# Drawing a Crowd

David Gosselin Pedro V. Sander Jason L. Mitchell

3D Application Research Group
ATI Research

## Introduction

In this chapter, we present a technique for efficiently rendering a large crowd of characters while taking steps to avoid a repetitious appearance. There are many typical scenes in games, such as large battles or stadium crowds, which require the rendering of large numbers of characters. We will describe techniques used to draw such crowd scenes, including methods for rendering multiple characters per draw call, each with their own unique animation. We will also outline tradeoffs which can be made between vertex and pixel processing in order to increase vertex throughput (the typical bottleneck in a crowd scene). Next, we will discuss a set of pixel shading tricks to allow the individual characters to have a unique look, despite being drawn from the same instanced geometry as the other characters in the scene. We will conclude with a discussion of instancing of shadow geometry.

## Instancing

In order to achieve our goal of drawing over a thousand characters on screen simultaneously, we must first reduce the number of API calls needed to draw the geometry. If we were to try to make a thousand or more draw calls through the API, we would quickly get swamped by API overhead and setup costs. Clearly, this means that we need to draw several characters per draw call. In order to accomplish this, we pack a number of instances of character vertex data into a single vertex buffer. We do all of the skinning on the graphics hardware, so we pack multiple character transforms into the constant store for each draw call. This allows us to draw several unique instances of the character each with its own unique animation in a single draw call.

Since we plan to do all character skinning in the vertex shader, the main factor which limits the number of characters we can draw per API call is the number of vertex shader constants available to store the characters' skeletal animation data. To keep the number of constants used by each character to a reasonable level, we limited the skeleton of each character to only twenty bones. While this number is quite low for a generic character, for a crowd scene this can be enough to create a good character animation. At first glance, this gives us three characters per draw call (20 bones * 4 vectors * 3 characters = 240 constants). Due to the fact that animation data typically contains no

shears, it is possible to shave off one of the columns of each matrix. This brought our total up to 4 characters per draw call (20 bones * 3 vectors * 4 characters = 240 constants). We investigated using a quaternion plus a translation to store each bone transform, which would have further compressed the transforms and allowed us to draw even more characters per draw call. However, the associated vertex shader cost (to effectively turn the quaternion into a transformation matrix) was a bit too high for our purposes and offset the gains made by being able to draw more instanced characters per call. By drawing groups of four characters, we can reduce the number of draw calls to between 250 and 300 for a crowd of a thousand plus characters, which is reasonable for today's hardware. The downside of this tight constant store packing is that there is not a lot of room left over for additional constants. This is not too bad since all of the lighting will be performed in the pixel shader. The few constants needed in the vertex shader are the view/projection matrix and the camera position.

It is also important to reduce the cost of the actual vertex shader processing in order to draw a large crowd, since vertex shading is generally the bottleneck for such scenarios. To save vertex shader operations, we can store our character's normal map in object space rather than tangent space and avoid having to skin the tangent and binormal vector, thereby saving two matrix multiplies. Using this method, we skin the normal vector in the pixel shader once it has been read from the normal map. This technique requires that we pass the blended skinning matrix down to the pixel shader. This blended matrix is computed before skinning the position in the vertex shader. The HLSL shader code for assembling and blending the matrices in the vertex shader is given below:

```
float4 rowA[80];
float4 rowB[80];
float4 rowC[80];
float4x4 SiComputeSkinningMatrix3Rows (float4 aWeights,
                                       int4 aIndices)
{
    float4x4 mat = 0;
    mat._m33 = 1.0f;
    for (int bone = 0; bone < 4; bone++)
    {
        mat._m00 += (rowA[aIndices[bone]].x * aWeights[bone]);
        mat._m10 += (rowA[aIndices[bone]].y * aWeights[bone]);
        mat._m20 += (rowA[aIndices[bone]].z * aWeights[bone]);
        mat._m30 += (rowA[aIndices[bone]].w * aWeights[bone]);

        mat._m01 += (rowB[aIndices[bone]].x * aWeights[bone]);
        mat._m11 += (rowB[aIndices[bone]].y * aWeights[bone]);
        mat._m21 += (rowB[aIndices[bone]].z * aWeights[bone]);
        mat._m31 += (rowB[aIndices[bone]].w * aWeights[bone]);

        mat._m02 += (rowC[aIndices[bone]].x * aWeights[bone]);
        mat._m12 += (rowC[aIndices[bone]].y * aWeights[bone]);
        mat._m22 += (rowC[aIndices[bone]].z * aWeights[bone]);
        mat._m32 += (rowC[aIndices[bone]].w * aWeights[bone]);
    }
    return mat;
}
```

Once this matrix has been computed, there is still a bit of work left for the vertex shader. It needs to compute the skinned position and then multiply that by the view and projection matrix as shown in the following code.

```
float4x4 mSkinning = SiComputeSkinningMatrix3Rows (i.weights,
                                                   i.indices);
float4 pos = mul (i.pos, mSkinning);
o.worldPos = pos;
o.pos = mul (pos, mVP);
```

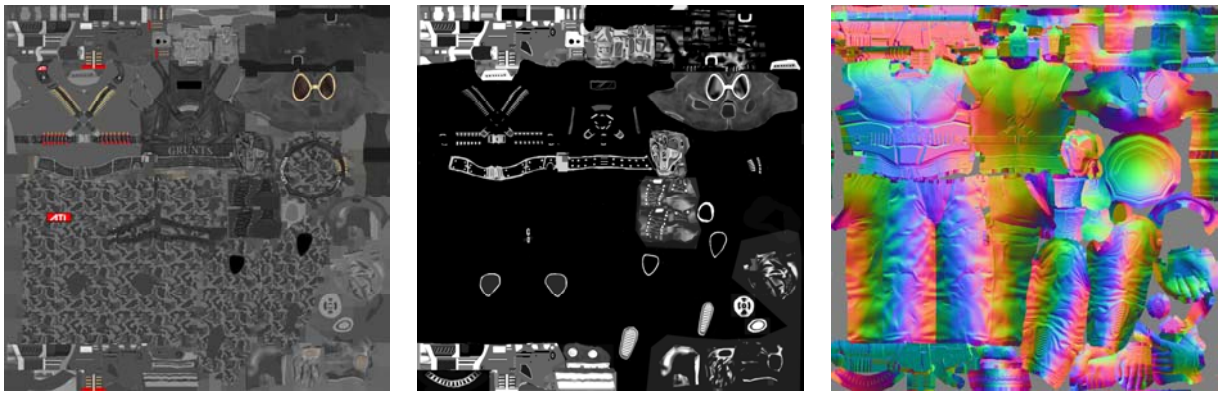
Additionally, the vertex shader needs to compute the view vector.

```
o.viewVec = normalize(worldCamPos – pos);
```

The rest of the vertex shader is dedicated to passing along all of this information to the pixel shader. With these optimizations, the resulting vertex shader code weighs in at around sixty instructions. By combining all of these techniques, we are able to reduce the cost of vertex processing, which is generally the bottleneck when drawing large crowds of characters.

## Character Shading

Even though the characters in our example are relatively low polygon models (1,100 triangles), we can still make them look quite detailed using per-pixel lighting. In our example, we use one directional light to simulate the lighting coming from the sun and up to three local diffuse lights. All of these lights use a normal map generated from a high resolution model as shown in Figure 1c. Since these normals are in object space, we need to skin them by the matrix computed in the vertex shader. The vertex shader passes down a 3×3 matrix and the pixel shader can simply multiply the fetched normal by this interpolated matrix. For diffuse lighting, we just use the Lambertian model.



(a) Base Map (b) Gloss Map (c) Object Space Normal Map

**Figure 1** - Character Textures

In our application, specular lighting is only computed for the directional "sun" light. This is computed based on the view vector computed in the vertex shader. The HLSL code fragment below shows this computation.

```
float3 view = normalize (i.viewVector);
float3 reflectionVec = reflect (view, normal);
float RL = saturate (dot (reflectionVec,
                          (float3)(vLightDirection));
float3 specular = vLightColor * pow (RL, vLightDirection.w) *
                  (gloss+.1);
```

As you can see from the code, a gloss map such as the one shown in Figure 1b is used to attenuate the specular term. This allows for regions of differing shininess. Also note that the specular exponent is packed in with the interpolated light direction to avoid using up an additional constant vector.

In order to give the characters an even more realistic look, an ambient occlusion map is used [Landis02]. This map is also generated from a high resolution model and roughly represents the amount of light that could possibly reach each texel on the model from the external lighting environment. The ambient occlusion map is shown in Figure 2a. This term is multiplied by the final lighting value (both diffuse and specular) and provides a realistic soft look to the character illumination. Since this map is pre-computed it is not technically correct for every frame of animation, however, the results are still quite compelling.

The final term in our basic character lighting is a ground occlusion texture. This is a projected texture that represents an occlusion term based on a given character's position on the terrain. It is similar to the technique described in [Hargreaves04]. In this technique we use our character's position to access a ground-based occlusion texture as shown in Figure 2b. This texture represents roughly the amount of illumination on the character from the "sun" based upon his position on the terrain. By using this texture, we can create the illusion that the characters are being softly shadowed by the terrain as they move around it.

(a) Ambient Occlusion

(b) Ground Occlusion Texture

**Figure 2** - Occlusion Maps

In order to implement this technique, the character's position on the terrain needs to be turned into a texture coordinate. The computation of the texture coordinates for this texture is a simple scale and bias on the world space position, which is computed in the vertex shader and passed down to the pixel shader via a texture coordinate interpolator.

```
float2 floorCoord = vFloorCoordScale.xy * i.worldPos.xz +
                    vFloorCoordBias.xy;
float floorAttenuation = tex2D (tFloor, floorCoord);
```

The product of just these two occlusion terms is shown in Figure 3. The ambient occlusion and terrain occlusion terms are both multiplied by the final lighting (both diffuse and specular).


**Figure 3** - Occlusion Terms

## *Random Coloring*

Up to this point, each of the characters has been shaded identically. Though the animations are distinct and characters are individually shadowed with the terrain occlusion term, they still have a very uniform coloring. While this can look reasonably good—and the sheer number of characters drawn gives an incredible impression—ideally each character could be distinctive in some way. In order to accomplish this task, the shaders need some way to figure out which of the four characters in a given draw call is currently being processed. If the constant store for the vertex shader is packed in such a way that all of the bones for the characters are contiguous within the constant store and the number of bones per character is known, the vertex shader can compute which character is being drawn. The code fragment for this is shown below.

```
float id = int((i.indices.x+.5)/nBones);
```

In our example, we just want the characters to look different, so a vector of four random numbers is generated for each draw group. For a given set of four characters, this set of four random numbers is the same every frame. Given the character ID computed in the above code fragment, the vertex shader can then select one of these four random numbers and send the result down to the pixel shader using the following code:

```
float4 pick = float4(id==0,id==1,id==2,id==3);
float fRand = dot(pick, vRandomVec);
o.texCoord = float3(i.texCoord, fRand);
```

A picture of the crowd drawn using just this random number can be seen in Figure 4.

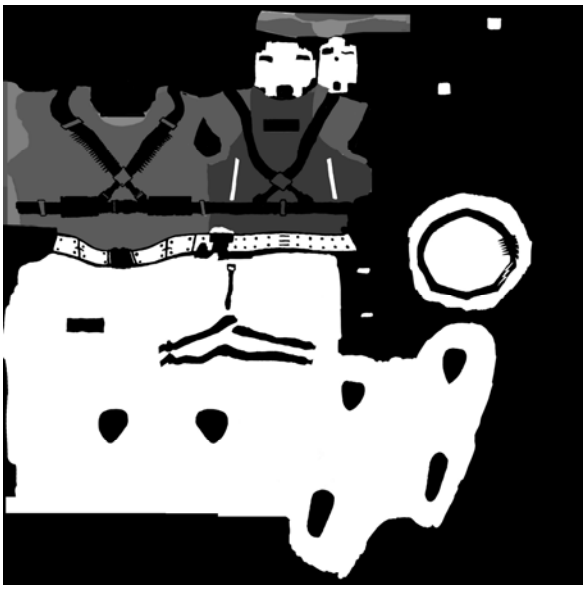

**Figure 4** - Character ID as color

In the pixel shader, this random number is used to change the tinting on the different characters in the scene. To do this, a small 2D texture, shown in Figure 5a, was created. Note that this texture is 16×2. That is, the texture contains 16 pairs of colors. Each pair of colors in this texture represents the darkest and lightest tinting of a given character. In addition to these colors, a mask texture, shown in Figure 5b, was generated to specify which portions of the characters should be tinted. The tinting then occurs by sampling both the dark and bright tint colors from the small 2D texture. The luminance of the base texture is then used to interpolate between these two tint colors. Finally, the mask texture is used to linearly interpolate between the tinted color and the untinted base color. The code fragment for this can be found below.

```
float4 cColorLow  = tex2D(tColor, float2(0, i.texCoord.z));
float4 cColorHigh = tex2D(tColor, float2(1, i.texCoord.z));
base = lerp(base, lerp(cColorLow, cColorHigh,
                       dot(float3(0.2125,0.7154,0.0721),base.xyz)),
            tex2D(tColorAlpha, i.texCoord.xy).r);
```

Using this technique allows us to give each of the characters a slightly different color, increasing the variability of our crowd of instanced characters without increasing the size of our dataset or the number of draw calls made.

(a) Color lookup texture



(b) Lerp texture

**Figure 5** - Character Color Modification Textures

## *Random Decals*

In order to add further uniqueness to the characters, we also use the random number described in the previous section to selectively add decals to the characters' uniforms. To accomplish this, we need two textures: A decal alpha texture which stores a different shade of gray as an id for each decal region (Figure 6a); and a decal color texture, which stores the decal images (Figure 6b). Both of these texture maps use the same texture coordinates as the base map.

In order to selectively add the decals based on the random number, the pixel shader first fetches the decal id of the current sample from the decal alpha texture. If the value is non-zero, the pixel shader then has to determine whether to add the decal or not. The following piece of code determines whether to add the decal based on the random number passed down to the pixel shader and the decal ID:

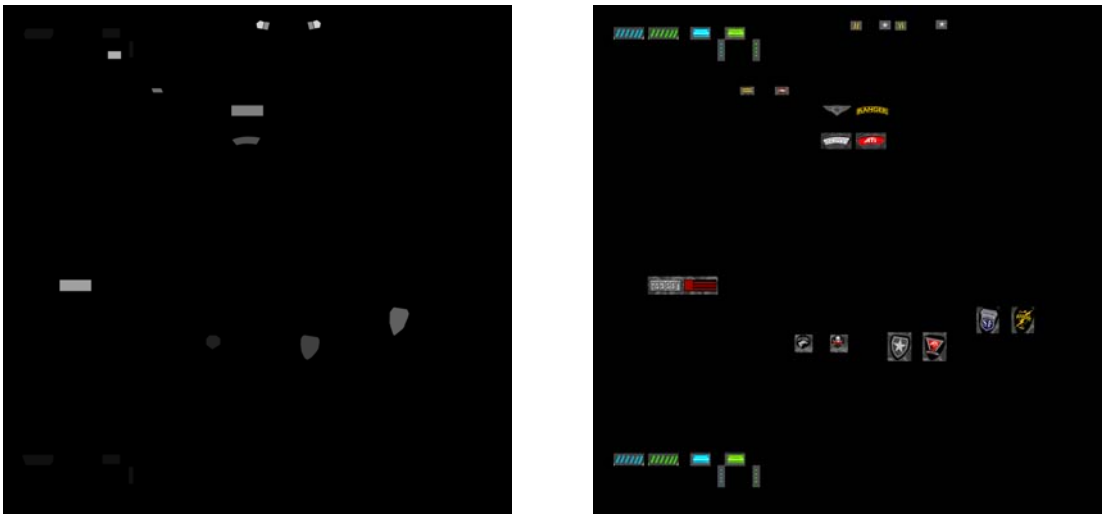
```
float temax = tex2D(tDecalAlpha, i.texCoord.xy).r;
float fDecalAlpha = step(0.2, (temax*i.texCoord.z*1000.) % 1.) > 0;
```

If `fDecalAlpha` is 1, then a decal sample is used for that pixel. Note that, because the computation `fDecalAlpha` uses the decal ID, some of the decals might be turned on while others might be turned off for a given character.

Figure 7 shows some examples of characters with different decals on the same location. By looking at the decal color texture (Figure 6b), one will notice that there are two color images stored for each decal, thus allowing us to randomly pick between the two different decals for a given decal region. One of the images is shifted a little bit to the right from the original decal region identified by the decal alpha texture (Figure 6a). This was possible because our decals were placed in such a way that we could create a shifted copy without overlapping the original. We would expect it to be possible to create a shifted copy in some direction in most cases, unless the character has decals covering a very large portion of the base map. Below is the code to randomly determine whether to use the shifted decal, and to set the base color to be either the value of the base map, or the value of the decal map, depending on `fDecalAlpha`.

```
float2 decalBias = float2(0.068359, 0.0);
//determines whether to bias or not on the decal lookup
decalBias *= int(i.texCoord.z * 4322. % 2.);

base = lerp(base, tex2D(tDecal, i.texCoord.xy+decalBias),
            fDecalAlpha);
```



(a) Decal Alpha Texture (b) Decal Color Texture

**Figure 6** - Character Decal Textures

An important observation is that the decal alpha texture must be sampled using "nearest" as the filter. That is because, if bilinear or trilinear interpolation is used, the border of the decals could assume different shades of gray, thus having different decal IDs, causing it to possibly be selected when it should not, or vice-versa. Even if "nearest" is used, we get some aliasing artifacts at the decal boundaries. In order to remove these artifacts, we make the colors of the decal and the base map match closely near the boundary.


**Figure 7** – Decals Applied

## Shadows

Shadows are an important visual cue for placing characters on the terrain. Given the number of characters we are drawing, it would be impractical to use a sophisticated shadow volume or shadow depth map approach to generate shadows. In our example, we have a fixed directional light and the characters' only animation is to run. These conditions allowed for a sequence of shadow maps to be pre-generated (see Figure 8). These shadow maps are stored as a 2D texture and, with a bit of math in the vertex shader, we can figure out the texture coordinates for each frame of animation. The trick then becomes how to position quads with these textures on the terrain in a convincing way as well as how to draw them efficiently.

**Figure 8** – Character Shadow Texture

Similar to the way we draw the actual characters, the shadow quads are drawn in large instanced batches. A vertex buffer is populated with two hundred quads onto which we will texture map our characters' shadows. We then draw several of these batches per frame to draw the shadows for all of the characters in just a few draw calls. As with the characters themselves, the shadow quads' transformations are handled by the vertex shader hardware.

Like the character animation, vertex shader constant store is the limiting factor which determines how many shadows we can draw in a single call. In order to draw the shadow quads efficiently, we pack the constant store with a very specific transformation that is compressed significantly. A single 4D constant vector is used to represent the transform for each shadow quad, allowing us to draw many shadow quads in one API call. The first three components of each shadow quad transformation vector represent the translation of the quad (i.e. its position in 3-space). The last component is divided into two parts. The integer portion is the frame number for the shadow map animation. This is later used to index into the precomputed shadow texture. The fractional part of the last component represents the slope of the running character. This slope is used to angle the quad to match the terrain. Each quad vertex contains a single "bone" index which is used by the vertex shader to reach into the constant store and get the proper transform. The vertex shader code to unpack and transform the vertices is shown below:

```
float4 vPosFrameSlope = vTransFS[i.indices.x];
float slope = frac (vPosFrameSlope.w);
float frame = vPosFrameSlope.w - slope;
slope = 2.0*slope - 1.0;
float4 pos = float4 (vPosFrameSlope.xyz, 0.0) + i.pos;
pos.y += slope * i.pos.z;
o.worldPos = pos;
```

By using this efficient packing, the shadow quads can be drawn in blocks of two hundred, making them very hardware and API friendly. In order to avoid incorrect Z occlusions, the vertex shader performs a pseudo Z-bias. The vertex shader code for this is shown below:

```
o.pos = mul (pos, mVP);
float invCamDistance = 1.0/sqrt (dot (pos - worldCamPos,
                                      pos - worldCamPos));
o.pos.z -= 2000.0 * invCamDistance;
```

The pixel shader then looks up the proper frame from the shadow map texture. In order to keep the shadows consistent with the character lighting, the shadow quad pixel shader also needs to perform the same dimming based on the ground occlusion texture. The results of all these steps can be seen in Figure 9 which shows a screenshot from the final demo.


**Figure 9**: Final Result

## Conclusion

We have presented a technique for efficiently rendering large crowds of characters in real time using standard APIs. We have demonstrated processing tradeoffs that can be made to reduce vertex shader load in order to increase the number of characters that can be drawn in a given scene. We have also discussed a number of pixel shader techniques which allow us to reduce the appearance of repetition in the crowds of instanced characters. Finally, we concluded with a discussion of instancing of shadow geometry to further integrate our animated crowd of characters into our scene.

While we have employed some clever tricks to implement instancing, we have done it in a robust manner on existing APIs. In fact, there is no reason that many of these techniques cannot be ported all the way down to 1.1 shader hardware. Microsoft has also recognized the importance of instancing for scenarios like large crowd scenes and has retrofitted an instancing API into DirectX 9. This allows devices which support instancing to use one `DrawIndexedPrimitive()` call to draw multiple instances of the same object with unique per-instance data such as transforms. This only works with indexed primitives. We refer you to the latest DirectX SDK for documentation and sample code which illustrates proper usage of this new instancing API.

# References

[Landis02] Landis, Hayden, "Production-Ready Global Illumination," RenderMan in Production (SIGGRAPH 2002): Course 16

[Hargreaves04] Hargreaves, Shawn, “Hemisphere Lighting with Radiosity Maps,” *ShaderX2*, Wordware Publishing Inc, 2003pp113-122