Efficient Shadow Generation of Massive Crowds Based on the Area of Screen Space

Mankyu Sung

Abstract
Shadows are critical for realistic and visually appealing images. In particular, for interactive applications such as video games, the shadows of objects significantly increase the realism of the virtual environment. However, for scenes in which a large number of characters are animated, it is not easy to generate shadows in real time because the shadow of each character must be rendered individually. This study introduces a simple yet efficient technique for generating shadows for large crowds. This technique considers the space occupied by each character in the final image. If the space is significantly small, the algorithm applies a technique that uses a simple and fast yet physically correct projection shadow matrix to generate shadows. If the space is relatively large, which means that the character is close to the camera, the algorithm applies a widely known shadow mapping technique. This technique is slower than the projection shadow matrix but is more physically practical. The rendering of large crowds is achieved using the GPU instancing technique. Because the algorithm adaptively chooses shadow generation techniques according to their visual saliency, the overall performance improves. Through several experiments, we demonstrated that our algorithm can generate shadows for more than 20,000 characters, all of which have a full-skinned mesh, at a real-time rate while maintaining the visually plausible shadow quality, with the rate being approximately 54% faster than the traditional shadow mapping method.

Keywords
Human-Computer Interaction and Multimodal, Computer Animation, Crowd Simulation, Shadow Generation, Shadow Mapping

1. Introduction
Shadows are important for creating synthetic scenes because they provide visual cues to clarify the geometrical relationship between light sources and objects [1]. For applications such as video games, shadows provide a more realistic atmosphere to the virtual environment because they establish spatial relationships between the objects in a scene. This increases the immersive feeling of the game for users. In addition, it can provide an important measure for the environmental quality of public space, such as buildings in real life [2].
Several techniques have been proposed to generate shadows of virtual objects. An excellent survey can be found in a previous study [1, 3–5]. Traditional global illumination algorithms such as ray tracing can solve the shadow generation problem implicitly [6, 7]. In particular, a previous study [7] proposed a baked technique that precomputes the global illumination data including shadow to reduce the computational overhead. By contrast, the rasterization pipeline requires a special technique to generate shadows because it is difficult to distinguish whether a particular pixel belongs to a shadow. Many of such algorithms are based on the so-called shadow map [8], which contains the closest distance from the light source to the surface of the objects in the scene. Given this map, a simple distance comparison was performed after all pixels were transformed into the light source space. However, the quality of shadows generated by the shadow map depends heavily on itself because it is only an image with finite resolution. Therefore, several nagging aliasing artifact problems still exist. Many additional works or variations have been proposed to reduce aliasing [9–11]. In this study, we address a target scene in which many three-dimensional (3D) characters are animated and rendered. For example, a big battle scene in which two camps are fighting with each other or a sports scene in which crowds are entering or exiting a stadium would require numerous 3D characters [12, 13]. Numerous characters imply more than 10,000 characters, and each character has a fully skinned mesh that animates over time. Conventional shadow mapping techniques are not adequate for this scene because a considerable amount of computational overhead is required to animate and render each character. In particular, when we assume that each character is fully skinned, animating and rendering takes time because all the vertices of the mesh must be updated during every time frame.

Among the many variations of the conventional shadow mapping technique, cascaded shadow mapping (CSM) was proposed by NVIDIA [14]. In this technique, rather than using a single shadow map, several shadow maps are used depending on the distance from the camera. In our approach, we modified the CSM technique so that it could be optimized for crowd scenes. The logic behind selecting the CSM method over other shadow generation techniques are as follows: for a crowd scene, the camera is sometimes located relatively far away from the crowds so that it can capture all of them. This type of camera setup makes some characters appear extremely small on the screen. For these small characters, highly detailed shadows are not required because shadows take only a few pixels in the final image. This approach is similar to the original CSM method. In the CSM method, to overcome the aliasing problem caused by the fixed resolution of the shadow map, multiple shadow maps are used instead. When the characters are closer to the camera, a highly detailed shadow map with a small field-of-view is used. When the characters are far from the camera, a shadow map with a wide field-of-view is used, which is a rough map. However, because multiple shadow maps must be created, the overall performance is not efficient. Our basic idea is to combine the existing CSM technique with a simple projected shadow matrix technique according to character’s importance. Although the projected shadow matrix can quickly generate a shadow of the character, it only creates a hard shadow. This drawback was mitigated by filtering. These shadows are sufficient for characters located far from the camera. In our interpretation, the important value is the extent to which the shadow takes space in the final rendered image. To approximate the shadow size of each character, we placed a bounding sphere on each character and projected the sphere on the final image. Thereafter, the area of the projected ellipses was calculated. The area was then adjusted according to the angle of the light source. All computations and rendering were performed on the GPU side for optimization.

Through several experiments, we verified that the proposed shadow generation algorithm can be at least 50% faster than the conventional method, and this improvement increases with an increase in the number of characters. For a maximum of 20,000 characters, we can render all crowds and their shadows in real time. The main contributions of the present study are as follows:

- New rendering scheme: we introduced a new rendering scheme for a large number of 3D characters based on their importance, which is the space that each character takes in the image plane.
- Hybrid shadow generation technique for large crowds: we used two methods and selected one of them automatically for each character. In this process, the parameters for the CSM method were set automatically according to the scene.
The remainder of this paper is organized as follows. Section 2 lists up the related work and compares the existing methods with the proposed method. Section 3 illustrates the proposed algorithms in detail. Section 4 shows the experiments and performance results. Finally, Section 5 concludes the paper with a summary and discussion.

2. Related Work

Shadow generation has been a widely researched topic in the computer graphics community for several decades [4]. Not only does the shadow upgrade the visual quality of presence of virtual objects, but also becomes an important measure for real life problem such as a right of light. For example, some researchers use the virtual shadow of 3D building built with computer graphics to estimate and visualize how much light enters into a specific region of building [2].

We can classify shadows into two types: hard and soft. The hard shadow has a sharp border outline and includes fully shadowed regions without any soft edges [3]. In contrast, the soft shadow has smooth edges across the shadow. Soft shadows are preferred because they are generally similar to real shadows in real life [5, 7]. To generate soft shadows from a light source, the shadow mapping technique has been used traditionally [4, 8, 15]. This algorithm operates in two stages. The entire scene must be rendered from the light position during the first stage and then rendered again from the viewpoint of the camera. If a pixel is observed in the second rendering phase, but not from the first rendering, it is located in the shadows. A simple comparison between the value of the shadow map and the depth of the pixel determines whether a particular pixel is inside the shadow or not. Although shadow mapping is powerful and convenient, it has an intrinsic limitation in that the result from the first stage must be stored in a texture with a finite resolution, which causes serious aliasing problems if the resolution is not significantly large [8]. To overcome the aliasing, if we increase the resolution of a shadow map substantially, it causes a significant memory use. Some researchers have proposed a compression technique of the shadow map to reduce the memory use [16] instead. However, it requires an extra computation time to compress the shadow maps.

Filtering- or image-based methods have been used to overcome this limitation practically. The filtering method, which is called percentage closer filtering (PCF) [9], undergoes a series of comparisons per pixel and is averaged together. The filtering-based method can generate smooth shadows; however, it is not efficient because multiple comparisons must be performed for each pixel, which incurs computational overhead. Instead, Macedo et al proposed a vectorizing technique that applies a set of PCF and smoothing technique to convert the jagged shadow map into a smoothed edge [17], but still requires an additional computation time for each pixel, which is not easy to achieve the real time performance especially when we need to render a large number of 3D objects.

Some of the popular image-based methods include variance soft shadow mapping (VSSM) or CSM. The VSSM method is based on the statistical moment value of the shadow map and applies the Chebyshev inequality to approximate the fraction value, which determines whether a pixel becomes a shadow [10]. Going one step further from VSSM, the moment shadow mapping method is turned out to create high-quality shadows more efficiently using higher-order statistical values of the shadow map [18]. Those variant methods can generate shadows with minimizing computation overhead but still have a same limitation for large number of moving objects.

As another approach, CSM, is useful for wide scenes, such as large terrains and forests [14]. Because the crowd scene also needs a wide scene in general, we chose the CSM for our basic algorithm but modified it to support a large crowd. Basically, this technique subdivides the user’s view frustum into several partitions and obtains different shadow maps for each partition. There are several methods for splitting view frustum [19, 20]. In the CSM approach, the sampling density decreases for each successive partition because the same number of shadow map samples covers a larger area [14, 20]. Among multiple shadow maps, the algorithm automatically chooses one pixel based on its distance from the camera.
Efficient Shadow Generation of Massive Crowds Based on the Area of Screen Space

Please refer to the paper in [14] and [19] for more details. In recent years, inspired by large emergence of machine learning techniques, many approaches have been proposed for solving computer graphics problem with deep learning method. Liu et al proposed a shadow-specific generative adversarial network (GAN) model for a single light source [21]. A similar technique was also proposed by [22]. Although these techniques are able to synthesize artificial shadow, they require large amount of training data, which is not easy for practical purpose.

One of the important problems of all previous work is that they have not considered the large number of moving objects and camera setup. In crowd simulation, large number of 3D characters are moving simultaneously, and each character is fully meshed, which requires additional computational power. Therefore, shadow generation must be light enough so that it does not affect the overall performance while maintaining the good visual quality. We address the problem by mixing up two methods, which are the CSM and shadow matrix method, in a seamless manner. One important aspect that must be considered while creating shadows of crowds is that each character may be considerably small on the final rendered image in general camera setup [12]. Therefore, a computationally expensive method is not required to create a shadow for such a small character. We divide the entire crowd in the final image into two classes, and thereafter apply different shadow generation methods for each class. The classification is based on the area in which each shadow occurs in the image plane. This approach is similar to the foveated rendering technique in virtual reality in some sense [23, 24]. Foveated rendering uses an eye tracker to reduce the rendering workload [23]. The eye tracker keeps tracking the user’s gaze, and its position provides a clue to render multiple rendering layers progressively from high resolution to low resolution. Such an attention model provides an important clue for crowd rendering [25]. In our approach, although we used multiple rendered layers for different sizes of characters to combine the final image, each layer had exactly the same resolution.

3. Proposed Algorithm

3.1 Overview

This subsection provides an overview of the entire algorithm. Fig. 1 shows an overview of the shadow generation process of crowds. The entire process is divided into two parts: classification and shadow generation. The classification step classifies the entire crowd into two groups \( L_i \) and \( S_i \) depending on their areas on the image plane. For this, we used a predefined threshold parameter \( t \) to determine the classes. The crowds in \( L_i \) are located far from the camera. Therefore, their sizes in the image plane are small, whereas the crowds in \( S_i \) are closer to the camera. After determining the two classes, the characters in each class were fed into different shadow generation algorithms. Because the characters in \( L_i \) are small, their shadows are small as well. We constructed a simple projection shadow matrix from the light sources and multiplied it with the current model matrix of the character. The shadow projection matrix projects the mesh of the character onto a flat ground. The result, which represents shadows, is stored as a texture. Subsequently, the normal characters were rendered as another texture. Because the characters in \( S_i \) are close to the camera, the shadows must be rendered carefully. We applied the CSM method to these characteristics. To apply this technique, a layered depth map was constructed from the cascaded view frustum. This technique overcomes aliasing artifacts in a wide screen scene by using several shadow maps. These shadow maps were then used to determine whether a particular pixel was in the shadow.

The rendered result was stored as another texture. To minimize the computation required to generate the third texture, the first and second textures were used to mask unnecessary fragment processing. The three textures obtained from the steps described above were then combined and blended together in the final stage. All computations and rendering were performed on the GPU side for the optimization.
Fig. 1. The classification step puts a bounding sphere for each character and then project it onto the image plane. The area of a projected ellipse decides one of the two shadow generation algorithms.

The shadow generation step creates three textures and blend them together at the final stage.

3.2 Classification

Given the initial model matrices for all characters that set their position and orientation, the classification step places a bounding sphere on each character. By assuming that we compute the area of the projected sphere in the camera space, a point \( p \) on a sphere can be described in a parametric form as follows:

\[
|p - o| = r^2
\]

(1)

where \( o \) is the center of the sphere and \( r \) is the radius of the sphere. Radius \( r \) must be sufficiently large to include the four limbs of the character in the sphere. If we cast a ray from the camera to the image plane in a pinhole camera model, in which the image plane is located between the camera and the scene, the ray goes to point \( p \) on the sphere as it intersects the image plane at \((x, y, 1)\) if we assume that the focal length is equal to 1,

\[
p = td, \quad d = (x, y, 1)/\sqrt{x^2 + y^2 + 1}
\]

(2)

where \( d \) is the normalized ray direction vector, \(-1 \leq x, y \leq 1\) and \(|d| = 1\); \( t \) is the distance from the intersection point to the camera. Substituting Equation (2) into Equation (1) and rearranging it with respect to point \( p \), the following equation can be obtained:

\[
|(td) - o|^2 - r^2 = 0
\]

(3)

After arranging Equation (3) with respect to \( t \), the quadratic equation \( At^2 + Bt + C = 0 \) can be obtained. The coefficients \( A, B, \) and \( C \) of the equation are described as follows, where \((\cdot)\) represents a dot product.
The only condition that this quadratic function has solutions is when its discriminant is greater than zero. From this fact, if we expand the discriminant, which has a $B^2 - 4AC$ form, and arrange it, we obtain the following inequality:

$$(\mathbf{o} \cdot \mathbf{d})^2 - (|\mathbf{o}|^2 - r^2) \geq 0 \quad (5)$$

By substituting $\mathbf{d}$ defined in Equation (2) and eliminating positive constants in Equation (5), we would obtain the following equation:

$$(o_x x + o_y y + o_z z)^2 - (x^2 + y^2 + 1)(|\mathbf{o}|^2 - r^2) \geq 0 \quad (6)$$

It turns out that the Equation (6) is the general implicit form of an ellipse. If we organize all terms of the equation, we obtain Equation (7).

$$ax^2 + by^2 + cxy + dx + ey + f \geq 0,$$

$$a = r^2 - o_y^2 - o_z^2$$

$$b = r^2 - o_x^2 - o_z^2$$

$$c = 2o_x o_y$$

$$d = 2o_x o_z$$

$$e = 2o_y o_z$$

$$f = (r^2 - o_x^2 - o_y^2) \quad (7)$$

In the general implicit representation of an ellipse, the major and minor axes, denoted as $\text{min}$ and $\text{max}$, are the longest and shortest diameters of the ellipse, respectively. Mathematically, the general conic section of an ellipse is given as follows [26]:

$$\text{max} = \frac{2\left(\frac{ae^2 - cde + bd^2}{4ad - c^2} - f\right)}{\sqrt{(a + b - \sqrt{(a - b)^2 + c^2})}}, \quad \text{min} = \frac{2\left(\frac{ae^2 - cde + bd^2}{4ab - c^2} - f\right)}{\sqrt{(a + b + \sqrt{(a - b)^2 + c^2})}} \quad (8)$$

If we substitute all coefficients from $a$ to $f$ in Equation (8) and simplify them, we obtain the final major and minor axes as follows:

$$\text{Min} = \frac{-r^2(r^2 - |\mathbf{o}|^2)}{|\mathbf{o}|^2 - o_x^2)(r^2 - o_y^2) - (r^2 - |\mathbf{o}|^2)} \quad , \quad \text{Max} = \frac{-r^2(r^2 - |\mathbf{o}|^2)}{|\mathbf{o}|^2 - o_x^2)(r^2 - o_y^2) - (r^2 - o_z^2)} \quad (9)$$

Once we know $\text{Min}$ and $\text{Max}$, projected area $A$ can be simply calculated as follows:

$$A = \pi \text{Min} \cdot \text{Max} \quad (10)$$

Because our ultimate goal is to approximate the shadow size of the character in the image plane, rather than the character size, we adjusted the areas of the projected bounding spheres by the current light direction with the plane. If we assume that the plane is flat, the modified area can be calculated as follows:

$$A = A \cdot \tan(l \cdot n) \quad (11)$$
where \( l \) is the unit light direction vector and \( n \) is the normal vector of the plane.

The modified area becomes larger or smaller depending on the direction of light. For example, if we compare the size of the shadow while the light source rotates around, the shadow is smaller when the light source is right above the character than when the light source is at the front or back. The area is modified considering the changes in the shadow size. If the area \( A \) of a character is smaller than the threshold value \( T \), then we put the character in list \( L_i \) otherwise, we put it in \( S_i \). To improve the performance, all calculations of \( A \) were performed on a GPU using general-purpose computing on GPU (GPGPU) technique. The threshold value \( T \) is set empirically. If the camera is looking at the crowd in a zoomed manner, then the value must be small so that high-quality shadows are generated. Otherwise, it can be set high so that the overall performance improves. More implementation details can be found in Section 4.

3.3 Texture Synthesis

After classification is completed, we have two lists of characters. In addition to their ID numbers, we stored the current model matrices in separate buffers. These two buffers were sent to the GPU for easy access. The basic idea of the shadow generation step is to create three textures and blend them together. Let us explain the first two textures. The first texture is the shadow texture representing the shadows of the characters in \( L_i \). This shadow texture is created by multiplying a projective shadow matrix with its current model matrix. Projective shadow matrix \( S_m \) can be defined from the position and direction of the light. Let us say that we have a light source at location \( l \). If we suppose that \( d \) is the height of the shadow projected on the plane and the light is lit to the origin (0,0,0), then the \( S_m \) can be defined as follows:

\[
S_m = \begin{bmatrix}
1 & -l_x/l_y & 0 & 0 \\
0 & 0 & 0 & d \\
0 & -l_x/l_y & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Matrix \( S_m \) projects the whole character mesh onto the plane, which represents the shadow of a character. This matrix dynamically changes according to the position of the current light. After applying the matrix, the rendering result is written as a texture using a user-created frame buffer. This texture is then subjected to Gaussian blurring filtering to emulate the smooth edges. The kernel size controls the blurring effect. We used a small 3×3 kernel considering the extra time for filtering.

The second texture is obtained by rendering a normal character. In this case, we apply the model matrix without a projective shadow matrix. This image shows the rendered characters located far from the camera.

![Top down view of View Frustum](image)

**Fig 2.** Splitting of view frustum for cascaded shadow mapping.

The third texture was obtained by applying the CSM technique to the characters in \( S_i \). There are several variations in this technique in terms of the methods by which we partition the view frustum [19, 20]. Fig.
Efficient Shadow Generation of Massive Crowds Based on the Area of Screen Space

2 depicts the case in which the view frustum is split into three parts based on the distance from the camera. It should be noted that the camera is located at the bottom. Three cascaded shadow maps were generated from these three segments [14]. A key insight is that the shadow map located close to the camera can have high details, whereas the map distant to the camera lacks this feature. The reason is that the close one has a narrow field-of-view, whereas the distant one has a wide angle. Because the two shadow maps have the same resolutions, the area covered by single pixel expands as the splitting goes deeper. Fig. 2 shows three shadow maps obtained by partitioning. The grid size of the map represents the level of details. A shadow map with a higher resolution results in more anti-aliased shadows. The splitting scheme that our approach uses is the practical split scheme proposed by Zhang et al. [19, 20]. In this scheme, the \( i \)th split distance \( C_i \) is determined using the following equation:

\[
C_i = \lambda C_i^{\text{log}} + (1 - \lambda) C_i^{\text{uni}} \quad 0 < \lambda < 1
\]

\[
\begin{align*}
C_i^{\text{log}} &= n(f/n)^{i/m} \\
C_i^{\text{uni}} &= n + (f - n)i/m
\end{align*}
\]

where \( f \) is the far plane, \( n \) is the near plane, and \( m \) is the number of splits. The logarithmic split scheme (\( C_i^{\text{log}} \)) and uniform split scheme (\( C_i^{\text{uni}} \)) manage perspective aliasing and projection aliasing, respectively. Note that \( f \) and \( n \) must be set as tightly as possible to observe the entire crowd. Otherwise, the quality of shadow maps will be poor. In our approach, we set these values adaptively by calculating the character’s maximum and minimum distances from the camera. If we assume \( p_i \) is assumed to be the character position in the camera coordinate system, then, the near plane and plane can be set as follows, where \( k \) is the number of characters:

\[
\begin{align*}
f &= \max_{i \in k} |p_i| \\
n &= \min_{i \in k} |p_i|
\end{align*}
\]

After the splitting is completed, the algorithm must find a bounding box for each segment from the perspective of light. This makes sense because the general shadow map, which contains the distance value, was created using orthographic projection, which has a box-shaped view frustum. Therefore, finding a bounding box would become the finding of a view frustum for the cascade. There are several methods for determining the bounding boxes. Among them, we used the simplest method, in which eight points forming the segments were found using a simple brute-force search. Fig. 3 depicts the bounding box for the middle segment. The bounding box is represented by a blue box. It should be noted that this was a top-down view. There are four other points underneath.

![Fig 3. Finding a bounding box for the cascade in the middle.](image-url)
The basic idea of finding a bounding box consists of two steps. The first step is to find the original eight points \((x_1-x_8)\) of the cascade from the camera perspective, which are represented by yellow dots in Fig. 3. The second step is to transform these eight points in the light space and find the minimum or maximum points to find the eight corner points \((x_1'-x_8')\) forming the bounding box, which are represented using red dots. The original eight points can be calculated for a given distance \(C_i\) from Equation (13), as follows:

\[
\begin{align*}
x_n &= C_i \tan(\theta/2) \\
x_f &= C_{i+1} \tan(\theta/2)
\end{align*}
\]  

where \(\theta\) is the field-of-view of the camera.

These eight points are then transformed to world coordinates by \(M^{-1}V\) where \(M\) is the model matrix and \(V\) is the view matrix. Once we have eight corner points, we translate and rotate them into the light coordinate system. Given the condition that the light source is looking at the \((0,0,0)\) point in the world, the new three axes of the light coordinate system are as follows:

\[
\begin{align*}
z &= \vec{l}, \\
x &= u \times z, \\
y &= z \times x
\end{align*}
\]

where \(u\) is the up vector and \(\vec{l}\) is the normalized light vector

\[
\begin{align*}
V^l &= RT = \begin{bmatrix}
x_x & x_y & x_z & 0 & 1 & 0 & 0 & -l_x \\
y_x & y_y & y_z & 0 & 0 & 1 & 0 & -l_y \\
z_x & z_y & z_z & 0 & 0 & 0 & 1 & -l_z \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix} 
\end{align*}
\]

The \(V^l\) was multiplied by the eight points to transform into the light coordinate system.

\[
\hat{x}_i = V^l x_i, \quad 1 \leq i \leq 8
\]

Once we find the \(\hat{x}_i\), a simple minimum or maximum comparison over the eight points can find the new eight corner points of the frustum \((x_1'-x_8')\). For each view frustum in the cascade, we rendered the crowds in \(S_i\) to store their depth values. These depth maps were then passed to the fragment processor to determine whether a pixel belonged to a shadow.

Algorithm 1: Masked texture synthesis

| Input: uv=a fragment in \(M \times N\) frame buffer |
| \(M_s=\)CSM |
| \(T_1=\)first texture |
| \(T_2=\)second texture |

| Output: \(w_s=\)shadow coefficient |

1. \textbf{for} \(\forall uv \in\) frame buffer \textbf{do}
2. \textbf{if} \(T_1 \leq 0 \lor T_2 \leq 0\) \textbf{then}
3. \(w_s = \text{SHADOW}(uv,M_s)\)
4. \textbf{else}
5. \(w_s = 0\)
6. \textbf{end if}
7. \textbf{end for}
One important improvement of our approach over the traditional CSM method is that fragment processing does not need to be performed for all pixels. Algorithm 1 explains these steps. As inputs, three textures are entered. The first texture is obtained by the projective shadow matrix \((T_1)\) and the second texture is obtained by rendering normal crowds \((T_2)\). Then, those two textures provide a mask for the third rendering. That is, because we subdivide the entire scene into three regions, they do not overlap each other. We can take advantage of this, so when fragments belong to the first or second texture, which corresponds to the if statement in the line 2 of the algorithm, then the algorithm skips the fragments, which can improve performance by minimizing the number of fragments to process.

In this step, the function \text{SHADOW}() performs a comparison between the fragment’s depth and the depth value of CSMs. More details on this comparison processing can be found in the original paper [14]. As a result of the function call of \text{SHADOW}(), it returns a shadow coefficient value, \(w_s\), which controls the final color \(c\) from the diffuse, specular color, and ambient color. This can be summarized as follows:

\[
c = C_a + (1 - w_{shadow})C_{ds}
\]

where \(C_a\) is the ambient color, and \(C_{ds}\) is the summation of the diffuse and specular colors. The rendering result was also saved into another texture, \(T_3\) for the final blending step.

### 3.4 GPU Instancing

To render crowds, various styles of meshes must be prepared for the characters. Therefore, when we render a large crowd, we have to re-use the meshes. Current 3D graphics APIs, such as Direct3D, OpenGL, or Vulkan, provide an in-built function for the efficient rendering of the same geometry repeatedly. This technique is known as hardware installation (geometry instancing) [27]. For instance, GPU instancing is applicable for rendering trees and grasses in which the same geometry is distributed randomly in a space. The motivation for GPU instancing is a bottleneck in data transfer between the CPU and GPU. Because the data transfer between the CPU and GPU occurs whenever a draw function is called, the number of draw calls must be minimized to achieve better performance. Because the CPU has the limitation of sending data to the GPU through the internal bus per second, it can draw only a fixed number of polygons per frame. To overcome this drawback, modern GPUs divide the geometry data into two parts: geometry packet and instance attributes, where the geometry packet is a description of the geometry to be installed, such as vertices and indices, and the instance attribute is instance-specific data such as position or orientation. The GPU combines these two data points inside the device and reuses them during each instance. Therefore, once they are fetched from the main memory, they are kept inside the device. There are several methods for specifying instance attributes. In our approach, we designed a simple array structure to contain each character’s position and orientation. The model matrices stored in \(L_i\) or \(S_i\) were used to place each character in a different position. These matrices must be updated to allow the character to move at every time frame.

### 3.5 Final Rendering

Texture blending was performed to create the final rendered image given the three textures obtained by texture synthesis processing. In this process, we used the alpha blending technique, in which the alpha value of the fragment determines the order of merging. The order is important for the first and second textures because they were created individually. We compared the two cases when the blending order was correct and when it was incorrect, as shown in Fig. 4. Clearly, we know that the shadows came up front of the character on the picture of the right, which is not correct.

Let us suppose that we have three colors, denoted from \(c_1\) to \(c_3\) from textures \(T_1\) to \(T_3\). \(T_1\) is the texture obtained by applying the projective shadow matrix, and \(T_2\) is the texture from the normal rendering of characters. Both \(T_1\) and \(T_2\) are for the crowds in the list \(L_i\), whereas \(T_3\) is the rendered result using CSM
for crowds in $S_i$. In the color buffer, we put $c_3$ first. Then, $c_1$ is set based on its alpha value and $c_2$ according to its alpha value. This formula can be expressed as Algorithm 2.

**Algorithm 2: Texture Blending**

**Input:**
- $T_1$: first texture (Shadow texture)
- $T_2$: second texture (normal characters in $L_i$)
- $T_3$: third texture (CSM texture in $S_i$)

**Output:** $c$: final color

1. for $\forall \ uv \in$ frame buffer do
2. $c_1 = T_1(uv)\ c_2 = T_2(uv)\ c_3 = T_3(uv)$
3. $c = c_3$
4. $c = \text{MIX}(c_2, \text{MIX}(c_1, c, \text{STEP}(c_1, a, 0)), \text{STEP}(c_2, a, 0))$
5. end for

In this formula, the function \text{MIX}(x, y, r) performs linear interpolation between $x$ and $y$ with weight $r$, and the function \text{STEP}(x, y) returns one when $x$ is less than $y$; otherwise, it returns to zero.

![Fig 4. (a) Correct texture blending order and (b) incorrect texture blending order.](image)

4. Experiments

We built a software system to verify the proposed algorithm. The hardware setup included a PC with an Intel i7 CPU, 32 GB main memory, and an NVIDIA GeForce RTX 2080 graphics card. We used the GLSL (OpenGL Shading Language) version 4.6 for implementing the GPU shaders and C++ for CPU coding. For character animation, we used two different speed running animations coded in Autodesk’s FBX format. The classification step was implemented using a compute shader. The compute shader invokes as many threads as the number of characters. Then, each thread computes the area of the projected ellipse of a character separately. Finally, a designated thread calculates the number of characters in $L_i$ and $S_i$. The shear storage buffer object structure was used to store the model matrices of each character [28]. This array structure can be accessed in the GPU and copied to the CPU side as well.

The first experiment was performed to check the visual quality of the proposed method against the pure CSM-only method. Fig. 5 depicts two cases of 1000 crowds. For a better understanding, please refer to the accompanying video. As we can see in the picture, when the characters are located far away from the camera, their space on the image plane is quite small. Therefore, even though we apply the simple projection matrix technique, the synthesized shadows do not look bad, which is almost indistinguishable from the CSM shadow perceptually.

The second experiment was conducted to check the performance between the proposed method and the CSM-only method as we increased the number of characters. For this experiment, we increased the number of characters from to 1,000–20,000. The camera was placed at a fixed position so that it could see the crowds from close to distant look simultaneously. The light was rotated around a circle above the plane to test the dynamic shadow. Figure 6 depicts a graph comparing the proposed method with CSM-
only and moment shadow mapping (MSM) methods [18]. The resolution of shadow maps is equal for different methods (1024×1024). We found that the proposed method was 54.8% faster on average than the CSM-only method, although the speed increased as the number of characters increased. Fig. 6 also lists the numerical data for the second experiment. All numerical measurements are the average values after 10 repetitions.

![Figure 5](image1.png)  
**Fig 5.** (a) Proposed method and (b) CSM-only method.

![Figure 6](image2.png)  
**Fig 6.** Frame rate comparison between the proposed method against the CSM-only and MSM method.

![Figure 7](image3.png)  
**Fig 7.** (a) 10,000 crowd movement in a circle and (b) zoomed view.
For all experiments, the threshold value $t$ of the screen area for classifying the crowds was set to 300 units, and the screen resolution was 800×800.

Fig. 7 shows a screenshot of the animation of a crowd of 10,000 people with shadows. The light was also set to rotate around a circle above the plane. The crowds moved around a circle so that they did not get out of the camera view. As we can see in the picture, all the shadows of characters are plausible. Please refer to the accompanying video clips.

5. Discussion and Conclusion

In this study, we propose an efficient shadow generation algorithm for large crowds. The basic idea of the proposed method is to divide the whole crowd into two groups and apply different methods for each group based on the space occupied by them on the final rendered image. Because we did not use any complicated methods for barely visible distant characters, the overall performance increased without degrading visual quality, even though we used the simple projective matrix method. Nevertheless, there are a few limitations to the proposed method. First, the simple projective matrix method does not consider the inter-shadow between characters. That is, when two characters are sufficiently apart, their shadows may shade each other. Because our shadow textures are generated without considering other nearby characters, it is not possible to simulate this effect. However, because these characters were positioned very far away from the camera, we believe that the inter-character shadows are not visually important.

Second, the proposed technique assumes that the plane on which the crowds are moving is flat. This is the same reason as in the first limitation. Although the shadow mapping technique can handle non-flat ground, the projective shadow matrix technique, on the other hand, is difficult to deform the shadow mesh for non-flat ground. In the future, we intend to develop a method for non-flat planes. However, we must consider the trade-off between performance improvements over the computation cost in this process, especially for very small characters on the back side.

The overall performance depends on the classification parameters. Parameter $t$ controls the ratio of the number of characters between the two groups. As the value increased, the overall performance also increased. In future work, we can set the threshold automatically according to scene analysis. Another possible extension is to apply the proposed technique to a multicore CPU device. Because all experiments were performed using OpenGL, the multicore CPU was not supported. Current 3D graphics APIs, such as Vulkan, support multicore CPUs. Therefore, we would like to test our algorithm on the device and determine the extent to which the performance can be improved.

Funding

This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2018R1D1A1B07048414 and 2021R1A2C1012316).

Competing Interests

The author declares that he has no competing interests.

References


