C64 Grafik Programlama

Bilgem ÇAKIR

C64 Grafik Programlama

Bilgem ÇAKIR

Yayımlanma 2007

İçindekiler

Giriş

Merhaba sevgili okuyucular. C64'ün grafik dünyasına, programcı olarak hükmetmenizi sağlamak için VIC programlama kursu ile karşınızdayız. Bu kursun bünyesinde şu konulara değineceğiz:

Başlangıç:

- VIC ve 6510 nasıl etkileşiyor
- VIC Ekrana nasıl bir çıkış gönderiyor.
- VIC interruptları
- Raster çizgileri
- VIC ve 6510'un cycle paylaşımı, DMA satırları
- Karakter Setleri
- Kayan yazılar(Smooth Scroll)
- Müzik çaldırma
- Logolar
- Sprite'lar
- İlk introlar

İleri teknikler:

- Titremeyen rasterlar yaratmak
- DMA satırlarının sırrı
- DMA satırlarını ertelemek (FLD)
- İstediğimiz zaman DMA satırı oluşturabilmek (FLI)
- Sprite çoğaltma (multiplexer)
- Sprite esnetme hileleri
- Ötesi...

Başlamadan önce

Bu kurs esnasında bazı konular ile hali hazırda aşina olduğunuzu varsayıyor olacağım:

- Bilgisayar mimarisindeki bazı temel kavramlar: CPU, bellek, register, cycle gibi
- 6510 assembly dili: lda, sta, ldx, stx, ldy, sty, inx, dex, jmp, jsr, rts, sei, rti, ve adresleme modları
- PC üzerinde ACME ve VICE ile C64 kodu yazıp çalıştırma.

Verdiğim örnek dosyalar hep ACME uyumlu olacak.

Kursu takip ederken bir defter bulundurmanızı ve aklınıza takılan soruları kaydetmenizi önereceğim. VIC ile ilgili pekçok konuyu tam olarak anlamanız için kursun cok büyük bir bölümünü okumuş olmanız gerekebilir. Her ne kadar kolay anlaşılabilecek bir sırada anlatmaya çalışacak olsam da anlatım boyunca eminim anlamadığınız yerler olacak. Okumaya devam ettiğinizde biraz önce aklınıza takılan sorunun cevabına rastlayacaksınız. Bu yüzden defterinizde o an aklınızda olan ve cevapsız durumdaki soruları tutarsanız, beyniniz ileriki bölümlerde cevap karşınıza çıktığında daha tetikte olacak.

C64 Grafik Mimarisi

VIC ve 6510 Nasıl Etkileşiyor

VIC ve 6510 bildiğiniz gibi iki bağımsız çip. İkisi de c64 içindeki adres ve data buslarına bağlılar. Dolayısıyla bu data bus'a bağlı olan bellek cipi ile de haberleşebiliyorlar.

Biz aslında yazdığımız programları 6510 üzerinde çalıştırıyoruz. bu yazdığımız programlar belleğe yerleşiyor ve 6510 her komutu bellekten sırasıyla okuyup işletiyor.

VIC ise grafik işlerinden sorumlu bir araç. O da aslında aktif olarak pek çok iş ile meşgul. Onun görevi bellekte bulunan bilgileri kullanarak ekrana gönderilecek görüntü sinyalini hazırlamak. Dolayısıyla VIC bellekten sürekli olarak ekran ile ilgili bilgileri (nerede hangi harf var o harfin pixelleri nasıl vs) okuyup içindeki devrelerde işledikten sonra üzerindeki iki pinden analog görüntü sinyalini üretiyor.

VIC'in görüntü sinyali üretmek dışında bir de yan görevi var. O da C64'te kullanılan dinamik bellek çiplerini tazelemek. Bunun tam olarak nasıl bir işlem olduğu bu kursun konusunun dışında.

VIC çipine belleğin neresinden hangi bilgiyi okuyup nasıl yorumlayacağını bildirebiliyoruz. Bu amaçla kullandığımız 50 civarında VIC registeri var çipin içinde. Bu registerlerin hepsi de çipin adres ve data bus pinlerine bağlı. 6510, adres ve data bus üzerinden VIC registerlerine erişebiliyor. Bu registerlere yazıp okuyabiliyor. VIC registerleri $d000 adresinden $d040 adresine kadarki bölgede bulunuyor. Dolayısıyla bizler ekrandaki görüntü ile ilgili birşeyler yaparken coğu zaman bu adres aralığındaki VIC registerlerini manipüle ediyor olacağız.

VIC ile 6510 arasında son olarak cok önemli bir bağlantı daha var. Kesinti isteği (Interrupt request veya IRQ) bağlantısı. Bu bağlantı VICten çıkıp 6510'a giren bir sinyal. Bu sinyali kullanarak VIC 6510'un o anda işletmekte olduğu komutu bitirir bitirmez başka bir bölgeye (bu sinyal olunca olmasını istediklerimizi yapan bölgeye) atlamasını sağlıyor. Bu önemli sinyal ile VIC 6510'a ekranda belli bir bölgenin çizilmeye başlamak üzere olduğunu veya iki grafik parçasının çarpıştığını (mesela bir oyunda uzay gemisinin mermiye çarptığını) bildirmek için kullanabilir.

Özet olarak 6510 ve VIC arasında üç tane haberleşme yolu var:

- bellek (6510 yazıyor VIC okuyor)
- VIC registerleri (6510 yazıyor veya okuyor)
- IRQ (VIC 6510'a birşeyleri haber veriyor)

VIC'in Çıkışı

VIC'in tam olarak nasıl çalıştığını anlayabilmeniz için VIC'in yapmaya çalıştığı şeyi anlamanız lazım. VIC video sinyali üretmeye çalışıyor. Burada bu sinyalin elektriksel özelliklerinden bahsetmeyeceğiz. Fakat genel olarak bir grup iki boyutlu pixelin bilgisinin tek bir kablo üzerinden nasıl taşındığından bahsedeceğiz.

Bir an için C64'ü unutalım. Diyelim ki sizin elinizde 100 pixele 100 pixellik bir resim var. Her bir pixel ya siyah ya beyaz olsun. Bana bu resmi konusarak iletmeniz gerekse, öyleki benim elimde 100 kareye 100 karelik bir kareli kağıt ve bir kalem olsa ve sizin bana iletmenizle ben resmi o kagıda çizsem. Ne yapardık?

Muhtemelen cok gecmeden siz bana pixelleri sırayla okumayı düşünürdünüz. Hangi sırayla? Mesela sol üstteki pixelden başlayıp sağa doğru ilerlerdiniz. “beyaz, beyaz, siyah, beyaz, siyah, siyah...” gibi. Diyelim ki ben size geri seslenemiyorum (ağzım bağlı mesela) Eğer size yetişemez de bir pixeli kacırırsam bütün pixelleri kaydırabilirim. Onun için aramızda anlaşıyoruz mesela her satırın sonuna geldiğinizde bana “bir alt satıra geç ” diyorsunuz. Böylece 100 tane renk soyleyip ardından bu komutu verdiğinizde bir alt satıra geciyorum. eğer bir şekilde bir satırda 76. rengi duyamaz veya yanlış anlarsam en kötü ihtimalle o satırdaki son 24 pixeli doğru koymuyorum. Ama alt satıra geç komutunu alınca bir alt satırdan doğru şekilde devam edebiliyorum.

Bu komik örneği biraz daha ileri götürelim. Diyelim ki ben yıllarca kareli kağıtlara pixel işaretlemekten manyaklaşmışım ve robot olmuşum. Artık bana pixellerin sabit bir hızda söylenmesi gerekiyor. mesela tam her saniye bir renk duydum duydum. bisey duymazsam sonraki pixele geciyorum. Duyarsam pixeli işaretliyorum. Hele bir de alt satıra geç komutunu duymazsam iyice salaklaşıyorum kağıdı buruşturup atıyorum ve yeni bir kagıt alıyorum. Diyelim ki benim hayatta başka işim de yok, durmadan birbiri ardına bana seslendiğiniz pıxel renkleri ile bir kağıt üstüne bir resmi çiziyorum o biter bitmez onu atıp yeni bir kagıda yeni bir resim ciziyorum. Hatta aramızda anlaşıyoruz yine aramızda senkronizasyonu sağlamak için bir resim bitince siz bana “yeni resme geç” diye bağırıyorsunuz. Tıpki “sonraki satıra geç” gibi. Bir resmi bi şekilde tamamen yanlış çizer ve toparlayamazsam en azından bir sonraki resimde normale dönebilmem için.

İşte bu örnekte ben bir televizyonum (ya da monitör). Siz ise VIC çipisiniz. Benim gibi akılsız ve hassas zamanlama ile pixel renklerini ve yeni satır veya yeni resim komutlarını duymayı bekleyen birisine sabit bir hızla bu bilgileri ulaştırmak zorundasınız. Bunun için sizin elinizde hangi pixellerin hangi renk olduğunu belirten bir resim veya kağıt var. Siz bana aynı kağıttaki aynı resmi tekrar tekrar okuyup benım her yeni sayfaya aynı resmi çizmemi sağlayabilirsiniz. Zaten TV'nizde veya monitörünüzde sabit bir ekran gördüğünüzde olan şey budur. 6510 sizin elinize sabit bir kağıt vermiştir. Siz ben salak olduğum ve bir şey gösterebilmek için sürekli olarak sizden sabit bir hızda renkleri duymam gerektiği için, pixelleri bana okursunuz.

6510 ise beraber çalıştığınız bir iş arkadaşı gibidir. Mesela ikiniz bir ofiste oturuyorsunuz ikinizin arasında kocaman bir masa var. Bu masanın üstünde her tarafta resimler rakamlar garip garip pekçok sey var. 6510 size masanın neresindeki resmi bana soylemeniz gerektiğini soyluyor. Daha sonra kendisi masanın baska bir tarafındaki rakamlara ve komutlara bakıp işlemler yapıp masanın üzerinde başka bir yerlere kaydediyor. Zaman zaman ma-

sada rastladığı bir komut yüzünen masanın öteki ucundan okumaya devam ediyor. Bazen masanın bir bölümüne bazı resimleri cizmeye başlıyor ardından size o resmi bana okumanızı söylüyor vs.

Ayrıca sizin kendinize ait bir ajandanız var. 6510 size masanın neresine bakmanız gerektiğini söyledikçe oraya kaydediyorsunuz böylece sürekli pixelleri bana okurken ajandada yazdığınız değerleri unutmuyorsunuz. Ayrıca siz bayagı yetenklisiniz. Mesela 6510 size bazen bir tane büyük resim bir tane küçük resim belirtiyor. Sonra da küçük resmin büyük resmin üstünde nereye konacağını söylüyor. Siz bütün bu bilgileri de ajandanıza kaydedip bana pixelleri okurken iki resmin üst üste binmiş halini söyleyebiliyorsunuz.

İşte bu büyük masa da bellek. Ajandanız ise VIC registerleri

Zaman zaman siz 6510'un kafasına plastik toplar atabiliyorsunuz. Bunun üzerine 6510 hemen gidip masanın özel bir ucuna bakıyor ve “hmm tamam” diyor size, plastik topu size iade ediyor ve bazı başka işlemler yapıyor. Eğer 6510 topu iade etmezse VIC elinde başka plastik top olmadığı için daha sonra 6510'u bir daha uyaramaz (kafasına plastik topu atamaz). Bu önemsiz görünen ayrıntı ileride pekçok defa programınız çalışmadığı zaman hatayı bulmanızı sağlayacak. yazdığınız koda bakıp birden farkedeceksiniz ki “aaa... 6510 plastik topu geri vermemiş ondan çalışmıyor.”

Bu komik örnek, komik olduğu kadar da bundan sonra faydalı olacak. Bir ofiste çalışan iki kanka 6510 ve VIC ve aralarındaki kocaman masa hep aklınızda olsun.

DİKKAT Çok Önemli

Bu örnekte farkedebileceğiniz çok önemli bir nokta var. Mesela siz bana bir resmin 45. satırını okuyorken eğer 6510 size masanın üstünde baktığınız yeri değiştirmenizi söylerse ne olur? örneğin masanın başka bir ucundaki 2. bir resme bakmanızı söylerse... Bu durumda siz hemen yeni resme bakmaya başladığınız için bana o resmin pixellerini söylemeye başlarsınız ve ben monitör olarak o zaman kadar 1. resmi cizmişken 46.satırdan itibaren ikinci resmi çizerim. Böylece çizdiğim resmin üst yarısı sizin masanızdaki bi bölgeden alt yarısı ise başka bölgeden gelebilir. Ben monitör olarak bunu bilemem, beni ilgilendiren tek şey pixeller ve senkronizasyon komutları.

Şu an fazla farkında olmasanız da bu aslında cok onemli ve güçlü bir özellik. Daha sonra göreceksiniz ki örneğin her satırda VIC'in resim olarak bellekte baktığı yeri değiştirerek cok etkileyici görsel efektler elde etmek mümkün. Üstelik buradaki güzellik şu. Eger bu özellik olmasa idi, ekrandaki görüntü ancak bir şekilde değişebilirdi. O da 6510'un masada VIC'in baktığı yere yeni bir resim çizmesi. Bu durumda 6510 üzerinde cok karmaşık işlemleri çok hızlı yapabilmemiz gerekir ki bu her zaman kolay veya mümkün olmayabilir. Oysa eğer 6510 önce masanın 4 değişik bölgesine 4 resim cizerse (mesela aynı resmin birazcık büyüyen 4 versiyonunu) ve ardından da her satır başında VIC'e bu 4 resimden birisini göstermesini söylese, resim dalgalanıyormuş gibi bir efekt elde edebiliriz. Üstelik bu her defasında resmin yeni bir dalgalanışını çizmeye çalışmaktan çok daha kolay ve hızlı olacaktır 6510 için. (Bkz. Şekil 1)

VIC Interruptları

Ofis örneğinden devam edelim. Diyelim ki 6510'un kulakları tıkalı.Siz Monitöre satırları ve pixelleri bağırırken duyamıyor. Ve demin bahsettiğimiz gibi ekranda belli bir yerdeki bir resmi dalgalandırmak istiyor. Diyelim ki resim ekranda dikey olarak 50.satırdan başlayıp 70.satıra kadar giden bir resim. Daha önce dediğimiz gibi 6510'yapması gereken şu: VIC tam monitöre 50 satırın pixellerini okuyacagı zaman 6510, VIC'e “birinci resme bak” diyecek. Ardından VIC tam 50. satırı bitirip 51.satırın pixellerini okuyacakken, 6510 “ikinci resme bak” diyecek. Böyle böyle 20 satır boyunca tam doğru zamanlarda (yani satır başlarında) VIC'e 6510'un komut vermesi gerekiyor

Bunun olabilmesi için 6510'un, VIC'in o an hangi satırı monitöre bağırdığını bilmesi lazım. Aynı zamanda tam olarak satırın kaçıncı pixelinde olunduğunu da bilmesi lazım. Yoksa tam olarak satır başlarında VIC'e komut vermeyi başaramaz.

İşte VIC'in 6510'un kafasına atabildiği plastik topların (yani kesinti isteğinin (IRQ)) önemi ve görevi bu noktada ortaya çıkıyor. VIC'in çok önemli bir yeteneği var. 6510, VIC'e “monitöre bilgileri gönderirken tam 50.satırın başına geldiğinde beni uyarır mısın lütfen” diyebiliyor. Bu kibar istek karşısında VIC'in 6510'un kafasına birşeyler atması biraz terbiyesizlik gibi gelse de çok önemli bir özellik ve hepimizin bazı kötü huyları var olduğunu kabul edip devam ediyoruz.

6510, IRQ sinyalini aldığı anda biliyor ki VIC tam 50. satırın başında. Artık 6510 o an her ne yapıyordu ise bırakıp derhal kafasına top yediği zamanlarda baktığı bölgeye bakıyor masada ve hemen o komutları yapmaya başlıyor. Bu örnekte yaptığı iş hemen VIC'e hangi resme bakacağını söylemek örneğin. Üstelik 6510 VIC'in pixelleri monitöre ne hızda okuduğunu bildiği için kendi içinden komutları sayarak ilerleyip tam olarak doğru anda bir sonraki satıra geçildiğini bilerek sonraki satırbaşlarında doğru zamanlarda komut verebiliyor VIC'e. Yani VIC ile 6510 bir kere senkronize olduktan sonra, bir daha her satır için VIC'in yeniden IRQ göndermesine gerek kalmıyor.

VIC'in işte bu şekilde bir satıra gelince yarattığı IRQ'lara “raster IRQ” deniyor. Raster IRQ'lar sayesinde 6510 hassas bir zamanlama ile tam doğru anlarda VIC'e komutlar göndererek çok ilginç ve çeşitli görsel efektler yaratabiliyor.

Raster Çizgileri

Şimdi buraya kadarki teorik bilgileri kullanarak ilk programımızı yazacağız. Bu programda C64'ün en klasik ve basit efektlerinden olan Raster Çizgileri ile tanışacağız. Aynı zamanda bu programı yaparken 6510'u kullanarak VIC'e “XX'inci satıra gelince beni uyar” demeyi öğreneceğiz. Yani VIC IRQ'larını programlamaya başlayacağız. Kocaman bir efektler dünyasının kapısını aralamak üzeresiniz. 25 yıldır tükenmeyen bir dünyanın...

Tipik bir VIC IRQ programı ile başlayalım.

```
!to "raster0.prg"
*=$c000
;; -----------------------------------------------
Baslangic:
```

```
jsr RasterIRQHazirla
Son:
jmp Son
;; -----------------------------------------------
RasterIRQHazirla:
sei

lda #$7f
sta $dc0d

lda $d01a
ora #$01
sta $d01a

lda $d011
and #$7f
sta $d011

lda #$20
sta $d012

lda #<IRQRutini
sta $0314
lda #>IRQRutini
sta $0315

cli
rts
;; -----------------------------------------------
IRQRutini:
inc $d019
lda #$01
sta $d020
lda #$00
sta $d020
jmp $ea81
;; -----------------------------------------------
```

Bu programda olan biteni inceleyelim.

```
Baslangic:
jsr RasterIRQHazirla
Son:
jmp Son
```

Aslında bu bölüm sadece kodun düzenli olması açısından var. Programın ana rutini diyebiliriz buna. Programımız ana rutininde önce IRQ hazırlayan alt rutine atlayıp geri dönüyor ardından da sonsuz bir döngüye giriyor. Yani 6510 normalde aynı komutta takılıp kalıyor. Sadece VIC'ten kesinti isteği geldiği zamanlarda geçici olarak IRQ rutinindeki işleri yapıyor ve o rutin bitince tekrar bu sonsuz döngüde kalmaya devam ediyor.

```
RasterIRQHazirla:
sei
```

6510 içinde “Dışarıdan Gelecek Kesinti İsteklerini Bloklama” imkanı mevcut. Bunun için Durum Registerinde Interrupt Flag adlı bir bit var. Bu bit 1 yapıldığı zaman 6510 IRQ pinine sinyal gelse bile kesinti uygulamaz ve hiçbirşey olmamış gibi devam eder. Sei komutu işte bu bit'i 1 yapar. Tersi olan cli komutu ise bu biti 0 yaparak bundan sonraki interruptlara izin verir. Yani sei ve cli komutlarının arasındaki komutlar çalışırken başka bir interrupt sinyali gelip de 6510'un herhangi bir yere atlama şansı yoktur. Bunun neden önemli olduğunu birazdan göreceksiniz.

```
lda #$7f
sta $dc0d
```

C64 içinde 6510'a kesinti isteği gönderebilen tek çip VIC değil. Disket sürücü, klavye gibi pekçok çevre birimini kontrol eden CIA çipleri de 6510'a kesinti isteği gönderebiliyor. Normalde C64 ilk açıldığında Kernal'deki başlangıç rutini içinde CIA çiplerinde bulunan timerlardan biri programlanır ve bu timer her 20 ms'de bir 6510'a kesinti isteği gönderir. Bu kesinti isteği ile beraber 6510 yine ROMdaki başka bir bölgeye atlar ve bazı rutin işlerle ilgilenir (klavyeyi kontrol etmek, cursoru yakıp söndürmek gibi). Bizim programımızda bu rutin hizmetlerin hiçbiri ile ilgilenmediğimiz için bu kesinti isteğinin gelmesini engellemek istiyoruz. Bunun için isteği gönderen CIA çipine “bana IRQ gönderme” dememiz gerekiyor. CIA çıpının ayrıntıları konumuz dışında. Fakat bu iki komutun bu işe yaradığını bilin yeter.

```
lda $d01a
ora #$01
sta $d01a
```

Burada ise VIC çipine “Bana her yeni sayfada X'inci satıra gelince haber ver” demek için gereken adımları atmaya başlıyoruz. Bu adımlar şöyle:

- VIC'de Raster IRQ fonksiyonunu aktifleştirmek
- Raster IRQ sinyalinin gönderilmesini istediğimiz satırı belirtmek

Bu adımlardan ilki için VIC'in $d01a adresindeki registeri kullanılıyor. Bu registerin en küçük biti Raster IRQ aktifleştirmeye yarıyor. Diğer bitler başka koşullarda oluşan IRQ'ları etkilediği ve biz onları değiştirmek istemediğimiz için bu üç komutu kullanıyoruz. Böylece diğer bitlerin değerini değiştirmeden sadece en küçük biti 1 yapıyoruz.

```
lda $d011
and #$7f
sta $d011

lda #$20
sta $d012
```

Bu komutlarla da ikinci adım gerçekleştiriliyor. Ne olduğunu anlamamız için biraz düşünmemiz lazım.

VIC Cipinin monitöre gönderdiği görüntü PAL sistemlerde tam 312 satırdan oluşuyor. Buna ekranın üstündeki ve altındaki Çerçeve bölgeleri de dahil. Dikkat ederseniz 312 sayısı tek bayt ile ifade edilemeyecek kadar büyük bir sayı. Bu büyüklükteki sayıları ifade etmek için gereken en az bit sayısı 9. 9 bit ile 512'ye kadarki sayılar ifade edilebilir. İşte bu yüzden VIC'e hangi satırda IRQ istediğimizi belirtmek için 9 bit göndermek gerekiyor. Bu 9 bit de VIC'in $d012 adresindeki registerinde bulunan 8 bit ve $d011 adresinde bulunan registerin en üst bitinden oluşuyor. Yani eğer 256'dan küçük bir değer yazmak istersek, $d011'in en üst bitine 0 yazıyoruz. Bu örnekte olduğu gibi.

d011 ve d012 VIC içindeki en hayati önemli registerler arasında. İkisinin farklı farklı ve beraber kullanımlarından inanılmaz efektler ortaya cıkıyor.

d012 yazıldığında ve okunduğunda farklı anlamlara gelen bir register. Okuduğumuz zaman bize VIC'in o an ekrana hangi raster satırını göndermekte olduğunu söylüyor. Yazdığımızda ise, eğer Raster IRQ aktifleştirilmiş ise (d01a ile) IRQ'nun olmasını istediğimiz satırı VIC'e belirtmiş oluyoruz. VIC bu belirttiğimiz satıra gelince IRQ sinyali gönderecek.

Böylelikle VIC'in istediğimiz satırda 6510'a IRQ göndermesini hazırlamış olduk. Peki bu IRQ sinyali geldiğinde 6510 ne yapacak. Önceki bölümlerde anlattığımız gibi 6510'un o an yaptığı işi bırakıp (ki bizim programımızda o an yaptığı iş sadece bir sonsuz döngü) özel olarak olmasını istediğimiz şeyi yapmaya başlaması lazım. Bunun için aşağıdaki komutları kullanıyoruz.

```
lda #<IRQRutini
sta $0314
lda #>IRQRutini
sta $0315
```

Böylelikle IRQ gelince olmasını istediğimiz şeyler için bir rutin yazıyoruz ve bu rutin bu örnekte IRQRutini adresinden başlıyor.

6510 IRQ sinyali aldığı zaman eğer IRQ biti 1 ise ROM'da bazı işlemler yaptıktan sonra bellekte $0314 ve $0315 adreslerinde alt ve üst baytı bulunan adrese atlar. Biz de burada $0314 ve $0315 içine IRQ rutinimizin başlangıç adresini yazıyoruz. Böylelikle 6510, VICten gelen IRQ sinyalini alınca bizim IRQ rutinimize atlayacak.

```
cli
rts
```

Artık bütün hazırlıkları tamamladığımız için cli komutu ile IRQ sinyallerini yeniden dikkate almaya başlayabiliriz. Eğer sei ve cli kullanmasaydık ne olurdu. Düşünün tam biz $0314'e IRQ rutinimizin adresinin alt baytını koyduktan sonra herhangi bir sebepten IRQ sinyali gelse 6510 $0314 ve $0315'teki adrese atlamak isteyecek. Oysa bu iki bayttan yalnızca biri dogru. 0315'i henüz yazmadık çünkü. Böylece 6510 alakasız bir adrese atlayıp büyük olasılıkla da kilitlenecektir. Bunun olmaması için biz IRQ koşullarını ve adreslerini değiştirirken geçici olarak IRQ sinyallerini görmezden geliyoruz. cli'nin hemen ardından rts ile ana programa dönülüyor.

```
IRQRutini:
inc $d019

lda #$01
sta $d020
lda #$00
sta $d020

jmp $ea81
```

Burada ise IRQ rutinini görmekteyiz. Rutin gerçekten çok basit bütün yaptığı çerçeve rengini önce beyaz yapıp hemen ardından da siyah yapmak. Bu rutinden sonra bütün ekran boyunca cerceve rengi siyah kalmaya devam edecek ve VIC yeni bir sayfayı monitöre gönderirken tekrar $20.satıra geldiğinde 6510 bu rutine girecek.

Bu rutinin sonucu olarak siyah çerçeve içinde tepede bir yerde kısa bir beyaz çizgi göreceğiz.

ea81 adresi ROM'da bulunan bir rutinin adresi bu rutinde IRQ ilk olduğunda yapılan işlemlerin tersleri yapılıyor ve CPU kesinti modundan çıkıp IRQ sinyali geldiği anda yapmakta olduğu işe dönüyor. Dolayısıyla biz bir miktar ilerleyene kadar ki örneklerimizde IRQ rutininden dönmek için hep jmp $ea81 komutunu kullanacağız.

Bir de en başta inc $d019 komutu var. Bu ne işe yarıyor diye merak ediyorsunuz. VIC ve 6510 arasındaki plastik topu hatırlayın. İşte bu komut ile 6510, VIC'e plastik topu geri veriyor. VIC'in d019 adresindeki registeri VIC tarafından oluşturulan IRQ sinyallerine karşı 6510'un “Hmm tamam. IRQ sinyalini aldım ve işledim. Bundan sonra tekrar olursa tekrar bana haber ver” demesini sağlıyor. Daha önce de belirttiğimiz gibi, VIC çeşitli IRQlar sinyalleyebildiği için 6510 da bu farklı IRQlardan hangilerini işlediğini ayrı ayrı bildirmek durumunda. Bu yüzden $d019 içindeki farklı bitler farklı IRQları işlediğimizi bildirmek için kullanılabiliyor. d01a da olduğu gibi ilk bit Raster IRQ biti. d019 ilk bitine 1 yazarsak VIC'e Raster IRQ sınyalını aldığımızı ve gerekeni yaptığımızı söylemiş oluyoruz.

d019 aynı zamanda VIC IRQ sinyalini 6510a göderdiği anda ilgili bitini sıfırlıyor. Yani bu bit 1 iken VIC d012'de yazılı olan satıra geliyor. IRQ sinyali gonderiliyor ve hemen bu biti VIC 0 yapıyor. Ardından 6510 IRQ sinyalini işledikten sonra onu tekrar 1 yapıyor.

Dolayısıyla Biz IRQ rutinimize girdiğimiz anda $d019'da 0 yazdığını biliyoruz. Bu yüzden inc komutu verdiğimizde aslında dolaylı olarak o biti 1 yapmıs oluyoruz. Bunun yerine lda #01 , sta $d019 da diyebiliriz. Fakat bunun yerine pek çok programcı inc komutunu kullanır. Burada bunu kullanmamızın sebebi, başka programcıların kodunda bunu görürseniz şaşırmayın diye.

Programdaki Problemler

Burada iki şey canınızı sıkabilir:

- Beyaz çizgi satırın ortasında başlıyor. Ben satırın başında başlayıp sonunda biten bir çizgiyi nasıl yapacağım?
- Bu çizgi sağa sola titriyor bunu sabitlemenin bir yolu yok mu?

Birinci soruyu cevaplamak daha kolay. Hatırlayın IRQ sinyali gelince 6510 önce ROMda bazı işlemler yapıyor ondan sonra bizim rutinimize atlıyor demiştik. İşte bu Rom rutininde yapılan işler zaman aldığı (cycle tükettiği) için sıra bizim komutlarımıza gelene kadar VIC monitöre o satırın bilgilerini soldan sağa göndermeye devam ediyor.

Demekki IRQ sinyalin alır almaz birşeyler yapmamız mümkün değil şu anda. Öyleyse yapmamız gereken şey renk değişikliğini yaptığımız komuttan önce biraz daha zaman kaybetmek ve bu

esnada VIC'in bir alt satıra geçmesini sağlamak.

```
IRQRutini:
inc $d019

ldx #4
gecik1:
dex
bne gecik1

lda #$01
sta $d020
lda #$00
sta $d020
jmp $ea81
```

Programı böyle denerseniz beyaz çizginin bir alt satırın başından başladığını göreceksiniz. Peki çizgiyi uzatmak için ne yapmalıyız? Tabii ki rengi siyaha çevirmeden önce oyalanmalıyız bu sefer de. Yani IRQ rutinimizin son hali aşağıda

```
IRQRutini:
inc $d019

ldx #4
gecik1:
dex
bne gecik1

lda #$01
sta $d020

ldx #$0a
gecik2:
dex
bne gecik2

lda #$00
sta $d020
jmp $ea81
```

Buradaki gecikme değerleri (ldx ile X registerine yüklenen değer) ile oynayarak değişik uzunluklarda çizgiler elde edebilirsiniz.

İkinci Problem yani cizgilerin titremesi ise çok daha derin bir konu. Problemin kaynağına şimdi kısaca değineceğiz ama çözümü kursun ileri teknikler bölümünde olacak.

6510 IRQ sinyali aldığında o an işletmekte olduğu komutu tamamlayıp ROM'a atlar demiştik. İşte o an işlenmekte olan komut çeşitli cycle uzunluklarında olabilir. 6510 komutları 2 cycle'dan 7 cycle'a kadar olabilir. Ayrıca IRQ sinyali komutun işlendiği cycle'lardan herhangi biri anında gelebilir. Mesela bizim örneğimizde durum nispeten basit. IRQ geldiği anda işlenen komutun jmp olduğundan eminiz. Çünkü ana programımız sonsuz döngüde kalıyor. Jmp komutu 3 cycle. Yani IRQ komutunun bu üç cycle'dan hangisinde geldiğine bağlı olarak, 6510 kesinti ile ilgili işlemlere bşlamadan önce 0 veya 1 veya 2 cycle harcayarak jmp komutunu tamamlayacak. VIC'ten IRQ sinyali her sayfada farklı zamanlarda gelebilir. Bu yüzden her sayfada 6510 bizim çizgiyi çizdiğimiz komutu satırbaşından üç farklı uzaklıkta işletiyor. hız olarak 6510da bir cycle'lık gecikme VIC'in 8 pixel gönderme zamanına tekabül ettiği için programın ilk versiyonunda çizgimiz aralarında 8'er pixel bulunan 3 pozisyondan rastgele bir şekilde titriyor görünüyordu. Eklediğimiz gecikmelerle bu tıtreyen bölümü ekranın dışında görünmeyen bölgeye sakladık (yani bir satırın sonu ile bir sonraki satırın başı arasındaki bölgeye). Ancak bu konunun kesin çözümü bu değil. Şimdilik bu çözümle idare edeceğiz. Gerçek çözüm kursun ileri teknikler bölümünde “Titremeyen Rasterlar Yaratmak” bölümünde anlatılacak.

Raster Bar Efektleri

Selam sevgili okuyucular. VIC kursuna kaldığımız yerden devam ediyoruz. Hatırlayacağınız gibi bir önceki bölümün sonunda ekranda tek bir raster çizgisi göstermiştik. Bu bölüme daha çok raster cizgisi göstererek başlayacağız.

Programımızın IRQ rutini bölümü değişecek sadece. En başta IRQ hazırlama rutini aynen kalacak. Sadece IRQ kesintisinin olacağı satırı değiştirip biraz daha ekranın ortasına alacağız.

IRQ Rutininde tek raster çizgisi koyarken ne yapmıstık?

- satır başına gelene kadar bekle
- renk değiştir
- bir sonraki satıra kadar bekle
- ekranın efektten sonraki rengini ver
- IRQ'dan çık (jmp $ea81)

Bu sefer birden fazla raster satırına renkler vereceğimiz için şöyle bir değişiklik yapıyoruz

- satır başına gelene kadar bekle
- renk değiştir
- bir sonraki satıra kadar bekle
- renk değiştir
- bir sonraki satıra kadar bekle

...

- renk değiştir
- bir sonraki satıra kadar bekle
- ekranın efektten sonraki rengini ver
- IRQ'dan çık (jmp $ea81)

Efektin dikey olarak ekranda ne kadar kalın olmasını istersek, o kadar çok satır için bu işlemi tekrarlayacagız. Bu örnekte $60 satır için bu işlemi tekrarlayacağız.

Programın tamamını raster2.a64 adlı dosyada görebilirsiniz. Biz şimdi burada IRQ rutinini inceleyeceğiz.

```
;; -----------------------------------------------
IRQRutini:
inc $d019

ldy #2
gecik1:
dey
bne gecik1

ldx #$00
dongu:
lda renk_tablo,x
sta $d020
sta $d021


ldy gecikme_tablo,x
gecik2:
dey
bne gecik2


inx
cpx #$60
bne dongu


lda #$00
sta $d020
sta $d021
jmp $ea81
;; -----------------------------------------------
```

Şimdi detaylara geçelim.

```
IRQRutini:
inc $d019

ldy #2
gecik1:
dey
bne gecik1
```

Buraya kadarki bölüm önceki program ile aynı. IRQ sinyalini aldıktan sonra çizgilerimizin düzgün bir şekilde satır başından başlamaları için küçük bir gecikme ile başlıyoruz.

```
ldx #$00
dongu:
```

$60 tane satırı tek tek kodlamak istemediğimiz için bir döngü hazırlıyoruz. Döngümüzde sayaç olarak X registerini kullanacağız.

```
lda renk_tablo,x
sta $d020
sta $d021
```

Satırların tek tek alacakları renkleri kolay erişim ve editleme için bellekte bir tabloya koyacağız. Tablo diye bahsettiğimiz şey bellekteki ardışık bir grup bayttan başka birşey değil. Bu komutlarla indexli adresleme kullanarak bellekte "renk_tablo" adresinden başlayan tablomuzdan sıradaki rengi (yani n'inci rengi, n X registerinde o an yazılı olan değer) alıp, VIC'in ekran ve çerçeve rengi registerlerine gönderiyoruz.

```
ldy gecikme_tablo,x
gecik2:
```

```
dey
bne gecik2
```

Bu komutlarla her satırda bir gecikme yapıyoruz. Dikkatli okurların aklına şu soru gelebilir. Neden gecikme değerlerini bir tablodan okuyoruz? VIC ekrana datayı sabit hızda gönderiyordu. Dolayısıyla bizim her satırda beklememiz gereken süre sabit olmalı, çünkü VIC bir alt satıra hep sabit bir hızla geçiyor olmalı.

Bu çok haklı bir soru. Cevabı ise biraz karmaşık. Şu an cevabın sadece bir bölümünü vereceğiz. Devamını kursun bu bölümünde bulacaksınız.

VIC'in ekrana bilgileri sabit hızda gönderdiği doğru. Fakat VIC bunu yapabilmek için her 8 satırda bir kere 6510'u kısa bir süreliğine "donduruyor". "Nasıl yani?" dediğinizi duyabiliyorum. Merak etmeyin daha sonra bu konuyu bütün detayları ile anlatacagım. Şu an anlamanız gereken şey şu. 8 satırda bir kere VIC 6510'u kısa bir süre durduruyor ve bazı işlemler yapıp sonra tekrar başlatıyor. Bu esnada ekrana sinyal göndermeye devam ediyor çünkü ekran ondan pixel bilgilerini sabit bir hızda almak zorunda. Düşünün... Eğer 6510 durdurulduğu satırda, diğer satırlarda işlettiği kadar komut işletmeye kalkarsa geç kalır. Çünkü zaten durdurulduğu için zaman kaybediyor.

İşte bu yüzden bizim yazdığımız kodun da 8 satırda bir diğer satırlardan daha az bekleme yapması gerekiyor. Yukarıda gördüğünüz komutlarla, her defasında bekleme miktarını tablodan alarak değiştiren bir gecikme rutini yazmış oluyoruz.

```
inx
cpx #$60
bne dongu
```

Burada da bütün raster satırlarımızı çizmeyi sağlayan ana döngümüzün kontrol noktasındayız. Döngünün sayacı olan X registerindeki değeri artırıp, henüz $60 olup olmadığını kontrol ediyoruz.Buradaki $60 değerini azaltarak efekt alanını daraltmayı deneyebilirsiniz

```
lda #$00
sta $d020
sta $d021
jmp $ea81
```

Bu komutlara vardığımızda artık bütün satırları çizmiş durumdayız. O yüzden ekrana efektin dışındaki bölgede alacağı rengi verip, IRQ rutininden çıkıyoruz.

Raster2.a64 dosyasına baktığınızda IRQ rutininin hemen ardından !align 255,0 satırını göreceksiniz. Bunu takiben de her biri $60 bayt uzunluğunda olan renk ve gecikme tablolarını göreceksiniz. Burada align satırı ile tabloların C64 belleğinde tam olarak bir sonraki bloğun başından (yani $c100) başlamasını sağlıyoruz. Bunu neden yaptığımızı açıklayalım.

6510 da indexli adresleme komutları (lda renk_tablo,x vs) her zaman sabit hızda (yani sabit sayıda cycle yiyerek) çalışmazlar. Eğer index registerindeki (X veya Y) değer tablonun taban adresine (mesela renk_tablo) eklendiği zaman bir sonraki bloktan bir adres elde ediliyorsa, normalden 1 cycle fazla harcanır. Örneğin

```
lda $c080,x
```

komutunun kac cycle sureceği X registerindeki değere göre değişir. Eğer X registerinde $7f veya daha küçük bir değer varsa bu komut 4 cycle sürer. Oysa X registerinde $80 veya daha büyük bir değer varsa bu komut $c100den sonraki bir adrese (yani $c080'in içinde olduğundan bir sonraki bloğa) eriştiği için 5 cycle sürer.

Biz kodumuzda bu kadar hassas bir zamanlama ile gecikme döngüleri yazmaya çalışıyorken, o döngüleri yazarken kullandığımız komutların hızları da değişirse çok zor durumda kalırız. Bunun olmaması için kullandığımız tabloları bellekte öyle yerleştirmeliyiz ki, bir bloktan taşıp başka bloğa yayılmasınlar. Burada 2 adet $60 baytlık tabloyu bir bloğun başından başlatarak emin oluyoruz ki ikisine de erişirken hep sabit hızda çalışacak komutlarımız.

Bu programı anladığınız zaman biraz renk tablosundaki değerlerle oynayın. Hatta biraz Gecikme tablosundaki değerlerle de oynayarak raster efektlerinde zamanlamayı tutturamazsanız oluşan görüntüleri de görün. Üstüne gidin bir çay için, bir tenefüs verin sonra gelin devam edelim. Şimdi çok eğlenceli ve "janjanlı" bir bölüme başlayacağız.

Hareketli Rasterlar

Raster2.prg'ye yeterince uzun süre baktıktan sonra herkes biraz hareket ister. Mesela bu raster çizgileri aşağı yukarı kayabilir. Raster çizgileri ile yapılmış uzun barlar birbirinin içinden geçebilir. Yapılabilecek tonlarca değişik ve özgün raster çizgisi efekti olabilir.

Bu efektleri nasıl yapabiliriz düşünelim. Her yeni sayfada ekrana farklı raster cizgileri çizmek istiyoruz. Mesela basit bir örneği ele alalım.

Yapmak istediğimiz efekt toplam 8 satırlık bir raster cizgi grubunu ekranda aşağı yukarı oynatmak olsun. Basit gibi görünen (ve aslında doğru yaklaşıldığında basit olan) bu problemi çözmeye çalışırken çok temel bazı intro teknikleri ile tanışacaksınız.

Düşünelim... 8 satırlık bir raster cizgi grubumuz var. Mesela $80 ve $88. satırlar arasında olsun bunlar. Biz her sayfada bunları bir aşağıya kaydırmak istiyoruz diyelim.

Bunun için VIC'ten gelen IRQ sinyalini her sayfada bir aşağıya kaydırmayı düşünebiliriz. Yani ilk sayfada VIC bize $80. satırda IRQ gönderir. ikinci sayfada $81. satırda vs... Biz bu sayede her sayfada bir alt satırdan itibaren renkleri değiştiriyor olacağımızdan, her yeni sayfada çizgiler bir satır aşağı kaymış görünür. Bunu yapmak için bütün yapmamız gereken IRQ rutinimizin sonunda VIC'in $d012 registerine yeni bir değer yazarak bir sonraki seferde IRQ'nun bir satır sonra gelmesini sağlamak olur.

```
ldx efekt_baslangic_satiri
```

```
stx $d012
inx
stx efekt_baslangic_satiri
```

Tam IRQ rutininden çıkmadan hemen önce (jmp $ea81 den önce) bu 4 satırı ekler ve bellekteki herhangi bir boşluğa efekt_baslangic_satırı baytını tanımlarsak, raster cizgilerimiz hareket etmeye başlar.

Fakat bunu yapmayı denediğinizde ciddi bir problemle karşı karşıya kalacaksınız.

Bu yaptığımız değişiklikle istediğimiz sonucu elde edebilmemiz için satır zamanlamalarının aynı olması gerekirdi. Fakat demin bahsettiğimiz gibi 8 satırda bir zamanlama değiştiği için problemle karşılaşıyoruz. IRQ rutinimizdeki ve gecikme tablomuzdaki zamanlamalar rutin $80. satırdan başladığı zaman gecerli. $81. satırdan bu rutini başlatırsak, VIC'in 6510'u durdurduğu satıra hesapladığımızdan bir satır erken geliyoruz. Bizim rutin normal bir satırda olduğunu zannederek uzun süre gecikme yapıyor. Bu da oradan sonraki bütün satırlardaki zamanlamaları alt üst ediyor.

Peki ne yapmalıyız? Bir çözüm yolu gecikme tablosunu okumaya her sayfada bir bayt sonradan başlamak olabilir. Mesela IRQ 80. satırda geldiğinde

```
ldy gecikme_tablo,x
```

daha sonra bir sonraki seferde IRQ 81. satırda gelince

```
ldy gecikme_tablo + 1 ,x
```

şeklinde okursak, her zaman gecikme tablosundaki aynı adresin ekrandaki aynı satırın gecikme rutinine denk gelmesini sağlamış oluruz. Bunun için de IRQ rutinimizin sonuna küçük bir ekleme yaparak rutinin kendi kendisini değiştirmesi sağlanabilir.

```
inc gecik2-2
```

gibi...

Fakat burada rasterlarınıza yaptırmak istediğiniz hareketler karmaşıklaştıkça bu yaklaşım ile işleriniz giderek zorlaşacaktır. O yüzden aslında hareketli rasterlar yapmanın çok daha kolay ve esnek bir yolu var.

Şimdi tekrar düşünelim... Rasterları hareket ettirmek istiyoruz. Fakat zamanlama çok hassas olduğu için IRQ sinyalinin bize hep aynı satırda gelmesi işimize gelir. Raster2 programımızdaki IRQ rutininde her satır için renkleri renk tablosundan okuyorduk. Gecikmeleri de tablodan okuyarak tam temiz bir görüntü elde etmiştik. O zaman bu düzeni hiç bozmayalım.

Bu rutin bize $80. satırdan $e0. satıra kadar düzgün bir şekilde renk tablosundaki değerlerde raster çizgileri çiziyor nasıl olsa. Biz bu çizim biter bitmez bir sonraki sayfada olmasını istediğimiz görüntüyü renk tablosuna çizelim ve IRQ rutininden çıkalım.

Böylelıkle aslında animasyon ile ilgili problemlerimiz (hangi satır hangi sayfada hangi renk olacak) ile zamanlama ile ilgili problemlerimizi (hangi satır ne kadar gecikme ile çizilecek) birbirinden soyutlamış oluyoruz. Bu soyutlama bize büyük bir esneklik sağlıyor. Artık istediğimiz hareketi rasterlara yaptırırken ekleyip çıkaracağımız her komutun zamanlamayı etkilemesi kaygısından kendimizi kurtarmış oluyoruz. Çünkü biz renk tablosunda istediğimiz değişiklikleri yaparken VIC efektin olduğu bölgeyi düzgün bir şekilde monitöre göndermiş ve efektten sonraki satırları göndermekte. Biz zaten orayı siyah yapıp geçtik. O bölgede artık hassas bir zamanlama ile VIC registerleri ile oynamıyoruz. O yuzden istediğimiz hızda renk tablosuna bir sonraki sayfanın renklerini çizebiliriz. Tabii çok da fazla zaman kaybetmek istemeyiz.

Bütün yapmamız gereken Raster2'deki IRQ rutininin sonuna jmp $ea81'den önceye

```
jsr renkleri_degistir
```

diye bir satır ekleyip sonra da renk değiştirme alt programını yazmak.

Renk değiştirme ile hareketli rasterlar

```
;; ------------------------------------------------
renkleri_degistir:
jsr renkleri_temizle

ldx bar_pozisyonu_okunacak_index
ldy sin_table+$00,x
jsr bar_ciz

inc bar_pozisyonu_okunacak_index
rts
;; ------------------------------------------------
renkleri_temizle:
ldx #$00
lda #0
rt_dongu1:
sta renk_tablo,x
inx
cpx #$60
bne rt_dongu1
rts
;; ------------------------------------------------
bar_ciz:
lda tek_bar + 0
sta renk_tablo+ 0,y
lda tek_bar + 1
sta renk_tablo+ 1,y
lda tek_bar + 2
sta renk_tablo+ 2,y
lda tek_bar + 3
sta renk_tablo+ 3,y
lda tek_bar + 4
sta renk_tablo+ 4,y
lda tek_bar + 5
sta renk_tablo+ 5,y
lda tek_bar + 6
sta renk_tablo+ 6,y
lda tek_bar + 7
sta renk_tablo+ 7,y
```

```
rts
;; ------------------------------------------------
bar_pozisyonu_okunacak_index:
!by 0
```

Şimdi bu kodu inceleyelim. Burada gördüğümüz bölüm, her sayfada bir defa çağrılıp, bir sonraki sayfada gösterilmek üzere renk tablosunu hazırlıyor. Bunun için şu adımları takip ediyor.

- renk tablosunu temizle (siyah ile doldur)
- animasyona göre olması gereken yere 8 satırlık bir bar çiz
- gelecek sefer için animasyondan bir sonraki değeri okumak üzere rutini hazırla

Bu adımları düşünerek alt rutinlerimizi planlıyoruz. Tablo temizlemeyi bir alt rutin yapıyoruz. Bir satırdan başlayan bar çizmeyi alt rutin yapıyoruz, çünkü sonra birden fazla bar çizmeyi isteyebiliriz. Alt rutini öyle ayarlayalım ki ona atlamadan önce bir registere renk tablosu içinde barın başlamasını istediğimiz satırı verelim ve rutin barı oraya çizsin.

```
renkleri_degistir:
jsr renkleri_temizle
```

İşte bu ana rutinin başı ekran temizleme rutinine atlayıp dönüyoruz.

```
ldx bar_pozisyonu_okunacak_index
ldy sin_table+$00,x
jsr bar_ciz
```

Bu komutlar ise kritik bir triği gösteriyor. Bunu açıklamamız gerekli.

Animasyon Tabloları

Birşeyleri anime etmek istediğimiz zaman en kolay yol nedir? Mesela bu örnekte her sayfada baska bir yere cizmek istediğimiz bir bar var. Yani barın başlangıç pozisyonu her sayfada yeni bir değer alacak. Öyleyse her sayfada alacağı değeri art arda baytlara koyarak bir tablo yapabiliriz. Her sayfada o tablodan sıradaki değeri okuyup barımızı okuduğumuz yerden başlayarak çizersek problem kalmaz.

Bu örnekte bar pozisyonunu anime etmek için bir sinüs tablosu kullanıyoruz. Sinüs tabloları sin cos gibi trigonometrik fonsiyonların çeşitli şekillerde birleştirilmesi ve ardından bu fonsiyonlara 128, 256 gibi işimize yarayacak sayıda değer sokulup çıktıların bir tabloya saklanması ile oluşturuluyor.

Bizim kullandığımız sinüs tablosu bize 0 ile $58 arasında yumuşak bir şekilde salınan 256 adet değer sağlıyor.

Dolayısıyla

```
ldx bar_pozisyonu_okunacak_index
ldy sin_table+$00,x
jsr bar_ciz
```

komutlarında yaptığımız şey basit aslında. Şu anda barın pozisyonunu, sinus tablosundan okuyacağımız yeri, bar_pozisyonu_okunacak_index adresinde saklıyoruz. Bu değeri index olarak kullanıp, sinus tablosundan istediğimiz değeri alıp y registerine koyup bar çizme rutinine atlıyoruz. Bu rutin Y registerinde yazılı pozisyona çizecek barı.

```
inc bar_pozisyonu_okunacak_index
rts
```

Ve son olarak da sonraki sayfada animasyonun bir sonraki değerini okumak üzere indexi bir artırıyoruz.

```
renkleri_temizle:
ldx #$00
lda #0
rt_dongu1:
sta renk_tablo,x
inx
cpx #$60
bne rt_dongu1
rts
```

Burada açıklanacak çok birşey yok. Bir döngü ile renk tablosunu sıfırlıyoruz

```
bar_ciz:
lda tek_bar + 0
sta renk_tablo+ 0,y
lda tek_bar + 1
sta renk_tablo+ 1,y
```

İşte bu da bar çizme rutinimiz. Bu rutinin amacı renk tablosu içindeki dogru adresten başlayarak barımızı oluşturan 8 rengi koymak. Bunun için 8 tane lda sta çifti kullanıyoruz. bir döngü yerine tek tek yapmamız sizi şaşırtabilir. Zaman zaman kısa döngüleri açıp yazmak daha pratik olabilir. Ayrıca döngü registeri olarak kullanmadığınız için X ve Y registerlerini başka amaçlar için kullanma şansınız olur. Bu örnekte Y registeri ile alt programa arguman taşıdık mesela.

Ayrıca renk tablosundaki adresleri bu rutinde nasıl kullandığımıza dikkat edin mesela Y değeri $10 olsa ilk sta komutu

```
renk_tablo + 0 + $10 = renk_tablo + $10
```

İkinci sta komutu

```
renk_tablo + 1 + $10 = renk_tablo + $11
```

Sekizinci sta komutu

```
renk_tablo + 7 + $10 = renk_tablo + $17
```

adreslerine erişerek doğru işlemi yapacaklar.

```
bar_pozisyonu_okunacak_index:
!by 0
```

Son olarak bu satırlarda da sakladığımız animasyon indexi için bellekte bir baytlık yer ayırmış oluyoruz.

Gerçekten çok kritik bir bölümü daha tamamladık şu an. Şimdi biraz eğlenelim

Hareketli Barlar

Şimdi küçük bir değişiklik ile efektimizin etkileyiciliğini 8 kat artıralım

```
renkleri_degistir:
jsr renkleri_temizle

ldx bar_pozisyonu_okunacak_index
ldy sin_table+$00,x
jsr bar_ciz

ldy sin_table+$04,x
jsr bar_ciz

ldy sin_table+$08,x
jsr bar_ciz

ldy sin_table+$0c,x
jsr bar_ciz

ldy sin_table+$10,x
jsr bar_ciz

ldy sin_table+$14,x
jsr bar_ciz

ldy sin_table+$18,x
jsr bar_ciz

ldy sin_table+$1c,x
jsr bar_ciz


inc bar_pozisyonu_okunacak_index
rts
```

Artık burada tek bir bar çizmek yerine birbirini 4 sayfa arkadan takip eden 8 bar çiziyoruz. Bar çizmeyi alt rutin yapmanın iyi bir fikir olduğunu söylemiştim.

ayrıca en altta sinüs tablosunu iki kere tekrar etmemiz gerekiyor.

```
!align 255,0
sin_table:
!src "sin_table_00.a64"
!src "sin_table_00.a64"
```

bunu yapmaz isek ne olacağını düşünün. bar_pozisyonu_okunacak_index değişkeninin değerinin $e4 olduğu anı düşünün (elbet bir an gelecek olacak çünkü sürekli birer birer artırıyoruz kendisini) Bu anda

```
ldy sin_table+$1c,x
```

komutu 256 baytlık sinüs tablosundan taşıp bir sonraki alakasız bir baytı okur. Bunu istemediğimiz için sinüs tablosunu devam ettiriyoruz. Elimizdeki tablo öyle ayarlandı ki sonundan başına geçiş yumuşak ve devamlı bir geçiş.

Sonuç heyecan verici değil mi?

Eğlenmeye devam... Renkleri temizlemek yerine raster2 programındaki güzel renk tablomuzu arka plan olarak kullanmaya ne dersiniz. Bunun için bütün yapmamız gereken renkleri_temizle rutini yerine arka_plan_kopyala rutini kullanmak tıpkı Hareketlı raster 2 programındaki gibi.

```
renkleri_degistir:
jsr arka_plan_kopyala

...
...
;; -----------------------------------------------
arka_plan_kopyala:
ldx #$00
apk_dongu1:
lda arka_plan,x
sta renk_tablo,x
inx
cpx #$60
bne apk_dongu1
rts
```

Bu kadar basit... Şu an itibariyle old school efektler aleminde gayet geçerli sayılabilecek süper görünen bir raster efekti elde etmiş bulunuyorsunuz. Bu rutinleri kendi kendinize kurcalamayı deneyerek daha kimbilir neler yapabilirsiniz.

Kursun bu bölümünü bitirmeden önce son olarak çok önemli bir konu var. İsterseniz bir tenefüs verip öyle gelin...

DMA Satırları

İşte bunlar şu başımızın belası gibi görünen 8 satırda bir olan VIC'in 6510'u durdurduğu satırlar.

Bu satırlarda ne olup bittiğini anlamamız için biraz VIC çipinin içine girmemiz gerekiyor. VIC çipinin grafik üretme şekli karakter tabanlı bir sistem. Yani VIC ekranı çizerken aslında 40 x 25 adet karakter slotu cizmek üzere davranıyor. bu 40 x 25 karakter slotluk ekrandaki her bir karakter slotu da 8 x 8 pixelden oluşuyor.

VIC görüntüyü oluşturmak için belleği tararken aslında yaklaşık olarak şu adımlarda düşünmesi gerekir.

- Su anki karakter slotuna hangi karakteri çizeceğim.
- Bu karakteri oluşturan 8x8 pixelin her biri ne renk.

Fakat hatırlayacağınız gibi VIC görüntüyü satır satır oluşturduğundan yukarıdaki adımların her karakter için işlenmesinde ufak bir fark vardır. Şimdi detaylı şekilde bunu anlatalım.

Önce terimler hakkında bir uyarı. Şimdiye kadar "satır" kelimesini bir pixel kalınlığındaki raster satırı anlamında kullandık. Bu noktadan itibaren "karakter satırı" diyerek 8 pixel kalınlıktaki satırlardan da bahsetmeye başlayacağız.Bu ikisini karıştırmayın.

VIC'in içinde bizim dışarıdan erişemediğimiz 40 tane register vardır. VIC bir karakter satırının başındaki ilk raster satırına geldiğinde, bellekten Video Matrix adını verdiğimiz bölgeden (40 x 25 = 1000 baytlık bir bölge), o karakter satırını oluşturan 40 baytı alır ve bu içindeki registerlere saklar. Başka bir deyişle VIC, önümüzdeki 8 raster satırı boyunca oluşturacağı görüntüde gecen 40 karakter slotunu göndermek üzere kendini hazırlamış olur.

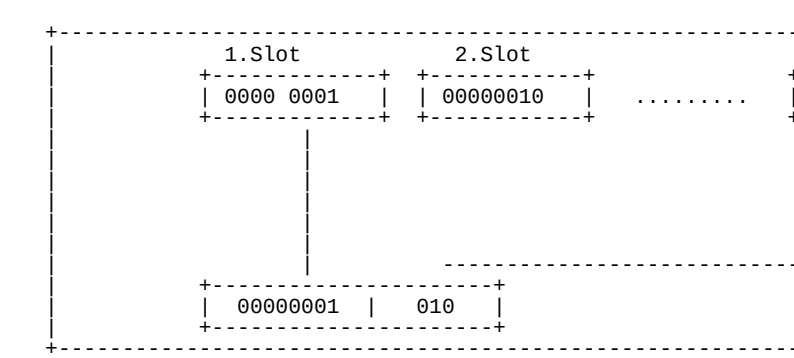
Daha sonra o kırk karakterin her birinin 8x1'lik ilk sıra pixelini bellekten okuyarak ilk raster satırının video sinyalini oluşturup monitöre gönderir. Ardından bir sonraki raster satırında bu aynı 40 karakter'in 8x1'lik ikinci sıra pixelini okur ve kullanır. Bu islem boylece 8 satır boyunca devam eder.

Bu tasarım sayesinde VIC 40 karakteri bir kere okuduktan sonraki 7 satırda donanım olarak çok basit ve hızlı şekilde (sadece 7'ye kadar sayan bir iç sayacı artırarak) bellekten okuması gereken baytları bulabilir.

Bunun bir sebebi de bellekte karakter pixelleri ve video matrisin ayrı ayrı organize edilmesidir.

Genel olarak bir karakter seti içinde 256 adet karakter bulunur. bu 256 karakterin her biri için bellekte 8 x 8 pixellik (yani 8 baytlık) olmak üzere toplam $800 baytlık yer ayrılır. bu $800 baytın ilk 8 baytı ilk karakterin pixelleridir. İkinci 8 bayt ikinci karakterin pixelleridir vs.

Dolayısıyla VIC bir karakter satırındaki slotlarda hangi karakterler olduğunu bir kere öğrendikten sonra (video matristen okuyarak) bu değerleri 3 bit kaydırıp (yani 8 ile çarpıp) alt 3 bite de 0 dan 7 ye kadar sayan ve her raster satırında bir artan bir iç sayactan gelen değerleri koyarak görüntüyü oluşturan asıl baytları yani karakter setinin içindeki adresleri bulabilir.



Biraz kafanız karışmış olabilir. Sakin olun... Durumu özetleyelim.

Mesela ekranın sol en üstünde A B ve C harfleri var diyelim. Ekrandaki ilk karakter satırının ilk raster satırı default olarak 50 satırdır.

50.satır: VIC en ust satırdaki 40 bayti VM'den okur. Sayacı %000 yapar. Her bir slottaki harflerin ilk baytlarının adresini, sayacı ve slotları kullanarak bulur. Sırayla bu adreslerdeki (yani karakter setine ait bölgedeki) baytları da okur ve görüntü sinyalini üretir.

51.satır: Sayacı %001 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

52.satır: Sayacı %010 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.(bkz sekil)

53.satır: Sayacı %011 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

54.satır: Sayacı %100 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

55.satır: Sayacı %101 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

56.satır: Sayacı %110 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

57.satır: Sayacı %111 yapar. Tekrar slotlarla sayacı tek tek yan yana getirerek ilgili karakter baytlarını okur ve görüntü sinyalini üretir.

Görüldüğü gibi VIC 8 satırda bir ekstradan Video matrise erişmek durumunda. İşte bu satırlar DMA Satırlarıdır.

DMA satırı olmayan satırlarda VIC bellege erişip oradan 40 tane

bayt alır bunlar o an bulunduğumuz karakter satırındaki karakterlerin "karakter seti belleğindeki" 8x1 pixellik sıralardır. Yani mesela 52. raster satırında (ılk karakter sırasının 3. raster satırı) VIC'in gidip 1. karakter satırındaki o an bulunan harflerin her bırının karakter setı belleğindeki 3.baytlarını alması gibi.

Kafanız karışıyor ama buna izin vermeyin... olay aslında anlatıldığından daha basit. Her raster satırında VIC her karakter için bir bayt okuyor. Çünkü karakterin kalınlığı 8 pixel ve bir bayta sığıyor. 40 karakter oldugu için 40 bayt okuyor.

DMA satırlarında ise o satırda karakter setinden okuyacağı 40 bayta ilave olarak 40 bayt da Video Matristen okuyor. İcerideki slotları hazırlamak için

Yani DMA satırlarında VIC diger satırlarda olmadığı şekilde ekstradan 40 bayt için daha belleğe erişecek.

Bir nefes alın...

Bir raster satırı kaç cycle:

Bunu anlamak için ürettiğimiz sinyale bakalım. PAL Video sinyali sabit hızda olduğu için VIC DMA veya değil bütün satırlarda aynı hızla geçmek zorunda. C64 clock cycle'ları da yaklaşık 1 us (mikro saniye). Hesaplar yapıldığında şu ortaya çıkıyor.

```
1 raster sat#r# = 63 cycle
```

Bu hayati önemli bilgiyle sanırım tarihte ilk defa Türkçe bir dokumanda tanışmanızı sağladığımız için çok mutluyuz.

Başka bir deyişle. VIC 63 cycle'da bir alt satıra geçiyor. Hatta isteyen okuyucular, önceki raster efektlerimizde IRQ rutini içindeki döngünün (gecikme tablosundan '8' değeri okunduğu zaman) bir ötelemedeki komutlarının cycle zamanlarının toplamının tam 63 cycle ettiğini hesaplayıp "aaaaaaa" diyebilirler. 6510 komutlarının yediği cycle sayısı için internette bulunabilen "Programcının El kitabı" adlı esere bakılabilir.

VIC ve 6510'un Cycle paylaşımı

Aslında burada ilginç bir nokta var. 6510 ve VIC ikisi de sürekli olarak bellekte değişik adreslere erişiyor. Adres ve data bus ortak olduğuna göre düşünürseniz aslında belleğe bir anda sadece biri erişebilmeli (adres busta sadece birisinin istediği adres yazıyor olabilir)

Ne zaman böyle iki çip aynı belleğe erişmeye çalışırsa aralarında bir çeşit paylaşım mekanizması kullanılır. C64'te de böyle bir sistem mevcuttur. Bu sistemde her cycle 2.ye bölünür. 6510 her cycle'ın ilk yarısında belleğe erişirken VIC her cycle'ın 2. yarısında belleğe erişir. Bu basit fakat dahiyane sistem sayesinde C64 belleğine 63 cycle içinde 126 defa erişilebilir.

6510 sadece ilk yarım cycleları kullanabilir. VIC ise hem kendi yarım cycle'ında hem de 6510'un yarım cycle'ında belleğe erişebilir.

Dayanın... olayın can alıcı noktasına geliyoruz.

VIC karakter ekranı için DMA olmayan bir satırda 40 bayt okur bellekten demistik. işte normalde böyle bir satırda, VIC kendisine ayrılmış olan 63 yarım-cycle'ın 40'ında bu baytları okur. kalan 23'ünde de şu an ilgilenmediğimiz işler yapar. Yani 63 cycle'ı da harcar.

Bu esnada 6510 da kendine ayrılmış olan 63 adet yarım cycle da istediği gibi belleğe erişerek normal şekilde çalışır. 6510 için zaten yarım cycle kavramı yoktur. o yine 1 cycle içinde 1 cylce'lık iş yapar. Ama o cycle içinde eğer belleğe erişmesi gerekiyorsa bunu cycle'ın ilk yarısında yapacak şekilde dizayn edilmiştir.

Gelelim DMA satırına....

İşte bu satırlarda VICin normalde kendisine ayrılan 63 yarım cycle'da yaptığı işlere ilave olaraki Video Matristen 40 bayt daha okuması gerekir. Dolayısyla Kendisine ayrılan 63 adet yarım cycle VIC'e yetmeyecektir.

Bu yüzden VIC normalde 6510a ayrılan 40 adet yarım cycle'ı 6510dan gaspeder. Bunları kullanarak ekstra 40 baytı okuyabilir.

6510 ile VIC arasında, bu gasp için dizayn edilmiş ve daha önce bahsetmediğimiz bir sinyal vardır. Bu sinyal gönderildiğinde 6510 her ne durumda ise donar. tamamen durumunu korur ve hic hareket etmez. Bu sinyal kalktığı anda 6510 ne olduğunun farkında olmaksızın kaldığı yerden devam eder.

İşte VIC bu sinyali kullanarak DMA satırlarında 6510'u tam 40 cycle dondurur. Bunun Sonucu olarak DMA satırlarında 6510 sadece 63 - 40 = 23 cycle harcadığında VIC satır sonuna gelmiştir.

İşte bu yüzden diyebiliriz ki 6510 açısından DMA satırları 23 cycle, diğer satırlar ise 63 cycle'dır. İsteyen okuyucular IRQ rutinlerimizdeki döngüde DMA satırlarında tam 23 cycle harcandığını test edebilir.

Daha ileriki bölümlerde göreceğiz ki bazı başka faktörlerden ötürü kimi durumlarda bu cycle sayıları daha da düşebilir. Ama şimdilik buna takılmayın.

DMA Satırları ile İlgili Geleceğe Bakış

Böylece VIC programcılığındaki en kritik konu olan DMA satırları ile ve 63 sayısı ile sonunda tanıştınız. Bayağı yorulduğunuzu da tahmin ediyoruz. Fakat bu yorgunluğa değeceğini size kanıtlamak için kursun ileri teknikler bölümündeki konuların nasıl bu konulara bağlı olduğundan kısa örnekler vererek bu kursun bu bölümünü bitiriyoruz.

İleri tekniklerin neredeyse tamamı DMA satırları ile ilgili cambazlıklarla gerçekleştirilir. Hemen hepsindeki ortak nokta DMA satırlarının yalnızca sabit olarak 8 satırda bir olması değil istediğimiz zamanlarda olması fikrine dayanmalarıdır.

Çeşitli triklerle istediğimiz satırlarda DMA olup istediklerimizde olmamasını sağlayacağız. Bu sayede İstediğimizde satırları tekrar ettirecek istediğimizde her satırda DMA yaptırıp her satırda yeni bir VM datası okutacağız. FLD, FLI, VSP hepsi buna dayanacak. Öte yandan bu efektlerin hepsinde ve bunların dışında borderları açarken tek tek yazdığınız komutların toplam cycle sürelerini ya 63'e ya 23'e eşitlemeye çalışacaksınız. Aslında bir sürü efektin aynı basit prensiplerden türediğini göreceksiniz.

DMA, 63, 23.... Siz artık bir VIC programcısısınız.

Karakter Setleri

Karakter setlerinden ve video matristen daha önce bahsettik. Ama şimdi onlarla ilgili lazım olan bütün bilgileri tamamlamaya başlayacağız.

Karakter nedir

Karakter, C64'te grafik sisteminin en temel parçalarından biridir. Bir karakter 8 x 8 pixellik bir karedir. Bu 8x8 pixel, 1 veya 0 değerleri alarak içlerinde küçük bir resim parçası taşırlar.

```
........
..****..
.******.
.**..**.
.******.
.******.
.**..**.
.**..**.
```

Örneğin yukarıda görüldüğü gibi bir karakterin 8x8 pixelinin içine A harfini çizebiliriz.

C64'te karakerlerin pixellerinin her biri 1 bit ile ifade edilir. Karakterin yan yana gelen 8 pixellik her bir sırası birer bayt oluşturur. Yani bir karakter bellekte toplam 8 bayt yer kaplar. Yukarıdaki örnekte kullandığımız A karakteri bellekte şöyle gözükür

```
%00000000 = $00
%00111100 = $3c
%01111110 = $7e
%01100110 = $66
%01111110 = $7e
%01111110 = $7e
%01100110 = $66
%01100110 = $66
```

Karakterlerin Karakter Setine Yerleşimi

C64'te bir karakter seti ise tam 256 ($100) karakterden oluşur. Bu karakterlerin her biri bellekte ard arda 8 baytlık alanlar olarak dizilirler. Böylece bir karakter seti bellekte toplam $800 bayt kaplar.

Her bir karakterin bir numarası (indeksi) var diye düşünebiliriz. Bu indeksler 0 ile 255 arasındadır. Karakterlerin baytları ard arda dizildiği için her karaktere ait 8 baytlık alanı bulmak için, o karakterin indeks numarasını sekiz ile çarpıp, bu değere karakter setinin ilk baytının adresini ekleyebiliriz.

Karakterin ilk baytının adresi = (karakterin indexi x 8) + karakter setinin başı

Örneğin, karakter setimiz $2000 adresinden başlayıp $27ff adresine kadarki alanı kaplıyor ise;

- $00 indeksli karakterin baytları $2000 - $2007 arasında,
- $12 indeksli karakterin baytları $2090 - $2097 arasında,
- $81 indeksli karakterin baytları $2408 - $240f arasında

yer alır.

Genellikle karakter setleri text görüntülemek amaçlı kullanılırlar. Bu yüzden bir karakter setindeki 256 karakterin içine çeşitli dillerde büyük ve küçük harfler, rakamlar, noktalama işaretleri veya sembollerin şekilleri çizilir.

Video Matris

C64 ilk açıldığında VIC çipi karakter modundadır. Bu moddayken VIC çipi ekranı 40 karaktere 25 karakterlik bir matris olarak oluşturur. Yani bu moddayken ekran 40 x 25 toplam 1000 karakter slotluk bir alandır. Bu alandaki her bir karakter slotuna karşılık bellekte ardışık olarak birer bayt tutulur. Bellekte bu amaçla ayrılan bu 1000 baytlık bölgeye Video Matris denir.

Ekranda 40x25 lik alandaki herhangi bir karakter slotuna karşılık gelen bellekteki baytın adresini bulmak çok kolaydır. Bellekteki adresler ekranın sol üstünden başlayıp, sağa doğru artarak ilerlerler, satır sonunda bir alt satırın en solundaki karakter slotu ile devam ederler. Genelde hesapları kolaylaştırmak için sütunları sayarken soldan sağa 0 dan 39'a, satırları sayarken de yukarıdan aşağıya 0'dan 24'e kadar sayarız. Bu şekilde sayarak örneğin A satırında B sütununda bulunan karakter slotunun adresi için:

Adres = Video Matris başlangıç + (40 x A) + B

işlemini yaparız. Örneğin, Video Matrisin başlangıç adresi $400 ise;

- 0. satırda 0. karakter slotunun adresi $0400
- 0. satırda 1. karakter slotunun adresi $0401
- 0. satırda 39. karakter slotunun adresi $0427 ($27 = 39)
- 1. satırda 0. karakter slotunun adresi $0428
- 1. satırda 1. karakter slotunun adresi $0429
- 8. satırda 9. karakter slotunun adresi $0549
- 24. satırda 0. karakter slotunun adresi $07c0
- 24. satırda 39. karakter slotunun adresi $07e7

Görüntünün Oluşturulması

Herhangi bir anda VIC'te yalnızca bir karakter seti aktiftir. Ekrandaki 1000 karakterin neler olacagi, Video Matris içerisindeki baytlarda 0'dan 255'e kadar olan karakter indeksi tarafından belirlenir. Başka bir deyişle, Video Matristeki bir adrese, ekranda o adrese karşılık gelen karakter slotunda hangi karakterin gözükmesini istiyorsak, o karakterin indeks numarasını koyarız.

Örneğin diyelim ki, Video Matrisimiz $0400 adresinden başlıyor. Ve biz aktif karakter setinden indeksi 2 olan karakteri ekranda 4. satırın en başına koymak istiyoruz. Bu durumda;

- 4. satır 0. sütundaki slotun adresini bulmalı
- bu adrese 2 değerini koymalıyız

Slotun adresi $0400 + 4 x $28 + 0 = $04a0. Yani bellekte $04a0 adresine 2 yazmamız gerekiyor.

Default Karakter seti ve Video Matris

C64 ilk açıldığında Video Matris, şimdiye kadar örneklerde kullandığımız gibi $0400 - $07e7 adresleri arasında yer alır. Ayrıca C64'te default olarak halı hazırda tanıdığımız o C64 fontları da aktif karakter setidir. Dolayısıyla $0400 - $07e7 arasına değerler yazarak normal C64 fontlarından karakterleri ekranda gösterebiliriz.

Default karakter setindeki hangi indekse karşılık karşılık hangi harfin geldiğini bilirsek kısa bir programla ekrana yazılar yazdırabiliriz. Default karakterlerde harfler alfabetik sırayla $01'den başlayıp (A ile), $1a'ya kadar (Z) dizilmişlerdir. Boşluk karakteri de $20 indexine sahiptir.

Hemen bir örnek verelim.

```
test:
lda #$01
sta $0400
lda #$02
sta $0401
lda #$03
sta $0402
lda #$04
sta $0403
```

Bu kod ekranda sol en üst köşeye ABCD harflerini koyar. Eğer koyacagımız harf sayısı artarsa döngüler kullanırız.

```
ldx #$00
dongu:
lda yaz#,x
sta $0400 + (40 * 10) + 12,x
inx
cpx yaz#_uzunlugu
bne dongu
rts

yaz#_uzunlu#u:
!by $10
yaz#:
!by $00, $01, $02, $03, $04, $05, $06, $07
!by $08, $09, $0a, $0b, $0c, $0d, $0e, $0f
```

Bu kod alfabenin ilk 16 harfini ekranda 10. satırın tam ortasına koyuyor. Bu kodda kullandığımız güzel bir trik var.

```
sta $0400 + (40 * 10) + 12,x
```

Bu satırda assemblerdan yardım alıyoruz. normalde sta adres,x diye kullandığımız komutta adres yerine herhangi bir matematiksel ifade kullanabiliyoruz. Bu sayede yapmamız gereken hesaplamayı, bizim yerimize Acme yaparak bu kodu aşağıdaki koda çeviriyor

```
sta $059c,x
```

Aynı zamanda yazıları bellekte hazırlayıp ekrana kopyalıyoruz. Bunun için bu örnekte !by komutunu kullandık. Bunun yerine Acme'nin bize sağladığı !scr komutunu da kullanabiliriz. Bu komut özellikle bellekte bir bölgeye bir grup harfin default karakter setindeki indexlerini yerleştirmeye yarar. Default karakter setinde bir harfe karşılık gelen indexe o harfin "Ekran Kodu" denir. Acmedeki !scr komutu ("Screen Code"), ardından gelen tırnak içindeki yazıda geçen bütün harflerin ekran kodlarını bellekte sıralar

```
yaz#_uzunlu#u:
!by $15
yaz#:
!scr "yeni bir gun ba#l#yor"
```

Yukarıdaki değişikliği yaparak yazıyı değiştirebilirsiniz. Dikkat edeceğiniz nokta yazı_uzunluğu'nu göstermek istediğiniz harf sayısına getirmek.

Default karakter setindeki harflerin indexleri ile video matrise yazı yazmak ile ilgili pekçok örneği 6510 Assembly kursunda bulabilirsiniz. Bizim asıl uğraşmak istediğimiz konu default karakter setini değil kendi karakter setlerimizi kullanmak

Kendi Karakter Setimizi Kullanmak

Neyse ki Commodore'un iş gören ama sıkıcı fontlarından başka dünyada pekçok grafikerin çizdiği binlerce font var. Ayrıca kendiniz de hazırladığınız bir programın temasına tam uyan bir karakter setini tasarlamak isteyebilirsiniz.

İşte bu durumlarda eninde sonunda elinizde $800 baytlık bir karakter seti dosyası ile baş başa kalacaksınız. Bu dosyayı bir karakter set tasarlama aracı kullanarak yaratabileceğiniz gibi, Web den veya bir arkadaşınızdan hazır olarak da alabilirsiniz. Bu kurstaki çalışmalarınızı kolaylaştırmak için biz de size bazı hazır karakter seti dosyaları sunuyoruz.

Şimdi asıl sormamız gereken soru şu. Elimizdeki bu dosyayı ne yapacağız.

Bir karakter setini kullanabilmek için yapmamız gerekenler şunlar:

- karakter setini belleğe yüklemek

- VIC'e karakter setinin bellekte nerede olduğunu söylemek
- Video Matrise yeni karakter setimizdeki karakterlerin indekslerini kullanarak yazı yazmak.

Karakter Setini Belleğe Yüklemek

VIC tasarlanırken donanımı kolaylaştırmak için karakter setlerinin bellekte yüklenebileceği yerlerin $800'ün katları olması kararlaştırılmış. Dolayısıyla biz karakter setimizi $2000, $2800, $3000 gibi adreslere yükleyebiliyoruz.

Genelde bu kursta biraz ilerleyene kadar karakter setlerimizi hep 4 adresten birine koyacağız: $2000, $2800, $3000, $3800. Bu adresleri neden kullandığımızı ileride öğreneceksiniz.

VIC'e Karakter Setinin Yerini Bildirmek

Bu amaçla Video Matris'te görevli bir register var. Kendisi beraber bu alemde en çok takılacağımız kanka registerlerden biri. Size kendisini tanıtmaktan çok mutluyuz. İşte karşınızdaaaa....

VIC Bellek Kontrol Registeri: $D018

Bu güzide registerimizin iki görevi vardır:

- Video Matris olarak bellekte neresi kullanılıyor
- Karakter seti bellekte nerede duruyor.

Kursun ilk başında 6510'un ara sıra VIC'e ekranı oluşturmak için bellekte nereye bakması gerektiğini söylediğini anlatmıştık. İşte bu işlem VIC'in $d018 registerinin programlanması ile yapılır. Bu registerda bulunan bilgiyi kullanarak VIC ekrandaki herhangi bir karakter slotuna hangi karakterin geldiğini, ve o karakterin 8x8 pixelinin hangilerinin 1 hangilerinin 0 olduğunu bilebilir

Bunun için $d018'in bitleri iki gruba ayrılmıştır. Üst 4 bit ile Video Matrisin yeri programlanırken, alt 4 bit ile de karakter setinin yeri programlanır.

Karakter setinin yerini belirten 4 bitten sadece üst 3'ü VIC tarafından dikkate alınır. Dolayısıyla normalde Karakter Seti için aşağıdaki değerler aracılığıyla VIC'e 8 farklı yer gösterilebilir.

```
$d018'in alt 4 biti:        Karakter Seti adresi:
--------------------        ---------------------
%0000 = $0                    $0000
%0010 = $2                    $0800
%0100 = $4                    $1000
%0110 = $6                    $1800
%1000 = $8                    $2000
%1010 = $a                    $2800
%1100 = $c                    $3000
%1110 = $e                    $3800
```

Daha önce belirttiğimiz gibi biz bir süre sadece son 4 olasılığı kullanacağız.

Nasıl donanımı basitleştirmek için Karakter Setleri ancak $800'ün katı olan adreslere konabiliyorsa, aynı şekilde VIC tasarımcıları Video Matrisin de konulabileceği adreslerin $0400'ün katı olması gerekliliğini getirmiş. Dolayısıyla $d018'in üst 4 biti kullanılarak VICin Video Matris için bellekte 16 değişik yere bakması sağlanabiliyor:

```
$d018'in üst 4 biti:        Video Matris adresi:
--------------------        --------------------
%0000 = $0                    $0000
%0001 = $1                    $0400
%0010 = $2                    $0800
%0011 = $3                    $0c00
%0100 = $4                    $1000
%0101 = $5                    $1400
%0110 = $6                    $1800
%0111 = $7                    $1c00
%1000 = $8                    $2000
%1001 = $9                    $2400
%1010 = $a                    $2800
%1011 = $b                    $2c00
%1100 = $c                    $3000
%1101 = $d                    $3400
%1110 = $e                    $3800
%1111 = $f                    $3c00
```

Görüldüğü gibi $d018'e yazacağımız değer ile, VIC'e bellekten alması gereken iki önemli bilgiyi (yani ekrandaki karakterlerin indekslerini ve o karakterlerin biçimlerini) bellekte nereden alacağını söylemiş oluyoruz.

Örnek: Eğer Video Matris olarak [$0400 , $07e7] arasını kullanmak istiyorsak ve karakter setimizi de $2800 adresine yüklemiş isek.

```
Video matris   $0400 => üst 4 bit = $1
Karakter seti  $2800 => alt 4 bit = $a
----------------------------------------
$d018'e yaz#lacak de#er = $1a
```

ve birkaç başka örnek daha:

```
Video matris   $0c00 => üst 4 bit = $3
Karakter seti  $2000 => alt 4 bit = $8
----------------------------------------
$d018'e yaz#lacak de#er = $38
```

```
Video matris   $3400 => üst 4 bit = $d
Karakter seti  $3800 => alt 4 bit = $e
----------------------------------------
$d018'e yaz#lacak de#er = $de
```

Aynı Ekranda Birden Fazla Karakter Seti Kullanmak

Artık istediğimiz karakter setini kullanmak için yapmamız ge-

reken şeyleri biliyoruz. Daha sonra göreceksiniz ki aslında pekçok efekti yaparken ekranda birden fazla karakter seti gösterebilmemiz gerekecek. Bunu nasıl yapabiliriz...

Aslında cevap basit. Interrupt'ları kullanacağız.

Bunun sayesinde VIC henüz ekranın yukarılarını çizerken bir karakter setini seçecek, ardından ekranın alt bölümlerini çizerken ise ikinci bir karakter setini sececeğiz. Bütün yapmamız gereken belli satırlarda IRQ oluşmasını sağlayıp, ardından o IRQ rutinlerinde $d018'e doğru değerleri yazmak.

Bunun nasıl yapıldığına dair bir örnek görmek için ornekler/src/karakter_seti.a64 dosyasına bakabilirsiniz. (henüz bu dosya hazır değil)

Şimdi bambaşka bir konuya geçiyoruz. Bu sayede karakter setleri ile yapabileceğiniz ilk efektlere başlıyoruz.

Yumuşak Kayan Yazılar

Raster çizgileri ile beraber C64'teki belki de en klasik intro efekti ünvanını hak eden "kayan yazılarla" tanışmak üzeresiniz.

Kayan yazılar (scroll olarak da bilinir) nedir. Genellikle bir satır yazının ekranın her cizilişinde bir piksel yana kayması ile elde edilen bir efektir. Böylece yazı çok yumuşak bir şekilde ekrana sağdan girip, yavaş yavaş sola doğru ilerler. Tıpkı ekonomi kanallarında ekranın en altından sürekli olarak akan hisse senedi fiyatları gibi.

Normalde video matristeki karakter slotlarına koyduğumuz karakterleri belli aralıklarla bir slot yana kaydırarak harflerin slotların icinden gecmesini sağlayabiliriz. Bu şekilde yapılan bir kayan yazı örneği 6510 Assembly kursunda var.

Fakat bu şekilde yaptığımız scrollda malesef karakterler her kaymalarında 8 piksel birden atlayarak kayarlar. Çünkü karakter slotlarının yeri sabittir ve ekranda 8 pixel kalınlığında bir alan kaplarlar.

Neyseki daha yumuşak kaymaları gerçekleştirebilmemizi sağlayan bir register var. $d016

VIC Kontrol Registeri $d016

Bu register VIC içindeki en önemli registerlerden biridir. Pekçok görevi vardır. Biz şimdilik kayan yazılar ile ilgili bölümlerinden bahsedeceğiz.

Bu registerin en alt üç biti kullanılarak ekrana yatay eksende birer piksellik kaymalar yaptırılabilir. Normalde bu üç bitin değeri 0'dır. Böyleyken ekran tam orta konumdadır ve butun karakter slotları tam olarak gorunebilir.

Bu üç piksele 1 (yani binary %001) yazıldığında ise ekrandaki karakter slotlarının hepsi bir pixel sağa kayarlar. Bunun sonucu olarak, ekranın en sağındaki karakterlerin 8x8 piksellerinden en sağdaki piksel sütunu çerçeve bölgesine girer ve çerçevenin arkasında kalarak görünmez olur. Başka bir deyişle, en sağ karakter slotlarında yer alan karakterleri 7x8 piksel olarak görürüz.

Bu üç piksele 2 yazıldığında ekran normal konumuna göre iki piksel sağa kayar ve biz en sağdaki karakterleri 6x8 piksel olarak görürüz.

Bu şekilde 7'ye kadar değerler yazarak, bütün ekranın 0 ila 7 piksel arasında uzaklıklarda sağa kaymasını sağlayabiliriz. Hemen örnekler arasındaki kayan_yazi0.a64 programını inceleyip çalıştırın. Olayın aslında kelimelerle anlatılmaya çalışıldığında olduğundan çok daha basit olduğunu görün.

Ama tabii biz bir yazıyı yalnızca 8 pixel kaydırmak istemiyoruz. Bütün ekran boyunca kaydırmak istiyoruz. Bunu yapabilmek için $d016'ya değerler yazmak dışında yapmamız gerekenler var.

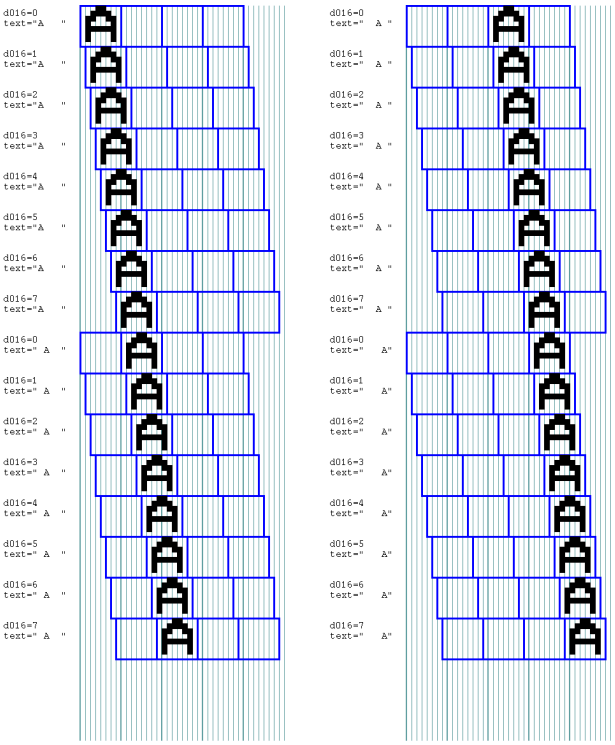
Yapmamız gereken şey önce ekranı piksel piksel 7 kere kaydırmak. Ardından 8. kere kaydırmak için iki işlem yapacagız.

Video Matristeki karakter kaydırma işlemlerinin 8 piksellik atlamalar yaptığını söylemiştik. İşte bundan faydalanarak, bütün ekranı 7 kez birer piksel kaydırdıktan sonra $d016'nın ilk 3 bitini tekrar 0 yapıp aynı zamanda karakterleri bir yandaki karakter slotuna cizeceğiz.

Bu olay bir döngü halinde devam edecek. Henüz kafanızda tam oturmadıysa da endişelenmeyin birazdan çok basit bir örnek üzerinde anlatacağız.

Problemi biraz basitleştirelim. Ekranda tek bir harfi toplam 32 piksel (yani 4 karakter slotu) kaydıracağız. Bu örnekle beraber herşeyin temelini anlayacaksınız.

Yukarıda anlatmış olduğumuz adımları izlediğinizde yazının neden kaydığını daha iyi anlamak istiyorsanız, Şekil-6.1'e bakabilirsiniz.



Şekil 1. Harflerin Piksel Piksel Kayışı

Örnek Kod

Şimdi bu işlemi kodda nasıl yapacağımızı anlatalım.

Örnekler arasındaki kayan yazı programını inceleyelim

```
;; ------------------------------------------------
IRQRutini:
inc $d019
```

```
jsr yaziyi_ekrana_koy
jsr yazinin_konumunu_degistir

jmp $ea81
```

Yine IRQ rutini üzerinde yoğunlaşacağız. Bu rutinde yaptığımız iki ana işlem var. Programımızı bu iki ana işlemi birer alt rutine organize ederek başlatıyoruz. yazıyı ekrana çizmek ve bir sonraki çizim için konumu değiştirmek.

```
;; ------------------------------------------------
ekran = $0410
yaziyi_ekrana_koy:
lda yazi_konumu
and #$07         ; %00000111
sta $d016
```

Bellekte bir baytı yazı konumunu saklamak için kullanıyoruz. Bu baytta yazı konumu olarak $00 ile $1f arası değişen değerler saklayacağız. İlk yaptığımız iş yazı konumunun alt 3 bitini kullanarak d016'daki pixel hassasiyetli kaymayı ayarlamak.

```
lda yazi_konumu
and #$f8         ; %11111000

cmp #$00
beq konum_0_7

cmp #$08
beq konum_8_15

cmp #$10
beq konum_16_23

cmp #$18
beq konum_24_31
```

Şekil 2'de dikkat ederseniz, bu 32 konumda aslında 4 karakter slotluk bölgeye 4 yazıdan birini koyuyoruz.

```
Yaz# konumu 0 ile 7 aras#ndayken   "A   "
Yaz# konumu 8 ile 15 aras#ndayken  " A  "
Yaz# konumu 16 ile 23 aras#ndayken "  A "
Yaz# konumu 24 ile 31 aras#ndayken "   A"
```

Programın bu bölümünde yaptığımız şey yazı_konumunun o anki değerine bakarak, video matrise hangi yazıyı yerleştireceğimizi bulmak.

```
;; ------------------
konum_0_7:
ldx #$0

k07_dongu:
lda yazi0,x
sta ekran,x
inx
cpx #$4
bne k07_dongu
rts

;; ------------------
konum_8_15:
ldx #$0

k815_dongu:
lda yazi1,x
sta ekran,x
inx
cpx #$4
bne k815_dongu
rts

;; ------------------
konum_16_23:
ldx #$0

k1623_dongu:
lda yazi2,x
sta ekran,x
inx
cpx #$4
bne k1623_dongu
rts

konum_24_31:
ldx #$0

;; ------------------
k2431_dongu:
lda yazi3,x
sta ekran,x
inx
cpx #$4
bne k2431_dongu
rts
;; ------------------------------------------------
yazi0: !scr "a   "
yazi1: !scr " a  "
yazi2: !scr "  a "
yazi3: !scr "   a"
```

Burada ise 4 kere aynı rutin tekrarlandı. Tabii normalde böyle bir tekrar gördüğünüz zaman hemen "Böyle birşeye ne gerek var" demelisiniz. Aslında yok ama olan biteni açık seçik görebilmeniz için bu örnek bu şekilde hazırlandı. 4 rutinde yapılan tek şey doğru 4 karakteri ekrana kopyalamak.

```
;; ------------------------------------------------
yazinin_konumunu_degistir:
lda yazi_konumu
clc
adc #$01
and #$1f
sta yazi_konumu
rts
;; ------------------------------------------------
yazi_konumu:
!by 00
;; ------------------------------------------------
```

Son olarak burada yazı_konumunun sürekli 0 ile $1f arasında saymasını sağlayan kodu görüyorsunuz.

Bu programı çalıştırdığınızda ekranda eğer başka yazılar da olursa onların başına ne geldiğini de göreceksiniz. İşte o yazılar $d016ya yazdığımız değerlerden ötürü 8 piksel içinde kendilerince kayıp dururken en üst satırdaki A harfimizin nasıl mağrur bir şekilde 32 piksel kaydığını görebilirsiniz. İşte bu ekstra mesafeyi A harfinin video matristeki slotunu değiştirerek kazandık.

Eğer burada olan biteni anladı iseniz, bazı ufak değişiklikler içeren kayan_yazı1.a64 orneğine bakalım.

```
ekran = $0402
yaziyi_ekrana_koy:
lda yazi_konumu
and #$07         ; %00000111
sta $d016

lda yazi_konumu
lsr
lsr
lsr
tay
ldx #$0
konumla_dongu:
lda yazi,x
sta ekran,y
iny
inx
cpx #$1a
bne konumla_dongu
rts

;; -----------------------------------------------
yazi: !scr "        kayan yazi        "
```

Aslında burada yazı_konumuna daha geniş bir salınım yaptırmak istiyoruz. Bu yüzden geçen seferki gibi 4 ayrı yazı yerine aynı yazıyı ekranda değişik konumlara kopyalamayı seçeceğiz.

Bu amaçla yazı konumuna bakarak yazıyı başlangıç noktasından ne kadar öteye koyacağımızı bulmalıyız. Bunun için yazı konumunu 8'e bölüyoruz. Çünkü yazı konumu her 8 artışında bizim yazıyı bir karakter slotu ileriye kopyalamamız gerekiyor.

6510 Assembly ile 8'e bölmenin en kolay yolu sayının bitlerini 3 kere sağa kaydırmak. Elde ettiğimiz değeri y registerine saklıyoruz. ardından kurduğumuz döngüde x registerini kopyalayacağımız harf sayısını takip eden sayaç olarak kullanırken y registerini de sta ekran,y komutu ile harfleri ekrana göndermek için kullanıyoruz.

Sürekli aynı yazıyı ekranda farklı yerlere kopyalarken dikkat edeceğimiz bir nokta var. Ekranı hiç silmediğimiz için aslında kopyaladığımız karakterlerden bazıları ekranda kalıyor (bir nevi iz bırakıyor) bu izi kalan karakterler bizi rahatsız etmesin diye yazdığımız mesajın önüne ve arkasına boşluk karakterleri yerlestirdik. Eğer bu boşluklar olmazsa ne olacağını görmek için boşlukları kaldırıp deneyin

```
yazinin_konumunu_degistir:
ldx yazi_konumu_okuyucu
lda konum_animasyon_tablosu,x
sta yazi_konumu
inc yazi_konumu_okuyucu
rts
;; -----------------------------------------------
yazi_konumu:
!by 00
yazi_konumu_okuyucu:
!by 00

!align 255,0
konum_animasyon_tablosu:
!src "ortak/sin_table_01.a64"
```

Ayrıca yazı konumunu daha estetik bir salınım ile değiştirmek için sinüs tablosundan okuyoruz.

Haydi iki küçük ekleme daha yapalım. Hemen kayan_yazı2.prg'ye bakın. İlk değişiklik ekranı hazırlayan bir rutin eklemek

```
;; -----------------------------------------------
EkraniHazirla:
lda #$18
sta $d018

ldx #$00
lda #$20
EH_dongu:
sta $0400,x
sta $0500,x
sta $0600,x
sta $0700,x
inx
bne EH_dongu
```

Bu örnekte karakter setini de değiştiriyoruz. Bunun için programın başında $d018 i programlıyoruz. $18 değeri Video Matrisi $0400 adresine yerleştirirken, karakter setini de $2000 adresinden okuyacağımızı VIC'e bildiriyor.

Ardından kurduğumuz basit döngüyle Video Matrisi boşluk karakteri ile dolduruyoruz.

```
ldx #0
EH_dongu2:
lda SabitMesaj,x
sta $0400 + (4 * 40) + 14,x
inx
cpx #13
bne EH_dongu2

rts
SabitMesaj:
!scr "bu yazi sabit"
```

Bu satirlarla da ekranda 4. satırın ortasına sabit mesajı yazıyoruz.

Eğer yalnızca bu iki değişikliği yaparsanızi, sabit olması gereken yazının hala ortada fıldır fıldır sallandığını göreceksiniz. Nedeni basit. $d016'ya yazdığımız değerler bütün ekranı sallıyor.

VIC'i artık tanıyorsunuz. Ve evet çözüm de tahmin ettiğiniz gibi interrupt rutininde

```
IRQRutini:
inc $d019

jsr yaziyi_ekrana_koy
jsr yazinin_konumunu_degistir

EfektinSonSatiriniBekle:
lda $d012
cmp #$3c
bmi EfektinSonSatiriniBekle

lda #$0
sta $d016

jmp $ea81
```

Yazının konumunu değiştirdikten sonra raster'ın $3A satırına

gelmesini bekliyoruz. Bu olduktan sonra da $d016yı 0 ile sabitliyoruz. Böylece ekranın bundan sonraki bölümü sallanmayacak. Ta ki bir sonraki sayfa çizilirken yeniden $20. satırda interrupt oluşana kadar.

Son olarak aşağıdaki komutla karakter seti dosyasını bellekte $2000 adresine yüklüyoruz.

```
*=$2000
!bin "../ext/c1x1.fnt",,
```

Ayrıca kayan_yazi2 programını çalıştırmak için

```
sys 3072
```

kullanmanız gerekiyor. cunku önceki örneklerden farklı olarak bu örnek $0c00 (3072) adresine yerleştirildi. Kursu okumadan direk programları çalıştıranlar cezasını bulacak böylece :)

Ekran Boyunca Kayan Yazı

Geçen bölümde karakter setlerini değiştirmekten ve kayan yazılardan bahsetmiştik. Fakat kayan yazı örneğimiz bir cümlenin sağa sola kayması şeklinde idi. Bu bölümde klasik anlamda uzun bir mesaj içeren ve ekrana sağdan girip sürekli sola devam eden kayan yazı örneği ile başlayacağız.

Aslında bu örnek birazdan göreceğiniz gibi kayan_yazı2 örneğinden daha basit. Geçen bölümün sonunda bu işi nasıl yapacağımızdan biraz bahsetmiştik. Hatırlarsanız, ana algoritmamız şöyleydi:

- d016 değerini bir azaltarak bir piksellik kayma yap
- eğer d016 0 olmuş ise karakter slotlarında kayma yap

Aslında bu algoritmayı biraz detaylandırmamız gerekiyor. Düşünürseniz d016'ya 0 değerini yazmamızdan hemen sonra eğer karakter slotlarında kaydırma yaparsak bir problemimiz var. Problemi anlamak için şu senaryoyu düşünelim.

Diyelim ki kayan satır en üst satır (yani $0400 - $0408). Diyelim ki yazı kaymış kaymış ve yazımızın ilk harfi tam 0401 adresindeki karakter slotuna gelmiş. Şimdi tam 0401 adresinde olan karakteri düşünün. Yani soldan 2. karakter. Ve diyelim ki d016'ya geçen sayfada 1 yazılmıştı. IRQ'muzun olacağı raster satırını ilk karakter satırının başına yakın ve ondan önceki bir raster satırına ayarlayacağız. Böylece d016'ya yapacağımız bir yazma işlemi kayan satırın çizilmesini etkileyecek.

Şimdi IRQ rutinine girdik. IRQ'ya girer girmez piksel kayması yapmak üzere d016'ya 0 yazdık. Normalde istediğimiz şey bu sayfa çizilirken 0401'deki karakterin (yani mesajımızın ilk harfinin) yine 0401'de kalması ve bu satırın d016'da 0 değeri ile çizilmesi. Geçen sefer 1 ile çizilip bu sefer 0 ile çizilmesi halinde geçen sayfadan bir piksel sola kaymış olacak yani ve zaten istediğimiz de bu.

Fakat biz IRQ rutininde d016'ya 0 yazdıktan hemen sonra d016'ya yazdığımız değerin 0 olduğunu görüp karakter slotlarını kaydırmaya başlarsak problemimiz var. Çünkü düşünürseniz henüz raster, satırımızın başında veya belki de öncesinde. Daha IRQ rutinine yeni girdik ve fazla bir zaman harcamadık (sadece d016'daki değeri manipule ettik). Dolayısıyla, eğer şu anda karakterleri slotların içinde kaydırırsak, ekranın bu çiziminde $0401'deki mesajımızın ilk harfini $0400'a kaydırmış olacağız. Yani bu ekranda $0401 de görünmesi gereken harf olması gerekenden 8 piksel solda görünecek. Bir sonraki ekranda d016'ya yeniden 7 yazdığımızda da 7 piksel sağa kayacak. yani bü ilk harfin çerçeveden piksel olarak uzaklığını düşünürsek ekranın 3 çiziminde olması gereken ve olan durumlar şöyle olacak:

olması gereken sıra: 9, 8, 7

olan sıra: 9, 0, 7

Bunun olmaması için basit bir önlem almamız gerekiyor. Karakter slotlarındaki kaydırmayı yapmak için ekranın kayan bölümünün raster satırlarının geçmesini beklemek. Yani algoritmamızı şöyle değiştiriyoruz:

- d016 değerini bir azaltarak bir piksellik kayma yap
- ekranda kayan bölümün sonuna gelmeyi bekle
- eğer d016 0 olmuş ise karakter slotlarında kayma yap

Şimdi son adımı biraz daha detaylı düşünelim. Yapmamız gereken işlem kayan satırdaki bütün karakterleri bir slot yana kaydırmak. Bu işlem hakkında düşünürken, satırın ilk ve son karakter slotlarında ne işlemler yapılması gerektiğine dikkat edelim. En soldaki slota (yani 0. slota) bir sağındaki slottaki (yani 1. slottaki) karakter kopyalanacak. Çünkü bütün karakterler bir slot sola kayıyor. Peki en sağdaki slot? Bu slottaki (yani 39.slottaki) karakter bir soldaki slota (yani 38. slota) kopyalanacak. Peki 39. slota sonra ne yapmalıyız. Düşünürseniz kayan mesajımızdaki bir sonraki harfi (yani ekrana sağdan yeni giren harfi) 39.slota koymamız gerekiyor. Böylece ekrandaki bütün harfler bir slot sola kaydırılıp, sıradaki harf için en sağdaki slotta yer açılıyor.

Bunun dışında 1'den 39'a kadarki 39 slottaki karakterin 0'dan 38'e kadarki slotlara kopyalanması işlemini nasıl yapmalıyız? Elbette bir döngüyle yapacağız. Fakat döngümüz harfleri hangi sırayla kopyalayacak?

İlk aklınıza gelen cevap once en sağdan yani 39. slottan başlayarak her slottaki harfi bir sola kaydırmak olabilir. Fakat biraz dikkatli düşünürseniz bu yaklaşımda problem olduğunu görürsünüz. Eğer döngümüzde;

- 39'dan 38'e kopyala
- 38'den 37'ye kopyala
- 37'den 36'ya kopyala

...

diye ilerlersek aslında yaptığımız şey 39. slottaki karakter neyse

bütün satırı o karakter ile doldurmak olacaktır. Çünkü birinci adımdan sonra 39 ve 38'de aynı karakter olacak. İkinci adımdan sonra 39 ve 38'de olan aynı değeri 37'ye kopyalamış olacağız. Üçüncü adımda 39, 38, ve 37'deki değeri 36'ya kopyalayacağız. Problemi görüyorsunuz.

Bunun yerine yapmamız gereken şey kopyalamaya diğer uçtan başlamak. Yani önce 1. slottan 0. slota, ardından 2. slottan 1. slota, ardından 3. slottan 2. slota şeklinde ilerlemeliyiz. Bu şekilde tam istediğimiz şeyi yapmış olacağız.

Şimdi bu noktaların ışığında algoritmamıza biraz daha detay ekleyelim:

- d016 değerini bir azaltarak bir piksellik kayma yap
- ekranda kayan bölümün sonuna gelmeyi bekle
- eğer d016 0 olmuş ise karakter slotlarında kayma yap
- 1'den 39'a kadarki toplam 39 slottaki karakterleri 0'dan 38'e kadarki slotlara kopyala
- 39. slota kayan mesajın bir sonraki karakterini yerleştir.

Artık kodu incelemeye hazırız. kayan_yazı2 örneğini IRQRutini bölümünden itibaren değiştiriyoruz. Tam kaynak kodu kayan_yazı3.a64 dosyasında bulacaksınız.

```
IRQRutini:
inc $d019

jsr bir_pixel_kayma

KayanBolgeninSonunuBekle:
lda $d012
cmp #$3c
bmi KayanBolgeninSonunuBekle

lda #$0
sta $d016

jsr karakter_slotlarinda_kayma
jmp $ea8
```

Burada algoritmanın ana iskeletini IRQ rutinine yerleştirilmiş halde görüyorsunuz. Ayrıca algoritmada belirtmediğimiz bir detay olan kayan bölgenin dışında $d016'nın 0 değerine sabitlenmesini de burada görüyorsunuz.

```
;; ----------------------------------------
kayan_satir = $0400 + (0 * 40)

bir_pixel_kayma:
lda bizim_d016
sec
sbc #$01
and #$07         ; %00000111
sta bizim_d016
sta $d016
rts
```

İşte burada bir piksellik kaymaları yapan kodumuzu görüyorsunuz. Buradaki bizim_d016 adresini (veya *değişkenini* de diyebiliriz) niçin kullandığımızı açıklayalım. Burada d016'daki değeri her defasında 0 ile 7 değerleri arasında bir azaltmak istiyoruz. Ve daha sonra 0 olduğu zaman ileriki bölümlerde slot kaydırmasını yapmamız gerekecek. Bu durumda ilk aklınıza gelen çözüm şöyle olabilir.

```
lda $d016
sec
sbc #$01
and #$07
sta $d016
rts
```

Fakat IRQ rutininin iskeletine bakarsanız, orada tam karakter slotlarında kayma zamanı gelmiş mi diye kontrol etmeden önce $d016'yı sıfırlıyoruz (ekranın geri kalanı kaymasın diye). Bu yüzden o rutinde d016 0 olmuş mu diye bakarsak her zaman 0 göreceğiz. Bu problemi aşmak için şöyle bir yol takip ediyoruz. d016 registerindeki değeri kontrol için kullanmak yerine kayan bölgede d016'ya yazdığımız değerin bir kopyasını da bizim_d016'ya yazıyoruz. Bir nevi d016'nın bir kopyasını tutuyoruz elimizde (sadece kayan bölgede geçerli olan değeri). Bu sayede ekranın başka bölümlerinde registere başka değerler yazsak bile kayan yazı rutini kendi en son yazdığı değeri hatırlıyor ve kendi sırası geldiğinde o değeri kullanarak işlemlerine devam edebiliyor.

```
karakter_slotlarinda_kayma:
lda bizim_d016
beq slotlari_kaydirma_vakti
rts
```

Demin açıkladığımız gibi bu bölüm bizim_d016'daki değerin sıfır olup olmadığını kontrol ederek slotları kaydırma zamanı gelip gelmediğine karar veriyor. Hatırlarsanız bu kaydırma sadece 8 ekranda bir oluyor. Yani d016 7'den 1'e kadar değerleri alırken bu işlem yapılmayıp sadece 0 değerini aldıktan sonra yapılıyor.

```
slotlari_kaydirma_vakti:
ldx #0
skv_dongu:
lda kayan_satir+1,x
sta kayan_satir,x
inx
cpx #39
bne skv_dongu
```

Ve burada da 39 karakteri bir sol slota kaydıran döngüyü görüyorsunuz. Bu döngüyü algoritma esnasında detaylı şekilde açıkladık.

```
ldx sonraki_karakter
lda yazi,x
sta kayan_satir+39

inx
stx sonraki_karakter
rts
```

Burada da son olarak satırın en sağındaki slota sıradaki karak-

terin konuşunu görüyorsunuz. Mesajın hangi karakterini koymamız gerektiğini takip etmek için sonraki karakter diye bir indeks değeri tutuyoruz. Bu değer bir bayt olduğu için 0 ile 255 arası değerler alabilir. Dolayısıyla bu rutinde kullandığımız mesaj en fazla 256 karakter olabilir.

Bu konuyu kapatmadan önce bir konuya daha değineceğiz. d016 registerinin kayma konusundan sorumlu bölümünün ilk 3 biti (0, 1, ve 2 nolu bitler) olduğunu söylemiştik. Bu konuyla alakalı bir bit daha var. 3 nolu bit. Bu bit ekranın görünen bölgesindeki genişliğini kontrol eder. Normalde bitin değeri birdir. Böyleyken ekran 40 karakter genişliğindedir. Eğer bu biti 0 yaparsanız, ekranın etrafındaki çerçeve biraz büyür. En soldaki karakter slotu çerçevenin arkasında kalır ve ekran 39 kararkter slotu genişliğinde görülür. Bizim örneğimizde de d016'ya hep 0 - 7 arası değerler yazdığımıza göre dikkat ederseniz bu biti 0 yaparak ekran kalınlığını 39 karaktere daraltıyoruz.

Peki bunu neden yapıyoruz. Bu sayede ekran sola kayarken en soldaki kararkter yok olmadan önce ekrandan çıkıyor. Yani örneğimizde, 0400 slotuna gelen bir karakteri d016'da 7 değeri varken düşünün. Bu karakterin 8x8 pikselinden en sol sütunu çerçeve arkasına giriyor. ardından birer piksellil kaymalarla d016 en son 0 değerini aldığında 0400'daki kararkterimiz tamamen çerçevanin arkasında kalmış oluyor. Biz de tam bu esnada 0401deki kararkteri onun üzerine kopyalıyoruz ve ertesi ekran çiziminde d016'yı tekrar 7 yapıyoruz. Bu sayede harfler yumuşak bir şekilde ekranın solundan dışarı çıkıyor. Bunu yapmasak ne olacağını görmek için bir piksel kayma rutinini şöyle değiştirin

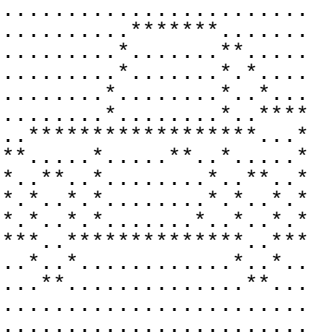
```
bir_pixel_kayma:
lda bizim_d016
sec
sbc #$01
and #$07         ; %00000111
sta bizim_d016
ora #$08                ; 3 nolu biti 1 yap
sta $d016
rts
```

Karakter Grafikleri

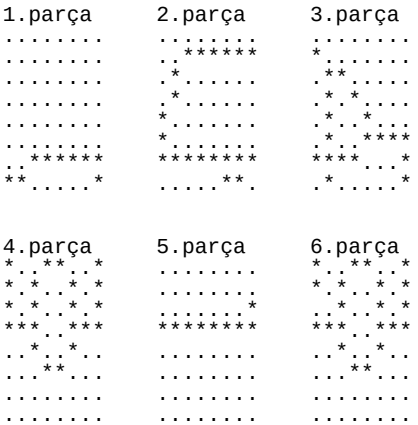
Karakterleri Birleştirmek

Geçen bölümde karakter setini nasıl değiştirebileceğimizi anlatmıştık. Hatırlarsanız, karakter setimiz her karakter için 8 bayt saklıyordu. Bu 8 baytın içine harfleri çiziyorduk.

Peki bu 8 baytın içine harf çizmek zorunda mıyız? Elbette hayır. Şimdi daha büyük bir resmi düşünelim. Diyelim ki elimizde 24 x 16 piksellik bir şöyle bir araba resmi var



Böyle bir resmi karakterleri kullanarak çizebilir miyiz? 24x16 piksel bir karakterin içine sığmıyor. Ama biz bu resmi 8 x 8 piksellik karelere bölsek ve ardından elde ettiğimiz bölümlerin her birini bir karaktere yerleştirsek karakter setinde, resmin 6 parçasını birer karakter olarak elde etmiş olacağız. Sonra bütün yapmamız gereken bu altı karakterin indekslerini video matris içinde birbirine göre doğru konumlarda yan yana getirmek.



İşte resmi 6 adet 8x8 piksellik parçaya böldük. Bu 6 parçayı karakter setindeki istediğimiz 6 indekse yerleştirebiliriz. Genelde bir karaker setindeki ilk 64 indeksin içinde harfler ve rakamlar olur. Pekçok karakter setinin sonraki indeksleri kullanılmaz. Bu yüzden bu 6 parçayı diyelim ki $40 - $45 arasındaki 6 indekse yerleştirelim. Bu ne demek? Yani bu parçaları oluşturan 8'er baytlık bayt gruplarını, bellekte, karakrter setinin içinde doğru adreslere oturtalım. Karakter setimizin $2000 adresinde olduğunu varsayarsak;

```
1.parçay# $2000 + ($40 * 8) = $2200 - $2207 aras#na,
2.parçay# $2000 + ($41 * 8) = $2208 - $220f aras#na,
3.parçay# $2000 + ($42 * 8) = $2210 - $2217 aras#na,
4.parçay# $2000 + ($43 * 8) = $2218 - $221f aras#na,
5.parçay# $2000 + ($44 * 8) = $2220 - $2227 aras#na,
6.parçay# $2000 + ($45 * 8) = $2228 - $222f aras#na,
```

yerleştireceğiz.

Artık kararkter setimizdeki $40 - $45 arası 6 karakter araba resmimizin 6 parçasını taşıyor. Dolayısıyla arabayı ekranda görmek için artık yapmamız gereken tek şey, bu 6 karakeri video matrise koymak. Örneğin arabayı ekranın en üst sol köşesine koymak istersek (ve video matris eğer $0400 adresindeyse);

```
en üst sat#r#n ilk üç karakter slotuna s#ras#yla $40, $41
bir alt#ndaki sat#r#n  ilk üç slotuna s#ras#yla $43, $44,
```

değerlerini koymamız yeterli olacak.

Karakter setlerine grafik parçaları koymanın en güzel taraflarından biri de bu parçaları ekranın değişik yerlerinde tekrar tekrar kullanabilmeniz. Düşünürseniz bunun ekranın değişik yerlerinde tekrar tekrar 'A' harfini kullanmaktan bir farkı da yok. Örneğin aşağıdaki kod parçası yan yana iki araba koymanızı sağlar.

```
*=$0400
!by $40, $41, $42, $40, $41, $42

*=$0400 + (1 * 40)
!by $43, $44, $45, $43, $44, $45
```

Bundan faydalanarak ekranınızı istediğiniz zaman çok kolayca grafiklerle doldurabilirsiniz.

Karakterlerin Renkleri

Deminki örneğimizde kullandığımız araba grafiği tek renkli bir grafikti. Yani karakterleri oluşturan pikseller ya *dolu* ya *şeffaf* olabiliyordu. Şeffaf olan pikseller ekran renginde (yani $d021) görünüyordu. Peki dolu olan piksellerin rengi nasıl belirleniyor.

İşte şu ana kadar değinmediğimiz basit bir konuya gelmiş oluyoruz böylece: Renk Belleği...

C64'te, $d800 - $dbff arasında yer alan ve hayattaki tek görevi ekrandaki karakterlerin (veya ileride göreceğiniz bitmap grafiklerin) renk bilgilerini taşımak olan, 1 K'lık bir bellek çipi vardır. Bu çipe Renk Belleği adı verilir.

Renk belleği aynı video matris gibi organize edilmiştir. Video matris nasıl ekrandaki kararkter slotlarına hangi karakterlerin çizileceğini belirliyorsa, renk belleği de aynı şekilde karakter slotlarına çizilen karakterlerin *dolu* piksellerinin alacağı rengi be-

lirler.

Yani daha önce raster çizgileri yaparken kullandığımız renk değerlerini (0'dan 15'e kadarki 16 C64 rengi), renk belleğindeki adreslere yazdığımızda, o adreslere karşılık gelen slotlara çizilen karakterlerin *dolu* piksellerinin rengini belirlemiş oluruz.

```
Örne#in deminki araba örne#inde,
$d800 : 01 01 01 02 02 02 ...
$d828 : 01 01 01 02 02 02 ...
```

şeklinde renk belleğine değerleri yazarsak, Ekranda sol üst köşedeki arabamız beyaz, yanındaki araba da kırmızı olur.

Fakat renkleri renk belleğine yerleştirirken çoğu zaman direkt olarak acmeyi kullanamıyoruz. Yani şöyle bir kod parçası yanlış çalışacaktır.

```
*=$d800
!by 1, 1, 1, 2, 2, 2        ;bu kod cal##mayacak
*=$d828
!by 1, 1, 1, 2, 2, 2
```

Bunun sebebi biraz uzun ve şu an konumuz dışında. Bunun yerine renk bilgilerini bellekte başka yerlere yerleştirip programınızın içinde renk belleğine kopyalamalısınız. Yani;

```
...
ldx #0
dng: lda rb_birinci_sat#r,x
sta $d800,x
lda rb_ikinci_sat#r,x
sta $d828,x
inx
cpx #6
bne dng
...

rb_birinci_sat#r:
!by 1, 1, 1, 2, 2, 2
rb_ikinci_sat#r:
!by 1, 1, 1, 2, 2, 2
```

Çok Renkli Karakterler

Şimdi C64'ü dönemdaşı makinelerden ayıran bir diğer karizmatik yeteneğiyle karşılaşmaktasınız. Şu ana kadar karakter slotlarında hep tek renkli karakterler gösterdik. Oysa C64 bir karakter slotunda *çok renkli* karakterler de gösterebilir. Buna *çok renkli karakter modu* denir.

Çok renkli karakter modundayken bir karakter slotundaki piksellerin VIC tarafından yorumlanışı ve kullanılışı değişir. Her karakterin şekli yine 8 bayt ile ifade edilir. Fakat tek renkli moddaki gibi bu baytların herbir biti bir piksele karşılık gelmez.

Bunun yerine bir bayttaki 8 bit ikişerli gruplara bölünür. Her bir iki bitlik grup bir pikselin rengini temsil eder. Yani bir bayt ile 4 pikselin rengi ifade edilir. Bununla beraber ekranda piksellerin ebadı değişir. Bir pixelin genişliği iki kat artar ve bir karakter slotuna 8x8 değil 4x8 piksel sığmaya başlar.

```
- - - -
- - - -
- - - -
- - - -
- - - -
- - - -
- - - -
- - - -
```

Geniş pikselleri ifade etmek için nokta yerine eksi işaretini kullandık.

Bu modda dikkat ederseniz bir pikselin rengini 2 bit ile ifade edebiliyoruz. Daha önce tek renkli modda bir pikselin rengini bir bitle ifade edebildiğimiz için bir piksel ya dolu ya boş olabiliyordu. Oysa şimdi bir pikseli ifade etmek için, bir piksele 4 farklı renk verebiliriz.

```
00 : #effaf renkteki pikseller (arkalar#ndaki ekran rengi
01 : Birinci Çok Renkli mod rengi ($d022)
10 : #kinci Çok Renkli mod rengi ($d023)
11 : Renk belle#inden gelen renk ($d800 - $dbff)
```

Çok renkli modda iken karakterlerinizin 01 ve 10 değerlerine sahip pikselleri renklerini sırasıyla d022 ve d023 registerlerinden alırlar. Bu renkler ekrandaki bütün karakterler için ortaktır (tabi IRQ rutinleri ile ekranın farklı bölümlerinde bu registerleri değiştirerek farklı renkler elde edebilirsiniz.). 11 değerine sahip pikseller ise o karakterin bulunduğu slota karşılık gelen renk belleğindeki renk değerine göre renklendirilir.

Çok Renkli Karakter Moduna Girmek

Normalda C64 ilk açıldığında tek renkli karakter modundadır. Çok renkli karakterler gösterebilmek için yapmanız gereken iki şey vardır:

- d016 registerindeki çok renkli mod bitini 1 yapmak
- çok renkli karakterleri yerleştirdiğiniz slotlara karşılık gelen renk belleği adreslerine 8 ile 15 arası bir renk değeri koymak.

Özellikle ikinci madde sıklıkla unutulan ilginç bir özelliktir. Aslında bu özellik sahnesinde aynı ekranda interrupt kullanmadan hem çok renkli hem tek renkli karakterlerin gösterilebilmesi sağlar. Ekranda kendiseine karşılık gelen renk belleği değerleri, 0 ile 7 arasında olan karakterler tek renkli, 8 ile 15 arasında olan karakterler çok renkli olarak gözükür. Bir diğer unutmamanız gereken ilginç nokta Renk belleğine 8 ile 15 arası bir değer koyduğunuzda aslında karakterin 11 değerli piksellerin hangi rengi alacağı. Bu pikseller beklediğiniz üzere 8 - 15 değerlerinin karşılık geldiği renkleri değil 0 ile 7 arası değerlere karşılık gelen renkleri alırlar. Yani 8-15 arası değerden önce 8 çıkarır sonra bulduğunuz değerin hangi renk olduğuna bakarsanız, işte o renk 11 değerli piksellerin görüneceği renktir. Örneğin sol üst köşeye çok renkli bir karakter koyar ardından d800 adresine 10 ya-

zarsak, kararkterimizdeki 11 değerli pikseller 2 nolu renk olan kırmızı renginde görünecekler.

Bu acayip davranışın kafanızı karıştırmasına izin vermeyin. Kısa sürede alışacaksınız. Şimdi bir örnek görmek isterseniz, yalnız_logo.a64 dosyasına bakabilirsiniz. Bu örnekteki amacımız sadece büyük bir çok renkli logo göstermek. Elimizde karakter setine çevrilmiş bir logo var. Tıpkı daha önceki araba resmi gibi bu büyük logo da 8 x 8 piksellik alanlara bölünmüş ve bu alanların her biri birer çok renkli mod karakteri olarak karakter setine yüklenmiş. Ardından karakter indekslerinin video matrise nasıl dizileceği de ayrı bir dosya olarak hazırlanmış. Dolayısıyla elimizde bellekte iki bölgeye yüklenecek iki adet bayt grubu var: Karakter seti ve video matris baytları.

Herhangi bir büyük grafikten bu şekilde karakter seti ve video matris elde eden pekçok araç vardır. Bunlara genelde *logo converter* veya *bitmap to chars converter* gibi isimler verilir. Webde pekçok örneği vardır. Ya da kendiniz pc'de basit çeviriciler yazabilirsiniz. Böyle bir c++ çevirici bu kursun dosyaları arasında da var. SDL kütüphanesi ile derleyip kullanablirsiniz.

Ya da bu detaylara takılmayıp bir süre çalışmalarınızda örnek verdiğimiz logoyu kullanabilirsiniz.

Demin de belirttiğimiz gibi örnekte video matris ve karakter seti baytlarını bellekte uygun yerlere yerleştiriyoruz (VM $3000 adresine, Karakter seti de $2800 adresine) ardından da $d018 registerini kullanarak VIC'in o bölgeleri kullanmasını sağlıyoruz. Sonra da d016 ile çok renkli modu açıyor, ve d021, d022, d023 registerlerine istediğimiz renkleri yerleştiriyoruz. Son olarak da renk belleğinde logonun olduğu bölümlerin rengini istediğimiz renk değerinin 8 fazlası ile dolduruyoruz (Çok renkli mod öyle gerektirdiği için).

Genelde bu şekilde kullanılan logolar veya grafikler hep üç renkli olarak hazırlanır. Böylece grafiklerin kullanımı kolaylaşır. Normalde zaten 2 renk (d022 ve d023'ten gelenler) bütün karakterlerde ortak olmak zorunda. Üçüncü rengi her karakterde farklı kullanabiliriz (renk belleğinde her karakter slotu için farklı renk tanımlayabildiğimiz için). Fakat daha sonra eğer grafiğimizi sağa sola yukarı aşağı hareket ettirirsek, renk belleğindeki değerleri de kaydırmamız gerekir. oysa grafiğimizi üç renkli (şeffaf rengi saymadan) hazırlarsak renk belleğini aynı üçüncü renk değeri ile bir kere doldurup bir daha da renklerle kafamızı yormadan devam edebiliriz.

Bu şekilde üç renkli grafikler bu sebeble, C64 oyun ve demolarında çok sıklıkla kullanılır.

Müzik Çaldırmak

Müzik ile VIC çipinin ne ilgisi var? Bu soru aklınıza gelebilir. Zaten çok az ilgileri vardır. Aslında C64'te müzik çaldırma işinin büyük bölümünü meşhur SID çipi üstlenir.

Fakat yıllar içinde C64'te güzel bir standartlaşma yaşanmıştır. Her müzik programı, yapılan müziği kullanılmak üzere kaydederken, müziğin nota ve ses bilgileri ile beraber SID çipine değerleri yazarak müziğin duyulmasını sağlayan *Oynatıcı Rutini* de beraber kaydeder. Yani oynatıcı ve müzik bilgilerini bir paket haline getirir.

Buna ilaveten bütün oynatıcı rutinlerinin kullanımı da aşağı yukarı aynıdır. Hepsi dışarıdan çağrılmak üzere iki ana alt rutin barındırırlar. Bu alt rutinlere *müziği çalmaya hazırlanma* ve *müziğin bir parçacığını çaldırma* rutinleri diyebiliriz.

Neredeyse bütün oynatıcı rutinler, müziği çaldırabilmek için şöyle bir yol izler. Müziği duyabilmemeiz için olması gereken şey SID çipinin registerlerinin sürekli olarak yeni müzik bilgileri ile yüklenmesidir. İşte bir müzik programı, genellikle yapılan müziği kaydederken ses ve nota bilgilerini şu şekilde organize eder.

Zamanı saniyenin ellide biri büyüklüğünde parçalara böler. Bu parçalara *zaman parçacığı* diyelim. Müzik programı verileri öyle organize eder ki, parçanın süresi boyunca her bir zaman parçacığında hangi SID registerlerinin değerinin hangi değere değişmesi gerekiyor bilgisi hazır olmuş olur. Daha sonra oynatıcıdaki *müziğin bir parçacığını çaldırma* rutini, her çağırılışında, rutin içinde tutulan bir *parçacık sayacını* bir artırır. Bu sayede rutin her defasında parçanın kaçıncı parçacığında olduğunu bilir. Ve bu rutin içinde bulunduğu *paketlenmiş müzik* içindeki verilere bakarak şu anki parçacıkta hangi SID registerlerini değiştirmesi gerekiyor ise , bunu yapıp geri döner.

Bunun anlamı şudur. Oynatıcının veya SID çipinin detaylarını bilmeseniz bile bu paketlenmiş müzikleri kolaylıkla çaldırabilirsiniz. Çünkü o bilmediğiniz detayların hepsi oynatıcı tarafından halledilir. Sizin bğtğn yapmanız gereken parçaya başlamadan önce *müziği çalmaya hazırlanma* alt rutinini çağırmak, ardından da saniyede elli kere sabit aralıklarla *müziğin bir parçacığını çaldırma* alt rutinini çağırmaktır.

İşte bu parçacık çaldırma rutinini de genellikle bir VIC IRQ rutini içinde yaparız. bu rutinin her ekran taramasında bir kere çağrılması yeterlidir. Örnek için muzik_caldırma.a64 dosyasını inceleyebilirsiniz. Örnek çok basit olduğu için detaylı açıklamayacağız.

Bir diğer nokta da oynatıcılardaki bu *hazırlama* ve *parçacık çaldırma* rutinlerinin adresleri de standartlaşmıştır. Müzik paketleri neredeyse her zaman $1000 adresinden başlayarak yerleştirilir. Hazırlama rutini her zaman $1000 adresindedir. Parçacık çaldırma rutini de $1003 adresindedir. Yani,

```
jsr $1000
```

ile parça hazırlanır ve sonra IRQ rutininde,

```
jsr $1003
```

ile çaldırılır.

İlk İntro

İlk Planlama

Şimdi bir durumumuza bakalım. Her türlü hareketli ve renkli raster çizgileri yapabiliyoruz. Süper 3 renkli logolar gösterebiliyoruz. Yazı kaydırabiliyoruz. Müzik de çaldırabiliyoruz. E o zaman biz bayağı bayağı intro yapabilecek kıvama geldik sevgili okuyucular. Artık boru değil :)

Bir intro yapacağımız zaman önce bir kağıt kalem alıp ekranda nereye ne koyacağımızı planlamakta fayda vardır. Şimdi ilk intromuzu planlayalım:

Ekranın üst bölgesinde logomuz dursun. Logomuzun arka planı normalde siyah. Ama bizim intromuzun bütün arka planı siyah olmasın. Diyelim ki hakim renk olarak kırmızı seçelim. Yani intromuz alev alev yansın.

Öyleyse logomuzun göründüğü alanda altına ve üstüne birer beyaz raster çizgisi çekelim. Aralarında kalan bölgede (yani logonun arka planında) ekran rengi siyah olsun.

Logonun bir miktar altında kırmızıdan beyaza sabit raster çizgileri ile yumuşak bir geçiş yapalım. Beyaz olan bölgeye *ilk intro* yazalım. Ardından aynı geçişin tersi ile kırmızıya dönelim. Yani ekranın orasından bir beyaz bant geçiyormuş gibi.

Biraz aşağıda aynı renk geçişleri ile yine aynı şekilde bir beyaz bant yapalım. Fakat bu sefer bu bantın içine bir kayan yazı yerleştirelim.

Son olarak da kayan yazının altında yine kırmızı zemin üzerinde hareketli_raster2 örneğimizdeki gibi bir efekt yapalım.

Bunlara ilave olarak bir yerlerde müzik çaldırma rutinini de çağıracağız.

Raster Zamanı Yetecek mi?

Genelde kendinize sormanız gereken ilk sorulardan biridir bu. Raster zamanı ne demek? Dikkat ederseniz bütün efektlerimiz IRQ rutini içindeki kodlarla yapılıyor. Bizim IRQ rutininde çalıştırdığımız her komuta karşılık VIC ekranı taramaya ve monitöre bilgileri göndermeye devam ediyor. Dolayısıyla biz ekranın bir bölümünde IRQ rutininde bir efekti gerçekleştiren komutları yazarken bizi kısıtlayan bir konu var. Raster ilerleyip bir altındaki efekti yapacağımız alana gelmeden önce işimizi bitimek durumundayız. Yoksa bir alttaki efekte yetişemiyoruz.

Örneğin diyelimki 120. raster satırından itibaren kayan yazı rutini var (yani kayan satır hakikaten oralarda bir yerde ekranda). Diyelim ki 128. satırda kayan bölgenin sonuna geldik. Burada slot kaydırma rutinini çağıracağız. Fakat slot kaydırma rutinindeki kopyalama işlemini tamamlayana kadar VIC 7-8 raster satırı daha ilerliyor. Dolayısıyla diyelim ki biz 130. raster satırına bir çizgi koymak istersek olamayacak. Çünkü bir önceki efekti 128 ila 130. satırlar arasında bitiremedik. Öyleyse ya raster çizgisini daha ileride bir yerde koymaya razı olacağız ya da efekt ile ilgili bazı değişiklikler yapacağız.

İşte ekrana koyduğunuz her rutin böyle bir miktar zaman yiyor. Ve bizim açımızdan bu *zaman*, ekranda *kaplanan* ve *orada başka işlem yapılamayan* bir raster aralığı demek. Bu yüzden genelde bir rutinin harcadığı zamandan bahsederken birim olarak saniye veya cycle kullanmak yerine raster çizgi sayısı kullanabiliyoruz. İşte buna *raster zamanı* deniyor.

Dolayısıyla İntronuzda ekrana koyduğunuz efektlerin raster zamanlarının toplamı, elinizdeki bir taramada kullanılabilen raster toplamından (yani 312'den) az olmak zorunda. İşte bu yüzden yaptığınız efektlerin her birinin raster zamanlarına dikkat etmeye otomatik olarak başlayacaksınız. Her efekt için ne kadar raster zamanı harcadığınızı bulmanın yollarına sonra değineceğiz.

IRQ Rutini Algoritması

IRQ rutinimiz ekrandaki bütün efektleri yapan rutin olduğu için onun algortimasını biraz düşünelim. Aslında çoğu zaman IRQ rutinimiz temel iskeleti yaklaşık olarak şöyle birşey olacak:

- inc $d019
- ilk efekti yap
- d012'nin ikinci efekt satırına gelmesini bekle
- ikinci efekti yap
- d012'nin üçüncü efekt satırına gelmesini bekle
- üçüncü efekti yap
- jmp $ea81

Basit bir mantık değil mi? Şimdi buraya kadar konuştuklarımızı ve intro planımızı göze alarak bu intro için IRQ rutinimizin algoritmasını biraz daha detaylı olarak ortaya çıkaralım:

- inc $d019
- müziğin bir parçacığını çaldır
- $30 nolu satıra kadar bekle
- $31 nolu satıra bir beyaz raster çizgisi koy. sonra ekranı siyah yap
- Çok renkli moda geç ($d016) ve $d018'i logoya yönlendir
- $70 nolu satıra kadar bekle
- bir beyaz çizgi koy. sonra ekranı kırmızı yap. tek renkli moda dön ve d018'i logoya değil fontlara bakacak şekilde değiştir
- $7e nolu satıra kadar bekle
- birinci beyaz bant rasterlerını çiz
- $93 nolu satıra kadar bekle
- bir piksel kayma rutinini yap

- beyaz bant rasterlarını çiz
- bantlar bittikten sonra d016'yı yine sabitle ve slot kaydırma rutinini çağır
- $be nolu satırına kadar bekle
- hareketli raster bölgesini çiz ($f0'a kadar)
- ekranı tekrar kırmızı yap
- raster animasyonunu yap (bir sonraki karenin raster renklerini hazırla)
- jmp $ea81

İşte bu kadar basit. IRQ rutinimiz çok önemli olmakla beraber bir diğer önemli nokta da intromuz başlarken, daha IRQ hazırlanmadan önce yapılan *ilk hazırlıklar*.

- renk belleğine gereken renkleri koy
- müziğin *ilk hazırlanma* rutinini çağır.
- IRQ Hazırla
- sonsuz döngü

İşte programın başında bu ilk işlemleri yapıp IRQ rutinimiz başlatarak intromuzu izleyenlere sunacağız.

Artık kafamızda algoritma iyi kötü oturmuş durumda. Kodu yazarken karşımıza yeni problemler çıkabilir ancak bunları karşımıza çıktıkça çözebilecek kadar hazırlıklıyız. Çünkü iskelet tasarımımızı oturttuk.

Şimdi kodu inceleyebilirsiniz. Örnekler klasöründen intro_01.a64 dosyasını açın ve inceleyin. Göreceğiniz bütün rutinler önceki örneklerin bir araya getirilmiş hali. Bu yüzden satır satır açıklamayacağız.

Dikkatinizi çekmek istediğimiz bir nokta RasterIRQHazırla rutininde. Burada CLI komutundan once d019'a bir yazarak bir nevi inc d019 yapmış oluyoruz. Bunun sebebi eğer biz IRQ kurarken herhangi bir IRQ VIC'ten gelmiş ise IRQ rutinine hemen CLI komutuyla beraber giriyoruz. Yani yanlış rasterda olsak bile. Bu şimdiye kadar problem olmamıştı. Çünkü kısa IRQ rutinleri kullanıyorduk. Birinci girişten sonraki girişler her zaman doğru rasterda oluyordu. Oysa bu örnekte IRQ rutinimiz çok uzun ve ilk girişte yanlış rasterda başlarsak biz çıkana kadar doğru raster'ı kaçırabiliriz. Bu konu kafanızı karıştırıyorsa endişelenmeyin çok önemli değil. Sadece bundan sonra IRQHazırlama rutinlerinizi hep böyle yapın yeter.

Evet şu an bu ilk ıntroyu kurcalama zamanıdır. Bir sonraki bölümümüzde introlarımızda kulanabileceğimiz bazı efektler öğreneceğiz. Bu efekleri bu bölümde oluşturduğumuz iskelet içine oturtacağız. Oldukça zevkli ve görkemli bir bölüm olacak.

O zamana kadar iyi eğlenceler.