Uludağ Üniversitesi Mühendislik Fakültesi Dergisi, Cilt 21, Sayı 1, 2016 ARAŞTIRMA

DOI: 10.17482/uujfe.97021

OPENGL TABANLI ANİMASYONLARDA GÖRÜNTÜ KALİTESİNİN CUDA MİMARİSİ İLE İYİLEŞTİRİLMESİ

Taner UÇKAN [*]
Deniz DAL [**]

Alınma: 11.06.2015; düzeltme: 24.07.2015; kabul: 31.10.2015

Öz: Gerçek hayatta meydana gelen birçok fiziksel olayın bilgisayarlar yardımıyla grafiksel olarak modellenebilmesi amacıyla 2 veya 3 boyutlu görüntü oluşturma teknolojilerinden faydalanılmaktadır. Öte yandan grafik uygulamalarının yoğunluğu arttıkça söz konusu bu modellemelerin hem daha hızlı yapılabilmesi hem de görüntü kalitelerinin artırılması gereksinimleri ortaya çıkmaktadır. Bu doğrultuda 2006 yılının sonlarında Nvidia firması tarafından CUDA isimli, yazılım ve donanım tabanlı bir mimari piyasaya sürülmüştür. Bu mimari sayesinde ekran kartları üzerinde bulunan çok sayıdaki grafik işlemcisi genel amaçlı problemlerin paralel olarak çözülebilmesine katkı sağlar hale gelmiştir. Bu çalışma kapsamında bu yeni paralel hesaplama mimarisi dikkate alınmış, C++ ve OpenGL kütüphanesi kullanılarak farklı davranış özelliklerine sahip insansı robotlardan oluşan bir animasyon uygulaması geliştirilmiştir. Bu animasyon öncelikle merkezi işlemci üzerinde seri olarak çalıştırılmış ve sonrasında CUDA mimarisi kullanılarak paralelleştirilmiştir. En sonunda aynı animasyonun seri ve paralel versiyonları saniyede oluşturulan görüntü karesi sayıları temel alınarak karşılaştırılmış ve paralel uygulamanın açık ara yüksek kaliteli görüntü ürettiği gözlemlenmiştir.

Anahtar Kelimeler: 3D Modelleme, OpenGL, C++, Nvidia, CUDA, GPGPU, Animasyon

Image Quality Improvement on OpenGL-Based Animations by Using CUDA Architecture

Abstract: 2D or 3D rendering technology is used for graphically modelling many physical phenomena occurring in real life by means of the computers. On the other hand, the ever-increasing intensity of the graphics applications require that the image quality of the so-called modellings is enhanced and they are performed more quickly. In this direction, a new software and hardware-based architecture called CUDA has been introduced by Nvidia at the end of 2006. Thanks to this architecture, larger number of graphics processors has started contributing towards the parallel solutions of the general-purpose problems. In this study, this new parallel computing architecture is taken into consideration and an animation application consisting of humanoid robots with different behavioral characteristics is developed using the OpenGL library in C++. This animation is initially implemented on a single serial CPU and then parallelized using the CUDA architecture. Eventually, the serial and the parallel versions of the same animation are compared against each other on the basis of the number of image frames per second. The results reveal that the parallel application is by far the best yielding high quality images.

Keywords: 3D Modelling, OpenGL, C++, Nvidia, CUDA, GPGPU, Animation

[*] Yüzüncü Yıl Üniversitesi, Başkale MYO, Bilgisayar Programcılığı Programı, Zeve Kampüsü 65080 VAN
** Atatürk Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, 25240 ERZURUM
İletişim Yazarı: Taner UÇKAN (taneruckan@yyu.edu.tr)

1. GİRİŞ

Günlük yaşantımızın birçok alanında karşılaştığımız fiziksel olayların sonuçlarının değerlendirilmesi ve/veya bu olayların modellenebilmesi/benzetimlerinin yapılabilmesi amacıyla bilgisayarlardan sıklıkla faydalanılmaktadır. Söz konusu bu alanların en başta gelenlerinden birisi bilgisayar grafiği ve görüntü işleme temeline dayanan uygulamaları içermektedir. Bu türden uygulamalara örnek olarak X-ray ve ultrason gibi medikal görüntüler üzerinden tanı koyabilmeye imkân sağlayan tıbbi görüntü işleme, jeolojide kullanılan zemin modelleme, otomotiv sektöründe kullanılan parça modelleme, inşaat sektöründe kullanılan yapı modelleme ve oyun programlama verilebilir.

Günümüzde teknolojinin gelişmesi ile karşılaştığımız problemlerin programlanabilme kapasitesi de işlenecek veri miktarı da bu gelişmelere paralel olarak artmaktadır. Artan bu veri miktarının tek bir işlemci (CPU) kullanılarak işlenmesi düşük veri miktarına göre daha çok zaman almakta ve verimlilik oranı da beraberinde düşmektedir. Bu problemler, grafik tabanlı uygulamalarda görüntüdeki netlik oranının düşmesi ve hareket ettirme işlemlerindeki yavaşlama olarak karşımıza çıkmaktadır. Bu amaçla, özellikle grafik yoğunluklu işlemlerde karşılaşılan görüntü problemlerinin çözülmesinde paralel programlamadan faydalanılmaktadır. Paralel programlama geleneksel olarak birden fazla bilgisayarın veya işlemcinin bireysel güçlerinin farklı bellek modelleriyle birleştirilmesi temeline dayanmaktadır. Bir ekran kartı üreticisi olan Nvidia firması bu yaklaşımın performans ve maliyet açısından yetersiz kaldığı tezinden hareketle 2006 yılının sonlarında CUDA mimarisi adını verdiği yeni bir paralel programlama modelini piyasaya sürmüştür. Bu model, sayıları artık binlerle ifade edilen ekran kartı işlemcilerinin (GPU) sadece grafik işlemleri için değil genel amaçlı hesaplamalarda da (GPGPU) kullanılabilmesi prensibini temel almaktadır. Bu makalede, OpenGL grafik ara yüz oluşturma kütüphanesi kullanılarak geliştirilen bir C++ animasyon uygulaması tanıtılmış, artan grafik verilerinin işlenmesi sırasında görüntü netliğinde meydana gelen bozulmaların giderilmesi için Nvidia CUDA teknolojisi tabanlı paralel bir çözüm önerilmiştir. Bu anlamda makalenin kalan kısmı şu şekilde organize edilmiştir. 2. Bölümde araştırma konusu ile ilgili çalışmalar incelenecektir. Araştırmada kullanılan materyalden 3. Bölümde söz edilecektir. 4. Bölüm yöntemin tanıtılmasına ayrılmıştır. 5. Bölümde ölçüm sonuçları tartışılacaktır. Makalenin son bölümü olan 6. Bölümde ise gelecek için planlanan araştırmalardan bahsedilecektir.

2. İLGİLİ ÇALIŞMALAR

Teknolojinin gelişimi ile birlikte grafik uygulamaları insan hayatının vazgeçilmez bir parçası haline gelmiştir. İhtiyaç duyulan grafik uygulamalarının geliştirilebilmesi için DirectX ve OpenGL gibi grafik ara yüzü oluşturma kütüphaneleri literatüre kazandırılmıştır. Günümüzde OpenGL, diğer grafik ara yüzü oluşturma kütüphanelerine göre daha esnek ve taşınabilirlik özelliğine sahip olduğundan, bilimsel çalışmalarda yoğun bir şekilde kullanılmaktadır. Li ve diğ. (1997), düşük maliyetli seri bilgisayarların düşük hızlı ağlarını temel alarak OpenGL özelliklerine uygun 3 boyutlu bir poligon modelleme sistemini PVM (Parallel Virtual Machine) üzerine uygulamışlardır. Benzer şekilde Kılıç (2001), insan dokularının 3 boyutlu gösterimi için MRI ve CT verilerini kullanarak OpenGL tabanlı, 3DVIEW isimli işleme ve görüntüleme araç kutusunu geliştirmiştir. Şaşoğlu (2010), bir poligon sadeleştirme yönteminin 3 boyutlu arazi üretimi algoritması ile birleştirilmesi ve sadeleştirme performansını incelemiştir. Tokatlı (2010), çeşitli istasyonlardan aldığı mağara fotoğraflarının verilerini farklı işlemlerden geçirdikten sonra OpenGL grafik kütüphanesini ve C# programlama dilini kullanarak geliştirdiği programa aktarmakta ve sonuç olarak mağaranın katı modelini oluşturmaktadır.

Wang ve Lei (2011), jeolojik nesneler arasındaki geçici, mekânsal ilişkileri tanımlamada ve mineral kaynaklarının sayısal hesaplanmasında jeologlara yardımcı olacak 3 boyutlu bir jeolojik modelleme (3DGM) programını geliştirmişlerdir. Ma ve diğ. (2013), Linux platformu üzerinde

C programlama dilini ve OpenGL grafik kütüphanesini kullanarak Yürüyen Küpler (Marching Cubes) algoritmasını geliştirmişler ve bu sayede moleküler yüzeylerin gösteriminde daha kaliteli görüntülerin oluşturulmasını sağlamışlardır.

2006 yılının sonlarında Nvidia'nın CUDA paralel hesaplama mimarisi ortaya çıkmıştır. Araştırmacılar, GPU kullanılarak mevcut grafik API'leri ile gerçekleştirilen uygulamalarda karşılaşılan problemleri çözüme ulaştırmak için C programlama diline benzer özellikler gösteren ve yüzlerce çekirdekli yüksek derecede paralel bir mimariden ve çok yüksek bellek bant genişliğinden faydalanmak istemişlerdir. CUDA mimarisinin yayınlanması ile birlikte geleneksel birçok problem bu mimari üzerinde uygulanmış ve yüksek performanslar elde edilmiştir. Manavski (2007), gelişmiş şifreleme standartları (AES) algoritmasını, Nvidia tarafından üretilmiş CUDA teknolojisi üzerine uygulamayı amaçlamıştır. Bu çalışma sonucunda, kullanılan yeni GPU teknolojisi ile Pentium IV 3.0 işlemcili sistemin kullanılması durumunda oluşan performans farkları karşılaştırılmış ve CUDA teknolojisi ile yapılan uygulamaların 20 kat daha hızlı oldukları belirlenmiştir. Benzer şekilde Okitsu ve diğ. (2010), CUDA destekli GPU'larda koni ışınlarının yeniden oluşumunu hızlandırmak için Feldkamp, Davis, and Kress (FDK) algoritmasını hızlandıran bir metot sunmuşlardır. Öte yandan CUDA mimarisi sıralama algoritmalarına da entegre edilmiş, yeni bir sıralama algoritması olan CUDA Shellsort algoritması Lin ve diğ. (2012) tarafından geliştirilmiştir.

CUDA ile birlikte gelen yüksek hesaplama kapasitesine ve programlamadaki kolaylıklara grafik programcıları tarafından kayıtsız kalınmamıştır. Özellikle grafik uygulamalarında karşılaşılan hız probleminin çözümünde CUDA mimarisi tabanlı yoğun çalışmalar yapılmıştır. Archirapatka ve diğ. (2011), GPU'lar üzerinde genel amaçlı hesaplama performansından faydalanarak 3 boyutlu görüntü oluşturmayı hedeflemiştir. GPGPU'lar yüksek hesaplama performansına sahip oldukları için gerçek zamanlı görüntü üretimi için yeni modeller oluşturmada bunlardan faydalanılabileceği düşünülmüştür. CUDA mimarisi grafik oluşturma uygulamalarında kullanılabildiği gibi var olan resimler üzerinde de yüksek performanslar sergileyebilmektedir. Yıldız (2011), resim işlemedeki bazı konvolüsyon ve filtreleme örneklerinin GPU üzerinde CUDA C dili ile CPU üzerindeki örneklerine göre daha yüksek performans gösteren türevlerini oluşturmuştur.

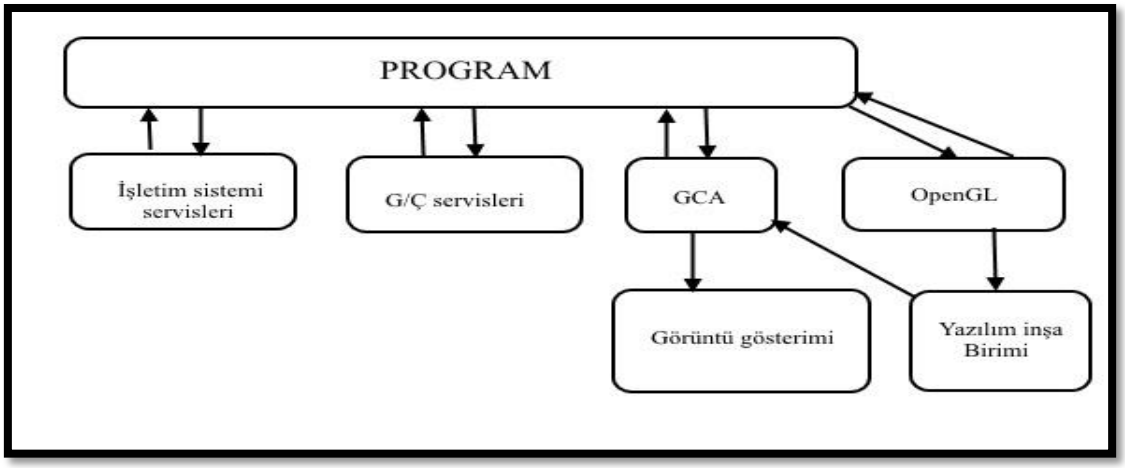
Benzer şekilde Centelles ve diğ. (2011), makalelerinde parçacık sistemine dayalı tamamen GPU üzerinde koşacak bir sistem sunmaktadırlar. Bu çalışma kapsamında yağmur yağışı esnasında oluşabilecek çarpışmaları yakalamak ve tespit etmek için CUDA teknolojisinin esnekliğinden faydalanılmıştır. Ayrıca, yağmur tanelerinin aynı zamanda sıçramalarının tespitinde de bu teknoloji kullanılmıştır. Bu çalışma neticesinde, CUDA'nın donanım programlama kapasitesinden dolayı çok yüksek performanslar elde edildiği görülmüştür. Öte yandan Nuli ve diğ. (2012), düzgünleştirilmiş parçacık hidrodinamikleri (SPH) modeline dayalı akışkan simülasyonunu CUDA destekli ekran kartları üzerinde modellemişlerdir. Yüksek hesaplama işlemleri içeren akışkan modelleme simülasyonunun CUDA üzerinde çalıştırılması sonucunda yüksek oranda hız kazancı elde edildiği gözlemlenmiştir. Benzer bir çalışmada Huaming ve diğ. (2014), yüzeyinde bozulmalar oluşabilen kumaş parçalarını modelleyen bir simülasyon çalışması gerçekleştirmişlerdir. Bu çalışmada yüzey köşegenleri doğrudan CUDA üzerinde çalıştırılarak yüksek hız kazanımı elde edilmiştir.

OpenGL ile gerçekleştirilmiş bir grafik uygulamasında veri yoğunluğundan dolayı iki temel problemle karşılaşılmaktadır. Bunlardan birisi görüntü netliğindeki bozulmalar, bir diğeri ise görüntü hızında gözlemlenen yavaşlamalardır. Yukarıdaki paragraflarda bahsedilen ve OpenGL-CUDA işbirliğini öngören çalışmaların tamamı görüntü hızındaki yavaşlamaları azaltmayı hedeflemektedir. Bizim çalışmamızın temelini oluşturan ve OpenGL tabanlı bir animasyonun görüntü netliğini CUDA kullanarak artırmaya yönelik bir araştırma bildiğimiz kadarıyla literatürde bulunmamaktadır.

3. MATERYAL

3.1. OpenGL

OpenGL (Open Graphics Library), iki ve üç boyutlu grafikleri ekrana çizdirmek amacıyla donanım özelliklerini kullanan ücretsiz bir grafik ara yüz oluşturma kütüphanesidir. İlk olarak 1992 yılında SGI firması tarafından etkili grafik görüntüleri elde etmek için geliştirilmiştir. 2006 yılından itibaren ise OpenGL'in gelişimi Khronos Group bünyesinde devam etmektedir. Bu grafik ara yüzü, bu çalışmanın yapıldığı tarih itibariyle 700'ün üzerinde fonksiyona sahip olan OpenGL 4.5 versiyonu ile kullanımdadır. Çoklu platform desteğinden dolayı Windows, Linux ve MacOS gibi farklı platformlar üzerinde sorunsuz bir şekilde çalışmaktadır. OpenGL'i diğer grafik ara yüz oluşturma kütüphanelerinden ayıran ve çoğunlukla tercih edilmesini sağlayan en önemli özelliği taşınabilir olmasıdır. Fakat her platformun kendine özgü pencere yönetimi ve dış birimlerden veri alma fonksiyonlarının farklı olması OpenGL'in taşınabilirlik özelliğini kısıtlamaktadır. Ayrıca OpenGL'de karmaşık grafik nesneleri oluşturmak için gereken fonksiyonların bulunmaması da programcılar tarafından hızlı program geliştirmeye engel oluşturmaktadır. Söz konusu bu kısıtlamalar OpenGL'de bulunan fonksiyonlara ek olarak GLU, GLUT ve GLEW adında ek kütüphaneler geliştirilerek ortadan kaldırılmıştır. Böylelikle OpenGL grafik ara yüzünün platform bağımlılığı tamamen giderilmiştir.



Şekil 1:
OpenGL Çalışma Mimarisi

Yazılım bazındaki OpenGL işlem/çalışma mimarisi Şekil 1'deki yapıya sahiptir. Tipik bir program birçok çağrıda bulunur. Bunlardan bir kısmı programcı, bir kısmı işletim sistemi ve bir kısmı da programlama dilinin kendi kütüphaneleri tarafından sağlanır. Windows uygulamaları, çıktıları ekranda göstermek için Grafik Cihaz Ara Yüzü (GCA-İngilizce GDI) denilen bir Windows API kullanırlar. Bu GCA, pencereye yazılar yazmak, çizgiler çizmek için kullanılır. Grafik kart üreticileri genellikle GCA'nın çıktı üretmek için etkileştikleri bir grafik sürücüsünü de kart ile birlikte vermektedirler. Grafik tabanlı oyunlar oynanırken grafik kartlarında problem çıkmasının sebeplerinden birisi de GCA ile kullanılan sürücünün uyuşmamasıdır. OpenGL'in yazılım tanımlaması grafik isteklerini bir uygulamadan alarak, 3 boyutlu grafiklerin renkli bir görüntüsünü oluşturur. Bu görüntünün oluşturulmasından sonra onu GCA'ya geri vererek monitöre yansıtılmasını sağlar. Windows dışındaki işletim sistemlerinde de benzer durumlar söz konusudur ancak onlarda GCA'nın yerini o işletim sistemine özgü grafik servisi almaktadır.

3.1.1. OpenGL ile Doku Haritalama

Tek veya çok boyutlu uygulamalarda doku haritalama işlemi, oluşturulan nesnelerin daha gerçekçi görünebilmesi için çok önemlidir. Günümüzde geliştirilen oyun uygulamaları incelendiğinde doku haritalama özelliğinden yoğun bir şekilde faydalanıldığı görülmektedir. OpenGL programlama kütüphaneleri tek boyutlu, iki boyutlu ve üç boyutlu doku haritalamaya destek vermektedir. Doku haritalamayı en temel ifade ile var olan nesnelerin istenilen resimler ile kaplanması olarak tanımlamak mümkündür. Doku haritalamada işlemler kare dizilerinden oluşmakta ve bu dizilerde bulunan her bir veriye texel denilmektedir. Her ne kadar kare bazlı işlemler yapılsa da, doku haritalama işlemi oval nesneler üzerinde de uygulanmaktadır.

İki boyutlu bir resmin bir nesne üzerinde haritalanması istenildiğinde öncelikle bu resmin hafızaya yüklenmesi gerekmekte, ardından bu resmin bir doku haritalama işlemi için kullanılacağı belirtilmelidir. OpenGL'deki diğer fonksiyonlarda/komutlarda olduğu gibi doku haritalama işleminde de komutlar yapacağı işlemlere göre biçimlendirilmektedir. Tek boyutlu bir doku haritalama işlemi yapılacaksa *glTexImage1D()* fonksiyonunun, iki boyutlu bir doku haritalama işlemi yapılacaksa *glTexImage2D()* fonksiyonunun ve 3 boyutlu bir doku haritalama işlemi yapılacaksa da *glTexImage3D()* fonksiyonunun kullanılması gerekmektedir. Bahsedilen bu doku haritalama fonksiyonlarının prototipleri/imzaları genel olarak aşağıdaki gibidir.

*void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * data);*

Birden fazla resmi hafızada tutup sırası geldiğinde kullanıma sunmak için OpenGL'de *glGenTextures()* fonksiyonundan faydalanılmaktadır. Bu fonksiyon sıfırdan ve birbirinden farklı işaretsiz rakamlar kullanarak doku nesneleri oluşturmaktadır.

*void glGenTextures(GLsize n, GLuint *textureNames);*

Yukarıdaki prototipe sahip *glGenTextures()* fonksiyonuna parametre olarak aktarılan *n* değeri kaç tane doku nesnesinin oluşturulacağını göstermektedir.

```
unsigned int dokuAdi[3];
glGenTextures(3, dokuAdi);
```

Doku isimlerinin oluşturulması yukarıda gösterildiği gibi yapıldıktan sonra bu isimlerin doku verilerine uygulanması gerekmektedir. Doku verilerinin bağlanması veya uygulanması işlemini yapabilmek için de *glBindTexture()* fonksiyonu kullanılmaktadır.

void glBindTexture(GLenum target, GLuint textureName);

```
unsigned int dokuAdi [3];
glGenTextures(3, dokuAdi);
glBindTexture(GL_TEXTURE_2D, dokuAdi [1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

Şekil 2'de görüldüğü üzere *glTexParameteri()* fonksiyonunun almış olduğu parametrelere göre dokuma işlemi farklılıklar göstermektedir.



Şekil 2:
OpenGL Doku Haritalama (https://open.gl/textures)

3.1.2. Merkezi İşlem Birimi (CPU) Üzerinde FPS Ölçümü

Grafik tabanlı uygulamalarda performans ölçütleri olarak oluşturulan görüntülerin hareket hızları ve görüntü kalitesindeki performansları, bir grafik uygulamasındaki görüntünün kalitesi ölçülürken de saniyede oluşturulan görüntü karesi sayısı (FPS-Frame Per Second) dikkate alınmaktadır. Bu çalışmada, OpenGL ile oluşturulan görüntülerin kalitesinin, yani görüntülerdeki netlik oranının iyileştirilmesi/artırılması hedeflenmektedir. Bu ölçüm yapılırken bir OpenGL kütüphanesi olan GLUT'tan faydalanılmaktadır. GLUT kütüphanesi sistemin birçok özelliğini sorgulamak için gerekli fonksiyonları barındırmaktadır. Bu fonksiyonlardan *glutGet()* milisaniyedeki işlem sayısını vermektedir. Bu fonksiyonun prototipi ve parametre özellikleri aşağıda verilmektedir.

```
int zaman;
zaman=glutGet(GLUT_ELAPSED_TIME);
```

Grafik uygulamalarında geçen zaman olarak bir saniyede oluşturulan ortalama görüntü karesi sayısı dikkate alınmaktadır. Oluşturulan görüntülerde işlem akışı sırasında görüntü dönüşümü veya sisteme işlenmesi gibi farklı işlemler olabilmektedir. Bu yüzden her görüntü karesinin oluşma süresi değişkenlik göstermektedir.

```
int frameSayisi=0;
float fpss=0;
int mevcutZaman=0,oncekiZaman=0;
void FpsHesapla()
{
        frameSayisi++;
        mevcutZaman=glutGet(GLUT_ELAPSED_TIME);
        int zamanFarki=mevcutZaman - oncekiZaman;
        if(zamanFarki>1000)
        {
                fpss=frameSayisi/(zamanFarki/1000.0f);
                oncekiZaman=mevcutZaman;
                frameSayisi=0;
        }
}
```

Yukarıdaki kod bloğunda belirtildiği gibi ölçüme öncelikle görüntü karesi sayısı artırılarak başlanmaktadır. Bir GLUT kütüphanesi fonksiyonu olan *glutGet(GLUT_ELAPSED_TIME)*

kullanılarak işlem anındaki süre bir değişken içerisinde tutulmakta ve başlangıç değeri sıfır olan bir değişkenin değeri ile karşılaştırılmaktadır. Bir saniye 1000 milisaniyeye karşılık geldiğinden ve sistem görüntü oluşumunu milisaniyeler içerisinde yaptığından dolayı başlangıç süresi ile geçen süre arasındaki farkın 1000 milisaniyeden yüksek olması durumunda görüntü karesi sayısı sıfırlanmakta ve geçen sürede oluşturulan görüntü karesi sayısı gösterilmektedir.

3.2. CUDA

Yüksek performanslı sistemler doğalarında paralellik barındırır. Tek işlemcili bir sistemden yüksek performans elde etmek hem fiziksel hem de ekonomik nedenlerden dolayı zordur. Bunun yerine ortak bellekli veya dağıtık bellekli sistemler olarak ifade edilen paralel sistemlerin kullanılması çözüm olabilir (Akçay ve diğ. (2011)). Ortak bellekli paralel sistemlerin programlanması kolaydır, ancak üretilmeleri zordur ve bu nedenle bu sistemler pahalıdırlar. Dağıtık bellekli sistemlerin ise programlanması zordur ama piyasadan kolaylıkla temin edilebilen bilgisayarların hızlı bir ağ ile haberleştirilmesi sonucu elde edildikleri için kurulumları kolaydır. Dağıtık sistemlerde mevcut sistem kaynakları kullanılabildiğinden maliyet de azalmakla beraber performansın iyileştirilmesi kullanılan haberleşme ağının hızı ile kısıtlanmaktadır. Bir ekran kartı üreticisi olan Nvidia firması yukarıda bahsedilen paralel programlama modellerinin performans ve maliyet açısından yetersiz kaldığı tezinden hareketle, 2006 yılının sonlarında CUDA mimarisi adını verdiği yeni bir paralel programlama modelini piyasaya sürmüştür. Bu model sayıları artık binlerle ifade edilen ekran kartı işlemcilerinin (GPU) sadece grafik işlemleri için değil genel amaçlı hesaplamalarda da (GPGPU) kullanılabilmesi prensibini temel almaktadır. Burada şunu da belirtmekte fayda vardır. CUDA mimarisinin ortaya çıkışında, özellikle son yıllarda, bilgisayar oyunlarının ve grafik tasarım programlarının gelişmesinin, "Grafik İşlem Birimi (GİB-İngilizce GPU)" adı verilen ve grafik kartlarının yerel merkezi işlem birimleri olarak nitelendirebileceğimiz bileşenlerin oldukça yüksek bir ivmeyle evrimleşmesine neden olmasının etkisi büyüktür (Çolak (2010)).

CUDA, temel olarak grafik kartlarının merkezi işlemciye ek olarak hesaplama işlemlerinin büyük bir kısmını yüklenmesi için tasarlanmıştır. Bu teknoloji ilk olarak GeForce 8 serisi üzerinde kullanılmış olup devamında Quadro ve Tesla ekran kartı serilerinde daha da geliştirilerek son kullanıcının hizmetine sunulmuştur.

3.2.1. CUDA Hafıza Modeli

CUDA mimarisinde Şekil 3'te belirtildiği gibi birden fazla ve farklı özelliklere sahip hafıza tipi bulunmaktadır. Bu hafızalar aşağıda detaylandırılmıştır.

Genel Bellek (Global Memory): Bütün iş parçacıkları (thread) ve CPU tarafından erişime açıktır, ayrıca okunabilme ve yazılabilme özeliğine sahiptir.

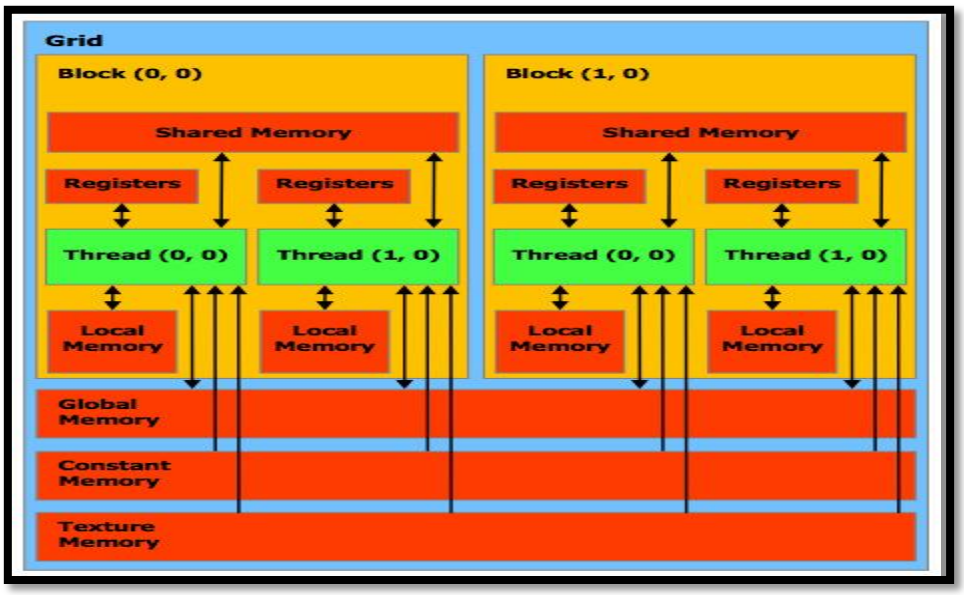
Paylaşımlı Bellek (Shared Memory): Aynı blokta bulunan iş parçacıkları tarafından erişime açıktır, ayrıca okunabilme ve yazılabilme özeliğine sahiptir.

Yerel Bellek (Local Memory): Aygıt hafızası üzerinde bulunmaktadır ve her iş parçacığı için ayrılmış özel bir alandır.

Kayıtçı Bellek (Registers): Erişim hızı yüksek olan ve her iş parçacığı için ayrılmış bir hafıza alanıdır.

Değişmez Bellek (Constant Memory): Sadece okuma özeliğine sahiptir ve sabit fonksiyonlar ile bu fonksiyonların parametrelerinin tutulduğu bir alandır.

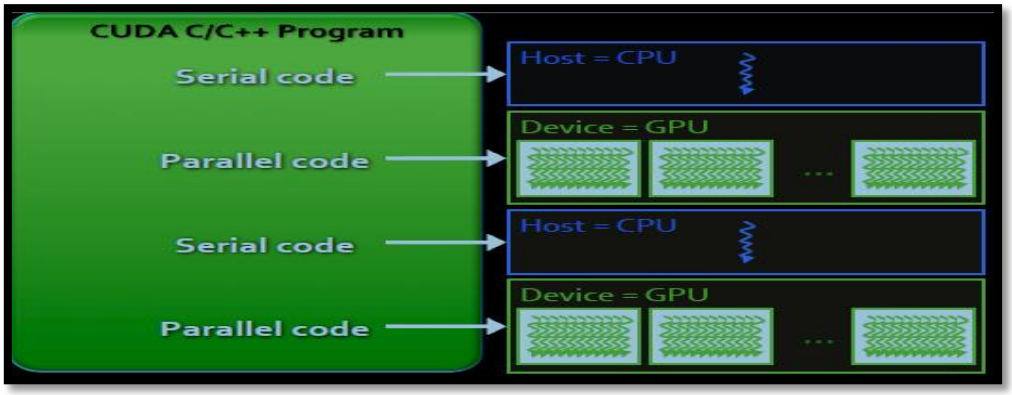
Doku Belleği (Texture Memory): Doku verilerinin tutulduğu ve sadece okuma yapılabilen bir bellek alanıdır.



Şekil 3:
CUDA Hafıza Hiyerarşisi (Nvidia, (2007))

3.2.2. CUDA ile Paralel Programlama

CUDA programlama modeli, GPU üzerindeki işlem birimlerini merkezi işlem birimine yardımcı bir işlemci olarak tanımlamaktadır. Merkezi işlem birimleri genel olarak birden fazla işlemi ardı ardına işleme ile görevlendirilmiş ve hafıza alanlarına ulaşımda oluşabilecek gecikmeleri önlemek için optimize edilmiştir. GPU'lar ise çoklu işlemleri ardı ardına yapmak ve yüksek çalışma performansları elde etmek amacıyla tasarlanmış olup, hafızaya ulaşım sırasında oluşabilecek gecikmeleri engelleyen çok kanallı bir çalışma yapısına sahiptirler. Bir yazılım uygulaması çalıştırılırken sadece belli bir kısmı paralel çalıştırmaya uygun olmaktadır. Bu durumda Şekil 4'te görüldüğü gibi programın seri kısımları merkezi işlem birimi **(CPU) (Host)** üzerinde, paralel kısımları ise **GPU (Device)** üzerinde çalıştırılmaktadır (Balfour, (2011)).



Şekil 4:
Çekirdek Fonksiyonlar

CUDA C yapısı standart C programlama dilini genişleterek programcılara çekirdek (kernel) denilen C fonksiyonlarını tanımlama imkânı sağlamaktadır. Yazılan çekirdek fonksiyonları standart C fonksiyonlarından farklı olarak *N* defa çalıştırıldıklarında *N* farklı kanalda paralel

olarak koşturulmaktadırlar. CUDA C ile çekirdek fonksiyonları **__global__** ifadesi ile tanımlanmaktadırlar. Tanımlanan çekirdek fonksiyonunun çalışacağı kanal sayısı **<<<…>>>** ifadesi kullanılarak belirtilmektedir. Her kanal kendisine verilen ve **threadIDx** diye isimlendirilen benzersiz bir numara kullanılarak çağrılır ve çalıştırılacak görevler atanır.

```
__global__ void VecAdd( float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i] ;
}
int main()
{
        …
        VecAdd <<<1,N>>>( A, B, C) ;
        …
}
```

Şekil 5:
Çekirdek Fonksiyon Yapısı (Nvidia, (2007))

Şekil 5'te gösterildiği gibi *N* kanal boyunca iki sayının (dizinin) toplanıp bir sayıya (diziye) atanma işlemi *VecAdd()* çekirdek fonksiyonu tarafından yapılmaktadır (Nvidia, (2014)).

4. YÖNTEM

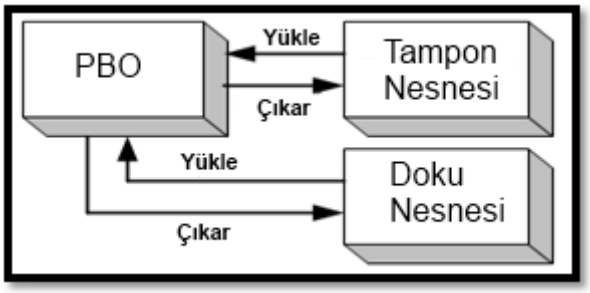
OpenGL ile gerçekleştirilen bir uygulamanın CUDA mimarisi ile paralelleştirilmesi için her iki teknoloji arasında iş birliğinin sağlanması gerekmektedir. OpenGL ve CUDA mimarisi arasındaki işbirliğinin sağlanabilmesi için öncelikle kullanılacak ortak hafıza bölgesinin (PBO-Pixel Buffer Object) OpenGL tarafından CUDA'ya kaydedilmesi ve CUDA tarafından işlenebilecek duruma getirilmesi gerekmektedir. Grafik uygulamalarında birden fazla PBO kullanılabilmektedir. Bu uygulamada gönderilecek ve tutulacak görüntüleri işaret eden tek bir PBO kullanılmaktadır.

OpenGL veri hafızası farklı özelliklerde verileri tutabilecek şekilde birden fazla bölüme ayrılabilmektedir. Bahsedilen bu bölümlerden en sık kullanılan *ARB_pixel_buffer_object* ve *ARB_vertex_buffer_object* eklentileri büyük benzerlikler göstermektedir. Bu eklentiler sadece koordinat bilgilerinin değil aynı zamanda piksel bilgilerinin de veri havuzunda depolanmasını sağlamaktadır. Piksel değerlerinin depolanma işlemini sağlayan veri havuzu PBO olarak isimlendirilmektedir. Bu veri havuzu bütün koordinat bilgilerini, kütüphane bilgilerini ve bunlara ek olarak 2 adet anahtar değeri içermektedir. Bu anahtar değerlerin yardımı ile depolanacak bilgilerin sistem hafızasında en uygun yere depolanması kontrol edilmektedir. Ayrıca Şekil 6'da gösterildiği gibi anahtar değerler piksel değerlerinin PBO'ya yüklenmesini belirten *GL_PIXEL_PACK_BUFFER_ARB* ve piksel değerlerinin PBO'dan alınmasını sağlayan *GL_PIXEL_UNPACK_BUFFER_ARB* değerlerini almaktadır (Ahn, (2014)).

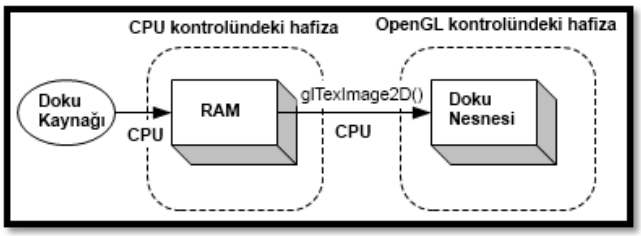
OpenGL ile CUDA arasında işbirliği yapılması durumunda PBO'nun kullanılması temelde 2 avantaj içermektedir. Bu yöntem sayesinde nesne ile hafızaya doğrudan erişim sağlanmakta ve herhangi bir CPU işlem çevrimine girmeden hafızaya bilgi yükleme ve hafızadan bilgi alma işlemleri gerçekleştirilmektedir. Ayrı PBO'lar ile sistem hafızası arasında veri transferi asenkron olarak yapılabilmekte ve bu özellik veri aktarımı esnasında büyük bir avantaj sağlamaktadır (Ahn, (2014)).

Şekil 7'de görüldüğü gibi görüntü bilgisi öncelikle sistem hafızasına yüklenmekte ve ardından *glTexImage2D()* fonksiyonu kullanılarak OpenGL doku nesnesine yüklenme işlemi yapılmaktadır. Doku verisi OpenGL tarafından işlenmeye başlamadan önce CPU tarafından iki

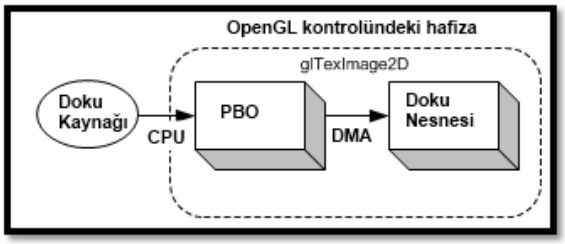
işlem döngüsü süresince kullanılmaktadır. Bu durum görüntünün hızlı bir şekilde işlenme performansında dezavantaj oluşturmaktadır.



Şekil 6:
OpenGL Veri Havuzu Yapısı (Ahn, (2014))



Şekil 7:
PBO Kullanmadan Veri Aktarım İşlemi (Ahn, (2014))



Şekil 8:
PBO Kullanarak Veri Aktarım İşlemi (Ahn, (2014))

Şekil 8'de görüldüğü gibi doku bilgisi OpenGL kontrolünde bulunan PBO'ya doğrudan yüklenmektedir. Veri transfer işleminin bu şekilde yapılması durumda CPU yalnızca doku bilgisinin ilk olarak PBO'ya yüklenmesi işlemini yapmaktadır. Ayrıca doku bilgisinin PBO'dan doku nesnesine transfer edilmesi işlemine karışmamaktadır. Bu durumda bilginin doku nesnesine transfer işlemi tamamen OpenGL sürücüsü tarafından doğrudan hafızaya erişim metodu kullanılarak yapılmakta ve performans artışı sağlanmaktadır (Ahn, (2014)).

5. ÖLÇÜM SONUÇLARI VE TARTIŞMA

5.1. Test Ortamı

Geliştirilen animasyon uygulamasının testlerinin yapıldığı platformun özellikleri aşağıda belirtilmektedir.

- Intel Core 2 Quad CPU Q8300 2.50GHz
- 4 GB 800 MHz DDR2 Bellek
- NVIDIA GeForce 9500GT Grafik Kartı
- Windows 7 Ultimate İşletim Sistemi
- CUDA SDK Versiyon 5.0
- Visual Studio 2010
- 32-Bit (x86) Mimarisi

5.2. Problemin CUDA ile Çözümü

OpenGL kütüphaneleri kullanılarak oluşturulan görüntü verilerinin kaydedileceği tampon nesnesinin CUDA işlemcileri tarafından işlenebilecek duruma getirilebilmesi gerekmektedir. Bu nesnenin oluşturulma ve CUDA'ya kayıt işlemleri aşağıdaki kod bloğu üzerinde gösterilmektedir.

```
void PboOlustur(GLuint* pbo)
{
	if(pbo)
	{
		int texelSayisi=genislik * yukseklik;
		int toplamAlan=texelSayisi*4;
		int texelVeriBoyutu=sizeof(Glubyte) * toplamAlan;
		glGenBuffers(1,pbo);
		glBindBuffer(GL_PIXEL_UNPACK_BUFFER,*pbo);
		glBufferData(GL_PIXEL_UNPACK_BUFFER,texelVeriBoyutu,NULL,GL_DY
		NAMIC_COPY);
		cudaGLRegisterBufferObject(*pbo);
	}
}
```

Gereken veri havuzu tanımlama ve CUDA'ya tanıtılma işlemleri yapıldıktan sonra aşağıdaki kod bloğunda görüldüğü gibi *cudaGLMapBufferObject()* fonksiyonu kullanılarak daha önceden tanıtılmış PBO nesnesi CUDA sürücüsü üzerinde haritalanmakta ve GPU işlemcileri üzerinde çalışacak fonksiyon çağrılmaktadır. GPU üzerindeki işlemler tamamlandıktan sonra da haritalanmış görüntü bilgisi serbest bırakılmaktadır.

```
void CudaIslem()
{
	uchar4 *dptr=NULL;
	cudaGLMapBufferObject((void**)&dptr,pbo);
	KernelCalistir(dptr,genislik,yukseklik,animasyonZamani);
	cudaGLUnmapBuffer(pbo);
}
```

GPU'lar üzerinde işlenecek verilerin CUDA'ya tanıtılma işlemleri ve sürücü üzerinde haritalama işlemleri yapıldıktan sonra bu işlemler için kaç adet blok oluşturulacağı ve her blokta kaç tane iş parçacığı çalıştırılacağı belirtilmektedir. Bu uygulama içerisinde 800*800 boyutunda bir görüntü alanı oluşturulmakta ve bunlar 800*800 adet iş parçacığı üzerinde çalıştırılmaktadır. Oluşturulan her bir blokta 256 adet iş parçacığının çalışması mantıksal olarak belirlenmektedir. Aşağıdaki kod bloğunda blok ve iş parçacığı sayılarının belirlenme işlemi gösterilmektedir.

```
void KernelCalistir(uchar4* pos,unsigned int imageGenislik,unsigned int imageYukseklik,float zaman)
{
        int threadSayisi=256;
        int toplamThread=imageYukseklik*imageGenislik;
        int blockSayisi=toplamThread/threadSayisi;
        blockSayisi+=((toplamThread%threadSayisi)>0)?1:0;
        kernel<<<blockSayisi, threadSayisi>>>(pos,imageGenislik,imageYukseklik,zaman);
        cudaThreadSynchronize();
        checkCUDAError("Cekirdek Fonksiyonu Calistirilamadi !");
}
```

GPU üzerinde çalışacak iş parçacığı ve blok sayıları belirlendikten sonra **<<<..>>()** belirteç yapısı kullanılarak iş parçacıkları üzerinde paralel olarak koşacak çekirdek fonksiyonu çağrılmaktadır. Çekirdek fonksiyonun çalışması ile Şekil 9'de görüldüğü gibi her biri 256 iş parçacığı içeren 2500 blok üzerinde PBO'ya yüklenen doku bilgisi işlenmektedir. Bu durum sonucunda eskisinden çok daha kaliteli bir görüntü elde edilebilmektedir. Tutarlı bir performans analizi yapılabilmesi için CPU üzerinde kullanılan doku verisi oluşturma fonksiyonunun aynısı GPU'lar üzerinde de çalıştırılmıştır. İstenilen amaçlar doğrultusunda bu fonksiyonların programcı tarafından değiştirilerek farklı problemlerin çözümünde kullanılması mümkündür.

5.3. Grafik Ortamı Oluşturma ve Elde Edilen FPS Değerlerini Karşılaştırma

OpenGL ile nesneler oluşturulurken ilkel nesnelerden faydalanılmaktadır. Ortam oluşturma işlemleri öncelikle OpenGL penceresinin ilkleme işleminin çağrılması ile başlamakta, sonrasında pencere boyutu, pozisyonu ve görüntü derinliğinin belirlenmesi işlemleri ile devam etmektedir. Oluşturulan ekranda takip eden adımlarda tasarlanacak robotların üzerinde hareket edecekleri zemin tasarımı GLUT kütüphanesinde tanımlı *GL_QUARDS* nesnesi kullanılarak hazırlanmaktadır. OpenGL uygulamasının başlangıç ayarlamaları aşağıda görüldüğü gibi yapılmaktadır.

```
int main (int argc, char **argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(1400,800);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Robot Animasyon Uygulamasi");
        init();
        GlewInit();
        if(!glewIsSupported("GL_VERSION_2_0"))
        {
                fprintf(stderr,"HATA:Gerekli OpenGL Eklentileri Bulunamadi");
                fflush(stderr);
                exit(0);
        }
}
```

Grafik ekranı için gereken başlangıç işlemleri yapıldıktan sonra 2 boyutlu zemin oluşturma işlemleri aşağıdaki kod bloğunda görüldüğü gibi *GL_QUARDS* nesnesi kullanılarak gerçekleştirilmektedir.

```
void ZeminCiz()
{
        glBegin(GL_QUARDS);
                glTexCoord2f(0.0f,1.0f);glVertex3d(80,-7.3,80);
                glTexCoord2f(0.0f,1.0f);glVertex3d(-80,-7.3,80);
                glTexCoord2f(0.0f,1.0f);glVertex3d(-80,-7.3,-80);
                glTexCoord2f(0.0f,1.0f);glVertex3d(80,-7.3,-80);
        glEnd();
}
```

Görüntü penceresi ve zemin için gereken OpenGL kodları yazıldıktan sonra programın derlenmesi sonucu Şekil 9'da görünen ekran çıktısı oluşmaktadır.



Şekil 9:
Ekran Çıktısı Görüntüsü

Performans analizinin ilk adımında, robot nesnelerinin ekranda oluşumundan önce hazırlanan grafik ortamın analizi yapılmaktadır. Tablo 1'de grafiksel ortamın oluşumu sonrasında merkezi işlem biriminin ve merkezi işlem birimi ile GPU'ların beraber çalışması sonucu oluşan FPS değerleri gösterilmektedir.

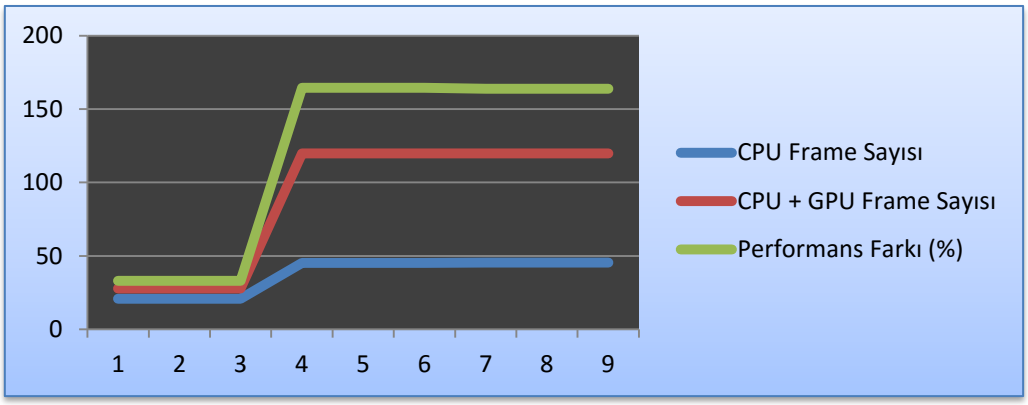
Tablo 1. Grafik Ortamın Hazırlanması Aşamasında Oluşan Görüntü Karesi Sayıları

Durum	CPU Görüntü Karesi Sayısı	CPU + GPU Görüntü Karesi Sayısı	Performans Farkı (%)
Başlangıç Aşaması	20,87	27,77	33,06
Ekran Oluşum Aşaması	45,27	119,76	164,51
Kararlı Durum	45,45	119,88	163,73

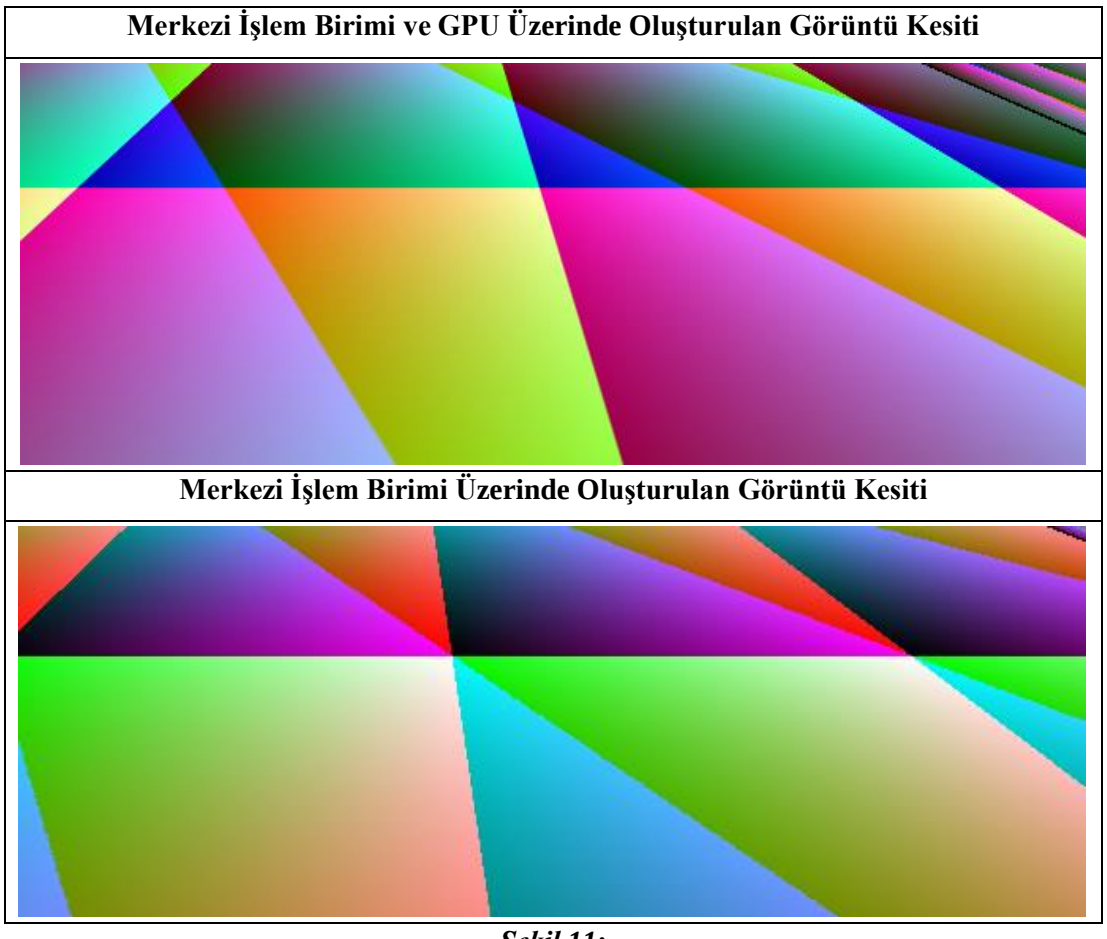
Elde edilen veriler doğrultusunda performans analizi grafiksel olarak Şekil 10'da gösterilmektedir.

Tablo 1 ve Şekil 10'dan anlaşılacağı üzere başlangıç aşamasında işletim sistemi tarafından yapılması gereken ortam hazırlama, kullanılacak kütüphane dosyalarının yüklenmesi ve kullanıcı arabirimi ile yapılacak giriş/çıkış gibi işlemler dolayısıyla oluşturulan görüntü karesi sayısı düşük seviyede başlamaktadır. Ayrıca oluşturulan her görüntü karesinin oluşum süreleri o anki yapılacak başka bir işlem veya dışarıdan talep edilen ek bir görev olması durumunda değişiklik göstermektedir.

Öte yandan doku haritalama noktasında CUDA çözümü ile elde edilen performans iyileştirmesinin %33 ile başlayıp %164'e kadar arttığı gözlenmektedir. Şekil 11'de görüldüğü gibi sadece CPU üzerinde gerçekleştirilen doku haritalama bölümüne dikkat edildiğinde renkler arası geçiş katmanlarında pürüzlerin oluştuğu fark edilmektedir. Bunun aksine GPU işlemcilerinin işlem akışına dâhil olması durumda oluşan görüntü pürüzlerinin ortadan kalktığı ve daha canlı, daha pürüzsüz bir görüntünün oluştuğu gözlemlenmektedir.



Şekil 10:
Grafik Ortamın Oluşturulmasının Analizi



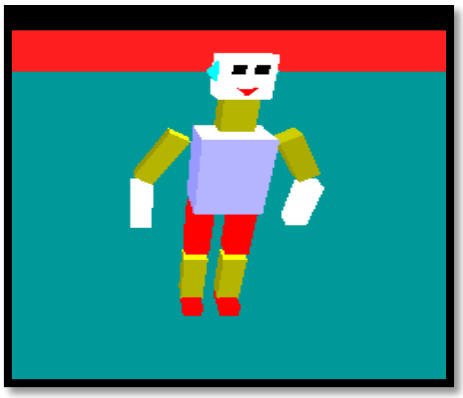
Şekil 11:
Grafik Ortamının Görüntüsü

5.4. 140 Adet Robot Oluşumu İçin FPS Değerlerinin Hesaplanması ve Karşılaştırılması

OpenGL ile robot tasarımı yapılırken, ilkel nesneler kullanılarak bir fonksiyon haline getirilen ve GLUT kütüphanesi altında bulunan 3 boyutlu küp oluşturma fonksiyonu olan *glutSolidCube()* fonksiyonundan faydalanılmaktadır. Kullanılan küp nesnelerinin birleşiminden ise robot elde edilmektedir. Yapılan robot uygulaması farklı boyuttaki küplerden oluşan 13 eklemden meydana gelmektedir. Bu eklemlerden üst kolu oluşturma kodu aşağıda verilmiştir. Robotun kalan diğer eklemleri, koordinat ve boyut değerleri farklı olmak koşuluyla benzer şekilde kolaylıkla gerçekleştirilmektedir.

Bütün eklemler için gerekli olan OpenGL kodları yazıldıktan sonra 13 bölümden oluşan bir robotun görünümü Şekil 12'deki gibi olmaktadır.

```
void UstKolCiz()
{
        glColor3f(0.5,0.5,0.5);
        glPushMatrix();
                glSolidCube(0.5);
                glTranslatef(0,-0.25,0);
                glSolidCube(0.5);
                glTranslatef(0,-0.25,0);
                glSolidCube(0.5);
                glTranslatef(0,-0.25,0);
                glSolidCube(0.5);
                glTranslatef(0,-0.25,0);
                glSolidCube(0.5);
        glPopMatrix();
}
```



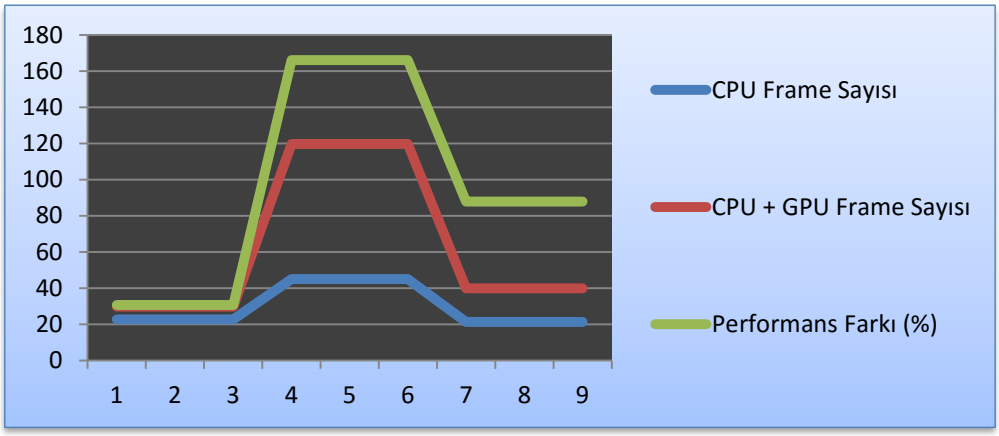
Şekil 12:
OpenGL ile Robot Tasarımı

Bu adımda ortam hazırlama işlemlerine ek olarak 140 adet robot oluşumu sonucunda meydana gelen görüntü karesi sayıları karşılaştırmalı olarak gösterilmektedir. Tablo 2'de 140 adet robot oluşturulması durumunda üretilen görüntü karesi sayıları verilmektedir.

Tablo 2. 140 Adet Robot Hazırlanması Aşamasında Oluşan Görüntü Karesi Sayıları

Durum	CPU Görüntü Karesi Sayısı	CPU + GPU Görüntü Karesi Sayısı	Performans Farkı (%)
Başlangıç Aşaması	22,79	29,79	30,69
Ekran Oluşum Aşaması	45,00	119,76	166,07
Kararlı Durum	21,27	39,96	87,81

Elde edilen veriler doğrultusunda 140 adet robot için performans analizi grafiksel olarak Şekil 13'de gösterilmektedir. Ekranda bulunan ortam görüntüsüne 140 adet robotun eklenmesi sonucunda işlenecek veri miktarı artmakta ve FPS değerleri merkezi işlemci üzerinde 140 adet robot için 21.27 değerine ve GPU üzerinde 39.96 değerine düşerek kararlı duruma gelmektedir. Böylelikle 140 adet robot için FPS değerlerine bakıldığı zaman kararlı durumda sadece CPU kullanımı ve CPU ile GPU'nun beraber kullanımı arasında %87.81'lik bir performans artışı olduğu gözlemlenmektedir.



Şekil 13:
140 Adet Robot İçin Karşılaştırmalı Performans Analizi

Şekil 13'teki performans analizi grafiğinin de belirttiği üzere haritalanması gereken yüzey sayısı arttıkça oluşturulan anlık görüntü karesi sayısı düşmektedir. Öte yandan görüntü karesi oluşumu esnasında dışarıdan verilen komutlar doğrultusunda robotlar gruplara (4 farklı grup mevcuttur.) özel atanmış hareketler yapmaktadır. Bu nedenle uygulamanın hem başlangıç aşamasında hem de robotların sisteme eklenmesi aşamasında animasyon hareketleri mevcuttur ve bu da performans analizi yapılırken hesaba katılmıştır. Yapılan bu performans analizleri ve elde edilen FPS değerleri doğrultusunda, doku haritalama işleminde CUDA mimarisi kullanılması durumunda yüksek oranda performans iyileştirilmesi elde edildiği gözlemlenmiştir. Aynı zamanda geliştirilen uygulamada nesnelerin hareket ettirilmesinin doku haritalama performansı üzerinde kayda değer bir değişiklik meydana getirmediği görülmüştür.

6. SONUÇ VE PLANLANAN ÇALIŞMALAR

Son yıllarda, "Grafik İşlem Birimi" adı verilen ve grafik kartlarının yerel merkezi işlem birimi olarak nitelendirebileceğimiz donanımlar oldukça yüksek bir ivmeyle evrimleşmiştir. Bu gelişme özellikle bilgisayar oyunlarında ve grafik tasarım programlarında görüntü kalitesinin artmasına neden olmuştur. Bu gelişimin öncülerinden birisi ve bir ekran kartı üreticisi olan Nvidia firması 2006 yılının sonlarında CUDA mimarisi adını verdiği yazılım ve donanım tabanlı yeni bir paralel programlama modelini piyasaya sürmüştür. Bu model, sayıları artık binlerle ifade edilen ekran kartı işlemcilerinin sadece grafik işlemleri için değil, genel amaçlı

hesaplamalarda da kullanılabilmesi prensibini temel almaktadır. Bu makalede, OpenGL grafik ara yüz oluşturma kütüphanesi kullanılarak geliştirilen bir C++ animasyon uygulaması tanıtılmış, artan grafik verilerinin işlenmesi sırasında görüntü netliğinde meydana gelen bozuklukların giderilmesi için Nvidia CUDA teknolojisi tabanlı paralel bir çözüm önerilmiştir. Söz konusu animasyon öncelikle merkezi işlemci üzerinde seri olarak çalıştırılmış ve sonrasında CUDA mimarisi kullanılarak paralelleştirilmiştir. En sonunda aynı animasyonun seri ve paralel versiyonları saniyede oluşturulan görüntü karesi sayıları temel alınarak karşılaştırılmış ve paralel uygulamanın açık ara yüksek kaliteli görüntü ürettiği gözlemlenmiştir.

Bu bilimsel çalışmanın gelecek planı ise OpenGL tabanlı bir animasyon uygulamasının hem görüntü netliğindeki bozulmaları ve hem de hareket hızlarında meydana gelen yavaşlamaları aynı anda azaltmayı hedefleyen bir bütün çözümün CUDA ile gerçekleştirilmesi şeklinde özetlenebilir.

KAYNAKLAR

1. https://open.gl/textures (Erişim Tarihi: 08.05.2015). OpenGL Doku Haritalama.

2. Ahn, H. S., (2014). OpenGL Pixel Buffer Object (PBO). *http://www.songho.ca/opengl/gl_pbo.html* (Erişim Tarihi: 05.03.2015).

3. Akçay, M., Şen, B., Orak, İ. Ö., Çelik, A., (2011). Paralel Hesaplama ve CUDA. *6. Uluslararası İleri Teknolojiler Sempozyumu* (İATS'11), Elazığ.

4. Archirapatkave, V., Sumilo, V., See, S.C.W., Achalakul, T., (2011). GPGPU Acceleration Algorithm for Medical Image Reconstruction. *Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications,* Singapore.

5. Balfour, J., (2011). Introduction to CUDA. *http://mc.stanford.edu/cgi-bin/images/f/f7/Darve_cme343_cuda_1.pdf* (Erişim Tarihi: 20.03.2015).

6. Centelles, A. P., Sunyer, N., Ripolles, O., Chover, M., Sbert, M., (2011). Rain Simulation in Dynamic Scenes. *International Journal of Creative Interfaces and Computer Graphics*, 2 (2), 23-36.

7. Çolak, M. A., (2010). Grafik Kartı Üzerinde Paralel Hızlandırılmış Işın İzleme.*Yüksek Lisans Tezi*, İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü, İstanbul.

8. Huaming, L., Baosheng, K., (2014). Real-time Physically Cloth Simulation with CUDA. *Computer Modelling & New Technologies*, 18 (12B), 28-32.

9. Kılıç, O. M., (1995). An Open Graphics Library (OpenGL) based Toolbox for Biomedical Image Display and Processing.*Yüksek Lisans Tezi*, Boğaziçi Üniversitesi Biyomedikal Mühendisliği Enstitüsü, İstanbul.

10. Li, W., Huang, X., Zheng, N., (1997). Parallel Implementing OpenGL on PVM. *Parallel Computing*, 23 (1997) 1839-1850.

11. Lin, C.Y., Lee, W.S., Tang, C.Y., (2012). Parallel Shellsort Algorithm for Many-Core GPUs with CUDA. *International Journal of Grid and High Performance Computing*, 4 (2), 1-16.

12. Ma, L., Zhao D.X., Yang Z.Z., (2013). A Software Tool for Visualization of Molecular Face (VMF) by Improving Marching Cubes Algorithm. *Computational and Theoretical Chemistry*, 1028 (2014), 34–45.

13. Manavski, S.A., (2007). CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography, *IEEE ICSPC 2007*, 24-27.

14. Nuli, U. A., Kulkarni P. J., (2012). SPH Based Fluid Animation Using CUDA Enabled Gpu. *International Journal of Computer Graphics & Animation (IJCGA)*, 2(4), 45-51.

15. Nvidia, (2007). Nvidia CUDA (Compute Unified Device Architecture). *http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/0.8/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf* (Erişim Tarihi: 25.05.2015).

16. Nvidia, (2014). CUDA C Programming Guide. *http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3LyTDrCdH* (Erişim Tarihi: 25.04.2015).

17. Okidsu, Y., Ino, F., Hagihara, K., (2010). High-Performance Cone Beam Reconstruction using CUDA Compatible GPUs. *Parallel Computing*, 36 (2010) 129–141.

18. Şaşıoğlu, S., (2010). OpenGL ile 3 Boyutlu Arazi Modellerinin Üretimi ve Çoklu Çözünürlükte Sadeleştirilmesi.*Yüksek Lisans Tezi*, Gazi Üniversitesi Fen bilimleri Enstitüsü, Ankara.

19. Tokatlı, A., (2010). Modelling of Geological Measurements with OpenGL.*Yüksek Lisans Tezi*, Eskişehir Osmangazi Üniversitesi Fen Bilimleri Enstitüsü, Eskişehir.

20. Wang, G., Huang, L., (2011). 3D Geological Modelling for Mineral Resource Assessment of the Tongshan Cu Deposit. *Elsevier*, 3(4), 483-491.

21. Yıldız, E., (2011). Nvidia CUDA ile Yüksek Performanslı Görüntü İşleme.*Yüksek Lisans Tezi*, İstanbul Üniversitesi Fen Bilimleri Enstitüsü, İstanbul.