GERÇEK ZAMANLI GÖRÜNTÜ OLUŞUMU İÇİN KULLANILAN GÖLGELEME ALGORİTMALARININ CUDA KÜTÜPHANESİ İLE GELİŞTİRİLMESİ

2014
YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ

MURAT KOCA

GERÇEK ZAMANLI GÖRÜNTÜ OLUŞUMU İÇİN KULLANILAN GÖLGELEME ALGORİTMALARININ CUDA KÜTÜPHANESİ İLE GELİŞTİRİLMESİ

Murat KOCA

Karabük Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalında
Yüksek Lisans Tezi
Olarak Hazırlanmıştır

KARABÜK
Ocak 2014

Murat KOCA tarafından hazırlanan "GERÇEK ZAMANLI GÖRÜNTÜ OLUŞUMU İÇİN KULLANILAN GÖLGELEME ALGORİTMALARININ CUDA KÜTÜPHANESİ İLE GELİŞTİRİLMESİ" başlıklı bu tezin Yüksek Lisans Tezi olarak uygun olduğunu onaylarım.

Prof. Dr. Abdullah ÇAVUŞOĞLU
Tez Danışmanı, Bilgisayar Mühendisliği Anabilim Dalı

Bu çalışma, jürimiz tarafından oy birliği ile Bilgisayar Mühendisliği Anabilim Dalında Yüksek Lisans tezi olarak kabul edilmiştir. 09/01/2014

Ünvanı, Adı SOYADI (Kurumu) — İmzası

Başkan : Yrd. Doç. Dr. Baha ŞEN (YBÜ)

Üye : Prof. Dr. Abdullah ÇAVUŞOĞLU (YBÜ)

Üye : Yrd. Doç. Dr. Şafak BAYIR (KBÜ)

..../..../2014

KBÜ Fen Bilimleri Enstitüsü Yönetim Kurulu, bu tez ile, Yüksek Lisans derecesini onamıştır.

Prof. Dr. Mustafa BOZ
Fen Bilimleri Enstitüsü Müdürü

"Bu tezdeki tüm bilgilerin akademik kurallara ve etik ilkelere uygun olarak elde edildiğini ve sunulduğunu; ayrıca bu kuralların ve ilkelerin gerektirdiği şekilde, bu çalışmadan kaynaklanmayan bütün atıfları yaptığımı beyan ederim."

Murat KOCA

ÖZET

Yüksek Lisans Tezi

GERÇEK ZAMANLI GÖRÜNTÜ OLUŞUMU İÇİN KULLANILAN GÖLGELEME ALGORİTMALARININ CUDA KÜTÜPHANESİ İLE GELİŞTİRİLMESİ

Murat KOCA

Karabük Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı:
Prof. Dr. Abdullah ÇAVUŞOĞLU
Ocak 2014, 84 sayfa

Gerçek zamanlı gölge oluşturma: bilgisayar grafiğinde hızlı ve hatasız görüntü elde etmek adına önemli bir kavramdır. Son zamanlarda bu konu üzerine birçok çalışma yapılmıştır. Ancak bu çalışmalar yapılırken bir takım zorluklar ile karşılaşılmıştır. Bu zorluklardan en önemlisi performans artırmak ve kaliteli gölge oluşturmaktır. Temel olarak bu konuda daha önce oluşturulmuş geometri tabanlı gölge eşleme algoritmaları kullanılmıştır. Günümüzde grafik işlem birimi gücünden yararlanılarak gölge eşleme algoritmasında paralel hesaplama mimarisine sahip CUDA Kütüphanesi kullanılarak performans ve kalite artışı sağlamaya çalışılmıştır. Üç boyutlu görüntü oluşturmada izlenilen görüntülerde gerçekçilik düzeyini ve performansı artırma üzerine yoğunlaşıp; saniye başına milyon sayıda üçgen gölgesi oluşturmaya çalışılmıştır.

Anahtar Sözcükler : CUDA, gölge eşleme algoritması, gerçek zamanlı görüntü oluşturma, grafik işlem birimi.

Bilim Kodu **:** 902. 1.014

ABSTRACT

M. Sc. Thesis

DEVELOPMENT OF SHADING ALGORITHMS USED FOR REAL-TIME IMAGE FORMATION WITH CUDA LIBRARY

Murat KOCA

Karabük University
Graduate School of Natural and Applied Sciences
Department of Computer Engineering

Thesis Advisor:
Prof. Dr. Abdullah ÇAVUŞOĞLU
January 2014, 84 pages

Real-time shadow formation: is an important concept in order to obtain fast and accurate images in computer graphics. Recently, several studies have been carried out on this topic. However, a number of difficulties have been encountered while these studies were being performed. The most important of these challenges was to increase performance and create a quality shadow. Basically, previously-created geometry-based shadow matching algorithms have been used in this issue. Today, performance and quality have been tried to be increased by using CUDA library which has the architecture of parallel computing in the shadow matching algorithm by utilizing graphic processing unit. It has been tried to create triangle shadow in million numbers per second by concentrating on increasing the level of realism and performance in images in creating three-dimensional images.

Key Word : CUDA, shadow mapping algorithm, real time rendering, GPU (Graphical Processing Unit).

Science Code : 902. 1.014

TEŞEKKÜR

Bu tez çalışmasının planlanmasında, araştırılmasında, yürütülmesinde ve oluşumunda ilgi ve desteğini esirgemeyen, engin bilgi ve tecrübelerinden yararlandığım, yönlendirme ve bilgilendirmeleriyle çalışmamı bilimsel temeller ışığında şekillendiren sayın hocam Prof. Dr. Abdullah ÇAVUŞOĞLU'na sonsuz teşekkürlerimi sunarım.

Desteği ve önerileri için, Yıldırım Beyazıt Üniversitesi Mühendislik Fakültesi öğretim üyesi Yrd. Doç. Dr. Baha ŞEN'e teşekkür ederim.

Sevgili aileme manevi desteklerini esirgemeden yanımda oldukları için tüm kalbimle teşekkür ederim.

Sevgili eşimin göstermiş olduğu destek ve sabrına tüm kalbimle teşekkür ederim.

Bu günleri görmemi, bu tezi yazmamı sağlayan zamanın ve mülkün sahibi olan yüce rabbim Allah (c.c.)'ha şükürler olsun.

İÇİNDEKİLER

Sayfa

Sayfa

Sayfa

ŞEKİLLER DİZİNİ

Sayfa

Sayfa

ÇİZELGELER DİZİNİ

Sayfa

SİMGELER VE KISALTMALAR DİZİNİ

SİMGELER

µs : mikrosaniye
FPS : frame per second
GB : gigabayt
KB : kilobayt
MB : megabayt
ms : milisaniye
TB : terabayt

KISALTMALAR

2D : two-dimensional
3D : three-dimensional
API : Application Programming Interface (uygulama programlama arayüzü)
BVH : Base Video Handler (temel video eğitici)
CPU : Central Processing Unit (merkezi işlem birimi)
CUDA : Compute Unified Device Architecture
CUDPP : CUDA Data Parallel Primitives Library
FBO : Frame Buffer Object
GFLOPS : Giga Floating Operations Per Second
GPGPU : General Purpose Computing on Graphical Processing Unit
GPU : Graphic Process Unit (görüntü işlem birimi)
SM : Streaming Multiprocessor

BÖLÜM 1

GİRİŞ

Işık kaynağından çıkan ışık bir nesneye çarptığında nesnenin aydınlanan yüzünün tersinde oluşan karanlığa gölge denmiştir. Noktasal bir ışık kaynağından çıkan ışınlar doğrusal bir yol izler. Kaynaktan çıkan ışınlar, önüne konulan ışık geçirmeyen nesnenin kapattığı alan kadarlık kısımdan geçmez. Cismin arkasında kalan ve ışık kaynağından çıkan hiçbir ışının ulaşamadığı bölge tam gölge olarak adlandırılır. Bir kısım ışınların ulaşabildiği bir kısmının da ulaşamadığı bölgede oluşan karanlık bölge ise yarı gölge olarak adlandırılır.

Bilgisayar grafiği, oyunseverlerin ve grafik uygulamalarına karşı gittikçe artan rağbet karşısında hem donanımsal hem de yazılım olarak sürekli gelişmektedir. Performanslarının yüksek olmasından dolayı daha çok kullanılmaya başlanan grafik kartları, günümüzde pahalı aygıtlar haline gelmiştir. Grafik kartlarındaki yüksek performanslar sayesinde merkezi işlem birimlerinin yükleri hafifletilmiştir. Gerçek zamanlı görüntü elde ederken birçok zorlukla karşılaşabilinir, bu zorluklardan en önemlisi birçok matematiksel hesaplamanın yapılması gerekliliğidir. Bunun yanında uzayda yer alan geometrik şekillerin karmaşık veri yapılarından dolayı grafik kartları performans ve kalitede çok yeterli değildir.

Donanımsal eksiklerin giderilmesi için yapılan iyileştirme çalışmalarının yanı sıra yazılım olarak da bilgisayar grafiği kendini sürekli geliştirmektir. 1977 yılında Crow tarafından gölgeleme algoritmaları ile yapılan çalışmalarda temel gölge algoritmalarından bahsedilmiştir. Bu algoritmaya göre gölgeyi oluşturan nesne ile gölgenin oluştuğu düzlem arasında kalan alan, gölgenin hacmi olarak belirtilmiş ve bu hacim içerisinde siyah olarak boyanan noktalar ise gölge noktaları olarak kabul edilmiştir [1].

1978 yılında da Williams tarafından gölge eşleme tekniği açıklandı. Bu algoritmaya göre, gözlemcinin pozisyonu ile ışık kaynağının birbirine uzaklıkları karşılaştırılarak noktanın siyaha boyanıp boyanamayacağına karar verilmiş; bu algoritmalara bu şekilde sürekli eklemeler yapılarak geliştirilmeye çalışılmıştır [2].

Bu tez çalışmasında, gerçek zamanlı görüntü elde etmenin önemli yöntemlerinden olan gölge eşleme algoritması incelenmiş ve gölge algoritmalarını NVIDIA CUDA teknikleriyle grafik işleyiciler kullanılarak hızlandırılmasına çalışılmıştır.

İkinci bölümde, bilgisayar grafiğinde gölgelemenin rolü, geleneksel gölge eşleme kavramları, resim karesi işlerken oluşan işleme hataları, üçgen poligonlarının pikselleştirilmesi ve CUDA üzerinde durulmuştur.

Üçüncü bölümde, bu çalışmada oluşturulan algoritmayla ilgili genel bir inceleme yapılarak OpenGL ve CUDA'nın kullanıldığı alanlar açıklanmıştır.

Dördüncü bölümde, bu çalışmadaki algoritmanın detayları incelenerek noktanın yeniden yapılandırılma süreçleri, bu süreçlerin hangi aşamalarında OpenGL ve CUDA'nın kullanıldığı izah edilmiştir. OpenGL ile oluşturulan nesnelerin CUDA ile işlenme süreçleri açıklanmış, bellek erişimlerinin hızı etkilemesi ve bu etkiyi minimal hale getirme yolları araştırılmıştır. Ayrıca bu etki ile ilgili yapılan performans sonuçları verilmiştir.

Beşinci bölümde performans artırıcı iki yöntemden bahsedilmiştir. Verilere erişimi hızlandırmak için uygulanabilecek yöntemler açıklanmıştır. Tamponda oluşturulacak kılavuz çizgilerinin veri yapıları izah edilmiştir. Noktaların tespitinde kullanılacak ve gereksiz hesaplamaları engelleyecek olan sınırlayıcı kutular açıklanmıştır. Sınırlayıcı kutuların veri yapılarındaki üçgen poligonlarının tespiti için geçerli üçgenleri ayrıştırma ve derleme sistematiği verilmiştir.

Çalışmanın altıncı bölümünde, sınırlayıcı kutulardaki üçgenlerin pikselleştirilmesi için uygulanacak iki yöntemden bahsedilmiştir. CUDA'daki iş parçacıklarının yarı kaydırma, kaydırma ve blokları detaylı olarak değerlendirilmiştir. CUDA

çekirdekleri farklı birimlerin yapılandırmaları ile daha sonra başlatılabilir, böylece üçgenleri pikselleştirmek için hız açısından en iyi birim yapılandırmaları yakalanmış olunur.

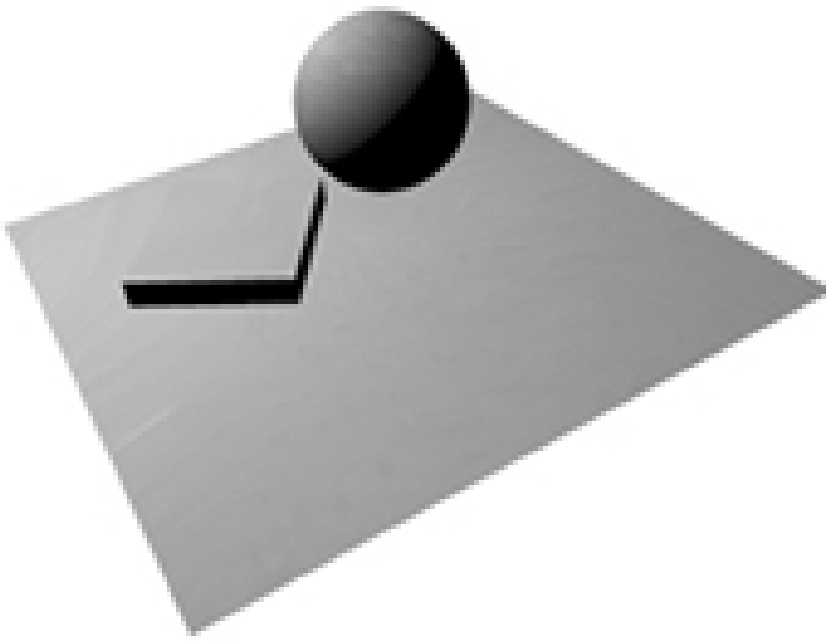
Nihai sonuçların açıklandığı yedinci ve son bölümde, CUDA çekirdeğinin yürütme hızının GPU'da aldığı süre NVIDIA Visual Profiler tarafından ölçülerek elde edilen bulgular ve uygulamalar yorumlanarak çalışma sonuçlandırılmıştır.

BÖLÜM 2

GERÇEK ZAMANLI GÖLGE ALGORİTMALARI

2.1. GÖLGENİN GÖRÜNÜMDEKİ ROLÜ

Gölgeler mekânsal ilişkiler ve verilerin algılanması için hayati görsel ipuçlarını barındırır. Genellikle nesneler arasındaki ilişkileri başka yollar ile tespit etmek zordur.

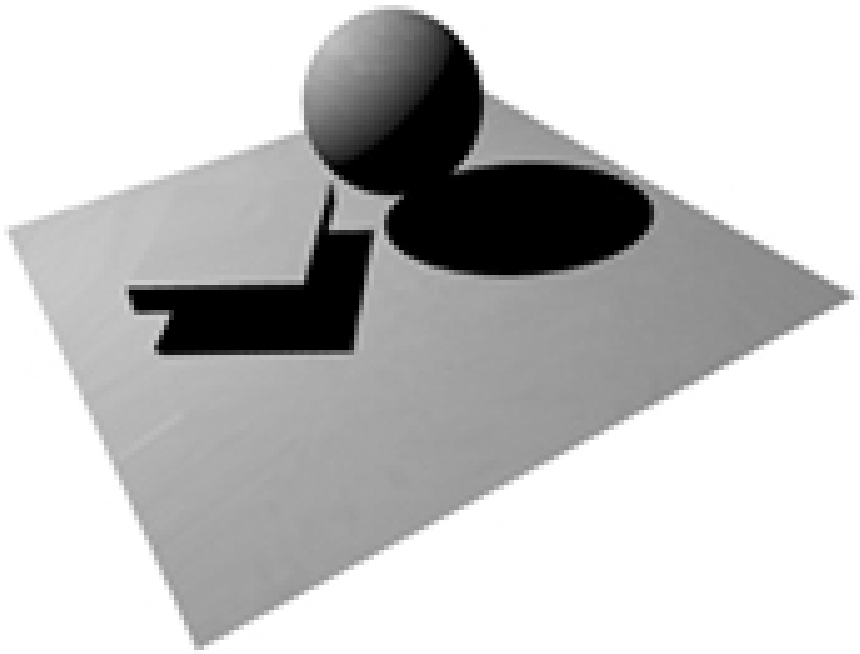


Şekil 2.1. Gölgesi işlenmemiş nesneler [3].

Şekil 2.1 incelendiğinde kendimize soracağımız sorulardan biri: "Bu nesnelerin zeminden ne kadar yüksekte olduğu?" olacaktır. Ancak buna verilecek cevap o kadar da kolay değildir. Bu nesnelerin zemin üzerinde nasıl konumladığı hakkında bilgi sahibi olunamamaktadır. Gölge bizlere aşağıdaki maddeler halinde sıralanan bilgileri sağlar:

1. Gölge, yüzeyin geometrik şeklinin anlaşılması için önemlidir. Gölge yardımıyla, gölgenin düştüğü yüzeyin kıvrımsal bir yüzey olup olmadığını görebiliriz.
2. Gölge, cisimlerin birbirleriyle konumsal ilişkisi ve cisimlerin boyutlarının anlaşılması için önemlidir. Örneğin, gölgeleme yapılmamış üç boyutlu bilgisayar görüntüsünde uzayda cismin pozisyonu hakkında bilgiye erişmek mümkün değildir. Gölgeleme sayesinde, görüldüğü gibi cismin yüzeye ne kadar yakın olduğu bilgisine ulaşılır.
3. Gölge, geometrik yapısı karmaşık olan şekillerin anlaşılması için önemlidir. Gölge yardımıyla, cismin gözükmeyen tarafının nasıl bir geometrik yapıya sahip olduğu bilgisine ulaşılır.
4. Gölge, göreceli olan nesnenin konumunu ve ışığın nereden geldiğini göstererek daha gerçekçi bir 3D sahnesini görmemize olanak sağlar.

Aşağıdaki görselde (Şekil 2.2) muhtemelen yukarıda cevabı bulanamayan sorulara daha net cevaplar verilebilir.



Şekil 2.2. Gölgesi işlenmiş nesneler [3].

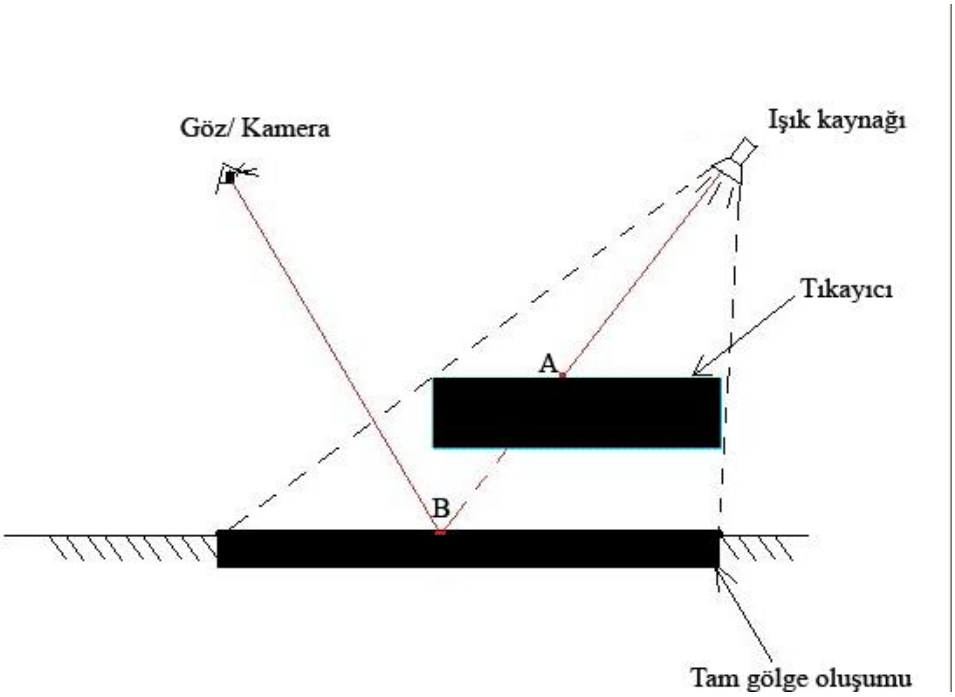
Bu sebepten, 3D görüntülerini işlerken görünümün önemi için gölgenin işlenmesi çok önemlidir. Bu tez çalışmasında, gerçek zamanlı gölge oluşumunun üzerine odaklanılmıştır.

2.2. GELENEKSEL GÖLGE EŞLEME YÖNTEMİ

Gölge eşlemeyi elde etmenin en yaygın tekniklerinden biri gerçek zamanlı işlemedir. Gölge Eşleme Algoritması ilk olarak 1978 yılında Williams tarafından tanıtıldı. Bu yaklaşım, çok basit ve iki geçişli resim işleme tekniğini gerektirir. Gölge eşleme yöntemi iki adımda uygulanır. İlk adımda, ışığın pozisyonuna göre sahnenin derinliği işlenerek ve bu derinlik tamponunun içine atılır. İkinci adımda ise, sahne ile gözün birbirine olan pozisyonu işlenerek gölge eşleme yöntemi ile karşılaştırılır. Gözün konumu işlenirken her bir noktanın ışık görüntü düzlemine izdüşümü gerekmektedir ve gölge eşlemelere bakmak için x ile y koordinatları kullanılır [1].

İlk adımda ışık kaynağının pozisyonuna göre manzaranın görüntüsü alınır. Işık kaynağına en yakın olan piksellerin derinlik değerleri gölge eşleme tampon alanında tutulur. Bu tampon alanı herhangi bir pikselin ışık kaynağına göre en yakın olan derinlik değerini tutar.

İkinci adımda ise gözlemcinin pozisyonuna göre manzaranın görüntüsü alınır. İlgili pikselin ışık kaynağına olan uzaklığı ile gölge eşleme tampon alanındaki derinlik değeri karşılaştırılır. Eğer ilgili pikselin ışık kaynağına olan uzaklığı gölge eşleme tampon alanındaki derinlik değerinden büyükse ilgili piksel gölgededir ve gölge rengine boyanır. Eğer ilgili pikselin ışık kaynağına olan uzaklığı gölge eşleme tampon alanındaki derinlik değerine eşit ve derinlik değerinden küçükse bu piksel aydınlıktır ve piksel kendi rengine boyanır.



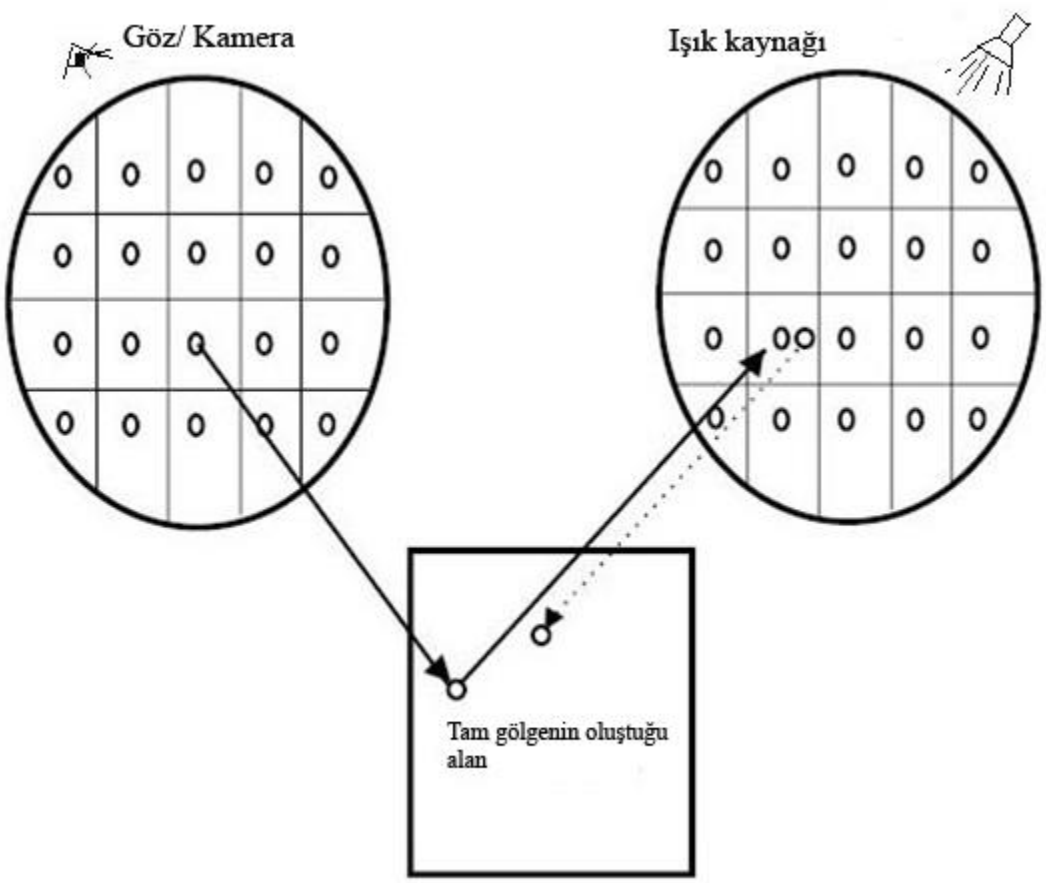
Şekil 2.3. Gölge eşleme tampon alanının kullanılması.

Şekil 2.3'te görüldüğü üzere, nesne üzerinde yer alan A pikselinin ışık kaynağına olan uzaklığı, gözlemcinin pozisyonu ile ışık kaynağı arasındaki uzaklıktan küçük ya da uzaklığa eşit olduğu için piksel aydınlık pikseldir. Düzlem üzerinde yer alan B pikselinin ışık kaynağına olan uzaklığı, gözlemcinin pozisyonuna göre ışık kaynağından daha uzakta olduğu için cisim gölgededir.

Böylece, gölge eşleme yöntemi ile noktanın derinliğinin karşılaştırılması, belirli bir noktada gölge olup olmadığını söylemek için bir çözüm sunar.

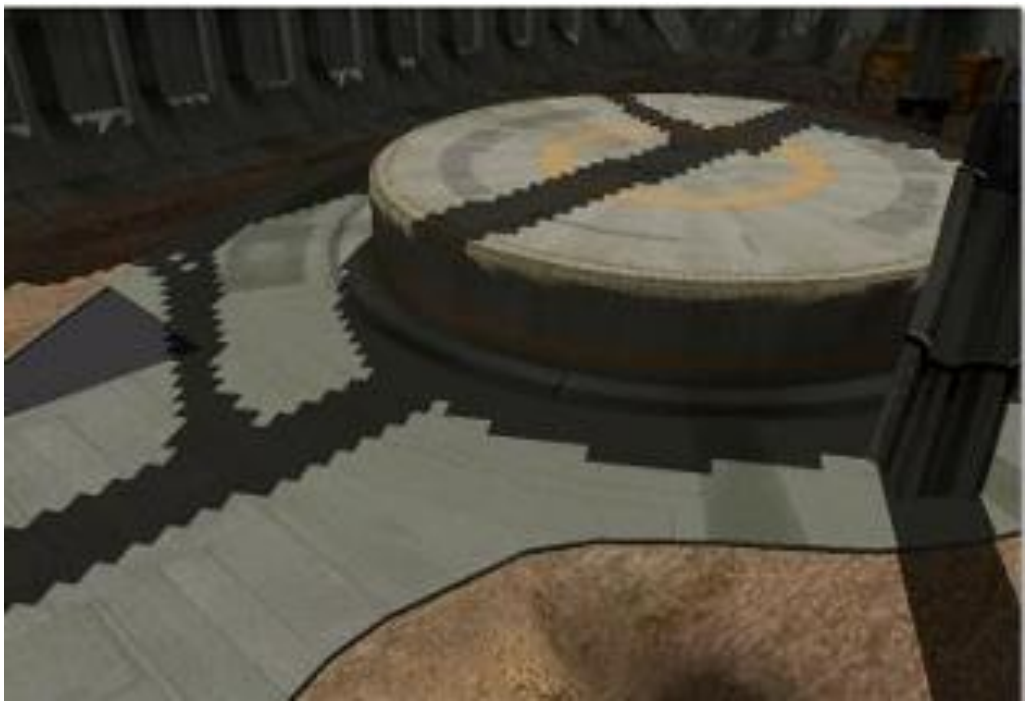
2.2.1. Geleneksel Gölge Eşlemede Oluşan İşlem Hataları

Şekil 2.4'te bir noktanın görüntüsünün düzlemindeki ışık görünümünün nasıl planlandığı ve bakılan yer ile ışık noktasını karşılaştırmak için derinlik değeri kullanılır. Kolayca karşılaştırmak istenilen derinliğin gerçek noktanın ışık konumu ile görülen piksel arasında uyum olup olmadığı kolaylıkla söylenebilir. Bu nedenle, derinlik değeri için yakın olan pikselin konumu tercih edilir.



Şekil 2.4. Gölge eşleme.

Tespit edilen örnek nokta ile gerçek noktanın pozisyonu arasındaki sapmadan doğan çözünürlük sorunlarına neden olmaktadır, bu işlem hataları sonraki şekil 2.5'te gösterilmiştir.



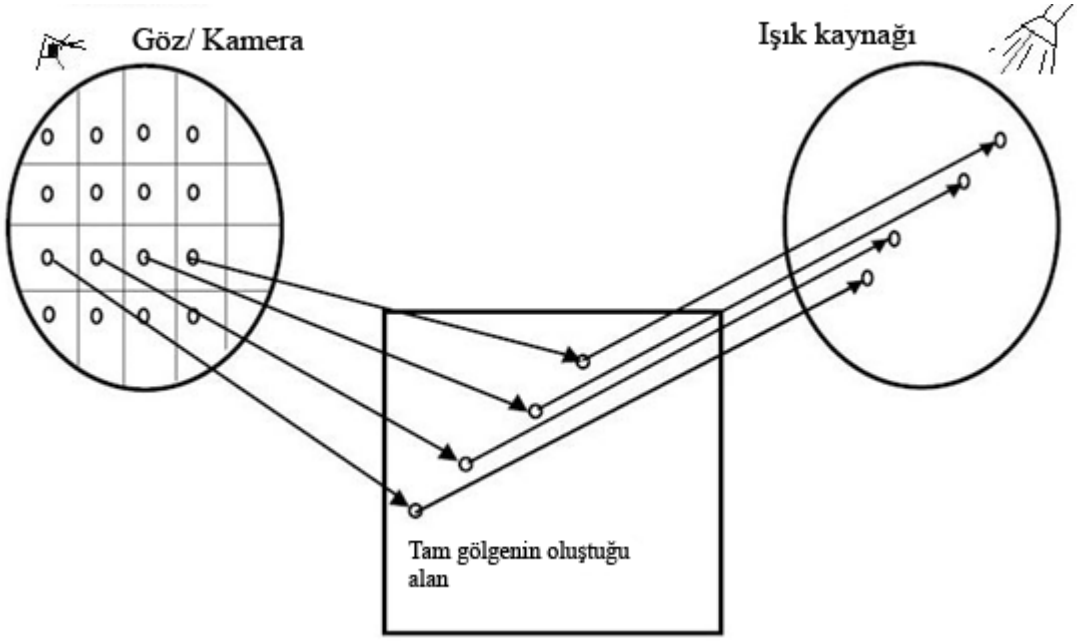
Şekil 2.5. Gölge eşleme sırasında oluşan işlem hataları [4].

2.3. EĞRİLERDEKİ GÖLGE EŞLEMENİN İŞLEM HATALARI

Başka bir yaklaşıma göre gölge eşlemedeki işlem hatalarını ortadan kaldırmak için gölge haritalarındaki çözünürlüğün artırılması gerekmektedir. Diğer yaklaşımlar ise şunlardır: 2002 yılında Stamminger ve Drettakis *Perspective Shadow Maps*, 2006 yılında Engel *Cascaded Shadow Maps* teorilerini geliştirmişlerdir. Ayrıca 2008 yılında Lloyd ve arkadaşları *Logarithmic Perspective Shadow Maps* yöntemleri ile gölge eşlemede oluşan işlem hataların ortadan kaldırılmasına çözüm aramışlardır [5-8].

İşlem hatalarını ortadan kaldırmak için, gözün gördüğü manzarada görülen her bir nokta için ışık kaynağında da doğru bir nokta bulmak gerekir. Gölge eşlemede işlem hatalarını ortadan kaldırmak için olası bir çözüm üretmenin yolu gözün gördüğü manzara içindeki her piksel noktasının ışık kaynağının gördüğü alandaki her piksel noktasına karşılık gelecek şekilde, 1:1 haritalama gibi noktalar birbirine karşılıklı gelmedir.

Şekil 2.5'de görüldüğü gibi görüntüde işlenmesi gereken ışık kaynağının aydınlattığı gölgenin oluştuğu alan içindeki piksel noktaları ve geometrik şekillere bağlı eğriler sapmaya uğrar. Bundan dolayı GPU üzerinde sabit fonksiyon birimi (piksel verilerini içine alan 3D uzayda, bir nokta verisine dönüştürme taramasını yapan) düzensiz noktaları işleme yeteneğine sahip olmadığından, düzensiz noktalar için yazılım geliştirme gereksinimi vardır.



Şekil 2.6. Düzensiz gölge eşleme.

Bu düzensiz noktaları işlem hatalarından arındırmanın gölge eşleme yönteminden bazı farklı yönleri vardır. İlk olarak sahne işlenirken göz pozisyonundan nesnenin tüm noktalarındaki renk, doku, vb. bilgiler ayıklanır. Örnek noktalar belirlenir ve bu örnek noktalar ışık kaynağının pozisyonundan sahneyi işlemek için kullanılır, böylece düzensiz gölge haritaları oluşturulur. Son adımda ise, gözün pozisyonuna göre sahne işlenir ve göz konumundaki her noktaya karşılık gelen noktanın tam olarak nerde olduğunu taramak için düzensiz gölge haritaları kullanılır. Gölge algoritmaları olarak bilinen bu haritalar, 2004 yılında Johnson ve diğerlerinin üzerinde çalıştığı Z-Tampon (Z-Buffer) veya 2004 yılında Aila ve Laine üzerinde çalıştığı Alias-Free Gölge Haritalarıdır [9,10].

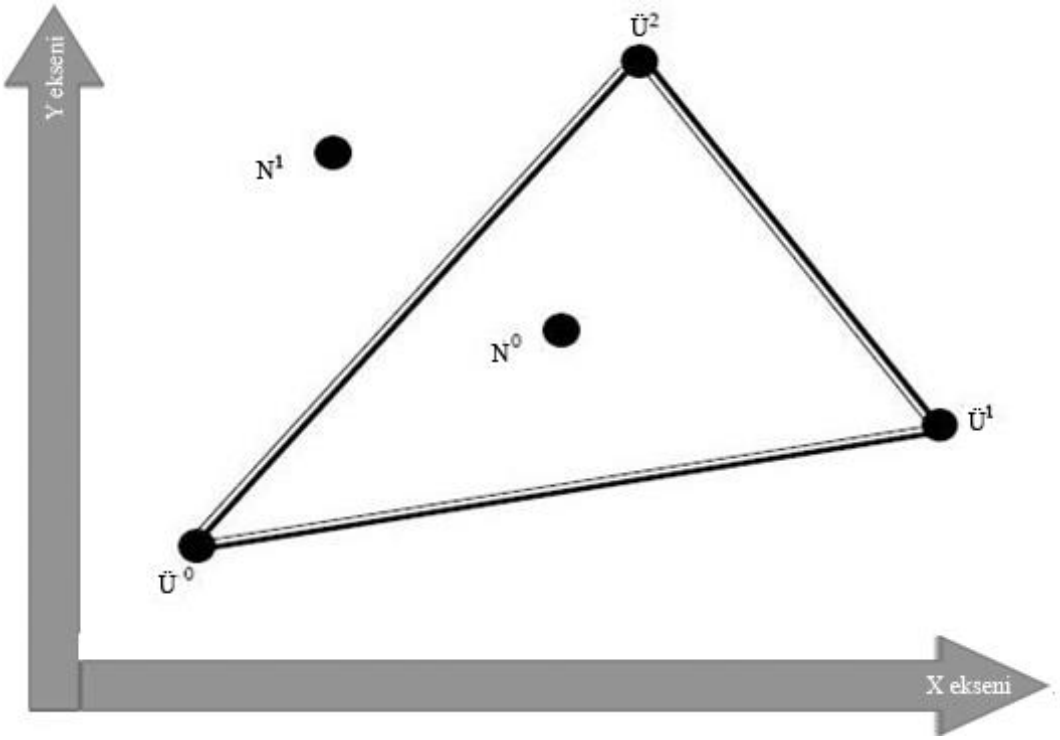
2004 yılında Johnson ve diğerleri çözüm olarak bir donanım mimarisi önerdiler. Aynı yıl Aila ve Laine ise CPU'lar üzerinde hiyerarşik bir yazılım uygulaması olarak özetlenecek bir öneride bulunarak ve yarı-saydam gölge kalıpları ve alıcıları için temel algoritmayı genişlettiler. 2008 yılında Sintorn ve arkadaşları tarafından GPU uygulaması olarak, yumuşak gölgeler üretme fonksiyonu da dâhil edildi. Bu uygulama hala öngörülen gerçek zamanlı işlemelerde teknik açıdan şimdiye kadar yapılmış GPU'lar üzerindeki en verimli düzensiz gölge eşleme algoritma yazılımıdır [9-11].

2.4. GEOMETRİK YAPILARI PİKSELLEŞTİRME

2007 yılında Akenine-Möller'in yaklaşımında, gerçek zamanlı görüntü işlemelerde üçgenler temel geometrik yapılar olarak kabul edilir ve gölgelendirme sistemi bir üçgen pikselleştirme sürecine dayandırılmaktadır. Bu nedenle üçgen pikselleştirmeyi aşağıda anlatılacak yöntemle tanımlamak gerekmektedir [12].

2.4.1. Kenar Fonksiyonu

Bu çalışmanın kapsamında düzensiz olarak yerleşen noktaların pikselleştirilmesi ve görüntü düzlemi üzerinde rastgele seçilmiş örnek noktalar ile ekran üzerindeki düzenli pikseller takas edilmiştir.



Şekil 2.7. ($Ü^0Ü^1Ü^2$) noktaları bir üçgeni göstersin ve görüntü alanı içinde düzensiz iki nokta olarak da (N^0 ve N^1) noktaları olsun.

Şekil 2.7'deki üçgende; $Ü^0$, $Ü^1$ ve $Ü^2$ üçgenin üç köşesi olarak tanımlanır ve bunların her birinin yeri $Ü^i = \left(p_x^i, p_y^i\right)$ formülü ile bulunur. $Ü^0$ ve $Ü^1$ köşe fonksiyonu aşağıdaki şekilde tanımlanmıştır [12]:

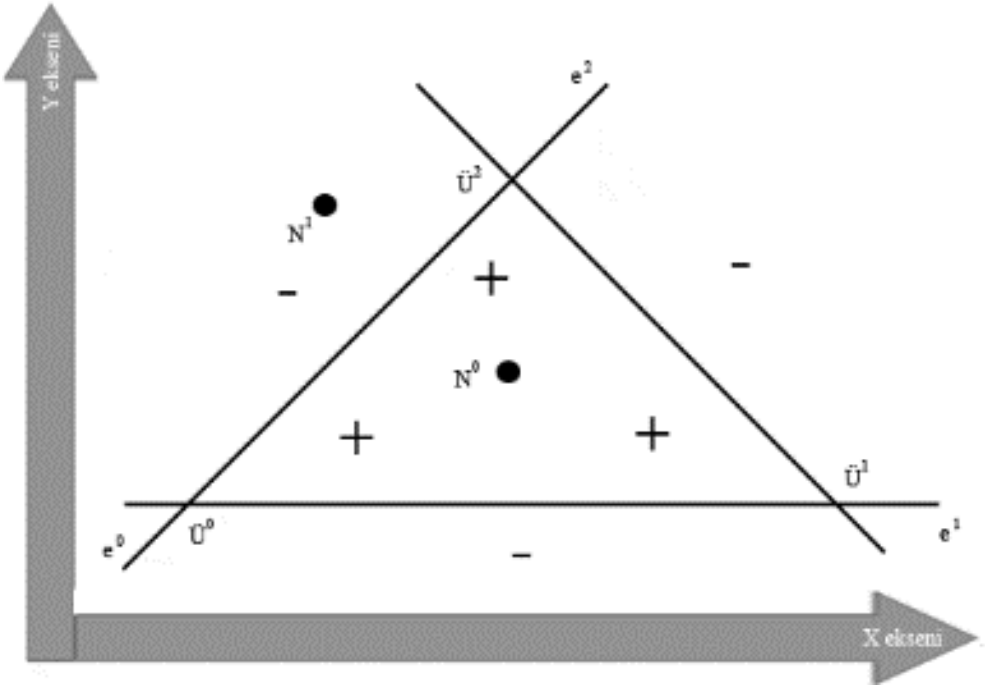
$$e\,(x,y) = \,(p_x^1 - p_x^0)\left(y - \,p_y^0\right) - \left(p_y^1 - p_y^0\right)(x - \,p_x^0) = ax + by + c \qquad (2.1)$$

Bu nedenle her bir üçgen için, tüm üç kenar fonksiyonları şunlardır:

$$e_0\,(x,y) = \,(p_x^2 - p_x^1)\left(y - \,p_y^1\right) - \left(p_y^2 - p_y^1\right)(x - \,p_x^1) = a_0x + b_0y + c_0$$
$$e_1\,(x,y) = (p_x^0 - p_x^2)\left(y - \,p_y^2\right) - \left(p_y^0 - p_y^2\right)(x - \,p_x^2) = a_1x + b_1y + c_1$$
$$e_2\,(x,y) = \,(p_x^1 - p_x^0)\left(y - \,p_y^0\right) - \left(p_y^1 - p_y^0\right)(x - \,p_x^0) = a_2x + b_2y + c_2 \qquad (2.2)$$

2.2 denklemi kullanılarak üçgende tekrar eden noktaların görünürlük testleri yapılmıştır. Örneğin, $\mathbf{N^i} = \left(s_x^i, s_y^i\right)$ kuralı ile yerlerini bulabileceğimiz, Şekil 2.7 'deki $\mathbf{N^0}$ ve $\mathbf{N^1}$ diye adlandırdığımız iki örnek noktamız vardır. Bu denklemlerden 2.2 denklemine $\mathbf{S^i}$ dâhil edip ve tüm kenar fonksiyonları değerlendirilebilir. Eğer tüm

kenar fonksiyonların değerleri sıfırdan büyük olursa, örnek nokta üçgen ile kaplıdır [12].



Şekil 2.8. Üçgen kenar fonksiyonun değerlendirilmesi.

$Ü^0$ Şekil 2.8'de gösterildiği gibi, S^0 için her üç kenar fonksiyonları değerlendirme sonuçları olumlu olduğu için, böylece üçgen içinde yer almaktadır. Üçgen gözle görülebilen bu örnek noktanın pozisyonu göstermektedir. $\mathbf{S^1}$ için durum böyle değildir. Böylece $\mathbf{S^1}$ pozisyonunda üçgen görülemez.

2.4.2. Derinlik Enterpolasyonu

Derinlik değerlerini elde etmek için üçgen içindeki örnek nokta pozisyonunda oluşan her köşe bilgisine enterpolasyon uygulamamız gerekmektedir. Renk veya doku koordinatları gibi diğer öznitelikler gölge eşleme ile ilgili değildir ve bu nedenle bunlar göz ardı edilmiştir.

Kenar fonksiyonları elde etmeden önce türetebilinecek yaygın enterpolasyonları gerçekleştirmek için barisentrik koordinatlar kullanılmıştır. Üçgenin bir köşesinden başlanıp, kenarları belli katsayılarıyla çarparak başlangıçta alınan köşeye eklenirse,

üçgen içinde istenen bir noktaya ulaşmak mümkündür. Kenarların çarpıldığı katsayılara barisentrik koordinatlar denir [12].

$$\left.\begin{aligned} u &= \frac{e_1(x,y)}{2A_\Delta} \\ v &= \frac{e_2(x,y)}{2A_\Delta} \\ \omega &= 1 - u - v \end{aligned}\right\} \quad (2.4)$$

A_Δ üçgenin işaret alanı olup, aşağıdaki kural ile hesaplanır:

$$A_\Delta = \frac{1}{2}\left((p_x^1 - p_x^0)(p_y^2 - p_y^0) - (p_x^2 - p_x^0)(p_y^1 - p_y^0)\right)$$

Tepe noktası olan Ü^{i}'yi d_i olarak varsayarak. 2.3 denkleminde elde edilen değerler aşağıdaki denklemde kullanılırsa N = (s_x, s_y) örnek noktası konumunda derinlik değerleri hesaplanabilir [12].

$$d(x,y) = \omega d_0 + u d_1 + v d_2 = d_0 + u(d_1 - d_0) + v(d_2 - d_0) \quad (2.5)$$

2.5. CUDA İLE GÖLGE EŞLEME

2013 yılında yayınlanan CUDA C Programlama Kılavuzundaki açıklamalara göre CUDA 2007'de NVIDIA tarafından tanıtıldı. Bu aslında heterojen bir bilgisayar platformudur ve grafik donanım kartlarındaki genel amaçlı hesaplamaları uygulamak için belirli API'ler sağlar. CUDA güncel grafik donanım kartları üzerinde bulunan bu paralellik mimarisi ile büyük miktarda GFLOPS hesaplama gücü ile geliştiriciler için büyük fırsatlar sunmaktadır. Karmaşık hesaplama sorunlarını çözmek için artık CPU üzerinde ihtiyaç duyulan zamanın bir kısmı GPU'larda çözülebilir.

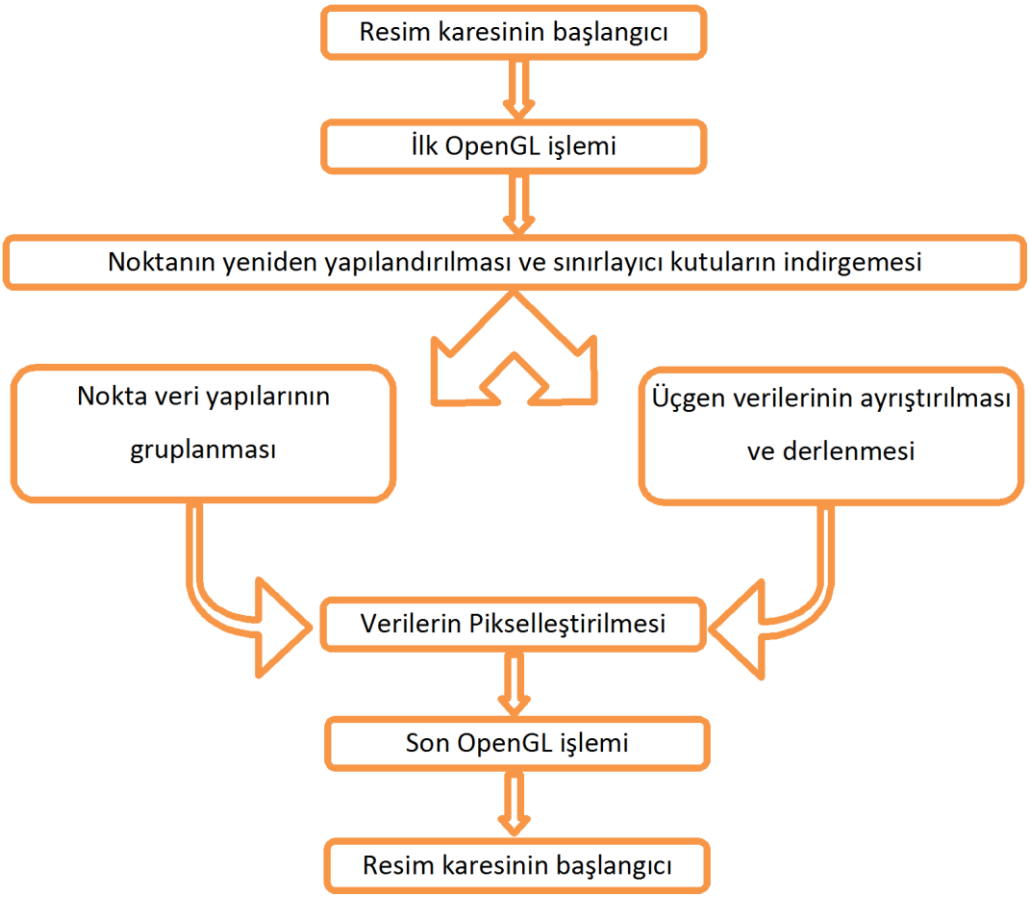
Bir piksel ve bir üçgen arasındaki tipik pikselleştirme görünürlük testi çok büyük hesaplamalar oluşturmaktadır. Pikselleştirme mevcut grafik kartları üzerindeki özel donanım birimi tarafından hızlandırılabilinir, yani doğal karekteristikliğinden dolayı

parelellenebileceğini gösterir. Bu da CUDA programlama modeli ile uyarlanmıştır. Düzensiz gölge eşleme algoritması CUDA performans hızı ile daha da iyileştirebilir.

BÖLÜM 3

ALGORİTMAYA GENEL BİR BAKIŞ

Bu bölümde, pikselleştirme tabanlı düzensiz gölge eşleme algoritmaları CUDA üzerinden açıklanmıştır. Tasarımın ana hatları tasarlanırken OpenGL işlemi dâhil edilmiş ve çeşitli CUDA çekirdekleri ile işlenip belirgin hale getirilmiştir.



Şekil 3.1. Algoritmaya genel bakış.

Genel işlem adımları ve veri akış diyagramı Şekil 3.1'de gösterilmiştir. Şekilde gösterilen tüm işlem adımları her resim karesine uygulanır, bunun anlamı dinamik

olarak güncelleştirilen piksel ve üçgen verilerinin alınıp, bunların doğru olarak işlenmesi sonucunda her bir resim karesini üretebilmektir.

Bütün sistem kavramsal olarak altı parçaya bölünür. Birbirinden bağımsız başlangıç ve son adım olarak iki OpenGL işlem geçişi vardır. Diğer dört parçada ise CUDA çekirdekleri uygulanır. OpenGL ve CUDA arasında uyum halinde çalışabilirlikten yararlanmak için CUDA 5.0 ile kodlanmıştır. Oklarla farklı parçalar arasındaki veri akışı gösterilmektedir ve okların yanındaki kutucuklar ile de diğer verilerin hangi bölüme geçeceği gösterilir. Üçgen verileri ayıklanır ve GPU'nun belleğine yüklenir. Şekilde gösterilen tüm diğer ara verilerin grafik işlemcisi üzerinde önceden yerleri ayrılmıştır ve her resim karesi geçişinde güncellenmektedir. Sistemin farklı parçaları arasındaki veri alışverişi tamamen GPU'nun belleğinde yer alır.

3.1. İLK OPENGL İŞLEMİ VE NOKTANIN YENİDEN YAPILANDIRILMASI

Tek bir resim karesi oluşturma ilk OpenGL işleminin geçişi ile başlar ve bununla beraber arzu edilen piksel derinlik değerleri oluşturulur. Sağlanan derinlik verilerine dayanarak noktanın yeniden yapılandırılması aşamasından sonra pikseller tekrardan yapılandırılarak ekranda görülen piksellerin 3D koordinatına göre uzayda nerede olduğu restore edilmiş olur. Uzaydaki koordinatları elde edildikten sonra, yeniden yapılandırılan piksel noktaları temel köşeler veya nokta olarak işlem görür. Daha sonra bunlar ışık kaynak bakış açısından görünen düzlemin içine aktarılır ve ışık kaynağından görünen düzlem üzerine izdüşümü alınır. Uzay düzleminden kamera alanına köşeler dönüştürülürken OpenGL'de de aynı yol izlenir ve sonra kamera görüntü düzlemine izdüşümü alınır.

Şimdiye kadar iz düşümü alınan piksel noktaları ışık kaynağından görünen görüntü düzlemi üzerindeki örnek noktalar olarak da görülebilir. Geçici süreli oluşturulan örnek noktalarının dağılımı ışık kaynağından bakıldığında görünen görüntü düzlemi ölçüleri düzensiz sanal bir düzleme benzer. Işık kaynağından görünen görüntü düzlemi bir sanal ekrana eşdeğerdir. Sanal ekranda her örnek noktasının konumu için

görünürlük testinin yapılacağı tanımlanır ve geometrik bilgileri piksel bilgisine dönüştürülür.

Bu düzensiz sanal düzlemin sabit bir boyutu olmadığından, görüntü bağlantı noktasının ölçeklendirilmesi ihmal edilebilir. Örnek noktaları ve üçgen verileri dönüştürüldükçe aynı yol ile aynı görüntü düzlemine izdüşümü alınacaktır, böylece pikselleştirme iyi çalışıyor olmalıdır ve görünürlük testlerinde doğru sonuçlar üretmek mümkün olacaktır.

Bu sanal ekranın sabit bir boyutu ve pozisyonu olmamasından dolayı düzlem üzerindeki tüm örnek noktaların sınırlayıcı kutularının hesaplanmasına ihtiyaç vardır. Aşağıdaki bölümlerde sınırlayıcı kutuların hesaplanması ayrıntılı olarak incelenecektir. Temel olarak, bu sınırlayıcı kutular veri yapılarını inşa edip; gereksiz hesaplamaları ortadan kaldırmak için kullanılmaktadır. Noktanın yeniden yapılandırılmasının bir parçası olarak da, paralel indirgeme ile sınırlayıcı kutusu içinde kalan bütün örnek noktalar hesaplanır.

3.2. NOKTA VERİ YAPILARININ GRUPLANMASI

Bu düzensiz noktaların pikselleştirme işleminin kamera görüntüsüne bağlı olduğu açıktır, bu durumda önemli olan piksel sayısı ve sahne içine bakma açımızdır. Sahnedeki üçgenlerin çokluğu görünürlülük testlerinin artmasına neden olmaktadır. Düzensiz noktaların pikselleştirme işlemi piksel sayısı katı kadar üçgen sayısı çarpımı şeklinde formüle edilebilir.

İşlem sürecini hızlandırmak amacıyla gereksiz hesaplamaları önlemek için, 2D tek tip kılavuz örnek noktaları daha hızlı veri yapısı oluşturur. Kılavuzun yapımında ışık kaynağından görünen görüntü düzlemindeki her örnek noktanın konumu esas alınır. Her örnek noktası x ile y koordinatlarına göre değerlendirilmiştir ve buna denk gelen hücre veya çember içine alınmıştır.

Noktanın verilerinin gruplanma işlemi iki farklı yöntem ile sağlanmıştır. 2008 yılında Green'in yaptığı çalışmalara göre, CUDA tarafından desteklenen veri yapılarının gruplanma işlemi atomik işlemleri desteklemektedir [13].

3.3. GEOMETRİK VERİ YAPILARININ AYRIŞTIRILMASI VE DERLENMESİ

Geometrik veri yapılarının ayrıştırılması gerçek zamanlı işleme tekniğini hızlandırır. Standart OpenGL ardışık işleminden farksız olarak düzensiz gölge eşleme haritaları pikselleştirme ile hareket eder. Bu yüzden OpenGL'de kullanılan ayrıştırma teknikleri burada da kullanılabilir, böylece geometrik veri yapılarının görünmeyen ve görünen tarafının ayrıştırılması ile sistem performansı artırılabilir.

Başlangıçta, üçgenlerin dönüştürülmesi ve örnek noktasına yapılan işlem gibi sanal alandan ışık kaynağına doğru izdüşümü alınır. Bundan sonra, işaret edilen alan her bir üçgen alanı için değerlendirilir, bu süre boyunca oluşan üç köşe dizisi ile üçgen belirlenir. Eğer üçgenin arka kısmının (görünmeyen taraf) ayrıştırılması etkin ise saat yönünde köşeler sıralanır ve bunlar ayrı yanlara ayrıştırılır. Önceden hesaplanmış sınırlayıcı kutular ışık kaynağından görünen görüntü düzlemi üzerindeki sanal ekrandaki örnek noktanın bölgesini belirler. Sınırlayıcı kutu dışında kalan üçgenlerin pikselleştirme ile hiçbir ilgisi yoktur, bu yüzden de geçersiz olarak işaretlenir.

2008 yılında Sengupta ve arkadaşlarıyla yaptıkları çalışmalarında, ayrıştırma değerlendirmesi sonucunda, üçgen verileri paralel bir derleme işlemi tarafından derlenmiştir [14].

3.4. DÜZENSİZ NOKTALARIN PİKSELLEŞTİRİCİLER TARAFINDAN TARANMASI

Buraya kadar, örnek nokta ve üçgen verileri ayrıştırılmış; iyi bir form içinde paketlenerek düzensiz pikselleştirme tarafından işlenmeye hazır hale getirilmiştir. Bu aşamada, üçgen verilerinin genel bellekten çekilmesi, kenarlara ait denklemlerin kurulması ve düzensiz örnek noktaları sınırlayıcı kutularda olan üçgen

poligonlarından geçirilme işlemleri değerlendirilmiştir. Bir örnek nokta pozisyonu üzerinde her bir üçgenin pikselleştirilmesi ile derinlik indirgemesi yapılır. Eğer üçgen oluşursa örnek noktası için gölge casteride oluşur, bunun anlamı noktanın ışık kaynağına daha uzak olmasıdır. Örnek noktası gölge şablon tamponunun içine gölgede olarak işaretlenir. Gölge şablon tampon boyutu sahip olduğumuz ekranın piksel sayısına eşittir. Ekrandaki her pikselin gölge içinde olup olmadığını gösteren işaretçisi vardır.

Son olarak, ikinci OpenGL işlemine geçilir, her piksel gölge şablon tamponundan taranarak çağrılır. Bu parçanın gölgede olup olmadığını denetler ve bu parça için doğru gölgelemeyi hesaplar.

3.5. SONUÇ

Bu altı adımdan sonra, gölgeler çerçeve (frame) tamponun içine doğru yürütülür ve sonunda ekranda görüntülenir.

Takip eden bölümlerde, düzensiz gölge eşleme tasarımı daha kapsamlı bir şekilde açıklanmıştır. Bölüm dörtte; derinlik üretimi, noktanın yeniden yapılandırılması, noktanın gruplanması ve üçgenlerin ayrıştırılması daha ayrıntılı olarak incelenmiştir. Uygulamada kullanılan performans artırıcı çeşitli teknikler de bölüm beşte açıklanmıştır. Bölüm altıda düzensiz pikselleştirme uygulamaları üzerine yoğunlaşılmıştır.

Sistem Gereksinimleri ve Tasarımın Tanımlamaları

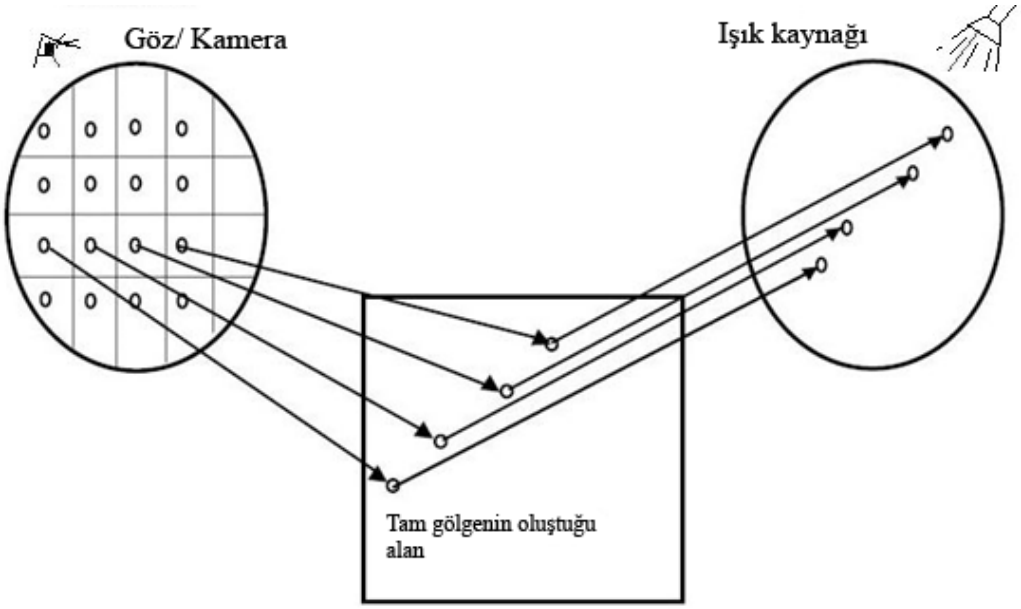
Geliştirilen tasarım da kullanılan sistem gereksinimleri:

1. CUDA versiyonu: 5.5
2. API: CUDA C
3. Donanım: Geforce GTX 555.45
4. Çözünürlük: 256x256

BÖLÜM 4

NOKTANIN VERİLERİNİN YENİDEN YAPILANDIRILMASI

Bu uygulamada girdi olarak alınan, düzensiz örnek noktalarının ilk standart OpenGL işleminin geçişinde yeniden yapılandırılmasına ihtiyaç duyulur. Ekranda her bir piksel için, ekran alanı içine piksel konumunun ve derinliğinin bilgisi alınır. Örnek noktalarının uzay düzleminden kamera görüntüsüne dönüşmesi için dönüşüm matrisleri uygulanır ve şekil 4.1'de de gösterildiği gibi ışık kaynağından görünen görüntü düzlemine iz düşümü alınır. Bu bölümde derinlik değerlerinin bulunması açıklanmıştır.



Şekil 4.1. Örnek noktalarının yeniden yapılandırılması.

4.1. DERİNLİK DEĞERLERİ

Kamera görüntü düzleminde, piksel pozisyonları elde etmek oldukça kolaydır. x ve x koordinatlarında 0,5 çap yönünde geçecek piksel endeks değerlerinin tespit edilmesi

ise bundan daha basit bir işlemdir. Derinlik değerlerinin tespiti için birden fazla işlem basamağı uygulanır. Bu işlem basamakları aşağıda sırasıyla açıklanmıştır.

4.1.1. OpenGL Komutlarından "glReadPixels" Komutunun Kullanımı

Derinlik değerlerini ayıklamak için yapılacak işlemlerden biri OpenGL işlev çağrısı olan Khronos Group (2008) tarafından tanıtılan **"***glReadPixels***"** kullanmaktır, ilk OpenGL işleminin geçişinde döndürülen piksel verileri için resim karesi (çerçeve) tamponu güncellenir. Parametreler **"***GL_DEPTH_COMPONENT, glReadPixels***"** olarak ayarlandığında derinlik tamponundan derinlik değerleri döndürülür, bu değişken noktaya dönüştürülür ve 0 ile 1 aralığındaki değerlerle eşleştirilir.

Bu yaklaşımı uygulamak basit olduğu kadar bu yaklaşımı uygulamanın dezavantajları da vardır. Bunlar, OpenGL özelliklerine göre **"***glReadPixels***"** istemci bellek içine çerçeve (resim karesi) tamponundaki verileri çıkarır. Bu, verilerin dolaylı olarak GPU'dan CPU'ya aktarılması anlamına gelir. Eğer derinlik değerleri bir tampon nesnesine paketlenirse veya bir **"***cudaGraphicsResource (CUDA 5.0)***"** komutu kullanılırsa bir sonraki CUDA çekirdeği içine geçirilmiş olur. GPU belleğine verileri geri getirmek için kapalı bir döngü komutu olan **"***cudaMemcpy***"** ile çağrılır.

4.1.2. Resim Çerçeve (Frame) Tampon Nesnesini Kullanmak

Resim çerçeve arabelleği verisini elde etmenin başka bir yolu ise resim çerçeve tamponu (FBO) 2008 yılında Ahn tarafından geliştirilen FBO nesnesi oluşturmaktır. FBO nesnesi OpenGL işlemindeki ardışık düzene bağlanır. Bu şekilde, ekranda görüntülenen OpenGL işlemi sonucu ilişkili çerçeve tampon nesnesi üzerine yönlendirilmiş olur. OpenGL derinlik değerlerini ortaya çıkarmak için FBO tasarımı kullanıma sokulmuştur [15].

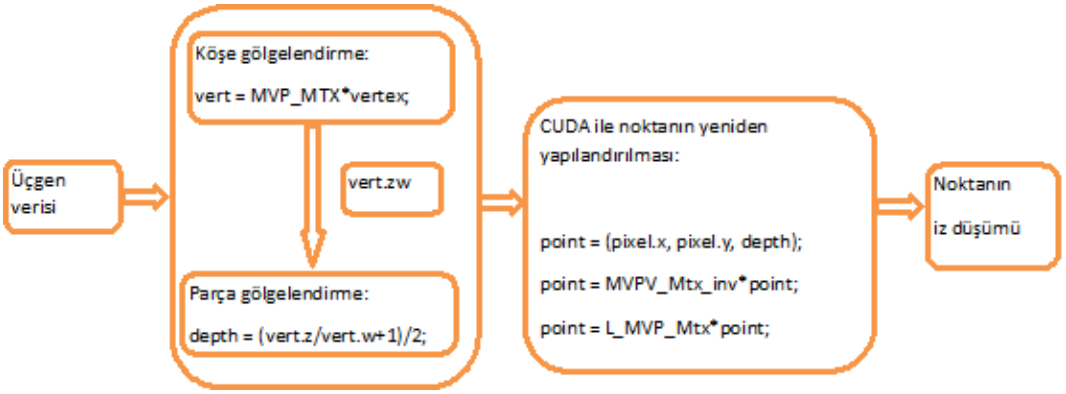
Resim çerçeve tamponunun nesnesi üzerinde bir dizi kapsamlı işlem yapılacaktır. FBO'lara eklenebilir iki tür görüntü vardır. Bunlar; doku görüntüleri ve tamponda işlenecek görüntülerdir. Doku veya tamponda işlenecek görüntülerdeki derinlik değerlerini yerleştirmek için bu değerler FBO'ların derinlik bağlanma noktasının

üzerine monte edilebilir. Ancak yapılan testler sonucunda, onlara CUDA çekirdekleri tarafından erişilemediği gözlenmiştir. FBO'nun işlevselliğinin OpenGL'e bağlı uygulama olduğu gözlemlenmiştir, bu sonuca göre CUDA FBO'larla çalışmayı desteklememektedir. Bu nedenle, bir FBO'larda oluşturulan derinlik verisi CUDA içinde bir **"***cudaGraphicsResource***"** olarak kullanılamaz. Derinlik değerlerinin tespiti için nesnelere gölge fonksiyonu oluşturma zorunluluğu doğmuştur. 2006 yılında Kessenich ve arkadaşlarının tanımına göre, özel değişken "*gl_FragCoord***"** komutu derinlik verisinin de dâhil olduğu konum verisini doğrudan parça gölge fonksiyonuna aktarılabilindiğini göstermiştir [16].

Şekil 4.2'de gösterildiği gibi, sadece her köşe için model-manzara iz düşümü dönüşümü uygulayan bir köşe gölgelendirici fonksiyonu oluşturulmuştur ve daha sonra Z- ve W- bileşenlerine dönüştürülmüş köşeleri parça gölgelendiriciden geçirilmiştir. Parça gölgelendirici olarak, iz düşüm bölünmeler gerçekleştirip ve derinlik değerleri istenen aralığa (0 ile 1 arasında) eşleştirilmiştir. Parça gölgelendirici programının sonunda, FBO gölgesiz kanallarından derinlik değerleri süzülmüş ve daha sonraki işlem aşamasındaki CUDA çekirdekleri tarafından kullanılmaya hazır hale getirilmiştir.

4.2. ÖRNEK NOKTALARININ YENİDEN YAPILANDIRILMASI

Örnek noktalarının yeniden yapılanması üç farklı alan arasındaki bir dizi köşe dönüşümünden başka bir şey değildir. Bunlar; ekranın alanı, manzara alanı ve ışığın alanıdır. CUDA ile noktanın yeniden yapılandırılması süreci bloklar halinde şekil 4.2'de gösterilmektedir. Çekirdeğin sonunda, iz düşümü alınan noktalar genel bellekte saklanır.



Şekil 4.2. İlk OpenGL işlem geçişi ve noktanın yeniden yapılandırılması [21].

4.2.1. OpenGL ve CUDA'nın Bellek Erişimlerinin Değerlendirilmesi

Bu bölümde OpenGL ile sadece derinlik değerleri elde edilmiş ve örnek noktaları ayrı bir CUDA çekirdeğiyle yeniden yapılandırılmıştır. Bu örnek noktaların doğrudan OpenGL gölgelendirme fonksiyonu tarafından yeniden yapılandırılabildiği ve FBO'da depolanacağı kesindir. Bu çalışmada bu durum CUDA çekirdeği ile gerçekleştirilmiştir, bundaki amaç nokta verilerine erişim süresini ve aynı zamanda her erişimdeki veri boyutunu yani bant genişliğini minimum seviyelere indirmektir.

Nokta verilerinin sağlam veri yapılarına doğru yapılandırılması tüm örnek noktalarının sınırlama noktasının neresinde olacağına bağlıdır. Sınırlayıcı kutuyu hesaplamak amacıyla, paralel bir indirgemenin tüm örnek noktaları üzerinden yapılması gerekmektedir. Eğer OpenGL ile yeniden yapılandırmak istenilirse, x ve y gibi iki koordinatlı her noktaya küçültülmüş bir çekirdek tarafından erişmek gerekmektedir. Bunun aksine, OpenGL sadece derinlik değerlerini oluşturursa, bu işlem yapılırken noktaları inşa edilebilir ve doğrudan indirgeme için çekirdek verilmesine gerek kalmaz. Aynı çekirdek indirgemesi ile yeniden birleştirilerek, genel bellek erişimi yapmaya gerek kalmaz, bu sayede her piksel başına derinlik değeri için sadece erişim yeterli olur.

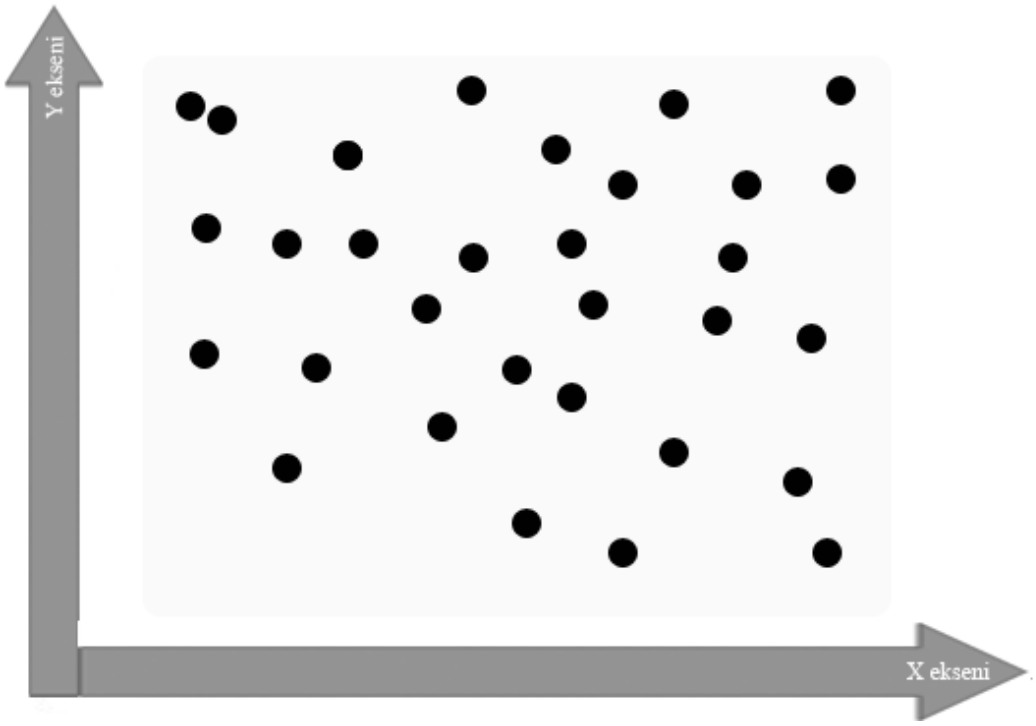
4.2.2. Gölgelendirme Fonksiyonunun Detayları

Kameranın alanı ile ışık alanının iz düşüm görünümünün ters matris alınır. Bunlar bir dizi içinde paketlenip GPU'nun sabit belleğine kopyalanır. Sabit bellekteki önbelleğe alınır ve sonradan başlatılan tüm CUDA çekirdekleri için erişilebilir durumda olur.

Homojenleştirme, sadece x ve y koordinatları üzerinde gerçekleştirilir, pikselleştirme aşamasında derinlik enterpolasyonunu kolaylaştırmak için lineer boşluğu içinde z-koordinatı değeri elde edilir. Her matris çarpma işlemi sonrasında W-bileşenine karşılık gelen değeri bulmak için ön hesaplama yaparak uygulama optimize edilir ve yüksek çözünürlüklü kayan noktaları kaydetmek için karşılıklı her koordinat bileşeni çarpılır.

4.3. SINIRLAYICI KUTULARIN OLUŞTURULMASI

Nokta yapılanması sonrasında, Şekil 4.3'de gösterildiği gibi ışık kaynağından görünen görüntü düzlemi üzerinde örnek noktaları düzensiz olarak dağıtılır.



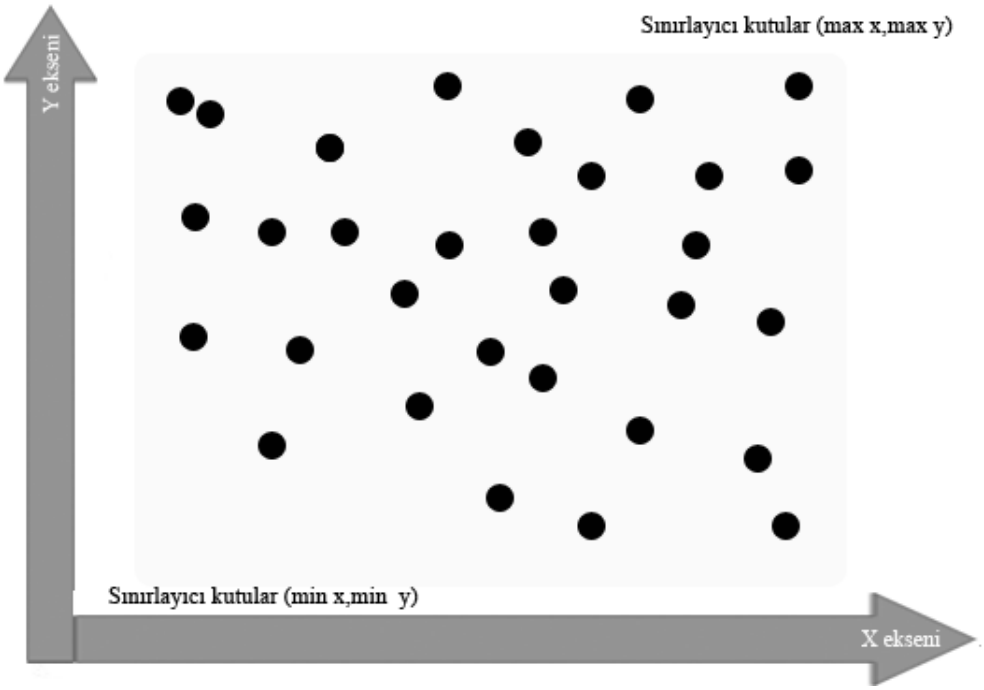
Şekil 4.3. Işık kaynağından görülen resim karesi alanı üzerindeki örnek noktalar [17].

4.3.1. Sınırlayıcı Kutu Kullanmanın Amacı

Sınırlayıcı kutular gölge fonksiyonundan gereksiz verileri ayrıştırmak için büyük önem taşımaktadır. Sınırlayıcı kutuların gerekliliğinin iki önemli nedeni vardır.

İlk nedeni, 2D homojen kılavuz çizgilerindeki örnek noktalarının sınırlayıcı kutusunun içine doğru yapılandırılmış olması gerekmektedir. O sınırlayıcı kutudaki düzensiz örnek noktalarının dağılımını soyutlayan ve yoğun bir dikdörtgen bölgede pikselleştirmek için gereken bütün örnek noktalar paketlenir. Bu nedenle, 2D kılavuz çizgilerindeki örnek noktaları x ve y eksenleri boyunca iki eşit sınırlama kutusunun alanını bölerek inşa edilmelidir. Sınırlayıcı kutunun büyüklüğü ve konumu değişince, 2D kılavuz çizgileri bu değişime kendiliğinden uyum sağlar. Böylece daha homojen nokta dağılımı sağlanır.

İkinci nedeni ise, üçgen veriler ile ilgilidir. Çünkü sınırlayıcı kutu dışında bulunan üçgenler pikselleştirme için önemsiz olduğundan, üçgen verileri ayrıca sınırlayıcı kutular dışında ayrıştırılmalıdır.



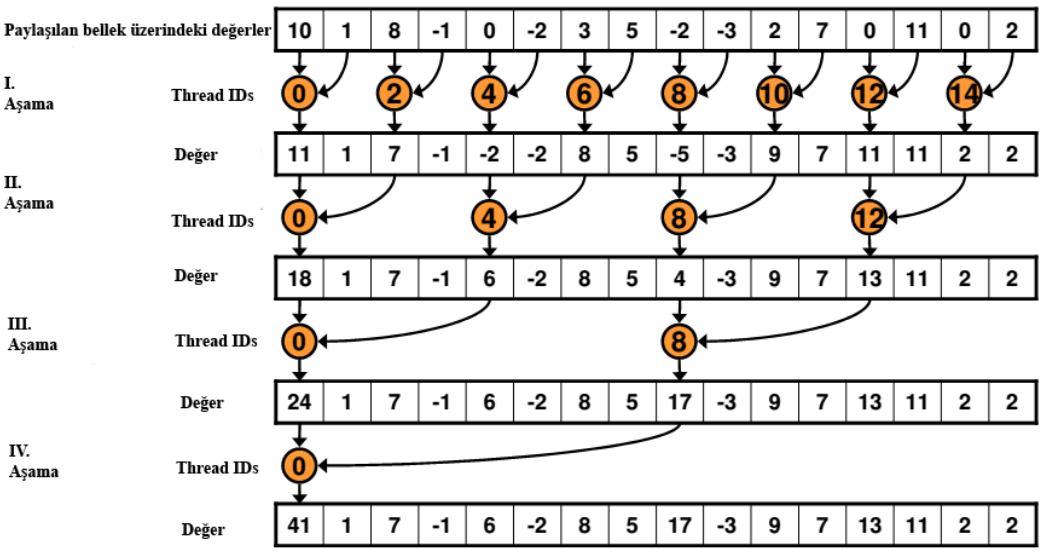
Şekil 4.4. Sınırlayıcı kutuların en büyük ve en küçük değerleri [17].

Şekil 4.4'de gösterildiği gibi, sınırlama kutuları x ve y koordinatları arasındaki tüm örnek noktaların minimum ve maksimum değerlerini hesaplama sürecidir.

4.3.2. Paralel İndirgeme Süreci

2007 yılında Owens ve arkadaşları tarafından, paralel indirgeme yaygın ve güçlü bir GPGPU teorisi olarak açıklandı. Girişteki büyük veri akışı küçük bir veri akışına indirgenir, yani indirgeme sonrası akış tek bir eleman akışına doğru azaltılmıştır. Bu veri akışındaki tüm elemanların maksimum veya toplam değerlerini hesaplamak için de kullanılabilir.

Şekil 4.5'de görüldüğü gibi, 2007 yılında Harris paralel indirgeme yöntemi GPU'larda bir ağaç veri yapısı yaklaşımı uygular. Şekilde, girişteki veri akışının maksimum değerlerini hesaplamak için maksimum indirgeme gerçekleştirilir. Her seviyede, ikili (binary) gruplanarak bir alt seviyeye maksimum olan değer aktarılır [19].



Şekil 4.5. Paralel indirgeme yöntemi ile maksimum değerin hesaplanması [19].

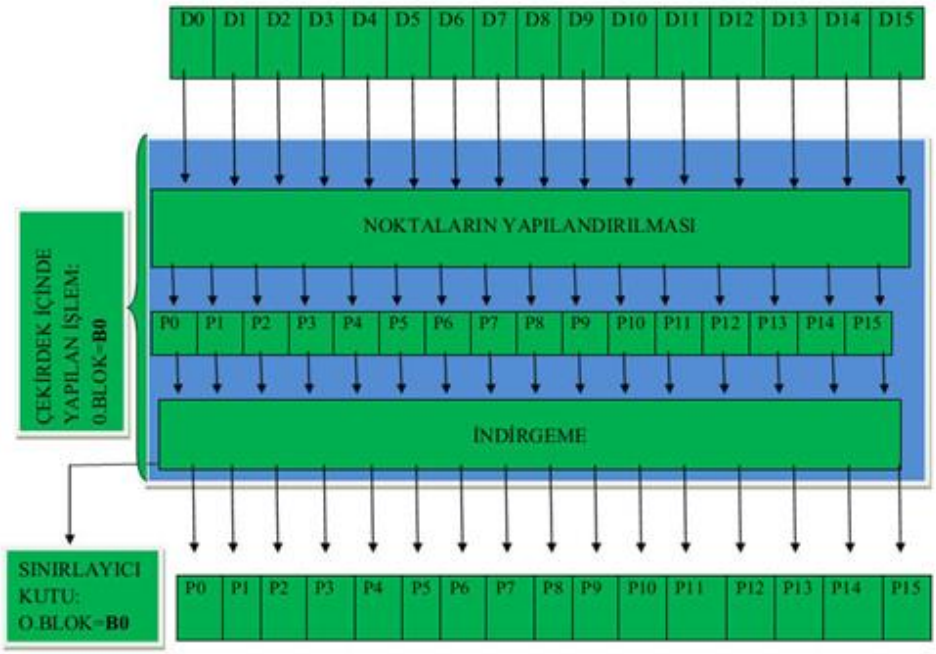
4.3.3. CUDA'nın Mevcut Kütüphanelerinin Kullanımı

CUDA'da uygulanan paralel algoritma verileri mevcut kütüphanelerinde vardır. Örneğin, 2007 yılında Harris'in CUDPP ve 2013 yılında Hoberock ve Bell'in CUDA'da uygulanan en yaygın GPGPU indirgeme algoritmalarıdır [19-20].

Bu kütüphaneler tarafından desteklenen standart dört farklı maksimum ve minimum değer hesaplanabilir, ancak bu performans açısından verimli bir çözüm değildir. Nokta verilerinin kütüphane işlevleri ile uyumlu olacak şekilde düzenlenmiş olması gerekir ve uyumsuz olanlar için gereksiz global bellek girişlerinin yapılması gerekecektir.

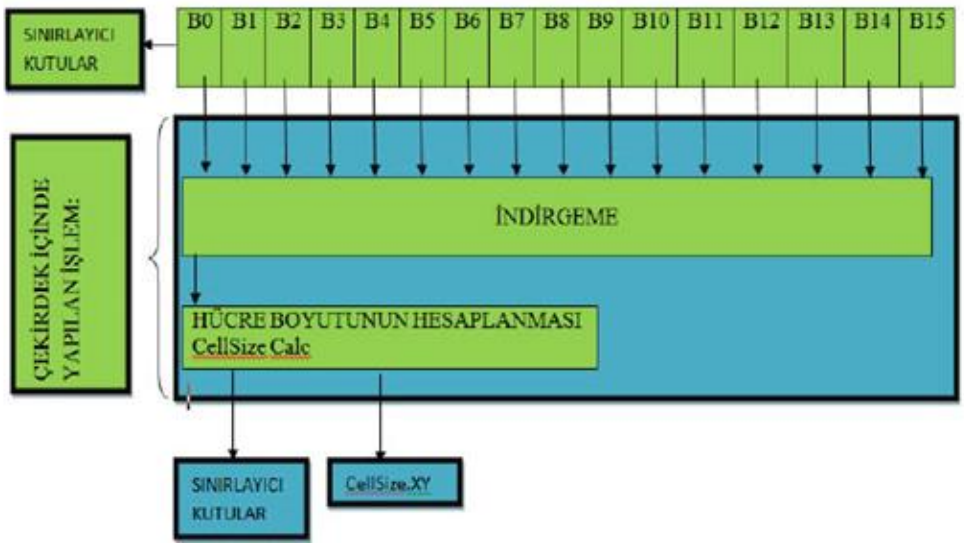
4.3.4. Özelleştirilmiş Çekirdek İndirgemeleri

Nokta yapılandırılması ile entegre edilerek özelleştirilmiş paralel (çoklu işlem parçacıkları-Multithreading*)* indirgeme çekirdeği bu aşamada en uygun çözümdür. İlk olarak, yeniden yapılandırılan noktalar paylaşılan bellek üzerinde tamponlanır ve indirgeme her blok içinde tampon üzerindeki noktalar karşılıklı bir tampon bölümünden diğer bölümüne geçişi uygulanarak yapılır. Dört ikili (binary) işlemlerde (x için minimum ve y için minimum, x için maksimum ve y için maksimum) her düzeyde bir araya dizilir. Son olarak, yeniden yapılandırılan noktalar genel hafızada depolanır ve her blok kendi *float4* sınırlayıcı kutusunu Şekil 4.6'daki gibi üretir.



Şekil 4.6. 0. blokta ki nokta yapılandırılması ve indirgenmesi [21].

Bir diğer indirgeme aşamasında ara sınırlayıcı kutusundaki bütün verileri bir araya getirmek gerekir. Şekil 4.7'de gösterildiği gibi, ikinci indirgeme aşamasında tüm bloklar arasında bütün sınırlayıcı kutuları kapsayan tam sınırlayıcı kutuları hesaplamak için çağrılır. Aynı zamanda, gerekli olan boyutlara göre kılavuz çizgileri üzerinde tekdüze kılavuz hücrelerinin boyutları x ve y yönlerinde hesaplanarak belirlenir.

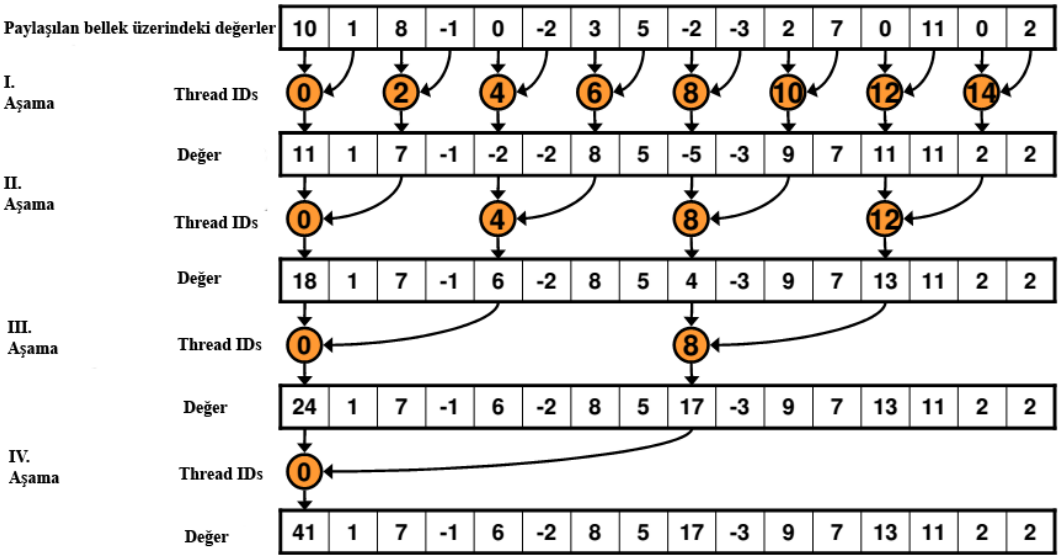


Şekil 4.7. Son indirgeme aşaması [21].

Bu çalışmada indirgeme çekirdeklerini optimize etmek için 2007 yılında Harris tarafından ortaya konulan standart kuralları esas alınmıştır [19].

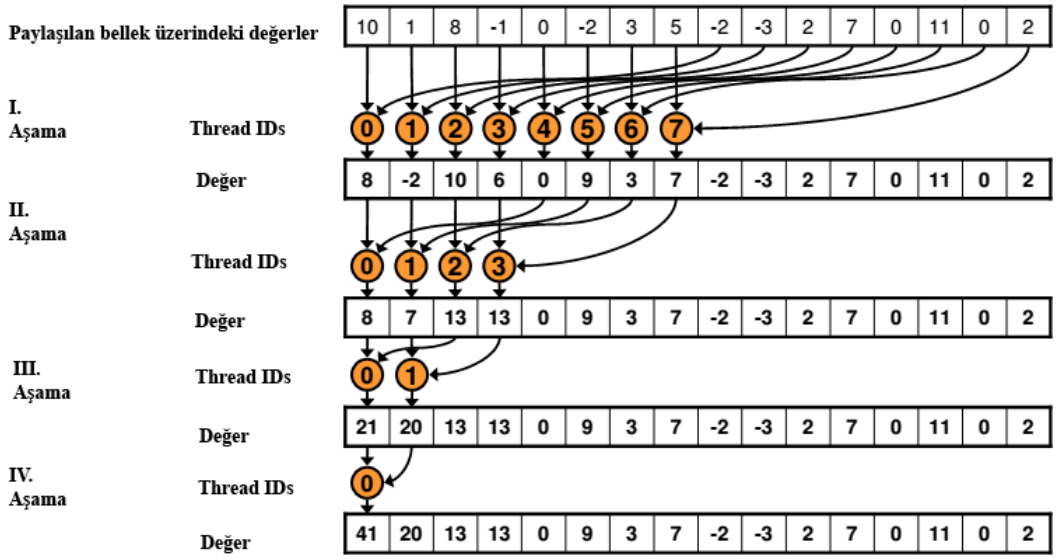
Bir CUDA GPU veri akışı, paylaşılan bellekte yerleşen çoklu işlemcisi SM (Streaming Multiprocessor) içinde 16 kümeye ayrılmıştır. Yarım zincir içinde, paylaşılan her hangi bir hafıza bankası çakışmalarında farklı iş parçacıkları aynı paylaşılan bellek bankasına erişilebilmesini serileştirmiş olacaktır.

Şekil 4.8'de gösterildiği gibi, bu serpiştirmeli bellek adresleme modeli paylaşılan hafıza bankasının her düzeyinde, indirgemeden oluşan çakışmaların nedenidir.



Şekil 4.8. Paralel indirgeme: serpiştirilmiş adresleme yöntemi [19].

Bu çakışma sorunlarını çözmenin bir yolu ise Şekil 4.9'da gösterilen sıralı adresleme yöntemini uygulamaktır. Sıralı adresleme kullanarak yarım zincir içindeki her iş parçacığı yalnızca kendine özgü paylaşılan bellek bankasının kümesine erişir, böylece kümelerdeki çakışma sorunları çözülmüş olur.



Şekil 4.9. Paralel indirgeme: sıralı adresleme yöntemi [19].

Çoklu Yükleme (Multiple Load)

İlk seviyeyi indirgemeden sonra iş parçacıklarının yarısı kullanılamaz haldedir. Aslında, çoklu yüklemelerde üst düzeyde her iş parçacığı tarafından işlenir ve sonuç olarak paylaşılan hafızada depolanmadan önce ikili (*binary*) işlem kayıtları üzerinde gerçekleştirilebilir. Bu şekilde, az oranda her iş parçacığının parçalı yapısı artmıştır ancak parçacıklarının toplam sayısı en az ikiye bölünür. İkinci aşamadaki çekirdekteki indirgeme için de aynı şeyler uygulanır.

Döngü Açma (Loop Unrolling)

Döngüler CUDA çekirdekleri için önemli bir yük ortaya çıkarır. Döngüler, başlangıçta bütün indirgeme seviyelerindeki tekrarların kontrol edilmesi için gereklidir. Tekrarların sayısı bilinmektedir, ancak her bir iş parçacık bloğu veri boyutu kadar uzun sürede işlem gerçekleştirir. Bu nedenle, döngüler tamamen açılabilir.

Performans Sonuçları

Yapılan testler sonucunda; yukarıda belirtilen çoklu yüklemeden oluşacak yük aşımı (*overhead)* önlemek için ve daha iyi bir performans elde etmek için sıralı adresleme yöntemi kullanılmıştır.

Noktaların yeniden yapılandırılması ile birlikte derinliğin yüklenme işlemi üst düzeyde yapılmaktadır; her bloğun boyutu 256 bit renk değerine ayarlanmasına rağmen daha az yer tutmaktadır ve döngülerin tamamı açılmaktadır. Son olarak, her iki aşamada çekirdeklerde yapılan indirgemelerin GPU üzerinde yürütme zamanı 154,98 µs (mikro saniye) olmuştur.

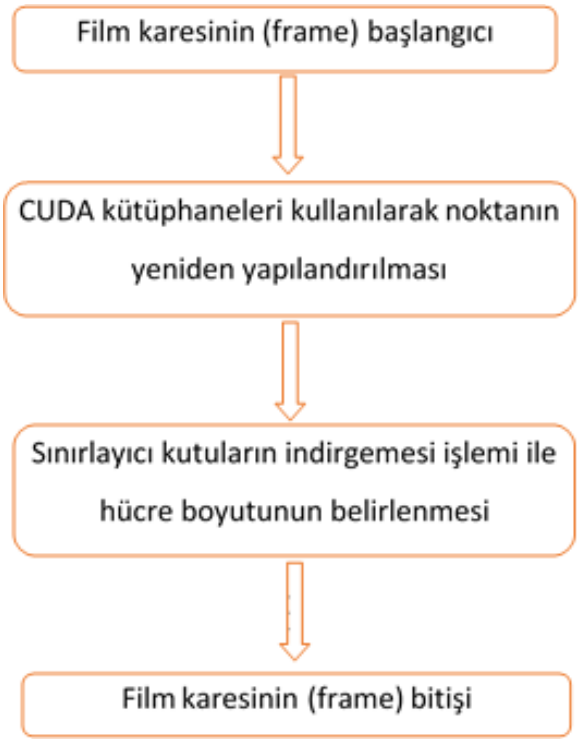
4.4 SONUÇ

Çizelge 4.1'de noktanın yeniden yapılanması ve sahne performansı özetlenmiştir. Özelleştirilmiş çekirdek ile karşılaştırmak için CudppScan kullanılır (indirgeme de henüz CUDPP uygulanmadı). OpenGL görüntü oluşturma süresi yazılım zamanlayıcısı tarafından ölçülmüş ve CUDA çekirdeğinin yürütme hızının GPU'da aldığı süre NVIDIA Visual Profiler tarafından ölçülmüştür. Sadelikten dolayı, çözünürlük ayarları 512 x 512 aralığında tasarlanmış, bu da 250.000 örnek noktanın işlendiği anlamına gelmektedir. 2010 yılında Zhang ve Majdandzic'in yaptığı çalışmada noktanın yeniden yapılanması sahneden bağımsız olmasına rağmen birinci OpenGL geçiş hızı sahne içindeki üçgen sayısına bağlıdır. Test amacıyla bu çalışmada yaklaşık olarak 14000 üçgen kullanılmıştır [21].

Çizelge 4.1. Noktanın yeniden yapılandırılması ve indirgenmesi performans özeti [21].

Yaklaşım	**cudppScan**		**Özelleştirilmiş İndirgeme**	
Maddeler	**Rutinler/Çekirdek**	**Süre(µs)**	**Rutinler/Çekirdek**	**Süre (µs)**
1	1'inci OpenGL	394.48	1'inci OpenGL	394.48
2	Noktanın Yeniden Yapılandırılması	111.39	Noktanın Yeniden Yapılandırılması ve Sınırlayıcı Kutuların İndirgenmesi	147.04
3	cudppScannX4	348.13	Son aşamadaki Sınırlayıcı Kutuların İndirgenmesi	7.94
Toplam		854.00		549.46

Yukarıda gösterilen çizelgedeki performanslara göre şekil 4.10'da gösterilen yöntem kullanılarak performans artırılmıştır.



Şekil 4.10. Noktanın yeniden yapılandırılması ve indirgenmesi.

BÖLÜM 5

PERFORMANS ARTIRMAK İÇİN KULLANILMIŞ TEKNİKLER

Bu bölümde tasarım ile örnek noktasına ve üçgen verilerine erişimi hızlandırmak için uygulanmış çeşitli teknikler anlatılmıştır.

5.1. ÖRNEK NOKTALARININ VERİ YAPILARI

Bu tasarımda, görüntü oluşturulurken çeyrek milyon örnek noktasını işlemek gerekir (512x512 çözünürlükte). Her bir örnek noktası değişken x, y ve z gibi üç koordinata sahiptir. Bu çalışmada noktaya veri yapısı paketi içinde var olabilecek herhangi bir diğer bilgi alınmamış, nokta verilerinin boyutu yaklaşık 3 MB civarında kabul edilmiştir.

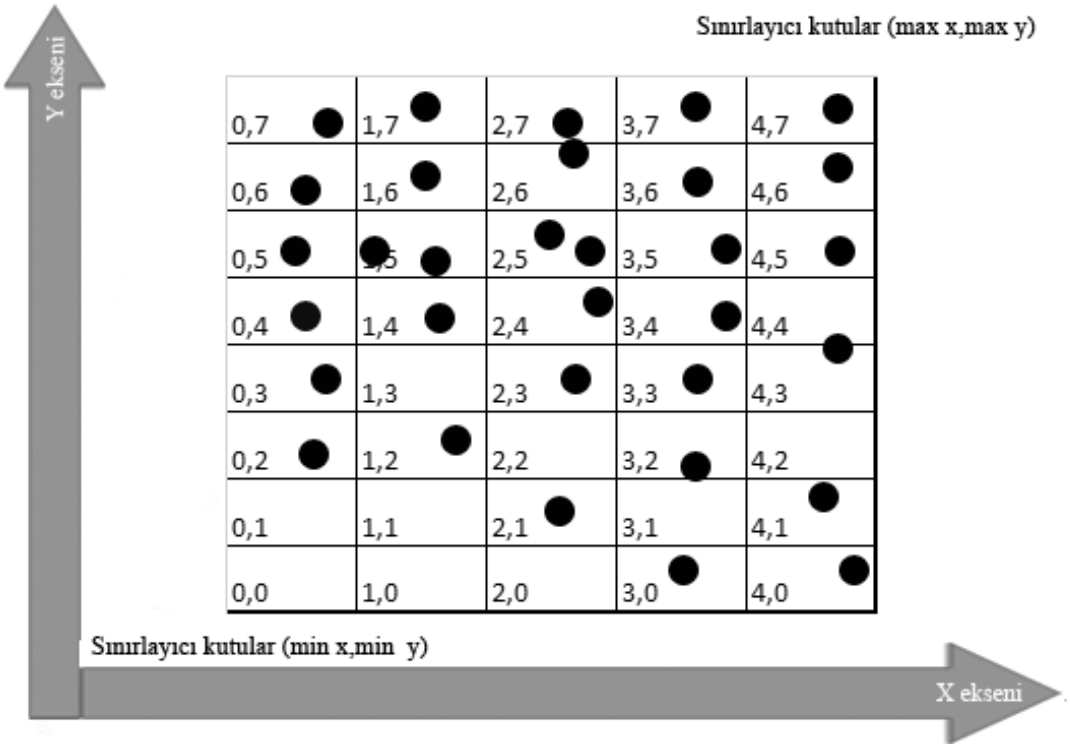
Örnek noktalar ışık kaynağından görünen görüntü alanı boyunca düzensiz olarak dağılmış durumdadır ve noktanın indeksi ile onun mekânsal konumu arasında hiçbir somut ilişki yoktur. Eğer herhangi bir hızlandırılmış veri yapısı uygulanmazsa, 100 bin üçgen için yeterli büyüklükte bir sahnede mekânsal bilgi eksikliği nedeniyle her üçgen bütün örnek noktalarını yinelemek zorunda kalacaktır. Eğer üçgenlerin sayısı kadar örnek noktalarının veri boyutu ayarlanmazsa, genel belleğin erişmek zorunda kalacağı 3 TB noktalı veri olacaktır. Ayrıca bellek erişimleri arasında hesaplamalar ağır olacaktır. Bu durumda tüm bellek erişimleri sadece pikselleştirme işlemi boyunca dağılmış olacak ve bellek erişim gecikmeleri sahnenin yinelenmesi arasında bu pikseller çok iyi gizlenmemiş olacaktır. Bu da çözünürlük kalitesini düşürecektir.

Yukarıda belirtilenler göz önüne alındığında nokta verileri için hızlandırılmış bir veri yapısı oluşturma gerçeği kaçınılmazdır.

5.1.1. 2D Homojen Kılavuz Çizgileri

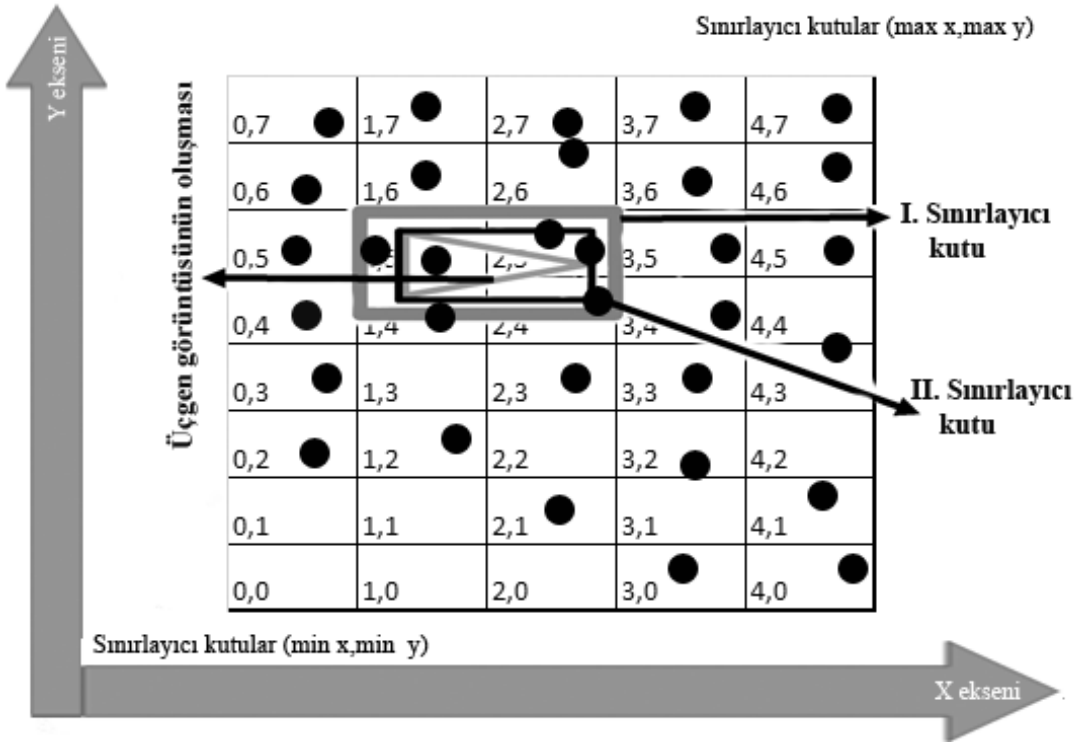
Bilindiği gibi pikselleştirme temelde iki boyutlu olarak çalışmaktadır. Bu nedenle daha komplike bir 3D homojen kılavuz çizgi veri yapısı benimsemek gerekli değildir. Bunun yerine homojen hücre yapısına sahip ve bu hücreler arasında geçiş kolaylığı sağladığından dolayı, nokta veri yapıları için 2D homojen kılavuz çizgi veri yapısı seçilmiştir [21].

Şekil 5.1, ışık kaynağından görünen görüntü düzlemi üzerinde kılavuz çizgilerinin nasıl göründüğünü gösterir. Sınırlayıcı kutudaki örnek noktaları alınır, x ve y eksenlerinde bir kılavuz çizgisi boyunca homojen olarak ayrılır. Kılavuz çizgileriyle meydana gelen her bir dikdörtgen bölgeye bir hücre ya da bir bölüm adı verilir. Hücre indeksi kılavuz çizgilerinin sol alt köşesinden sağ üst köşesine doğru artarak belli bir sırada düzenlenmiştir. Şekilde görüldüğü gibi farklı gridlerin hücre indeksleri iki boyutlu vektörleri göstermektedir ancak, gerçek bir uygulamada bunlar tek boyutlu ve işaretsiz tamsayı olarak saklanacak hale getirilmektedir. Sınırlayıcı kutunun değişik bölgelerinde yer alan noktalar kendi indeksli hücrelerine kaydedilir.



Şekil 5.1. 2D homojen kılavuz çizgileri [21].

Şekil 5.2'de nokta verilerine erişimin nasıl hızlandırılabileceğine dair temel fikir gösterilmiştir. Üçgenin görüntüsü oluşturulduğunda II. sınırlayıcı kutu ile gösterilen alan da üçgenin sınırlayıcı kutusu hesaplanır ve I. sınırlayıcı kutu olarak işaretlenen kutu ise üçgenin sınırlayıcı kutuları ile kaplı olan kılavuz çizgilerinin hücre numarasını belirlemek içindir. Noktalar I. sınırlayıcı kutusu içinde olmalıdır ve bunun dışındaki noktalar atlanır.



Şekil 5.2. Kutu geçişi sınırlandırma üçgeni [21].

2009 yılında Fatahalian ve arkadaşları ile 2010 yılında Liu ve arkadaşlarının yaptıkları çalışmalarda üçgenlerin sınırlayıcı kutusundaki çapraz geçişi üzerinde durmuşlardır [22,23].

Üçgenler dönüştürülüp ışık kaynağından görünen görüntü düzlemi üzerine iz düşümü alındıktan sonra, $Ü^0Ü^1Ü^2$ üçgeninin sınırlayıcı kutusu aşağıdaki yöntem ile hesaplanır:

$$Minimum\ sınırlayıcı\ kutu = \left(\min(p_x^0,\ p_x^1,\ p_x^0), \min\left(p_y^0, p_y^1, p_y^2\right)\right)$$
$$Maksimum\ sınırlayıcı\ kutu = \left(\max(p_x^0,\ p_x^1,\ p_x^0), \max\left(p_y^0, p_y^1, p_y^2\right)\right)$$
(5.1)

Minimum sınırlayıcı kutusunun sol alt köşesinde ve maksimum sınırlayıcı kutusunun sağ üst köşesindeki değerler kılavuz hücrelerin sınırlarını gösterir. Üçgen sınırlayıcı kutusunun kapsadığı kılavuz hücreleri şunlardır:

$$Hücre_x = \left[\left(\frac{Minimum\ Sınırlayıcı\ Kutu}{Hücre\ Boyutu_x} \right), \left(\frac{Maksimum\ Sınırlayıcı\ Kutu}{Hücre\ Boyutu_x} \right) \right]$$
$$Hücre_y = \left[\left(\frac{Minimum\ Sınırlayıcı\ Kutu}{Hücre\ Boyutu_y} \right), \left(\frac{Maksimum\ Sınırlayıcı\ Kutu}{Hücre\ Boyutu_y} \right) \right] \quad (5.2)$$

Hücre boyutu$_x$ ve hücre boyutu$_y$ aralığı hücreleri kapsayan xve y koordinat yönleri üzerine sıralanır. Tüm hücreleri kapsayan hücrelerin tam listesini aşağıdaki formül ile ifade edilebiliriz:

$$Hücre_{listesi} = [\ (Hücre_x[0], Hücre_y[0]), \dots, (Hücre_x[son\ k.], Hücre_y[son\ k.])]. \quad (5.3)$$

5.1.2. Nokta Veri Yapılarının Ayrıntıları

2D homojen gridlerdeki nokta verilerinin hızlandırılmış veri yapısının inşası önceki konuda detaylıca incelenmişti. Burada ise nokta verilerinin GPU bellekte nasıl hafızaya alındığına ilişkin bazı detaylar anlatılmıştır.

Şekil 5.3'de nokta düzeyinde grid veri yapısı gösterilmektedir. Buradaki *P* noktayı, yanındaki sayı ise noktanın indeksini belirtir; örneğin *P38* bu noktanın *38***.** indekste olduğunu gösterir. **C** ise hücreyi gösterir, yanındaki sayı ise indeksini verir: Örneğin *C0* bize *0***.** indeksteki hücreyi gösterir. Gridin ilk hücresine noktanın yerleştirilmesi ile veri yapısı başlar ve onları diğer noktaların ikinci hücreye ait noktaları takip ederek gridin son hücresine noktaları yükleyene kadar devam eder. Nokta dağılımının düzensizliği nedeniyle hücre boyutları farklı olma eğilimindedir [21-23].

C0		**C1**					**C2**			**C3**	
P38	**P3**	**P15**	**P22**	**P33**	**P26**	**P10**	**P6**	**P20**	**P40**	**P17**	**P30**
C3					**C4**						**C5**
P13	P34	P8	P5	P43	P46	P0	P24	P2	P29	P45	P12
C6				**C7**			**C8**	**C9**		**C10**	
P44	P25	P1	P31	P18	P11	P41	P46	P16	P0	P36	P47
C11			**C12**		**C13**	**C14**			**C15**		
P19	P7	P35	P23	P14	P32	P4	P28	P39	P9	P42	P21

Şekil 5.3. 2D Homojen gridlerdeki nokta verileri [21].

Ayrılan bellek bölgesi kavramsal olarak farklı hücre bölümlerine ayrılmış olmasına rağmen veri iyice sıkıştırılmış bir şekilde düzenlenmeli ki ilave fiziksel hafıza alanı gereksinimi olmasın. Şekil 5.3'de yalnızca önceki satırlar lineer belleğin devamlılığını temsil etmektedir. Bütün veri yapısı aslında sürekli bir lineer yığın bellek içinde depolanmaktadır. Örnek verecek olursak, şekilde gösterildiği gibi, P43 ve P13 noktalarının her ikisi de C3 hücresine aittir ve P13 noktası bellek adres alanında P30 noktasının yanında durmaktadır.

Belirli bir hücre ya da hücre parçasına erişimi hızlandırmak için gerekli olan veri farklı gridlerdeki hücrelerin başlangıç ve bitiş indeksleridir. Şekil 5.4'deki iki indeks dizisi önceki noktanın veri yapısı şekli ile ilişkili olduğunu gösterir. Bu çalışmada bunlar hücre başlangıç indeksleri dizisi ve hücre bitiş indeksleri dizisi olarak adlandırılmıştır. Hücre başlangıç indeksleri hücre dizilerine yüklenen ilk noktadır. Hücre bitiş indeksleri dizisi ise farklı hücrelerin özel bitiş konumlarını içerir. Yani bir hücrenin bitiş indeksi bir sonraki hücrenin başlangıç indeksidir. Örneğin; şekil 5.4'de gösterilen *C0* hücresinin bitiş indeksi 2'dir. Bu aynı zamanda *C1* hücresinin başlangıç indeksidir. Bu şekilde hücre bitiş indeksi daha kolay bir şekilde elde edilmiş olur.

Hücre başlangıç indekseleri

C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
0	2	7	10	17	23	24	28	31	32	34	36	39	41	42	45

Hücre bitiş indekseleri

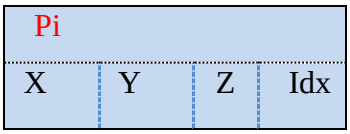
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
2	7	10	17	23	24	28	31	32	34	36	39	41	42	45	48

Şekil 5.4. Hücre başlangıç ve bitiş indeksleri [21].

Her noktanın x, y ve z olmak üzere üç tane kayan koordinat noktası vardır. Muhtemel nokta verilerinin bir birleşimi *float3* veri tipinde saklanır fakat bunun dezavantajı *float3*'ün vektör tipinde yapılandırılmasına karşı CUDA'da *float3* dizin yapısında yapılandırılmaktır. Bunun anlamı bir iş parçacığı bir *float3* yapısındaki veriyi okuduğunda, üç bellekteki hareket herhangi bir *float*'a ulaşmak için bir seferde

üç farklı dizi içine birbiri ardına çağrılarda bulunur. Bu şekilde birleştirme korunmasına rağmen, sadece 64 bayt bellek hareketi (iş parçacığı başına 4 bayt) verilir ve bellek bant genişliği iyi bir şekilde kullanılamamış olur [19].

Alternatif olarak bir dördüncü bileşen dolgusunu ve nokta verileri depolayabilen *float4* türü seçilmiştir. Bir iş parçacığı tarafından bir *float4*'e erişilmeye başladığı zaman 128 bayt birleştirme bellek hareketi gerçekleştirilir. Bu iki yapı, yani *float3* ile *float4* karşılaştırıldığında *float4*'ün daha iyi bant genişliği kullanımı sağladığı görülür.



Şekil 5.5. *Float4* nokta veri yapısı.

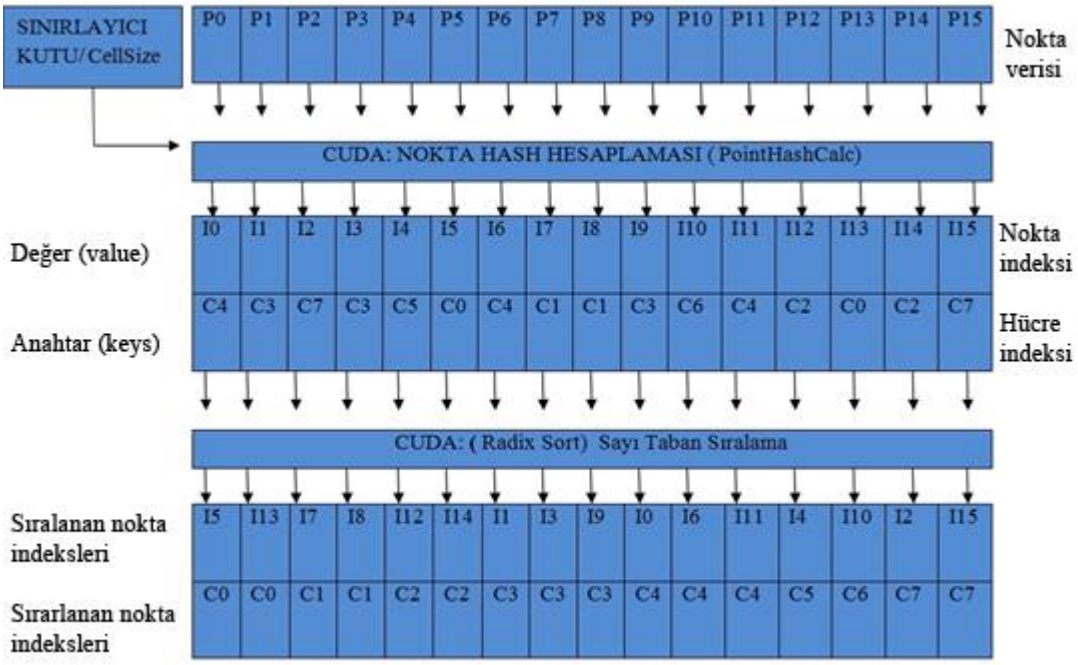
Şekil 5.5'de *float4* nokta veri yapısını örneklenmektedir. Performansı dışında dördüncü bileşen bu çalışmada pikselleştirme aşamasında ihtiyaç duyulan bir bileşendir. Gruplama sürecinden sonra nokta verileri tekrar sıralanır. 1991 yılında Heidmann tarafından açıklanan gölge stensil tamponundaki (stencil buffer) tarama çıktıları, ekran alanı içindeki piksellerde olduğu gibi hücre başlangıç ve bitiş indeks yöntemini uygular. Yeniden yapılandırılan örnek noktalarının gerçekteki orijinal indekslerinin izini bulmak için gerçekteki orijinal ekran alanının indeksini içeren ve dördüncü bir bileşen yapısına sahip *float4* veri yapısı kullanılmıştır. Bu veri yapısı sayesinde dördüncü bileşen verisi olarak vektör verisi de eklenerek adresleme sorunu da *float4* ile giderilmiştir [21].

5.2. NOKTALARIN GRUPLANMASI

Düzensiz örnek noktalarının gruplamasını gerçekleştirmek için iki yöntem incelenmiştir. Bunlar 2D sayı tabanlı sıralamaya (radix sort) dayalı gruplama ve atomik işlem tabanlı gruplamalardır.

5.2.1. 2D Sayı Tabanlı Sıralamaya (Radix Sort) Dayalı Gruplama

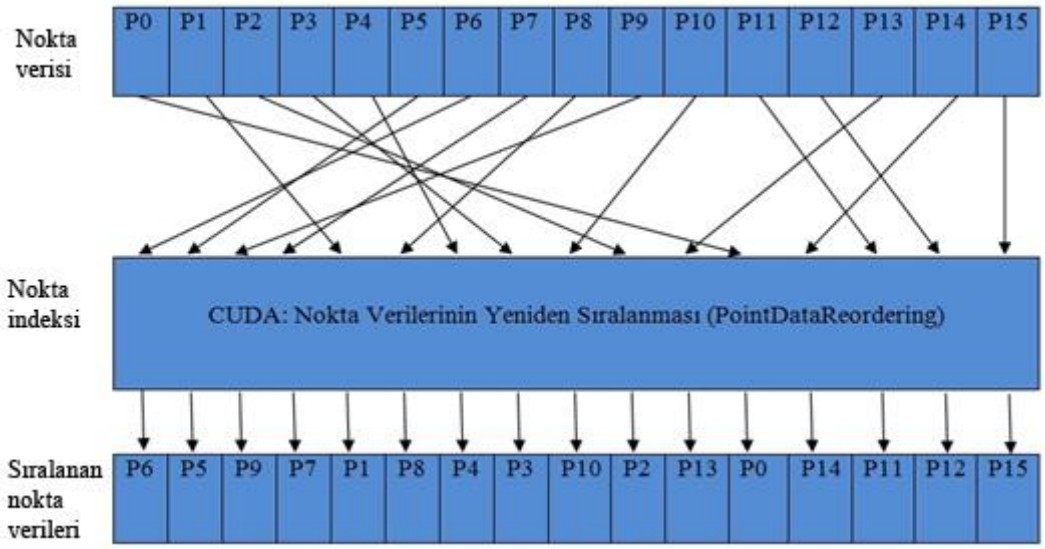
İlk olarak GPU üzerinde çalışan sayı tabanlı sıralı **(**radix sort) gruplama yöntemi 2008 yılında Sengupta ve arkadaşları tarafından açıklanmıştır. Şekil 5.6'da hash hesaplaması ve verinin yeniden sıralanması olarak adlandırılan sürecin ilk bölümü açıklanmıştır. Hash hesaplaması ve verinin yeniden sıralamasında her bir noktaya ait olan hücre indeksi hesaplanır ve nokta verileri buna uygun olarak yeniden düzenlenir [14].



Şekil 5.6. Sayı tabanlı sıralı (radix sort) gruplama I [14].

PointHashCalc (nokta hash değeri hesaplama anlamına gelir), her iş parçacığı bir nokta çeker ve örnek noktalarının sınırlayıcı kutusuna ait hücre indeksini hesaplar ve grid hücresinin boyutlarını sanal sahneden alarak oluşturur. Hash değeri esas olarak lineer hücre indeksi *unsigned int* (işaretsiz sayı) olarak depolanır. Taşma (Overflow), sınır değerlerinin sıkıştırılması ile engellenir. Hesaplanan hücre indeksleri dizisi radix sort içindeki anahtar veri olarak işlem görür. Orijinal nokta indeksleri dizisi ayrıca oluşturulur ve ardından radix sort değer olarak kullanılır.

Sıralama işleminden sonra nokta indeksleri radix sort çekirdeği tarafından karışık şekilde yürütülür. Bu yol ile hücre indeks dizisi sıralanarak aynı hizaya getirilir. Şekil 5.7'de gösterildiği gibi, bir nokta verisi yeniden sıralama çekirdeğine yeniden düzenlemek için çağrılır. Sıralanan nokta indeksleri tarafından nokta verileri toplanmıştır ve daha sonra bütünleşmiş bir biçimde genel belleğe tekrar yüklenmiştir.



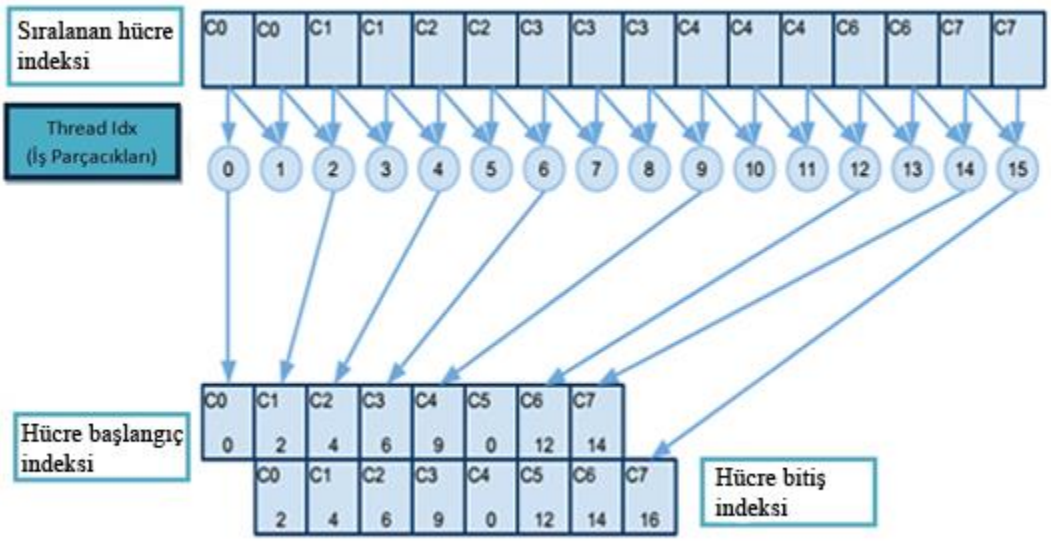
Şekil 5.7. Sayı taban sıralı (radix sort) gruplama II [14].

Sıralamadan sonra nokta indeksleri radix sort çekirdeği tarafından karışık şekilde yürütülür. Bu yol ile hücre indeks dizisi sıralanarak aynı hizaya getirilir. Şekil 5.7'de gösterildiği gibi, bir nokta verisi yeniden sıralama çekirdeğine önceki bölümde açıklandığı şekilde istenilen biçim içine yeniden düzenlemek için çağrılır. Sıralanan nokta indeksleri tarafından belirtilen çeşitli yerlerden nokta verilerini toplanmıştır ve daha sonra bütünleşmiş bir biçimde genel belleğe tekrar yüklenmiştir.

Kullanılan radix sort uygulaması sadece 32-bitlik adres yapılarını desteklediğinden, bir ara veri yapısı olarak nokta indeks dizisi kullanılır ve ekstra verilerle yeniden sıralama yapılır. Ancak radix sort yürütme zamanı ile karşılaştırıldığında, çekirdek belleğinin hala küçük bir bölümünün kullanıldığı görülmüştür. Bu da daha fazla alan ihtiyacını ortadan kaldırmış olur.

Hücre İndeksini Oluşturma

Daha önceki bölümde belirtilen hücre başlangıç ve bitiş indeksleri adres belleklerindeki çapraz geçişlerde kullanılır.

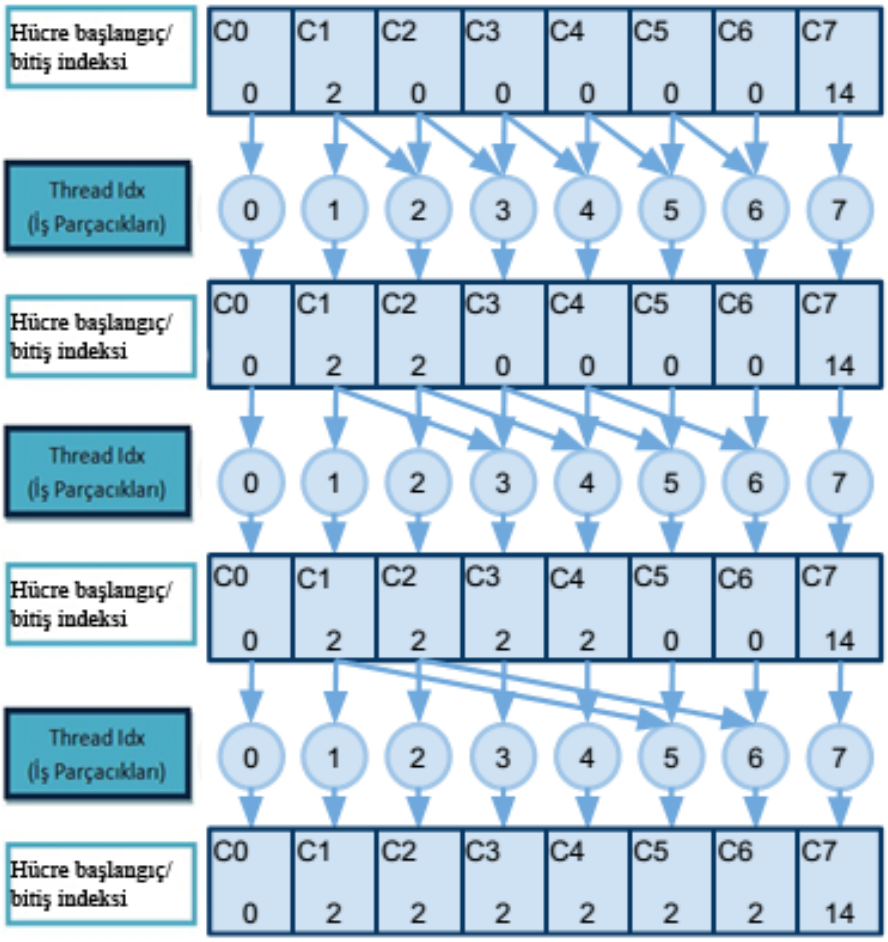


Şekil 5.8. Hücre başlangıç ve bitiş indekslerinin oluşturulması [21].

Şekil 5.8'de gösterildiği gibi 0 numaralı ilk hücre hariç sıralama kriteri her iş parçacığı tarafından hücre indekslerini yüklenir. Sonra iki hücre indeksi karşılaştırılır ve eğer arada fark varsa, karşılık gelen hücre başlangıç ve bitiş indeksleri iş parçacığının indeksi ile güncellenir.

Prefix fill (Çağrı İşaretçisinin Verisinin Yüklenmesi)

Eğer bir hücrenin boş olduğu fark edilirse hiçbir noktanın bu hücreye taşınmadığı anlamına gelir. Şekil 5.8'de görüldüğü gibi 5 numaralı hücreye her hangi bir ok inmemiştir. Bu şekilde 5 numaralı hücre boş olarak işaretlenmiştir. Bu yolla hücrenin başlangıç ve bitiş indekslerini oluşturmada hata alınır. Bunu düzeltmek için boş hücre indeksleri doldurulana kadar özel bir prefix tarama (çağrı işaretçisinin verisinin yüklenmesi) fonksiyonu eklenir. Prefix fill fonksiyonu şekil 5.9'da detaylıca gösterilmiştir.



Şekil 5.9. Hücre başlangıç ve bitiş indekslerinin doldurulması I [21].

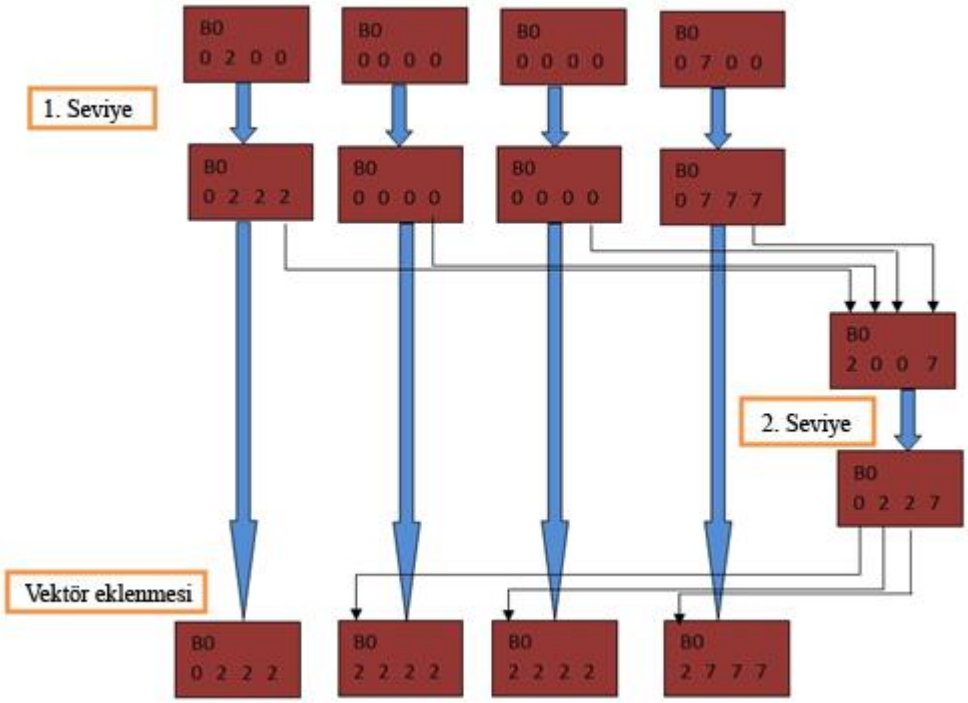
Bir prefix toplamı için ikili tabanda veri ekleme işlemi prefix fill yer değiştirme fonksiyonu ile yapılır. Bu fonksiyon aşağıda gösterilmiştir;

1: if cellStartIdx[threadIdx] = 0 then
2: cellStartIdx[threadIdx] = cellStartIdx[threadIdx - 1]
3: end if

Eleman alanı içindeki değer 0 ya da başka bir başlangıç değerleri alır ise boş olarak işaretlenir. İkili işlem süreci bir önceki işlemde elde edilen değerle güncellenir. GPU bellekte bu tarama işlemi sorunsuz işlemektedir.

Bunun dışında, *float4* veri yapısındaki vektör ekleme işlemi prefix fill - tarama ile tam olarak aynı yolu izler. Bundan dolayı bu işlem de üç adımı içermektedir.

Çekirdeğin her bir blok içinde segment taramasına ek olarak tüm bloklar ve eklenen vektörler üzerinde recursive tarama yapılır.



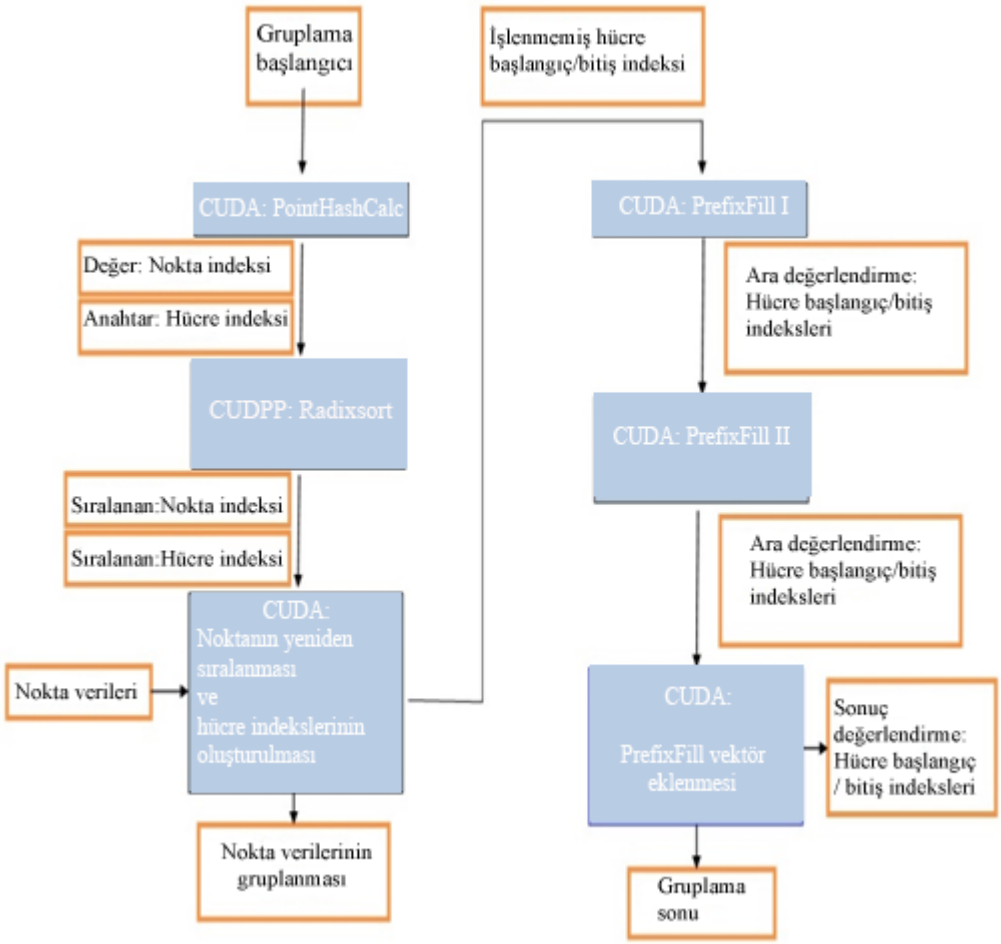
Şekil 5.10. Hücre başlangıç ve bitiş indekslerinin doldurulması II [21].

Şekil 5.10'da gösterildiği gibi kısmi tarama çıktılarını genel tarama işleminde kullanmak için ikinci seviye gereklidir. İkinci seviyedeki vektör ekleme işlemi ise karşılaştırmada kullanmak için uygulanır. Tüm seviyelerde hücre boş ise dolgu (fill) fonksiyonu uygulanır.

2008 yılında Sengupta ve arkadaşlarının yaptığı çalışmalarda prefix fill küçük bir veri kümesi üzerine uygulanmıştır (grid genişliği x grid yüksekliği= 128 X 64 tasarım). Bu yapılan işlem radix sort gruplama işlemine engel olmamaktadır. Hücrenin başlangıç ve bitiş indeks dizilerinin önceki resim kareleri ile uyumsuzlukları önlemek için her resim karesine bu işlemi uygulamak gereklidir.

Sonuç

Şekil 5.11'de GPU çekirdeği üzerinde radix sort gruplanması gösterilmektedir. CUDPP ile radix sort tek bir rutin olarak sıralama işlemi yapılmıştır. Sıralama yapıldıktan sonra, GPU çekirdeği içindeki indeksleri düzeltmek için çağrılan prefix fill rutini, geçici hücrenin başlangıç ve bitiş indeks dizileri ile yeniden sıralanmış nokta verilerinin birleştirme işlemi yapılmıştır.



Şekil 5.11. Sayı tabanlı sıralı (radix sort) nokta gruplama [21].

Radix sort gruplama performansı Çizelge 5.1'de özetlenmiştir. GPU çekirdeğindeki yürütme süreleri NVIDIA CUDA görsel profiller tarafından ölçülen GPU zamanlayıcısı tarafından alınmıştır. Çözünürlük 512 x 512 olarak ayarlandığından dolayı örnek noktaların veri boyutu 250,000 olmuştur. Bu radix sort gruplama yönteminin performansı örnek noktalarının veri boyutuna göre belirlenir.

Çizelge 5.1. Sayı tabanlı sıralı (radix sort) gruplama performans özeti.

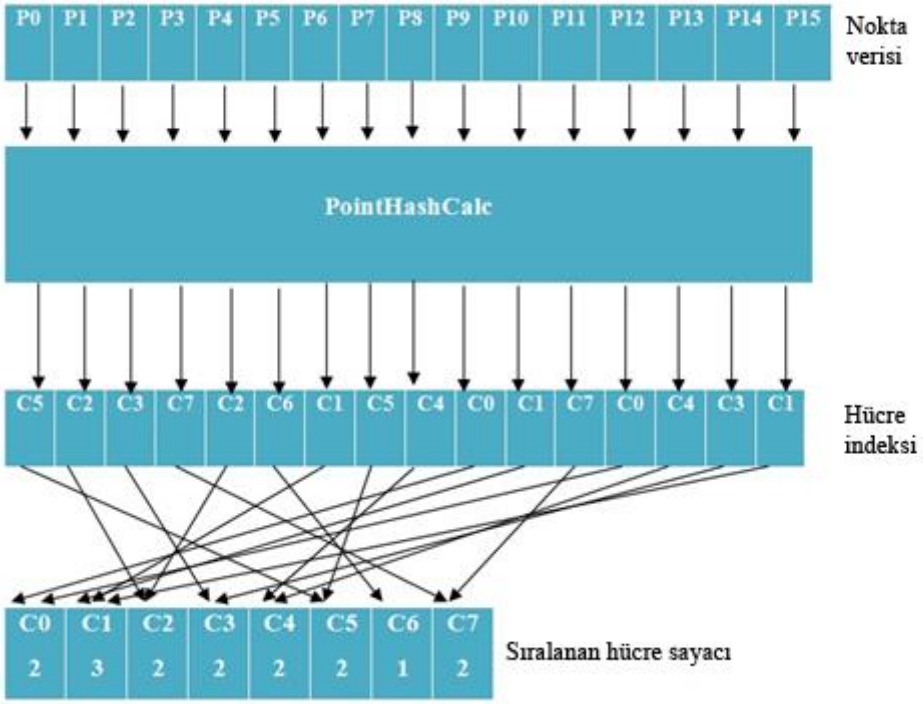
Yöntem	**Radix Sort Gruplama**	
Madde	**Rutin İşlemler/ Çekirdek**	**Süre(µs)**
1	PointHashCalc	244.35
2	CUDPP: RadixSort	1266.43
3	Noktanın Yeniden Sıralanması & Hücre İndekslerinin Oluşturulması	121.63
4	PrefixFill I	12.74
5	PrefixFill II	9.12
6	PrefixFill Vektör Eklenmesi	8.80
Toplam		1663.07

5.2.2. Atomik İşlem Tabanlı Gruplama

İkinci nokta gruplama yöntemi ise 2011 yılında Balfour tarafından GPU çekirdeğinde CUDA'nın desteklediği atomik işlem tabanlı gruplama yöntemidir. Bu yaklaşımın radix sort gruplamasından farklı yönleri incelenmiştir [25].

Atomik gruplama yöntemini sadece genel bellekle ilgili atomik işlemlere uygulamak oldukça kolaydır.

Radix sort yaklaşımında uygulanan yöntemin aynısı atomik işlemlerde kullanılır. Atomik işlemlerde bulunduğu noktanın hücre indeksini belirli iş parçacıkları belirli nokta verilerini getirip ve bu nokta verilerinin hash değerini hesaplanır. Şekil 5.12'de gösterildiği gibi her sahneye ait tüm grid hücrelerinde kaç tane nokta olduğu sayılır. Her hücrenin gruplanan nokta sayısının genel bellekteki iz genişliğini elde etmek için bir hücre sayaç dizisi muhafaza edilir.

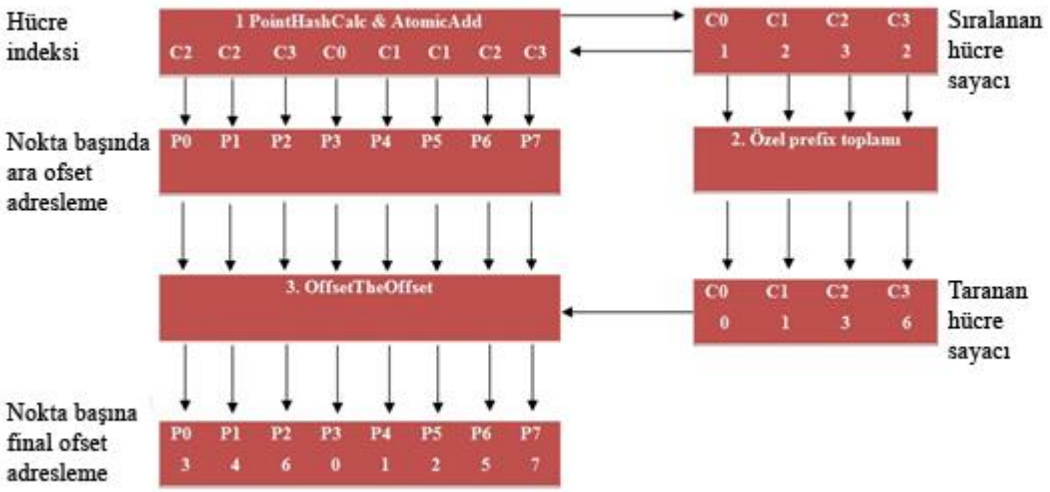


Şekil 5.12. Atomik gruplama I [21].

Hücre sayacı her bir hücrenin değerini artırmak için her iş parçacığında kullanılmak üzere *atomicAdd* fonksiyonu çağırılır. Tespit edilen noktanın ait olduğu hücreye gönderilmesi için işaret bayrağı konulur. Birden çok iş parçacığına aynı hücre sayacı erişirken, senkronizasyonlar uygulanması gerekir. Tüm atomik işlemler tamamlandıktan sonra her hücreye gönderilen noktaların sayısını içeren bir hücre sayacı dizisi ile sahne sayacı bitirilir. Bu sayede, nokta verilere dokunulmamış olur. Nokta verilerine dokunulmadığı için de parça parça yeniden tahsis edilmesine ilişkin bilgiyi sayaç dizisine eklemek gerekir. Bu şekilde atomik gruplama işlemi iki aşamada yapılabilir.

İlk aşama olarak hücreye başlangıç indeksleri dizisinden elde edilen hücre sayaçlarına özel bir prefix (dolgu) toplamı uygulanmıştır. Ayrıca hücre başlangıç indeksleri dizisi Şekil 5.12'de gösterilen grid yapısı içindeki her bir hücre için başlangıç noktalarının bilgisini elde etmeye yarar. Her bir noktanın kendine özgü benzersiz bir konuma sahip olmasından dolayı noktaların sıralama işleminin yapılması gerekmektedir. Bu sıralama işlemi atomik işlemlerin ikinci aşamasında oluşturulur.

İkinci aşama olarak da Şekil 5.13'de gösterildiği gibi ayrı bir dizi sayacı ile her atomik işlemden dönen her bir nokta için benzersiz bir iç ofset adres değeri kaydedilir. Her nokta başına gelen final ofset adres değeri, hücre başlangıç indeksine karşılık gelen her bir nokta için hücre içi ofset adresi tarafından üretilen ofsetlerdir.



Şekil 5.13. Atomik gruplama II [21].

Şekil 5.13'de gösterildiği gibi nokta dizisi başına ara ofset adresi yapılandırılmıştır. Her nokta için hücre ara ofset adresi ve hücre indeksi 16 bitlik adreslerde kodlanır. Bu her iki değer de 32 bitlik *unsigned int* veri tip şeklinde tanımlanır. Bu güvenli bir yapı oluşturur. Çünkü hücre indeksinin maksimum değeri olarak 8191'i veren 128 X 64'lük grid boyutudur. Yapılan testlerde karşılaşılacak en büyük veri yapılarında bile hücre başına düşen maksimum nokta sayısı 10000'den fazla olmayacağıdır. Bu nedenle, 16-bitlik (0 ile 65535, *unsigned int*) veri tipi, ofset adres ve hücre indeks değerlerini yerleştirmek için yeterli olacaktır. Bunu yaparak prefix toplam fonksiyonundan sonra şekil 5.13'deki 3 adımda gösterilen *OffsetTheOffset* fonksiyonu çağrılarak tanımlama işlemi yapılır. Bu şekilde ofset adres ve hücre indeks verilerine daha verimli bir şekilde erişilir ve gereksiz hesaplamalar ile zaman harcanmaz [26].

Atomik gruplama işleminin son adımı olarak nokta verileri yeniden sıralanmalıdır. Her bir nokta için final ofset adresinde dağınık bir şekilde bulunan nokta verilerinin yeniden yapılandırma işlemine geçilir. Bundan sonra da nokta verileri istenilen

şekilde düzenlenecek ve pikselleştirilme işlemine geçilirken hücre başlangıç indeksi ile hücre bitiş indeks dizileri hazır hale gelecektir.

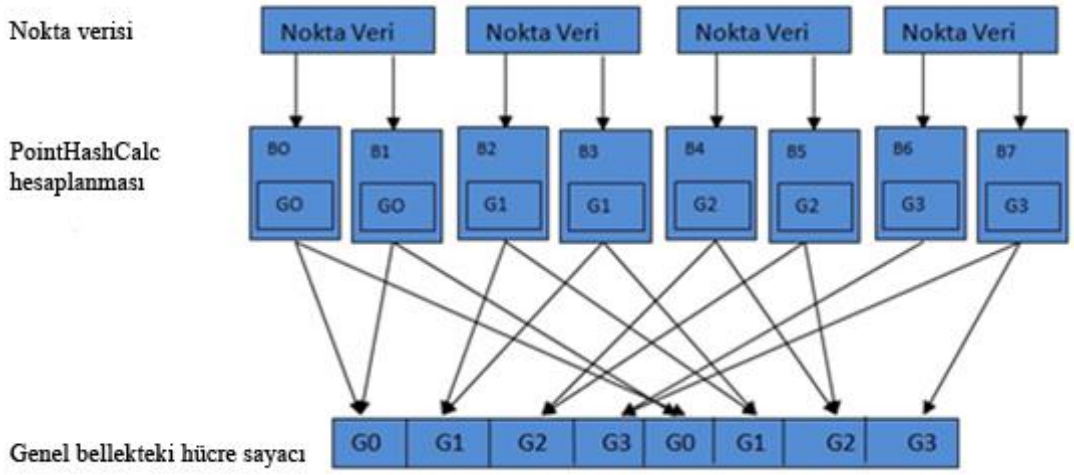
Bu yaklaşımın performansı genel bellek üzerinde yapılan çok sayıdaki atomik işlem nedeniyle bir yığılmaya neden olmaktadır. Bir örnek verecek olursak: 256x256 çözünürlükte elli binin üzerinde atomik işleminin CUDA çekirdeği tarafından yapılması demektir. Bu şekilde atomik işlemler büyük bir performans kaybı oluşturur. Bu performans kaybını önlemek için aşağıda açıklanan yöntemler incelenmiştir.

Atomik İşlemlerde Performans Artırma Yöntemi

Atomik işlemlerdeki performans kaybının nedeni CUDA'nın paylaşımlı belleğe erişiminin genel belleğe erişim hızına oranla daha büyük olmasıdır. Performansı artıracak eylem olarak paylaşımlı bellek üzerinden ilk atomik işlemleri gerçekleştirmek ve sonra ikinci aşamada genel bellek üzerinde işlemleri birleştirmektir.

Çoklu işlemcilerin veri akışındaki SM'nin paylaşımlı bellekte muhafaza edilmesinin bellek boyutu 16KB ile sınırlı olması engel olarak karşımıza çıkmıştır. Bir grid yapısı 128x64 = 8,192 boyutlarındadır. Buda 8KB'lık bir alan demektir. 4-Byte'lık *unsigned ints* veri tipi hesaplandığında 8Kx4B = 32 KB'lık bir alana karşılık gelmektedir. Bu nedenle bir SM'nin içine hücre sayacının bütün kopyasını bir tampon olarak yüklemek mümkün değildir. Bundan dolayı *unsigned ints* veri tipini 4 gruba ayırmak gerekmektedir. Bu da her SM başına 8 KB'lık veri akışı sağlayacaktır [26].

Şekil 5.14'de atomik gruplama işleminin özelleştirilmiş hücre sayacı gösterilmiştir. Başlatılan iş parçacık blokları 4 gruba ayrılır. Her bir gruptan sorumlu olan hücre sayacı dizisi için özel bir alt küme vardır. Daha iyi bir yük dengeleme elde etmenin yolu hücre sayaçlarını serpiştirerek farklı gruplara atamaktır.



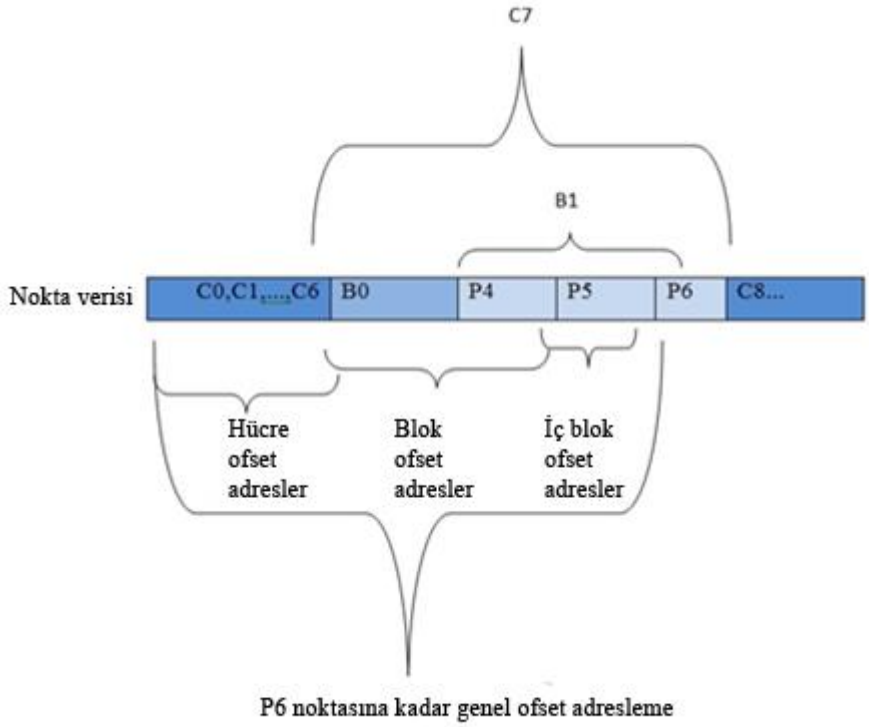
Şekil 5.14. Paylaşılmış bellek üzerinde atomik sayaç [26] .

Gruba atanan nokta verisi hücre içine düştüğünde, şekil 5.14'de görüldüğü gibi her grubun tüm veri noktalarının üzerinden geçmek gerekir. Bu şekilde *atomicAdd* sadece paylaşılan bellek üzerinde gerçekleştirilmiş olur. Her bir grup için sorumlu olduğu noktaya ait dağıtım bilgisi atomik işlemler yoluyla toplanır. Tüm farklı gruplardaki hücre sayacı dizisi tamamlandığında genel belleğe çağrılarak birleştirilir. Bu şekilde veri bütünlüğü korunmuş olur.

Şekil 5.14'de görülebileceği gibi genel bellek üzerinde gerçekleştirilmesi gereken atomik işlemlerin toplam sayısı, üretilen grup başına blok sayısı ve hücre sayaç dizi boyutunun çarpımıdır. Bu da gerekli atomik işlemlerin sayısının ekran çözünürlüğüne bağlı olmadığını gösterir.

Yukarıda belirtildiği gibi her grup başına düşen iş parçacığı, daha az atomik işlemler uygulanarak, blok sayısı genel bellek üzerinde yapılması gerekir. Öte yandan tüm örnek noktalarının her bir grup ile işlenmesinin anlamı, çeyrek milyon noktanın aynı gruba ait benzer parçacık blokları arasında paralelleştirilmesidir. Bu nedenle, grup boyutu daha büyük olan her blok, daha az nokta işlemi ile karşılaşır. Bu sorun daha büyük ve daha küçük grup boyutu arasında bir değiş tokuş yapılarak bu sorun dengeleme yolu ile aşılır. Farklı parametre kombinasyonları ile bir dizi test sonrasında çıkan sonuçlarda 4 grup ile 32 blok elde edilir. Böylece her grup başına en iyi performansın sağlandığı bulunmuştur.

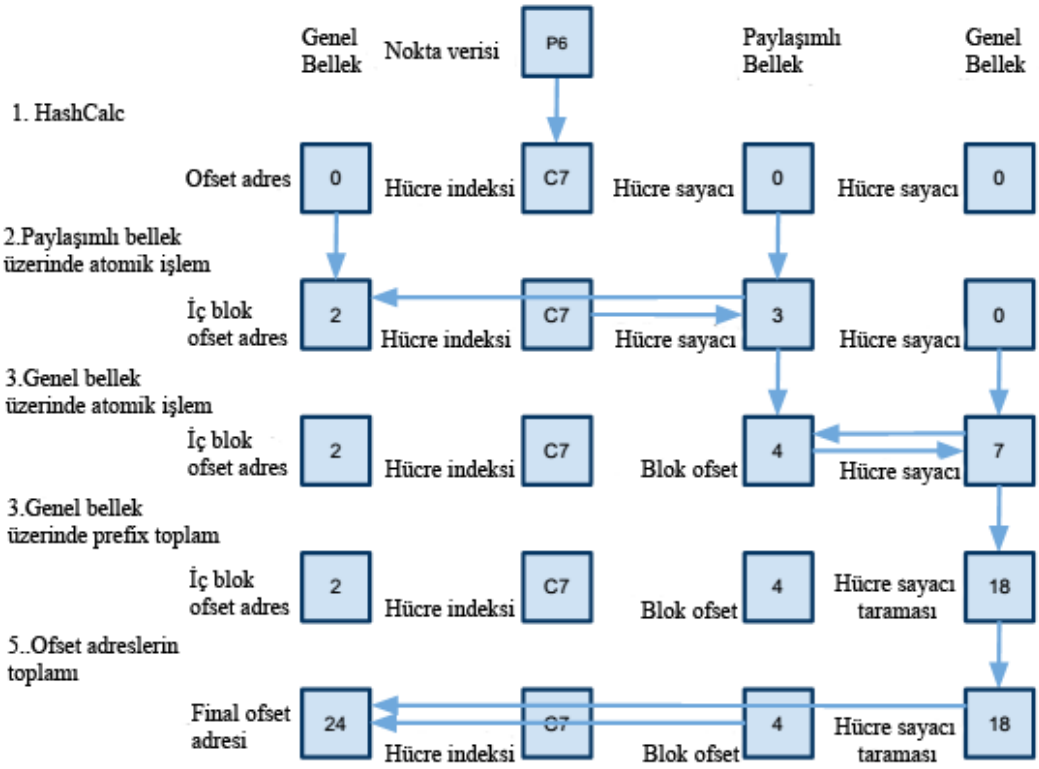
Şekil 5.15'de gösterildiği gibi, hangi gruba ait olup olmadığına bakılmaksızın nokta numarası 6 olan nokta P6, blok numarası 1 olan blok B1 ve hücre numarası 7 olan C7 olarak gösterilmiştir. P6 noktası C7 hücresi içinde gruplanmıştır. P6 için istenilen yeniden yapılandırma pozisyonu şekilde gösterilmiştir. Genel ofset o_{global}^{i} üç bölümden oluşmaktadır: hücre ofset adresi C7, C7'nin içerisindeki blok ofset adresi B1 ve P6 içinde özel olarak iç blok nokta ofset adresi yapılandırılıyor. Böylece, her bir örnek noktası için p^{i}, genel ofset o_{global}^{i} olarak ifade edilebilir:



Şekil 5.15. Örnek noktanın ofset adresleme yöntemi [21].

$$o_{global}^{i} = o_{hücre}^{i} + o_{blok}^{i} + o_{iç\ blok}^{i} \tag{5.4}$$

Her üç ofset bileşenini elde etmek için Şekil 5.16'da görüldüğü gibi her bir noktanın ofset adresini doğru hesaplama aşamaları beş adımda gösterilmektedir ve bu adımlar aşağıdaki gibi açıklanmıştır:



Şekil 5.16. Örnek nokta ofsetinin hesaplanması [21].

Şekil 5.16'daki aşamalar aşağıda açıklanmıştır.

HashCalc: Hash hesaplama hücresine ait her noktayı tanımlamak için nokta başına *HashCalc* gerçekleştirilir.

1. Paylaşımlı Bellek Üzerinde Atomik İşlemler: İlk önce hücre sayacı artışı paylaşılan bellek üzerinde gerçekleştirilir. Hangi hücre içindeki noktanın hangi blok içi ofset adrese ait olduğunun tespiti burada yapılır. Atomik işlemlerin döndüreceği değerler ve blok içi ofset değer gibi iki etiket paylaşımlı bellek üzerinde hafızaya alınır. Daha sonra hesaplanan blok iç ofsetin hash değeri (noktasının hücre indeksi) bir 32-bit *unsigned int* veri tipinde kodlanarak genel belleğe kaydedilir.

2. Genel Bellek Üzerinde Atomik İşlemler: Paylaşılan bellekteki hücre sayacının değeri genel bellek sayacı üzerinde birleştirilir ve genel bellek

sayacının eski değeri paylaşılan bellek konumundaki değer ile takas edilir. Genel bellekte döndürülen bu değer sayaç tarafından temsili hücre içindeki ofset bloğun değeridir. Son olarak da blok ofset dizisi genel bellekte kaydedilir.

3. Genel Bellek Üzerinde Prefix Toplama: Hücre başlangıç indeksleri dizisi oluşturmak için genel bellek üzerinde oluşan hücre sayacı dizisi oluşturulur.

4. Ofset Toplamı: İkinci adımda kodlanarak oluşturulan iç blok ofset dizisi genel bellekten çekilir ve çözülür. Sonra iç blok ofseti 3. adımdaki blok ofseti ve 4. adıma karşılık gelen hücre indeksi ile toplanır. Bu toplam her bir nokta için genel ofset değerini verir.

SONUÇ

Atomik işlem tabanlı gruplama performansları Çizelge 5.2'de özetlenmiştir. Buradaki sonuçlarda çekirdekteki yürütme hızları ve GPU zamanlaması CUDA Visual Profiler programı kullanılarak hesaplanmıştır. Yük dengeleme tekniklerinin uygulanır olması nedeniyle performansı artırdığını göstermiştir. Yapılan analiz 14KB üçgen içeren sahne ile test edilmiştir.

Çizelge 5.2. Atomik işlem nokta gruplanmasının performans özeti.

Yöntem	Atomik İşlem Gruplaması	
Madde	**Rutin İşlemler/ Çekirdek**	**Süre(µs)**
1	HashCalc ve Atomik Sayma	503.01
2	CUDPP:Tarama	29.66
3	Noktanın Yeniden Sıralanması	339.97
Toplam		968.67

5.3. ÜÇGEN VERİ YAPILARI

5.3.1. Veri Yapısı İncelenmesi

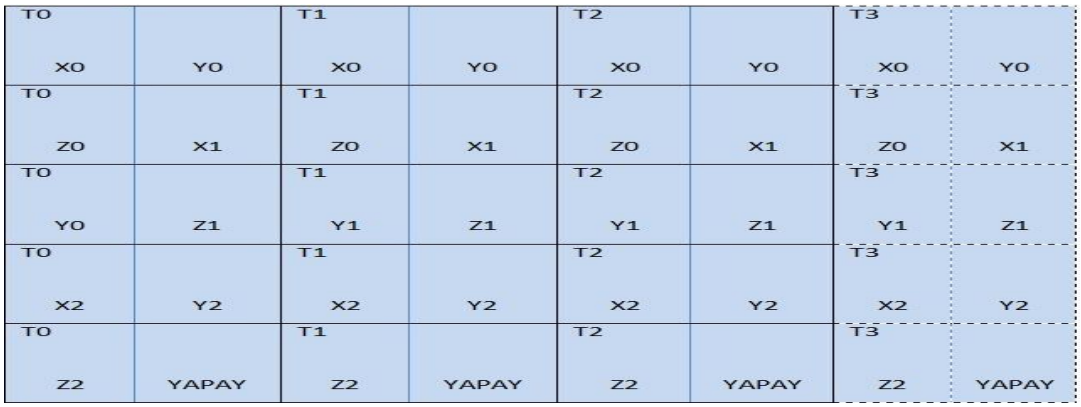
GPU'ların programlanabilir olması uygulamalarda büyük performans gelişimlerine neden olmuştur. Bu gelişim sayesinde grafik donanım üzerindeki geometrik şekillerin oluşumunu hızlandıracak veri yapılarını oluşturmak için önerilen yeni algoritmalar bulunmuştur. 2008 yılında Zhou ve arkadaşları GPU'larda ilk gerçek zamanlı kd-tree algoritması açıklamıştır. 2009 yılında Lauterbach ve arkadaşları GPU'lardaki hızlı BVH yapım yöntemlerini açıklamıştır. 2009 yılında Kalojanov ve arkadaşları gerçek zamanlı düzgün gridlerin yapımındaki algoritmaları tanımlanmıştır [27-29].

Bu yöntemler ışın izleme gibi daha gelişmiş işleme teknikleri ile oldukça iyi çalışır. GPU'larda BVH oluşturmak için yaklaşık 30 ms. sürer. Düzensiz gölge eşleme ışın izlemesinin küçültülmüş bir versiyonu olarak görülebilir olsa da bu yöntem sadece gölgelerden oluşturur. Işın izleme yöntemi ile gölgelere daha gerçekçi işleme efekti verilebilir. Bununla kalite ve hız arasında tatmin edici bir denge oluşturulması için ihtiyaç duyulan gerçek zamanlı gölge oluşturma algoritmaları ile ışın izleme algoritmaları bir araya getirilmelidir. Bu şekilde gerçek zamanlı gölge oluşturma sağlam bir konuma getirilmiştir. Söz konusu nedenden ötürü bu çalışmada düzensiz tarama mümkün olduğunca basit tutulmuş ve hatta biraz eski donanımlarla uygun olmasına dikkat edilmiştir.

5.3.2. Üçgen Veri Yapılarının Detayları

Genel bellekten üçgen verilerine erişirken daha iyi bir verim elde etmek için şekil 5.17'de gösterildiği gibi üçgen dizisi bir *float2* veri tipinde düzenlenmiştir. T0 bölgesi üçgen numarası 0 olan üçgeni gösterir. X0 ise T0 için X-koordinatındaki ilk köşeyi göstermektedir. Ele alınacak en büyük veri yığınının hangi bellek tarafından işleme alınacağı belirlenir. Bu yığın içinde her işlem parçacığı başına CUDA'daki 128-Bytelık *coalescing* bellek işlemi ile *float2* yükleme yapılır. Üçgen verileri bir araya getirilmiş *float2* dizilerinde hafızaya alınır. Üçgenin sadece dokuz kayan

noktası olduğu için son *float2* veri tipindeki üçüncü köşeyi Z-koordinatı ile beraber yapay bir kayan nokta değeri oluşturur.



Şekil 5.17. Üçgen veri yapılarının dizisi [21].

5.4. ÜÇGEN AYRIŞTIRILIP VE DERLENMESİ

5.4.1. Üçgen Ayrıştırıp ve Derlenme Nedeni

Üçgenlerin ayrıştırılmasının önemi büyüktür. Burada sadece pikselleştirilecek sahne dışındaki gereksiz hesaplamaları yapmamak için değil ayrıca yüklemedeki dengelemeye de katkı sağlamak hedeflenmektedir. Işık kaynağından bakıldığında sahnenin uzağında ayrıştırılan üçgenin arka yüzü (gölge olan kısım) ile sınırlayıcı kutuların dışında kalan örnek noktaları pikselleştirme işlemi dışında olacağından performans artacaktır. Tatmin edici performans sağlamak için, her bir üçgenin ayrıştırılması için daha fazla paralellik gerekmektedir. Yani, üçgen verileri filtrelenebilir, derlenip toparlanabildikten sonra pikselleştirilerek sahneye aktarılabilir ise, pikselleştirmede benzer üçgen üzerinde çalışan daha çok iş parçacık görevlendirilerek dinamik bir konfigürasyon oluşturulmuş olur. Bu şekilde planlanan ve başlatılacak olan istenmeyen iş parçacıkları engellenir. Bunun sonucunda pikselleştirme tarama aşamasında önemli performans geliştirmesi sağlanır.

5.4.2. Üçgen Ayrıştırılması

Bu çalışma düzensiz noktaların pikselleştirilmiş gölge bilgisinin oluşturulması için özelleştirilmiştir. Bu nedenle pikselleştirmeye veri akışını başlatmadan önce ilgisiz üçgen verilerini ayrıştırarak doğru gölge üretilmeye çalışılmıştır. Eğer sahnedeki nesne 1:1 olarak modellenirse, ışık kaynağından bakıldığında nokta sadece üçgenin ön yüzünü gösteren tıkayıcı görevini alır. Böylece üçgenin görünmeyen kısımları ayrıştırılır. Bunu yaparak, pikselleştirme sırasında işlenmesi gereken üçgen sayısı yarıya indirilebilir.

Üçgenin görünmeyen kısımlarının ayrıştırılması işlemi, kenar denklemi kurulumundan sonra üçgenin işaretlenmiş alanı değerlendirilerek yapılır. Üçgenin işaretli alanı sıfırdan küçük ise ayrıştırılır, sıfırdan küçük değilse ayrıştırılmaz. Düzensiz pikselleştirme sadece örnek noktanın sınırlayıcı kutusunun kapsadığı bölgede oluşur. Böylece sınırlama kutusunun dışında bulunan üçgenler yok sayılır ve de ayrıştırılabilir.

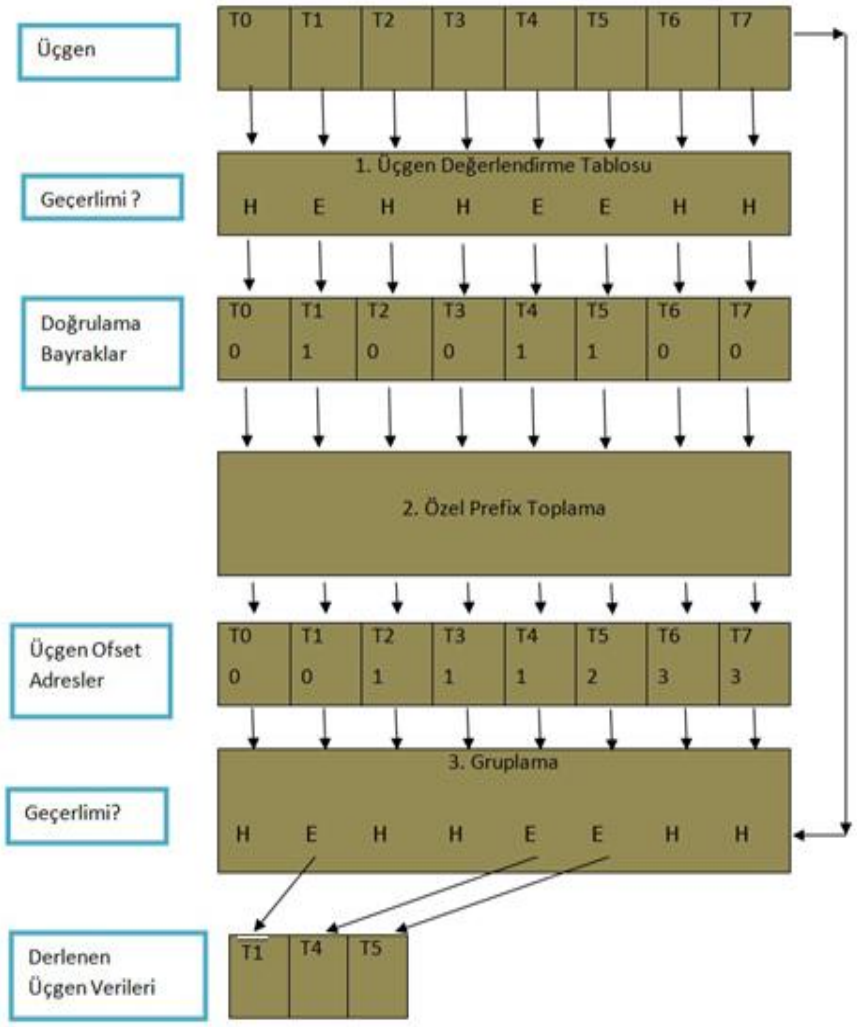
Üçgen ayrıştırılması yukarıda açıklanan ayrıştırma kurallarına uygun olarak her üçgen değerlendirilerek yapılır ve aşağıdaki Şekil 5.18'de gösterildiği gibi değerlendirme sonuçlarını göre doğrulama bayrak dizileri oluşturulur. Burada geçerli üçgen 1 ve geçersiz olanlar 0 olarak işaretlenir.

İlaveten yapılan araştırmada bu yönde gölge oluşturmak için tıkayıcılara ihtiyaç duyulduğu görülmüştür. Pikselleştirmede potansiyel tıkayıcılara üçgen verileri göndererek, alıcılardaki gölgeler ve hatta tıkayıcının kendi gölgesinden doğacak işlem hatalarından sahne muhafaza edilir. Üçgen verilerinin boyutunu küçültmek için her karenin işlenmesi gerekir.

5.4.3. Üçgen Verilerinin Derlenmesi

Derlenme işlemi prefix toplam rutini tarafından yönlendirilir. Bu aşamada CUDPP tarama kütüphane fonksiyonu kullanılmıştır. 2008 yılında Sengupta ve arkadaşları tarafından tanımlanan paralel prefix toplama algoritması CUDPP tarama kütüphanesine uygulanmıştır [14].

Üçgen sıkıştırma işlemi Şekil 5.18'de gösterildiği gibi ayrıştırma adımı tarafından oluşturulan doğrulama bayrakları dizisinde, özel prefix toplamı uygulayarak başlatılır. Prefix toplamından sonra, doğrulama bayrak dizisi üçgenler için yeniden gruplanacak ofset listeleri oluşturur. Daha sonradan yeniden gruplamada GPU çekirdeğinde tüm üçgen verilerini getirmek için çağırılır ve tekrar okunaksız olanlar değerlendirilir. Verilerin dağıtımında ofset adres tarafından gösterilen pozisyonlar geçerli bir üçgen üzerinde gerçekleşir. Şekil 5.18'de gösterilen üç adımdan sonra, üçgen veri istenilen şekilde filtre edilir ve sıkıştırılır.



Şekil 5.18. Üçgen ayrıştırılması ve derlenmesi [14].

5.4.4. SONUÇ

Şekil 5.18'de çekirdek düzeyindeki üçgen ayrıştırma ve derlenme aşaması özetlenmiştir. Görünüşe göre bu aşamanın performansı üçgen veri boyutu ve kamera bakış açısına bağlıdır. Çizelge 5.3'de gösterilen performans 14KB üçgen içeren bir model ile test sonucudur. Test edilen modelinin sayısı çok olmasına rağmen, üçgen ayrıştırması ve derlenmesi aşamasında getirdiği yük azdır.

Çizelge 5.3. Üçgen ayrıştırması ve derlenme performans özeti.

Yöntem	Atomik İşlem Gruplaması	
Madde	**Rutin İşlemler/ Çekirdek**	**Süre(µs)**
1	Ayrıştırma	19.14
2	CUDPP: Tarama	31.30
3	Yeniden Sıralanma	11.58
Toplam		62.02

BÖLÜM 6

GEOMETRİ TABANLI NESNELERİN PİKSELLEŞTİRİLMESİ

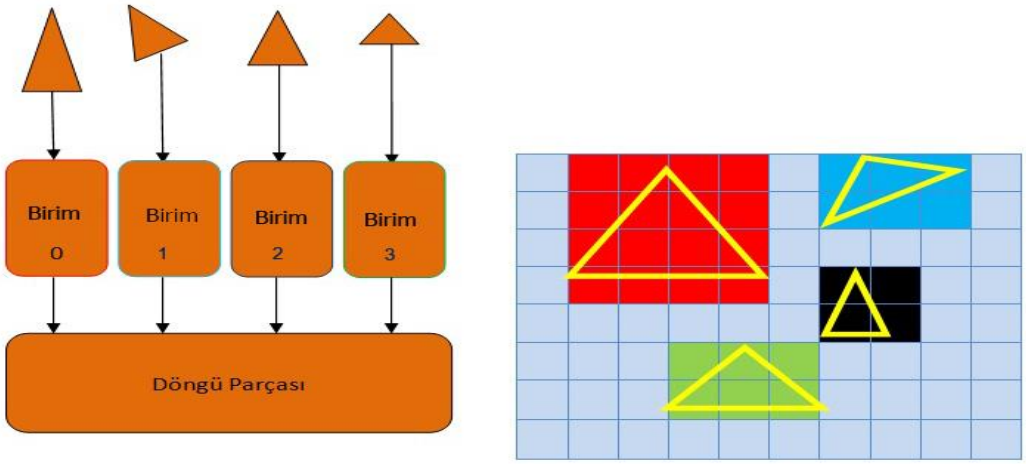
Bu bölümde iki farklı üçgen pikselleştirme algoritması tarif edilecektir. CUDA'daki iş parçacıklarını yarı-kaydırma, kaydırma ve bloklar incelenecektir. Bu çalışmada "birim" olarak adlandırılan kısımlar çalışan bir iş parçacığı grubudur. Birimin boyutu blok boyutlandırma sınırı ihlal edilmediği müddetçe herhangi bir boyutta olabilir.

Temel olarak birkaç iş parçacığı içindeki birimler kendi içinde bloklara bölünmüştür. Yani birim boyutu, birim başına düşen iş parçacığı sayısı ile blok başına birim sayısının çarpımı sonucu oluşan konfigürasyona sahiptir. Tek bir üçgenin pikselleştirme işleminin hesaplanması belirli birimler arasında paralelleştirilir. CUDA çekirdekleri farklı birim yapılandırılmaları ile daha sonra başlatılır. Böylece üçgenleri pikselleştirmek için hız açısından en iyi yapılandırmaya ulaşılmış olunur.

CUDA'da yazılacak uygulama için 2008 yılında NVIDIA CUDA C Programming Guide'da yer alan iki algoritma incelenmiştir. Birinci algoritma "Kernel1" ve ikinci algoritma "Kernel2" olarak belirtilmiştir. Kernel, belirli bir veri kümesine uygulanacak işlemleri tanımlar. Her iki algoritmanın da üçgenler üzerindeki gridlerin çapraz geçişindeki kullanımı test edilmiştir [30].

6.1. KERNEL 1

Şekil 6.1(a)'da üçgen başına bir birim temelli bir yapı kurulmuştur. Her birim üçgen veri akışından benzersiz bir üçgen getirir, kenar denklemlerini kurar ve üçgenin sınırlayıcı kutularını hesaplar. 2010 yılında Zhang ve Majdandzic'in yaptığı çalışmadasınırlayıcı kutu işleme alınması gereken noktaların sayısını azaltmak için bir hızlandırma tekniği olarak hizmet vermektedir. Şekil 6.1(b)'de ise dört farklı birim tarafından işlenen dört üçgenin kapsadığı parça bölgeleri gösterilmektedir [21].



a) Her üçgen başına bir birim

b) Her birim çapraz geçiş sınırlayıcı kutuları

Şekil 6.1. Kernel 1'in yapısı [21].

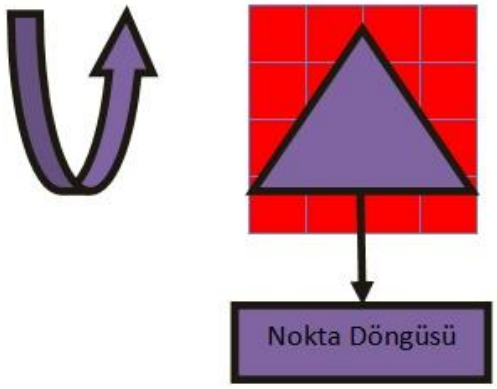
6.1.1. Döngü Parçası

Üçgenin kapsadığı bölgenin parçası üçgenin sınırlayıcı kutusuna göre hesaplanır. Bu işlemden sonra ilk döngü X-ekseni boyunca her birimde birbiri ardına iç içe geçer ve ikinci döngü olarak da Y-ekseni boyunca bölgeden parçaya doğru işleme geçer.

Döngü parçası içinde belirli bir parçaya, hücreye ait örnek noktanın ilişkili hücresinin başlangıç ve gruplama aşamasında oluşturulan son indekslerine göre erişilir. Noktalar her birim boyunca paralelleştirilir. Bu paralelleştirme işlemine göre her iş parçacığı benzer üçgenlerin oluşturacağı bölgelerdeki örnek noktalar pikselleştirmiş olacaktır. Noktanın dağılımının düzensizliğinden ötürü bazen aynı

birim içindeki tüm iş parçacıklarından yararlanılamaz. Aynı parça içinde gruplanan noktaların sayısı birimin boyutundan daha büyükse, bu parça döngüsünün yinelemesi gerekecektir.

X-ekseni boyunca, iki bitişik parçaya ait örnek noktası genel bellekte bitişik olarak yer alır. Böylece iki ayrı yineleme tarafından işlenmesi gerekmez. Başka bir açıdan bakıldığında X ekseni boyunca, bir hücrenin sonundaki indeks bir sonraki başlangıç indeksidir. Bu nedenle X-ekseni boyunca bütün parça hattına, ilk hücrenin başlangıç indeksi ve son hücrenin son indeksi referans verilerek ulaşılabilir. Diğer bir deyişle, Şekil 6.2'de gösterildiği gibi X-ekseni boyunca her parçanın hattı, büyük bir parça olarak işlem görmüş olur.

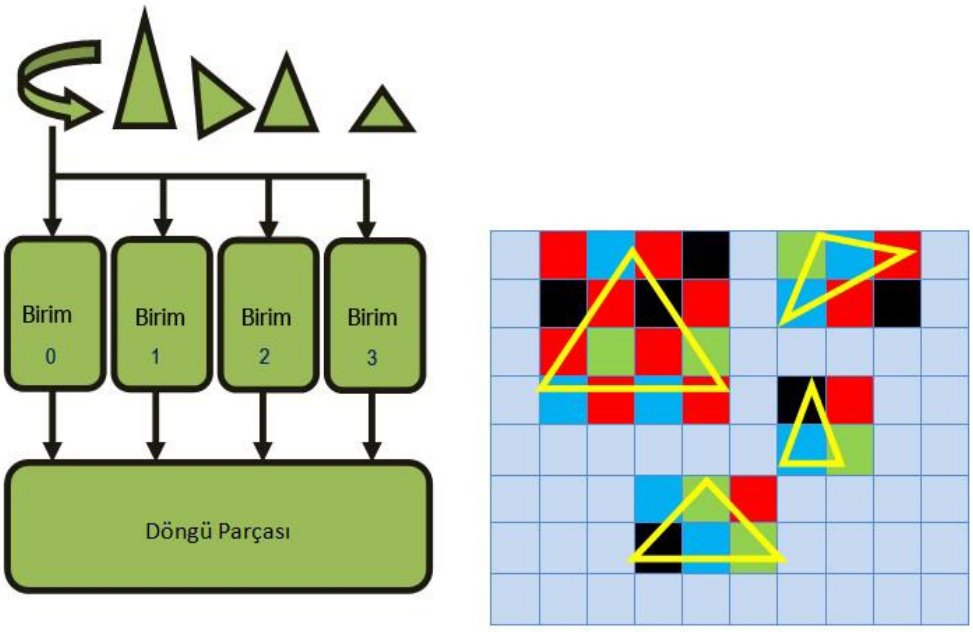


Şekil 6.2. Kernel1 döngü parçası [21].

Kernel1 parça döngüsü sadece Y ekseni boyunca gerçekleştirilir ve her parça X ekseni hattına karşı paralelleştirilir. Bu şekilde hücre başlangıç indeksiyle hücre bitiş indeks dizilerinin referansı minimize edilerek ve her bir birimden her yineleme sırasında daha iyi yararlanılmış olur. Buradaki hesaplamalar bölüm 2'deki kenar fonksiyonlarındaki hesaplamalar etrafında yapılmıştır.

6.2. KERNEL 2

Şekil 6.3 (a)'da gösterildiği gibi Kernel2 fikri, Kernel1'dan daha farklı bir yapı ortaya koymaktadır. Kernel2'de her üçgen için bir birim çalıştırılarak paralelleştirilir. Kernel1, üçgen veri akışında paralelliği sağlamaktadır. Kernel2 ise tüm örnek nokta parçalarını gruplara ayırır ve her parça grubunu belirli bir birime atar. Her birim tüm üçgenlerin üzerinde döngü oluşturur ve sadece sorumlu olduğu parçaları işler.



a) Her üçgen başına bir birim.

b) Her birim çapraz geçiş sınırlayıcı kutuları.

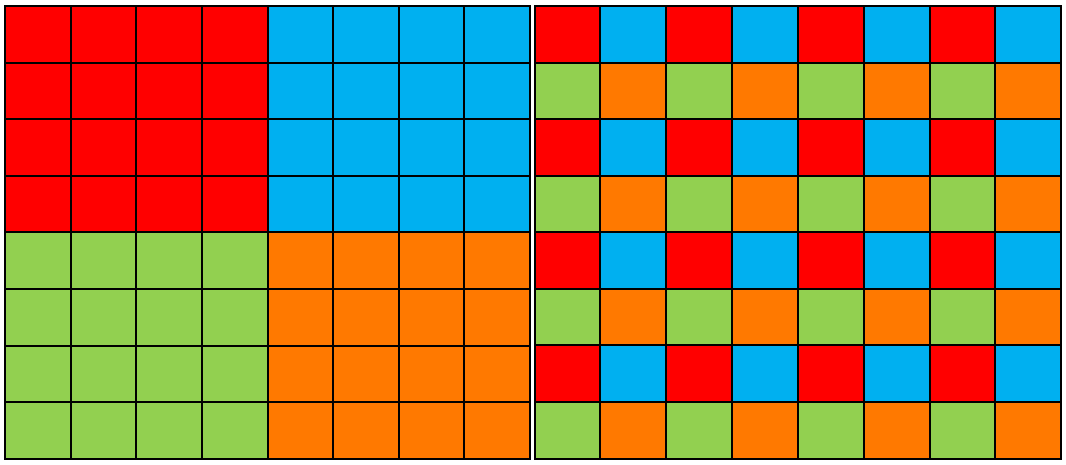
Şekil 6.3. Kernel 2'nin yapısı [21].

Şekil 6.3 (b)'de gösterildiği gibi parçalar farklı gruplara ayrılır ve her bir parçanın grup kimliğini temsil eden renk ile işaretlenir. Her birim bir döngü içindeki tüm üçgen veri akışı arasında dolaşır. Bir üçgen pikselleştirilirken, Kernel1'deki yöntemin aynısı olan sınırlayıcı kutuların kapsadığı parça bölgelerini ve üçgen sınırlayıcı kutularının hesaplanması benzer olarak yürütülür. Kernel2'deki farklı yöntem ise tüm bölge üzerinde her birim sadece kendine ait iş parçacıkları ile eşleşir. Yani Kernel2'de her üçgenin pikselleştirilmesi süreci farklı birimler arasına yayılmıştır.

Benzer üçgenler için birden çok kez sınırlayıcı kutuları ve kenar denklemleri hesaplamadan doğacak yük aşımını (*overhead*) önlemek için, üçgen verisiyle ilişkili üçgen bilgisinin ayrıştırma ve derlenme aşamasından önce hesaplanmalıdır.

6.2.1 Yük Dengeleme Şeması

Şekil 6.4'de iki farklı şemada farklı birimler arasında parçaların dağılımının nasıl olacağı gösterilmiştir. Birbirine yakın bulunan parçaların kümelendiği aşamalı şemada örnek bir sayaç çalışmaktadır. Bu şekilde, Kernel2 tasarımının amacına ulaşmasını engelleyen dengesiz üçgen dağılımının sebep olduğu farklı birimler arasındaki dengesiz iş yükü oluşur. Bu yüzden Kernel2 tasarımında dönüşümlü şema elde edilmeye çalışılmıştır. Her üçgenin kapsadığı parçaların büyüklüğü ve üçgenin konumu ne olursa olsun birimler arasında ortalama olarak dağıtılır. Parçalar dengeli bir şekilde dağıtılmış olmasına rağmen, nokta dağılımının hala dengesiz olduğunun farkında olunmalıdır.

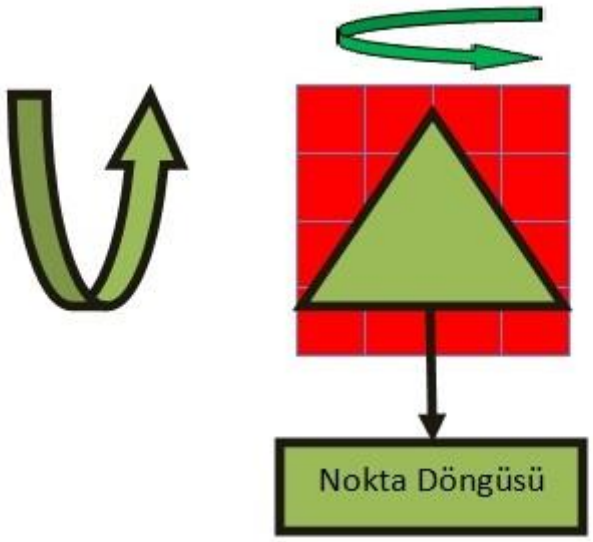


a) Aşamalı b) Dönüşümlü

Şekil 6.4. Parçaların dağılım şeması [30].

6.2.2. Döngü Parçası

Kernel2 çalışma mantığından dolayı istenen üçgen parçaları genel bellekte yan yana durmamaktadır. Şekil 6.5'de de gösterildiği gibi X ve Y ekseni her seferinde bir parçaya erişmek zorundadır.

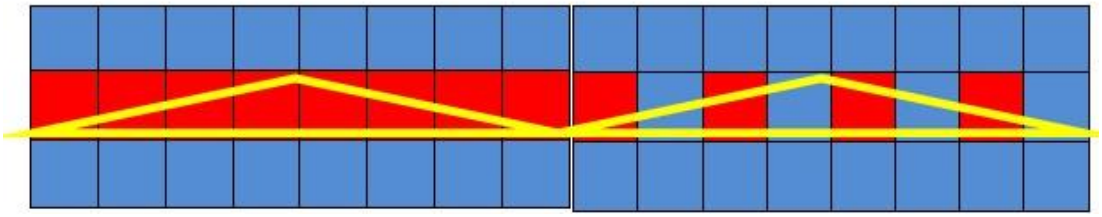


Şekil 6.5. Kernel2 döngü parçası [21].

6.2.3. Üçgenlere Kernel1 ve Kernel2'nin Uygulama Sonuçları

Kernel1 büyük çaptaki üçgenleri işlerken yük aşımı olur. Büyük üçgenler, daha fazla parçayı kapsamakta ve daha uzun parça döngüleri oluşmasına neden olmaktadır. Kernel2 tüm yonga boyunca her üçgenin iş yükünü yok etmek için statik bir çözüm sağlar ancak ek yükleri ihmal etmek gerekmektedir.

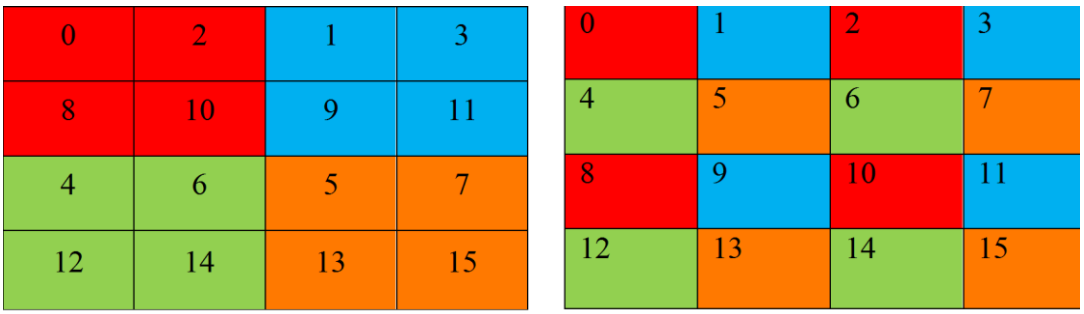
Her çekirdek işleme yöntemi ile parça döngüleri inşa edilir. Örnek noktalarının gridleri için parçalara (hücrelere) pikselleştirme esnasında başvurulur. Bu yol örnek nokta parçalarının farklı pikselleştirme çekirdekleri arasındaki performanslarını farklılaştırır. Şekil 6.6'da görüldüğü gibi hücre başlangıç indeksi ile hücre bitiş indeksi dizilerindeki noktalar tespit edip getirilirken daha büyük iş boyutu ve döngü yükü referanslarına göre diyebiliriz ki; Kernel2 ile karşılaştırıldığında parça geçişinde Kernel1 daha etkilidir.



a) Kernel 1 b) Kernel 2

Şekil 6.6. Parçanın her satırı tespit edip getirmesi [25].

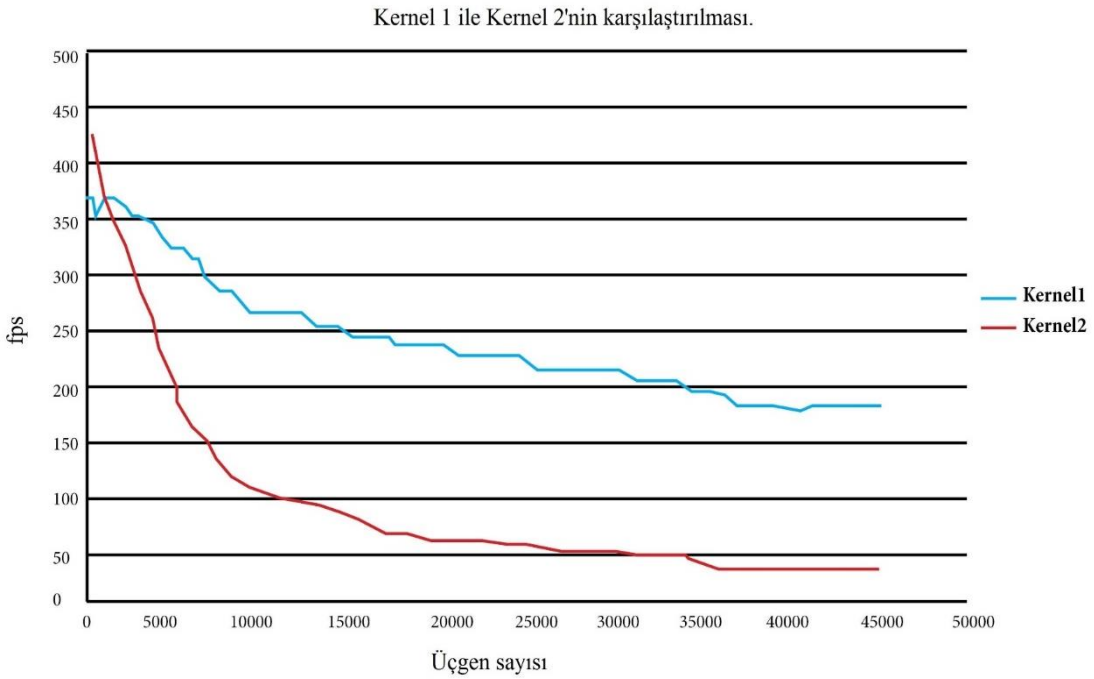
Bununla birlikte Şekil 6.7'de gösterildiği gibi Kernel2 için benzer parça desenlerine erişimi kolaylaştırmasından dolayı, nokta gruplama bu şekilde tasarlanabilinir. Mekânsal konumuna göre nokta verilerinin yeniden sıralanması yerine, parça grubuna ait örnek noktalarını tercihen X ekseni boyunca birbirine doğru yerleştirilir. Bu değişiklik Kernel2 içindeki parça döngü sürecini daha da hızlandıracaktır.



a) Karıştırılmış b) Düzenli

Şekil 6.7. Parçaların daha iyi düzenlenmesi [30].

Üçgen verilerinin farklı sayıda işlendiği durumlardaki çekirdek performansı Şekil 6.8'de karşılaştırılmıştır. Karşılaştırma aynı perspektiften ve aynı üçgen verileri ile yapılır. Çok sayıda üçgenler işlenirken, Kernel1'in performansı Kernel2'den daha iyidir.

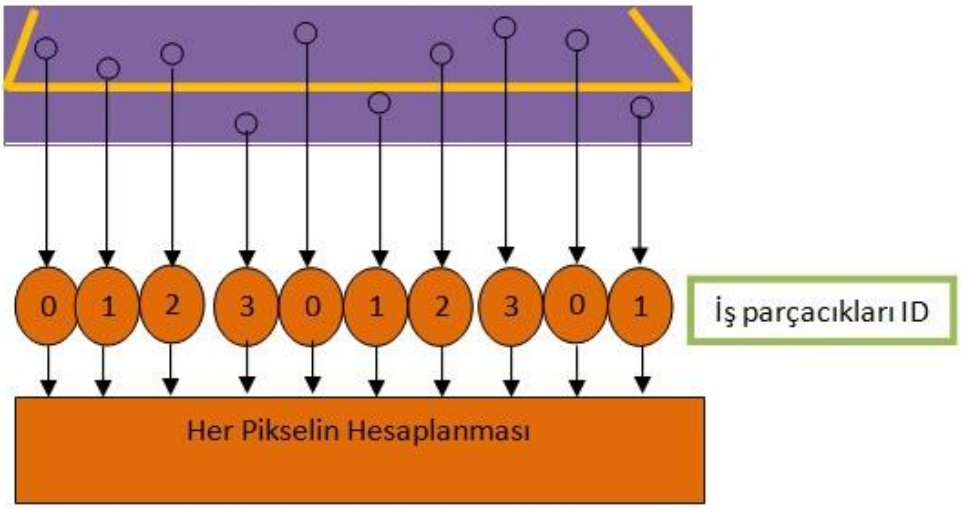


Şekil 6.8. Kernel1 ve Kernel2'nin karşılaştırılması [21].

Buna ek olarak üçgen verileri örnek noktalar için yapılmış benzer 2D şeklinde gruplanabilir. Bunu yaparak Kernel2 içindeki üçgen döngü yükü belli bir ölçüde hafifletilir.

6.3. NOKTA DÖNGÜSÜ

Her parça döngü tekrarı kapsamında bir nokta döngü gerçekleştirir. Her birimin iş parçacığına bir numara verilerek, bu iş parçacıkları veri hattı doğrultusunda noktaları tespit edilir. Tespit edilen noktalar piksel hesaplamalarında kullanılan ağırlıklı ortalama performansına katılır. Veri tipinin dört birim olduğu durumlarda Şekil 6.9'de görüntülendiği gibi bir veri hattı hesaplanır. Yapılan başka bir gözlemde de iş parçacığının indeksinin tekrarlı olduğudur. Veri hattı içindeki bölgede nokta sayısı birim boyutundan büyükse tüm iş parçacıkları bu nokta üzerinde yenilenerek tekrar ayarlanmalıdır.



Şekil 6.9. Örnek nokta döngüsü [21].

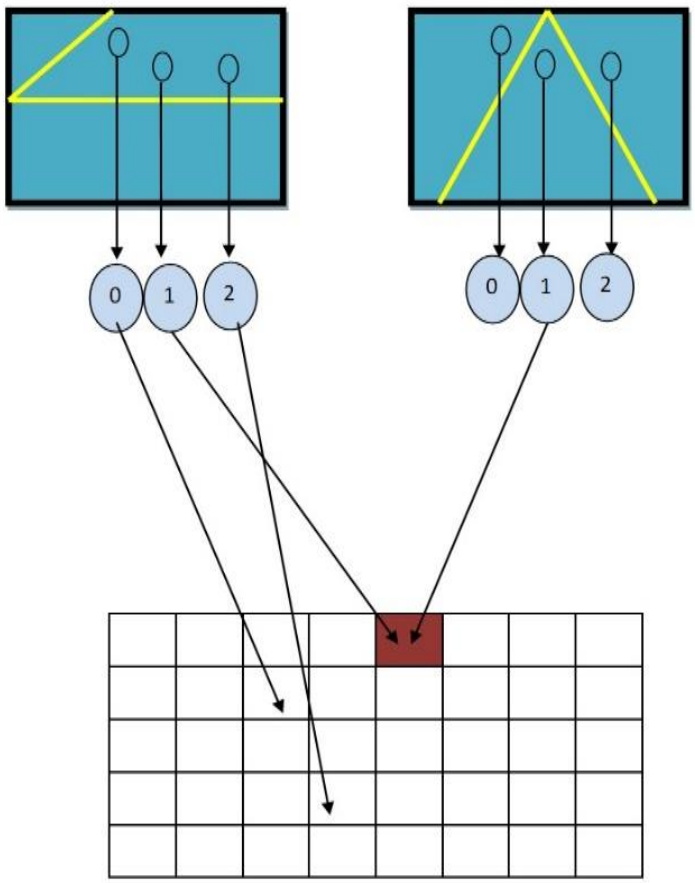
6.4. HER PİKSELİN HESAPLANMASI

Çekirdek üzerinde pikselleştirme, döngü işlem hiyerarşisinin en düşük düzeyinde gerçekleştirilir. Her pikselin hesaplanması belirli bir iş parçacığı tarafından gerçekleştirilir. Görünürlük testi noktanın üçgenin içinde olup olmadığının kontrolü için yapılır. Böylece ikinci bölümde açıklandığı gibi, üçgen derinlik değerinin enterpolasyon hesabı yapılır. Bu hesaplama işlemi yapıldıktan sonra noktanın derinliği ile üçgenin derinliği karşılaştırılır. Bu nokta üçgen içinde ise, iş parçacıkları nokta verisinin orijinal indeks bileşen bölgesi gölgede olarak işaretlenir. Bu şekilde gölge şablon tamponuna 0 değeri atanır.

Gölge şablon tamponu, daha sonra son işleme geçişte kullanılacaktır. Son işleme geçişte, ışık kaynağı doğrultusundaki gölge olmayan pikseller kendi rengine boyanır ve sadece gölgede olan pikseller için siyaha boyanır. Eğer gölge şablon tamponunda 1 değerini almış hiç bir resim çerçevesi yoksa başlangıç değerlerini resetlemek gerekir.

Şekil 6.9'da benzer veri hattı iki farklı üçgen için iki farklı birim tarafından işlendiği durumu göstermektedir. Bu durum sadece Kernel1'de oluşur çünkü Kernel2'de her parça belirli bir birim tarafından işlenir. Gölge şablon tamponu güncellenirken sağ birimde sadece bir pozisyon güncellenirken, sol birimde üç farklı pozisyonda

güncelleme yapma ihtiyacı oluşur. 2010 yılında Zhang ve Majdandzic'in yaptıkları çalışmalarda, gölge şablon tamponun güncellenmesi sırasında ekranın daha büyük bir alanının gölge olduğu durumlarda performans düşüklüğü oluşacağı sonucuna varmıştırlar. Çünkü bu işlem dağınık bir şekilde yapılır. Şekil 6.10'da koyu renk ile işaretli alan benzer tampon değerlerini farklı birimler tarafından güncellenmiştir. Bu durumda senkronizasyon gerekli değildir. Çünkü gölge şablon tamponu her zaman aynı değer ile güncellenir [21].



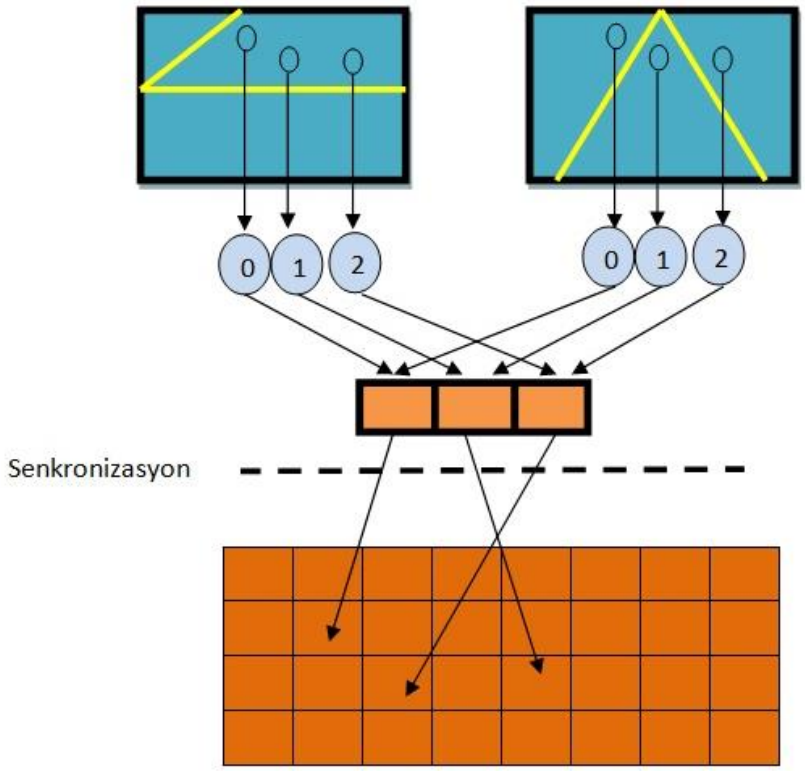
Şekil 6.10. Gölge şablon güncellenmesi [21].

6.4.1. Noktaya Kernel1 ve Kernel2'nin Uygulama Sonuçları

Piksel başına gölgelerin tespit edilmesi, işlem hesaplamalarında büyük çapta işlemler gerektirir. Bu olası durumu bir başka ara gölge şablon tamponu oluşturularak düzeltilir. Ara tampondaki her öğe gerçekten de her nokta için gölge tampon değerindedir. Ancak sıralama düzeni gruplanan nokta verisindekine benzer bir

sıralamadır. Bu nedenle ara tampon, her birim içinde birleşmiş olan örnek noktaları benzer bir düzeni takip edecek şekilde güncellenir.

Şekil 6.11'de gösterildiği gibi tüm üçgenleri pikselleştirildikten sonra, ara tampon değerleri istenen sırayla gölge şablon tamponuyla karşılıklı yer değiştirilir. Bu şekilde dağınıklık en aza indirilmiş olur.



Şekil 6.11. Gölge şablon güncellenmesinden önce senkronizasyon [21].

6.5. UYGULAMANIN ÇALIŞMA ZAMANININ ANALİZİ

İki çekirdekteki pikselleştirme arasındaki farkı anlamak için, aşağıdaki analizler yapılmıştır. Her zaman çekirdeği yürütmek için yeterli işlemci olduğu varsayılmıştır, bu şekilde birimler paralel olarak yürütülür. Noktanın dağınıklığının yaratacağı sorunları ortadan kaldırmak için her birimin çalışma zamanını aynı sürede olacağı varsayılmıştır.

Çekirdeğin yürütme zamanı belirli birimler tarafından belirlenmesi uzun zaman alır. Kernel1 ve Kernel2'nin çalıştırma zamanı aşağıdaki yöntem ile ifade edilir:

f^i: i kadar üçgenin parçalarının sayısı.

t_{par}: Bir parçanın yürütülme zamanı.

n: Üçgen sayısı.

m: Birim sayısı

$T_{kernel1}$: kernel1 yürütülme zamanı.

$T_{kernel2}$: kernel2 yürütülme zamanı.

$$T_{kernel1} \approx t_{par} \times \max(f^1, f^2, f^3, \dots, f^n)$$
$$T_{kernel2} \approx \frac{t_{par}}{m} \times \sum_{i=1}^{n} f^i. \quad (6.1)$$

Denklem 6.1'e göre, tüm üçgenlerde az ya da çok benzer boyut var ise birimler işleme başlayabilir. Bu şekilde Kernel1 ve Kernel2'nin yürütülme zamanı aynı olacaktır. Grid boyutunu aşmayacak şekilde başlatılacak birimlerin sayısı grupları parçaların içine bölünen sayı ile ilişkilidir. Bu nedenle m normal olarak n'den daha küçüktür. Diğer bir deyişle, üçgen boyutları eşit büyüklükte olan parçaların işlenmesinde Kernel1'in daha hızlı olduğu görülmüştür.

Öte yandan üçgen boyutları ne kadar dengesiz olursa yüklenme de o denli dengesiz olacaktır. Bu nedenlerden ötürü Kernel2'nin yürütülme zamanı Kernel1'e göre daha küçük olmuştur.

Bir üçgene poligon erişim süresi ve üçgeni işleme süresi denklem 6.1'e dâhil edilmemiştir. Bunun için denkleme bunu da ekleyebiliriz:

$t_{\text{üç}gen}$: Bir üçgene erişim süresi ve üçgeni işleme süresi

$$T_{kernel1} \approx t_{par} \times \max(f^1, f^2, f^3, \dots, f^n) + t_{\text{üç}gen}$$
$$T_{kernel2} \approx \frac{t_{par}}{m} \times \sum_{i=1}^{n} f^i + n \times t_{\text{üç}gen}. \quad (6.2)$$

Bu denklemlere göre yapılan işlemler sonucunda benzer boyutta olan üçgen poligonlarının pikselleştirilme işleminde Kernel1'in performansının daha yüksek olduğu görülmüştür.

BÖLÜM 7

SONUÇLAR

Bu çalışmada CUDA aracılığı ile düzensiz gölge haritalama tasarımının tüm farklı bölümlerinin üzerinden geçilmiştir. Burada yapılan çalışmalar birkaç farklı açıdan sonuçlandırılmıştır. İlk olarak, karşılaşılan sorunlar ve bu sorunlara bulunan çözümler bu çalışmada sunulmuştur. Gölgelerin kalite ve performans sonuçlarını verilmiştir.

Yapılan bu araştırmalar sonucunda;

1. Kayan noktalar beklenmeyen siyah noktalara neden olmuştur. Manzara düzleminden, yeniden oluşturulan düzensiz örnek noktaların üçgen yüzeylerine uzanan ışık düzlemindeki, örnek noktanın derinliği ve örnek noktasının konumundaki üçgen enterpolasyonuyla aynı olmalıdır. Bu iki derinlik değeri önceden öngörülmeyen bir şekilde farklı olursa, kayan noktalar saptırıcı çözünürlüklere neden olur.

2. Kayan noktaların derinlik karşılaştırmaları sırasında doğan saptırıcı hataları 2004 yılında Aila ve Laine'nın yaptıkları çalışmalarda elde ettikleri biasing değerleri kullanılarak ortadan kaldırılmıştır. Bu çalışma ile düzensiz gölge haritalarında her örnek nokta içinde mekânsal bilgi eksikliğinden oluşan işlem hataları ortadan kaldırılmıştır [10].

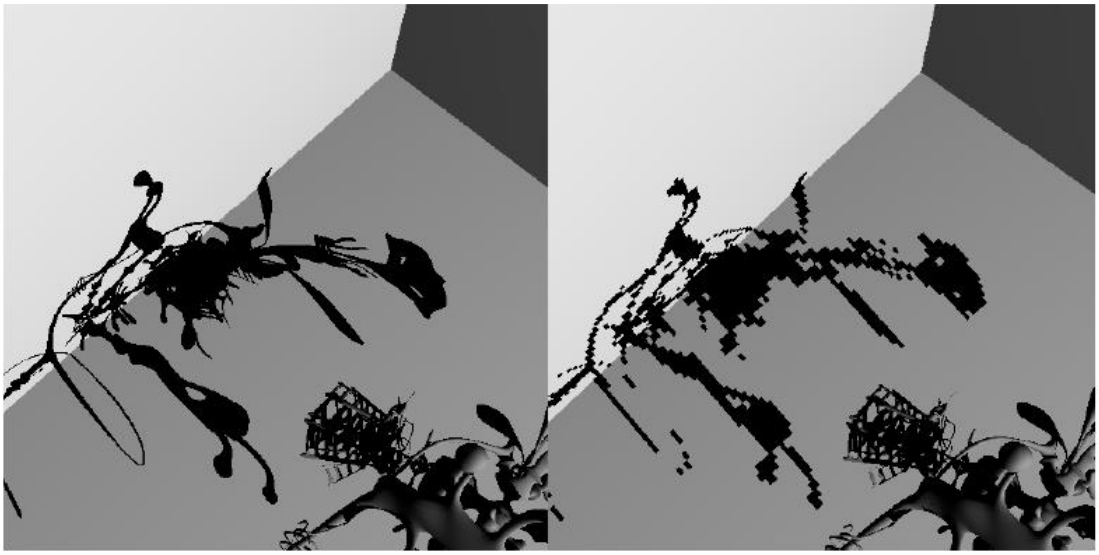
3. İyi bir kesik koni şeklinin oluşturulması için düzgün kamera bakış açısını ayarlamak gerekmektedir. Örnek noktalarının yeniden yapılandırılması kamera bakış açısıyla oluşturulan derinlik değerlerine bağlıdır. Derinlik değerleri logaritmik ölçekte 0 ile 1 aralığına eşleştirilir. Yeniden yapılandırılma sırasında örnek noktalarının derinlik değerinin daha kapsamlı incelenmesi gerekir.

4. Bunun için de kamera bakış açısındaki kesik koninin en yakın ve en uzak düzlemlerinin mümkün olduğunca yakın olması gerekir.

   Özellikle kamera bakış açısından dolayı uzak düzleme maruz kalındığında, uzak düzlem gereksiz yere sahnenin gerisinde hareketsiz olarak durur. Bu durumda örnek nokta dağılımını son derece düzensiz hale getirir. Sadece noktaların yeniden yapılanması pikselleştirme performansını tek başına yeterli kılmaz.

   Tüm pikselleştirme süreci tek hassasiyetli kayan nokta işleminde kullanılır. Bu bazı hatalara neden olsa da kolayca ortadan kaldırılır. Yani çift hassasiyetli kayan nokta işlemi performans düşüşüne yol açacağı için bunu kullanmaya gerek olmayacaktır.

5. Sadece geleneksel gölge haritalama ile bir karşılaştırma yaparak düzensiz gölge haritalama oluşturma kalitesi gösterilmiştir. Şekil 7.1'de iki farklı gölge oluşturma yaklaşımı tarafından verilen iki resim gösterilmiştir. Tüm resimler aynı sahnede aynı kamera konumundan işlenir ve aynı ışık konumu her iki yöntem tarafından paylaşılır.



a) Düzensiz noktalar (512x512) b) Düzenli noktalar (512x512)

Şekil 7.1. Gölge kalitesinin karşılaştırılması [21].

Geleneksel gölge eşleme algoritması gölge kalitesini artırmak için, farklı gölge haritaları kullanılıp çözünürlükleri karşılaştırılmıştır. Şekil 7.1 (a) ile Şekil 7.1 (b)'de görüldüğü gibi iki yöntemin resim kareleri benzerdir.

Ama her zaman çözünürlüğü artırarak geleneksel gölge dönüşümlerinden benzer bir gölge kalitesi elde edilemeyebilir. Bundan da anlaşılacağı üzere, kamera gölgenin küçük bir kısmını yakına taşır. Çünkü düzensiz gölge haritaları her zaman gölgeyi oluşturan nesnenin görünümüyle uyumludur, bu sayede gölge kalitesi her zaman korunur. Geleneksel gölge haritaları her zaman sabittir. Sadece yeterince çözünürlük değeri verilirse sorun çözülür.

6. Gerçek zamanlı resim karesi işlemelerde performansın yüksek olması çok önemlidir. Aksi takdirde bu çalışmanın bir anlamı yoktur. Sistem performanslarını etkileyecek engeller birçok yönden incelenmiştir. Veri yapısı ve dengesiz iş yükü düzensizliği nedeniyle, görünüm değiştikçe düzensiz pikselleştirici başkalaşarak genel performansı düşürür. Ancak bu çalışmada, üçgen ayrıştırma ve dinamik birim yapılandırması gibi teknikleri uygulanarak performanstaki dalgalanmalar en aza indirilmiştir.

   Düzensiz pikselleştiriciler ile test edilen iki model gösterilmiştir. Çözünürlük 512x512'ye sabitlenerek, gruplama işleminde atomik gruplama kullanılarak ve çekirdek yapış olarak da Kernel1 yöntemi seçilerek en iyi performans yakalanabilmiştir. İlk model (Şekil 7.1(a)) 123KB üçgen içeren şekildir. Bu çalışmada elde edilen ortalama performans 90fps civarındadır. Bu iki değer birlikte hesaplanırsa, bu çalışmadaki düzensiz pikselleştirici veriminin saniyede 11,1 milyon üçgen oluşturulabileceği görülür. İkinci model ise 190KB üçgen içerir, bu da saniye başına 12.4 milyon üçgen olduğunu gösterir, ortalama performans 64fps'dir.

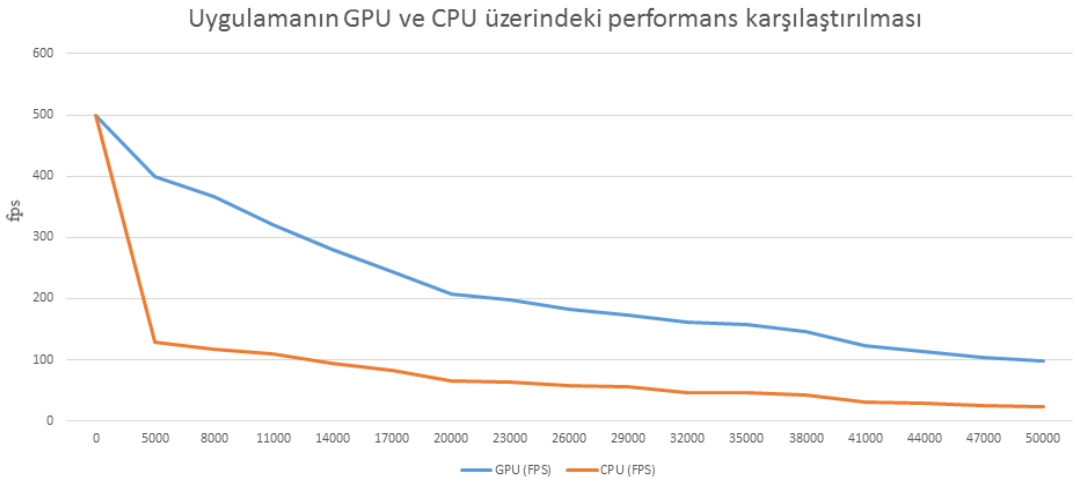
7. Uygulamada kullanılan üçgen poligon sayısına göre GPU ve CPU üzerindeki performans karşılaştırılması NVIDIA Visual Profiler tarafından ölçülerek elde edilmiştir. Elde edilen bulgular çizelge 7.1'de verilmiştir. Bu bulgulara göre,

üçgen poligon sayısı artıkça CUDA çekirdeğinin yürütme hızı CPU üzerindeki yürütülme hızına göre 4 kat hızlandığı gözlemlenmiştir.

Çizelge 7.1. Üçgen poligon sayısının GPU ve CPU üzerindeki performans karşılaştırılması.

Üçgen Poligon Sayısı	**GPU (FPS)**	**CPU (FPS)**	**GPU/CPU**
0	500	500	1,000
5000	400	128	3,125
8000	367	117	3,137
11000	320	109	2,936
14000	280	95	2,947
17000	243	83	2,928
20000	207	66	3,136
23000	197	63	3,127
26000	182	58	3,138
29000	173	55	3,145
32000	162	47	3,447
35000	157	46	3,413
38000	146,8	43	3,414
41000	123	31	3,968
44000	113	28	4,036
47000	104	25	4,160
50000	98	23	4,261

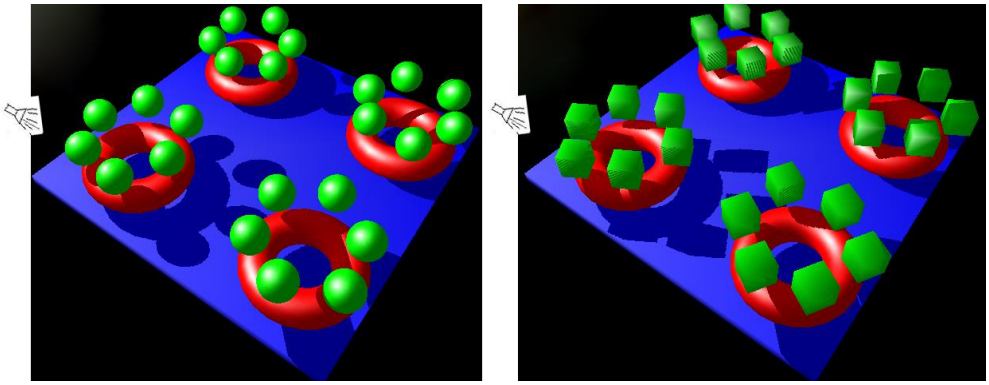
8. Bu çalışma ile geliştirilen uygulamamıza NVIDIA tarafından geliştirilen Nsight ile gerekli performans analizleri yapılmıştır. NVIDIA Nsight Microsoft Visual Studio üzerine bütünleşik olup GPU üzerindeki hesaplamaların performans analizleri işlemlerinde kullanılmaktadır. Uygulamanın zamanını NVIDIA Tesla GPU kartı üzerinde NVIDIA Nsight programı ile yaptığımız analizler sonuçları aşağıda verilmiştir [31].



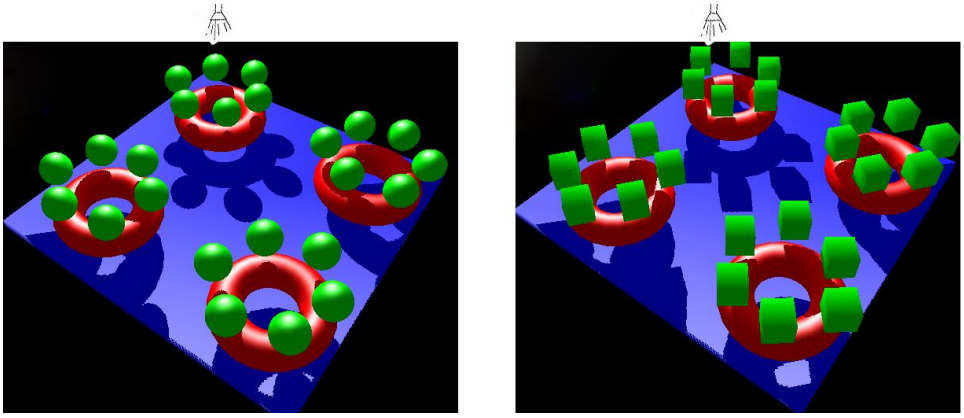
Şekil 7.3. Uygulamanın CPU ve GPU üzerindeki performans karşılaştırması.

Şekil 7.3'de gösterildiği gibi çalışmamız sonucu geliştirdiğimiz uygulama GPU üzerinde yürütülme performansı CPU üzerinde yürütülme zamanına göre %45,3 oranında performans artışı sağlanmıştır

9. Uygulamaya aynı kamera bakış açısıyla farklı ışık kaynaklarından oluşturulan gölgeler Şekil 7.4'de gösterilmiştir. Ayrıca farklı sayılarda küre ve küpler kullanılarak uygulamanın performansı karşılaştırılmıştır. Yapılan analizler sonucunda farklı nesnelerin kullanımı performansı etkilememiştir.



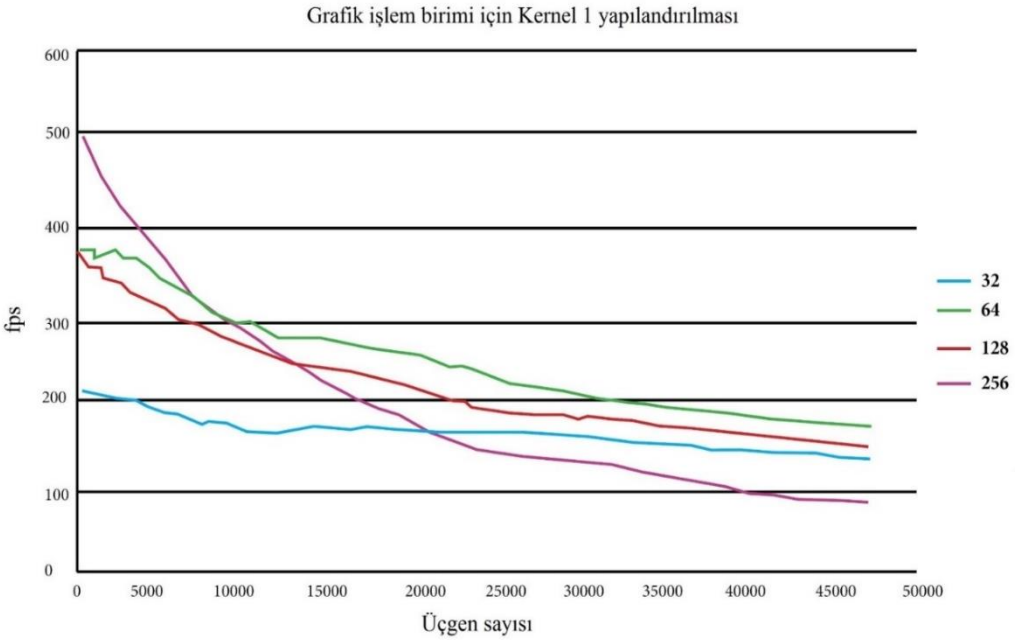
a) Küreleri ile gölgelendirme. b) Küpleri ile gölgelendirme.



b) Küreleri ile gölgelendirme. d) Küpleri ile gölgelendirme.

Şekil 7.4. Aynı kamera açısından farklı ışık kaynağı ile farklı nesnelerin performansa etkisi.

10. CUDA çekirdekleri dinamik birim yapılandırmalarında kullanmak için geliştirilmiştir. Böylece, bir üçgenin işleme sürecine katılabilecek iş parçacıklarının sayısının değiştirilmesine olanak sağlar. Şekil 7.5'de aynı sahne işlenirken bir dizi birim yapılandırmalarıyla karşılaşılmıştır. Grafikteki performans rakamları farklı kamera açılarından ve farklı üçgen sayıları ile derlenmiştir. Farklı renklerde çizilen eğriler dört farklı birim boyutunun performans eğilimlerini temsil eder. Üçgen sayısı 5.000'den küçük olduğunda 256 birim boyutu en iyi performansı verir. Büyük üçgen sayılar için 64 birim boyutu en iyi performansı verir. Bu nedenle farklı birim yapılandırmaları farklı durumlara göre değişeceği sonucuna varılmıştır.



Şekil 7.5. Grafik işlem biriminin yapılandırılması [21].

KAYNAKLAR

1. Crow, F., C., "Shadow algorithms for computer graphics", ***SIGGRAPH '77 Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques***, 11 (2): 242-248 (1977).

2. Williams, L., "Casting curved shadows on curved surfaces", ***SIGGRAPH '78 Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques***, 12 (3): 270-274 (1978).

3. İnternet: NVIDIA, Inc., "Shadow Mapping", **http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html** (2013).

4. İnternet: Microsoft Corparation, "Common Techniques to Improve Shadow Depth Maps", **http://msdn.microsoft.com/en-s/library/windows/desktop/ee416324(v=vs.85).aspx** (2013).

5. İnternet: NVIDIA, Inc., "The Advanced NVIDIA Vertex Program Profile for OpenGL",**http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter10.html** (2013).

6. Engel, W., "Cascaded shadow maps", ShaderX5: Advanced Rendering Techniques, ***Charles River Media***, Cambridge, MA, US, 197-206 (2006).

7. Stamminger, M. and Drettakis, G., "Perspective shadow maps", ***SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques***, New York, US, 557-562 (2002).

8. Lloyd, D. B., Govindaraju, N. K., Quammen, C., Molnar, S. E. and Manocha, D., "Logarithmic perspective shadow maps", ***ACM Transactions on Graphics***, 27 (4): 1-32 (2008).

9. Johnson, G. S., Mark, W. R. and Burns, C. A., "The irregular z-buffer and its application to shadow mapping", Technical Report, ***University of Texas Department of Computer Sciences***, TX, US (2004).

10. Aila, T. and Laine, S., "Alias-free shadow maps", ***Proceedings of Eurographics Symposium on Rendering Techniques***, 161-166 (2004).

11. Sintorn, E., Eisemann, E. and Assarsson, U., "Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps", ***Proceedings of Eurographics Symposium on Rendering Techniques***, 27 (4):1285-1292 (2008).

12. Akenine-Möller, T., "Mobile graphics hardware", Course Notes, ***Lund University Department of Computer Science***, Lund, Sweden (2007).

13. Green, S., "Particle Simulation using CUDA", ***NVIDIA CUDA SDK***, Santa Clara, Kaliforniya, US (2012).

14. Sengupta, S., Harris, M. and Garland, M., “Efficient parallel scan algorithms for GPUs”, ***Nvidia Technical Report***, Santa Clara, CA, US (2008).

15. İnternet: Ahn, S. H., “OpenGL Frame Buffer Object (FBO)”, **http://www.songho.ca/opengl/files/fbo.zip** (2013).

16. İnternet: Kessenich, J., Baldwin, D. and Rost, R., “The OpenGL Shading Language Version 1.20”, **http://www.cs.purdue.edu/homes/aliaga/cs535-12/GLSL-spec-4.20.8.pdf** (2013).

17. İnternet: NVIDIA, Inc., “Efficient Occlusion Culling”, **http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html** (2013).

18. Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E. and Purcell, T., “A survey of general purpose computation on graphics hardware”, ***Computer Graphics Forum***, 26 (1): 80-113 (2007).

19. Harris, M., “Optimizing parallel reduction in CUDA”, ***NVIDIA Developer Technology***, Santa Clara, CA, US (2007).

20. İnternet: Hoberock, J. and Bell, N., “C++ Template Library for CUDA-Version 1.2”, **http://code.google.com/p/thrust/** (2013).

21. Zhang, W. and Majdandzic, I., "Fast triangle rasterization using irregular z-buffer on cuda", ***Chalmers Unıversıty of Technology Department of Computer Engineering***, Göteborg, Sweden (2010).

22. Fatahalian, K., Luong, E., Boulos, S., Akeley, K., Mark, W. R. and Hanrahan, P., “Data-parallel rasterization of micropolygons with defocus and motion blur”, ***HPG '09: Proceedings of the Conference on High Performance Graphics***, New York, US, 59-68 (2009).

23. Liu, F., Huang, M-C. Liu, X-H and Wu, E-H, “Freepipe: A programmable parallel rendering architecture for efficient multifragment effects”, ***I3D '10: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games***, New York, US, 75-82 (2010).

24. Heidmann, T., “Real shadows, real time”, ***Iris Universe***, 18: 23-31 (1991).

25. Balfour, J., “CUDA Threads and Atomics”, ***NVIDIA Corparation***, Santa Clara, Kaliforniya, US (2011).

26. İnternet: NVIDIA, Inc., “Shadow Map Antialiasing”, **https://developer.nvidia.com/content/gpu-gems-chapter-11-shadow-map-antialiasing** (2013).

27. Zhou, K., Hou, Q., Wang, R. and Guo, B., “Real-time Kd-Tree construction on graphics hardware”, ***ACM Transactions Graphics***, 27 (5): 1-11 (2008).

28. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D. and Manocha, D., “Fast BVH Construction on GPUs”. ***Computer Graphics Forum***, 28 (2): 375-384 (2009).

29. Kalojanov, J. and Slusallek, P., "A parallel algorithm for construction of uniform grids", ***Proceedings of the Conference on High Performance Graphics 2009***, New York, USA, 23-28 (2009)

30. İnternet: NVIDIA, “Nvidia Cuda™ programming guide, Version 5.0 Kernels”, **http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels** (2013).

31. İnternet: NVIDIA, Nvidia Developer Zone, “NVIDIA Nsight Visual Studio Edition”, **https://developer.nvidia.com/nvidia-nsight-visual-studio-edition** (2013).

ÖZGEÇMİŞ

Murat KOCA 1979 yılında Hakkâri'de doğdu; ilk ve orta öğrenimini aynı şehirde tamamladı. Hakkâri Anadolu Lisesi Fen Bilimleri Alanından mezun oldu. 2000 yılında Azerbaycan'daki Nahçıvan Devlet Üniversitesinin Matematik ve İnformatik Öğretmenliği Programında öğrenime başlayıp 2005 yılında iyi derece ile mezun oldu. 2005-2007 yılları arasında Hakkâri'deki Sınav Dergisi Dershanesinde Matematik Öğretmeni olarak görev yaptı. 2007 yılında Amasya Üniversitesi'nde Bilgisayar Programcısı olarak göreve başladı. Amasya Üniversitesi'nin Öğrenci İşleri Otomasyon Sistemi, Personel Bilgi Sistemi, Taşınır Mal Programı ve Evrak Kayıt Programları gibi programlarını geliştirdi. 2009 yılında Hakkâri Üniversitesi'ne Uzman olarak göreve başladı. Yeni kurulan üniversitede network sistemi, güvenlik duvarı ve otomasyon sistemleri üzerine çalıştı. 2010 yılında aynı kurumda Öğretim Görevlisi olarak göreve başladı. Burada bilgisayar alanındaki dersleri vermeye başladı ve halen aynı yerde aynı görevi yapmaktadır.

ADRES BİLGİLERİ

Adres : Cezaevi Karşısı Aslan Apt.
Eylül Gıda Üstü 3/6
Kıran Mah. / HAKKÂRİ

Tel : (506) 242 22 03

E-posta : muratkoca30@gmail.com