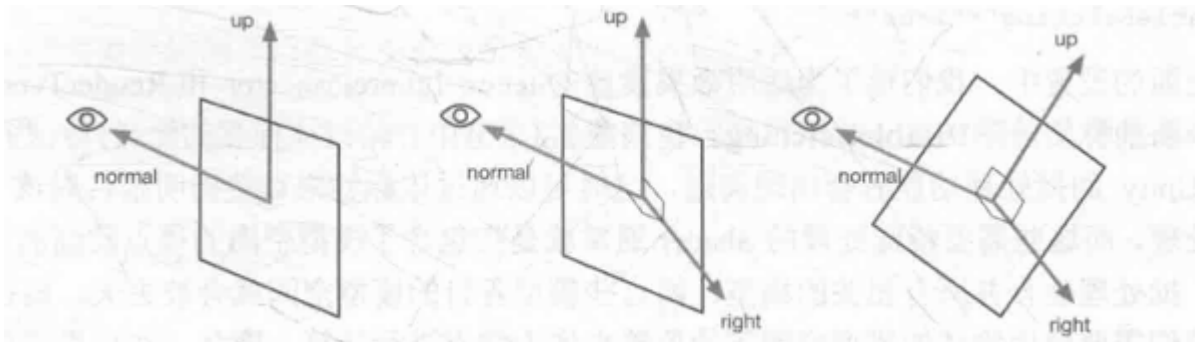
根据视角方向旋转一个被纹理着色的多边形，使得多边形看起来好像总是面对摄像机
广告牌技术的本质是构建旋转矩阵，需要 3 个基向量和1 个锚点

- 表面法线 normal
- 指向上的方向 up
- 指向右的方向 right
- 锚点 anchor location（在旋转过程中固定不变）

计算过程

通过初始计算得到目标的表面法线和指向上的方向，两者往往是不垂直的，但是两者其中之一是固定的
我们假设法线方向是固定的

- 首先，我们根据初始的表面法线和指向上的方向来确定目标方向指向右的方向（通过叉积操作）
  - right = up × normal
- 对其进行归一化后，再有法线方向和指向右的方向计算出正交的指向上的方向即可
  - up' = normal x right



▲图 11.5　法线固定（总是指向视角方向）时，计算广告牌技术中的 3 个正交基的过程

实践 - 广告牌 Shader

```
Shader "Unity Shaders Book/Chapter 11/Billboard" {
    Properties {
        _MainTex ("Main Tex", 2D) = "white" {} // 广告牌显示的透明纹理
        _Color ("Color Tint", Color) = (1, 1, 1, 1) // 整体颜色
        _VerticalBillboarding ("Vertical Restraints", Range(0, 1)) = 1
    }
    SubShader {
        // Need to disable batching because of the vertex animation
        Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"
"DisableBatching"="True"}

        Pass {
            Tags { "LightMode"="ForwardBase" }

            ZWrite Off
            Blend SrcAlpha OneMinusSrcAlpha
            Cull Off // 显示广告牌的每个面

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "Lighting.cginc"

            sampler2D _MainTex;
            float4 _MainTex_ST;
            fixed4 _Color;
            fixed _VerticalBillboarding;

            struct a2v {
                float4 vertex : POSITION;
                float4 texcoord : TEXCOORD0;
            };

            struct v2f {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
            };

            v2f vert (a2v v) {
                v2f o;

                // Suppose the center in object space is fixed
                float3 center = float3(0, 0, 0);
                float3 viewer = mul(unity_WorldToObject,float4(_WorldSpaceCameraPos, 1));

                float3 normalDir = viewer - center;
                // If _VerticalBillboarding equals 1, we use the desired view dir as the
normal dir
                // Which means the normal dir is fixed
                // Or if _VerticalBillboarding equals 0, the y of normal is 0
```

```
            // Which means the up dir is fixed
            normalDir.y = normalDir.y * _VerticalBillboarding;
            normalDir = normalize(normalDir);
            // Get the approximate up dir
            // If normal dir is already towards up, then the up dir is towards front
            float3 upDir = abs(normalDir.y) > 0.999 ? float3(0, 0, 1) : float3(0, 1, 0);
            float3 rightDir = normalize(cross(upDir, normalDir));
            upDir = normalize(cross(normalDir, rightDir));

            // Use the three vectors to rotate the quad
            float3 centerOffs = v.vertex.xyz - center;
            float3 localPos = center + rightDir * centerOffs.x + upDir * centerOffs.y +
normalDir * centerOffs.z;

            o.pos = UnityObjectToClipPos(float4(localPos, 1));
            o.uv = TRANSFORM_TEX(v.texcoord,_MainTex);

            return o;
        }

        fixed4 frag (v2f i) : SV_Target {
            fixed4 c = tex2D (_MainTex, i.uv);
            c.rgb *= _Color.rgb;

            return c;
        }

        ENDCG
    }
  }
  FallBack "Transparent/VertexLit"
}
```

- 上面的例子中，我们使用 Unity 自带的四边形 Quad 作为广告牌，而不使用自带的平面 Plant，我们的代码是建立在一个竖直摆放的多边形的基础上

注意事项

- 模型空间下的定点动画无法跟批处理一起使用，它会破坏动画效果，这时候我们通过 SubShader 的 DisableBatching 标签来强制取消对 Unity Shader 的批处理，取消批处理会带来一定的性能下降，增加 Draw Call
- 如果我们想要对包含顶点动画的物体添加阴影，那么使用内置的包含阴影的 Pass 无法得到正确的阴影效果，Unity 的阴影绘制会调用 ShadowCaster Pass，然而这个 Pass 并没有进行相关的定点动画，所以我们要提供自定义的 ShadowCaster Pass