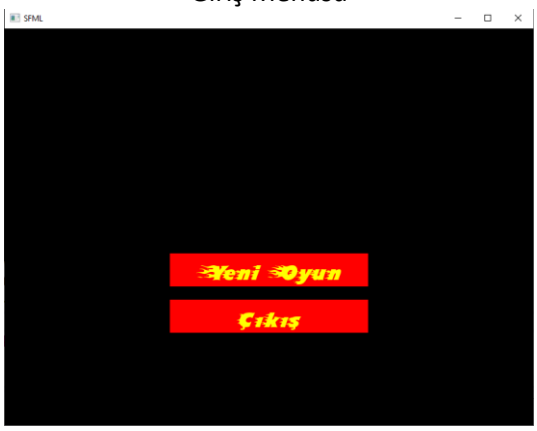
# HAFTA 3: İlk Oyun – Yılan Tasarımı

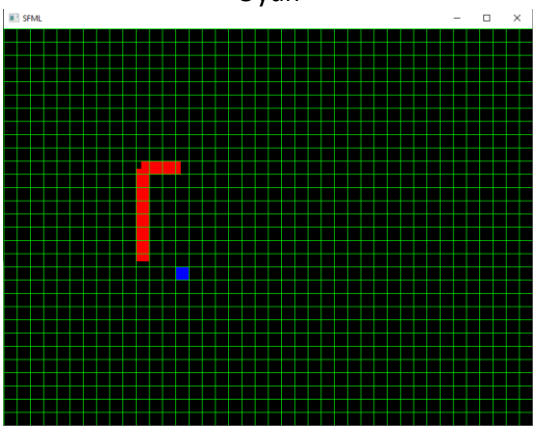
## Yılan Oyunu

Geliştireceğimiz ilk oyun bir yılan oyunu olacaktır. Oyunu tasarlamaya başlamadan önce ufak bir plan yapmamız gerekir.  İlk olarak program çalıştırıldığında aşağıda şekilde de görüldüğü gibi kullanıcının karşısına bir giriş menüsünün çıkmasını istiyoruz. Başlangıçta amacımız bu giriş menüsünü tasarlamak ve çalışır hale getirmek olacaktır.

Giriş Menüsü




Oyun



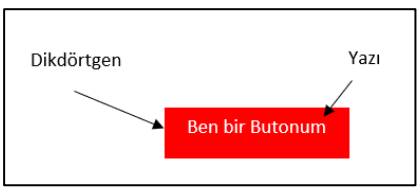

Şekil 3.1

## Giriş Menüsü

Sfml kütüphanesi, içerisinde bir ara yüz kütüphanesi barındırmamaktadır. Bu yüzden butonları kendi başımıza tasarlamamız gerekmektedir. Butonlar sadece üzerlerinde yazı olan şekillerden ibaret değildir. Farenin üzerinde olup olmadığını anlayabilmeli ayrıca fare tıklandığında istenilen işlemleri gerçekleştirebilmelidir. Bütün bunları göz önünde bulundurarak bir Buton sınıfı tasarlamak çok daha mantıklı olacaktır.

## Buton Sınıfı

Bir buton, aşağıdaki gibi dikdörtgen bir şekil ve yazıdan ibaret olacaktır. Bu bilgilerden yola çıkarak geliştireceğimiz buton nesnelerinin iki tane çizilecek şekli olacağı sonucu çıkartabiliriz. Buton sınıfımızın ilk prototipi kod 3.1.'deki gibi olacaktır




Şekil 3.2

```
01  #pragma once
02  #include<SFML/Graphics.hpp>
03  class Buton
04  {
05  public:
06        Buton();
07        void yazi(sf::String yeniYazi);
08        void renk(sf::Color renk);
09        void font(sf::Font& yeniFont);
10        void ciz(sf::RenderWindow& pencere);
11        void boyut(float genislik, float yukseklik);
12        void konum(float x, float y);
13  private:
14        sf::RectangleShape  m_arkaPlan;
15        sf::Text            m_yazi;
16  };
```

Kod 3.1.

Kodun 14.satırında butonun arka planını temsil edecek olan dikdörtgen nesnesi tanımlanmıştır. Hemen altında 15.satırda buton üzerinde görünecek olan yazı için **`sf::Text`** türünden bir nesne tanımlanmıştır. . **`sf::Text`** nesneleri Sfml tarafından çizilebilen nesnelerdir. Bu nesnenin kullanımı ilerleyen sayfalarda detaylı bir şekilde gösterilecektir. Buton sınıfına ait metotların işlevleri isimlerinden anlaşılmaktadır. Bu fonksiyonların gövdeleri Kod.3.2' de gösterilmektedir. **`font`** fonksiyonu çizdirilecek olan yazının fontunu atamaktadır. Yazıyı çizdirmek için bu işlemin yapılması şarttır.

```
01  #include "Buton.hpp"
02  Buton::Buton()
03  {
04  }
05  void Buton::yazi(sf::String yeniYazi)
06  {
07        m_yazi.setString(yeniYazi);
08  }
09  void Buton::renk(sf::Color renk)
10  {
11        m_arkaPlan.setFillColor(renk);
12  }
13  void Buton::font(sf::Font& yeniFont)
14  {
15        m_yazi.setFont(yeniFont);
16  }
17  void Buton::ciz(sf::RenderWindow& pencere)
18  {
19        pencere.draw(m_arkaPlan);
20        pencere.draw(m_yazi);
21  }
22  void Buton::boyut(float genislik, float yukseklik)
23  {
24        m_arkaPlan.setSize(sf::Vector2f(genislik,yukseklik));
25  }
26  void Buton::konum(float x, float y)
27  {
28        m_arkaPlan.setPosition(sf::Vector2f(x, y));
29  }
```

Kod 3.2.

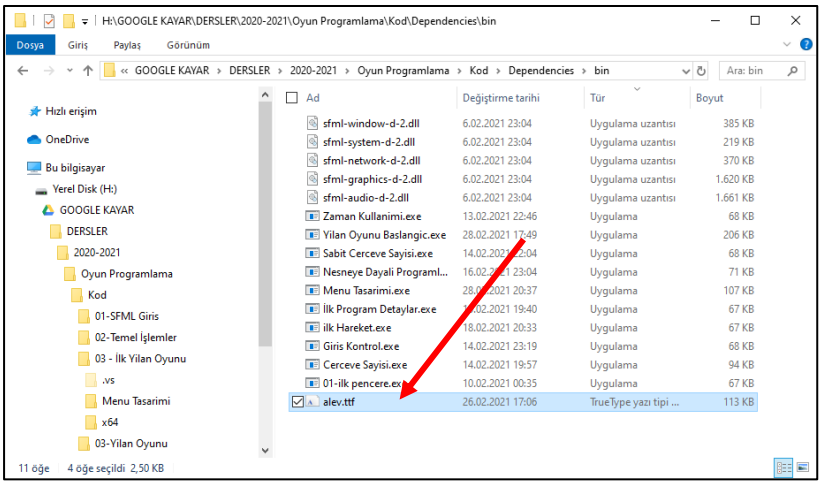
İlk denemelerimizi yapmak için **Oyun.cpp** içerisinde aşağıdaki gibi bir değişiklik yapacağız (Kod 3.3). Kod içerisinde **Oyun** sınıfının değiştirilen fonksiyonları gösterilmektedir. Diğer fonksiyonları bir önceki dersteki gibi kalacaktır. Kodun içerisine eklenen yeni satırlar kalın fontla belirtilmiştir. Deneme yaptığımız için öncelikle 2.satırda bir buton nesnesini sınıf dışında global olarak oluşturuyoruz. Hemen ardından yüklenecek olan bir font nesnesi oluşturuyoruz. **Oyun** sınıfının kurucu fonksiyonuna ilk olarak 12.satırdaki font yükleme satırını ekliyoruz. **Font** dosyalarını https://www.yazitipleri.net benzeri sitelerden indirebilirsiniz. Bu işlem doğru şekilde gerçekleşirse yüklenilen **Font** nesnesi **Buton** nesnemizin font atayan fonksiyonuna verilmektedir. 16.satırda butonumuzun boyutu belirlenirken 17.satırda buton üzerinde çıkacak yazı belirtilmektedir.

**sahneCiz** fonksiyonu içerisine sadece 24.satır eklenmiştir. **Buton** nesnemizin kendisini çizmesi mesajı gönderilmektedir. (Bir nesnenin metodunu çağırmaya nesneye mesaj göndermede denmektedir). Projemizi bu hali ile çalıştırırsak font yüklemede hata alacağız.

```
01 #include "Oyun.hpp"
02 Buton b;
03 sf::Font font;
04 Oyun::Oyun()
05 {
06       m_sekil.setRadius(30);
07       m_sekil.setFillColor(sf::Color::Green);
08       m_artis = 1.0f;
09       m_konumX = 10.0f;
10       m_cerceveSuresi = 1.0f / 60.0f;
11
12       if (font.loadFromFile("alev.ttf") != false)
13       {
14            b.font(font);
15       }
16       b.boyut(100, 200);
17       b.yazi("Merhabaa");
18       b.renk(sf::Color::Red);
19 }
20 void Oyun::sahneCiz()
21 {
22       m_pencere.cizimeBasla();
23       m_pencere.ciz(m_sekil);
24       b.ciz(m_pencere.pencereGetir());
25       m_pencere.cizimiBitir();
26 }
```

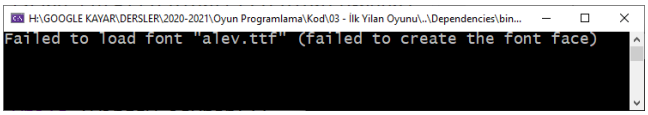
Kod 3.3.

Peki font dosyasını hangi klasöre yükleyeceğiz. Bütün projelerimizde kullanacağımızı düşünürsek font dosyasını çalıştırılabilir dosyanın bulunduğu klasöre yüklemek daha mantıklı olacaktır. Font dosyasını **"Dependencies\bin\"** klasörüne şekil 3.3'deki kopyalayıp projemizi çalıştırıyoruz.

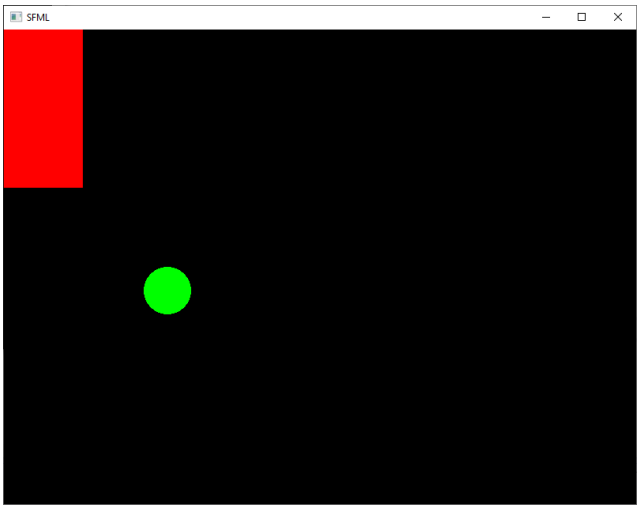



Şekil 3.3

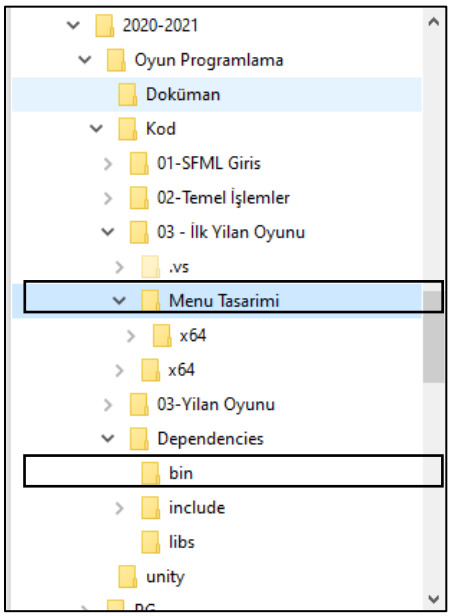
Projeyi derleyip çalıştırdığımızda ekrana bir yazı çıkmadığını göreceğiz konsol ekranında ise aşağıdaki gibi bir hata alacağız(Şekil 3.4). Bu hatanın sebebi font dosyasının bulunamamasıdır. Eğer **"Dependencies\bin"** klasörü içerisine girip programımızın çalıştırılabilir dosyasını ("Menu Tasarimi.exe") çalıştırırsak şekil 3.5'deki gibi hatasız bir sonuç alırız. Problem, çalıştırılabilir dosyamız **"Dependencies\bin"** klasöründe olmasına rağmen bu dosyayı çalıştırma işlemini projemizin bulunduğu klasör içerisinden yapılmasıdır. Dolayısıyla font dosyası ile aynı klasörde bulunmuyoruz. Şekil 3.6'da bu durum gösterilmiştir. **Visual Studio** derleme ve çalıştırma işlemlerinde çalıştırılabilir dosyayı "**working directory**" adını verdiği klasörden çağırmaktadır. Bu klasör başlangıçta proje dosyamızın bulunduğu klasördür.




Şekil 3.4

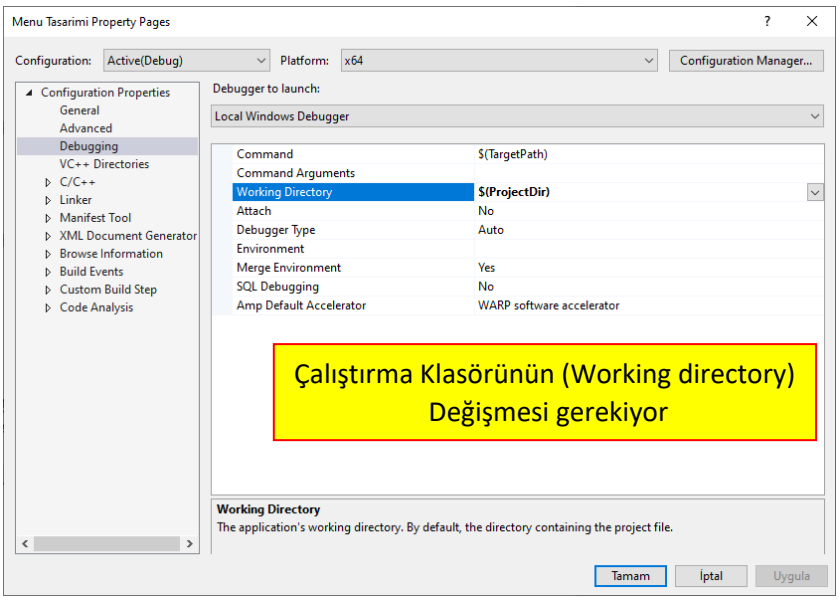



Şekil 3.5




Çalıştırma işleminin gerçekleştiği klasör

Çalıştırılabilir dosyanın ve fontun bulunduğu klasör




Şekil 3.6

İstenirse çalıştırma işlemini de "**`Dependencies\bin`**" klasörü içerisinde yapılabilir. Bunun için proje ayarlarımızda değişiklik yapmamız gerekmektedir. Proje ayarlarından "**`debugging`**" seçeneğini tıklayıp "**`working directory`**" ayarını değiştirmemiz gerekiyor. Buradaki değeri, çalıştırılabilir dosyanın bulunduğu klasör ile aynı yapacağız yani **"`$(SolutionDir)..\Dependencies\bin\`"** değerini atayacağız. Değişiklikleri uygulayıp ayarları aktif hale getirmeyi unutmayın. Programı derlediğimizde bu sefer hem doğru derleyecek hem de sorunsuz çalıştıracaktır. Şekil 3.7'de programın çalışır hali görünmektedir.


Şekil 3.7

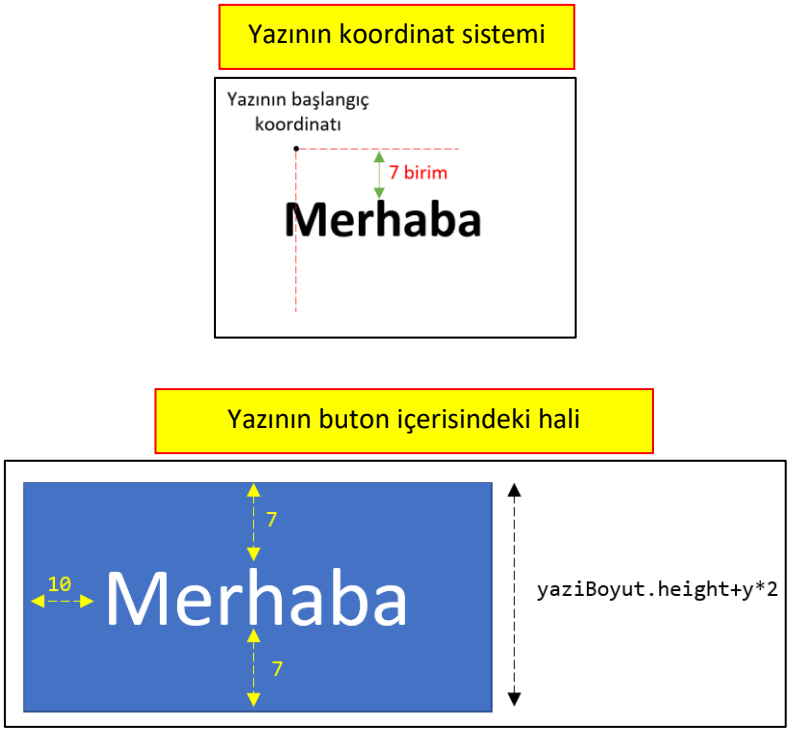
# Yazı Hizalama

Şekil 3.7'deki görüntüde yazının butonun arka planına sığmadığı görülmektedir. Sfml bizim yerimize hizalama yapmayacaktır. Bu işlemi programcı olarak bizim yapmamız gerekmektedir. Bunun için en uygun yer yazının atandığı **yazi** fonksiyonu olacaktır. Bu fonksiyon çağrıldığında butonun yazısı değişecektir dolayısıyla boyutu da buna göre değişmelidir. Kod 3.4.'de **yazi** fonksiyonun değiştirilmiş hali bulunmaktadır. 5.satırda girilmiş olan yazının boyutları alınmaktadır. **getLocalBounds** fonksiyonu yazının boyutlarını getirmenin yanı sıra sol ve üst boşluk miktarlarını da getirmektedir. Burada **auto** kelimesi **c++ 11** de getirilen bir yeniliktir. **auto** ifadesi **yaziBoyut** değişkeninin türünü **getLocalBounds** fonksiyonunun dönüş türü ile aynı yapmaktadır. 7 ve 8. satırlarda sırasıyla yazının sol ve üst boşlukları **x** ve **y** adındaki değişkenlere atanmaktadır (sadece kolaylık için). Yazı tipine göre sol boşluk negatif çıkabilmektedir. O yüzden **abs** fonksiyonu kullanıyoruz. 10. Satırda yazının genişliği alınıyor ve bu değere sol boşluğun iki katı ve 20 değeri ekleniyor (20 değeri başka bir değişkene bağlanabilir çok önemli değil). Amacımız butonun boyutunu yazıdan biraz büyük yapmaktır. Aynı şekilde 11.satırda yazının yüksekliği getiriliyor ve bu değere yazının üst kısımdaki boşluğunun iki katı eklenip sonuç yeni bir değişkene atanmaktadır. Yeni elde edilen **genislik** ve **yukseklik** değerleri **boyut** fonksiyonu aracılığıyla butonun arka planının genişlik ve yüksekliği olarak atanmaktadır. Son olarak butonun arka planının x koordinatına 10 eklenerek elde edilen sonuç yazının x koordinatına atanmaktadır.

```
01  void Buton::yazi(sf::String yeniYazi)
02  {
03        m_yazi.setString(yeniYazi);
04
05        auto yaziBoyut = m_yazi.getLocalBounds();
06
07        float x = abs(yaziBoyut.left;
08        float y = yaziBoyut.top;
09
10        float genislik = yaziBoyut.width + x*2+ 20;
11        float yukseklik = yaziBoyut.height + y *2;
12
13        boyut(genislik, yukseklik);
14
15        m_yazi.setPosition(m_arkaPlan.getPosition().x+ x + 10,0);
16  }
```

Kod 3.4.

Peki neden bu kadar işleme gerek duyuldu. Öncelikle yazının sahip olduğu boşluklardan bahsedelim. Yazının x ve y koordinatları hem ilk karakterin türüne hem de yazı fontuna bağlı olarak değişmektedir. **yaziBoyut** nesnesinin x ve y koordinatları girilen yazının solda ve üstte ne kadar boşluğa sahip olduğunu belirtmektedir. Örneğin "Merhaba" yazısını girdiğimizde **yaziBoyut** nesnesinin x ve y değerleri sırasıyla -1 ve 7 olmaktadır. Şekil 3.8'de bu durum gösterilmektedir. Buna göre amacımız yazının butonun arka planını temsil eden dörtgenin ortasına yerleştirmektir. Mükemmel olmak zorunda değil ama gözle görülür bir bozulmaya izin vermemeliyiz. **yazi** metoduna şimdi tekrardan bakarsanız içerisindeki hesaplamalar daha mantıklı gelecektir. Şekil 3.7'de gösterilen yazının global koordinatı olarak adlandırmaktadır. Yani yazıya pozisyon atadığımızda bu değeri değiştiriyoruz. Yazı da kendi içerisinde yerel değişkenlerini kullanarak global konuma göre kendisini ayarlıyor. Yazdığımız kod ile bizlerde bu problemin üstesinden geliyoruz. Şekil 3.8.'de elde ettiğimiz sonuç gösterilmektedir.




Şekil 3.8

Butonun farklı konumlara da yerleştirileceği düşünülürse **konum** metodunun da güncellenmesi gerekmektedir. Buton konum değiştirdiğinde yazının da konumunu adapte etmesi gerekir. Kod 3.5'de **konum** fonksiyonunun güncellenmiş hali verilmiştir.

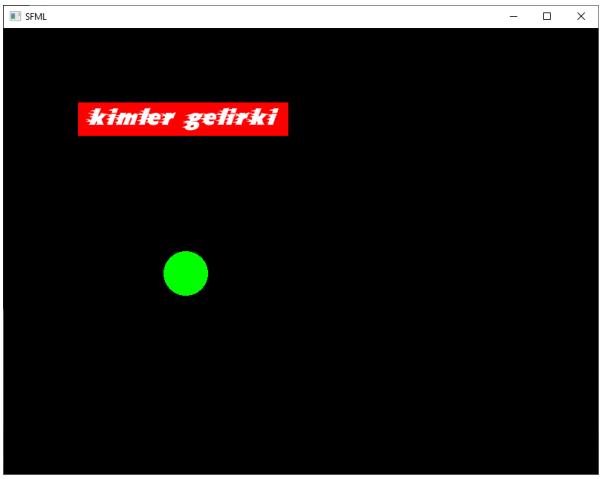
```
01  void Buton::konum(float x, float y)
02  {
03        auto yaziBoyut = m_yazi.getLocalBounds();
04        float yazix = abs(yaziBoyut.left);
05
06        float yaziy = yaziBoyut.top;
07
08        m_yazi.setPosition(x + yazix + 10, y);
09
10        m_arkaPlan.setPosition(sf::Vector2f(x, y));
11  }
```

Kod 3.5.

Oyun sınıfının kurucu fonksiyonunda aşağıdaki gibi değişiklik yaparsak (17.satır eklendi) programın farklı sonuç verdiğini görürüz (Şekil 3.9).

```
01  Oyun::Oyun()
02  {
03          m_sekil.setRadius(30);
04          m_sekil.setFillColor(sf::Color::Green);
05          m_artis = 1.0f;
06          m_konumX = 10.0f;
07          m_cerceveSuresi = 1.0f / 60.0f;
08
09          if (font.loadFromFile("alev.ttf") != false)
10          {
11                  b.font(font);
12          }
13          b.boyut(100, 200);
14          b.yazi("kimler gelirki");
15          b.renk(sf::Color::Red);
16
17          b.konum(100, 100);
18  }
```

Kod 3.6.




Şekil 3.9

# Fare Kontrolü

Butonun çalışabilmesi için farenin üzerinde tıklandığını algılaması gerekmektedir. Ayrıca fare üzerine geldiğinde kullanıcıya aktif olduğunu belirtecek bir işlemde yapabilmelidir. Bunun için fare hareketlerinin buton tarafından yakalanması gerekmektedir. Fare bilgilerini **Pencere** sınıfımızda yakalayabiliriz. Windows benzeri işletim sistemlerinde pencere üzerinde gerçekleşen olaylar pencere nesnesine gönderilmektedir. **Pencere** sınıfının **olayKontrol** metodu pencere üzerinde gerçekleşen olayları yakalamaktadır. Olay döngüsü içerisinde yakalanan olayın fareden gelip gelmediğini Kod 3.7'deki gibi kontrol edebiliriz. 9.satırda olay türünün **MouseMoved** olup olmadığını yani farenin hareket edip etmediğini kontrol ediyoruz. Koşul içerisinde de farenin koordinatlarını komut satırı penceresinde ekrana çıkartıyoruz. Programı denerken **Pencere.cpp** dosyasında <**iostream**> kütüphanesini eklemeyi unutmayın.

```
01  void Pencere::olayKontrol()
02  {
03          sf::Event event;
04          while (m_window.pollEvent(event))
05          {
06                  if (event.type == sf::Event::Closed)
07                          m_window.close();
08
09                  if (event.type == sf::Event::MouseMoved)
10                  {
11                          std::cout << event.mouseMove.x << ","
12                                  << event.mouseMove.y << std::endl;
13                  }
14          }
15  }
```

Kod 3.7.

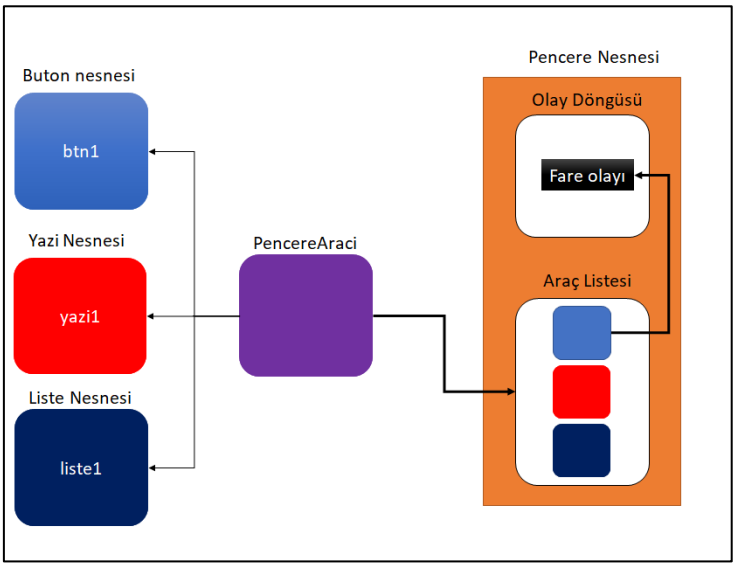
**Pencere** sınıfı içerisinde fare hareketlerini yakalamamız **Buton** nesneleri için hiçbir anlam ifade etmiyor. Bir şekilde bu iki sınıf arasında ilişki kurmamız gerekiyor. İlişki kurarken sınıflar arası bağımlılığı en az seviyede tutmamız gerekiyor. Pencere üzerindeki bütün araçlar (textbox,buton vs) fare hareketlerinden haberdar olmak isteyeceğini düşünürsek ortak bir ara yüz sınıfı tasarlamak daha mantıklı olacaktır. Ara yüz sınıfımız Kod 3.8'de gösterilmiştir. C++ dilinde soyut fonksiyon yazmak için **virtual** anahtar kelimesi kullanılmalıdır. Ayrıca fonksiyon tanımının sonunda "=0" ifadesi eklenmesi gerekmektedir. Sınıfımızın bütün fonksiyonları soyut olduğu için **PencereAraci** bir arayüz (Interface) olarak kabul edilmektedir.

```
01  #pragma once
02  #include<SFML/Window/Event.hpp>
03  class PencereAraci
04  {
05  public:
06     virtual void fareHareket(int x, int y) = 0;;
07     virtual void fareButonBasildi(sf::Event::MouseButtonEvent btnOlay)= 0;
08     virtual void fareButonBirakildi(sf::Event::MouseButtonEvent btnOlay)=0;
09     virtual bool icerdemi(int x, int y)=0;
10  private:
11
12  };
```

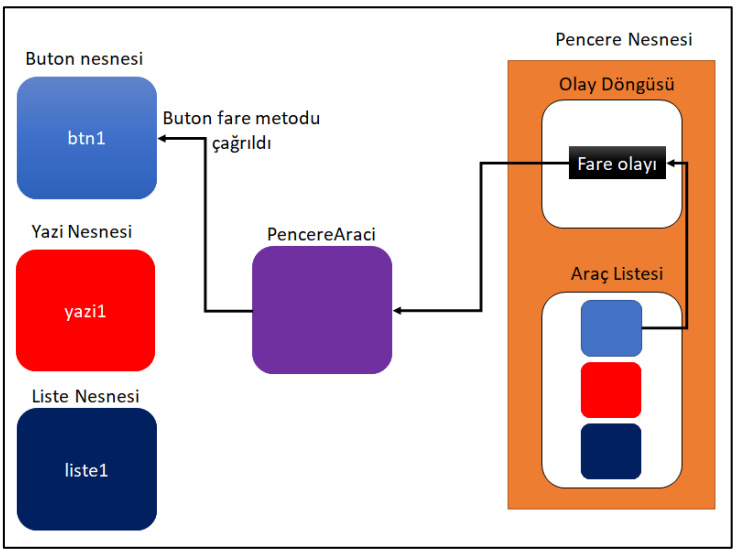
Kod 3.8.

Pencere üzerinde bulunacak olan bütün araçların ortak bir ara yüzden kalıtım alması Pencere sınıfının işini kolaylaştıracaktır. Bu sayede Pencere sınıfının üzerindeki araçların türünü bilmesine gerek kalmayacak ve bir olay gerçekleştiğinde çağrılacak fonksiyonun ismi ve argümanları değişmeyecektir. Pencere sınıfı tanımladığımız ara yüzün fonksiyonlarını çağıracaktır. Fonksiyonların gövdelerini ise ara yüzden kalıtım alan programcılar yazacaktır (Yani biz). Böylece bizlerin yazdığı hareket fonksiyonları **Pencere** sınıfı tarafından çağrılmış olacaktır. Şekil 3.10'da bu işlem şematik olarak gösterilmektedir. Burada pencere üzerindeki araçların ara yüzlerinin adreslerini tutacak bir liste görünmektedir (Araç Listesi). Bu liste PencereAraci adresi türünde verileri tutacaktır. Yani Pencere için üzerindeki araçların hepsi aynı türden görünmektedir (PencereAraci olarak). Pencere olay döngüsünde de bu listedeki bütün ara yüzlere ait ilgili metotlar tek tek çağrılacaktır. Şekil 3.11'de fare olayına karşılık sıradaki ara yüzün fare fonksiyonunun (Örneğin hareket fonksiyonu) çağrıldığı görünmektedir. Sıradaki ara yüz bir

buton nesnesine ait olduğundan fare fonksiyonu olarak buton nesnesine ait olan fare fonksiyonu çağrılacaktır. Aynı durum yazi ve liste nesneleri içinde geçerlidir(yazi ve liste sınıflarını henüz tanımlamadık). Bu sayede ilerde farklı pencere araçları oluşturmak istersek tek yapmamız gereken bu ara yüzden kalıtım alıp fonksiyon gövdelerini yazmak olacaktır.




Şekil 3.10




Şekil 3.11

**`Pencere`** sınıfının yeni hali Kod 3.9'da görülmektedir. Sınıf fare bilgisini üzerindeki araçlara gönderebilmesi için araçların ara yüzlerine bir şekilde erişmesi gerekiyor. Bu yüzden Pencere sınıfı içirişinde **`PencereAraci`** adresleri tutan bir **`vector`** nesnesi oluşturuldu. **`vector`** sınıfı c++ dilinin standart dinamik dizisi olarak düşünülebilir. Ayrıca araç listesine eleman eklemek için **`aracEkle`** isimli bir metotta eklendi (20.satır).

Kodun 4. satırında bir ön tanımlama yapılmaktadır. Böylece .**`hpp`** dosyasında **`PencereAraci`** dosyasını eklemekten kurtuluyoruz. Ön tanımlama sadece ilgili sınıfa ait işaretçi kullanıldığında işe

yaramaktadır. Sınıfa ait herhangi bir üyeye erişmek istersek **PencereAraci.hpp** dosyasını eklemek durumunda kalacağız. **Pencere.cpp** dosyası içerisinde zaten bu işlem mecburen yapılacaktır.

```
01  #pragma once
02  #include<SFML/Graphics.hpp>
03  #include<vector>
04  class PencereAraci;
05
06  class Pencere
07  {
08  public:
09        Pencere();
10        void ayarla(unsigned int genislik,
11                    unsigned int yukseklik,
12                    sf::String baslik);
13        void cizimeBasla();
14        void cizimiBitir();
15        bool kapandimi();
16        void olayKontrol();
17        void ciz(sf::Drawable& sekil);
18        sf::RenderWindow& pencereGetir();
19
20        void aracEkle(PencereAraci* arac);
21  private:
22        sf::RenderWindow m_window;
23
24        std::vector<PencereAraci*> m_pencereAraclari;
25  };
```

Kod 3.9.

**aracEkle** fonksiyonuna Kod 3.10'da görüldüğü gibi tek satırlık kod yeterli olmaktadır. **push_back** fonksiyonu ilgili vektöre yeni bir eleman eklemektedir. C++ dilinde referans türündeki değişkenlerin değerleri ilk atamadan sonra değiştirilememektedir. **vector** sınıfı bu yüzden referans olarak değer tutamaz. İçindeki değerleri değiştirilebilmesi gerekir. Bu yüzden nesnelerin adresi tutulmaktadır.

Pencere sınıfının **olayKontrol** fonksiyonu da yine kod 3.10'da görülmektedir. 12.satırda okunan olayın fare hareket olayı olduğu keşfedilmektedir. Farenin koordinatları alındıktan sonra pencere içerisine eklenmiş bütün araçları tek tek gezen bir döngü oluşturuluyor. Döngü içerisinde her araca fare koordinatlarının içinde olup olmadığı soruluyor. Eğer araç **true** olarak cevap dönerse aracın **fareHareket** metodu çağrılarak fare koordinatları argüman olarak veriliyor. **icindemi** ve **fareHareket** fonksiyonlarının gövdeleri **PencereAraci** ara yüzünden kalıtım alan sınıf tarafından yazılması gerekir. **Pencere** sınıfı ne tür bir araca sahip olduğunu bilmiyor tek bildiği **PencereAraci** ara yüzünden adreslere sahip olduğu ve bu ara yüz içerisinde **icindemi** ve **fareHareket** fonksiyonlarının bulunduğudur.

```
01  void Pencere::aracEkle(PencereAraci* arac)
02  {
03        m_pencereAraclari.push_back(arac);
04  }
05  void Pencere::olayKontrol()
06  {
07        sf::Event event;
08        while (m_window.pollEvent(event))
09        {
10              if (event.type == sf::Event::Closed)
11                    m_window.close();
12              if (event.type == sf::Event::MouseMoved)
13              {
14                    int x = event.mouseMove.x;
15                    int y = event.mouseMove.y;
16                    for (auto siradaki : m_pencereAraclari)
17                    {
18                          if (siradaki->icerdemi(x, y))
19                          {
20                                siradaki->fareHareket(x, y);
21                          }
22                    }
23              }
24        }
25  }
```

Kod 3.10.

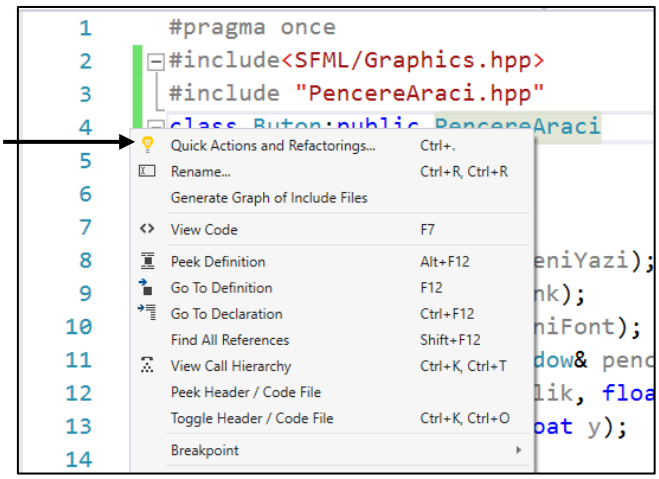
Şu an için fare hareket sistemimiz hazır yalnız bir deneme yapabilmemiz için öncelikle **Buton** sınıfının **PencereAraci** ara yüzünden kalıtım alması ve soyut fonksiyonlarının gövdesi yazılması gerekiyor. Kalıtım için **Button.hpp** dosyasında öncelikle **PencereAraci.hpp** dosyasını eklememiz gerekiyor (Kalıtım alırken ön tanımlama yeterli olmaz). Daha sonra aşağıdaki kod parçasının 4. satırında olduğu gibi kalıtım alma isteği belirtilir.

```
01  #pragma once
02  #include<SFML/Graphics.hpp>
03  #include "PencereAraci.hpp"
04  class Buton:public PencereAraci
05  {
06  public:
07  ……
08  ……
09  ……
10
11  };
```

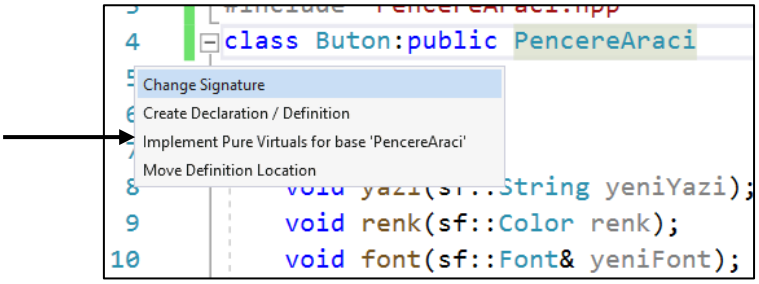
Kod 3.11.

Projemiz bu durumuyla derlenmeyecektir. **Buton** sınıfı kalıtım aldığı ara yüzün bütün soyut fonksiyonlarının gövdelerini yazması gerekir. Bu işlemi el ile tek tek de yapabiliriz veya **visual studio'nun** bize sağladığı "**quick action**" aracıyla da gerçekleştirebiliriz. Bunun için **Buton.hpp** içerisindeyken fareyi **PencereAraci** sınıfı üzerine getirip farenin sağ tuşuna basmamız gerekmektedir. Karşımıza aşağıdaki Şekil 3.12'deki panel çıkacaktır . Buradan ilk seçeneği seçeceğiz. Ardından karşımıza şekil 3.13'deki panel çıkacaktır buradan da 3.seçeneği seçerek fonksiyon gövdelerinin otomatik olarak oluşturulmasını sağlayacağız. Bu işlem sırasında gövdelerin yazılmasının

yanı sıra **Buton** sınıfının prototipine de fonksiyon isimleri eklenecektir. Kod 3.12'de **Buton** sınıfına eklenen fonksiyon prototipleri görülmektedir. C++ programlama dili sınıfları tanımlarken, tanımı iki parçaya ayırmaktadır. O yüzden fonksiyon ezerken de iki parçada güncellenmesi gerekiyor (override C++11 ile gelen bir anahtar kelimedir).




Şekil 3.12




Şekil 3.13

```
01  class Buton:public PencereAraci
02  {
03     //Buton sınıfı üst kısmını yazmadık
04     ……..
05     virtual void fareHareket(int x, int y) = 0;;
06     virtual void fareButonBasildi(sf::Event::MouseButtonEvent btnOlay)= 0;
07     virtual void fareButonBirakildi(sf::Event::MouseButtonEvent btnOlay)=0;
08     virtual bool icerdemi(int x, int y)=0;
09  };
```

Kod 3.12.

Ara yüz fonksiyonlarından ilk olarak **icerdemi** fonksiyonunun gövdesini yazacağız. Bu fonksiyon kendisine verilen koordinatların butonun içerisinde olup olmadığını kontrol edecek. Butonumuz bir dikdörtgen olduğu düşünülürse bu işlemi aşağıdaki kodda olduğu gibi halledebiliriz. İlk olarak butonumuzun sınırlarını temsil eden dikdörtgen şeklinin konum ve boyutunu bir değişken

içerisine yerleştiriyoruz. Amacımız koşulumuzu daha kolay yazabilmektedir. Ardından koşulumuzu yazıyoruz. Koşulumuz 4 durumun aynı anda doğru olmasına bağlıdır. Bu durumda **x,y** koordinatları dikdörtgenin içerisindedir ve fonksiyonumuz **true** değeri döndürür. Aksi halde koşul içerisine girilmez ve fonksiyonumuz otomatik olarak **false** değeri döndürecektir. **fareHareket** fonksiyonunun da çalıştığını göstermek için gövdesini aşağıdaki gibi koordinatları ekrana çıkartacak şekilde yazıyoruz. **Buton.cpp** içerisinde <**iostream**> kütüphanesini eklemeyi unutmayın.

```
01  bool Buton::icerdemi(int x, int y)
02  {
03          auto konum = m_arkaPlan.getPosition();
04          auto boyut = m_arkaPlan.getSize();
05
06          if (konum.x <= x &&
07              konum.x + boyut.x >= x &&
08              konum.y <= y &&
09              konum.y + boyut.y >= y)
10          {
11              return true;
12          }
13          return false;
14  }
15  void Buton::fareHareket(int x, int y)
16  {
17          std::cout << x << "," << y << std::endl;
18  }
```

Kod 3.13.

Şu an her şey hazır gibi duruyor. Tek bir sıkıntımız var. Buton nesnemiz ile Pencere nesnesi arasında herhangi bir bağ kurmadık. Projemiz içerisinde iki nesne de **Oyun** nesnesine aittir. O yüzden bağ kurma işlemi Oyun nesnesi içerisinde yapılacaktır. Bu işlem için **Oyun** sınıfının kurucu fonksiyonun son satırına aşağıdaki kod satırını eklememiz yeterli olacaktır.

```
m_pencere.aracEkle(&b);
```

Projemizi çalıştırıp fareyi butonun üzerine getirdiğimizde komut satırı ekranında koordinat değerlerinin ekrana çıktığını göreceğiz. Fare, butonun üzerinde değilken ekrana herhangi bir yazı çıkmayacaktır. Kod 3.14 içerisinde **Pencere** sınıfının **olayKontrol** metodunun içeriği verilmiştir. Fare tuşlarına basılıp bırakılma olayları da yakalanıp ilgili fonksiyonlar çağrılmaktadır.

```
01  void Pencere::olayKontrol()
02  {
03          sf::Event event;
04          while (m_window.pollEvent(event))
05          {
06                  if (event.type == sf::Event::Closed)
07                          m_window.close();
08
09                  if (event.type == sf::Event::MouseMoved)
10                  {
11                          int x = event.mouseMove.x;
12                          int y = event.mouseMove.y;
13                          for (auto siradaki : m_pencereAraclari)
14                          {
15                                  if (siradaki->icerdemi(x, y))
16                                  {
17                                          siradaki->fareHareket(x, y);
18                                  }
19                          }
20                  }
21                  if (event.type == sf::Event::MouseButtonPressed)
22                  {
23                          int x = event.mouseButton.x;
24                          int y = event.mouseButton.y;
25                          for (auto siradaki : m_pencereAraclari)
26                          {
27                                  if (siradaki->icerdemi(x, y))
28                                  {
29                                    siradaki->fareButonBasildi(event.mouseButton);
30                                  }
31                          }
32                  }
33                  if (event.type == sf::Event::MouseButtonReleased)
34                  {
35                          int x = event.mouseButton.x;
36                          int y = event.mouseButton.y;
37                          for (auto siradaki : m_pencereAraclari)
38                          {
39                             if (siradaki->icerdemi(x, y))
40                             {
41                               siradaki->fareButonBirakildi(event.mouseButton);
42                             }
43                          }
44                  }
45          }
46  }
```

Kod 3.14.

# Buton Animasyonu

Butonumuz artık fare olaylarından haberdar olmaktadır. Şimdi bu olaylara karşılık butonumuzun fiziksel olarak karşılık vermesidir. Örneğin fare buton üzerine geldiğinde butonun renginin değişmesini sağlayabiliriz. Bunun için butonumuzun **`fareHareket`** fonksiyonunda arka planın rengini Kod 3.15 deki gibi değiştirmemiz yeterli olacaktır.

```
01 | void Buton::fareHareket(int x, int y)
02 | {
03 |       m_arkaPlan.setFillColor(sf::Color::Green);
04 | }
```

Kod 3.15.

Artık fareyi buton üzerine getirdiğimizde butonumuzun arka planı yeşil olmaktadır. Fakat fareyi butonun üzerinden çıkardığımızda renk eski haline dönmemektedir. Farenin buton üzerine olduğunu kestiriyoruz fakat dışına çıktığını Buton sınıfımız algılayamamaktadır. Bu problemin üstesinden gelmek için farklı çözümler geliştirilebilir. Biz olabildiğince basit bir çözüm ile problemin üstesinden gelmek istiyoruz. Amacımız her şeyi bulunan bir pencere yönetim sistemi geliştirmek değil. Pencere araçlarının **`icerdemi`** fonksiyonu farenin içerde olup olmadığını kontrol ettiğinden bu fonksiyonun gövdesini aşağıdaki gibi yapmamız problemimizi çözecektir. Fonksiyon 13. Satıra sadece fare butonun dışındayken gelmektedir. Bu durumda butonun rengini eski haline getirecektir. Şekil 3.14'de sonuç görülmektedir.

```
01 | bool Buton::icerdemi(int x, int y)
02 | {
03 |       auto konum = m_arkaPlan.getPosition();
04 |       auto boyut = m_arkaPlan.getSize();
05 |
06 |       if (konum.x <= x &&
07 |             konum.x + boyut.x >= x &&
08 |             konum.y <= y &&
09 |             konum.y + boyut.y >= y)
10 |       {
11 |             return true;
12 |       }
13 |       m_arkaPlan.setFillColor(sf::Color::Red);
14 |
15 |       return false;
16 | }
```

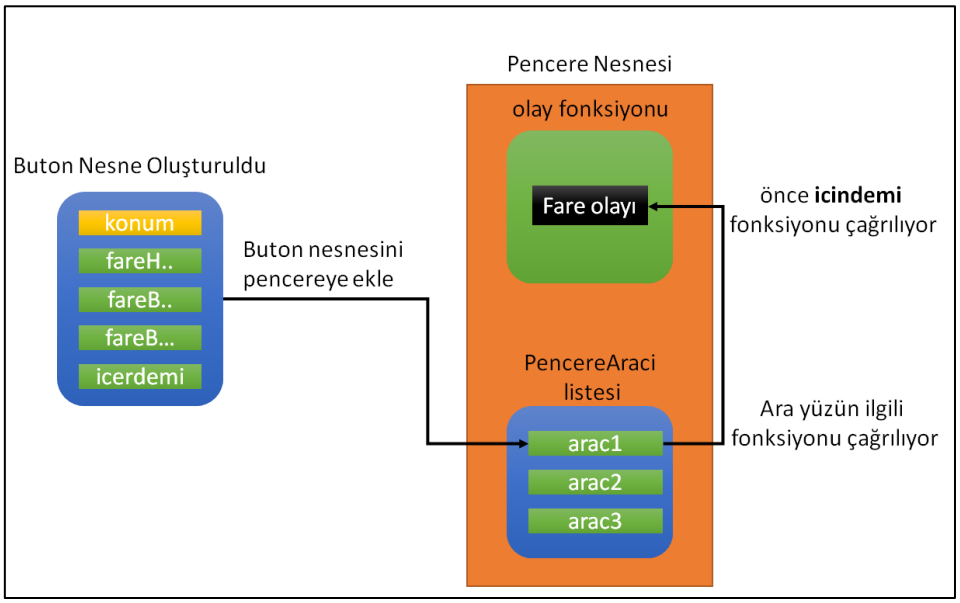
Kod 3.16.


Fare üzerinde değilken Fare üzerindeyken

Şekil 3.14

Şekil 3.15'de fare olaylarına karşılık, **`PencereAraci`** ara yüzlerine ait fonksiyonların çağrılması şematik olarak gösterilmiştir. Buton nesnesi, **`aracEkle`** metodu ile pencereye ekleniyor. Metot ise kendisine verilen adresi **`PencereAraci`** listesine ekliyor. Bu işlemden sonra gerçekleşen fare

olaylarında liste içerisindeki bütün araçların önce **icindemi** fonksiyonu çağrılıyor ve fare olayının aracı ilgilendirip ilgilendirmediğine karar veriliyor. Eğer fonksiyon **true** dönerse ara yüzün ilgili olay fonksiyonu çağrılıyor.




Şekil 3.15

# Olay Yönetimi

Tasarladığımız butonların üzerinde fare olaylarını yakalıyoruz fakat bu olaylara verilecek tepkiyi sadece **Buton** sınıfı içerisinde yazabiliyoruz. Yani bütün buton nesnelerinin verdiği tepkiler aynı olacaktır. **Buton** nesnesini kullanan programcılarında kendi fonksiyonlarını buton olaylarına bağlayabilmeleri gerekir. Aksi halde tasarladığımız buton pek kullanışlı olmayacaktır. Örneğin buton üzerinde fare butonuna tıklandığında buton nesnesini kullanan programcının bir fonksiyonu çağrılabilmelidir.

Şimdi **Buton** nesnesi ile başka bir nesnenin fonksiyonları arasında ilişki kurmak için basit bir olay yönetim sistemi oluşturacağız. Olay yönetim sisteminin temelinde geri çağırma(callback) fonksiyonlarını bulunacaktır. Pencere sınıfı her olaya karşılık bir geri çağırma fonksiyon listesi barındıracak, isteyen modüller kendi fonksiyonlarını bu listenin içerisine ekleyerek olaydan haberdar olacaklardır. Öncelikle fonksiyonları saklamaktan bahsedelim. C++ 11 standarttı üye fonksiyonları saklayabilmek için **function** nesnelerini geliştirmiştir. Örneğin Kod 3.17'de **tikla** fonksiyonunu saklaması için bir **function** nesnesi oluşturulmuştur (7.satır). Fonksiyon nesnesi şablon formunda tanımlanmıştır. Şablonun veri türü olarak fonksiyonumuzun yapısını veriyoruz. Fonksiyon nesnesine bir fonksiyonun atanması için **bind** fonksiyonunun kullanılması gerekmektedir. 11.satırda **bind** fonksiyonu kullanılıyor. Atama yapılacak fonksiyonun parametresiz olması bind fonksiyonunun tek parametre almasına sebep oluyor. 12.satırda fonksiyon nesnemizin "()" operatörü kullanılarak bağlı olduğu fonksiyon çağrılmaktadır. Bu örnekteki fonksiyonumuz herhangi bir sınıfa bağlı değildir. Üye fonksiyonların bağlanmasında **bind** fonksiyonunun kullanımı değişmektedir.

```
01 #include <iostream>
02 #include<functional>
03 void tikla()
04 {
05         std::cout << "tikla" << std::endl;
06 }
07 std::function<void()> functionTikla;
08
09 int main()
10 {
11         functionTikla = std::bind(tikla);
12         functionTikla();
13 }
```

Kod 3.16.

Fonksiyon nesnesinin kullanımını öğrendikten sonra artık buton tıklaması için çağrılacak olan standart bir fonksiyon nesnesi türü belirlememiz gerekiyor. Programımız içerisinde farklı fonksiyon nesnelerinin kullanılacağını düşünerek fonksiyon nesnelerin tanımını saklaması için "Olaylar.hpp" isimli bir dosya oluşturacağız. Dosyanın içeriği şu an için aşağıdaki gibi olacaktır. 5.satırda fonksiyon nesnesi için bir tür oluşturmaktadır. **FonksiyonTikla** bir tür ismi olarak kullanılacaktır. 7.satırda **FonksiyonTikla** türünde nesneler barındıracak bir liste türü oluşturulmaktadır.

```
01 #pragma once
02 #include<functional>
03 #include<vector>
04
05 typedef std::function<void()>          FonksiyonTikla;
06
07 typedef std::vector<FonksiyonTikla>    TiklaListesi;
```

Kod 3.17.

Tasarım yapmadan önce düşünmemiz gerekiyor. Fare tıklama olayı sadece buton araçlarının ilgilendiği bir olay mıdır? Başka araçlarında fare tıklama işlemine karşılık fonksiyon ekleyebilmesini sağlamak daha mantıklı görünüyor. Bu yüzden tıklama olayının eklenmesi işlemini **Buton** sınıfı yerine kalıtım aldığı **PencereAraci** sınıfında gerçekleştireceğiz. Kod 3.18. de **PencereAraci** sınıfının yeni tasarımı görünmektedir. Dikkat ediyorsanız bazı soyut fonksiyonlar standart sanal fonksiyona dönüştürülmüştür.

```
01 #pragma once
02 #include<SFML/Window/Event.hpp>
03 #include "Olaylar.hpp"
04 class PencereAraci
05 {
06 public:
07         PencereAraci();
08         virtual void fareHareket(int x, int y);
09         virtual void fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi);
10         virtual void fareButonBirakildi(sf::Event::MouseButtonEvent butonOlayi) ;
11         virtual bool icerdemi(int x, int y)=0;
12
13         void tiklaFonksiyonuEkle(FonksiyonTikla fonk);
14 protected:
15         TiklaListesi m_tiklaFonksiyonlari;
16 };
```

Kod 3.18.

Akla şu soru takılabilir neden soyut fonksiyonları normal fonksiyonlara çevirdik. İlk tasarımı yaparken olay yönetimini düşünmemiştik o yüzden her aracın ortak yapması gereken işlemleri tam olarak belirleyemedik ve tasarımımızda bir hata oluştu. Bu hata tasarımı tümüyle değiştirmemize sebep olacak boyutlarda değil. Amacımız **fareButonBasildi** fonksiyonunun bütün pencere araçları için geri çağırma fonksiyonlarının çağrılmasıdır. Kod 3.19'da **PencereAraci** sınıfının fonksiyonlarına ait gövdeler görülmektedir. **tiklaFonksiyonuEkle** metodu araca tıklandığı zaman devreye girecek yeni bir fonksiyonu eklememize yarıyor. Listeye eklenen fonksiyonlar **fareButonBasildi** metodu içerisinde sıra ile çağrılmaktadır.

```
01 #include "PencereAraci.hpp"
02 void PencereAraci::fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi)
03 {
04        for (auto siradaki : m_tiklaFonksiyonlari)
05               siradaki();
06 }
07 //Diğer fonksiyonların gövdesi şimdilik boş olacak
08 void PencereAraci::tiklaFonksiyonuEkle(FonksiyonTikla fonk)
09 {
10        m_tiklaFonksiyonlari.push_back(fonk);
11 }
```

Kod 3.19.

Şimdi deneme zamanı. **Buton** nesnemizi **Oyun** sınıfının kurucusunda oluşturmuştuk. Bu yüzden butona tıklandığında çağrılacak olan fonksiyonu da **Oyun** sınıfına ekleyeceğiz. Öncelikle fonksiyonu **Oyun.hpp** dosyasında tanımlayalım. Kod 3.20 üye fonksiyonun ön tanımı verilmiştir. Kod 3.21'de fonksiyonun gövdesi gösterilmiştir.

```
01 #pragma once
02
03 #include "Pencere.hpp"
04 #include "Buton.hpp"
05 class Oyun
06 {
07 public:
08 //Sınıfın diğer kısımlarında değişim yok
09 //……
10        void butonTikla();
11 //Sınıfın diğer kısımlarında değişim yok
12 //……
13 };
```

Kod 3.20.

```
01 //Sınıfın diğer kısımlarında değişim yok
02 void Oyun::butonTikla()
03 {
04        std::cout << "Merhaba Buton Tikla çalıştı" << std::endl;
05 }
06 //Sınıfın diğer kısımlarında değişim yok
```

Kod 3.21.

Butonun tıklama olayına yeni üye fonksiyonumuzu bağlama işlemi **Oyun** sınıfının kurucusunda yapılacaktır. Fonksiyonun ekleme işlemi Kod 3.22' de gösterilmektedir. Kurucu fonksiyonun değişmeyen kısımları bu kod parçası içerisinde gösterilmemiştir. 9.satırda üye fonksiyonumuzu bir fonksiyon nesnesine çeviriyoruz. Bu fonksiyon nesnesi daha önceden tanımladığımız **FonksiyonTikla** ile aynı türden olacaktır. 10. satırda buton nesnesinin **tiklaFonksiyonuEkle** metodu kullanılarak fonksiyonumuzun bağlantısı kurulmaktadır. Şu an programı çalıştırıp fare tıklamasını denerseniz, bağlantı kurduğunuz fonksiyonun çalışmadığını göreceksiniz.

```
01 #include "Oyun.hpp"
02 #include<iostream>
03 Buton b;
04 sf::Font font;
05 Oyun::Oyun()
06 {
07 //Sınıfın diğer kısımlarında değişim yok
08 //……
09         auto fonk = std::bind(&Oyun::butonTikla, this);
10        b.tiklaFonksiyonuEkle(fonk);
11 }
```

Kod 3.22.

Programın doğru bir şekilde çalışmamasının sebebi bağlantının kurulmasından kaynaklanmamaktadır. **Pencere** sınıfı butonun **fareButonBasildi** fonksiyonunu çağırmaktadır. Fakat olay fonksiyonlarını çağırma işlemi **PencereAraci** içerisindeki **fareButonBasildi** fonksiyonunda çağrılmaktadır Fakat Buton sınıfı **PencereAraci** sınıfından kalıtım almasına rağmen **fareButonBasildi** fonksiyonunu ezdiği için **Pencere** nesnesi en son tanımlanan fonksiyonu kullanacaktır(Problemi görmek için Buton sınıfındaki **fareButonBasildi** fonksiyonunda komut satırına bir mesaj çıkartabilirsiniz). Bu problemi, **Buton** sınıfında bulunan **fareButonBasildi** fonksiyonunu kaldırarak çözebiliriz. Ya da bu fonksiyon içerisinde kalıtım alınan temel sınıfın **fareButonBasildi** fonksiyonunu çağırabiliriz. Bu işlem yazılım dünyasında sıkça kullanılmaktadır. Zinciri tamamlamak olarak düşünebiliriz. Temel sınıfın fonksiyonu bütün kalıtım alan sınıfların kullanacağı bir fonksiyon tanımlıyor ama kalıtım alan sınıfların fonksiyondan beklediği ek görevlerde olabilir. O yüzden ilgili fonksiyon ezilerek tekrardan yazılmaktadır. Temel sınıftaki fonksiyonun işlemlerini de gerçekleştirmesi için ilk satırda çağrılıyor.

```
01 void Buton::fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi)
02 {
03 }
04 //……
05 void PencereAraci::fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi)
06 {
07        for (auto siradaki : m_tiklaFonksiyonlari)
08               siradaki();
09 }
```

Kod 3.23.

```
01 void Buton::fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi)
02 {
03         PencereAraci::fareButonBasildi(butonOlayi);
04 }
05 //……
06 void PencereAraci::fareButonBasildi(sf::Event::MouseButtonEvent butonOlayi)
07 {
08         for (auto siradaki : m_tiklaFonksiyonlari)
09                 siradaki();
10 }
```

Kod 3.24. Buton sınıfının fonksiyonu temel sınıfın aynı isimli fonksiyonunu çağırıyor

Pencere araçları daha detaylı işlemler ve ince ayarlar yapılabilir. Ama bunları yaparak konunun fazla dışına çıkmış oluruz.

# Menü Tasarımı

Menü kontrolü için ayrı bir sınıf oluşturulabilir. Fakat projemizi daha karmaşık hale getirmemek için menü yönetimini **Oyun** sınıfına bırakacağız. Öncelikle **Oyun** sınıfına iki tane **Buton** nesnesi ve bu nesnelerin kullanımı için dört metot ekleyeceğiz. Aşağıdaki kod bloğunda **Oyun** sınıfına yeni eklenen elemanlar gösterilmektedir.

```
01 #pragma once
02 #include "Pencere.hpp"
03 #include "Buton.hpp"
04 class Oyun
05 {
06 public:
07 //……
08         void btnYeniOyunTikla();
09         void btnCikisTikla();
10         void menuAyarla();
11         void menuCiz();
12 private:
13 //……
14         Buton               m_btnYeniOyun;
15         Buton               m_btnCikis;
16         sf::Font            m_font;
17 //……
18 };
```

Kod 3.24.

**menuAyarla** fonksiyonu iki buton nesnesinin sahne üzerindeki konumlarından boyutlarına kadar bütün işlemlerini gerçekleştirmektedir. **menuCiz** metodu butonları çizdirmek ile görevlidir. **btn** ile başlayan diğer iki fonksiyon ise ilgili butonlara tıklandığında çağrılacak olan geri çağırma fonksiyonlarıdır. Oyun sınıfının kurucu fonksiyonunun son hali aşağıdaki gibi olacaktır. Ayrıca şekil çizimini denemek için kullandığımız kodları kurucu fonksiyondan çıkardık.

```
01 Oyun::Oyun()
02 {
03         m_cerceveSuresi = 1.0f / 60.0f;
04         menuAyarla();
05 }
```

Kod 3.25.

**menuCiz** fonksiyonunun görevi iki butonun çizim fonksiyonlarını çağırmaktan ibarettir.

```
01 void Oyun::menuCiz()
02 {
03       m_btnCikis.ciz(m_pencere.pencereGetir());
04       m_btnYeniOyun.ciz(m_pencere.pencereGetir());
05 }
```

Kod 3.25.

**menuAyarla** fonksiyonun gövdesi aşağıdaki gibidir. Bu fonksiyon farklı parçalara ayrılabilir. Örneğin olay fonksiyonlarının bağlanması ayrı fonksiyona konumların belirlenmesi ayrı fonksiyonlara ayrılabilir. Konunun daha da uzamaması için hepsi bir fonksiyon içerisine yerleştirildi. Fonksiyonun ilk dört satırında butonların tıklama olaylarında çağrılacak olan **Oyun** sınıfına ait olan fonksiyonların bağlantısı kurulmaktadır. 8-12 satırları arasında butonlara ait yazıların font türleri atanmaktadır. (Font yükleme işlemleri buton sınıfı içerisine de saklanabilir). 20.satırdan sonra butonların sahne üzerindeki koordinatları belirlenmektedir. Öncelikle sahnenin ve ilk butonun boyutu daha kolay kullanılabilecek değişkenler içerisine atılıyor. Ardından ilk butonun yatayda ve dikeyde sahnenin ortasına yerleşmesi için gerekli olan x ve y koordinatı hesaplanmaktadır (dikey eksende tam ortada değil). Daha sonra aynı işlem ikinci buton için gerçekleştirilmektedir. Fonksiyonun son iki satırında da butonlarımız pencereye eklenmektedir.

```
01 void Oyun::menuAyarla()
02 {
03       auto fonk = std::bind(&Oyun::btnYeniOyunTikla, this);
04       m_btnYeniOyun.tiklaFonksiyonuEkle(fonk);
05       fonk = std::bind(&Oyun::btnCikisTikla, this);
06       m_btnCikis.tiklaFonksiyonuEkle(fonk);
07
08       if (m_font.loadFromFile("alev.ttf") != false)
09       {
10             m_btnCikis.font(m_font);
11             m_btnYeniOyun.font(m_font);
12       }
13
14       m_btnYeniOyun.renk(sf::Color::Red);
15       m_btnCikis.renk(sf::Color::Red);
16
17       m_btnYeniOyun.yazi("Yeni Oyun");
18       m_btnCikis.yazi(L"Çıkış");
19
20       auto sahneBoyut = m_pencere.pencereGetir().getSize();
21       auto btnBoyut = m_btnYeniOyun.boyutGetir();
22       float x = (sahneBoyut.x - btnBoyut.x) / 2.0f;
23       float y = (sahneBoyut.y - btnBoyut.y) / 2.0f;
24       m_btnYeniOyun.konum(x, y);
25       btnBoyut = m_btnCikis.boyutGetir();
26        x = (sahneBoyut.x - btnBoyut.x) / 2.0f;
27       m_btnCikis.konum(x, y+50);
28
29       m_pencere.aracEkle(&m_btnYeniOyun);
30       m_pencere.aracEkle(&m_btnCikis);
31 }
```

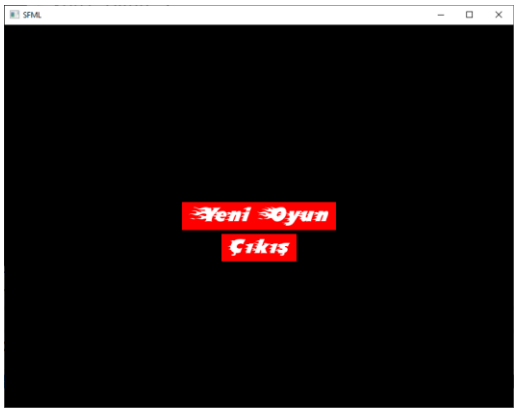
Kod 3.26.

Son olarak **sahneCiz** fonksiyonunda menünün çizilmesi için bir satır kod eklenmiştir.

```
01  void Oyun::sahneCiz()
02  {
03          m_pencere.cizimeBasla();
04
05          menuCiz();
06          m_pencere.cizimiBitir();
07  }
```

Kod 3.27.

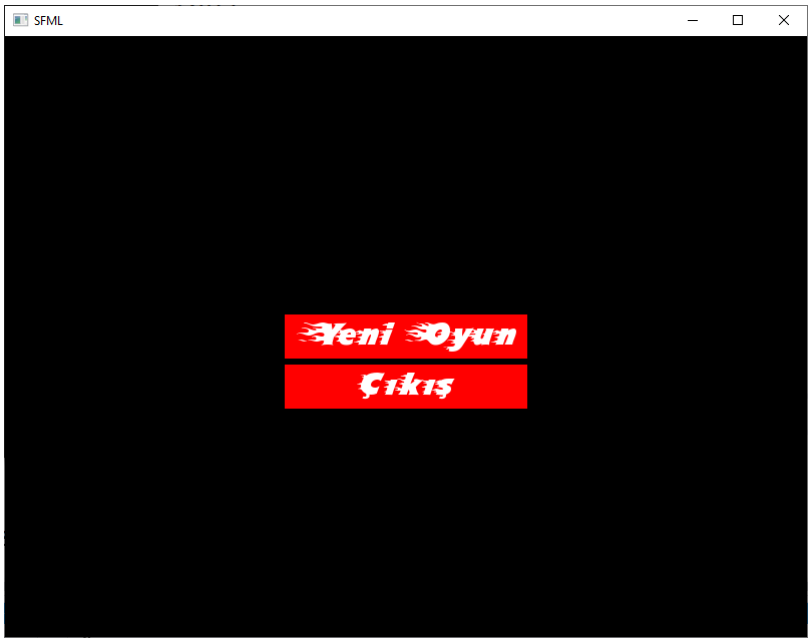
Programı çalıştırdığımızda elde edeceğimiz görüntü aşağıdaki gibi olacaktır. Dikkat ediyorsanız çıkış butonunun genişliği ile ilgili bir problemimiz var. Butonları tasarlarken genişliği yazı ile orantılı yaptık. Bu yüzden de çıkış daha küçük bir genişliğe sahip. Eğer çıkış butonunun genişliğini değiştirirsek bu sefer içerideki yazı solda kalacaktır. Buton sınıfında boyut ataması yaparken eğer dışarıdan girilen değer yazının boyutundan büyük ise yazının ortaya gelmesini sağlamamız gerekmektedir. Bunun için Buton sınıfının boyut ve konum fonksiyonlarını Kod 3.28'deki gibi değiştireceğiz. Boyut atama işlemi yeni boyut yazının boyutundan büyük olduğu durumlarda devreye girecektir (3.satır). Butonun konumu atanırken artık butonun genişliği düşünülerek hareket edilecektir. Programın son hali Şekil 3.17'deki gibi olacaktır.




Şekil 3.16

```
01  void Buton::boyut(float genislik, float yukseklik)
02  {
03          if (m_yazi.getLocalBounds().width < genislik)
04          {
05                  m_boyut = sf::Vector2f(genislik, yukseklik);
06                  m_arkaPlan.setSize(m_boyut);
07          }
08  }
09  void Buton::konum(float x, float y)
10  {
11          m_konum = sf::Vector2f(x, y);
12          auto yaziBoyut = m_yazi.getLocalBounds();
13
14          float farkx = (m_boyut.x - yaziBoyut.width) / 2.0f;
15
16          float yazix = abs(yaziBoyut.left);
17          m_yazi.setPosition(x + yazix + farkx, y);
18          m_arkaPlan.setPosition(m_konum);
19  }
```

Kod 3.28.




Şekil 3.15