# Xlib ve İlgili Diğer Teknolojilere Giriş


**Figure 1:** Xlib ve İlgili Diğer Teknolojilere Giriş

## İçindekiler

- Bölüm 6 - Değişen pencere boyutuna uyum
- Bölüm 7 - Bu titreme hakkında…
- Bölüm 8 - KeyPress ve KeyRelease
- Bölüm 9 - XInput'tan UTF-8 Karakterler (Hayır, oyun çubuğu olanından değil)
- Sonuç

### Derleme

Rehberi doğrudan GitHub üzerinden okuyabileceğiniz gibi PDF ve EPUB gibi biçimlere çevirip bu dosyalardan da okuyabilirsiniz. Örneğin PDF olarak derlemek için uzun kod satırları sorunlu olsa da iş görecek şu basit yöntemi kullanabilirsiniz:

```
pandoc README.md \
       giris/README.md \
       bir-pencere-olusturma/README.md \
       uygulamayi-kapatma/README.md \
       minimum-boyut/README.md \
       pencereyi-buyutme/README.md \
       daha-iyi-kapatma/README.md \
       arabellek/README.md \
       degisen-pencere-boyutuna-uyum/README.md \
       bu-titreme-hakkinda/README.md \
       keypress-ve-keyrelease/README.md \
       utf8/README.md \
       sonuc/README.md --pdf-engine=xelatex -o xlib-rehberi.pdf
```

pandoc-latex-template veya benzeri bir şablonla daha iyi sonuçlar elde etmeniz olası, örneğin:

```
pandoc README.md \
       giris/README.md \
       bir-pencere-olusturma/README.md \
       uygulamayi-kapatma/README.md \
       minimum-boyut/README.md \
       pencereyi-buyutme/README.md \
       daha-iyi-kapatma/README.md \
       arabellek/README.md \
       degisen-pencere-boyutuna-uyum/README.md \
       bu-titreme-hakkinda/README.md \
       keypress-ve-keyrelease/README.md \
```

```
utf8/README.md \
sonuc/README.md --pdf-engine=xelatex -o xlib-rehberi.pdf --template eisvo
```

## Giriş

Programcılar sadece Microsoft dünyasında kalmak istemediklerinde Linux'ta pencerelerle doğrudan çalışmamak konusunda uyarılmışlardır. Bunu yapmak yerine GTK, Qt veya SDL kullanmaları söylenmiştir.

Bu, akıl sağlığınız için değerli bir öneri olsa da kütüphanelerin yardımı olmadan bunun nasıl yapılacağına dair eğitim içeriklerinin eksikliğine de yol açmıştır. Bu, sadece bu tür programlama yapmak isteyenler veya yeni kütüphaneler yazmayı düşünenler için işleri daha da zorlaştıran bir durumdur.

Tüm bu nedenlerden dolayı bu konuda yazılmış derli toplu bir kaynak olan A tour through Xlib and related technologies'i çevirmeye karar verdim. Bu yazı büyük oranda aslına sadık bir çeviridir.

Ben buradaki bilgilerden SFML'i Haiku'ya aktarırken Xlib ile ilgili fonksiyonları Haiku'nun yerel fonksiyonlarına çevirebilmek için yararlı olacak genel bilgileri edinmek için yararlanacağım.

### Başlamadan Önce

Başlamadan önce tartışmamız gereken birkaç şey var. Bu eğitici X ile kullanım için Xlib kütüphanesiyle ilgilidir, yeni xcb kütüphanesi hakkında değildir. Xlib'in kullanımdan kaldırılması ve yerine xcb'nin geçmesi gerektiği düşünülse de xcb, opengl ile bağlantı kurmak için kullanılamaz ve ek olarak Xlib'e de çok benzerdir. Anlayabildiğim kadarıyla bugün Xlib olmadan bir şey yapmanın gerçekten bir yolu yoktur. Bu durumda, bir projede iki çok benzer kütüphaneyi kullanmanın bir mantığını görmüyorum ve bu bence Xlib'i herhangi bir durum hakkında üstün bir seçim haline getiriyor. Ayrıca, Xlib'in kullanımdan kaldırılmış olarak tanımlandığı gerçeğinden endişelenmenize gerek olmadığını düşünüyorum. Xlib hala en çok kullanılan iki kütüphaneden biridir. Büyük olasılıkla Xlib desteğinin sonunu görmeden önce X sisteminin ölümünü göreceğiz.

Bunları aklımızda tutarak, xlib belgelerini elimizin altında tutmalıyız. Resmi belgeler www.x.org/wiki/ProgrammingDo adresinde bulunabilir. Bununla birlikte belgeler için çoğunlukla tronche.com/gui/x/xlib adresindeki belgeler kullanılmıştır.

Bu eğitseli bitirdiğinizde her zaman belgelere erişin ve yeni fonksiyonlara denk geldikçe bunları okuyun. Bu belge ne yapacağınız ve nereden başlayacağınız konusunda size bir fikir verebilir, ancak resmi ve daha detaylı belgelerdeki özellikleri okumak her zaman daha iyidir.

İleride gerekli olabilecek ilgili standart belgeleri specifications.freedesktop.org/wm-spec/1.3 ve www.x.org/releases/X11R7.6/doc/xorg-docs/specs/ICCCM/icccm.html adreslerinden bulunabilir.

Bunlar pencere yöneticisini kontrol etmek için gereklidir. Pencere yöneticisinin ne olduğu hakkında daha sonra kısaca konuşulacak.

Bu örnekleri inşa etmek için çok fazla kurulum gerekli değildir, sadece kodu derleyicinize aktarın ve -lX11 seçeneğiyle bağlayın. Ayrıca libx11-devel veya benzeri bir pakette bulunması muhtemel xlib geliştirme dosyalarına da ihtiyacınız vardır.

### Genel Bakış

Linux'taki pencere sisteminin birkaç parçası vardır. Aşağıdakilerin, ayrıntıların çoğunu özetleyen çok kaba bir genel bakış olduğunu unutmayın. Daha çoğunu bilmek istiyorsanız, tüm bu şeyler hakkında epey uzun Vikipedi makaleleri vardır.

- **X:** Bazen X.org, X-Server veya sadece X olarak adlandırılır. Bu şeylerin hepsi biraz farklı anlamlara sahiptir, ama bunun için endişe etmeyelim. Bu belgede buna sadece X denecektir. X, tüm pencerelerin durumunu depolayan yazılımdır. X, cihaz girişini alır ve bunu bir olay sistemi aracılığıyla uygulamalara sunar. Bu sayede pencereleri oluşturmayı kolaylaştırır. X aynı zamanda GPU sürücüsünün sıklıkla bağlandığı Linux sisteminin bir parçasıdır. Genel olarak Linux'ta OpenGL kullanmak için bir X uygulaması yapmanız gerekir. X, sunucu yazılımıdır, istemciler aynı veya farklı bilgisayarlarda olabilir.
- **Ekran yöneticisi:** Bu bir X istemci oturumunu tanımlayan programdır. Aynı veya farklı bir makinede bir X sunucusuna bağlanır. X'in hemen hemen tüm çağrıları için ekranı, bir istemci kimliği olarak sunmanız gerekir.
- **Pencere yöneticisi:** X çoğunlukla sadece pencerelerin durumunu depolar ve onlara olayları aktarır, pencere yöneticisi pencerelerinizin ne yaptığını, nasıl göründüklerini, ilk olarak ekranda nerede ve ne boyutta göründüğünü önemseyen programdır. Pencere yöneticisi ayrıca Linux diyarına pencere dekorasyonu adı verilen, başlık çubuğu ve düğmelerini de katar. Bu bizim için önemlidir çünkü bazı durumlarda pencere yöneticisiyle etkileşimde bulunmak için kullanılan UPA, X'in kendisiyle etkileşim kurmak için olandan farklıdır.
- **X uygulaması**. Bu biziz, daha doğrusu yapacağımız şeylerdir! Bu ayrıca Xlib'in de devreye girdiği yerdir. X, orijinal olarak ağ üzerinden kullanılmak üzere tasarlandı ve bugün hala bir dereceye kadar bu şekilde çalışır. Ama çoğunlukla bunun bizim için anlamı, Xlib'i kullandığımız zaman, X'in fonksiyonlarını doğrudan X'ten çağırmadığımızı bilmemiz gerektiğidir. Xlib, X tarafından kullanılan ağ protokolü etrafında bir C sarmalıyıcısıdır. Bu, yaptığımız her şeyin başka bir iş parçacığında, başka bir işlemde eşzamansız olduğu anlamına gelir. Değişikliklerin gerçekten etkili olduğundan emin olmak istediğimizde komut tamponunu temizlememiz gerekir.

### Sonunda Kodlar

Bu belgede kullanılan tüm kodlar GitHub'da mevcuttur. Parça numarası dosya adında belirtilmiştir. Bu dosyaların referans olması için muhtemelen iyi bir fikirdir.

<- İçindekiler Bölüm 0 - Bir Pencere Oluşturma ->

## Bölüm 0 - Bir Pencere Oluşturma

Adım adım kodlarda ilerleyelim. Kodları tam haliyle görmek için GitHub'daki dosyalara bakabilirsiniz.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>

int main(int argc, char** args)
{

}
```

Bu muhtemelen Windows'tan gelen insanlar için ilk küçük farktır. Pencereyi oluşturmak isteyen programlar için özel bir main fonksiyonu yoktur, sadece standart C `main()` fonksiyonu vardır. Sonraki birkaç kod parçası `main()` fonksiyonuna aittir.

```
    int width = 800;
    int height = 600;

    Display* display = XOpenDisplay(0);

    if(!display)
    {
        printf("Hiçbir ekran yok\n");
        exit(1);
    }
```

```
Window root = DefaultRootWindow(display);
int defaultScreen = DefaultScreen(display);
```

İlk olarak daha sonra kullanmak için genişlik ve yüksekliği tanımlarız. Daha sonra (0 geçerek) standart ekranı açıyoruz. X, bir ağ protokolüne sahip bir istemci-sunucu modeli olarak kurulmuştur. Bu, sunucunun, hangi istemcilerin var olduğu ve hangi pencerelerin başka pencerelerle etkileşimde bulunabileceği hakkında bir fikre sahip olması gerektiği anlamına gelir. X sunucusunun kiminle konuştuğunu da bilmesi gerekiyor, bu yüzden display (bazı yerlerde ekran olarak da kullanılabilir, kastedilen her zaman display'dir) işaretçisini tutmalıyız. Bunu diğer çoğu Xlib fonksiyonuna ileteceğiz. Ayrıca, çağrının geçerli bir ekran döndürdüğünü kontrol ederiz.

Daha sonra bir kök pencere alırız. Pencereler bir ağaç yapısında düzenlenmiştir, bu nedenle oluşturduğumuz her pencerenin bir ebeveyninin olması gerekir. Burada mormal bir en üst düzey pencere oluşturuyoruz, dolayısıyla ebeveynimizin masaüstü olması gerekiyor, bu kök penceredir. Ayar penceresi olan bir uygulama yapıyor olsaydık ana penceremiz için ayarlar penceresini ebeveyn olarak kullanabilirdik.

Son olarak öntanımlı ekranı alırız. X'teki bir ekran genellikle sadece bir alan veya oluşturulacak bir arabellektir, bu bir grafik aygıtından gerçek bir monitöre veya tamamen yazılım içinde bir bellek içi hedefe olabilir.

```
int screenBitDepth = 24;
XVisualInfo visinfo = {};

if(!XMatchVisualInfo(display, defaultScreen, screenBitDepth, TrueColor, &vis
{
    printf("Hiçbir eşleşen görsel bilgi yok\n");
    exit(1);
}
```

Burada tarama hedefimiz için ne tür bir gereksinim olduğunu tanımlarız. Bu, tek renkli ekranlar ve farklı renk derinliklerinde görüntüler olduğunda, yani X'in çıktığı 1984'te çok daha ilginçti. Bugünlerde bu neredeyse her zaman aynıdır. RGB-8-8-8 ve TrueColor için 24 bit.

`XMatchVisual` gereksinimlerimizi alır, bunları bu donanımla desteklenen bir dahili `visualInfos` listesiyle eşleştirir ve en uygun olanı döndürür. Ya da bizim gereksinimlerimizi karşılamak mümkün değilse uygun olmadığını söyler ama bu günlerde bu pek olası değildir.

```
XSetWindowAttributes windowAttr;
windowAttr.background_pixel = 0;
windowAttr.colormap = XCreateColormap(display, root,
```

```
                                        visinfo.visual, AllocNone);
    unsigned long attributeMask = CWBackPixel | CWColormap;

    Window window = XCreateWindow(display, root,
                                  0, 0,
                                  width, height, 0,
                                  visinfo.depth, InputOutput,
                                  visinfo.visual, attributeMask, &windowAttr);

    if(!window)
    {
        printf("Pencere düzgün şekilde oluşturulamadı\n");
        exit(1);
    }
```

Pencereyi oluşturduğumuz yer burası. `XSetWindowAttributes` yapısında, pencere için bazı öznitelikleri ayarladık. Arkaplan dolgu pikselini siyah olarak belirledik ve daha önce oluşturduğumuz visualinfo'dan bir renk haritası oluşturduk. Sonra `XCreateWindow`'a hangi özelliklerin kullanılmasını istediğimizi söyleyen maskeyi, `attributeMask`'ı tanımlarız. Çünkü bütün bu ayarladıklarımızı kullanmak istiyoruz.

Belgelerine bakarsanız `XCreateWindow` bu noktada oldukça açıklayıcıdır. Şimdiye kadar hazırladığımız tüm bilgileri ona aktarıyoruz. `InputOutput`, bu pencerenin gelen olayları alıp ele alacağı ve aynı zamanda ekrana çıkış vereceği anlamına gelir. Daha sonra, pencere oluşturmanın çalışıp çalışmadığını kontrol ederiz.

```
    XStoreName(display, window, "Merhaba Dünya!");

    XMapWindow(display, window);
    XFlush(display);

    return 0;
```

Önce penceremize uygun bir başlık verelim. Pencere başlığı, pencere yöneticisinin bir özelliğidir. `XStoreName()`, bu UPA'yı bizden bir süre gizleyen kolaylaştırıcı bir fonksiyondur. Şimdi pencereyi ekrana eşleştiriyoruz. Bu, pencerenin görünür hale geleceği anlamına gelir. Daha sonra tüm komutlarımızın sunucuya geçtiğinden emin olmak için temizleriz. Şimdi bunu çalıştırırsak hiçbir şey olmaz. Bunu bir hata ayıklayıcıda başlattığımızda ve son satırda duraklattığımızda pencerenin başarıyla oluşturulduğunu ancak hemen kapatıldığını görüyoruz. Şimdilik fena değil.

Aslında bir pencereyi hiç uğraşmadan göstersek daha iyi değil mi? Bunun için `XFlush()`'ın sonrasına aşağıdakileri ekleyin:

```
while(true)
{
}
```

Programımızı çalıştırdığımızda, pencerenin açık kaldığını görebiliyoruz. Şimdi pencerede kapat düğmesine basabiliriz ve pencere kapanır!

Ama bekle bir saniye. Programı sonsuz bir döngüde sıkıştırdık, bu döngüden nasıl uzaklaşabiliriz? Görev yöneticimize bakarsak, mutlu küçük penceremize ait sürecin hala devam ettiğini görebiliriz. Bunun nedeni pencerenin kendi programımızın sürecine doğrudan bağlı olmamasıdır. Pencereyi pencere yöneticisinden ve X sunucusunun kenarından kapatmak, sürecimizi durdurmaz.

<- Giriş Bölüm 1 - Uygulamayı Kapatma ->

### Bölüm 1 - Uygulamayı Kapatma

Bu rahatsızlığı çözmek için, pencere kapalıyken `while` döngüsünü `false` olarak ayarlamanın bir yolunu bulmaya ihtiyacımız var. Bunu yapmanın yolu olaylardan geçiyor.

```
int windowOpen = 1;
while(windowOpen)
{
    XEvent ev = {};
    while(XPending(display) > 0)
    {
        XNextEvent(display, &ev);
        switch(ev.type)
        {
        case DestroyNotify: {
            XDestroyWindowEvent* e = (XDestroyWindowEvent*) &ev;
            if(e->window == window)
            {
                windowOpen = 0;
            }
        }
        break;
        }
```

```
        }
    };
```

Önceki bölümdeki `while` döngüsünü bununla değiştiriyoruz. Bu temel olay döngüsüdür. Bekleyen olaylar varken çalışır. Bir olay olduğunda, olay yığınından bir olay alır ve olay türüne göre anahtarlama yapar. Burada önemsediğimiz `DestroyNotify` olayıdır, genel olayı bu belirli olaya dönüştürmemiz gerekiyor. Şimdi olayın kendi penceremizde hedeflenip hedeflenmediğini kontrol ediyoruz ve eğer durum buysa `windowOpen`'ı `false` olarak ayarlıyoruz.

Burada dikkat edilmesi gereken, her bir olayın hangi pencere için kullanıldığını söylemesidir. Zaten sadece kendi ana penceremize ait olan olayları almalıyız ancak bu, birkaç pencere açtıysanız doğru olmayabilir, emin olmak için kontrol ediyoruz. Bu kontrolün neden mevcut olduğunu göz önünde bulundurmayı unutmayacaksanız yalnızca bir pencereniz açık olduğu sürece onu kullanmayabilirsiniz.

Olay işlemenin çalışmasını sağlamak için yazdığımız kodda ayarlanması gereken bir şey var. `windowAttrs`'ta bir ek değer daha ve `attributeMask`'ta bazı yeni değerler eklemek zorundayız (Bu kod `XCreateWindow` ile pencere oluşturmadan önce gelmeli).

```
windowAttr.event_mask = StructureNotifyMask;
unsigned long attributeMask = CWBackPixel | CWColormap | CWEventMask;
```

Pencere niteliklerimizi ayarladığımızda, bir başka nitelik olan `event_mask`'ı ekliyoruz. Belgelere bulabileceğiniz gibi, `DestroyNotify` olayı, `StructureNotify` grubunun bir parçasıdır, bu yüzden pencerede bu tür olayları almak istediğimizi X'e bildirmek için `StructureNotifyMask` değerini ayarlamak zorundayız. Bu tür bir olaya dahil olma durumu, aldığımız olayların miktarını ve dolayısıyla aslında önemsemediğimiz olaylar aracılığıyla çalışan olay döngüsünde geçirdiğimiz zamanı sınırlar. Hangi özniteliklerin ayarlandığını `XCreateWindow()`'a bildiren öznitelik maskemizde `CWEventMask`'ı ekliyoruz.

Şimdi açık kalan bir pencere ve pencere kapatıldığında sona eren bir uygulamamız var. Oldukça muntazam ha?

<- Bölüm 0 - Bir Pencere Oluşturma Bölüm 2 - Minimum Boyut ->

### Bölüm 2 - Minimum Boyut

Artık işin zor tarafını hallettik, UPA'nın geri kalan işleyişine daha çok alışmak için bazı incelikler ekleyelim. Basit bir şeyle başlayalım.

Pencerenin kenarını tutup daha küçük olacak şekilde yeniden boyutlandırdığımızda pencereyi ne kadar küçük yapabileceğimize dair bir sınır olmadığını fark ederiz. Bunu değiştirelim ve en küçük boy için bir sınır verelim.

```
void setSizeHint(Display* display, Window window,
                 int minWidth, int minHeight,
                 int maxWidth, int maxHeight)
{
    XSizeHints hints = {};
    if(minWidth > 0 && minHeight > 0)
        hints.flags |= PMinSize;
    if(maxWidth > 0 && maxHeight > 0)
        hints.flags |= PMaxSize;

    hints.min_width = minWidth;
    hints.min_height = minHeight;
    hints.max_width = maxWidth;
    hints.max_height = maxHeight;

    XSetWMNormalHints(display, window, &hints);
}

// ...
// main içinde, XMapWindow()'dan önce

setSizeHint(display, window, 400, 300, 0, 0);
```

Bir `XSizeHints` yapısını doldurup ayarlıyoruz. Bayrakları sadece sıfır olmayan bir boyut geçildiğinde ayarlayarak, 0'ın ayarlanmamış anlamına gelmesini sağlarız. Verilen parametrelerle fonksiyonumuzu çağırdığımızda istediği kadar büyük, ancak 400x300'den küçük olmayan bir pencere alırız.

Boyut ipuçları, pencere başlığı gibi bir başka pencere yöneticisi özelliğidir ve kolaylık fonksiyonlarından bir diğeridir.

<- Bölüm 1 - Uygulamayı Kapatma Bölüm 3 - Pencereyi Büyütme ->

### Bölüm 3 - Pencereyi Büyütme

Şimdi de pencereyi büyütmeye çalışalım. Bunun için herhangi bir kolaylık yok. Bu bölümde pencere yöneticisi, atomları ve özellikleriyle iletişim kurmak için kullanılan UPA hakkında bilgi edineceğiz. Bu kesinlikle başlangıçta belirtilen pencere yöneticisi belgelerini açmanızın gerektiği zamandır.

`Atom`'ları anlamak zor değildir, onların karmaşık görünmesine neden olan sadece saçma isimleridir.

Penceremize ait bilgi ve özellikler, X'in içinde saklanır ve bunlar özellik olarak adlandırılır. Özellikler benzersiz etiketlerle tanımlanır, bu etiketlere de `atom` denir. Özünde, X bazı nedenlerden dolayı anahtar-değer çiftleri olan atom-özellik çiftlerini çağırır. Muhtemelen nedenini sormamak en iyisi.

```
Status toggleMaximize(Display* display, Window window)
{
    XClientMessageEvent ev = {};
    Atom wmState = XInternAtom(display, "_NET_WM_STATE", False);
    Atom maxH = XInternAtom(display, "_NET_WM_STATE_MAXIMIZED_HORZ", False);
    Atom maxV = XInternAtom(display, "_NET_WM_STATE_MAXIMIZED_VERT", False);

    if(wmState == None)
        return 0;

    ev.type = ClientMessage;
    ev.format = 32;
    ev.window = window;
    ev.message_type = wmState;
    ev.data.l[0] = 2; // belirtime göre _NET_WM_STATE_TOGGLE 2; Benim başlıkları
    ev.data.l[1] = maxH;
    ev.data.l[2] = maxV;
    ev.data.l[3] = 1;

    return XSendEvent(display, DefaultRootWindow(display), False,
                      SubstructureNotifyMask,
                      (XEvent *)&ev);
}

    // ...
    // main içinde, XMapWindow() sonrasında

    toggleMaximize(display, window);
```

Atom-özellik çiftleri olay sistemi aracılığıyla X'e gönderilir. İstemciden sunucuya akan, `ClientMessage` isminde özel bir olay türü vardır.

Atom'ların ismine göre sorgulanması gerekir. Fonksiyon ayrıca eğer mevcut değilse bir `Atom`'un yaratılıp yaratılmayacağını bilmek ister, bunu istemeyiz. Var olması gereken standartlaştırılmış `Atom`'ları kullanıyoruz, eğer yoklarsa muhtemelen bir yazım hatası yaptık ve bunu bilmek isteriz. `Atom`'lar sadece tamsayı indeksleridir, ama özel bir `None` değeri de vardır, bu yüzden buna karşı

kontrol ediyoruz. Ayrıca, sadece _NET_WM_STATE Atom'unu kontrol ederiz çünkü belirtime göre bu atom'un varlığı diğer ikisinin varlığını da gerektirir.

Şimdi istemci mesajını dolduruyoruz. Tür, önceden tanımlanmış bir int olan `ClientMessage`'dır.

Biçim 32'dir. Bu, 32 bit anlamına gelir ve bu mesajın alıcısına `ev.data.*` dizisine ne tür veriler eklediğimizi anlatır. Bu veri dizisi parçası aşağıdakileri içeren bir birleşik yapıdır:

```
union {
    char b[20];
    short s[10];
    long l[5];
} data;
```

Biçim için olası değerler, `char`, `short` ve `long` türlerine karşılık gelen bit boyutları olan 8, 16 veya 32'dir.

Sonra penceremizi ve `atom message_type`'ımızı geçiyoruz. Veri dizisi, belirtime göre anahtar belirtmek için 2 ile başlar (NETWMSTATETOGGLE öğesini 2 olarak tanımlar, ancak bendeki başlık dosyaları bu tanımlamayı içermiyor). Bunu takiben, dikey ve yatay maksimizasyon için iki `atom` vardır. Son numara bir kaynak göstergesidir, bu programınızın normal bir kullanıcı uygulaması, bir görev çubuğu gibi bir şey veya protokolün önceki ve sonraki bir sürümü olup olmadığını gösteren bir sayıdır. Normal bir uygulama için kaynak göstergesi 1'dir.

Olayımız bittiğinde onu yoluna göndeririz. Yine, _NET_WM_STATE belirtimine uygun olarak, bu tür bir mesajın kök pencereye (masaüstü gibi bir şey) gönderilmesi gerektiğini biliyoruz. Kullanılacak `event_mask` benim için tamamen anlaşılır değil, fakat benim testlerimde maksimizasyon sadece verilen bir maske varsa etkili oluyor ve istenen pencere yapısındaki herhangi bir değişiklik olarak ifade edilebilecek `SubstructureNotifyMask`, bu durumda bir anlam ifade eder.

Tüm bu bilgilerin nereden geldiğini ve bu bilgileri nasıl bulabileceğinizi anlayabilmeniz için bu noktada belgelendirmeye ve belirtimlere kendinizi alıştırdığınızdan emin olun. Farklı sorunları kendiniz çözmeniz gerektiğinde öğrendiğiniz en önemli beceri bu olacaktır.

<- Bölüm 2 - Minimum Boyut Bölüm 4 - Daha İyi Kapatma ->

### Bölüm 4 - Daha İyi Kapatma

İlk iş olarak, `toggleMaximize()` fonksiyonunu kaldıralım. Bu, pencere yöneticisi UPA'sını göstermek için iyi bir yoldu ama şu anda ona ihtiyacımız yok.

```
    XMapWindow(display, window);
```

```
    //toggleMaximize(display, window);
    XFlush(display);
```

Artık atom'ları ve ClientMessage olaylarını öğrendik, programımızın kapanmasını ve pencereyi kapatmasını iyileştirebileceğiz. Programınızı bir terminalden çalıştırıyorsanız, pencereyi kapatırken bir hata mesajı aldığınızı fark etmiş olabilirsiniz. Bunun nedeni, pencere yöneticisinin pencereyi sizin için kapatması ve bunun en zarif şekilde olmamasıdır. Bu nedenle pencere yöneticisinin, size pencerenin kendisini kapatmadan önce pencere kapatma düğmesine basıldığını söylemesinin özel bir yolu vardır. Ama bunu kullanmak için bu özelliği tercih etmeliyiz ve yine de işe yaramazsa diye eski yöntemle yapma yolunu da tutmaya devam etmeliyiz.

```
    // XFlush()'tan sonra
    Atom WM_DELETE_WINDOW = XInternAtom(display, "WM_DELETE_WINDOW", False);
    if(!XSetWMProtocols(display, window, &WM_DELETE_WINDOW, 1))
    {
        printf("WM_DELETE_WINDOW özelliği kaydedilemedi\n");
    }

                // DestroyNotify case'inden sonra
                case ClientMessage:
                {
                    XClientMessageEvent* e = (XClientMessageEvent*)&ev;
                    if((Atom)e->data.l[0] == WM_DELETE_WINDOW)
                    {
                        XDestroyWindow(display, window);
                        windowOpen = 0;
                    }
                }
                break;
```

Burada yaptıklarımız çoğunlukla tanıdık olmalı. Bir atom alıyoruz, bununla ilgili ayrıntılar belirtimde yer almakta.

Daha sonra pencere yöneticimize pencere silme hakkında bir olay almak istediğimizi bildirmek için penceremizde bir özelliği ayarlayacak XSetWMProtocols fonksiyonunu kullanırız, böylece silmeyi kendimiz yapabiliriz.

Bu, bizim olay döngüsümüzde yaptığımız şeydir. Bir ClientMessage alırız, pencerenin silinmesiyle ilgili bir mesaj olup olmadığını kontrol ederiz ve sonra penceremizi kendimiz yok ederiz ve artık penceremiz gittiğinden döngümüzden çıkılmasını söyleriz. Bu, herhangi bir hata mesajının oluşmasını önler.

<- Bölüm 3 - Pencereyi Büyütme Bölüm 5 - Arabellek ->

### Bölüm 5 - Arabellek

Pencereye nasıl bir şey ekleyeceğimizi göstermek için doğrudan bir bellek arabelleğine yazacağız, daha sonra da bunu bir Xlib fonksiyonu kullanarak pencereye çizeceğiz.

`XFlush()` fonksiyonunun hemen ardından aşağıdaki büyük kod parçasını ekliyoruz.

```
int pixelBits = 32;
int pixelBytes = pixelBits / 8;
int windowBufferSize = width * height * pixelBytes;
char* mem  = (char*)malloc(windowBufferSize);

XImage* xWindowBuffer = XCreateImage(display, visinfo.visual, visinfo.depth,
                                     ZPixmap, 0, mem, width, height,
                                     pixelBits, 0);
GC defaultGC = DefaultGC(display, defaultScreen);
```

İlk iki satır kendini açıklıyordur zaten. Sadece resmimizin 32 bit boyutunda piksellere sahip olacağını, sonra gerekeceği için bunu bayt haline çevrilmesini, daha sonra da ihtiyacımız olan bellek miktarını hesaplamak için genişliğimiz ve yüksekliğimizle çarpımını tanımlarız. Sonra bu hafıza miktarını `malloc` ile bellekte tahsis ederiz.

İş sonraki iki fonksiyondadır. Bu görüntü arabelleğini sadece kendimiz için istemediğimizden ve pencerede görüntülemek için X'e vermemiz gerektiğinden onu X uyumlu bir görüntü yapısına sarmamız gerekiyor.

`XCreateImage`'e ekranı ve penceremiz için yarattığımız `visinfo`'daki görseli geçeriz. Daha sonra, yine `visinfo`'dan piksel derinliğini veririz. Bir sonraki argüman ilginçtir.

Burada `ZPixmap`'i biçim için geçiyoruz. Ama neden `ZPixmap`? Belgelere göre, bu argüman `XYBitmap`, `XYPixmap` veya `ZPixmap`'ten biri olablir. Burada yaptığımız şeyin bir `XYBitmap` olduğunu düşünürseniz sizi suçlamam. Her şeyden önce bir genişliğimiz ve yüksekliğimiz var ve böyle bir şey görsem ben de biteşlem derim.

Bir `ZPixmap` seçmemizin temeline ulaşmak için, bir bitişlem ve bir `pixmap` (piksel haritası) arasındaki farkı elde etmemiz gerekiyor. X-konuşmalarında bir bitişlem, programınızda yalnızca yerel olarak kullandığınız bir görüntüdür. Bir `pixmap` ise X sunucusuna gönderilebilen bir bitişlemdir ve bu nedenle bir monitörde görüntülenebilir. Bu, bir `pixmap` istediğimiz anlamına geliyor.

Yani geriye kalan, `ZPixmap` ve `XYPixmap` arasındaki seçimdir. Bunun hakkında bilgi bulmak zor, ancak X sunucu belirtiminde konusu geçer.

> Pixmap görüntüler sunucuya iki yoldan biriyle aktarılır: `XYPixmap` veya `ZPixmap` ile. XYPixmap'ler, bağlantı yapılandırmasındaki bitmap dolgu kurallarını kullanan, görüntünün her bit düzlemi için bir tane olan bitmapler dizisidir. `ZPixmap`'ler, uygun derinlik için biçim kurallarını (dolgu vb.) kullanan, her piksel için bir tane olan bir dizi `bit`, `nibble`, `byte` veya `word`'dür.

Bu yüzden `XYPixmap`'ler birkaç bit düzlemini geçmek içindir. Bitplane'ler, X'in ilk çıktığı 1984 yılında oldukça havalı olan, ancak temel olarak artık kullanılmayan şeylerden bir diğeridir. Bu makalenin kapsamını tam olarak aşmamak için hadi gelin onları göz ardı etmeyi kabul edelim. Daha çok okuduğumuzda, `ZPixmap`'in dolambaçlı açıklamasının kullanım durumumuza daha iyi uyduğunu görüyoruz. Bizim durumumuzda bu, her piksel için 4 baytlık bir word dizisidir.

Bu büyük olanı hallettikten sonraki üç argüman bize biraz soluk aldırtıyor. Basitçe tahsis ettiğimiz hafızayı, penceremizin ve resmimizin genişliğini ve yüksekliğini iletiriz.

Bir sonraki değişken belgelerde `bitmap_pad` olarak geçiyor. Nasıl tanımlandığına bakmanızı istiyorum.

> Bir tarama çizgisinin nicemini belirtir (8, 16 veya 32). Başka bir deyişle, istemci belleğinde bir tarama çizgisinin başlangıcı, bir sonraki tarama çizgisinin başlangıcından bu bitler kadar ayrılır.

Bu, piksel boyutunu bit cinsinden tarif etmenin çok dolambaçlı bir yoludur. X belgelerini okurken bunun gibi birçok can sıkıcı açıklama göreceksiniz.

Açıklamada ayrıca tarama çizgilerinden de bahsedilmektedir. Amaçlarımız için bir `scanline`, basitçe bitmap'te bir satır veya yatay bir çizgidir. `width * pixelBytes` boyutuna sahiptir.

Son argüman `bytes_per_line`'dır. Bu argüman aslında her satırdaki bayt sayısıyla hiçbir ilgisi yoktur. Bir satırın başlangıcından bir sonraki satırın başlangıcına kadar olan uzaklıktır. Böylece, görüntünüz 3 piksel genişliğinde, 4 bayt pikselse ve her satır arasında 2 bayt boşluk varsa, bu argüman `3 * 4 + 2` olarak ayarlanır. Fakat eğer bizim durumumuzda olduğu gibi resmimizin çizgileri bitişikse burada bir `0` geçebiliriz ve `XCreateImage()` değeri kendisi hesaplayacaktır.

Oluşturduğumuz son şey grafik bağlamıdır. Bu sadece X için, çizdiğimiz asıl donanıma özgü bir bilgidir. Gerçek ekranda Xlib ile bir şeyler çizmek istediğimizde buna her zaman ihtiyacımız var.

Ardından, olay döngüsünden sonra görüntü hafızamıza bir miktar desen çizmek için bir şeyler ekleriz. Sonuçta işe yarayıp yaramadığını görmek isteriz. Bu kod, resmin tamamı üzerine siyah beyaz bir ızgara çizer.

```
// burada Event döngüsü ve ondan sonra da aşağıdakiler

int pitch = width * pixelBytes;
for(int y=0; y<height; y++)
{
```

```
            char* row = mem+(y*pitch);
            for(int x=0; x<width; x++)
            {
                unsigned int* p = (unsigned int*) (row+(x*pixelBytes));
                if(x%16 && y%16)
                {
                    *p = 0xffffffff;
                }
                else
                {
                    *p = 0;
                }
            }
        }
```

Bu bölümdeki son adım olarak, kendi hafızamızdaki görüntüyü pencereye çekmemiz gerekir. Aşağıdakileri ızgara çizim kodumuzdan hemen sonrasına ekleyin.

```
XPutImage(display, window,
          defaultGC, xWindowBuffer, 0, 0, 0, 0,
          width, height);
```

`XPutImage()`, `display`'i, pencereyi ve daha önce oluşturduğumuz `defaultGC`'yi alır. Pencereye çizdiğimizden GC'ye ihtiyacımız var. Daha sonra oluşturduğumuz `XImage`'i veriyoruz.

Sonraki dört değer, kaynak görüntünün yanı sıra hedef görüntüdeki ofsetlerdir. Bunu, pencerenin bir bölümünü görüntümüzün bir bölümünden güncellemek isteseydik kullanırdık. Ama işleri basit tuttuğumuzdan burada sadece dört sıfır var. Ardından gelenler genişlik ve yükseklik.

Bunu derleyip çalıştırırsanız beyaz bir arka planda siyah bir ızgara görürsünüz.

<- Bölüm 4 - Daha İyi Kapatma Bölüm 6 - Değişen pencere boyutuna uyum ->

### Bölüm 6 - Değişen pencere boyutuna uyum

Pencereyle oynadığınızda farkedeceğiniz şeylerden biri, pencerenin boyutunun değiştirilmesinin çok fazla bir şey yapmamasıdır. Daha büyük bir pencereden elde ettiğimiz yeni alan tamamen siyahtır ki hiç hoş değil gerçekten! Siyahlığın nedeni pencereyi oluştururken arka plan pikselini siyah olarak ayarlamamızdır.

Pencere boyutu değiştiğinde `xWindowBuffer` resmimizin boyutunu değiştirmek için pencere

boyutu değişikliğinde bize gönderilen başka bir olayı yakalamamız gerekir. Sonra görüntüyü yok edip yeni genişlik ve yükseklikle yeniden oluşturmalıyız. Tüm bu kod parçalarının birbirine nasıl uyduğunu takip etmekte sorun yaşıyorsanız depoda yer alan kaynak kodların tam haline bakabileceğinizi unutmayın.

```
// windowOpen = 1;'den ve ana döngüden önce

int sizeChange = 0;

            // switch(ev.type) içinde

            case ConfigureNotify:
            {
                XConfigureEvent* e = (XConfigureEvent*) &ev;
                width = e->width;
                height = e->height;
                sizeChange = 1;
            }
            break;

        // event döngüsünden sonra, ızgara çiziminden önce

        if(sizeChange)
        {
            sizeChange = 0;
            XDestroyImage(xWindowBuffer); // Ayırdığımızı belleği serbest bırakı
            windowBufferSize = width * height * pixelBytes;
            mem  = (char*)malloc(windowBufferSize);

            xWindowBuffer = XCreateImage(display, visinfo.visual, visinfo.depth,
                                         ZPixmap, 0, mem, width, height,
                                         pixelBits, 0);
        }
```

Öncelikle `sizeChange`'i deklare eder ve `0` olarak ayarlarız. Sonra `ConfigureNotify` olaylarını ele almak için başka bir `case` ekleriz. Bunlar, pencerenin geçerli genişliğini ve yüksekliğini içerir. Burada genişliğin ve yüksekliğin değiştiğini belirtmek için `sizeChange`'i `1` olarak ayarladık. Tek bir karede çeşitli yapılandırma olayları alabileceğimizden (ana program döngümüzün bir dönüşü) olay döngüsünden sonra biraz geçene kadar halihazırdaki resmin imhasını ve yeniden oluşturulmasını erteleriz.

`if(sizeChange)`'de görüntüyü imha ederiz, yeni genişlik ve yüksekliği kullanarak belleği tekrar ayırırız, sonra `XCreateImage`'i ilk olarak yaptığımız gibi çağırırız. `XDestroyImage()`, kullandığımız belleği serbest bırakır, eğer `malloc` kullanmaktan kaçınmak ve daha fazla pratik bellek yönetimi yapmak istiyorsak `XImage` yapısını kendimiz doldurmak zorunda kalacağız.

Tüm bunlarla artık penceremizi yeniden boyutlandırabilir ve ızgarayı pencereyi dolduracak şekilde genişletebiliriz! Ama belki şimdiden yaptığımız şeyle ilgili bir sorun tespit etmiş olabilirsiniz.

<- Bölüm 5 - Arabellek Bölüm 7 - Bu titreme hakkında... ->

### Bölüm 7 - Bu titreme hakkında...

İşleri yapma şeklimiz şimdi pencere boyutunu değiştirirken titremeye neden oluyor. Ana döngümüzü tam hızda çalıştırdığımız için bu titreme MİB'iniz yavaşsa daha belirgindir.

Olan şey şu ki, penceremizin boyutu değişir değişmez tüm pencerenin rengi siyah (pencere arka planı piksel rengi) olur. Bir hata ayıklayıcıda çalıştırılsa ve `ConfigureNotify` olayına ulaşır ulaşmaz durdursak bile, pencere halihazırda siyahtır. Burada, X sisteminin tutarsız doğası biraz sıkıntı vericidir. X iç pencere mantığının yürütülmesi ve kodumuz eşleşmemiştir.

Bu sorun için tam bir çözüm olmasa da bu eğitimin kapsamı için yeterli olacak çözüme bakalım şimdi.

Suçlu, penceredeki `gravity` özniteliğidir. Öntanımlı olarak bu, X'e yeniden boyutlandırmada pencere içeriğini atmasını söyleyen `ForgetGravity` değerine ayarlanır. Bunun olmasını istemiyoruz, bu yüzden `StaticGravity`'e ayarlamalıyız. Bu titremeyi durdurur.

Bu, pencereyi oluşturmadan önce belirlediğimiz pencere özniteliklerimizin artık şunun gibi görüneceği anlamına gelir:

```
XSetWindowAttributes windowAttr;
windowAttr.bit_gravity = StaticGravity;
windowAttr.background_pixel = 0;
windowAttr.colormap = XCreateColormap(display, root,
                                      visinfo.visual, AllocNone);
windowAttr.event_mask = StructureNotifyMask;
unsigned long attributeMask = CWBitGravity | CWBackPixel | CWColormap | CWEv
```

Sadece `windowAttr.bit_gravity`'i `StaticGravity` olarak ayarlamamız ve `attributeMask`'a `CWBitGravity`'i eklememiz gerekir.

<- Bölüm 6 - Değişen pencere boyutuna uyum Bölüm 8 - KeyPress ve KeyRelease->

### Bölüm 8 - KeyPress ve KeyRelease

Şimdi klavye girdilerinin nasıl idare edildiğini görelim. Bunu olayların kaynağından alırız.

`ConfigureNotify` olayı ile daha önce yaptığımız gibi, X'e bu tür bir olayı almak istediğimizi söylemek için pencere üzerinde bir bayrak ayarlamalıyız.

```
windowAttr.event_mask = StructureNotifyMask | KeyPressMask | KeyReleaseMask;
```

Bir tuşa ne zaman basıldığını ve tuşun ne zaman serbest bırakıldığını bilmek istiyoruz.

```
case KeyPress:
{
    XKeyPressedEvent* e = (XKeyPressedEvent*) &ev;

    if(e->keycode == XKeysymToKeycode(display, XK_Left))
        printf("sol ok tuşuna basıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Right))
        printf("sağ ok tuşuna basıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Up))
        printf("yukarı ok tuşuna basıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Down))
        printf("aşağı ok tuşuna basıldı\n");
}
break;
case KeyRelease:
{
    XKeyPressedEvent* e = (XKeyPressedEvent*) &ev;

    if(e->keycode == XKeysymToKeycode(display, XK_Left))
        printf("sol ok tuşu bırakıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Right))
        printf("sağ ok tuşu bırakıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Up))
        printf("yukarı ok tuşu bırakıldı\n");
    if(e->keycode == XKeysymToKeycode(display, XK_Down))
        printf("aşağı ok tuşu bırakıldı\n");
}
break;
```

Bu iki tür olayı ele almak için olay döngüsüne `case`'ler eklememiz gerekir. Bu tür tuş yönetimi çok

basittir. Gelen olayı daha önce yaptığımız gibi topladık ve kontrol etmek istediğimiz tuşlardan birine ait olup olmadığını kontrol ettik. Xlib'de burada yer alan iki farklı kavram vardır, biri `keycode`, diğeri ise `Keysym` veya tuş sembolüdür.

Bir `keycode`, klavyenizdeki gerçek tuşlara atanan bir sayıdır. Sol ok tuşunuz, sağ ok tuşunuzdan farklı bir `keycode`'a sahiptir ve her ikisi de A tuşundan farklıdır. Ancak, sadece hangi fiziksel tuşa basıldığını bilmek istemiyorsak, kullanıcının hangi sembole de bastığını bilmek istiyorsak? Tuş sembolleri bunun için var.

Bir sembolden bir keycode sorgulamak için `XKeysymToKeycode()`'u kullanıyoruz. Hangi klavye düzeniyle çalışacağını bilmesi için `display`'imizi ve ardından tanımlanmış `keysym` makrolarından birini geçeriz. Tanımlı `keysym`'lerin tam listesini `/usr/include/X11/keysymdef.h` dosyasında bulabilirsiniz.

Bunun ve sonraki örneğin terminalde çalıştığına emin olun; böylece `printf` çıktısını görebilirsiniz.

Bu, temel X UPA'sındaki temel klavye kullanımıdır. Daha sonra yapacaklarımız için daha derine inmeliyiz.

<- Bölüm 7 - Bu titreme hakkında… Bölüm 9 - XInput'tan UTF-8 Karakterler (Hayır, oyun çubuğu olanından değil) ->

### Bölüm 9 - XInput'tan UTF-8 Karakterler (Hayır, oyun çubuğu olanından değil)

Burası, işlerin başa çıkması biraz daha sinir bozucu olduğu yer oluyor. UTF-8 bu günlerde öntanımlı karakter kodlaması, iyi ki de öyle. Ancak X eskidir ve temel klavye UPA'sı da öyledir. O zamanlar UTF-8 bile mevcut değildi, bu yüzden bir noktada UTF-8 desteği için X'in temel klavye özelliklerine yönelik bir uzantı tanımlandı, ancak bu da daha çok ayarlama zahmeti getirdi. Bu eklenti `XInput` olarak adlandırılır ve bir uzantı olarak adlandırılmasına rağmen, bugünlerde herhangi bir Linux sisteminde mevcut olduğu varsayılabilir. Bu fonksiyonların çoğuna, Xlib belgelerinde yer verilmiştir, `Xutf8LookupString`, farklı standartlar kümesindendir ve bir man sayfasına sahiptir.

```
    // setSizeHint(display, window, 400, 300, 0, 0); satırından sonra

    XIM xInputMethod = XOpenIM(display, 0, 0, 0);
    if(!xInputMethod)
    {
        printf("Giriş Yöntemi açılamadı\n");
    }

    XIMStyles* styles = 0;
```

```
if(XGetIMValues(xInputMethod, XNQueryInputStyle, &styles, NULL) || !styles)
{
    printf("Giriş Biçimleri alınamadı\n");
}

XIMStyle bestMatchStyle = 0;
for(int i=0; i<styles->count_styles; i++)
{
    XIMStyle thisStyle = styles->supported_styles[i];
    if (thisStyle == (XIMPreeditNothing | XIMStatusNothing))
    {
        bestMatchStyle = thisStyle;
        break;
    }
}
XFree(styles);

if(!bestMatchStyle)
{
    printf("Eşleşen bir giriş biçimi belirlenemedi\n");
}

XIC xInputContext = XCreateIC(xInputMethod, XNInputStyle, bestMatchStyle,
                              XNClientWindow, window,
                              XNFocusWindow, window,
                              NULL);
if(!xInputContext)
{
    printf("Giriş Bağlamı oluşturulamadı\n");
}
```

Çok az kazanç için çok fazla kod. Bu, bize tamamen özelliksiz bir `XInput` ayarlaması verir ve temel klavye UPA'sının herhangi bir ayarlama yapmadan yaptığını yapar.

İlk önce `XOpenIM()` ile bir `XInput` Yöntemi açtık. Bu `display`'i alır ve sonrasında bir kaynak veritabanıyla ilgisi olan üç değer vardır. Öntanımlı olarak bunları görmezden gelebiliriz. Bu giriş yöntemiyle `XGetIMValues()`'u çağırabiliriz, `XInput` UPA'sındaki birçok fonksiyon gibi bu da bir değişken argüman sayılı fonksiyondur. `XNQueryInputStyle` makrosunu kullanarak ne tür

bir değer elde etmek istediğimizi belirtiriz, daha sonra bunu doldurmak için bir işaretçi veririz ve değişken sayılı argümanı NULL ile sonlandırırız. Şimdi girdi biçimlerinin bir listesine sahibiz. Bir giriş biçiminin ne olduğundan tam olarak emin değilim, fakat daha sonra ihtiyacımız olacağından bir tane almak zorundayız.

Şimdi tüm girdi biçimlerinin bir listesine sahip olduğumuzdan birini seçip onu kullanalım. Geri aldığımız tüm biçimlerin üzerinden geçiyoruz, eğer biçim XIMPreeditNothing ve XIMStatus-Nothing bayraklarını içeriyorsa bunu alıp çıkarıyoruz. Belgelendirme de bu konuda epey belirsiz, tek bildiğim, bu iki tanımlama ve girdi biçiminin hiçbir şekilde özel olmadığıdır.

Girdi yöntemi ve sorgulanan girdi biçimiyle bir girdi bağlamı oluşturabiliriz, bu asıl ihtiyaç duyduğumuz kısımdır. XCreateIC() başka bir değişken argüman sayılı fonksiyondur. Girdi yöntemini alır, bestMatchStyle'ımızı sağlarız ve penceremizin ne olduğunu ve buna odaklanması gerektiğini söyleriz (odaktayken olayları ona gönderir). Varargs, NULL ile sonlandırılır.

Şimdi neden girdi bağlamına ihtiyacımız olduğuna bakalım.

```
// KeyPress: case'i içinde
// XKeyPressedEvent* e = (XKeyPressedEvent*) &ev; satırından

int symbol = 0;
Status status = 0;
Xutf8LookupString(xInputContext, e, (char*)&symbol,
                  4, 0, &status);

if(status == XBufferOverflow)
{
    // 24 bitden büyük utf-8 karakterleri olmadığından olmam
    // Ancak bir karakter katarı tamponuna doğrudan yazmak i
    printf("Klavye sembolü eşlemesi oluşturmaya çalışırken a
}
else if(status == XLookupChars)
{
    printf("%s\n", (char*)&symbol);
}
```

Xutf8LookupString() fonksiyonu bir girdi bağlamı, bir XKeyPressedEvent, döndürülen UTF-8 karakteri için bir arabellek ve arabellek boyutu alır, bir işaretçi içinde karşılık gelen KeySym'i ve son bir argüman olarak aldığı Status işaretçisini döndürebilir.

Bundan sonra olanlar yine çok basit. status'u kontrol ediyoruz ve eğer bir karakterse yazdırıyoruz.

<- Bölüm 8 - KeyPress ve KeyRelease Sonuç ->

## Sonuç

Xlib ve ilgili teknolojiler aracılığıyla yaptığımız bu küçük turu tamamlıyoruz. Bu noktada, zemine iyi bir temel setiyle basabilmeli ve pencereleme ekosisteminin daha da belirsiz özelliklerinde ve ayarlarında da yolunuzu bulabilmelisiniz.

Gördüğünüz gibi, bu sistemde birçok farklı katman var. Bazıları uzantıların yerini aldığı eski özelliklere sahip eski fonksiyonlar, bazı özellikler sadece bir pencere yöneticisiyle etkileşime girerek kullanılabilir (ortada böyle bir pencere yöneticisi bulunmayabilir).

Tüm bu nedenler yüzünden insanlar bu teknolojiyle etkileşime girmek konusunda isteksizdirler. Belgeler birçok belgeye yayılmıştır (ve genellikle bulunması zordur), ancak biraz zaman harcayarak da olsa Xlib ile bir şeyler yapabileceğinizi görebilirsiniz!

A tour through Xlib and related technologies burada son buluyor. Bununla birlikte henüz öğrenmem gereken her şeyi öğrendiğimi düşünmediğim için çeşitli farklı kaynaklardan da yararlanarak bu rehbere ekleme yapmaya devam edeceğim. Siz de belgenin yazılmış kısımlarıyla ilgili iyileştirme isteklerinde bulunabilir veya yeni bölümler eklenmesine yardımcı olabilirsiniz.

### Bundan Sonrası

Aşağıdaki kaynaklar uzun vadede X ile ilgili okuma listemde olan kaynaklar. Bunlarla bu rehberi zenginleştiremesem bile en azından kaynak olarak burada bulunsun.

- Xlib tutorial
- Basic Graphics Programming With The Xlib Library
- How to create semi transparent white window in XLib
- Xlib - Wikipedia
- Xlib - C Language X Interface
- Xlib Programming in C and Linux Tutorial
- Moving a window programmatically with C and Xlib, the right way
- X uygulama tanıtımları
- Replace Xlib by XCB
- The Hello Wayland Tutorial
- The Wayland Protocol
- A hello world Wayland client, 2018 edition
- The X New Developer's Guide: Xlib and XCB
- Porting legacy X11/GL applications to Wayland

- Porting Qt applications to Wayland
- x11tutorial
- xecho
- Tiny-X11-Trans-Flag
- X11-CW
- xgraphsin
- xcolor
- Tutorial on creating an X11 OpenGL window
- xlines
- A port of the X11 application to Mac OS X as a screensaver
- xlib2metal