VISIBILITY GRID METHOD FOR EFFICIENT CROWD RENDERING WITH SHADOWS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS INSTITUTE
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ŞAHİN SERDAR KOÇDEMİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF MODELING AND SIMULATION

OCTOBER 2012

VISIBILITY GRID METHOD FOR EFFICIENT
CROWD RENDERING WITH SHADOWS

Submitted by **Şahin Serdar KOÇDEMİR** in partial fulfillment of the requirements for the degree of **Master of Science in the Department of Game Technologies**, **Middle East Technical University by**,

Prof. Dr. Nazife Baykal _____
Director, Informatics Institute

Assist. Prof. Dr. Hüseyin Hacıhabiboglu _____
Head of Department, MODSIM, METU

Assoc. Prof. Dr. Veysi İşler _____
Supervisor, Compter Engineering, METU

Examining Committee Members

Prof. Dr. Bülent Özgüç _____
ARCH, BİLKENT

Assoc. Prof. Dr. Veysi İşler _____
CENG, METU

Assist. Prof. Dr. Ahmet Oğuz Akyüz _____
CENT, METU

Assist. Prof. Dr. Hüseyin Hacıhabiboglu _____
MODSIM, METU

Dr. Erdal Yılmaz _____
SOBEE STUDIOS

Date: 12.10.2012

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ŞAHİN SERDAR KOÇDEMİR

Signature :

ABSTRACT

VISIBILITY GRID METHOD FOR EFFICIENT
CROWD RENDERING WITH SHADOWS

Koçdemir, Şahin Serdar
M.Sc., Department of Modeling and Simulation
Supervisor : Assoc.Prof.Dr. Veysi İşler

October 2012, 57 pages

Virtual crowd rendering have been used in film industry with offline rendering methods for a long time. But its existence in interactive real-time applications such as video games is not so common due to the limited rendering power of current graphics hardware. This thesis describes a novel method to improve shadow mapping performance of a crowded scene by taking into account the screen space visibility of the casted shadow of a crowd instance when rendering the shadow maps. A grid-based visibility mask creation method is proposed which is irrelevant to scene complexity. This improves the rendering performance especially when there are many occluded instances of the crowd which is a common scenario in urban environments and accelerates the usage of crowds in real time applications, such as games. We compute visibility of all agents in a crowd in parallel on the graphics processing unit(GPU) without having a requirement of a stencil buffer or light direction dependent shadow mask. Technique also improves the view space rendering time by reducing the visibility check cost of the agents that are located on the invisible areas of the scene.

The methodology introduced in this thesis gets more effective in each shadow map rendering pass by re-using the same visibility mask for shadow caster culling and enables many local

lights with shadows. We also give a brief information about the state of the art of crowd rendering and shadowing, explaining how suitable the method with the implementations of different shadow mapping approaches. The technique is very well compatible with the modern crowd rendering techniques such as skinned instancing, dynamic level of detail(LOD) determination and GPU-based simulation.

Keywords: Visibility, Crowd Rendering, Shadow mapping, Shadow Caster Culling, Video Games

ÖZ

VERİMLİ GÖLGELENDİRMELİ KALABALIK ÇİZİMİ İÇİN
GÖRÜNÜRLÜK IZGARASI METODU

Koçdemir, Şahin Serdar
Yüksek Lisans, Modelleme ve Simulasyon
Tez Yöneticisi : Doç.Dr. Veysi İşler

Ekim 2012, 57 sayfa

Sanal kalabalık çizim teknikleri gerçek zamanlı olmayan metodlar ile film endüstrisinde uzun zamandır kullanılmaktadır. Ancak video oyunları gibi etkileşimli gerçek zamanlı uygulamalarda kullanımı, günümüz limitli grafik işlemci performansı nedeniyle yaygın olmamıştır. Bu tez kalabalık içerisindeki bir karakterin ekran üzerinde gölgesinin görünürlüğünü test ederek gölge haritası (shadowmap) oluşturulmasının hızlandırılmasını sağlayan orjinal bir yöntemi tanıtmaktadır. Bunu sağlamak için sahne karmaşıklığından bağımsız bir ızgara tabanlı görünürlük maskesi oluşturma yöntemi öneriyoruz. Yöntem, özellikle kentsel ortamlar gibi görüş kapalılığının etkin olduğu sahnelerde gölge haritası oluşturma performansını iyileştirmektedir. Stencil hafızası veya ışık yönüne bağlı bir görünürlük maskesi oluşturulmasına gerek kalmadan tüm ajanların görünürlüğünü paralel olarak hesaplıyoruz. Teknik, görünmeyen alanlardaki karakterlerin görünürlük testlerini de hızlandırarak aynı zamanda kamera bakışındaki çizim zamanını da iyileştirmektedir.

Bu tezde sunulan metodoloji her gölge haritası çiziminde daha verimli olmaya başlamakta ve bu sayede birçok lokal ışık kaynağının gölgelerinin çizilmesini sağlamaktadır. Ayrıca kalabalık çizim metodları hakkında günümüzde kullanılan tekniklerden bahsedilerek uygu-

lanan metodun uyumlulugu incelenmiştir. Sunulan teknik, kopyalama, dinamik detay seviyesi hesaplama ve GPU-tabanlı simülasyon gibi modern kalabalık çizim tekniklerine ve değişik gölgelendirme metodlarına da uygundur.

Anahtar Kelimeler: Görünürlük, Kalabalık Çizimi, Gölge dokulaması, Gölgeleyici Ayıklama, Video Oyunları

To Tuba and Kadir Agah...

ACKNOWLEDGMENTS

I would to express my deepest appreciation to my advisor Assoc. Prof. Dr. Veysi İşler for his guidance, advices and encouragement throughout the research.

Special thanks to Assist.Prof.Dr. Ahmet Oğuz Akyüz and Dr. Erdal Yılmaz for their support and contribution to the my knowledge and vision. Many thanks to my friend Gökalp Doğan for the graphical contents and his valuable comments for this thesis.

Finally, I am heartily thankful to my wife Tuba, for her endless support and understanding during the study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Virtual world simulation applications such as military training, emergency planning, computer games and architectural design applications, frequently require effective rendering of large numbers of animated characters. Due to the limited rendering power, real-time rendering of massive crowds has always been a challenge. The computational requirement becomes much more higher when rendering the crowd with realistic lighting effects due to shadowing costs. This massive computational and rendering power requirement of crowded environments can be handled effectively with today's GPU architecture by using parallel computation. In this thesis, we describe a GPU-based implementation of shadowed crowd rendering by using a novel visibility masking method to optimize shadowmap rendering by culling the instances that are not going to cast any visible shadows on the screen. Our method also improves the view space rendering by reducing the computational cost of culling invisible agents with the same mask. System supports multiple types of light sources and latest crowd rendering technologies such as skinned instancing, multiple stream output and optimized GPU based simulation.

Many of the today's real-time applications use different implementations of shadow mapping [1] methods for realistic shading of the virtual scenes. Shadow mapping implementations require one or more additional rendering of the scene geometry for the construction of the shadow map for each light source. This has a huge performance cost when rendering a massive crowd.

Various GPU accelerated eye space instance culling and level of detail management techniques are being used to achieve efficient rendering of massive virtual crowds. However effective light space rendering is also important for realistic rendering with shadow mapping.

Common techniques use partitioned shadow maps for better quality to utilize detail management which introduces the problem of culling and rendering the crowd multiple times. Similar problem also arises when using local point and spot lights with shadows. Efficient shadow caster culling becomes crucial to reduce overhead of rendering massive crowd geometry multiple times.

View space frustum and occlusion tests of an agent does not always guarantee the visibility or invisibility of its shadow. These culling methods can be used to reduce shadow mapping cost by applying them in the light space when rendering a shadowmap, but even if they are inside the frustum we can further cull the agents that are not going to cast any visible shadow on the screen according to the occlusion on the screen space.

Naive occlusion query based solution for shadow caster culling is not effective when the light view depth complexity is not high. In addition to light space occlusion, we need to cull any shadow caster geometry which is not going to cast any visible shadow on the screen. One recent approach to shadow caster culling [2] determines visible casters using a visibility mask of visible shadow receivers on the screen. Their implementation depends on a sufficient hardware occlusion testing or predictive rendering with stencil masking. In addition, since their visibility mask is created according to the light view, it should be recomputed for each shadow map rendering of global cascades and any local lights.

1.1 Scope

The main focus of this thesis is improving the rendering performance of shadowed crowded environments for real-time applications. Our main contribution is a novel visible instance culling method which is specially designed for rendering large-scale crowds with shadow mapping from multiple types of lighting sources. Our technique does not force a certain shadow mapping method but improves the shadowmap rendering performance by culling the instances in the crowd that are not going to cast any visible shadows on the screen. Therefore, we provide an overview but do not focus on the analyzing of shadow mapping errors and different warping or filtering methods.

1.2 Outline

The thesis has been organized into the following chapters:

- Chapter 2 provides an overview of the related work and state of the art about the effective rendering of crowded environments and shadow mapping methods in the literature, including various techniques related to crowd rendering, culling methodologies and shadow mapping.
- Chapter 3 explains of the crowd simulation implementation with OpenCL. Its usage to avoid GPU to CPU data transfers and optimization details are e
- Chapter 4 describes the proposed visibility mask system, detailing the implementation side of this thesis and demonstrates the details of general purpose parallel computing for shadowed large-scale crowd rendering with multiple types of lights.
- Chapter 5 details of experiments that have been done to test the proposed method. Explaining the GPU-driven LOD determination and culling optimization details. Chapter covers case studies that demonstrates the integration and the performance of the algorithm. The first case study experiments an open terrain scenario and the other study evaluates the performance in an urban environment. We give a short discussion about the results of the system in various crowd population and different numbers of visible lights.
- Chapter 6 concludes with a summary of our contributions and a discussion of potential future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this thesis, we are focusing on the performance optimizations to shadowing computations on a massive crowd by using the power of GPU. In this chapter, we first introduce general crowd visualization techniques in the literature and show how recent implementations are using to graphics hardware to render a massive crowd efficiently. We describe GPU-friendly acceleration and level of detail (LOD) methods to reduce rendering complexity regarding to the large-scale crowd rendering. Then, we clarify culling methodologies that can be used with scenes with many dynamic agents which include GPU based frustum and occlusion culling. We describe how parallel computation can be used for culling crowd instances efficiently and using the results for rendering without having any GPU to CPU readback. Next, we explain shadow mapping, it's implementations over crowded scenes, problems and recommended solutions to the problems. We also express our main contribution related to 'Shadow Caster Culling' by considering the related work and recent approaches in the literature. Finally, we briefly explain the general purpose GPU usages which became very popular in the recent years.

2.1 Crowd Rendering

Virtual crowds are popular in the films industry since many years. But due to limited computational power, their existence are still quite rare in real time applications, such as games. Rendering cost of a virtual crowd is commonly high and when working on the visualization of large-scale crowds, common character rendering approaches become too much expensive to achieve interactive framerates. In order to decrease the rendering costs, extensive research conducted on using different culling methods and level of detail management systems. In

this section we explain some of the state of the art techniques for rendering crowded scenes efficiently. Also beside the performance problems of the crowd rendering, we explain other problems of virtual crowd generation, such as creating mesh and animation variations to obtain realistically looking crowds.

2.1.1 Crowd Variety

One of the main problems to achieve realistic looking crowds is the variation of the characters. Their appearance, shape, animation and behavior will make them look different. There are many researches to create realistic looking crowds efficiently. Perceptual impact and saliency based methods are studied in recent work of McDonnell et al [3] [4]. While they are focusing on the perceptual recognition of the crowd variety to avoid easily identified clones that look similar, there are researches to support different parametric character creation methods. Rather than creating all variations of a crowd, using randomization methods such as the one that is introduced by Ciechomski et al. [5] is very time-saving for crowd representation creation. They are applying different texture maps to same geometric model which is also a memory saving and GPU friendly method for rendering the crowd. Maim et al. introduced a variation system which can modify individuals' shape, textures and accessories to provide variety with uniquely generated characters in their real time crowd simulation engine, YaQ [6].

Many crowd rendering implementations use instanced rendering with limited base geometric models for performance. In such applications renderer modifies the geometric data and its visual appearance on the GPU by using some kind of randomization techniques for rendering the characters with different textures, colors and scaling. Many of the randomization methods might be handled by the GPU directly with pseudo-random data generation according to an agent identifier for each instance in the crowd. If the application requires any user-defined specific visualization of the characters, commonly a separate data buffer is used to define the per-instance attributes rather than generating them on the fly.

Crowd variety is also important to generate realistic results with the simulation of the characters. This can be achieved by using different per-instance characteristics of agents such as different reaction times or different movement speeds. Another visually effective variation generation technique is changing the animations according to the character which is also used in the work of Ciechomski et al [7]. Different walking and running animations can be used

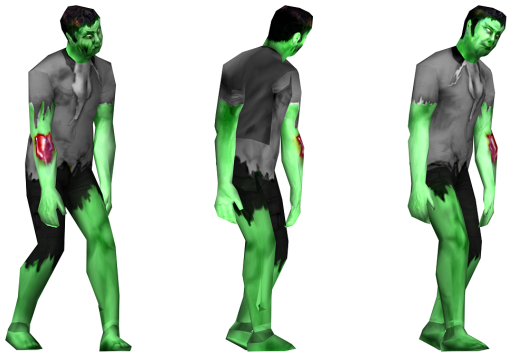

Figure 2.1: Winding order fix with shaders is required after mirroring the animation data for motion variety

to enrich the user experience. Techniques to modify the animation data on the fly in order to differentiate the visualization and the simulation without preparing large animation data-sets might be considered. A simple modification on the animations is to create a symmetry effect in the vertex shader by inverting side axis vector of the instance world matrix. This modification will change the winding order of the vertices and this will cause your front faces appear as back faces (Figure 2.1). Changing vertex order in the geometry shader stage can be used to swap the ordering but we achieved better performance by disabling the back-face culling altogether. This result is relevant to the complexity of the pixel shader used in the mesh rendering and the overdraw performance. Another option is to separate the instanced render calls for the meshes with modificated animations.

In our implementation, we use single texture-atlas to define different diffuse maps for the characters. Texture coordinates of the original mesh are rescaled and biased according to the agent ID and the number of atlased textures. In our sample we use two cascaded textures together and adjust the x-axis of the original texture coordinate. Each diffuse map in the texture-atlas contains a colormask in the alpha channel. For each agent instance, a random color is fetched from a color palette and diffuse color is changed according to the colormask. The generation of the instance color can also be performed HSV space to create more randomized colors [5]. We also achieve geometric variation by scaling the agents randomly according to their IDs. Figure 2.2 shows the diffuse and color mask maps used for the rendering and the different characters generated with the methods explained.

Figure 2.2: Variety of characters generated using the randomization techniques with a color-masked texture atlas

2.2 Level of Detail Techniques

Figure 2.3: Discrete geometric detail level meshes of the same character with 652, 548 and 382 polygons

Level of detail (LOD) has always been a popular research area that is focused on the trade-off between the performance and the complexity. In terms of graphics, different methodologies are used to create less visual artifacts when trying to reduce the rendering costs by controlling the importance of a rendering element. Resolution management is done by courtesy of limited human perception and is related to the final image. But the rendering detail level is also effective on the inner states of the pipeline such as shadowmap rendering. We have discussed the recent techniques to control shadowmap detail over the scene in the shadow mapping section. Beside the visual level of detailing methods, there are also different methods to avoid computational costs of a massive crowd simulation by applying different detailing systems to

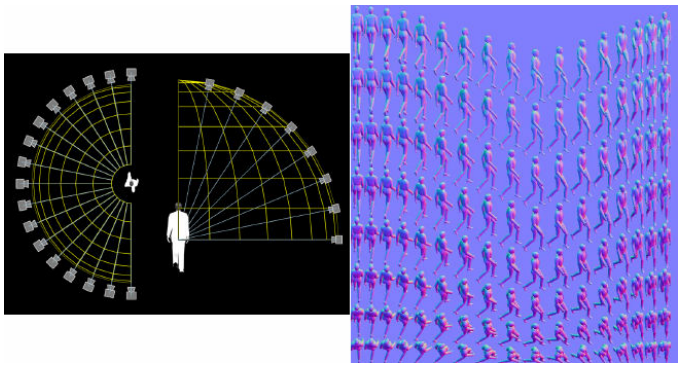

Figure 2.4: Creating impostor atlas of a character (Courtesy of Simon Dobbyn)

artificial intelligence, physics and animation computation [8]. But details of these methods are out of the scope of this thesis.

Discrete geometric LOD levels are very common in real time applications. Figure 2.3 shows different LOD levels of the same mesh where in each level polygon count is reduced to give better performance. These meshes can be generated automatically, scanned or can be hand-made by graphics artists. While instanced rendering saves us CPU cycles by reducing the required draw calls to render high numbers of meshes, we also need to reduce the required rendering power for the meshes by simplifying them. A crowded scene with 10k visible 1000-polygon agents means 10 millions of polygons to render. We don't need that much detailed meshes for far distances. Using very low-polygon meshes for distant characters would result in visual artifacts that are noticeable on LOD transitioning due to the inaccurate mesh silhouette. Impostors and geopostors could be used to keep the similar silhouette on the low polygon meshes as well as improve the rendering performance [9, 10]. Impostors are multiple 2D projections of a character rendered from different horizontal and vertical angles at different keyframes. Pre-rendered image count directly effects the visual quality and for performance, all the images are packed to a single texture atlas as shown in the Figure 2.4. Polypostors are using similar technique but they support animations and targeting to reduce texture memory usage [11]. Both approaches are assuming crowd is using a limited set of animations which is a common assumption when rendering massive crowds.

On the other hand we might need to render very high-polygon, maybe tessellated geometry for the agents near to the camera. Basic discrete geometric level-of-detail (LOD) determination could be applied to all meshes in parallel. Multi-pass rendering of all agents to define LOD

targets by changing the visible range in every pass could be used to define detail levels. But with the features of modern graphics APIs, re-arranging and sorting them can be done easily in a single pass by using the multiple stream outputs feature. After checking the visibility of an individual, we can set the LOD level according to the it's distance and select the corresponding stream buffer target for the agent which will be then used for instanced rendering.

Another type of graphical level of detail representation for crowd rendering is using points to render the massive crowd instances at further distances. Using crowd rendering method used in a work of Rudomin et al. [12], point based rendering replaces the mesh with a pixel-sized points cloud.

Keeping the performance of the crowd rendering same during the runtime is important for the robustness of the system. When many characters get near to the camera we need to reduce the detail level distances according to the visible mesh counts to keep the rendered high polygon mesh count alike. Similar automatic level of detail adjustment is also used in ATI March of Froblins demo [13] which also use hardware accelerated tessellation for further detailing the original mesh on the fly for better visualization of the characters near to the camera.

2.3 Visibility

Hidden surface removal algorithms are being used in graphical applications for a long time. They enable avoiding waste of rendering power for the objects that will not contribute any visible effect on the final image. Visibility operations can be accelerated by using pre-computed data sets which are called; Potentially Visible Sets (PVS). PVS data creation is a time-consuming task and effects the production pipeline negatively. It creates a visibility set output from the static obstacles in the scene with a limited range of sectors. At runtime only the given subset of static scene elements and sectors are drawn according to the sector that is the camera is located. In a crowded scene PVS methods become impractical due to large numbers of dynamic elements and run-time visibility calculations become practical to avoid keeping the scene visibility hierarchy over the entities stable. Basically visibility calculations can be classified into the three types which are shown in Figure 2.5:

- Back face culling enables to avoid drawing back facing polygon pixels of closed meshes where its guaranteed to be over drawn by the front facing polygons of the same mesh.

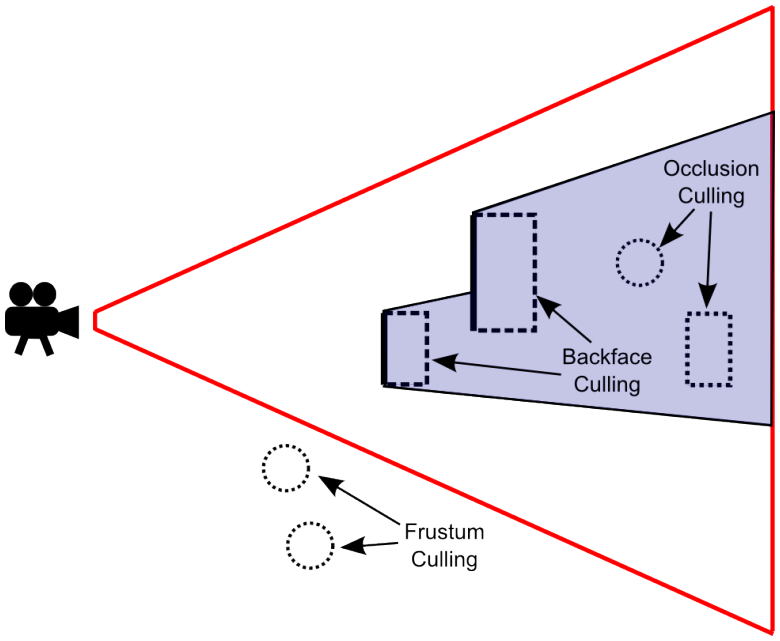

Figure 2.5: Illustration of back-face, frustum and occlusion of culling methods

- Frustum culling avoids drawing the objects that are outside of the view frustum.
- Occlusion culling culls the objects that are occluded by some other object in the scene.

Back face culling is one of the simplest culling methods to apply on any scene. It assumes that the model's polygons are modeled as being only shown from one front side. For double sided objects, back facing culling is disabled or back faces are added to the original mesh as well. Frustum culling on the other hand takes bounding volume of an entity and the camera parameters as the inputs and checks if the object is laying inside or outside of the view frustum. Aggressive frustum culling can be achieved for the triangles inside a model but modern GPUs' primitive level clipping mechanisms are designed to apply that culling functionality in a hardware accelerated manner. Further details about optimized frustum culling techniques by utilizing bounding volume hierarchies can be found in the work of Assarsson and Möller [14].

2.3.1 Occlusion Culling

Back face culling and frustum culling is being used in real time applications. On the other hand, by using occlusion culling, any entities hidden by others can be discarded. The idea

behind the occlusion culling is to make some simple tests before sending all rendering data to the pipeline. Depth complexity of the scene is the key factor of this method's advantage which is illustrated in Figure 2.6. High depth complexity is a common scenario when rendering an urban or an architectural scene. Both cases might have crowded environment that the individuals need to be occlusion tested to reduce overdrawing.

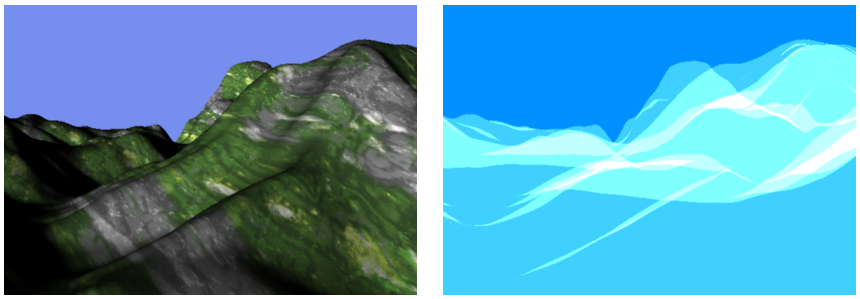

Figure 2.6: Visualization of the depth complexity on a terrain scene

Umbra Software develops one of the leading culling libraries for real time applications, Umbra 3 [15]. Their solution optimize visibility checks of the scene entities and provides better performance by optimizing the rendering, content streaming and the game logic. They use pre-computation for to create better runtime performance and their Umbra Occlusion Booster application use GPU accelerated occlusion and avoids GPU to CPU read backs when rendering the scene. Hierarchical visibility (HV) algorithm for occlusion culling is proposed by Greene et al [16]. They partition the scene model in an octree structure and creates an image pyramid from the frame's z-buffer. Hierarchical culling of occluded scene areas is maintained by the octree structure and the z-buffer pyramid enables the occlusion tests of each bounding volume. However, the octree structure requires non GPU-friendly recursive iterations by nature due to the octree and visible object count read backs of visibility checks to stop or continue to subdivision operations [17]. GPU-friendly hierarchical z-buffer creation part of the method is explained in the implementation chapter of the thesis. Different techniques has been researched to achieve effective occlusion culling. See [18] for a detailed survey of the techniques.

In complex scenes, testing all the entities' visibility one by one is a time consuming job and the cost can be reduced by using a scene visibility hierarchy or parallel computation. Special rendering pipeline of crowded scenes can also benefit from occlusion culling by customized

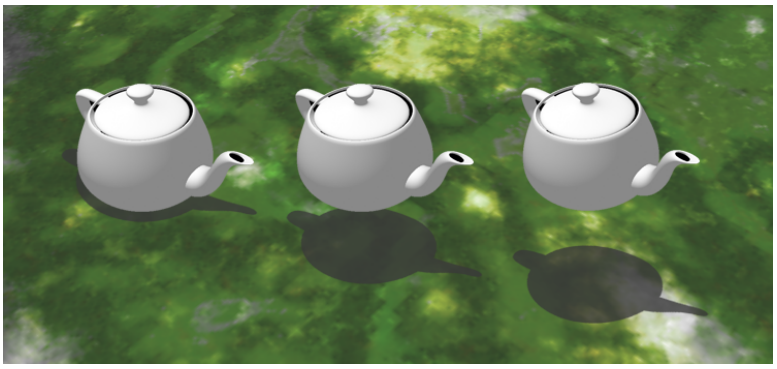

Figure 2.7: Shadows clarify the geometric coordinates of the objects

and parallelized visibility tests. When simulating and rendering all of the crowd agents with geometric instancing, CPU will not be aware of their visibility as all the position data remains on the GPU side for better performance. Fortunately, we are able to use stream output and transform feedback features of modern GPUs to perform frustum culling and query how many of the characters are visible in parallel. Those features enable us to use a vertex shader to implement culling tests and a geometry shader to emit only the visible instance vertices to the destination buffer. In our implementation frustum and occlusion culling is done in vertex shader stage. Then, at the geometry shader stage we do not emit the vertices to the target buffer if the agent is not visible. This way we are able to check if an agent is culled or not and get how many agents are visible to create LOD-distance optimization. Details of the Hi-Z buffer construction and visibility checking for the crowd instances are explained in the implementation chapter of the thesis.

2.4 Shadow Mapping

Lighting effects provides realistic rendering and shadows are an important element of lighting in a scene. Figure 2.7 shows the effect of shadowing and how they give visual hints to clarify the geometric hierarchy of the objects and lights.

Main shadowing techniques that are commonly used in real time applications are are planar shadows, shadow mapping and shadow volumes. Ray traced shadowing, on the other hand, is a popular method for offline rendering which also enables soft shadows.

Planar shadows can be created with an object's projected geometry onto a surface, called

shadow plane. Rendering the projected geometry with a customized shader will give the effect of shadowing for that object. The technique is limited scenes with flat floor geometry. Planar shadows are fast for the scenes that do not contain many shadow caster polygons as it only requires to re-render the same geometry for shadow creation. No self shadowing, z-fighting and high overdraw complexity is other problems of the technique. Kilgard [19] presented a solution regarding z-figting problem using the stencil buffer but the flat shadow surface requirement of this technique makes its impractical for many scenarios and scenes. In crowd rendering context, Loscos et al. [20] use similar planar shadows with projected imposter images and 2.5D environment shadow maps.

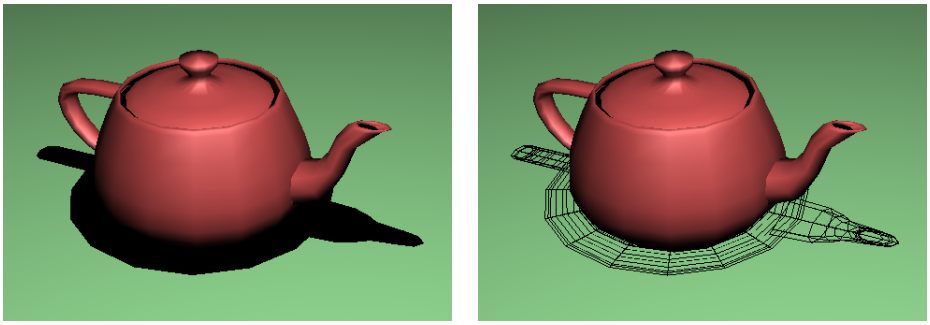

Figure 2.8: Planar shadows projects the shadow caster geometry onto the shadow plane

Geometry based shadow volumes algorithm is first described by Crow [21]. Technique avoids the aliasing problem of shadow mapping by introducing pixel perfect hard shadowing by using extruded volumes of the shadows casters. Figure 2.9 shows the constructed volume geometry and the shadowed area which is identified with the intersected pixels being inside of the volume. Common implementations of shadow volumes use stencil buffers to check

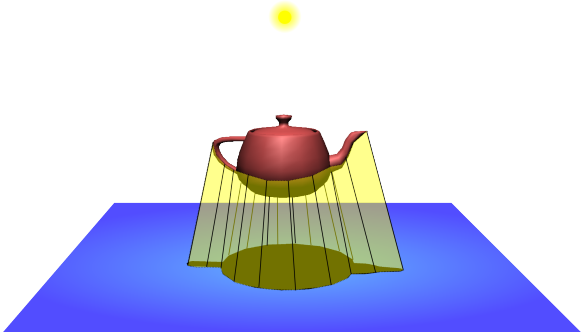

Figure 2.9: Shadow volumes technique extrudes the shadow caster to create a volume to be shadowed

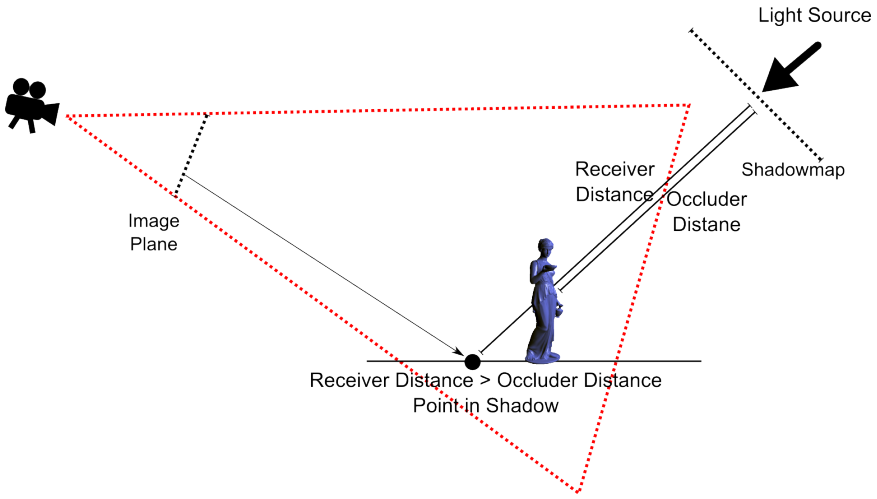

Figure 2.10: Shadowmap technique illustrated.

if a pixel is inside or outside of the volume area by making multiple render passes of the volume geometry. Despite the fact that shadow volumes create less aliasing, their complexity and performance depends on the polygon count of shadow caster polygons and technique is not work well with the alpha blended geometry. High fill rate makes the algorithm polygon limited for real time applications. Another problem of the method is, it creates very sharp edged shadows and soft shadowing needs to solved by introducing smoothies [22] or hybrid approaches [23].

Image based shadow mapping first described by Williams [1]. Standard Shadow mapping technique is illustrated in Figure 2.10. Technique requires a separate shadow map rendering pass to create shadows on any type of surfaces. Shadow map creation pass stores the depth values of the scene elements from light point of view. At camera view rendering, we can check if any point is in shadow or not by evaluating the difference of its real distance to the light source and it's light-space projected distance at shadowmap texture. If point distance is greater than the distance that is fetched from the shadowmap, point is has a shadow caster between itself and the light source. Sample shadowmap and resulting shadowed scene is shown in Figure 2.11.

Shadow mapping is widely used and become the de-facto standard in recent years for real time applications. Resolution of the shadow map is a factor to the aliasing due to the resulting discretized image. Many techniques have been developed to avoid aliasing and over sampling problems to provide better shadow quality. For a detailed analyses of different approaches

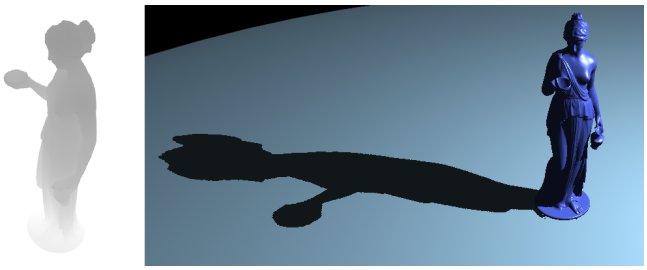

Figure 2.11: Rendered shadowmap(left) and the resulting final image(right).

to shadow mapping, we refer to a recent survey created by Scherzer et al [24]. Their survey focus on hard shadow mapping algorithms and gives hints about which algorithms are suitable in what situations. Shadow mapping optimization techniques in the literature can be divided into four parts:

- Utilizing Shadowmap Space
- Shadow Sampling Performance
- Shadow Update Frequency
- Shadow Caster Culling

2.4.1 Utilizing Shadowmap Space

Projecting a single shadowmap texture onto a large perspective view frustum creates under and over sampling errors that can be seen in Figure 2.12. The shadowmap space become under sampled at near distances and oversampled at further distances. Utilizing the shadow map space for these scenarios is crucial for optimized shadowed scene rendering without using huge-sized shadow map textures.

2.4.1.1 Warping

For many scenarios a transformation matrix can be applied to shadow rendering for providing better shadowmap sample distribution at closer distances to near plane. Perspective shadowmap maps (PSM) is proposed to apply such a warping effect by Stamminger and Drettakis

Figure 2.12: Illustration of the undersampling and oversampling scenarios.

[25]. Their algorithm apply a perspective transformation which is created from viewer camera projection. Changing the sampling densities globally before rendering the projected scene entities is useful to utilize shadowmap space. PSM transform the scene geometry view dependently and quality is dependent on the near plane of eye-view. Wimmer et al. [26] introduced a new method, light space perspective shadow maps (LiSPSM), which avoids this problem by warping the the shadow space according to the light and view transformation.

2.4.1.2 Partitioning

Providing higher shadowmap resolution at near distances to the camera is possible by using multiple shadow maps that totally covers the effective shadowed area. For a global directional light sources, such as sun, partitioning the scene according to depth is practical as the shadow resolution requirement changes perceptually due to the distance. Parallel split shadow maps (PSSM) [27] and cascaded shadow maps [28, 29] use similar idea to utilize shadowmap space. The partitions can be rendered to a single shadowmap with an offset applied to each split or texture arrays can be used. Figure 2.13 shows a scene using CSM with three splits. Sampling density decreases at far distances as the same sized shadowmap texture covers a larger area at each cascade.

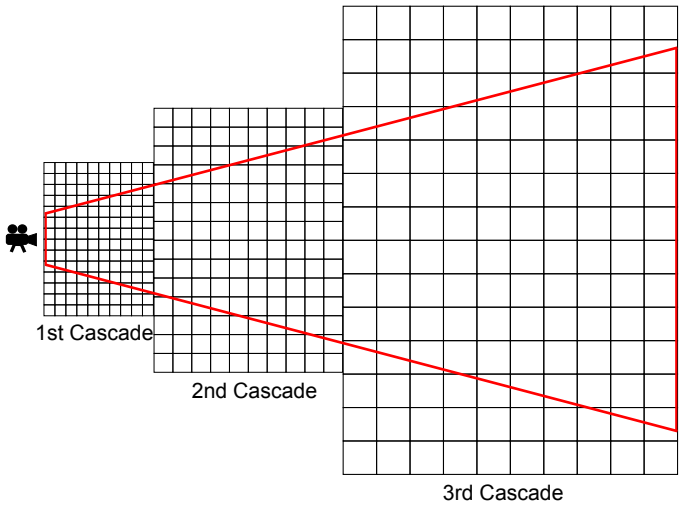

Figure 2.13: Cascaded shadow mapping technique with 3 cascades

Different approaches have been studied to create the cascades at optimal distances. One of the recent algorithm use histogram of the scene depth values to create better shadow distribution over the scene [30]. Their method works well especially on the scenes that have many occluders as the depth of furthest visible pixels changes a lot and makes the view frustum tighter for shadow mapping. Warping algorithms can also be combined with partitioning to create better sampling distribution at each partition.

2.4.1.3 Irregular Sampling

Checking a pixel's shadowing state from the shadowmap requires a projective texture sampling. Aliasing artifacts happens as the sampling location commonly do not correspond to the exact query position and the resolution of the shadowmap. Using irregular sampling methods targets to create a shadowmap with the samples from the positions that will be queried later on. This way the artifacts could be minimized. Finding desired query locations needs another eye-space rendering pass and projection of the visible scene pixels to shadow map space. But irregular rasterization is required to find depth values of those query locations which is not map well to current regular grid rasterization hardware. Johnson at al. propose using a list of queries at each render buffer to enable irregular sampling as a hardware extension [31, 32].

2.4.2 Shadow Sampling Performance

Shadow sampling becomes costly especially when dealing with soft shadows. Blurring the shadowmap itself will not produce the correct and smooth results as the shadow map results are binary. Dong and Yang proposed variance shadow mapping [33] which enables editing the shadow map. On the other side, many soft shadowing techniques requires blurring multiple shadow sapling results to visualize the penumbra region of the shadows. Minor sampling cost is also included when rendering a scene with high depth complexity as the results of the shadow calculation might be discarded by another entity that is drawn over it. Today many real-time rendering systems choose deferred approaches to avoid pixel overdraw costs by calculating the complex lighting and shading in image space. Image space shadow mapping is not a new concept and can also be easily used to create soft shadows by applying a smart blur filter to the deferred shadow mask. Deferred shadow calculation also avoid over sampling where the calculated shadow will not be visible due to a depth test.

2.4.3 Shadow Update Frequency

Many shadow mapping methods use view dependent optimizations and requires shadowmap update at each frame even there is no entity movement in the scene. Using a lower update frequency when rendering of far shadow splits is an optimization used for many game engines. In some cases, a static shadow split could be used for furthest split. This optimization reduces the rendering cost of huge geometry which is encapsulated by the largest cascade in partitioned shadow mapping implementations. Local lights are also benefit from shadow update frequency editing as well as the global lights. Visibility checks for local lights could be used to check if any shadowmap update is required for a local light. In this manner, Ryder and Day [34] use shadow resolution determination according to distance to reduce fill rate required to render all point light shadows.

Using static shadows is not always practical where the entities in the scene changes the shadow structure frequently. However, we can also optimize the rendering cost by not updating the full frame of the shadowmap but implementing incremental updates. Rendering only the most important parts of the shadowmap each frame would give better performance by enabling a semi-static shadow map.

2.4.4 Shadow Caster Culling

Culling methods target reducing the number of triangles to be rendered without effecting the final image, such as discarding any primitives outside of the view frustum. In case of large scenes with high depth complexity, occlusion tests becomes practical to optimize rendering. Scene hierarchies such as octress or quadtrees could be used efficiently for the scenes with many static obstacles. Scenes containing many dynamic elements such as a crowd instances, could use same hierarchies but the complexity of maintaining the hierarchy up to date increases. We refer to [14] and [18] for further details on optimized frustum and occlusion culling techniques.

Occlusion culling method can improve the performance of crowd rendering in urban environments as a many of the crowd instances will be occluded by buildings. Rendering the crowd for main camera view is only one of the most complex passes in a shadow mapped scene. Many more rendering passes would be required to create shadow maps of the crowd. Occlusion culling from the light point of view would not create much performance improvement for the renderings with low depth complexity. But the idea behind the shadow caster culling is making the visibility calculations for the casted shadow according to the camera view. Any entities that is not going to cast any visible shadows to final image does not needs to be rendered to shadow maps too. See Figure 4.1 for an illustration of this situation. One of the recent work of Bittner et al [2] creates a shadow mask with various types of receiver mask generation methods and use stencil masking to optimize shadow mapping cost for the entities that to not contribute to final image. They show that best performance is achieved by creating a per-fragment mask for each shadow rendering. They only use global directional lights but for a crowded scene, shadow caster culling could be used for local lights too. Updating the shadow mask becomes a overhead when rendering multiple shadows. Our technique avoids this overhead by creating a shadow mask that is irrelevant to scene complexity which can be used for any shadow map rendering efficiently.

2.4.5 Crowd Lighting

As shadowing play an essential part in real-world lighting, realistic rendering of large-scale crowds requires shadows of the characters. Absence of shadows is easily noticeable for the

human eye, and their presence improves the realism even if the scene is not rendered in a photo realistic way [35]. Many large-scale shadowing technique use partitioning methods for utilizing shadowmap space and this means multiple rendering of the massive crowd geometry. Altough its advised to use partitioned shadow maps for crowd lighting [13], some methods tries to reduce vertex processing cost by rendering the skinned vertices of the crowd geometry to a separate buffer, then rendering the contents of the buffer for each shadowmap pass. This kind of implementation only suitable for the scenes that have no more than a limited numbers of characters as it requires a buffer area for dumping all the vertex shader output. Also technique does not support geometry variation, level of detailing and culling effectively.

The performance requirement is much more higher when we need to visualize the crowd with correct lighting and shadowing for better visual experience. Loscos et al. propose a shadowing technique for crowds in virtual cities which creates polygons with the imposters that are projected onto the flat ground geometry [20]. They use global shadow to the crowd with a 2.5D shadowmap from which application could query the coverage of shadow at any given area. Their technique is not suitable for complex environments, self shadowing and shadows between dynamic objects conditions. Dobbyn et al. presented and improvement for the method by using a stencil buffer to avoid z-buffer fighting[dobbyn DHOO05] but planar shadows also create high fill rate for massive crowds. Ryder and Day present an optimized technique for high quality shadows for real time crowds [34]. Their algorithm support uneven surfaces and self shadowing via augmented depth imposters. Their work offers using the light's radius and a spatial hierarchy to cull unnecessary agents while rendering the shadow maps of the local lights. We are using a similar approach in our implementation and additionally we support occlusion culling tests to avoid non required shadow casters and cast shadows from non local lights which can effect all the individuals in the crowd.

Perceptual effects of low frequency lighting over dynamic crowded scenes is analyzed with a recent research of Jarabo et al [36]. They claim that they do not examine hard frequency shadows as they cannot be ignored in many conditions due to the hard shadow edge contacts but their results are important for optimizing crowd lighting without observable artifacts. They express that the lighting errors are masked more easily in complex aggregates with random motion and coarser interpolation schemes becomes usable. Future work of such a research might include hard shadows and propose a method for optimizing shadow generation of aggregated environments.

2.5 Graphics Processing Unit For Parallel Computing

Recent developments on the technology of the Graphics Processing Units (GPU) brought excessive interest to solve common computing problems with parallel processing using the GPU. Central Processing Units (CPU) have high processor speeds but unlike GPUs they are not designed to run hundreds of threads in parallel. Architecture differences force developers to implement data parallel algorithms and reduce the memory exchanges for effective usage of many cores on the GPU. In this section, we describe general purpose computing on graphics processing unit (GPGPU) motivation and recent approaches that use GPU in crowded environments for performance. We briefly introduce the mechanics, execution and memory model of OpenCL, as we are using it in the implementation phase of this thesis.

2.5.1 General Purpose Computing on Graphics Processing Unit

Having a GPU consists of multiple cores, utilization can be achieved by doing the computation in a massively parallel way. In a crowd simulation scenario, we can compute each individual agent's artificial intelligence updates in a separate thread to create a highly parallel crowd integration. But any data dependency, like requirement of another agent's position or other data from an integrator thread -for collision or avoidance- would result in conflicts on the memory reading of the kernels and cause performance drops if implemented recklessly. We explained how we can achieve better performance when checking collisions between the agents in the implementation chapter. A common GPU based computation model which illustrates data and computation flow is shown in the Figure 2.14.

Programming on GPUs is being performed since 1978 [37]. But GPUs are not limited to usage of being a graphics engine but with the GPGPU concept, using the GPU to accelerate general purpose computation became popular in recent years. Enabling huge performance with the parallel computing, there turned out to be a rapidly growing interest in developing applications for these units. Today's modern GPUs support data-parallel computations which enables multiple code executions at the same time which provides more than teraflops per second computing power [38]. The performance commonly depends on the data dependencies and branches to be executed.

Using parallel computation for the massive crowds are being researched and common tech-

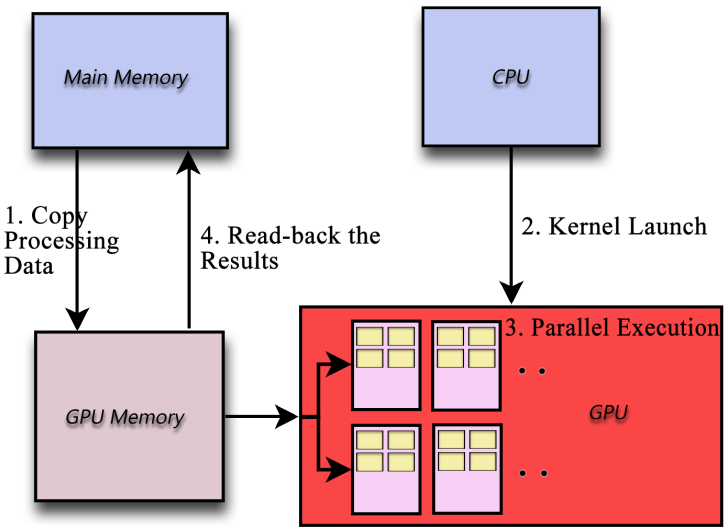

Figure 2.14: Common GPU-based computation model.

nique is to simulate of the crowd by running each instance's AI with a thread [39] [40]. In ATI's 'March of the Froblins' demo, similar technique is applied with the vertex and pixel shaders by using regular rendering operations such as depth testing and alpha blending [41]. All path finding and collision detection is done on the GPU, but this kind of implementations becomes rather complex due to the dependencies and rendering operations to get outputs of the simulation computations, as we have the new options like CUDA, OpenCL and comput shaders. In a recent work Yilmaz et al. modeled soccer game spectator behaviors with parallel computation using CUDA [42].

2.5.2 OpenCL

OpenCL is a heterogeneous computation framework. OpenCL is the open standard for paralel computation proposed by Khonos Group. OpenCL is supported by multiple vendors and unlike CUDA can be used with non-nVidia devices too. Also while CUDA only being able to run on the GPU side, OpenCL supports different devices like CPUs, GPUs and any other hardware that is available on the system. This improves the performance by utilizing the maximum computing power by using all computational units. OpenCL was firstly initiated by Apple and being developed until then. See latest specification guide for the recent changes and improvements [43].

In this section we present the OpenCL architecture for better understanding of the related implementation details of our thesis. The OpenCL specification identifies the system with four different models: the platform model, the execution model, the memory model and the programming model [43]. We are using OpenCL to simulate the crowd and the detailed explanation of OpenCL is beyond of this thesis. In the following parts of this chapter, we briefly explain how the architecture works, how and when it achieves better performance than the CPU and the necessary parts to figure out the remaining chapters.

The platform model includes a host device and one or more OpenCL devices. These devices are named as Computate Units (CU). Several Processing Elements(PEs) are located in each CU. When we consider a modern CPU as a compute device for OpenCL platform, each core is a processing element for the execution. Host device issues the commands to available OpenCL devices over a context and command queue of OpenCL.

OpenCL device codes needs to be compiled before the execution on the GPU. This is similar to regular shader compilation for graphics rendering but OpenCL programs are called as kernels. Each kernel has an entry point with custom arguments and might use other device functions and resources. Kernel execution commands are enqueued by using a command queue defined with an OpenCL context. Command queue is also used for memory operations and synchronization commands. Command queue can be executed out-of-order to provide task parallelism when doing independent device operations. Kernels are executed by threads, called work items which are divided in work groups. Kernels are invoked over work items that are belonging to an 1,2 or 3 dimensional index space(NDRange) which defines the indexing domain of the kernel.

Memory operations should be handled very carefully when programming GPU kernels. Most of the time, memory operations are the bottleneck of many implementations and the algorithms should be designed by considering the effect of memory transfers and inconvenient memory read writes. Hierarchical memory model splits the memory into a thre types: Global, local, constant or private [43]. Global memory can be read and written by both the compute device and the host device but its performance is low. Higher performance memory read/writes can be performed using local memory. But local memories are only accessible to work items that belong to the same work group. Scalar objects that are defined within a kernel program use constant memory space.

Task parallelism is also supported by OpenCL as well as the data parallelism by being able to execute many kernels simultaneously. This improves the performance of the applications that have kernels with no inter-dependence to each others outputs.

2.5.3 OpenGL Compute Shaders

Recently there have been some work on OpenGL to support computing mechanism cleanly without having need to learn OpenCL or to compete with the interoperation issues. OpenGL Compute Shaders are the new concept to make compute-intensive operations with the ease of classic shader management [44]. These shaders are comparable to 'Direct Compute' mechanism of DirectX which can be used with DirectX 11 compute shaders. But currently its a new technology which needs to be supported with new driver updates and it seems to take some time to be a reliable standard for compute operations.

One of the main advantages of OpenGL compute shaders is, being able to decrease the code modifications and make you to be able to bother with the initialization of GPU resources and kernels when doing computations as in the OpenCL or CUDA. Another advantage is that, OpenGL 4.3 can be used to execute mobile OpenGL ES 3.0 applications on desktop. Additionally texture compression formats like EAC and ETC2 are supported. While presenting a richer feature set than OpenGL compute shaders, OpenCL requires installing a separate driver and libraries. Writing kernels with GLSL is again a plus for the compute shaders as many of the OpenGL developers are familiar with the GLSL language.

CHAPTER 3

GPU ACCELERATED CROWD SIMULATION

Many of the today's high quality video games and real time applications have performance bottleneck due to the high amount of data to be processed on the GPU and the existing data bandwidth of the CPU and the GPU. Minimizing the data traffic and the synchronization is crucial to avoid high performance and stalls. On the other hand, we can use the parallel processing power of the GPU to make heavy computations. This section describe the method we are using to simulate a crowd in a video game scenario. Data decomposition is performed to reduce memory footprint of the crowd instances and OpenCL is used to handle agent-agent interactions with a 2D spatial grid structure.

3.1 Data Decomposition

Implementing GPU-oriented solutions to problems requires both parallelizing the algorithm according to the parallel code execution architecture and planning the memory transfer and data decomposition correctly. Memory transfer speed may be the bottleneck for many of the applications that require output results from the GPU process. It is recommended to reduce real-time GPU-CPU memory transfers. Removing those transfers by doing even serial parts of the application in the GPU might result in better performance in some cases. Memory footprint is also important for cache efficiency and supporting much more simulation instances at once effectively, even at the absence of any dynamic memory transfers during the simulation.

We present a GPU-driven crowd simulation model which is not doing any GPU to CPU data transfer in the simulation step. We have separated our implementation into two steps to show how to simulate crowds with both GLSL and OpenCL. In the linear simulation step with

Table 3.1: Memory decomposition of a crowd instance for the crowd simulation

Texture Buffer Object	**r**	**g**	**b**	**a**
Base data	position.xyz			agent_id
Simulation data	keyframe	rotation	velocity	ai_state

shaders, we use texture buffer objects to hold per-instance data and make a draw call with point primitives to integrate each agent within a vertex and geometry shader processing stages. In the more complex OpenCL simulation part, we run one thread for each agent and query local neighborhood via spatial grid optimization. In both cases we should keep our per-agent data small to be able to use local memory more efficiently and decrease memory transfer amount. Fortunately, latest shader models support bitwise operations and we have the ability to pack many low-range variables such as state flags to a single variable. This technique is also very commonly used in many programming and scripting languages. Quantization techniques can also be used to reduce memory footprint for the variables. Choosing computation over data size generally yields better performance in the applications with memory-related bottleneck. Table 3.1 shows the usage of our texture buffer objects to hold per-instance data.

3.2 AI State Management

Creating an AI mechanism for a single crowd member which will be applied to all characters generally requires some knowledge about the scenario. Yilmaz implemented different scenarios with Fuzzy Logic using CUDA [45]. The scenario rules of their systems are hardcoded to the kernels for better performance. Output variables of the simulation might also vary according to the scenario elements. Some scenarios, such as simulating team fans in a stadium, do not require instance position changes and movement physics but try to achieve natural looking randomized movements at close and far distances. Another common usage area for crowd simulation is emergency planning which need to be implemented by being aware of aggregate human body physics and navigation intelligence. A popular simple method to simulate crowd movements is flocking which applies three forces of seperation, alignment and cohesion to the agents for realistic flock behavior [46]. While flocking is suitable for simulating bird or fish flocks, social forces model which applies agent and obstacle avoidance forces to the crowd members creates better pedestrian movement [47]. There are many navigation and

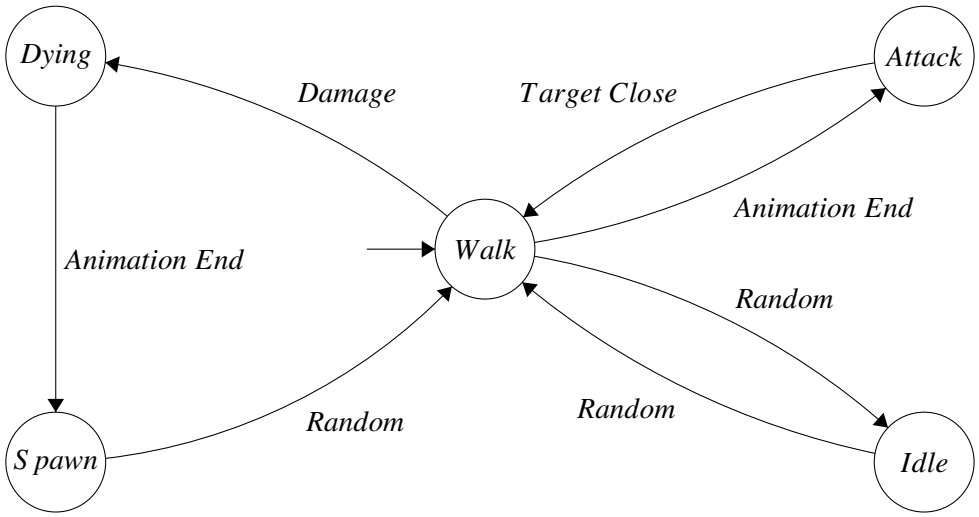

Figure 3.1: State machine of the crowd agent

social interaction related researches for effective and natural crowd movement. We are not going to cover complex AI mechanisms and physical solvers since they are beyond the scope of this chapter. We have used finite state machine (FSM) systems since they are enough to implement required simple AI management and support the capabilities of the GPU. Figure 3.1 illustrates the FSM used to make decisions and move our agents.

There is no data dependence between the agents in the linear integration step and we have the ability to integrate this part of the system in parallel for all agents by using a shader-based way easily. By making a draw call of n-points, after disabling the fragment output, we can use the vertex and geometry shaders to integrate the state machine and set new attributes for the agent. Every character decides what to do on their own at this linear stage of the crowd simulation. There are conditional branches which bring the AI to a new state or change other properties of itself. We also provide an array of uniform randomization variables to shaders and use computation based pseudo-random number generation mechanism together to be able to create more realistic random behaviors between the crowd individuals.

3.3 Collision Detection and Avoidace

Real world situations requires different characteristics for the simulation world. Characters cannot move at the same speed on the all areas of a terrain. Similarly, there might be some

Figure 3.2: Crowded scene with terrain, static obstacles and agent-agent interaction

places or objects that characters cannot walk through. We can model those objects and places with simple primitives such as bounding boxes or spheres rather than their original 3d mesh representations. By giving the bounding volumes of world's collision objects, we can easily perform world collision detection and avoidance operations in the linear character's integration phase. In this chapter we have implemented bounding sphere and box tests for world static collision objects. User can specify several different bounding sphere areas and the characters will do the collision tests with those objects. Since the data required to represent those volumes is small, we are able to use uniform buffers to specify them.

Using a physical structure for the terrain is not required for many scenarios since we are able to use terrain heightmap texture directly on the GPU side. Sampling the terrain height at the culling phase and caching it for per-instance, rather than using character's vertex shader at the mesh rendering stage, prevents us to fetch the height value more than once for each agent.

3.4 Interacting Agents with OpenCL

Repositioning the agents based on their current states and attributes might result in undesirable situations such as penetrating bodies. Using vertex and pixel shaders to simulate the crowd can be implemented very easily for the scenarios with $O(n)$ complexity, such as the velocity and acceleration integration or animation control of the character. But for the interactions of

agents with each other to create realistic and non-penetrating characters, we need to create a data structure to handle this excessive non-linear $O(n^2)$ operation effectively. Its apparent that brute force collision detection or avoidance checking for all agents by going trough all other agents is not an efficient method for the GPU architecture. This would result in crucial performance problems when simulating thousands of agents because of the workload, memory accessing conflicts and data dependence between the threads.

In a common scenario, we don't need an agent to check collision with the agents far away. Since we are not concerned about very high number of distant agents, we can early-out them by using a scene hierarchy. There are numbers of optimization techniques to find the nearest and k-nearest neighbors on a 2D/3D spatial grid [48]. Spatial subdivision techniques divide the simulation space so that it is easier to find the neighbor elements of a given element by just checking the elements in the neighboring cells. Many of the collision detection, scene management and ray casting systems use a scene hierarchy to decrease intersection tests by using early-out mechanisms. One of the most popular techniques is using uniform grids. A uniform grid subdivides the simulation space into a grid of same-sized cells. The fastest subdivision method would differ according to the entities' placement, count and the platform. We have selected uniform grid system to implement agent-agent collision checks because of the re-usability of grid index for different algorithms and fast re-calculation of grid indices. Renderer API based techniques could be used to create the spatial grid structure in some of the previous GPU-based collision detection tests [49]. Similarly ATI's 'March of the Froblins' demo [13], uses multi-pass rendering approach to bin the agents to a regular spatial grid consisting of a color buffer to control bin-load and a depth texture array to store agent IDs. These implementations use pixel shaders to produce agent neighborhood data by the help of depth testing and additive blending operations to bin the correct agents to different depth targets and count them. But this kind of implementation has a rigid structure with specific rendering pipeline and rasterization operations; changing it to a different spatial partitioning system, maybe to KD-trees [50], to handle different scenarios and agent distribution over the scene, would be difficult to implement and control. Using OpenCL to control spatial structure creation, update and agent collision checks is a cleaner way to create an extendible and easy-to-control system since we are not dealing with any graphics rendering operations such as blending or depth testing. In the following sections, we will describe the implementation of a 2d uniform grid with OpenCL to optimize agent-agent interactions.

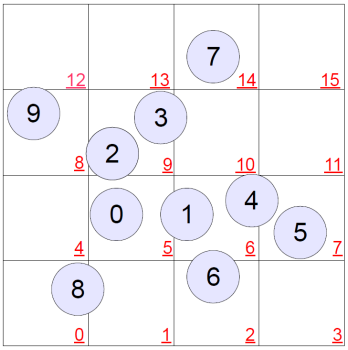

Figure 3.3: Sample 4x4 scene grid with indices of cells and characters

In the simulation environment, our characters are only concerned about the entities within a constant radius. In a regular grid structure with a known cell edge length we are able to define which cells of the grid are in our collidable area. We can define a minimum cell size of 'agent_radius x 2' to ensure that the agent can only intersect with the agents in the neighboring cells. Then all we need to do is search through the entities inside those cells and perform the collision detection operation. The grid data structure is re-generated in each time step but it is possible to perform incremental updates according to the agent's speed.

Let's examine the scene with 16 grid cells shown in the Figure 3.3. The algorithm first calculates every character's grid hash index value which is a single unsigned integer which is used as an index to the grid cells. For this implementation we have used z-axis as the world up-vector so spatial grid is laid out on the xy plane. Table 3.2 shows the results of the hash indices found by using characters' position.xy. Note that, modulo operations are optimized when the grid size is power of two and this linear operation is done very effectively with the parallel computation.

After determining which character is assigned to which cell, every character owns a hash index value. We just sort the cell indices and the corresponding character index array to get a usable list of character indices for collision detection (Table 3.3). Bitonic sorting [51] is used to provide parallel execution on the GPU.

In order to get a better memory coherence in the collision checks for a known cell, kernel should be able to access the character indices that are assigned to that grid cell and its neighbor cells. We can iterate from the start to the end of the sorted array and find the begin and end indices of the sorted cells, as shown in the Table 3.4.

Table 3.2: Calculated hash values of the characters

Array Index	Character Index	Grid Hash
0	0	5
1	1	6
2	2	9
3	3	9
4	4	6
5	5	7
6	6	2
7	7	14
8	8	0
9	9	8

Table 3.3: Sorted character indices

Array Index	Sorted Grid Hash	Sorted Character Index
0	0	8
1	2	6
2	5	0
3	6	1
4	6	4
5	7	5
6	8	9
7	9	2
8	9	3
9	14	7

Table 3.4: Begin and end indices of cells

Array Index	Grid Hash	Particle Index	Cell Start	Cell End
0	0	8	0	1
1	2	6	-1	
2	5	0	1	2
3	6	1	-1	
4	6	4	-1	
5	7	5	2	3
6	8	9	3	5
7	9	2	5	6
8	9	3	6	7
9	14	7	7	9
10			-1	
11			-1	
12			-1	
13			-1	
14			9	10
15			-1	

Since we are using texture buffers to store our base and simulation data for agents, using interoperability features of OpenCL rather than creating separate memory buffers, reading from and writing to them every frame; it is faster to synchronize OpenCL objects. This gives high performance improvements. Details of OpenCL kernels execution and OpenGL interoperability are beyond the scope of this chapter, but it's worth noting that for the interoperability, buffer objects should be arranged carefully to remove non-necessary data transfers. Restricting the kernel-output buffers will improve the performance by reducing the objects that requires synchronizing between OpenCL and OpenGL. Our OpenCL simulation step updates only the position information and we only require to synchronize the base buffer after kernel executions.

CHAPTER 4

PROPOSED METHOD

Using traditional character rendering methods to render thousands of characters in an interactive application might result in poor performance due to the expense of making lots of draw calls and memory transfers. In this section we present our proposed method to render high numbers of animated characters by using the features of latest graphics processing units. We describe a GPU-friendly way to render, animate, cull and control level of details of the randomized characters as well as explaining the visibility grid method for efficient crowd rendering with shadows.

Proposed grid-based visibility mask creation method is irrelevant to scene complexity. We begin by constructing a mip-mapped hierarchical-z buffer by rendering the depth of all the occluder geometry in the scene to be able to test occlusion of any given object or bounding volume according to eye view. This depth rendering pass is common in many of the today's rendering engines especially for the ones using deferred shading pipeline.

We are using spatial grid structure to define visible areas in the scene. Each cell in the visibility grid is computed according to the minimum and maximum walkable height values of the corresponding 3D area. The minimum and maximum heights of walkable areas of a scene is commonly static and that data could be pre-computed for many scenarios or updated iteratively. We perform frustum and occlusion culling tests for each area volume element and fill the visibility mask texture. Building a quad tree structure to handle the area visibility queries faster could be achieved by creating the mip-maps of the visibility mask texture. To make crowd instance culling faster, we add padding to the volumes for the visibility checks, according to the agent radius. This way we are able to check if a crowd instance is visible from eye view with a single query to the regarding cell rather than checking the neighbor cells

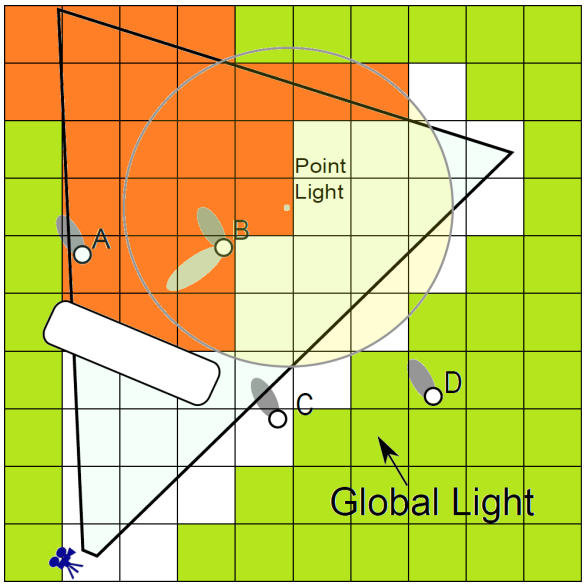

Figure 4.1: Grid based visibility mask. Green cells are culled with frustum check and orange cells are not visible due to occlusion. Visible shadow caster agents are determined with the same visibility mask. In this scenario, only Agent C is needs to be rendered for shadow mapping.

separately. For shadow maps, the point presentation becomes a line segment along the light direction and multiple cells' visibility needs to be checked according to the light's position and attenuation to check occlusion culling of casted shadow volume correctly. The visibility grid also enables us to perform shadow focusing according to the visible cells that is similar to previous work of Lauritzen et al [30]. The visibility-aware distribution analysis becomes much faster as it only needs to check the samples in the visibility mask which are far less than the count of pixels on the screen.

Visibility grid accelerates view space rendering by becoming an early-out mechanism of per-agent visibility queries and preventing unnecessary frustum and hi-z checks with a single texture fetch. In the scenario of Figure 4.1 Agents A and B are culled by checking the visibility of their cell from the visibility mask. Details and performance results of the culling and LOD determination methods for view space rendering is given in the results chapter.

4.1 Culling with the Hierarchical-Z map

In a typical game scene, there can be many objects occluded by walls, buildings and other objects. Distinctively from the frustum culling, occlusion culling depends on the scene hierarchy

and occlusion determination of an object requires one or more occluder checks.

Regular hardware support of occlusion queries is being supported for several generations of the GPUs. One can query the rendered pixel count with the graphics APIs. But having a read back from GPU to CPU causes performance stalls. In recent updates to modern graphics libraries such as OpenGL and DirectX, performing predictive rendering which could avoid the read-backs but the graphics processing unit would still require a draw call and perform the vertex transformations with a vertex shader.

Figure 4.2: Illustration of the 3 mip levels of hierarchical-z buffer

While the hardware accelerated queries for occlusion culling is useful, they are impractical to be used to cull thousands of characters one-by-one. Hierarchical-z mapping is a way to handle occlusion culling checks effectively without any dependence of pre-computations or manual occluder placing.

Construction of the hierarchical-z (Hi-Z) map performed at the beginning of each frame by rendering the depth values of all occluder entities. Many games use simplified occluder geometries for the objects to improve the hi-z map rendering. Maximum depth of the corresponding values are stored in the mip-maps of the hi-z map. So the final mip level with one pixel stores the maximum visible depth value of the scene. The construction of the hi-z map with whole mip-map chain takes about 0.25 miliseconds at HD resolution in our testing scenes. Figure 4.2 shows various mip levels of the hi-z map.

After constructed the Hi-Z map, we can use it for the occlusion tests of the agents and visibility grid cells. The occlusion check from the hi-z map requires the clip space bounding rectangle of the object to be tested. Clip space bounding rectangle of a world space bounding box can be found by transforming all the 8 vertices of the box into the clip space and performing a min/max search. Same bounding box corners that we have already calculated for frustum

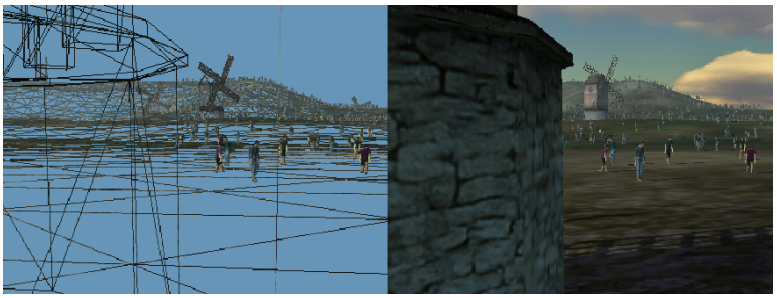

Figure 4.3: Agent behind the windmill is not visible, but still cast shadow on the ground

culling can be used to avoid multiple clip space conversion. Corner coordinates of bounding rectangle is used to calculate texture coordinates and the size of the bounding box determines the mip-level of the hi-z map to be used for the occlusion test.

4.2 Effective Shadowmap Rendering

Shadowmap rendering of complex scenes become very slow due to the vertex processing cost. Pixel shader stage of shadowmapping is not costly as it only outputs the depth of the current fragment, a single pixel could be rendered more than a hundred times due to the rendering of a small mesh to a shadowmap which is a common case for global lighting scenarios. Vertex processing cost of skinned instanced meshes are much more higher than regular meshes because of the requirement of texture fetches and skinning. We need to avoid rendering unnecessary characters to shadowmap for better rendering performance.

One of the most common test when rendering the shadowmaps is to make frustum culling according to the light view. This optimization would reject many of the characters if we are using a tight light camera frustum which is generated according to the main camera view. Occlusion culling of characters from the light view enables us to not render the characters that are fully in shadow but is not effective when the depth complexity of the shadowmap rendering is not so high.

Shadow caster culling algorithms focus on the occlusion of casted shadows within the main camera view for better performance. Figure 4.3 shows a scenario where an agent is culled by the occlusion culling for main view rendering but still casts shadow to the ground. Our method creates a visibility grid which is not relevant to the scene complexity, light source count and

crowd density. Once we got the visibility mask ready we perform a visibility search on the line segment of the shadow volume casted by the given agent. For global directional lights method would require multiple cell visibility checks on the mask and the local lights limits the number of cells to be checked by their attenuation factor.

The technique does not require any stencil buffer to cull the characters with hardware accelerated masking and avoids the render calls totally by testing the visibility of character shadows first. This improves the vertex processing performance which is a posible bottleneck of crowd the rendering stage.

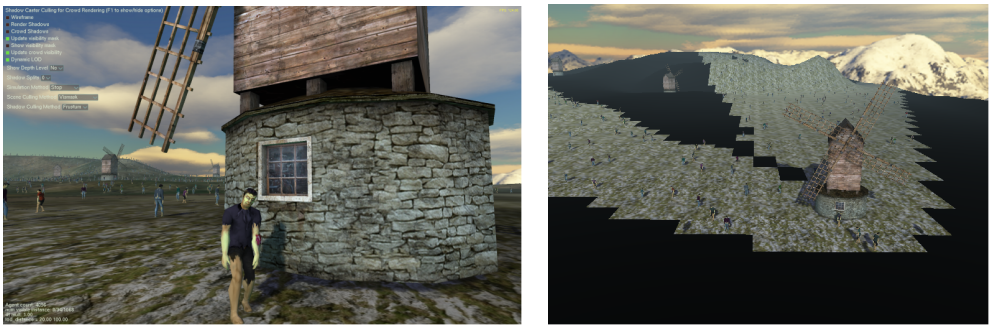

Figure 4.4: Visibility grid visualized.

The visibility grid we have generated can be used from any view angles independently from any light source position. Figure 4.4 visualizes the visibility grid. Grid is re-used for each shadowmap pass and re-calculation of a shadow mask is prevented. This provides us better performance when using multiple light sources in a crowded scene. Even for a single global light source, partitioning algorithms lead to render multiple shadowmaps for better resolution utilization.

4.3 Deferred Shadowing Calculation

Deferred shading methods are being used to avoid pixel overdraw costs. Shading computations are postponed to a final stage and only the visible pixels are shaded. Since standard shadowmapping results are binary, smoothing the results require multi sample computation and generating soft shadows is a time consuming job. In our method we implemented deferred cascaded shadow mapping to avoid unnecessary high cost shadowing computations.

Figure 4.5: Left: 1 cascade 2048x2048 Shadowmap, Right: 4 cascades of 1024x1024 shadowmaps. Better shadowing quality achieved while same amount of texture memory being used.

4.4 Global Directional Lighting

In computer games, one of the most common way to simulate the global lighting from sun or moon is using directional lighting. Sun is an omni-directional light source but since the light source is very far the effect of the minor direction changes could be ignored when rendering small and mid-scale scenes. All entities in the scene will receive the same light from the same direction by using directional lights.

Global lighting of a large scene would require some kind of level of detailing techniques for shadowmapping. A set of shadowmaps are used widely to increase the shadowmap detail around the regions that are near to the camera. Figure 4.5 shows how using cascaded shadow maps improves the rendering quality while keeping amount of memory used the same. Rendering multiple shadowmaps in each frame makes our technique more effective than other shadow caster culling methods which needs different shadowmap masks for each cascade.

Directional lights have color and direction properties and the shading does not depend on the position of the light source. Our algorithm culls the agents in parallel and each kernel computes an agent's visibility. First the cells are calculated which can get any shadowing from the corresponding agent using a method similar to ray marching along the visibility grid by being aware of the cell's minimum and maximum heights. Then each cell's visibility is checked to figure out if the agent can cast any visible shadow on the screen. If all of the cells are not visible then the agent is culled. Global directional lights are commonly used with

zero attenuation, which could enforce the algorithm to check many cells for an agent along the volume of it's shadow, especially in lighting scenarios such as dusk and dawn. So we limit the cells to be checked and apply fading out along the distance to the shadows in the pixel shader. The positive side of such lighting scenarios is the size of the scene gets smaller from the lights point of view and the depth complexity gets higher, so the shadowing quality increases with same sized shadow maps and we get a chance to use occlusion tests effectively from lights point of view.

4.5 Local Lighting

While global directional lights are the main source of the many of the outdoor game scenes, especially the interior rendering requires local lighting. Local lights are effective on limited area and creates a shading contrast according to its intensity. Local lighting changes and shadows are detected easier due to this contrast.

Depending on the light type, scene can be illuminated according the light's position and attenuation factors. Attenuation controls the light's intensity change over the distance. Different methods can be used to simulate complex attenuation effects.

There are two basic light types that are usually used in video games; point lights and spot lights. In the following sections, we cover how our shadow caster culling method works for these kind of lights.

4.5.1 Point Lights

Point lights simulate light radiating out from a point in space. A light bulb can be thought as a point light but in real life, its unlikely to find any uniform point light with infinitely small size. Point lights are omni-directional, meaning that they emit light towards every direction around them.

Rendering shadowmaps for a pointlight requires the rendering of the illuminated geometry again. Common method is to render the scene for each side of a cubic shadowmap. With the dual paraboloid mapping method one can reduce the draw call count by rendering to two render targets rather than six but introduce some rendering artifacts for low polygon meshes.

Any method would require to perform a culling to avoid rendering unnecessary meshes to the shadowmap.

We first check the lights visibility according to the Hi-z map and if its not visible at all, we do not update any shadowmap for that light. If the light is visible the agents around it needs to be culled according to the occlusion of their shadow volume. A scene hierarchy like quad-tress could be used to speed up the shadow caster culling with faster radius checks while rendering the shadowmaps by reducing the agents to be processed. Our culling program runs in parallel on the GPU for all agents and first checks if the corresponding agent is inside the range of the point light. Then we apply similar approach to directional lights and find the shadowing cells from the visibility grid. The main difference of shadow caster culling method for point lights is that, they have a constant range of illumination and the cell searching is interrupted according to the attenuation of the light.

4.5.2 Spot Lights

Spotlights are very similar to point lights but they have a direction that can be controlled usefully to aim the light at a particular target. There are three fundamental elements to form a spotlight: direction, cutoff angle, and the attenuation factor.

Spot lights limit the illumination of the scene with a cone along its direction. Similar to point lights, spot light also have attenuation factor. From the shadowmapping side of view, spot lights are easier to manage since only a single texture is enough to render the shadowmap of the scene from the lights position towards its direction. Single shadowmap generation cost makes spot lights more suitable for many cases. Our method can be used in similar way like point lights rendering. Care must be taken on the frustum of the camera that is used to render shadowmaps and it should be adjusted correctly to cull the agents that are not going to be shaded due to the cutoff angle.

CHAPTER 5

IMPLEMENTATION AND RESULTS

In this chapter we give details about the implementation specific optimizations and the results of the algorithm in different cases. First we define our GPU-based visibility culling and LOD selection implementations and explain our crowd simulation technique to show that the technique is suitable with GPU based collision detection algorithms that can be used in interactive game scenarios.

Here is the configuration of the machine that we have used for profiling the performance of the method:

- Processor: AMD Phenom II x4 965 3.40 GHz
- RAM: 4 GB DDR3
- GPU: NVIDIA GeForce GTX 460

  Memory: 1GB DDR5

  Graphics Clock: 675 MHz

  Cores: 336
- OS: 64-bit Windows 7
- Resolution: 1920x1200

5.1 Level of Detail Determination and Visibility Culling

Video games require real time reactive gameplay for better interactive user experience. Real time applications should display at least 30 frames per seconds to keep interactivity smoothly.

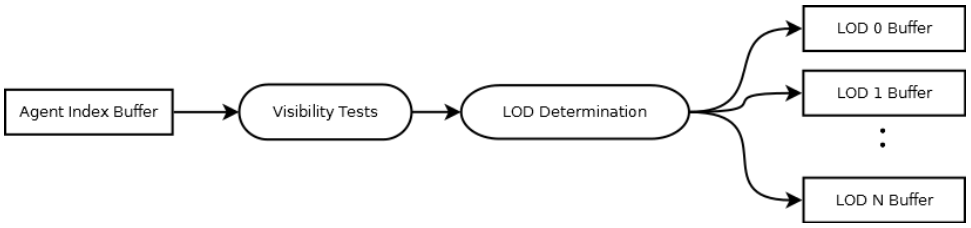

Figure 5.1: GPU-based Culling and LOD determination pipeline.

Therefore, a game only have maximum 1/30 seconds to apply user interactions, integrate, simulate and render the whole crowded scene per frame. Using data and task parallel computation and avoiding data traffic between the CPU and the GPU in such an environment is crucial to provide better performance. Consequently, we use GPU-based algorithms for crowd rendering and simulation. Our method is using vertex and geometry shaders for visibility and LOD computations with multiple stream output feature of modern graphics hardware. Method fills different index buffers for each detail level and use instanced draw calls at each level of detail and does not read backs the results by directly rendering from the output data. The level of detail system with selective readout is similar to append buffers used in compute shaders. The draw calls are divided into number of LODs which also enables us to set different materials for level of detailing the shading complexity by binding different shaders to the near and far meshes for performance. This way we can disable heavy specular shading or normalmapping calculations for distant LOD meshes. Technique also provides the ability of using billboards for distant characters. Figure 5.1 shows the operation pipeline where the input is a vertex stream whose elements count is equal to number of agents in the scene. By making a draw call, each agent's visibility is tested in the vertex shader stage then the LOD determination outputs the visible agents to corresponding LOD buffer.

How many primitives are passed to which buffer can be queried by using API commands. Functionality is supported by the extension GL_ARB_transform_feedback3 in OpenGL, which has been made part of the 4.0 version. These asynchronous queries can be removed by using glDrawTransformFeedback() command for rendering simple primitives, but we need to use indexed queries since we require the visible instance count on the CPU side for instanced rendering and dynamic detail level distance adjustment to keep the performance similar on different visible crowd density near the camera. Multi-passing can be used on older hardware to achieve this detail level determination with single output at each pass. About 10 percent

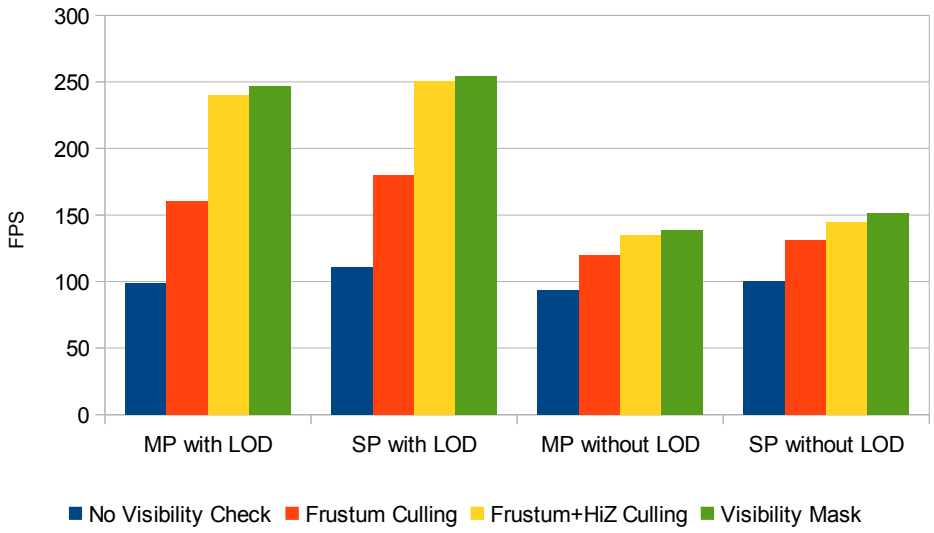

Figure 5.2: Performance of culling and level of detail mechanisms on a crowded scene without shadows. MP: Multi-passing method, SP: Single-pass method using multi-stream output.

performance improvement over single-pass method is measured in test scenarios with about 16k agents. Figure 5.2 shows performance improvement of using the LOD and culling mechanism with single output multi-passing and multi stream output with single-passing at various locations of a crowded scene with no shadowing.

5.2 Skinned Instancing with Animation Baking

Rendering multiple objects at the same time by using hardware accelerated instancing is supported as a core feature for many of the today's graphics processing units and APIs. For most cases, another data buffer is used to provide per-instance data, such as world space position, rotation or color to the GPU. For a skinned object, per-instance data might be the current bone frames of its skeleton but this will create a high bandwidth overload since we should update all the buffer every frame with new skin data including many matrix frames. Requirement to build a big per-instance data buffer may limit our maximum instance count that can be rendered. Also, simulating each instance animation on the CPU side might effect the application performance negatively.

The preferred solution is to move all the animation system to the GPU and perform parallel

integration in order to render large number of animated characters effectively. Such an animation system can be implemented by locating the bone animations on the GPU side by using floating point textures. In our method, all the animations of the crowd character are baked to a texture at a constant framerate and we just store current animation number and frame time as per-instance data. Note that, we have to apply skeleton's resting position transformations and write the bone transformations at model space to be able to skin the meshes correctly without making many parent bone frame multiplications. We can create local deformations such as bone scaling or inverse kinematics by providing bone hierarchy and the resting frames to the GPU and performing local transformations if required. A sample animation data structure is given in the Table 5.1.

Table 5.1: Animation data texture structure

	Bone0				..	BoneN			
	Row0	Row1	Row2	Row3	..	Row0	Row1	Row2	Row3
Frame0	xyzw								
Frame1									
Frame2									
.									
.									
.									
FrameN									

The chosen animation data structure helps to keep the animation texture width and height dimensions close to each other. For 256 animation frames with 32 bones, the texture data takes about 1MB with 32-bit floating point format. If the application requires more data for the animations, 16-bit floating point format might be considered. Quaternions could also be used to reduce animation texture size for large animation data sets, but since it will increase the instruction count of our skinning shader we have chosen to use rotation matrices directly. Skinning the mesh vertices in the vertex shader is not so different than classic skinning with uniform matrices; we just create the necessary bone matrices from the texture by making four texture look-ups to create a 4x4 matrix. Since the 'w' components of the vectors are known (0.0 for rotation vectors and 1.0 for position vector) it is also possible to pack a matrix data to 3 texels.

If the application requires those animated meshes more than once every frame, like for shadowmap rendering, it might become effective to skin all the instances in the vertex shader and

Figure 5.3: Terrain environment does not contain high occlusion culling opportunity with a wide open area.

output them to a separate vertex stream to avoid re-skinning the characters. This is suggested way to render crowds in the work of Vykruta [52]. It directly uses same skinned vertices data and reduces texture fetch count and the skinning cost. But in such an implementation, the visibility of the crowd should be computed according to union of all frustums of possible renderings so that same output vertices can be used in all shadowmapping render passes. This would create huge overload for the scenes with many local lights and cause to render all the characters at each frame.

5.3 Open Terrain Environment Scenario

In this study we apply the proposed method to an interactive game scenario and perform performance benchmarks from various locations of the scene. Different culling techniques and different global shadow cascade counts are used to test the adaptability of the method. Figure 5.3 shows the typical view of the game.

Terrain structure that we are using is not so much mountainous which provides a natural occlusion of a similar game. We have a zombie crowd over the terrain and they attack the windmill entities. We can interact with the zombies by moving the camera near them and them will die.

This section will also cover the details of the scenario and the performance results. We plan to benchmark another scenario within an urban environment, where the pedestrians are travelling around. We estimate a better performance on urban environment due to the higher depth

complexity.

5.4 Effects of Agent Count and Shadowmap Size

Shadowmap rendering performance directly depends on the shadow maps size. Rendering to very large textures also slows down runtime shadowmap sampling performance due to limited cache memory of the GPU. Here we present our results of the method with different shadowmap sizes and character count. Rendering performance is evaluated while the crowd simulation and collision detection is disabled.

Table 5.2: Average framerates for 4096 Agents, 1024x1024 Shadow maps

Cascade Count:	0	1	2	3	4
No shadow culling	105	69	56	47	40.7
Frustum Culling	105	70	65	56	50.5
Visibility Grid	105	85	76	70	62.1

Table 5.2 shows the average performance of our method on open terrain scene with different numbers of cascades. While frustum culling could achieve 20 percent speedups, visibility grid system provides more than 50 percent improvement in framerate.

Table 5.3: Average framerates for 65536 Agents, 1024x1024 Shadow maps

Cascade Count:	0	1	2	3	4
No shadow culling	45	16	10	7.2	5.6
Frustum Culling	45	22	20	18.3	16.4
Visibility Grid	45	35	32	29.4	26.2

Increasing the number of agents to be rendered increases the importance of having a good culling scheme. Table 5.3 presents visibility grid could make 6 times better than rendering without any culling and again much better than standard frustum-only culling.

Table 5.4: Average framerates for 65536 Agents, 4096x4096 Shadow maps

Cascade Count:	0	1	2	3	4
No shadow culling	44	14.1	7.9	5.6	4.2
Frustum Culling	44	16.8	12.5	9.8	7.9
Visibility Grid	44	25.5	17.5	13.2	10.3

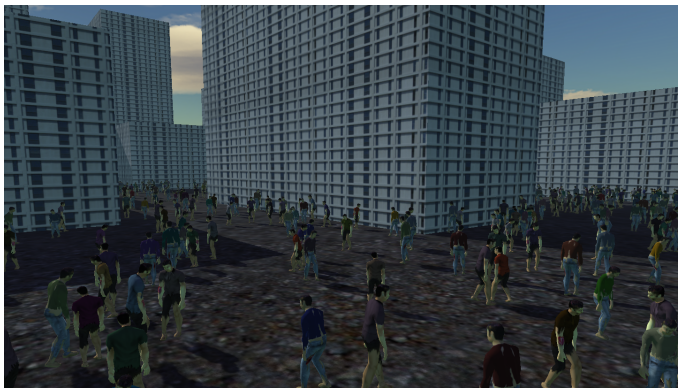

Figure 5.4: Urban scene with limited view range.

Shadowmap size improves the visual quality of the application. But having big render targets for shadowmaps causes performance drops due to large number of pixel fillrate cost and cache misses on shadow sample fetching in the GPU. Results in table 5.4 shows an important result for performance test of our method. As you can see, there is no such a big difference between rendering and culling the scene 4 times with 1024x1024 shadowmaps than rendering the scene 1 time with 4096x4096 shadowmap. While filling the same texture memory, shows that there is no big impact of the culling algorithm on the framerate.

5.5 Urban Environment Scenario

Occlusion effect is much more effective on urban environments where buildings limits the view range. Depth complexity of shadow mapping is also becomes higher and doing Hi-z culling for shadowmap rendering becomes feasible according to the global lighting direction. Objects that are fully in shadow can be culled with occlusion tests from the camera point of view. Figure 5.4 shows the typical view of a urban environment with crowd instances walking around.

Performance measurements for urban scene scenario can be viewed in tables 5.5 and 5.6. Results show that, even the additional rendering cost for the urban environment, the buildings, methods starts to be effective when there are many agents to be culled by occlusion.

Table 5.5: Urban Scene, Average framerates for 4096 Agents, 1024x1024 Shadow maps

Cascade Count:	0	1	2	3	4
No shadow culling	147	92	65	51	44
Frustum Culling	147	93	72	59	52
Visibility Grid	147	131	104	88	79

Table 5.6: Urban Scene, Average framerates for 65536 Agents, 1024x1024 Shadow maps

Cascade Count:	0	1	2	3	4
No shadow culling	108	21	11.3	7.6	5.7
Frustum Culling	108	32	26.2	20.6	17.1
Visibility Grid	108	94	72	58	52

Figure 5.5: Local lights with limited range can be used to simulate street lights.

5.5.1 Local Lighting Performance

Local lighting elements have limited range of illumination. Visible cell search of our method becomes much more effective for these kind of lights. We first perform frustum and occlusion checks for the sphere of the local lights to ensure that if they are visible on the screen. Visible point light's shadowmap rendering consist of six render calls and spotlights have one render call. There are also some other GPU-based implementations for point lights to avoid calling so many draw commands but they are out of the scope of this thesis. For better performance comparison we used spotlights which are widely used in games and other graphical applications.

Table 5.7 shows the average framerates achieved with the proposed method and the standard frustum culling method while each local light rendering around 0-50 agents. Spotlight shadowmap size is 1024x1024 and updated every frame.

Table 5.7: Average framerates for Spotlighted Urban Scene with 65536 Agents

Spotlight Count:	1	2	5	10	20
No shadow culling	21	11	4.7	2.43	1.2
Frustum Culling	86	74	51	32	20.1
Visibility Grid	88	77	57	40	27

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The aim of the thesis has been to improve the shadow mapping performance of the crowded scenes in video games by culling the crowd instances according to visibility of their casted shadow. We gave the background of the techniques that are used to perform the method and explained the related work about the research area to clarify the technique. Recent development in GPU-based technologies and the performance cost of data transfers between GPU and CPU lead us to perform all the computation in GPU, including the culling system and the crowd simulation.

We have described the related researches and implementations related to the subject including hierarchical-z occlusion culling which use a mip-mapped texture to perform culling operations on the GPU. Alternative methods could use hardware accelerated occlusion queries but they are very slow due to the CPU readback. Usage of predicated rendering by keeping the occlusion result on the GPU commonly depends on pre-rendering of the meshes and becomes inefficient for a vertex shader computation bounded crowd rendering. We have showed the results of the proposed method and its usability in a typical game scenario with user interaction.

6.1 Contributions

We proposed an algorithm to accelerate shadow mapping performance by using view dependent occlusion culling operations on shadow caster. Method updates a visibility grid data structure at the beginning of each frame render, then use the area visibility results to determine agents' shadow visibility on screen. Visibility of all character instances in the crowd are

computed in parallel without having a requirement of a stencil buffer or light-space shadow mask.

Another contribution of the thesis is a complete crowd simulation system with agent-agent interactions by using OpenCL. We performed finite state machine system and implemented a grid-based collision check optimizer for agents' collision avoidance. We demonstrated how we can reduce GPU-CPU bandwidth usage by baking animations to textures and compressing the data of the crowd instances.

Grouping visibility queries for crowd instances with spatial grid cells also gives better frame rate for the view space rendering without any shadowing computation. The method gets more effective in each shadow map rendering pass by re-using the same visibility mask on the shadow caster culling stage. The technique is very well compatible with the modern crowd rendering techniques such as skinned instancing, dynamic LOD determination and GPU-based simulation.

Our method removes the creation and usage of a screen or shadow space mask requirement. The visibility grid is computed in world space and reused for each shadow pass. This improves the performance especially when there are lots of shadow mapping passes for global lighting cascades or local lights. We have shown performance of the method on an open terrain scenario where a global directional light is effective, and on urban environment which additionally have some local lights.

6.2 Future Work

For the future researches related to the subject of the thesis we recommend to perform perception based analysis and optimizations for shadow mapping of crowded scenes where shadowed pixels gets less noticeable due to the distance, local lighting and light intensity. Recent work of Jarabo et al. [36] researches how the global illumination approximations effect the final image perceptually. Their work only includes the low-frequency lighting considerations and admits that its hard to approximate the high frequency shadows without any noticeable artifacts. But having such a technique to be used in aggregated environments could decrease the rendering time.

Another practice that we have noticed during the study is the possibility of using an approach similar to impostor method when sampling the shadow maps. Rather than rendering the crowd instances to shadow map textures, one can use a deferred approach and perform a ray trace to the depth impostors of the agents along the ray's path. Then we could use a precomputed shadow map of any crowd instance and even use distance field shadows effectively for soft shadows. Such a technique would change the shader caster culling methods since the calculations will always be according to a visible pixel.

Furthermore, the technique depends on characters with similar sizes. This effects the search space calculation for the shadow casted area visibility checks. But the method can be extended to support heterogeneous set of objects and previously computed visibility sets. For this kind of implementations, usage of compute libraries could perform better than native shaders by not changing the graphics render states and having ability to use different data structures.

REFERENCES

[1] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[2] Jiří Bittner, Oliver Mattausch, Ari Silvennoinen, and Michael Wimmer. Shadow caster culling for efficient shadow mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2011*, pages 81–88. ACM SIGGRAPH, ACM, February 2011.

[3] Rachel McDonnell, Michéal Larkin, Simon Dobbyn, Steven Collins, and Carol O'Sullivan. Clone attack! Perception of crowd variety. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, New York, NY, USA, 2008. ACM.

[4] Rachel McDonnell, Michéal Larkin, Benjamín Hernández, Isaac Rudomin, and Carol O'Sullivan. Eye-catching crowds: saliency based selective variation. *ACM Transactions on Graphics*, 28(3):1–10, 2009.

[5] Pablo de Heras Ciechomski, Sébastien Schertenleib, Jonathan Maïm, Damien Maupu, and Daniel Thalmann. Real-time shader rendering for crowds in virtual heritage. In Mark Mudge, Nick Ryan, and Roberto Scopigno, editors, *VAST*, pages 91–98. Eurographics Association, 2005.

[6] Jonathan Maim, Barbara Yersin, Julien Pettr6eacute;, and Daniel Thalmann. Yaq: An architecture for real-time navigation and rendering of varied crowds. *IEEE Computer Graphics and Applications*, 29(4):44–53, 2009.

[7] Pablo De Heras Ciechomski, Sébastien Schertenleib, Jonathan Maïm, and Daniel Thalmann. D.: Reviving the roman odeon of aphrodisias: Dynamic animation and variety control of crowds in virtual heritage. In *In Proc. 11th International Conference on Virtual Systems and Multimedia (VSMM 05*, pages 601–610, 2005.

[8] C. O'Sullivan, J. Cassell, H. Vilhjálmsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang. Levels of detail for crowds and groups. *Computer Graphics Forum*, 21(4):733–741, 2002.

[9] Franco Tecchia, Céline Loscos, and Yiorgos Chrysanthou. Image-based crowd rendering. *IEEE Computer Graphics and Applications*, 22:36–43, 2002.

[10] Simon Dobbyn, John Hamill, Keith O'Conor, and Carol O'Sullivan. Geopostors: A real-time geometry / impostor crowd rendering system. In *IN SI3D '05: PROCEEDINGS OF THE 2005 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES*, pages 95–102. ACM Press, 2005.

[11] Ladislav Kavan, Simon Dobbyn, Steven Collins, Jiří Žára, and Carol O'Sullivan. Polypostors: 2d polygonal impostors for 3d crowds. In *Proceedings of the 2008 symposium*

on Interactive 3D graphics and games, I3D '08, pages 149–155, New York, NY, USA, 2008. ACM.

[12] I. Rudomín and E. Millán. Point based rendering and displaced subdivision for interactive animation of crowds of clothed characters. *VRIPHYS 2004: Virtual Reality Interaction and Physical Simulation Workshop. Mexican Society of Computer Science, SMCC*, pages 139–148, 2004.

[13] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*. ACM, 2008.

[14] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, January 2000.

[15] Umbra Software. Umbra 3 rendering optimization middleware, 2012. This is an electronic document. Date of publication: January 16, 2012. Date retrieved: September 3, 2012. Date last modified: January 16, 2012, http://www.umbrasoftware.com/en/.

[16] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93. ACM, 1993.

[17] Hanan Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[18] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, July 2003.

[19] Mark J. Kilgard. Improving shadows and reflections via the stencil buffer, 1999. This is an electronic document. Date of publication: 1999. Date retrieved: September 3, 2012 from http://assassin.cs.rpi.edu/ cutler/classes/advancedgraphics/S11/papers/kilgard_stencil_buffer.pdf.

[20] Céline Loscos, Franco Tecchia, and Yiorgos Chrysanthou. Real-time shadows for animated crowds in virtual cities. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '01, pages 85–92, New York, NY, USA, 2001. ACM.

[21] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 242–248, New York, NY, USA, 1977. ACM.

[22] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Proceedings of the 14th Eurographics workshop on Rendering*, EGRW '03, pages 208–218, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[23] KyoungSu Oh and SunYong Park. Realtime hybrid shadow algorithm using shadow texture and shadow map. In *Proceedings of the 2007 international conference on Computational science and its applications - Volume Part I*, ICCSA'07, pages 972–980, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] Daniel Scherzer, Michael Wimmer, and Werner Purgathofer. A survey of real-time hard shadow mapping methods. *Computer Graphics Forum*, 30(1):169–186, 2011.

[25] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 557–562, New York, NY, USA, 2002. ACM.

[26] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In Alexander Keller and Henrik Wann Jensen, editors, *Proceedings of the 15th Eurographics Workshop on Rendering Techniques, Norköping, Sweden, June 21-23, 2004*, pages 143–152. Eurographics Association, 2004.

[27] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, pages 311–318, New York, NY, USA, 2006. ACM.

[28] Wolfgang Engel. Cascaded shadow maps. In *Shader X5: Advanced Rendering Techniques*, pages 197–206. Ed. Charles River Media, 2007.

[29] Shang Ma, Xiaohui Liang, Zhuo Yu, and Wei Ren. Light space cascaded shadow maps for large scale dynamic environments. In *Proceedings of the 2nd International Workshop on Motion in Games*, MIG '09, pages 243–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 97–102, New York, NY, USA, 2011. ACM.

[31] W. R. Mark G. S. Johnson and C. A. Burns. The irregular z-buffer and its application to shadow mapping. Technical report, 2004. TR-04-09, April 15.

[32] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics*, 24(4):1462–1482, October 2005.

[33] Zhao Dong and Baoguang Yang. Variance soft shadow mapping. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 18:1–18:1, New York, NY, USA, 2010. ACM.

[34] G. Ryder and A. M. Day. High Quality Shadows for Real-Time Crowds . In *EG Short Papers*, pages 37–41. Eurographics Association, 2006.

[35] Mel Slater, Martin Usoh, and Yiorgos Chrysanthou. The influence of dynamic shadows on presence in immersive virtual environments. In *Computer Science, editor, Virtual Environments '95*, pages 8–21. Springer, 1995.

[36] Adrian Jarabo, Tom Van Eyck, Veronica Sundstedt, Kavita Bala, Diego Gutierrez, and Carol O'Sullivan. Crowd light: Evaluating the perceived fidelity of illuminated dynamic scenes. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, 31(2pt4):565–574, May 2012.

[37] J. N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Computer Graphics*, 12(3):336–340, August 1978.

[38] Cuda compute unified device architecture - programming guide, 2012. This is an electronic document. Date of publication: July 4, 2012. Date retrieved: August 30, 2012. Date last modified: July 4, 2012.

[39] Falco Wockenfuss and Christoph Lürig. Introducing congestion avoidance into cuda based crowd simulation. In Jan Bender, Kenny Erleben, and Eric Galin, editors, *VRIPHYS*, pages 101–110. Eurographics Association, 2011.

[40] Erdal Yilmaz, Veysi Isler, and Yasemin Yardimci Çetin. The virtual marathon: Parallel computing supports crowd simulations. *IEEE Computer Graphics and Applications*, 29(4):26–33, 2009.

[41] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. March of the froblins: simulation and rendering massive crowds of intelligent and detailed creatures on gpu. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 52–101, New York, NY, USA, 2008. ACM.

[42] Erdal Yilmaz, Eray Molla, Cansin Yildiz, and Veysi Isler. Realistic modeling of spectator behavior for soccer videogames with cuda. *Computers and Graphics*, 35(6):1063–1069, 2011.

[43] Khronos OpenCL Working Group. The opencl specification, 2011. This is an electronic document. Date of publication: November 14, 2011. Date retrieved: August 28, 2012 from http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[44] Mike Bailey. Opengl compute shaders, 2012. This is an electronic document. Date of publication: August 11, 2012. Date retrieved: September 3, 2012 from education.siggraph.org/media/conference/S2012_Materials/ComputeShader_1pp.pdf.

[45] Erdal Yilmaz. *Massive Crowd Simulation with Parallel Processing*. PhD thesis, Graduate School of Informatics of the Middle East Technical University, 2010.

[46] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. ACM, 1987.

[47] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Phys. Rev. E*, 51(5):4282–4286, May 1995.

[48] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008.

[49] Takahiro Harada. Real time rigid body simulation on gpus. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley Professional, August 2007.

[50] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–11. ACM, 2008.

[51] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the 1968 spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, April 1968. ACM.

[52] Tomas Vykruta. Building an uber-fast crowd renderer for your next xbox 360 engine. GDC '10. Game Developer Conference 2010, 2010. Date retrieved: September 3, 2012 from http://www.microsoft.com/en-us/download/details.aspx?id=27403.

ODTÜ
ENFORMATİK ENSTİTÜSÜ

YAZARIN

Soyadı : .....

Adı : .....

Bölümü : .....

TEZİN ADI (İngilizce) : .....

.....

.....

.....

.....

TEZİN TÜRÜ : Yüksek Lisans ..... Doktora .....

1) Tezimden fotokopi yapılmasına izin vermiyorum ☐

2) Tezimden dipnot gösterilmek şartıyla bir bölümünün fotokopisi alınabilir ☐

3) Kaynak gösterilmak şartıyla tezimin tamamının fotokopisi alınabilir ☐

Yazarın imzası ..... Tarih .....