A Rendering Pipeline for Real-time Crowds
Benjamín Hernández and Isaac Rudomin

In motion picture movies, large crowds of computer-generated characters are usually included to produce battle scenes of epic proportions. In video games, it is common to find crowds that conform armies controlled by users (or AI) in real-time strategy games, or crowds made of non-player characters (e.g., groups of spectators in a stadium). In virtual environments, it is common to find crowd simulations that interact with other characters and with their surrounding environment.

In all cases, optimizations such as level of detail and culling should be performed to render the crowds. In this chapter, we propose a parallel approach (implemented on the GPU) for level of detail selection and view frustum culling, allowing us to render crowds made of thousands of characters.

1.1 System Overview

Our rendering pipeline is outlined in figure 1.1. First, all necessary initializations are performed on the CPU. These include loading information stored on disk (e.g., animation frames and polygonal meshes) and information generated as a preprocess (e.g., character positions) or in runtime (e.g., camera parameter updates). This information is used on the GPU to calculate the characters’ new positions, do view frustum culling, assign a specific level of detail (LOD) for each character and for level of detail sorting and character rendering. A brief description of each stage is given next:

- **Populating the Virtual Environment and Behavior.** In these stages we specify the initial positions of all the characters, how they will move through the virtual environment and how they will interact with each other. The result is a set of updated character positions.

- **View Frustum Culling and Level of Detail Assignment.** In this stage we use the characters’ positions to identify which characters will be culled. Additionally, we assign a proper LOD identifier to
1. A Rendering Pipeline for Real-time Crowds

![Figure 1.1. Rendering pipeline for crowd visualization. Dashed arrows correspond to data transferred from main memory to GPU memory only once at initialization.](image)

the characters’ positions inside the view frustum according to their distance to the camera.

- **Level of Detail Sorting.** The output of the View Frustum Culling and Level of Detail Assignation stage is a mixture of positions with different LODs. In this stage we sort each position according to its LOD identifier into appropriate buffers such that all the characters’ positions in a buffer have the same level of detail.

- **Animation and Draw Instanced.** In this stage we will use each sorted buffer to draw the appropriate LOD character mesh using instancing. Instancing allows us to translate the characters across the virtual environment and add visual and geometrical variety to the individuals that conform the crowd.

In the following sections we will present a detailed description of how we implemented these stages.
1.2 Populating the Virtual Environment and Behavior

For simplicity, our virtual environment is a plane. It is parallel to the plane formed by the $xz$ axes. The initial positions of the characters are calculated randomly and stored into a texture.

For behavior, we implemented finite state machines (FSM) as fragment shaders. A finite state machine is used to update the characters' positions following [Rudomin et al. 05, Millán et al. 06], in which a character will consult the value of a labeled world map and follow a very simple FSM that causes it to move right until it reaches the right edge of the map, at which point the agent changes state and starts moving left until it gets to the left edge. However, other GPU approaches such as [Erra et al. 09, Silva et al. 09] are ideal for this pipeline.

Implementing FSM as fragment shaders needs three kinds of textures: a world space texture, an agent texture and an FSM table texture. World space textures encode values for each location in the virtual environment. This would cover many types of maps: height maps, collision maps, interest area maps or action maps. We consider these maps as maps labeled with some value on each pixel. Agent textures have a pixel for each character and encode the state $s$ of the character and its position ($x, z$) in the world map. Finally, the finite state machine is represented as a texture where given a certain state of the character and a certain input, we can obtain a new state and position of the character following the basic algorithm showed in listing 1.1.

```c
given agent i
    state=agent[i].s; x=agent[i].x; z=agent[i].z;
    label=world[x,z];
    agent[i].s=fsm[state,label];
    agent[i].x += fsm[state,label].delta_x;
    agent[i].z += fsm[state,label].delta_z;
```

Listing 1.1. Basic algorithm to implement FSM as fragment shader.

1.3 View Frustum Culling

View frustum culling (VFC) consists of eliminating groups of objects outside the camera’s view frustum. The common approach for VFC is to test the intersection between the objects and the six view frustum planes using their plane equations to determine the visibility of each object. In our

\[\text{1. The reason we recommend methods that use the GPU for behavior, in addition to the fact that these methods can simulate the behavior of tens of thousand characters efficiently, is that approaches using the GPU eliminate the overhead of transferring the new characters’ positions between the CPU and and the GPU on every frame.}\]
case, we implement a simpler method called radar VFC [Puig Placeres 05]. Radar VFC is based on the camera’s referential points. The method tests the objects for being in the view range or not, thus there is no need to calculate the six view frustum plane equations.

On the other hand, objects tested against the view frustum are usually simplified using points or bounding volumes such as bounding boxes (oriented or axis-aligned) or spheres. In our case, we use points (the characters’ positions) together with radar VFC to perform only three tests to determine the character’s visibility. In addition, to avoid the culling of characters that are partially inside the view frustum, we increase the view frustum size by $\Delta$ units\(^2\) (figure 1.2).

As mentioned earlier, radar VFC is based on camera referential points. In other words, the camera has a referential based on the three unit vectors $x, y$ and $z$ as shown in figure 1.3, where $c$ is the position of the camera, $n$ is the center of the near plane and $f$ is the center of the far plane.

The idea behind radar VFC is that once we have the character’s position

\(^2\)The value of $\Delta$ is obtained by visually adjusting the view frustum.
1.3. View Frustum Culling

$p$ to be tested against the view frustum, we find the coordinates of $p$ in the referential and then use this information to find out if the point is inside or outside the view frustum.

The first step is to find the camera’s referential. Let $d$ be the camera’s view direction, $u$ the camera’s up vector, then unit vectors $X, Y, Z$ that conform the referential are calculated using equations 1.1, 1.2 and 1.3.

\[
Z = \|d\| = \sqrt{d_x^2 + d_y^2 + d_z^2} \tag{1.1}
\]

\[
X = \|Z \otimes u\| \tag{1.2}
\]

\[
Y = \|X \otimes Z\| \tag{1.3}
\]

Once we have calculated the referential, the next step is to compute the vector $v$ that goes from the camera center $c$ to the agent’s position $p$ using equation 1.4:

\[
v = p - c \tag{1.4}
\]

Next, the vector $v$ is projected onto the camera referential, i.e., onto $X, Y, Z$ unit vectors.

Radar VFC first tests vector $v$ against the $Z$ unit vector; $v$ is outside the view frustum if its projection $\text{proj}_Z v \notin (\text{Near}_{plane}, \text{Far}_{plane})$. Notice that the projection of a vector $a$ into a unit vector $B$ is given by the dot product of both vectors, i.e. $\text{proj}_B a = a \cdot B$.

If $\text{proj}_Z v \in [\text{Near}_{plane}, \text{Far}_{plane}]$, then vector $v$ is tested against the $Y$ unit vector; $v$ will be outside the view frustum if its projection $\text{proj}_Y v \notin (-h/2 + \Delta, h/2 + \Delta)$ interval. Where $h$ is the height of the view frustum at $v$’s position and $\Delta$ is the value used to increase the view frustum size as shown in figure 1.2. The height $h$ is calculated using equation 1.5:

\[
h = \text{proj}_Z v \times 2 \times \tan \frac{\text{fov}}{2} : \text{fov} \in [0, 2\pi] \tag{1.5}
\]

where $\text{fov}$ is the field of view angle.

If $\text{proj}_Y v \in (-h/2 + \Delta, h/2 + \Delta)$, then vector $v$ is tested against $X$ unit vector, i.e. $v$ is outside the view frustum if its projection $\text{proj}_X v \notin (-w/2 + \Delta, w/2 + \Delta)$ interval, where $w$ is the width of the view frustum, given in equation 1.6:

\[
w = h \times \text{ratio} \tag{1.6}
\]

where $\text{ratio}$ is the aspect ratio value of the view frustum.
1. A Rendering Pipeline for Real-time Crowds

VFC and LOD assignment stages are performed using a geometry shader. This shader receives as input the agent texture that was updated in behavior stage (section 1.2), and it will emit the positions \((x, z)\) which are inside the view frustum and a \(LOD_{id}\). The resultant triplets \((x, y, LOD_{id})\) are stored in a vertex buffer object using the OpenGL transform feedback feature. Listing 1.2 shows the code that performs radar VFC in GLSL.

\[
\begin{align*}
&\text{[vertex program]} \\
&\text{void main(void)} \\
&\quad \{ \\
&\quad \quad \text{glTexCoord[0] = gl_MultitexCoord0;} \\
&\quad \quad \text{gl_Position = gl_Vertex;} \\
&\quad \} \\
&\text{[geometry program]} \\
&\#define INSIDE true \\
&\#define OUTSIDE false \\
&\text{uniform sampler2DRect position; } \\
&\text{uniform float nearPlane, farPlane, tang, ratio, delta;} \\
&\text{uniform vec3 camPos, X, Y, Z;} \\
&\text{bool pointInFrustum(vec3 point)} \\
&\quad \{ \\
&\quad \quad \text{vec3 v = point - camPos;} \\
&\quad \quad \text{// calculating the projection of v into Z unit vector} \\
&\quad \quad \text{float pcz = dot(v, Z);} \\
&\quad \quad \text{// First test: test against Z unit vector} \\
&\quad \quad \text{if (pcz > farPlane || pcz < nearPlane)} \\
&\quad \quad \quad \text{return OUTSIDE;} \\
&\quad \quad \text{// calculating the projection of v into Y unit vector} \\
&\quad \quad \text{float pcy = dot(v, Y);} \\
&\quad \quad \text{float h = pcz * tang;} \\
&\quad \quad \text{h = h + delta;} \\
&\quad \quad \text{// Second test: test against Y unit vector} \\
&\quad \quad \text{if (pcy > h || pcy < -h)} \\
&\quad \quad \quad \text{return OUTSIDE;} \\
&\quad \quad \text{// calculating the projection of v into X unit vector} \\
&\quad \quad \text{float pcx = dot(v, X);} \\
&\quad \quad \text{float w = h * ratio;} \\
&\quad \quad \text{w = w + delta;} \\
&\quad \quad \text{// Third test: test against X unit vector} \\
&\quad \quad \text{if (pcx > w || pcx < -w)} \\
&\quad \quad \quad \text{return OUTSIDE;} \\
&\quad \} \\
\end{align*}
\]
1.3. View Frustum Culling

Listing 1.2. Code for radar view frustum culling in GLSL.

```glsl
void main()
{
  vec4 pos = texture2DRect(position, gl_TexCoord[0].st);
  if (pointInFrustum(pos.xyz))
  {
    gl_Position = pos;
    EmitVertex();
    EndPrimitive();
  }
}
```

1.3.1 Assigning Level of Detail

After determining which positions are inside the view frustum, the next step is to assign a LOD\textsubscript{id} according to a given metric. In this case, we use discrete LOD\textsuperscript{3} which consists of creating different LODs for each character as a preprocess. At run-time, the appropriate character’s LOD is rendered using its LOD\textsubscript{id}.

Metrics for assigning values to LOD\textsubscript{id} can be based on distance to the camera, model size in screen space, eccentricity of the model with respect to the camera or perceptual factors among others. For performance and simplicity, we are using the distance to the camera as our metric; we also use visual perception to reduce the popping effect. The idea behind the distance to the camera metric is to select (or in our case, assign) the appropriate LOD based on the distance between the model and the viewpoint (i.e., coarser resolution for distant geometry). Nevertheless, instead of computing the Euclidean distance between the object and the viewpoint, we define the appropriate LOD as a function of the view range and the far plane. These values are obtained by the camera referential points.

The camera’s view range is given by unit vector \( Z \) and it is limited by the distance between the camera center, \( c \), and the Far\textsubscript{plane} value. Thus, we test the projection of \( v \) onto \( Z \), \( \text{proj}_{Zv} \), against different fixed intervals of the view range to assign a value to LOD\textsubscript{id}.

A common approach for manually assigning values to LOD\textsubscript{id} is using if statements as shown in listing 1.3. Nevertheless, we can reduce GPU branching by eliminating the if statements and by using a sum of unit step functions instead (equation 1.7):

\footnote{It has been shown in [Millán et al. 06] that 2D representations such as impostors make it possible to render tens of thousands of similar animated characters, but 2D representation approaches need manual tuning and generate huge amount of data if several animation sequences are present and/or geometrical variety is considered.}
1. A Rendering Pipeline for Real-time Crowds

Listing 1.3. Assigning \( LOD_{id} \) using if statements.

\[
LOD_{id} = n - 1 \sum_{i=0}^{n-1} U(\text{proj}\,Z_v - \text{far}_\text{distance} \times \tau_i) \quad (1.7)
\]

where \( n \) is the number of LOD meshes per character, \( \tau_i \in (0,1] \), is a threshold that isotropically or anisotropically divides the view range visually calibrated to reduce popping effects and \( U \) is the unit step function given by:

\[
U(t - t_0) = \begin{cases} 
1 & \text{if } t \geq t_0, \\
0 & \text{if } t < t_0.
\end{cases}
\]

Notice that if \( n = 3 \) (three LOD meshes per character), then \( LOD_{id} \) can receive three values, 0 when the character is near the camera (full detail), 1 when the character is at medium distances from the camera (medium detail) and 2 when character is at distances far from the camera (low detail).

Listing 1.4 shows the changes made in listing 1.2 to add \( LOD_{id} \) calculation.

[geometry shader]

```cpp
... bool pointInFrustum(vec3 point, out float lod) {
  ... // calculating the projection of \( v \) into \( Z \) unit vector
  float pcz = dot(v, Z);
  ... // For 3 LOD meshes:
  lod = step(farPlane*tao0, pcz) +
       step(farPlane*tao1, pcz) +
       step(farPlane*tao2, pcz);

  return INSIDE;
}
```
1.4 Level of Detail Sorting

The result of the VFC and LOD assignation stage is that we have filled up a vertex buffer object (VBO) for all the characters with positions inside the camera’s view frustum. And in accordance to their distances to the camera, we have assigned a LOD id for each position (figure 1.4.a.). On the other hand, hardware instancing requires a single LOD mesh to draw several instances of the same mesh, thus we need to organize these positions according to their LOD id in different VBOs.

Following [Park and Han 09], we will sort the output VBO from the VFC and LOD assignation stage into appropriate VBOs (that we will call VBOs<sub>LOD</sub>) such that all of the characters’ positions in a VBO have the same LOD id (figure 1.4.b.). Since we are using three LODs then we will use three VBOs.
1. A Rendering Pipeline for Real-time Crowds

In this case, we use transform feedback to populate each $VBO_{LOD}$. In total, we perform three transform feedback passes. In addition, transform feedback allows us to know how many primitives were written in each $VBO_{LOD}$. The number of primitives written will be used when calling the Draw Instanced routine.

For each transform feedback pass, a geometry shader will emit only the vertices of the same $LOD_i$. This is shown in listing 1.5. Notice that the uniform variable $lod$ is updated each pass. In our case it will be set to 0 for a full-resolution mesh, 1 for a medium-resolution mesh and 2 for a low-resolution mesh.

![Geometry shader](image)

Listing 1.5. Geometry shader used to populate each $VBO_{LOD}$.

Figure 1.5 shows the output of this stage and the VFC and LOD assignment stage. The characters’ positions are rendered as points. We have assigned a specific color for each $VBO_{LOD}$. In this case, red was assigned to $VBO_{LOD0}$, green to $VBO_{LOD1}$ and blue to $VBO_{LOD2}$.

1.5 Animation and Draw Instanced

As a preprocess, we load all of the character meshes and textures that will conform the crowd. We also define several character groups according to their geometrical appearance. Animation is performed with a technique that reuses the character’s rig on the other characters of the group. In this method, the animation sequence is stored in a texture array where each layer stores an animation frame. Animation frames specify the rotation angles of the character’s joints. These angles are used to pose a character and by interpolating them we perform character animation totally on the GPU. However, approaches such as AniTextures [Bahnassi 06] and Skinned Instancing [Dudash 07] can be used as an alternative.

At runtime, for each character group, we render the high-res instances first using the positions stored in $VBO_{LOD0}$ to world-transform each instance, then we render the medium-res instances using $VBO_{LOD1}$, and
1.6 Results

Finally the low-res instances using $VBO_{LOD_2}$. In each call, we use the function `glDrawElementsInstanced` available in OpenGL. This function generates a unique instance value called `gl_InstanceID` which is accessible in a vertex shader and it is used as an index to access the instance’s specific position, animation frame and its visual characteristics.

1.6 Results

We designed two tests to verify the performance of our pipeline. These tests were performed under Windows Vista using an Nvidia 9800GX2 card with SLI-disabled using a viewport size of 900x900 pixels.

The goal of the first test is to determine the execution time of the Behavior, VFC & LOD assignation and LOD sorting stages\(^4\). The goal of the second test is to determine the execution time of the complete pipeline. The first test consisted of incrementing the number of characters from $1K$ to $1M$, each character with three LODs. Timing information was obtained using timer queries (`GL_EXT_timer_query`) which provides a mechanism used to determine the amount of time (in nanoseconds) it takes to fully complete a set of OpenGL commands without stalling the rendering pipeline.

Results of this test are shown in the graph 1.6 (timing values are in

---

\(^4\)We do not provide the execution time of the Animation & Draw Instanced stage since timing results are bigger by several orders of magnitude.
1. A Rendering Pipeline for Real-time Crowds

Figure 1.6. Test 1 results. Notice that timing results are in milliseconds.

milliseconds). In addition, figure 1.5 shows a rendering snapshot for 4096 characters rendered as points. Notice that the elapsed time for VFC & LOD assignment and LOD sorting stages remains almost constant. Besides, when performing transform feedback, we do not need any other subsequent pipeline stages thus rasterization is disabled.

The second test consists of rendering a crowd of different characters. Each character has three LODs, the character’s LOD-0 mesh is made of 2500 vertices, the LOD-1 mesh 1000 and LOD-2 300. The goal of this test is to determine the execution time of all the stages of our pipeline using two different camera perspectives. In perspective A (fig. 1.7), almost all characters are visible, while in perspective B (fig. 1.8) almost all characters are culled.

These results are shown in table 1.1 for perspective A, and in table 1.2 for perspective B. The first column of both tables shows the number of rendered characters, the second one shows the time, in milliseconds, measured for each case. Columns three to five show how many characters per level of detail are rendered, and column six shows the total number of vertices, in millions, transformed by our animation shader. Finally, the last two columns show the percentage of characters that are visible or culled.
1.7 Conclusions and Future Work

We have shown that optimization techniques such as view frustum culling and level of detail selection in the GPU result in a very small time penalty. In our practical case, the stage that takes more time to execute was Animation & Draw Instanced, which is expected. Moreover, extra memory...
1. A Rendering Pipeline for Real-time Crowds

Table 1.1. Results obtained in perspective A

<table>
<thead>
<tr>
<th>Agents</th>
<th>t ×10^{-3}</th>
<th>LOD0</th>
<th>LOD1</th>
<th>LOD2</th>
<th>Vertices ×10^6</th>
<th>Visible %</th>
<th>Culled %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>47.62</td>
<td>274</td>
<td>644</td>
<td>54</td>
<td>1.35</td>
<td>95</td>
<td>5</td>
</tr>
<tr>
<td>2048</td>
<td>62.11</td>
<td>365</td>
<td>688</td>
<td>397</td>
<td>1.70</td>
<td>71</td>
<td>29</td>
</tr>
<tr>
<td>4096</td>
<td>95.60</td>
<td>450</td>
<td>1388</td>
<td>943</td>
<td>2.80</td>
<td>68</td>
<td>32</td>
</tr>
<tr>
<td>8192</td>
<td>159.24</td>
<td>483</td>
<td>1438</td>
<td>5085</td>
<td>4.17</td>
<td>86</td>
<td>14</td>
</tr>
<tr>
<td>12288</td>
<td>194.93</td>
<td>400</td>
<td>1905</td>
<td>7694</td>
<td>5.21</td>
<td>81</td>
<td>19</td>
</tr>
<tr>
<td>16384</td>
<td>261.78</td>
<td>476</td>
<td>2477</td>
<td>10388</td>
<td>6.78</td>
<td>81</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 1.2. Results obtained in perspective B

<table>
<thead>
<tr>
<th>Agents</th>
<th>t ×10^{-3}</th>
<th>LOD0</th>
<th>LOD1</th>
<th>LOD2</th>
<th>Vertices ×10^6</th>
<th>Visible %</th>
<th>Culled %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>19.49</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>0.40</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td>2048</td>
<td>22.91</td>
<td>204</td>
<td>0</td>
<td>0</td>
<td>0.51</td>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td>4096</td>
<td>49.33</td>
<td>412</td>
<td>314</td>
<td>0</td>
<td>1.34</td>
<td>18</td>
<td>82</td>
</tr>
<tr>
<td>8192</td>
<td>57.77</td>
<td>503</td>
<td>317</td>
<td>0</td>
<td>1.60</td>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td>12288</td>
<td>72.99</td>
<td>471</td>
<td>995</td>
<td>0</td>
<td>2.20</td>
<td>12</td>
<td>88</td>
</tr>
<tr>
<td>16384</td>
<td>97.18</td>
<td>541</td>
<td>1546</td>
<td>0</td>
<td>2.90</td>
<td>13</td>
<td>87</td>
</tr>
</tbody>
</table>

requirements do not exceed the amount needed to store a 32-bit floating texture of 512 × 512 pixels, i.e. for sixteen thousand characters we needed to allocate four floating point vertex buffer objects of 128 × 128, one auxiliary vertex buffer object to store partial results obtained from the VFC & LOD assignation stage, and three vertex buffers to store the positions of each level of detail.

However, performance results can be improved and memory requirements can be reduced by using the new OpenGL 4.0 characteristic called “multiple transform feedback”, exposed in the ARB_transform_feedback3, ARB_gpu_shader5 and NV_gpu_program5 extensions, which allows geometry shaders to direct each vertex arbitrarily to a specified vertex stream. Therefore, we will only require one transform feedback call for LOD id sorting and by combining the VFC & LOD assignation and LOD sorting stages we could dispense the auxiliary vertex buffer object used to store partial results obtained from the VFC & LOD assignation stage.

This pipeline can be extended by adding an occlusion culling stage. Complex scenes such as those where crowds are needed or those where landscapes are depicted with indigenous vegetation, human elements, buildings and structures can be favored by this. One approach is to perform occlusion culling via OpenGL occlusion queries. Nevertheless, the number of
1.8. Acknowledgements

queries needed might not be enough for complex scenes made of hundreds of thousands of elements. In addition, it requires synchronizing the CPU and the GPU, which might stall the pipeline. Another approach is to put extra cameras in the positions of big structures and perform radar view frustum culling (using the normalized version of the view frustum) and then take the complement of the visible set. This stage can be performed after the behavior stage and before the VFC & LOD assignation stage.

1.8 Acknowledgements

We wish to thank Nvidia for its kind donation of the GPU used in the experiments.

Bibliography


