TOBB Ekonomi ve Teknoloji Üniversitesi

ELE142 – Bilgisayar Programlama II

Dönem Projesi Raporu

Adı Soyadı	Ege Gökçen / Kaan Seyhan
Numara 1	221201022
Numara 2	221201034
Tarih	22.03.2025

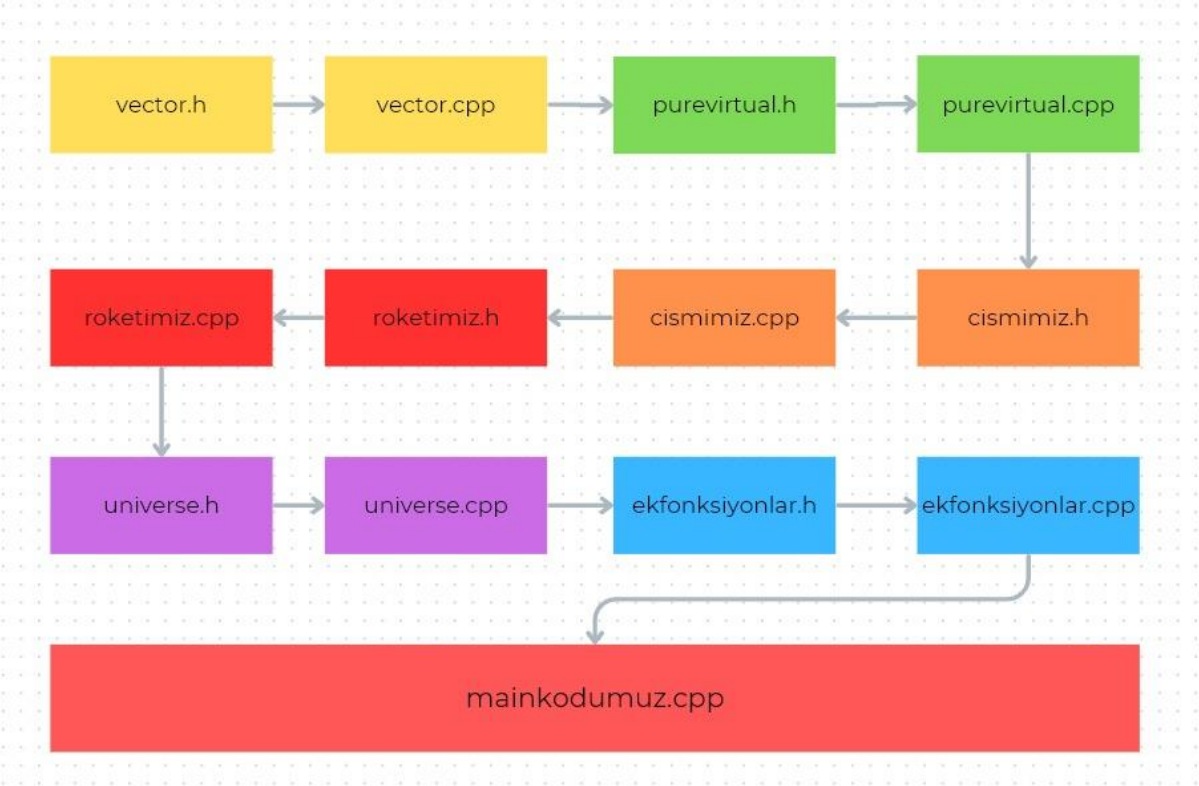
İçindekiler

1. Giriş

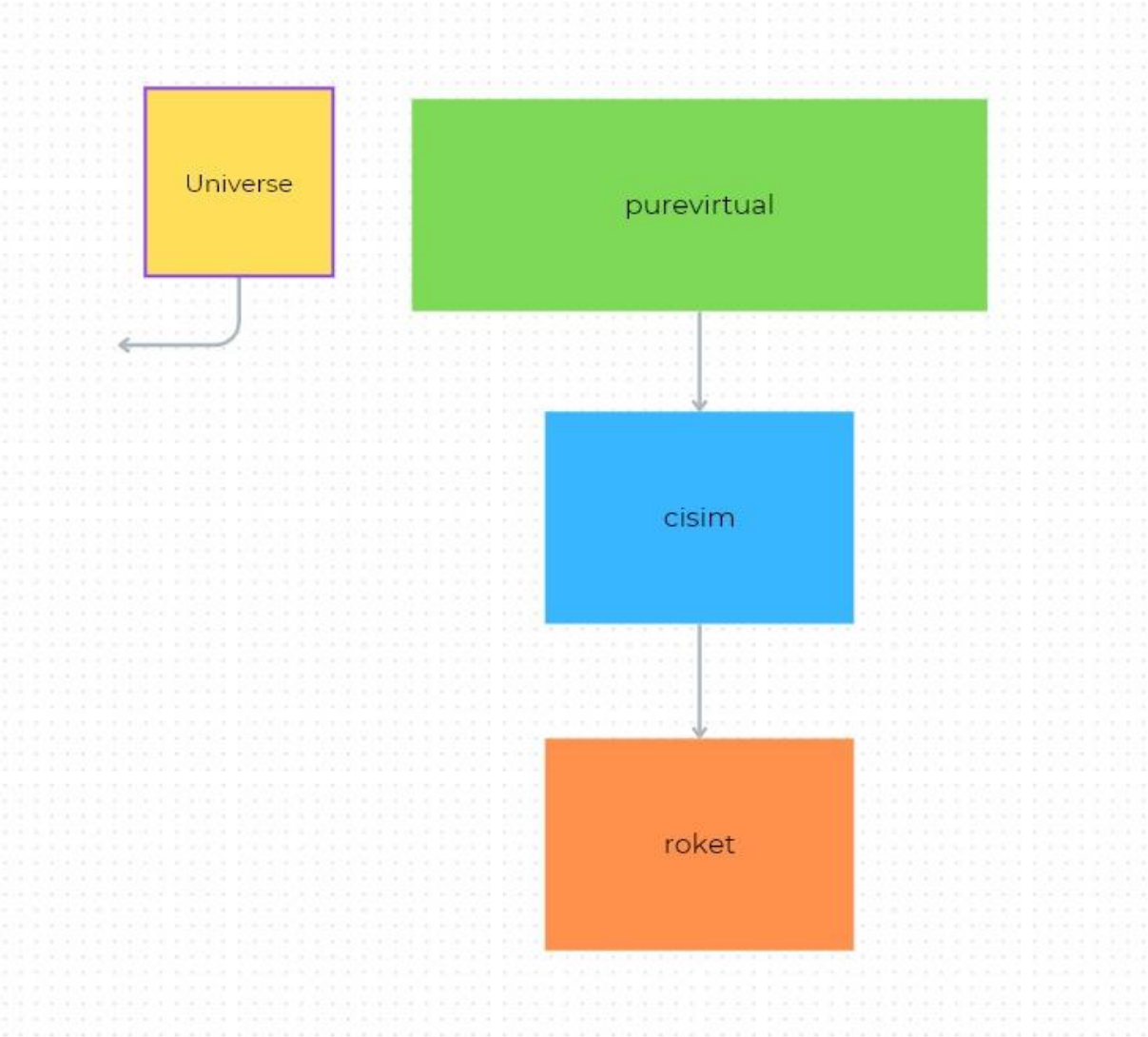
1.1 Ön Bilgiler Ve Rapor Okuyucusuna Notlar

Bu proje, C++ dilinde geliştirilmiş bir dinamik fizik simülasyonudur. Başlangıçta Dev C++ üzerinde yazılan kod, daha sonra Visual Studio Code (VSCode) ortamına uyarlanarak çalıştırılabilir hale getirilmiştir. Bu geçiş sırasında, kullanılan "canvas" kütüphanesinin de doğru bir şekilde bilgisayara uyarlanması gerekmektedir. Kullanıcıların, bu proje dosyalarını çalıştırmadan önce, "canvas" kütüphanesinin gerekli bağlantı ve yapılandırmalarının doğru bir şekilde yapılması gerektiği unutulmamalıdır.

Ayrıca, bu projede kullanılan tüm dosyaların belirli bir çalıştırma sırası vardır. Kodun düzgün bir şekilde çalışabilmesi için, dosyalar aşağıdaki sırayla derlenmeli ve çalıştırılmalıdır.



Resim 1.1.1: Kodun Optimal Şekilde Çalışması İçin Gereken Çalıştırım Sırası



Resim 1.1.2: Simülasyon İçin Tanımlanmış Sınıfların Kalıtım Şeması

Simülasyon esnasında oluşturulan “Universe” sınıfı, oluşturulan diğer sınıflardan algoritmik olarak bağımsızdır. Bu nedenle herhangi bir türetilmiş sınıf ile birleştirilmemiştir.

1.2 Teorik Altyapı

$$\vec{F} = -G\frac{m_1 m_2}{|\overrightarrow{r_2}-\overrightarrow{r_1}|^3}(\overrightarrow{r_2} - \overrightarrow{r_1}) \quad (1)$$

Denklem 1'deki $\vec{F}$, iki cisim arasındaki kütle çekim kuvvetini ifade eder. G, evrensel kütle çekim sabitidir ve başındaki eksi işareti, kuvvetin çekici olduğunu gösterir. m_1 ve m_2, sırasıyla birinci ve ikinci cismin kütleleridir. Paydada bulunan $|\overrightarrow{r_2} - \overrightarrow{r_1}|^3$ifadesi, iki cismin merkezleri arasındaki uzaklık fark vektörünün genliğinin küpünü belirtir ve kuvvetin büyüklüğünün mesafenin karesi ile ters orantılı olmasını sağlar. Son olarak, $(\overrightarrow{r_2} - \overrightarrow{r_1})$ ifadesi, bir cismin diğerine göre konum farkını gösterir ve kuvvetin yönünü belirler.

$$r = ae^{b\theta} \quad (2.a)$$

$$\frac{dr}{d\theta} = abe^{b\theta} \quad (2.b)$$

$$\rho = \frac{r}{\sqrt{1+(b^2)}} \quad (2.c)$$

$$s = \frac{a}{b}\sqrt{1+b^2}\left(e^{b\theta_2} - e^{b\theta_1}\right) \quad (2.d)$$

Logaritmik spiral, kutupsal koordinatlarda (2.a) denklemi ile ifade edilen bir eğridir. Burada r, orijinden olan uzaklık, theta açısal koordinat, a başlangıç yarıçapı, b ise spiral açılma oranını belirleyen sabittir. Spiralin türevi (2.b) formülüyle bulunur ve bu ifade, açısal değişime karşılık yarıçap değişiminin oranını verir. Eğrilik yarıçapı ise (2.c) formülüyle hesaplanır ve bu değer, spiralin kıvrılma derecesini belirler. Logaritmik spiralin belirli bir açısal aralıktaki yay uzunluğu, (2.d) formülüyle hesaplanır. Görece, bu tür spiraller doğada galaksi kolları ve fırtına sistemlerinde yaygın olarak bulunur. Projede gerçekleştirilecek olan simülasyonun uzay ortamında geçeceği varsayıldığından ve uzayda kara deliklerin bulunması gerçeğinden dolayı nesnelerin serbest hareketi ve yörüngesel dinamiklerini temsil etmek için bu logaritmik spiral hareket formu tercih edilmiştir.

Bu eğrinin matematiksel karakteristiğini daha detaylı incelemek için aşağıdaki bağlantıyı ziyaret ediniz.

(https://mathcurve.com/courbes2d.gb/logarithmic/logarithmic.shtml)

1.3 Projenin Amacı

Bu proje, Üç Cisim Problemi'nin C++ kullanılarak Nesne Yönelimli Programlama (OOP) prensipleri çerçevesinde simüle edilmesini amaçlamaktadır. Fizikte klasik mekanik çerçevesinde çözülemeyen Üç Cisim Problemi, üç farklı kütleli cismin kütleçekim etkisi altında nasıl hareket edeceğini belirlemeye çalışır. Bu simülasyon, üç cismin dinamik etkileşimini modelleyerek kapalı matematiksel analizi yapılamayan hareket formlarını görselleştirmeye olanak tanımaktadır.

Newton'un hareket yasaları ve kütleçekim teorisi, iki cismin birbirine uyguladığı kuvvetleri hesaplamada oldukça başarılıdır. Ancak sisteme üçüncü bir cisim dahil edildiğinde, problem lineer olarak çözülebilir durumda olmaktan çıkar ve karmaşık, kaotik(belirlenemez) yörüngeler oluşur. Bu problem, gezegen sistemlerinden yıldızların hareketlerine, galaksi çarpışmalarına kadar birçok fiziksel fenomenin temelinde yer alır.

Projede C++'ın Nesne Yönelimli Programlama (OOP) özellikleri etkin şekilde kullanılmıştır:

- **Polimorfizm:**
  Üç cisim ve roket gibi farklı nesneler, ortak bir temel sınıftan türetilmiş ve dinamik bağlama (dynamic binding) ile yönetilmiştir.

- **Kapsülleme:**
  Cisimlerin konumu, hızı ve kütlesi gibi veriler özel üyeler olarak tanımlanmış ve bunlara erişim için get/set metodları kullanılmıştır.

- **Miras:**

  "cisim" sınıfından türetilen "roket" sınıfı hem kütleçekim etkisini hem de kendi motor itişini hesaplayarak bileşik hareket sergileyen bir nesne olarak modellenmiştir. Yalnızca kütleçekim yasalarına bağlı kalmayıp kendi motor gücüyle hareket edebildiği için sistemde kontrol edilebilir bir değişken eklemektedir. Roket sayesinde, yalnızca pasif hareket eden gökcisimlerinin aksine, aktif olarak yörünge değişiklikleri yapabilen bir nesnenin sistem içindeki etkisi de incelenebilmektedir. Bu sınıf tanımları hakkında daha detaylı bilgi raporun ilerleyen kısımlarında verilecektir.

2. Genel Yazılım Mimarisi

Resim 1.1.1'de gösterildiği gibi, bu projede tüm sınıf ve fonksiyon tanımlarının farklı dosyalarda yapılıp birbirine “#include” yöntemiyle bağlanması, yazılım geliştirme sürecinde çeşitli avantajlar sağlamaktadır. Öncelikle, modülerlik ilkesi gereği kodun daha düzenli ve okunabilir olması sağlanmış, her bileşenin kendi dosyasında tanımlanmasıyla belirli bir sınıf veya fonksiyon üzerinde çalışırken diğer kodlara bir bağımlılık koşulu olmaksızın değişiklik yapılabilmesi mümkün hale getirilmiştir. Bunun yanı sıra, yeniden kullanılabilirlik açısından, belirli bir fonksiyon veya sınıfın farklı projelerde ya da aynı proje içinde farklı bölümlerde tekrar kullanılabilmesi sağlanmıştır. Aynı zamanda, bu yöntem derleme süresini optimize etme açısından da faydalı olmuştur(ilerleyen kısımlarda çalışma süresi analizlerinde görülebilir), yalnızca değişiklik yapılan dosyanın ilgili kısımlarının tekrar derlenmesi sayesinde tüm projenin baştan derlenmesine gerek kalmamıştır. Bu durum, geliştirme sürecinin hızlanmasına katkıda bulunmuştur. Dahası, eş zamanlı geliştirme imkânı sunarak, birden fazla geliştiricinin aynı proje üzerinde paralel çalışabilmesine olanak tanımış ve iş akışının kesintisiz devam etmesini sağlamıştır. Son olarak, bakım ve genişletilebilirlik açısından, kodun tek bir merkezi dosya yerine bağımsız bileşenler halinde bulunması, yeni özelliklerin eklenmesini veya mevcut yapıların güncellenmesini daha sistematik ve sürdürülebilir bir hale getirmiştir.

Fonksiyonlar belirli görevleri yerine getirecek şekilde ayrılmış, her bileşenin yalnızca kendi sorumluluğunu üstlenmesi sağlanmıştır(Single-responsibility principle). Dinamik bellek

yönetimi uygulanarak heap üzerinde nesne ataması gerçekleştirilmiş, böylece bellek kullanımında esneklik sağlanmış ve nesneler çalışma sürecinde gerektiği gibi oluşturulup silinebilmiştir. Kalıtım yapısı kullanılarak, temel bir soyut sınıf olan “purevirtual “tanımlanmış ve “cisim“ile “roket” gibi nesneler bu sınıftan türetilerek ortak bir arayüz üzerinden yönetilmiştir. Bu yapı, polimorfizm sayesinde türetilmiş sınıfların kendi özel fonksiyonlarını bastırarak çok biçimliliği desteklemesine olanak tanımıştır. Sanal fonksiyonlar kullanılarak, temel sınıfta tanımlanan metotlar alt sınıflarda farklı işlevler gerçekleştirecek şekilde yeniden yapılandırılmıştır. Özellikle kuvvet ve hareket hesaplamalarında, türetilmiş sınıflar kendilerine özgü hesaplama yöntemlerini uygulamış, böylece her nesne için spesifik fiziksel davranışlar tanımlanmıştır. Bağlı liste veri yapıları kullanılarak dinamik eleman yönetimi sağlanmış, yeni nesnelerin bellek ataması ile oluşturulup gerektiğinde bellekte yer açacak şekilde silinmesi mümkün hale getirilmiştir. Bu sayede sistem, önceden belirlenmiş sabit bir dizi yapısına bağlı kalmadan değişken sayıda nesne ile çalışabilmiştir. Simülasyon sürecinde, nümerik zaman ilerletme adımlamaları ve nesne güncellemeleri belirli bir sıralama ile işlenmiş, her adımda tüm nesnelerin etkileşimleri hesaplanarak sistemin gerçekçi bir dinamik yapı sergilemesi sağlanmıştır.

Bu projede, "ifndef", "define" ve "endif" derleme öncesi komutlarının kullanılması, hem kod organizasyonu açısından kolaylık sağlamakta hem de teknik olarak bir zorunluluk oluşturmaktadır. Projede çok sayıda başlık ve kütüphane dosyası ve birbirine bağlı modüller bulunduğundan, aynı dosyanın birden fazla kez dahil edilmesi durumu ortaya çıkabileceği açıktır. Eğer bu önlem alınmazsa, bir başlık dosyasının içeriği birden fazla kez derleyiciye sunulabilir ve bu da yeniden tanımlama hatalarına neden olabilir. "ifndef" ve "define" komutları kullanılarak bir başlık dosyasının içeriği yalnızca bir kez derleyici tarafından işlenir, böylece çakışmalar ve gereksiz tekrarlar önlenmiş olur. Aynı zamanda, bu yapı sayesinde bağımlılıklar arasındaki çakışmaların engellenmesi ve kodun modülerliğinin korunması sağlanır. Özellikle, proje içinde birçok bileşenin farklı dosyalarda tanımlandığı göz önüne alındığında, her bileşenin bağımsız olarak derlenebilir olması önem taşımaktadır. Sonuç olarak, "ifndef", "define" ve "endif" yapılarının kullanımı, yalnızca iyi bir kod organizasyonu sağlamakla kalmayıp, aynı zamanda projenin sorunsuz çalışmasını garanti eden temel bir gereklilik olarak kabul edilir.

3. Simülasyon

3.1 Grafik Çizim Kütüphanesi(“canvas.cpp”)

Bu kütüphane, bir simülasyon ortamında nümerik olarak hesaplanan noktaların görselleştirilmesi amacıyla geliştirilmiştir. Simülasyon sırasında elde edilen verilerin, grafiksel bir çıktı olarak elde edilmesi için boş bir koordinat sistemi benzeri yapı oluşturulmuş ve çeşitli çizim fonksiyonları ile desteklenmiştir.

3.2 “vector2d” Sınıfı

```
#ifndef VECTOR_H
#define VECTOR_H
#define G  1.0
#define dt 0.012


class vector2d

{
public:

    vector2d();
    vector2d(double , double );
    vector2d(const vector2d& );
    ~vector2d();
    double getx() const;
    double gety() const;
    void   setx(double );
    void   sety(double );
    double vmagnitude() const;
    vector2d operator+(const vector2d& ) const;
    vector2d operator-(const vector2d& ) const;
    vector2d operator*(double)       const;
    vector2d operator/(double )       const;
    vector2d& operator=(const vector2d& );

private:

    double* px;
    double* py;
};

#endif
```

Resim 3.2.1 : “vector2d” Sınıfı Kod Bloğu

“vector2d” sınıfı, Resim 3.2.1’den de görüleceği üzere 2 boyutlu vektörlerle çalışmak için tasarlanmış bir sınıftır ve genellikle fiziksel simülasyonlarda, proje parametrelerinden olan kuvvet, hız, ivme ve konum gibi vektörel niceliklerin hesaplanmasında kullanılır. Bu sınıf, vektörlerle çeşitli işlemleri daha verimli ve düzenli bir şekilde gerçekleştirebilmelerini sağlar. “vector2d” sınıfı, vektörün x ve y bileşenlerini saklayan işaretçilerle ("px" ve "py") donatılmıştır. Bu işaretçiler, bellekte dinamik olarak ayrılan alanlarda vektörün bileşenlerini tutar ve sınıf, hafıza yönetimi konusunda daha esnek bir yapıya sahip olur. Vektörlerle yapılan işlemleri hızlandırmak ve daha etkili hale getirmek amacıyla, sınıf üzerinde C++ standart kütüphanesinde bulunan operatörlere aşırı yükleme (overloading) yapılmıştır. Bu sayede, kullanıcılar doğrudan operatörlerle toplama, çıkarma gibi işlemleri gerçekleştirebilirler. Ayrıca, vektörün x ve y bileşenlerine erişim için "getx" ve "gety" fonksiyonları kullanılırken, bu bileşenlere değer atamak için "setx" ve "sety" fonksiyonları kullanılır. Bu fonksiyonlar, sınıfın barındırdığı verileri güvenli bir şekilde değiştirmeye olanak sağlar.

“vector2d” sınıfındaki "px" ve "py" değişkenleri private olarak tanımlanmıştır, bu da değişkenlere dışarıdan doğrudan erişimi engeller ve sınıfın iç mantığını korur. Bu, enkapsülasyon ilkesine dayalı bir tasarımıdır. Fonksiyonlar “const” olarak tanımlandığı için, dışarıdan yapılan çağrılarla veri üzerinde değişiklik yapılamaz, bu da yazılımın hata yapma olasılığını azaltır. Böylece, sınıfın parametresel durumunu değiştirmeden, sadece gerekli hesaplamalar yapılır ve verilerin tutarlılığı sağlanır. Bu tür bir kurulm, yazılımın bakımını kolaylaştırır ve kodun daha güvenli çalışmasını sağlar.

```
vector2d vector2d::operator+(const vector2d& rhs) const
{
    vector2d result;
    result.setx(this->getx() + rhs.getx());
    result.sety(this->gety() + rhs.gety());
    return result;

}
```

Resim 3.2.2 : Operatör Aşırı Yüklemesi İçin Örnek Kod Bloğu

Bu kod parçası Resim 3.2.2'den görüleceği üzere, "vector2d" sınıfında bulunan toplama operatörünün aşırı yüklenmesini (operator overloading) göstermektedir. Bu fonksiyon, iki vektörün toplanmasını sağlar. İlk olarak, bir "vector2d" türünde "result" adlı bir değişken tanımlanmıştır. Bu değişken, iki vektörün toplamını saklayacak ve geri döndürülecektir. "result" değişkeni, sonuç vektörünü tutar ve bu nedenle bir "vector2d" türünde tanımlanmıştır. Operatördeki diğer iki ifade, iki vektörün "getx()" ve "gety()" fonksiyonlarıyla erişilen x ve y bileşenlerinin toplanmasını sağlar. Burada, "this->getx()" ve "rhs.getx()" ifadeleri, sırasıyla sol ve sağ vektörün x bileşenlerini temsil eder. Aynı şekilde, "this->gety()" ve "rhs.gety()" ifadeleri de y bileşenlerini temsil eder. Bu değerler toplandıktan sonra, "result.setx()" ve "result.sety()" fonksiyonlarıyla, bu toplamlar "result" vektörüne atanır. Son olarak, "result" geri döndürülür.

Bu mantık, diğer operatör aşırı yüklemelerinde de benzer şekilde uygulanır. Örneğin, çıkarma, çarpma veya bölme operatörleri için de vektör uzayında vektörler için tanımlanmış benzer işlemleri gerçekleştirir.

3.3 "purevirtual" Sınıfı

```
#ifndef PUREVIRTUAL_H
#define PUREVIRTUAL_H

#include "vector.h"


class purevirtual
{
public:
    virtual ~purevirtual() {}
    virtual void kuvvethesapla(purevirtual** arr, int n) = 0;
    virtual void ivmehesapla()                           = 0;
    virtual void hizhesapla()                            = 0;
    virtual void konumhesapla()                          = 0;
    virtual void update(purevirtual** arr, int n)        = 0;
    virtual void       settf(vector2d force)       = 0;
    virtual vector2d   gettf()               const = 0;
    virtual double     getkutle()            const = 0;
    virtual vector2d   getkonum()            const = 0;
    virtual vector2d   gethiz()              const = 0;
    virtual void       kutlehesapla()              = 0;
};

#endif
```

Resim 3.3.1 : "purevirtual" Sınıfı Kod Bloğu

Resim 3.3.1’de daha önce de bahsedildiği gibi, bu kod parçasında “#ifndef” ve “#define” gibi derleme öncesi yapılar kullanılmıştır. Burada tanımlanan purevirtual sınıfı, safsanal bir sınıf olarak tasarlanmış ve türetilmiş sınıflara temel oluşturması amaçlanmıştır. Saf sanal sınıflar üzerinden doğrudan bir nesne oluşturumu standart C++ yazılım dilinde tanımlı değildir; bunun yerine, başka sınıflar bu sınıfları türeterek fonksiyonlarının içeriklerini sağlamak zorundadır.

“Virtual” anahtar kelimesi, Resim 3.3.1’de görülen üye fonksiyonların fonksiyonların türetilmiş sınıflarda geçersiz kılınabileceğini belirtir. Yani, purevirtual sınıfında tanımlanan her fonksiyon, türetilen sınıflarda kendi işlevselliğine göre yeniden yazılabilir. Bu yaklaşım, polimorfizm yani çok biçimliliği sağlar ve farklı türdeki nesnelerin aynı arayüzü kullanarak farklı şekillerde işlem yapmasına olanak tanır. Fonksiyonların sonunda = 0 ifadesi ise, bu fonksiyonların saf sanal olduğunu belirtmek için konulmuştur. Bu, fonksiyonların sadece birer deklarasyon olduğuna, yani içeriği olmayan ve türetilen sınıflarda tanımlanması gereken fonksiyonlar olduklarına işaret eder.

Fonksiyonlar, daha sonra türetilmiş sınıflarda açıklanacak ve her bir fonksiyonun işlevi, türetilen sınıflarda daha detaylı şekilde tanımlanacaktır. Bu yapı, yazılımın esnekliğini artırır, çünkü her türetilmiş sınıf kendi özgün işlevselliğini sağlayabilirken, ortak bir arayüz üzerinden işlem yapılmasını da mümkün kılar.

Projede bulunan "purevirtual.cpp" dosyası içeriği boş bırakılmıştır. Bunun nedeni, bu dosyanın yalnızca bir arayüz görevi gören "purevirtual.h" başlık dosyasına karşılık gelmesidir. "purevirtual" sınıfı tamamen soyut bir sınıf olarak tanımlandığından, herhangi bir fonksiyonun doğrudan uygulanmasını gerektirmemektedir. "purevirtual.cpp" dosyasında herhangi bir fonksiyon veya veri üyesinin tanımlanmasına gerek duyulmamış, yalnızca başlık dosyası aracılığıyla diğer sınıflara kalıtım yoluyla aktarılması sağlanmıştır.

3.4 “cisim” Sınıfı

```
#ifndef CISIM_H
#define CISIM_H

#include "purevirtual.cpp"
#include "vector.cpp"
#include <iostream>

class cisim : public purevirtual
{
public:
    static int cisimsayisi;

    cisim(const double& mkutle, const vector2d& mkonum, const vector2d& mhiz);
    cisim();
    cisim(const cisim& other);
    virtual ~cisim();

    virtual void kuvvethesapla(purevirtual** arr, int n);
    virtual void ivmehesapla();
    virtual void hizhesapla();
    virtual void konumhesapla();
    virtual void update(purevirtual** arr, int n);
    virtual void       settf(vector2d force);
    virtual vector2d   gettf()       const;
    virtual double     getkutle()    const;
    virtual vector2d   getkonum()    const;
    virtual vector2d   gethiz()      const;
    virtual void       kutlehesapla();
    virtual void sethiz(const vector2d& v);
    virtual void setkutle(double m);
    virtual void setkonum(const vector2d& p);

protected:
    double*   kutle;
    vector2d* konum;
    vector2d* hiz;
    vector2d* totalforce;
    vector2d* ivme;
};

#endif
```

Resim 3.4.1 : “cisim” Sınıfının Kod Bloğu

Resim 3.4.1’deki “cisim” sınıfı, bu projede simülasyonu gerçekleştirilecek olan üç cisim problemi gibi karmaşık fiziksel simülasyonlarda önemli bir rol oynamaktadır. Bu sınıf, uzayda varolabilecek herhangi bir cismi modellemek için kullanılan temel yapı taşlarından biridir. Sınıf, “purevirtual” sınıfından türetilmiş olup, bu, fonksiyonlarının türetilen sınıflarda özelleştirilmesi gerektiğini belirtir. Bu da demektir ki, her cisim türü, kendine özgü fiziksel

davranışlarını bu temel sınıf üzerinden alacak şekilde geliştirilmiştir. Bu sınıfın “vector2d” kütüphanesi ile bir bağlantısı olduğu açıktır. “vector2d” sınıfı, cisimlerin konum, hız, ivme gibi vektörel büyüklüklerini temsil etmekte kullanılır ve bu nedenle cisim sınıfı, bu kütüphaneye başvurmaktadır. “vector2d” sınıfı olmadan, cisimlerin dinamikleri ve hareketleri doğru bir şekilde hesaplanamaz, dolayısıyla bu kütüphane, bu sınıf için anahtar bir parametredir.

```
int cisim::cisimsayisi = 0;

cisim::cisim(const double& mkutle, const vector2d& mkonum, const vector2d& mhiz)
{
    cisimsayisi++;
    kutle       = new double(mkutle);
    konum       = new vector2d(mkonum);
    hiz         = new vector2d(mhiz);
    totalforce  = new vector2d(0, 0);
    ivme        = new vector2d(0, 0);
}
```

Resim 3.4.2 : “cisim” Sınıfının Kurucu Fonksiyonu

Resim 3.4.2’deki kurucu fonksiyon, "cisim" sınıfının bir nesnesi oluşturulduğunda çağrılan bir constructor(kurucu) olup, nesnenin başlangıç değerlerini belirlemek açısından kritik bir rol oynamaktadır. "mkutle", "mkonum" ve "mhiz" parametreleri nesnenin fiziksel özelliklerini temsil etmekte olup, her biri anahtar bir parametredir. "mkutle" değişkeni, nesnenin kütlesini belirlediğinden, bellekte doğrudan bir alan tahsis edilerek "kutle" değişkenine atanmıştır. "mkonum" ve "mhiz" değişkenleri, nesnenin konumunu ve hızını temsil ettiğinden, iki boyutlu bir vektör yapısı olan "vector2d" sınıfı aracılığıyla saklanmaktadır. "totalforce" ve "ivme" değişkenleri, nesneye etki eden toplam kuvveti ve ivmeyi temsil etmekte olup, default(varsayılan) olarak sıfır vektörleri ile başlatılmıştır. Tüm bu değişkenlerin heap bölgesinde dinamik olarak oluşturulması, nesnenin çalışma döngüsü boyunca esnek bir bellek yönetimi sağlamakta ve gerektiğinde farklı nesneler arasında bağımsız olarak kullanılmasına olanak tanımaktadır. Parametrelerin "const" referans olarak tanımlanması, yazılım dili parametrelerinden hem değer hem de ifade kavramlarının fonksiyona parametre olarak girilebilmesi amacıyla gerçeklenmiştir.

Sınıfta kullanılan değişmez(const) anahtar kelimesi, belirli verilerin değiştirilemez olduğunu derleyiciye sunar. Örneğin, “kutle” ve “konum” gibi özellikler dışarıdan değiştirilemez, sadece belirli fonksiyonlarla erişilebilir ve yönetilebilir. Bu, sınıfın güvenlik faktörünü artıran bir özelliktir. “set” ve “get” gibi fonksiyonlar, bu değişmez verilerle etkileşime geçmek için kullanılır. “getkutle()”, “getkonum()” gibi fonksiyonlar, cisimlerin kütlesine ve konumuna erişmek için kullanılırken, “sethiz()”, “setkonum()” gibi fonksiyonlar, cisimlerin hız ve konumlarını değiştirebilmek için kullanılır. Bu fonksiyonlar, dışarıdan gelen verilere zarar vermeden, yalnızca sınıfın belirlediği kurallar çerçevesinde veri manipülasyonu yapılmasına olanak tanır. “kuvvethesapla()”, “ivmehesapla()”, “hizhesapla()” gibi hesaplama fonksiyonları, ilerleyen kısımlarda daha detaylı bir şekilde açıklanacaktır. Bu fonksiyonlar, cisimlerin fiziksel hareketlerini belirleyen temel hesaplamaları yapar ve simülasyonun doğruluğu için büyük önem taşır.

Bu projede "cisimsayisi" değişkeni, "static" olarak tanımlanmış olup, tüm "cisim" nesneleri arasında ortak bir değer olarak tutulmaktadır. "cisimsayisi" değişkeni, "cisim" sınıfına ait her yeni nesne oluşturulduğunda "cisim" sınıfının oluşturucu fonksiyonu içerisinde bir artırılmakta ve her nesne silindiğinde, yani yıkıcı fonksiyon çalıştırıldığında bir azaltılmaktadır. Böylece, programın herhangi bir anında mevcut "cisim" nesnelerinin sayısı takip edilebilmektedir. Bu statik üye değişken, daha sonra "cisim" nesnelerinin sayısını esas alarak döngüler içerisinde nesnelerin güncellenmesi, simülasyon işlemlerinin gerçekleştirilmesi ve bellek yönetiminin sağlanması gibi işlemlerde kritik bir rol oynayacaktır.

```
void cisim::kuvvethesapla(purevirtual** arr, int n)
{
    totalforce->setx(0);
    totalforce->sety(0);

    for(int i = 0; i < n; i++)
    {
        if(arr[i] == this)
            continue;

        cisim* other = dynamic_cast<cisim*>(arr[i]);
        if(!other)
            continue;

        vector2d diff = this->getkonum() - other->getkonum();
        double r = diff.vmagnitude();
        if(r == 0)
            continue;

        double forceVal = -G * this->getkutle() * other->getkutle() / (r*r*r);
        vector2d force = diff * forceVal;

        *totalforce = *totalforce + force;
    }
}
```

Resim 3.4.3 : “cisim” Sınıfındaki Kuvvet Hesaplama Üye Fonksiyonu

Resim 3.4.3’teki "kuvvethesapla" fonksiyonu, her bir "cisim" nesnesine etki eden toplam kuvveti hesaplamak için tasarlanmıştır. Bu hesaplama sırasında, tüm diğer nesnelerle olan etkileşim dikkate alınarak ve Denklem 1’den faydalanarak net kuvvet vektörü elde edilmektedir. Fonksiyon, öncelikle "totalforce" değerini sıfırlayarak başlar ve ardından "purevirtual" türündeki nesnelerle olan etkileşimleri göz önünde bulundurur. "dynamic_cast" kullanılarak, yalnızca "cisim" sınıfından türetilmiş nesnelerle işlem yapılması sağlanmıştır. İlgili nesnenin kendisiyle işlem yapılmasını engellemek amacıyla bir kontrol mekanizması eklenmiştir. Nesneler arasındaki konum farkı parametresi vektörel bir nicelik olarak hesaplanmakta, bu vektörün genliği belirlenerek kütle çekimi kuvveti hesaplanmaktadır. Newton’un evrensel çekim yasasına göre, iki nesne arasındaki çekim kuvveti, kütlelerin çarpımı ile doğru, aralarındaki mesafenin karesi ile ters orantılıdır. Bu hesaplamada "G" yerçekimi sabiti olarak tanımlanmıştır ve kuvvet bileşenleri, konum farkı ile orantılı olacak şekilde hesaplanarak "totalforce" değişkenine eklenmektedir.

```
void cisim::ivmehesapla()
{
    double m = this->getkutle();
    if(m <= 0)
        return;

    vector2d accel = (*totalforce) / m;
    ivme->setx(accel.getx());
    ivme->sety(accel.gety());
}
```

Resim 3.4.4 : “cisim” Sınıfındaki İvme Hesaplama Üye Fonksiyonu

$$\vec{a} = \frac{\vec{F}}{m} \tag{3.a}$$

Resim 3.4.4’teki "ivmehesapla" fonksiyonu, "kuvvethesapla" fonksiyonunda belirlenen net kuvvete bağlı olarak nesnenin ivmesini hesaplamaktadır. İvme Denklem 3.a’dan da görülebileceği üzere, Newton mekaniğinin ikinci yasasına göre elde edilmektedir. Fonksiyon, öncelikle nesnenin kütlesini kontrol eder ve negatif veya sıfır bir kütleye sahip olup olmadığını değerlendirir. Kütlesi sıfır olan nesneler için ivmenin bulunmayacağı bir gerçektir, çünkü fiziksel olarak anlamlı değildir. Kuvvet vektörü, kütleye skaler(yönsüz) olarak bölünerek ivme vektörü elde edilmekte ve bu değer "ivme" değişkenine atanarak saklanmaktadır. Hesaplama sürecinde, bellek yönetimi açısından verimliliği artırmak amacıyla "this" işaretçisi kullanılarak doğrudan nesnenin üye değişkenlerine erişim sağlanmıştır.

```
void cisim::hizhesapla()
{
    vector2d deltaV = (*ivme) * dt;
    *hiz = *hiz + deltaV;
}
```

Resim 3.4.5: “cisim” Sınıfındaki Hız Hesaplama Üye Fonksiyonu

$$\vec{v}(t + \Delta t) \approx \vec{v}(t) + \vec{a}(t)\,\Delta t \tag{3.b}$$

Resim 3.4.5’teki "hizhesapla" fonksiyonu, nesnenin mevcut hızını, daha önce hesaplanan ivme bilgisi ile güncelleyen bir işleve sahiptir. Denklem 3.b’ye göre, hız vektörü zamanla ivmenin etkisiyle değişmektedir. "dt" zaman adımı kullanılarak, ivme ile çarpılan bu küçük zaman dilimi, hız vektörüne eklenmekte ve böylece nesnenin bir sonraki adımda sahip olacağı hız değeri belirlenmektedir. Bu işlemde, nesnenin mevcut hızı doğrudan değiştirilmekte ve bu güncelleme heap üzerinde dinamik olarak oluşturulan "hiz" değişkeni üzerinde gerçekleştirilmektedir. "->" operatörü ile dinamik bellek içerisinde saklanan vektör değişkenlerine erişim sağlanmış ve doğrudan güncellenmiştir.

```
void cisim::konumhesapla()
{
    vector2d deltaPos = (*hiz) * dt;
    *konum = *konum + deltaPos;
}
```

Resim 3.4.6 : “cisim” Sınıfındaki Konum Hesaplama Üye Fonksiyonu

$$\vec{x}(t + \Delta t) \approx \vec{x}(t) + \vec{v}(t)\,\Delta t \qquad (3.c)$$

Resim 3.4.6’daki "konumhesapla" fonksiyonu, nesnenin daha önce verilen ilkelerle güncellenmiş hızı ile bir sonraki konumunu belirlemek için kullanılır. Temel fizik prensiplerine göre Denklem 3.c’den de görülebileceği üzere, bir nesnenin konumu belirli bir zaman dilimi (dt) boyunca değişimi ile hesaplanmaktadır. Bu fonksiyon, "hizhesapla" fonksiyonunun ürettiği yeni hız değerini kullanarak, "dt" zaman adımı ile çarpıp nesnenin mevcut konumuna eklemektedir. Böylece nesne, yeni hızı doğrultusunda hareket ettirilerek konum değişikliği sağlanmaktadır. Güncellenen konum değeri heap üzerinde saklanan "konum" değişkeni üzerinde tutulmakta ve "->" operatörü aracılığıyla erişilerek güncellenmektedir. Bu süreç, her adımda nesnenin doğru bir şekilde hareket etmesini sağlayarak simülasyonun gerçekçi bir şekilde ilerlemesini garanti etmektedir. Ayrıca, daha önce bahsedilen grafik çizim kütüphanesi ile burada hesaplanan koordinatlar çizdirilecek ve simülasyondaki nesnelerin hareketleri görselleştirilerek kullanıcıya sunulacaktır.

```
void cisim::update(purevirtual** arr, int n)
{
    for(int i = 0; i < n; i++)
    {
        cisim* c = dynamic_cast<cisim*>(arr[i]);
        if(!c)
            continue;

        c->kuvvethesapla(arr, n);
        c->ivmehesapla();
        c->hizhesapla();
        c->konumhesapla();
    }
}
```

Resim 3.4.7 : “cisim” Sınıfındaki Update Üye Fonksiyonu

Resim 3.4.7’deki "update" fonksiyonu, simülasyon içerisindeki tüm nesnelerin teorik altyapı kısmındaki bilgiler ve verilen denklemlerdeki fizik kurallarına göre güncellenmesini sağlamaktadır. Bu fonksiyon, "cisim" sınıfından türetilmiş tüm nesneler üzerinde sırayla işlem yaparak, onların kuvvet, ivme, hız ve konumlarını belirlenen sıralama ile güncellemektedir. Fonksiyon, önce "dynamic_cast" kullanarak yalnızca "cisim" sınıfına ait nesnelerle işlem yapılmasını sağlamaktadır, ardından "kuvvethesapla", "ivmehesapla", "hizhesapla" ve "konumhesapla" fonksiyonlarını sırasıyla çağırarak her nesne için fizik hesaplamalarının eksiksiz şekilde gerçekleştirilmesini sağlamaktadır. Bu süreç, tüm nesnelerin birbirleriyle etkileşimini hesaplamasına olanak tanımakta ve simülasyonun her adımda doğru bir şekilde ilerlemesine katkıda bulunmaktadır.

3.5 “roket” Sınıfı

```
#ifndef ROKETIMIZ_H
#define ROKETIMIZ_H

#include "cismimiz.cpp"

class roket : public cisim
{
public:
    static int roketsayisi;

    roket();
    roket(const double& mkutle, const vector2d& mkonum, const vector2d& mhiz,
          const vector2d& thrustDir, double fuelRate);
    virtual ~roket() {}
    roket& operator=(const roket& r);
    virtual void kuvvethesapla(purevirtual** arr, int n);
    virtual void kutlehesapla();
    virtual void ivmehesapla();
    virtual void hizhesapla();
    virtual void konumhesapla();
    virtual void update(purevirtual** arr, int n);

protected:
    double   wp;
    vector2d vp;
};

#endif
```

Resim 3.5.1: “roket” Sınıfı Kod Bloğu

Bu sınıf, simülasyon içinde hareket eden ve kendi itme kuvveti ile yönlendirilmesi gereken bir nesneyi proje özelinde bir roket olarak temsil ettiğinden kritik bir öneme sahiptir. "roket" sınıfı, "cisim" sınıfından türetilerek oluşturulmuş olup, kalıtım mekanizması sayesinde "cisim" sınıfının tüm temel fiziksel özelliklerini miras almakta, ancak ek olarak itme kuvveti ve yakıt tüketimi gibi roket dinamiğine özgü bileşenler içermektedir. Kalıtım şemasında en üst basamakta bulunan "purevirtual" soyut sınıfı, "cisim" sınıfı tarafından somutlaştırılmakta, "roket" sınıfı ise bu yapıyı genişleterek simülasyonun içindeki kalıtım mekanizmasının en alt tabakasındaki hareketli nesnelerinden biri haline gelmektedir. "public"(genel) kalıtım yöntemi kullanılarak, "cisim" sınıfının tüm açık fonksiyonları "roket" sınıfında da erişilebilir hale getirilmiş ve böylece çok biçimcilik mekanizmasının başarılı bir şekilde uygulanması sağlanmıştır. "roket" sınıfının en belirgin farklarından biri, "cisim" sınıfının yalnızca dış etkenlerle hareket eden pasif nesneleri temsil etmesine karşın, "roket" sınıfının itme kuvveti

üreterek kendi hareketini belirleyebilmesidir. Bu bağlamda, "roket" nesneleri için "kuvvethesapla" fonksiyonu genişletilerek, itme kuvvetinin hesaba katılması sağlanmıştır. Bunun yanı sıra, "kutlehesapla" fonksiyonu, yakıt tüketimi nedeniyle roketin kütlesinin zamanla azalmasını sağlamak üzere bastırılmıştır(override). "static" olarak tanımlanan "roketsayisi" değişkeni, simülasyonda bulunan toplam roket sayısını takip etmek için kullanılmaktadır ve her yeni roket oluşturulduğunda artırılmakta, yok edildiğinde azaltılmaktadır. "const" anahtar kelimesi ile tanımlanan parametreler, değiştirilemez olarak belirlenerek bellek verimliliği artırılmış ve gereksiz kopyalama işlemlerinin önüne geçilmiştir.

Çok biçimcilik açısından bakıldığında, "roket" nesneleri "cisim" veya "purevirtual" sınıfı türünden göstericiler ("pointer") ile yönetilebilmekte, ancak kendi özelleşmiş fonksiyonlarını çağırarak farklı bir hareket mekaniği sunabilmektedir. Böylece, simülasyonda bulunan tüm nesneler ortak bir yapı üzerinden yönetilebilirken, aynı zamanda roketler için itme kuvveti ve yakıt tüketimi gibi ek hesaplamalar gerçekleştirilebilmektedir.

```
roket& roket::operator=(const roket& r)
{
    if(this == &r)
        return *this;

    *(this->kutle)      = r.getkutle();
    *(this->konum)      = r.getkonum();
    *(this->hiz)        = r.gethiz();
    *(this->ivme)       = *(r.ivme);
    *(this->totalforce) = r.gettf();
    this->wp = r.wp;
    this->vp = r.vp;
    return *this;
}
```

Resim 3.5.2 : "roket" Sınıfı Operatörünün Aşırı Yüklenmesi

Resim 3.5.2'deki "operator=" aşırı yükleme fonksiyonu, bir "roket" nesnesinin başka bir "roket" nesnesine atanmasını (değer cinsinden) sağlamak amacıyla uygulanmıştır. C++ dilinde, nesneler varsayılan olarak sığ kopyalama (shallow copy) ile kopyalandığından, "operator=" fonksiyonu ile derin kopyalama (deep copy) işlemi gerçekleştirilmiştir. Bu fonksiyon,

kendisine atanan "roket" nesnesinin tüm üye değişkenlerini, hedef nesnenin üye değişkenlerine kopyalamaktadır. "this" işaretçisi kullanılarak, fonksiyonun çağrıldığı nesneye erişim sağlanmıştır. Bir nesnenin kendisine atanmasını önlemek için, "this" işaretçisinin parametre olarak verilen nesneyle aynı olup olmadığı kontrol edilmiştir. Bu, özellikle bellek yönetimi açısından önemli bir adımdır ve istenmeyen kopyalamaları önleyerek performansı artırmaktadır. Değişken atamalarında "*this->degisken" şeklinde erişim sağlanarak, heap bellekte tutulan veriler güncellenmiştir. Açıktır ki, bu aşırı yükleme işlemi, nesne yönelimli programlamada büyük bir esneklik sağlamakta ve "roket" nesnelerinin kopyalanmasını güvenli hale getirmektedir.

```
void roket::kutlehesapla()
{
    *(this->kutle) = *(this->kutle) - wp * dt;
    if(*(this->kutle) < 0)
        *(this->kutle) = 0;
}
```

Resim 3.5.3 : “roket” Sınıfı Kütle Hesapla Üye Fonksiyonu

$$m_R(t + \Delta t) = m_R(t) - w_P \Delta t \tag{3.d}$$

Resim 3.5.3’deki "kutlehesapla" fonksiyonu, "cisim" sınıfından türetildiği (cisim sınıfında boş fonksiyon olarak tanımladık) için bastırılmış ve roketin zamanla kütle kaybetmesini sağlamak amacıyla özelleştirilmiştir. Denklem 3.d’den de görülebileceği üzere bir roketin kütlesi sabit kalmaz; yakıt tüketimi nedeniyle zaman içinde azalır. Bu fonksiyon, "wp" değişkeni ile temsil edilen yakıt tüketim oranını "dt" zaman adımıyla çarparak, mevcut kütleden çıkarma işlemi yapmaktadır. Eğer kütle sıfırın altına düşerse, fiziksel olarak geçerli olmayan bir durum oluşacağından, kütle sıfıra sabitlenmektedir. Bu sayede, roketin negatif kütleye ulaşması gibi hatalı durumlar önlenmiş olur. Bu fonksiyon, simülasyonda dinamik bir kütle değişimini sağlamak için anahtar bir parametredir ve itki hesaplamalarının doğruluğu açısından büyük önem taşımaktadır.

```
void roket::kuvvethesapla(purevirtual** arr, int n)
{

    this->kutlehesapla();
    cisim::kuvvethesapla(arr, n);
    vector2d thrust = vp * wp;
    vector2d curF = this->gettf();
    vector2d newF = curF - thrust;
    this->settf(newF);
}
```

Resim 3.5.4 : “roket” Sınıfı Kuvvet Hesapla Üye Fonksiyonu

$$\overrightarrow{F_R} = -w_P \overrightarrow{v_P} \tag{3.e}$$

Resim 3.5.4’deki "kuvvethesapla" fonksiyonu, "cisim" sınıfında tanımlanan fonksiyonu bastırarak, roketin maruz kaldığı toplam kuvveti hesaplamaktadır. Denklem 3.e’den görüleceği üzere roketler yalnızca dış kuvvetlere değil, aynı zamanda kendi ürettikleri itme kuvvetine de tabidir. Öncelikle, roketin kütlesi "kutlehesapla" fonksiyonu ile güncellenir, ardından "cisim" sınıfının "kuvvethesapla" fonksiyonu çağrılarak yerçekimi ve diğer dış kuvvetlerin etkisi hesaplanır. Bundan sonra, roketin ürettiği itme kuvveti hesaplanır. "vp" değişkeni, roketin itme yönünü temsil ederken, "wp" değişkeni yakıt tüketim oranını ifade etmektedir. İtme kuvveti, bu iki değişkenin çarpımı ile elde edilmektedir. Son olarak, bu kuvvet mevcut net kuvvetten çıkarılmaktadır. Açıktır ki, bu işlem roketin kendi ürettiği kuvvet nedeniyle hareket etmesini sağlamakta ve simülasyonun gerçekçiliğini artırmaktadır.

```
void roket::ivmehesapla()
{
    cisim::ivmehesapla();
}
```

Resim 3.5.5 : “roket” Sınıfı İvme Hesapla Üye Fonksiyonu

Resim 3.5.5’deki "ivmehesapla" fonksiyonu, "cisim" sınıfında tanımlanmış olan fonksiyonu bastırarak, roketin toplam kuvvetten elde ettiği ivmeyi hesaplamak için kullanılmaktadır. Bu fonksiyon, "cisim" sınıfının "ivmehesapla" fonksiyonunu çağırarak mevcut ivmeyi hesaplamaktadır. Newton’un ikinci yasasına göre Denklem 3.a’dan da görüleceği üzere, ivme toplam kuvvetin kütleye bölünmesiyle elde edilir. Burada, "cisim" sınıfındaki hesaplamaların tekrar edilmesine gerek duyulmadan miras alınması sağlanmıştır. Roket karakteristiği olan yakıt püskürtme gibi olayların etkisi daha önce verilen “kuvvet hesapla” fonksiyonunda analiz edildiği için doğrudan cisim sınıfından çağırım geçerlidir. "this" işaretçisi ile doğrudan nesnenin ivme değişkenine erişilmiş ve hesaplamalar heap üzerinde gerçekleştirilmiştir. Bir gerçektir ki, roketin ivmesi yalnızca dış kuvvetlerden değil, aynı zamanda değişen kütlesinden de etkilenmektedir ve bu nedenle doğru hesaplanması simülasyonun fiziksel tutarlılığı açısından önem taşımaktadır.

```
void roket::hizhesapla()
{
    cisim::hizhesapla();
}
```

Resim 3.5.6 : “roket” Sınıfı Hız Hesapla Üye Fonksiyonu

Resim 3.5.6’daki "hizhesapla" fonksiyonu, "cisim" sınıfındaki fonksiyonun işlevini aynen koruyarak aşırı yüklenmiş ve roketin hızını güncellenen ivmeye göre belirlemek için kullanılmaktadır. Denklem 3.b’den de görüleceği üzere hız, ivmenin zaman ile çarpımı sonucu artmakta veya azalmaktadır. "cisim" sınıfında tanımlanan "hizhesapla" fonksiyonu, burada tekrar yazılmadan çağrılmış ve temel hız güncellemeleri gerçekleştirilmiştir. "this" işaretçisi aracılığıyla nesnenin hız bileşenlerine doğrudan erişilmiş ve güncellenen değerler heap bellekte

saklanmıştır. Cisim sınıfından bu fonksiyonun çağrılması ivmehesaplada bahsedilen sebeple aynıdır. Gerekli roket karakteristik özellikleri zaten kuvvethesapla fonksiyonunda verilmiştir.

```
void roket::konumhesapla()
{
    cisim::konumhesapla();
}
```

Resim 3.5.7: “roket” Sınıfı Konum Hesapla Üye Fonksiyonu

Resim 3.5.7’deki "konumhesapla" fonksiyonu, "cisim" sınıfından türetildiği için bastırılmış ve roketin hızına bağlı olarak konumunu güncellemek amacıyla tasarlanmıştır. "cisim" sınıfından miras alınan bu fonksiyon, Denklem 3.c’den de görüleceği üzere roketin mevcut hızını "dt" zaman adımı ile çarparak konum değişimini hesaplamaktadır. Böylece, roketin uzaydaki hareketi belirlenmektedir. Bir gerçektir ki, bu fonksiyonun doğru çalışması, simülasyonun görsel ve matematiksel doğruluğunu bizzat etkilemektedir.

```
void roket::update(purevirtual** arr, int n)
{
    cisim::update(arr, n);
}
```

Resim 3.5.8: “roket” Sınıfı Update Üye Fonksiyonu

Resim 3.5.8’deki "update" fonksiyonu, "cisim" sınıfındaki fonksiyonu bastırarak, roketin tüm fiziksel büyüklüklerini bir adım ileri taşımak için kullanılan bir fonksiyondur. Solid prensipleri gereği güncellemeler bu fonksiyonda yapılmıştır "cisim" sınıfının "update" fonksiyonunu çağırarak, kuvvet, ivme, hız ve konum hesaplamalarını sırasıyla gerçekleştirir. Bu fonksiyon, simülasyon döngüsü içinde her adımda çağrılarak roketin fiziksel özelliklerinin güncellenmesini sağlar. Açıktır ki, bu işlem olmadan simülasyon ilerleyemez ve tüm hesaplamalar geçersiz olur. "update" fonksiyonu, her nesnenin fizik kurallarına uygun olarak hareket etmesini garanti altına alarak, simülasyonun dinamik yapısını oluşturan en kritik bileşenlerden biri haline gelmektedir. Fonksiyonun aldığı "purevirtual** arr" ve "int n"

parametreleri, simülasyonda yer alan tüm nesneleri ve bunların toplam sayısını temsil etmektedir. Bu parametreler sayesinde, roket yalnızca kendi fiziksel durumunu güncellemekle kalmaz, aynı zamanda çevresindeki diğer nesnelerle olan etkileşimini de hesaplar. Bir gerçektir ki, simülasyondaki nesneler birbirleriyle sürekli olarak etkileşim halindedir ve bu etkileşimlerin doğru modellenmesi, fiziksel doğruluğun korunması açısından kritik bir rol oynamaktadır. Bu nedenle, "update" fonksiyonu yalnızca bireysel bir nesnenin hareketini belirlemekle kalmaz, aynı zamanda sistemin genel tutarlılığını sağlamak için tüm nesneleri tümsel olarak ele alır.

```
roket::roket(const double& mkutle, const vector2d& mkonum, const vector2d& mhiz,
             const vector2d& thrustDir, double fuelRate)
    : cisim(mkutle, mkonum, mhiz)
{
    roketsayisi++;
    vp = thrustDir;
    wp = fuelRate;
}
```

Resim 3.5.9: “roket” Sınıfı Kurucu Fonksiyonu

Resim 3.5.9’daki bu kurucu fonksiyon, "roket" nesnesi oluşturulduğunda çağrılmakta ve roketin fiziki özelliklerini kurmaktadır. "cisim" sınıfından türetildiği için, ilk olarak "cisim" sınıfının kurucu fonksiyonu çağrılmakta ve "mkutle", "mkonum" ve "mhiz" parametreleri bu temel sınıfa aktarılmaktadır. Böylece, "cisim" sınıfında tanımlanan belleksel işlemler ve fiziksel büyüklüklerin atanması gerçekleştirilerek roketin temel özellikleri oluşturulmuş olur. Ardından, "roketsayisi" değişkeni artırılarak simülasyondaki toplam roket sayısı güncellenmektedir. "vp" değişkeni, roketin itme yönünü belirtmek amacıyla "thrustDir" değişkenine atanmakta, "wp" değişkeni ise roketin yakıt tüketim oranını temsil eden "fuelRate" değerini saklamak üzere kullanılmaktadır. "vector2d" sınıfının kurucu fonksiyonları da çağrılarak, "mkonum", "mhiz" ve "thrustDir" değişkenleri uygun biçimde nesneye aktarılmaktadır.

3.6 “Universe” Sınıfı

```
#define UNIVERSE_H

#include "purevirtual.h"
#include "cismimiz.h"
#include "vector.h"
#include <map>


struct SpiralData {
    bool   started;
    double r;
    double theta;
};

class Universe
{
public:
    Universe();
    ~Universe();
    void insertCisim(double mass, const vector2d& pos, const vector2d& vel);
    void insertRoket(double mass, const vector2d& pos, const vector2d& vel,
                     const vector2d& thrustDir, double fuelRate);
    void createBlackHole(double mass, const vector2d& position, double radius);
    void step();
    purevirtual** getObjects() const;
    int getSize() const;
    void removeObject(purevirtual* obj);
private:
    // Linked List
    struct Node
    {
        purevirtual* object;
        Node* next;
    };
    Node* head;
    int size_;
    mutable purevirtual** objectCache;
    // Kara delik dataları
    bool    blackHoleExists;
    double  blackHoleMass;
    double  blackHoleRadius;
    vector2d blackHolePosition;
    std::map<purevirtual*, SpiralData> spiralMap;
private:
    void applyBlackHoleEffect();
    void applyLogSpiral(cisim* obj);
    Universe(const Universe&);
    Universe& operator=(const Universe&);
};
#endif
```

Resim 3.6.1: “Universe” Sınıfı Kod Bloğu

Resim 3.6.1’de gösterilen "Universe" sınıfı, simülasyon ortamındaki tüm nesnelerin yönetimini sağlayarak fiziksel etkileşimlerin kontrol edilmesi amacıyla tasarlanmıştır. Bu sınıf, uzay boşluğunda serbest hareket eden nesneleri takip etmek, güncellemek ve birbirleriyle olan

etkileşimlerini hesaplamak için geliştirilmiştir. Singüler nesnelerin modellenmesi yeterli değildir, çünkü sistemdeki cisimlerin toplu olarak nasıl bir dinamik oluşturdukları da önem taşımaktadır. "Universe" sınıfı, tüm nesneleri kapsayan bir yapı olarak hareket etmekte ve simülasyon sürecinin sistematik bir şekilde ilerlemesini sağlamaktadır. Sistemde yer alan tüm nesneler "purevirtual" safsanal temel sınıfından türetilmiş olup, bu sayede "cisim", "roket" veya "kara delik" gibi farklı fiziksel varlıkların ortak bir yapı üzerinden yönetilmesi sağlanmaktadır. Nesnelerin konumları sürekli olarak güncellenerek fizik kurallarına uygun bir hareket modeli oluşturulmaktadır.

Bu sınıfın temel bileşenlerinden biri de dinamik veri yapılarıdır. Bağlı liste (linked list) kullanımı, simülasyonun esnekliğini artırmakta ve nesnelerin dinamik olarak eklenip silinmesine olanak tanımaktadır. Önceki kısımlarda yapılan dizi olarak nesne saklanması yerine bağlı listeler cisim veya roket sayısının önceden belirlenme kısıtını ortadan kaldırmaktadır. "Node" adı verilen yapı, her nesneyi işaret eden bir gösterici içererek listenin diğer elemanlarıyla bağlantı kurmaktadır. Dinamik bellek yönetimi, "new" ve "delete" operatörleri ile gerçekleştirilmekte ve her nesne yalnızca ihtiyaç duyulduğunda bellekte yer kaplamaktadır.

Ek olarak, "std::map", logaritmik spiral hareketini takip etmek için her nesneye karşılık gelen parametreleri saklamak amacıyla kullanılmaktadır. "std::map", anahtar-değer çiftleriyle çalışan bir yapı olup, nesnelere hızlı erişim sağlayarak hesaplama süreçlerini optimize etmektedir. Bu yapı sayesinde, belirli bir nesnenin hareket ederken izlediği yörünge ve spiral parametrelerine anlık olarak erişim mümkün olmaktadır. Logaritmik spiral modeli, 2a, 2b, 2c ve 2d numaralı denklemlerle tanımlanan hareket kurallarına uygun bir şekilde çalışarak simülasyondaki nesnelerin karmaşık hareket dinamiklerini yönetmektedir.

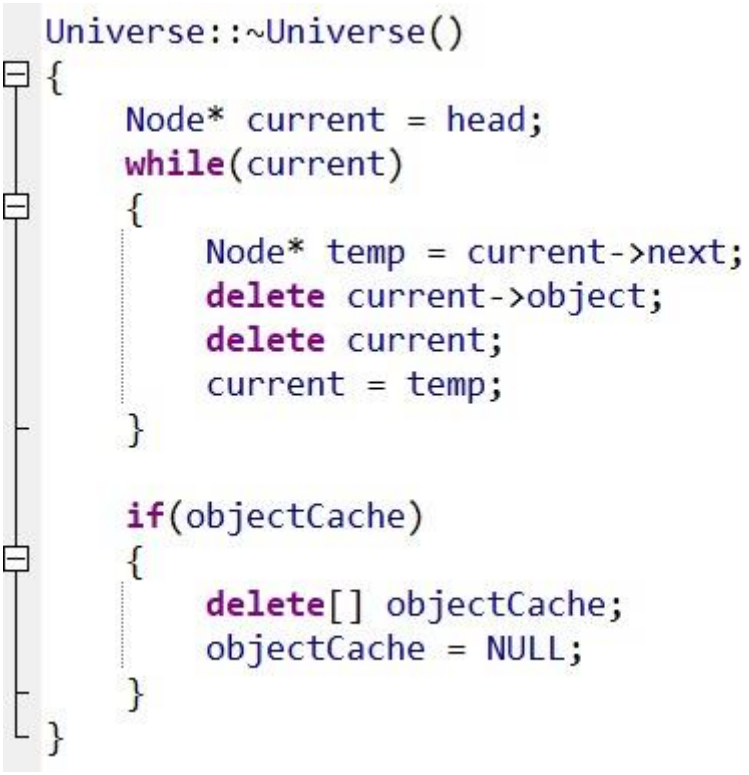
Bu modelleme yaklaşımı, simülasyondaki nesnelerin yalnızca doğrusal veya dairesel hareketlerle sınırlı kalmayıp, daha doğal ve gerçekçi bir şekilde ilerlemelerini sağlamaktadır. Logaritmik spiral hareketi, gök cisimlerinin yörüngeleri, galaksi kollarının dağılımı ve merkezcil kuvvetlere bağlı hareket modelleri gibi birçok fiziksel süreci modellemek için büyük önem taşımaktadır. "std::map" veri yapısının kullanımı, spiral hareketin parametrelerini düzenli bir şekilde yöneterek simülasyonun doğruluğunu artırmakta ve fiziksel gerçekçiliğini güçlendirmektedir.

```
Universe::Universe()
{
    head = NULL;
    size_ = 0;
    objectCache = NULL;

    blackHoleExists = false;
    blackHoleMass   = 0.0;
    blackHoleRadius = 0.0;
    blackHolePosition = vector2d(0,0);
}
```

Resim 3.6.2 : “Universe” Sınıfının Kurucu Fonksiyonu

Resim 3.6.2’deki "Universe" sınıfının kurucu fonksiyonu, evrenin başlangıç koşullarını belirleyen, simülasyonun temel başarımlarını düzenleyen ve tüm bileşenlerin eksiksiz bir şekilde derleyiciye sunulmasını sağlayan görece kritik bir yapıdır. Bu fonksiyon, simülasyon ortamı ilk oluşturulduğunda çağrılmakta ve bellek yönetimi açısından en optimize başlangıç değerlerini belirlemektedir. İlk olarak, simülasyonda var olan nesneleri tutacak olan "head" işaretçisi sıfırlanarak, herhangi bir nesnenin henüz dahil edilmediği bir boşluk yaratılmaktadır. Ardından, "size" değişkeni sıfırlanarak, sistemde başlangıçta hiç nesne bulunmadığı açıkça belirtilmektedir. Bunun yanında, "blackHoleExists" bayrağı(flag) "false" olarak ayarlanarak evrenin henüz bir kara delik tarafından şekillendirilmediği ortaya konulmaktadır. Ek olarak, "mutable" olarak tanımlanan "objectCache" değişkeni, simülasyonun performansını artıran bir önbellek mekanizması olarak kullanılmakta ve hızlı nesne erişimi sağlamak amacıyla özel olarak optimize edilmektedir. "std::map" veri yapısında tanımlanan "spiralMap" ise, evrende belirli bir logaritmik spiral hareketine tabi olan nesnelerin hareket bilgilerini saklamakta ve ilerleyen adımlarda ilgili nesnelerin yörünge hareketlerini takip etmek için kullanılmaktadır. Açıktır ki, kurucu fonksiyon, yalnızca nesneleri değil, aynı zamanda sistemin bellekte nasıl organize edileceğini de belirleyen ve evrenin tüm süreçlerini yönetecek olan ana yapının temellerini atan bir sistematiktir.



Resim 3.6.3: “Universe” Sınıfının Yıkıcı Fonksiyonu

Resim 3.6.3’teki yıkıcı(destructer) fonksiyon, "Universe" nesnesi silindiğinde çağrılan ve bellek yönetiminin sağlıklı bir şekilde gerçekleştirildiğini garanti altına alan bir fonksiyondur. Simülasyon boyunca heap bellekte dinamik olarak oluşturulan nesneler, zaman içinde birikerek belirli bir alanı işgal etmekte ve eğer bu nesneler uygun bir şekilde serbest bırakılmazsa görece önemli sayılabilecel bellek sızıntılarına neden olabilmektedir. Bu nedenle, yıkıcı fonksiyon, öncelikli olarak bağlı listeyi baştan sona tarayarak içerdiği her bir düğümü tek tek serbest bırakmaktadır. Bu işlem sırasında, "delete" operatörü kullanılarak her bir nesneye ayrılan bellek geri kazanılmakta ve sistem kaynaklarının boşa harcanması önlenmektedir. Ayrıca, "objectCache" sıfırlanarak, simülasyon sırasında önbellekte saklanan geçici verilerin gereksiz yere bellek kullanmasını engellemek için temizlenmektedir.

```
void Universe::insertCisim(double mass, const vector2d& pos, const vector2d& vel)
{
    cisim* yeni = new cisim(mass, pos, vel);

    Node* node = new Node;
    node->object = yeni;
    node->next = head;
    head = node;
    size_++;
}
```

Resim 3.6.4: “Universe” Sınıfının Cisim Ekleme Üye Fonksiyonu

Resim 3.6.4’teki görülen fonksiyon, evrenin içine yeni bir "cisim" ekleyerek simülasyona yeni nesneler kazandıran bir yapıdır. Fonksiyon, parametre olarak eklenmek istenen nesnenin kütlesini, konumunu ve hızını almakta ve bu değerler doğrultusunda yeni bir "cisim" nesnesi oluşturmaktadır. Ancak, bu nesneler statik olarak değil, heap bellekte dinamik olarak oluşturulmakta ve "new" operatörü aracılığıyla oluşturulmaktadır. Dinamik bellek yönetimi sayesinde, simülasyondaki nesne sayısı zamanla değişebilmekte ve bellek yalnızca gerektiği kadar kullanılmaktadır. Oluşturulan "cisim" nesnesi, bağlı listeye yeni bir düğüm olarak eklenmekte ve böylece sistemin içindeki varlıklar arasında yerini almaktadır. Açıktır ki, "insertCisim()" fonksiyonu, evrenin sürekli genişleyebilmesini sağlayan, yeni gök cisimlerinin veya hareketli nesnelerin simülasyona dahil edilmesine imkan tanıyan kritik bir mekanizmadır.

```
void Universe::insertRoket(double mass, const vector2d& pos, const vector2d& vel,
                           const vector2d& thrustDir, double fuelRate)
{
    roket* yeni = new roket(mass, pos, vel, thrustDir, fuelRate);

    Node* node = new Node;
    node->object = yeni;
    node->next = head;
    head = node;
    size_++;
}
```

Resim 3.6.5 : “Universe” Sınıfının Roket Ekleme Üye Fonksiyonu

Resim 3.6.5’deki bu fonksiyon, "insertCisim()" fonksiyonuyla benzerlik göstermesine rağmen, içerdiği ek parametreler sayesinde simülasyona farklı bir dinamik kazandırmaktadır bu ek parametreler roket ve cisim maddelerini roket karakteristikleri ile birbirinden ayırmaktadır.

"insertRoket()", klasik bir "cisim" nesnesinin sahip olduğu kütle, konum ve hız bilgilerine ek olarak, roketin itme yönünü belirleyen bir "thrustDir" vektörü ve yakıt tüketim oranını ifade eden bir "fuelRate" değişkeni almaktadır. Bir gerçektir ki, klasik cisimler yalnızca dış kuvvetlere bağlı olarak hareket ederken, roket nesneleri kendi ürettikleri kuvvet ile belirli bir yön doğrultusunda hızlanabilmekte ve böylece sistemde aktif olarak yönlendirilmesi gereken nesneler olarak işlev görmektedirler. Oluşturulan "roket" nesnesi de aynen "cisim" nesnesinde olduğu gibi heap bellekte dinamik olarak oluşturulmakta ve bağlı listeye eklenmektedir.

Karadelik

```
void Universe::createBlackHole(double mass, const vector2d& position, double radius)
{
    blackHoleExists   = true;
    blackHoleMass     = mass;
    blackHolePosition = position;
    blackHoleRadius   = radius;
}
```

Resim 3.6.6 : “Universe” Sınıfının Kara Delik Yaratma Üye Fonksiyonu

"Kara delikler, uzay-zamanın en derin sırlarını barındıran kozmik girdaplardır; öyle ki, Dünya'yı bir kara deliğe dönüştürmek isteseydik, yalnızca bir küp şeker boyutunda olurdu." (Kaynak: Barış Özcan - Spagettileştiren Kara Delikler)

Resim 3.6.6’daki "createBlackHole()" fonksiyonu, belirli bir konumda kara delik oluşturulmasını sağlayarak simülasyona derinlik ve dramatik etki eklemektedir. Bu fonksiyon, verilen konum ve yarıçap bilgileri doğrultusunda bir kara delik yaratmakta, "blackHoleExists" değişkenini "true" olarak işaretleyerek evrende aktif bir kara delik bulunduğunu derleyiciye sunmaktadır. Oluşturulan kara deliğin kütlesi ve çapı saklanarak, ilerleyen adımlarda etrafındaki nesneler üzerinde uygulanacak olan çekim kuvvetinin hesaplanmasına olanak tanımaktadır. Bir gerçektir ki, kara deliklerin fiziksel etkileri, simülasyondaki nesneler için hatrı sayılır değişiklikler yaratmakta ve zaman içinde bir noktaya doğru kaçınılmaz bir çekim oluşturmaktadır

```
void Universe::removeObject(purevirtual* obj)
{
    Node* prev = NULL;
    Node* curr = head;

    while (curr)
    {
        if (curr->object == obj)
        {
            if (prev)
                prev->next = curr->next;
            else
                head = curr->next;

            delete curr->object;
            delete curr;
            size_--;
            return;
        }
        prev = curr;
        curr = curr->next;
    }
}
```

Resim 3.6.7: “Universe” Sınıfının Nesne Çıkarma Üye Fonksiyonu

Resim 3.6.7’deki bu fonksiyon, bağlı listede bulunan herhangi bir nesneyi silmek için tasarlanmıştır. Liste boyunca ilerlenerek, silinmek istenen nesneye ulaşıldığında, bağlı liste yeniden düzenlenmekte ve ilgili nesne heap bellekte serbest bırakılmaktadır. Eğer silinmek istenen nesne listenin başında ise, "head" işaretçisi bir sonraki düğümü göstermekte; değilse, önceki düğümün "next" işaretçisi değiştirilerek listenin bağlantıları korunmaktadır. "size_" değişkeni bir azaltılarak nesne sayısı güncellenmektedir. Bu mekanizma olmadan simülasyon dinamik olarak değiştirilemez ve nesneler yalnızca eklendikleri yerde sabit kalmak zorunda olacağı açıktır.

```
void Universe::step()
{
    purevirtual** arr = getObjects();

    Node* iter = head;
    for(int i = 0; i < size_; i++)
    {
        iter->object->update(arr, size_);
        iter = iter->next;
    }


    applyBlackHoleEffect();
}
```

Resim 3.6.8 : “Universe” Sınıfının Step Üye Fonksiyonu

Resim 3.6.8’deki bu fonksiyon, simülasyonun zaman içinde ilerlemesini sağlayan ana kontrol mekanizmasıdır. "getObjects()" fonksiyonu çağrılarak mevcut tüm nesneler bir diziye aktarılmakta, ardından bağlı liste üzerinde ilerlenerek her nesne için "update()" fonksiyonu çağrılmaktadır. Her nesne, bu süreçte kuvvet, ivme, hız ve konum hesaplamalarını gerçekleştirerek 2a, 2b, 2c ve 2d numaralı denklemler doğrultusunda hareket etmektedir. Eğer kara delik mevcutsa, "applyBlackHoleEffect()" fonksiyonu çağrılarak kara deliğin çekim alanındaki nesnelere uygulanacak etkileşimler belirlenmektedir.

```
void Universe::applyBlackHoleEffect()
{
    if (!blackHoleExists) return;

    Node* iter = head;
    while (iter)
    {
        cisim* obj = dynamic_cast<cisim*>(iter->object);
        if (obj)
        {

            vector2d diff = obj->getkonum() - blackHolePosition;
            double dist = diff.vmagnitude();

            if (dist < blackHoleRadius)
            {

                if (spiralMap.find(obj) == spiralMap.end())
                {
                    SpiralData sd;
                    sd.started = true;
                    sd.r = dist;
                    sd.theta = std::atan2(diff.gety(), diff.getx());
                    spiralMap[obj] = sd;
                }

                applyLogSpiral(obj);
            }
            else
            {

                if (spiralMap.find(obj) != spiralMap.end())
                {
                    spiralMap.erase(obj);
                }
            }
        }
        iter = iter->next;
    }
}
```

Resim 3.6.9 : “Universe” Sınıfının Kara Delik Analizör Üye Fonksiyonu

Resim 3.6.9’daki bu fonksiyon, simülasyonda bulunan kara deliğin çevresindeki nesnelere uyguladığı fiziksel etkiyi hesaplayarak bu nesnelerin hareketlerini değiştirmektedir. Eğer bir kara delik oluşturulmamışsa, "blackHoleExists" değişkeni "false" olarak ayarlanmış olacağından, fonksiyonun başında yapılan kontrol ile herhangi bir işlem yapılmadan fonksiyondan çıkılmaktadır. Kontrol, gereksiz hesaplamaları engelleyerek performans kaybını önlemektedir bu kod performansı için projedeki en önemli işlevlerden biridir. Eğer bir kara delik var ise, fonksiyonun içerisinde "head" işaretçisi kullanılarak bağlı liste üzerinde bir döngü başlatılmakta ve simülasyonda bulunan her nesne tek tek kontrol edilmektedir.

Her düğüm(node) üzerinde bulunan nesneye erişebilmek için, düğümün içinde tutulan "object" işaretçisi kullanılmaktadır. Ancak "object" göstericisi "purevirtual" türünden

tanımlandığından ve kara delik etkileşimi yalnızca "cisim" sınıfından türetilen nesneler üzerinde gerçekleştirildiğinden, burada "dynamic_cast" operatörü kullanılarak nesnenin "cisim" olup olmadığı kontrol edilmektedir. Eğer nesne bir "cisim" değilse, fonksiyon o nesne üzerinde herhangi bir işlem yapmadan bağlı listedeki bir sonraki düğüme geçmektedir. Açıktır ki, "dynamic_cast" operatörü, nesne yönelimli programlamada türetilmiş sınıfların güvenli bir şekilde dönüştürülmesini sağlamakta ve yalnızca uygun türdeki nesnelerin işlenmesine izin vermektedir.

Eğer nesne bir "cisim" olarak tanımlanmışsa, öncelikle "getkonum()" fonksiyonu çağrılarak nesnenin uzay içerisindeki mevcut konumu elde edilmekte ve bu konum, kara deliğin merkezi olan "blackHolePosition" değişkeninden çıkarılarak, nesne ile kara delik arasındaki vektörel mesafe belirlenmektedir. Bu vektör "diff" olarak adlandırılmakta ve nesnenin kara deliğe olan yönü ile mesafesi hakkında bilgi vermektedir. Elde edilen mesafe vektörünün büyüklüğü, "vmagnitude()" fonksiyonu kullanılarak hesaplanmakta ve böylece nesnenin kara deliğe olan skaler uzaklığı "dist" değişkenine atanarak belirlenmektedir. Eğer "dist" değeri, "blackHoleRadius" değişkeninden küçükse, yani nesne kara deliğin etkili alanı içerisindeyse, o nesne artık doğrudan kara deliğin çekimine maruz kalmaktadır.

Eğer bu nesne kara deliğe zaten yakalanmışsa, yani daha önce spiral harekete girmişse, "spiralMap" veri yapısında bu nesneye ait bir giriş bulunacaktır. Ancak eğer bu nesne ilk defa kara deliğin çekim alanına giriyorsa, "spiralMap" veri yapısı içerisinde bu nesneye ait bir kayıt bulunmayacaktır. Açıktır ki, bu durum kontrol edilerek, nesne daha önce yakalanmamışsa yeni bir "SpiralData" nesnesi oluşturulmakta ve nesne bu veri yapısına eklenmektedir. "SpiralData" veri yapısında, nesnenin başlangıçta spiral harekete geçtiği andaki uzaklığı "r" olarak saklanmakta ve nesnenin açısal konumu "atan2" fonksiyonu ile hesaplanarak "theta" değişkenine atanarak belirlenmektedir.

Eğer nesne spiral harekete girmeye uygunsa, "applyLogSpiral()" fonksiyonu çağrılarak nesnenin hareketi güncellenmektedir. Eğer nesne kara deliğin etki alanının dışına çıkmışsa, "spiralMap" veri yapısında bu nesneye ait kayıt silinerek, nesnenin normal hareketine devam etmesi sağlanmaktadır. Bir gerçektir ki, bu fonksiyon yalnızca nesneleri kara deliğin içine doğru

çekmekle kalmamakta, aynı zamanda onların fiziksel olarak doğruluğa uygun bir hareket sergilemelerini de sağlamaktadır.

```
void Universe::applyLogSpiral(cisim* obj)
{
    SpiralData& sd = spiralMap[obj];

    double k = 0.5;
    double w = 2.0;

    // dt: vector.h içindeki sabit zaman adımı
    sd.r     = sd.r * std::exp(-k * dt);
    sd.theta = sd.theta + w * dt;

    double newX = blackHolePosition.getx() + sd.r * std::cos(sd.theta);
    double newY = blackHolePosition.gety() + sd.r * std::sin(sd.theta);


    obj->setkonum(vector2d(newX, newY));


}
```

Resim 3.6.10: “Universe” Sınıfının Spiral Analizör Üye Fonksiyonu

Resim 3.6.10’daki bu fonksiyon, kara deliğe yakalanan nesnelerin yalnızca doğrusal bir şekilde merkeze düşmemesini, bunun yerine belirli bir spiral yörünge izleyerek içeriye doğru çekilmelerini sağlamaktadır. Bu hareket, logaritmik spiral olarak adlandırılmakta ve özellikle kara delik etrafında oluşan girdap benzeri hareketleri modellemek için oldukça uygundur. 2a ve 2b numaralı denklemlerle nesnenin spiral hareketi doğrultusunda yeni konumu belirlenmektedir. Aslında gerçek varlıkta karadelik hareketleri bu eğri ile modellenmez ancak simülasyon düzeyi ve yaklaşımları ile gerçeğe log-spiral bir hareket ile yakınsama yapılmıştır.

Fonksiyon, "applyBlackHoleEffect()" tarafından çağrıldığında, spiral harekete girmiş olan nesne "spiralMap" üzerinden çekilerek, ona ait "SpiralData" nesnesi referans olarak alınmaktadır. Bu veri yapısı içinde, nesnenin kara deliğe olan uzaklığı "r" ve açısal konumu "theta" değişkenleri ile saklanmaktadır. Spiral hareketin zaman içerisinde nasıl değiştiğini belirlemek için iki temel parametre kullanılmaktadır: "k" ve "w". "k" değeri, nesnenin kara deliğe doğru ne kadar hızlı çekildiğini belirleyen bir sönümlenme katsayısıdır ve negatif üstel fonksiyon yardımıyla nesnenin giderek içe doğru çekilmesini sağlamaktadır. "w" değeri ise

nesnenin açısal hızını belirlemekte ve spiral hareketin nasıl bir yayılım gösterdiğini ifade etmektedir.

Bu iki değer kullanılarak nesnenin uzaklığı “r(t+dt) = r(t) * e^{-k dt}” ifadesiyle güncellenmektedir. Bu denklem, nesnenin zaman içinde kara deliğe yaklaşmasını ifade eden üstel bir çarpan içermektedir. Açıktır ki, bu formül sayesinde nesne, zaman ilerledikçe giderek daha küçük bir yarıçapta spiral bir yörünge çizmektedir.

Nesnenin açısal konumu ise “theta(t+dt) = theta(t) + wdt” ifadesiyle hesaplanmaktadır. Burada açısal hız sabit kabul edilmekte ve zaman ilerledikçe nesnenin belirli bir açısal hızda kara deliğin çevresinde döndüğü varsayılmaktadır.

Bu iki hesaplama yapıldıktan sonra, nesnenin yeni konumu Kartezyen koordinat sistemine çevrilerek “x = blackHolePosition.x + r * cos(theta) ve y = blackHolePosition.y + r * sin(theta)” denklemleri yardımıyla belirlenmektedir. Burada, nesne yeni konumuna "setkonum()" fonksiyonu çağrılarak taşınmaktadır.

Bu fonksiyonun sonunda, eğer nesne kara deliğin merkezine çok fazla yaklaşmışsa ve belirli bir eşik değerin altına düşmüşse, nesne "removeObject()" fonksiyonu ile sistemden silinerek tamamen kara delik tarafından yutulmuş olarak kabul edilmektedir. Eğer böyle bir işlem gerçekleşirse, "spiralMap" veri yapısından da bu nesneye ait kayıt silinmekte ve böylece sistemde artık o nesneye ait herhangi bir bilgi bulunmamaktadır.

"applyLogSpiral()" fonksiyonu, yalnızca nesneleri içeriye doğru çekmekle kalmamakta, aynı zamanda onların fiziksel olarak gerçekçi bir yörünge izlemelerini sağlayarak simülasyonun doğruluğunu artırmaktadır. Eğer bu fonksiyon olmasaydı, kara deliğe yaklaşan her nesne doğrudan merkeze düşerdi ve bu durum fiziksel olarak tutarsız bir sonuç yaratırdı. Ancak, bu spiral modelleme sayesinde, nesneler tıpkı gözlemlenen kara delik disklerinde olduğu gibi, merkeze doğru hareket ederken bir yandan da belirli bir açısal hızla dönmektedir.

```
purevirtual** Universe::getObjects() const
{
    if(objectCache)
    {
        delete[] objectCache;
        objectCache = NULL;
    }

    objectCache = new purevirtual*[size_];

    Node* iter = head;
    int i = 0;
    while(iter)
    {
        objectCache[i++] = iter->object;
        iter = iter->next;
    }

    return objectCache;
}

int Universe::getSize() const
{
    return size_;
}
```

Resim 3.6.11 : “Universe” Sınıfının Nesne Döndürücü Üye Fonksiyonu

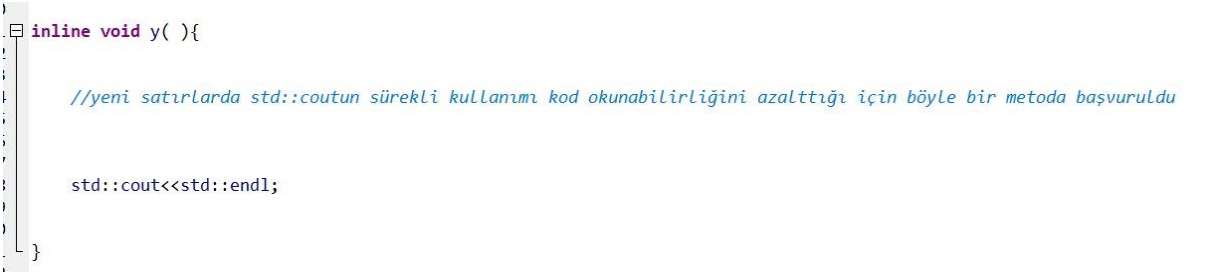
Resim 3.6.11’deki "getObjects()" fonksiyonu, simülasyonda bulunan tüm nesnelerin bir dizisini döndürerek, diğer fonksiyonların nesnelere erişmesini sağlayan bir yapıdır. Öncelikle, "objectCache" değişkeninin önceden oluşturulmuş olup olmadığı kontrol edilmekte ve eğer bellekte bir önceki çağrıdan kalan bir dizi mevcutsa, "delete[]" operatörü kullanılarak serbest bırakılmakta ve "NULL" olarak sıfırlanmaktadır; bu işlem, bellek sızıntılarını önlemek için öneme sahiptir. Ardından, simülasyonda bulunan nesne sayısı kadar yeni bir "purevirtual*" dizisi heap bellekte dinamik olarak oluşturulmakta ve bu dizi, simülasyondaki nesneleri işaret eden bir önbellek mekanizması olarak kullanılmaktadır. Daha sonra, bağlı listenin başındaki "head" işaretçisi ile bir "Node" nesnesi alınmakta ve "while" döngüsü kullanılarak liste boyunca ilerlenerek her bir nesne "objectCache" dizisine eklenmektedir. "i" değişkeni, dizinin indeksini takip etmek için kullanılmakta ve her adımda bir artırılmaktadır. Döngü tamamlandığında, simülasyondaki tüm nesneleri içeren "objectCache" dizisi döndürülerek, diğer fonksiyonların

bu nesnelere erişmesini sağlayan bir yapı sunulmaktadır bu mekanizma olmadan simülasyon içerisindeki nesnelere verimli bir şekilde erişmek mümkün olmazdı veya farklı bir erişim mekanizm. "getSize()" fonksiyonu ise görece basit bir işleve sahip olup, simülasyonda bulunan nesne sayısını temsil eden "size_" değişkeninin değerini döndürmektedir. Bu fonksiyon, simülasyonda kaç nesne bulunduğunu öğrenmek isteyen diğer fonksiyonlar tarafından çağrılmakta ve "size_" değişkeninin doğrudan geri döndürülmesiyle oldukça düşük maliyetli bir işlem gerçekleştirilmektedir. Bir gerçektir ki, "getObjects()" ve "getSize()" fonksiyonları, simülasyonun düzgün bir şekilde yönetilebilmesi için gerekli olan temel yapı taşlarından biridir; biri nesneleri doğrudan erişime açarken, diğeri bu nesnelerin sayısını belirleyerek dinamik yapıyı kontrol altında tutmaktadır.

3.7 “Ek Fonksiyonlar” Kütüphanesi

Bu kısımda proje isterlerinden biri olan projeye eklenen ek özelliklerin analizi için oluşturulmuş bir kütüphane incelenecektir.

"ekfonksiyonlar.h" dosyasında inline fonksiyonların tercih edilmesinin temel sebebi, fonksiyon çağrılarında oluşan zaman kaybını azaltarak simülasyonun performansını artırmaktır. Normal bir fonksiyon çağrısında, program yürütme sürecinde çağrı yığını kullanılarak fonksiyonun adresine bir atlama yapılır ve dönüş işlemi gerçekleşir. Ancak inline fonksiyonlarda, derleyici fonksiyon gövdesini çağrıldığı yere doğrudan yerleştirerek bu ek işlemleri ortadan kaldırır. Bu yöntem, özellikle sık kullanılan ve kısa fonksiyonlarda hız kazandırarak işlem süresini optimize etmektedir. Bellek açısından bakıldığında, inline fonksiyonlar çağrı yığını kullanımını azaltarak yığın yönetimini daha verimli hale getirir. Ancak büyük fonksiyonların inline yapılması, kod tekrarına yol açarak derlenen dosyanın boyutunu artırabilir ve kod şişmesi (code bloat) *(Kaynak: https://www.mdpi.com/2227-7390/11/17/3744)* sorununa neden olabilir. Açıktır ki, "ekfonksiyonlar.h" içindeki inline fonksiyonlar, özellikle temel matematiksel işlemler gibi sık kullanılan hesaplamalarda performans kazanımı sağlamak amacıyla bilinçli bir şekilde tercih edilmiştir.



Resim 3.7.1: “Ek Fonksiyonlar” Kütüphanesinin Yeni Satır Fonksiyonu

Resim 3.7.2: “Ek Fonksiyonlar” Kütüphanesinin Oluşturum Fonksiyonu

Resim 3.7.2’deki "create()" fonksiyonu, simülasyonun son aşamasında nesnelerin oluşturulmasını sağlayan ve kullanıcıdan aldığı veriler doğrultusunda cisimleri, roketleri ve kara deliği evrene ekleyen oluşturum parametreleri ile çalışan oluşturulmuş bir bileşendir. Inline olarak tanımlanmış olması, fonksiyon çağrı maliyetini ortadan kaldırarak çalışma süresini optimize etmekte ve özellikle sık kullanılan hesaplama bloklarında performans artışı sağlamaktadır. Kullanıcı girdilerine bağlı olarak farklı türde nesnelerin oluşturulması gerektiğinden, fonksiyon içerisinde döngü yapıları kullanılarak birçok nesnenin ardışık bir şekilde heap belleğe dinamik olarak tahsis edilmesi sağlanmaktadır. Her yeni "cisim" ve "roket", bağlı liste yapısında organize edilerek "insertCisim()" ve "insertRoket()" fonksiyonları aracılığıyla evrene dahil edilmekte, bu sayede bellek yönetimi verimli bir şekilde gerçekleştirilmekte ve nesneler sistemden gerektiğinde kolayca kaldırılabilmektedir. Kullanıcının tercihine bağlı olarak simülasyona bir kara delik eklenebilmesi, sistemin esnekliğini artırmakta ve üç cisim probleminin ötesine geçen dinamik çekim kuvvetlerini modelleme imkanı sunmaktadır. Fonksiyonun son basamakta çalışması, simülasyona dahil edilecek nesnelerin belirlenmesi ve kullanıcı girdilerinin alınarak işlenmesi açısından önemli bir oluşturum adımı olarak karşımıza çıkmaktadır. Kullanıcı girdileri doğrudan nesnelerin fiziksel parametrelerine çevrildiğinden, sistemin bağımlılıkları en aza indirgenmiş, kod modüler bir hale getirilmiş ve SOLID prensipleri (Kaynak *: https://www.geeksforgeeks.org/solid-*

principle-in-programming-understand-with-real-life-examples/) Doğrultusunda genişletilebilir bir yapı oluşturulmuştur. Örneğin, bağlı liste yapısının kullanılması, yeni nesnelerin eklenmesini veya var olan nesnelerin silinmesini yüksek performanslı bir hale getirmekte ve böylece bellek verimliliği artırılmaktadır. Aynı zamanda, kullanıcıya önerilen kara delik parametrelerinin dahil edilmesiyle birlikte sistemin ön tanımlı varsayılan değerlerle çalışabilmesi sağlanmış, manuel giriş ihtiyacı ortadan kaldırılarak kullanım kolaylığı sunulmuştur. Açıktır ki, bu fonksiyon olmadan, simülasyon dinamik bir şekilde şekillendirilemez, nesneler organize edilemez ve oluşturum süreci tamamlanamazdı. Sonuç olarak, "create()" fonksiyonu üç cisim probleminin son aşamasındaki ana oluşturum süreci olup, simülasyonun başlangıcında kullanıcı girdileriyle evrenin dinamik bileşenlerinin belirlenmesini sağlamakta ve sistemin bellek, performans ve kod yönetimi açısından verimli çalışmasına olanak tanımaktadır.

3.7 Ana Fonksiyon

Bu kod, simülasyonun yürütüldüğü ana kontrol mekanizmasını oluşturarak nesneleri yönetmek, fiziksel hesaplamaları gerçekleştirmek ve görselleştirmek amacıyla çalışmaktadır. Programın başlangıcında grafik çizim kütüphanesi kullanılarak "canvas" sınıfından bir nesne oluşturulmakta ve "startDoc()" ile çizim alanı başlatılmaktadır. Ardından, "Universe" sınıfından bir nesne türetilerek simülasyon ortamı oluşturulmaktadır. Kullanıcı girişine dayalı olarak nesnelerin eklenmesi için "create()" fonksiyonu çağrılmakta ve tüm nesneler heap bellekte dinamik olarak tahsis edilerek bağlı liste yapısı içinde organize edilmektedir. Simülasyon, 60000 adımlık bir iterasyon sınırıyla çalışacak şekilde tasarlanmış olup, bu değer ilerleyen analizlerde farklı parametreler için test edilecektir.

Ana döngüde, "evren.getObjects()" ile nesneler çekilmekte ve "getSize()" ile nesne sayısı belirlenmektedir. Her nesne, "graphic.drawPoint()" ile uygun renkte çizdirilmekte ve "evren.step()" çağrılarak kuvvet, ivme, hız ve konum hesaplamaları güncellenmektedir. Açıktır ki, "step()" fonksiyonu olmadan fizik motoru çalışmayacak ve simülasyon ilerleyemeyecektir. Döngü tamamlandığında, "graphic.finishDoc()" çağrılarak sonuç görselleştirilmekte ve HTML tabanlı çıktı üretilmektedir.

Bu yapı, modüler programlama prensiplerine uygun olarak, bağımsız bileşenlerin birlikte çalışmasını sağlamakta ve heap bellek, bağlı liste yönetimi, gibi tekniklerle performans açısından verimli bir model oluşturmaktadır. Daha başarılı bir analiz için, çıktılar ve parametrelerin simülasyon üzerindeki etkisi bir sonraki aşamada incelenecektir. Özellikle iterasyon sınırının farklı parametreler için nasıl değiştiği ve simülasyon stabilitesine etkisi detaylandırılacaktır.

4. Fiziksel Gerçekliğin Dijital Temsili: Çıktıların Derinlemesine İncelenmesi

4.1 Üç Cisim Analizi

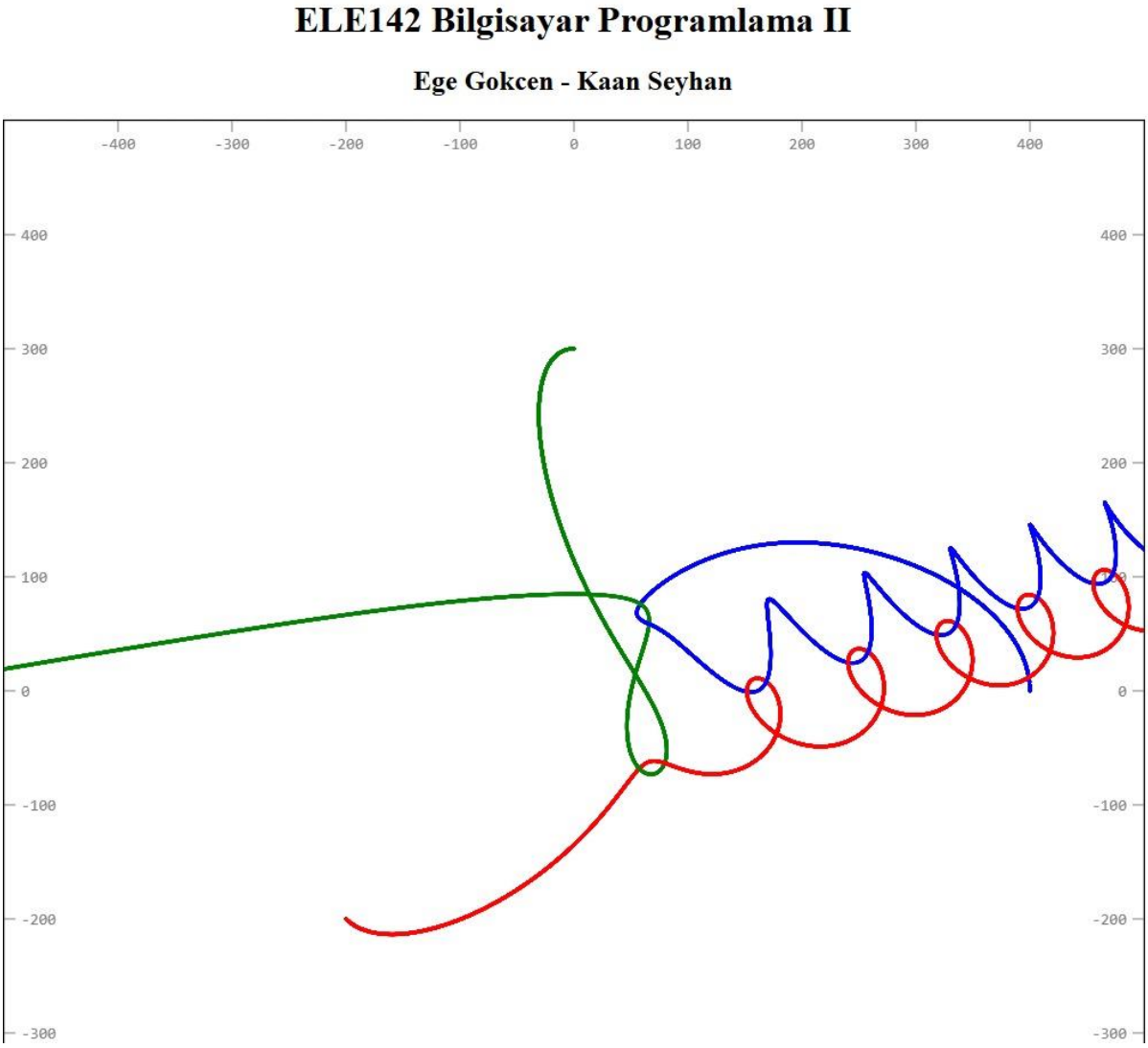
Bu kısımda proje evrağındaki klasik ister olan üç tane cisimin serbest uzayda yaptığı hareketi nümerik olarak simüle edip kapalı matematik form çıktılarına yakınsama yapılacaktır. Önceki kısımlarda bahsedilen matematik çizim kütüphanesi ile bu yakınsama görselleştirilecektir.

```
evren.insertCisim(150, vector2d(400, 0), vector2d(0, 0.2));
evren.insertCisim(70,  vector2d(0, 300), vector2d(-0.15, 0));
evren.insertCisim(150, vector2d(-200, -200), vector2d(0.1, -0.1));
```

Resim 4.1.1: Proje Evrağında Verilen Değerler ile Üç Cisimin Oluşturumu

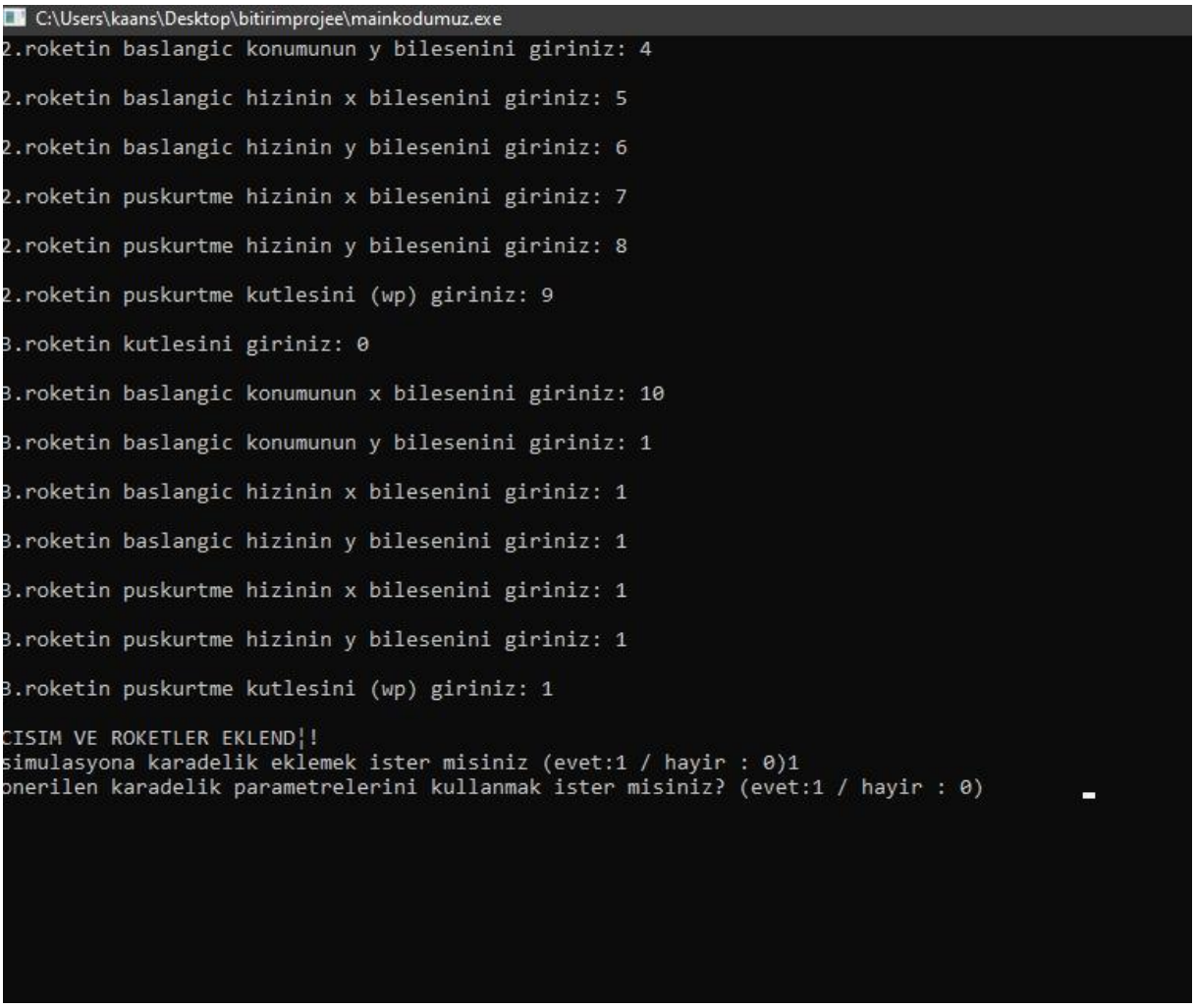
Projenin ana fonksiyonu incelendiğinde tasarım girdilerinin kullanıcıdan iki farklı şekilde alınabildiği görülür. İlki proje evrağında istenilen ve ek analiz olarak kabul edilebilecek terminal ekranında bilgi edinimidir, diğeri ise doğrudan kurucu fonksiyona parametre olarak girilebilecek değerlerdir.

Okunabilirlik açısından burada ikinci yöntem tercih edilmiştir.



Resim 4.1.2: Proje Evrağında Verilen Değerler ile Oluşturulan Üç Cisimin Grafiksel Çıktısı

Resim 4.1.2'deki yörüngeler, üç cismin birbirleriyle olan karmaşık etkileşimlerinin doğrudan bir sonucu olarak gözlemlenmiştir. Cisimler, kütleçekim kuvvetleri nedeniyle sürekli olarak hız ve yön değiştirerek, öngörülemez ve düzensiz bir hareket sergilemiştir. Özellikle yakın geçişler sırasında, ivmelerindeki ani değişimler yörüngelerin beklenmedik şekilde sapmasına yol açmıştır. Bu kaotik dinamikler, sistemde belirli bir düzen yerine, giderek karmaşıklaşan ve iç içe geçen yolların oluşmasına neden olmuştur. Hareketin bu şekilde gelişmesi, üç cisim problemine özgü belirlenemezliğin açık bir göstergesidir.

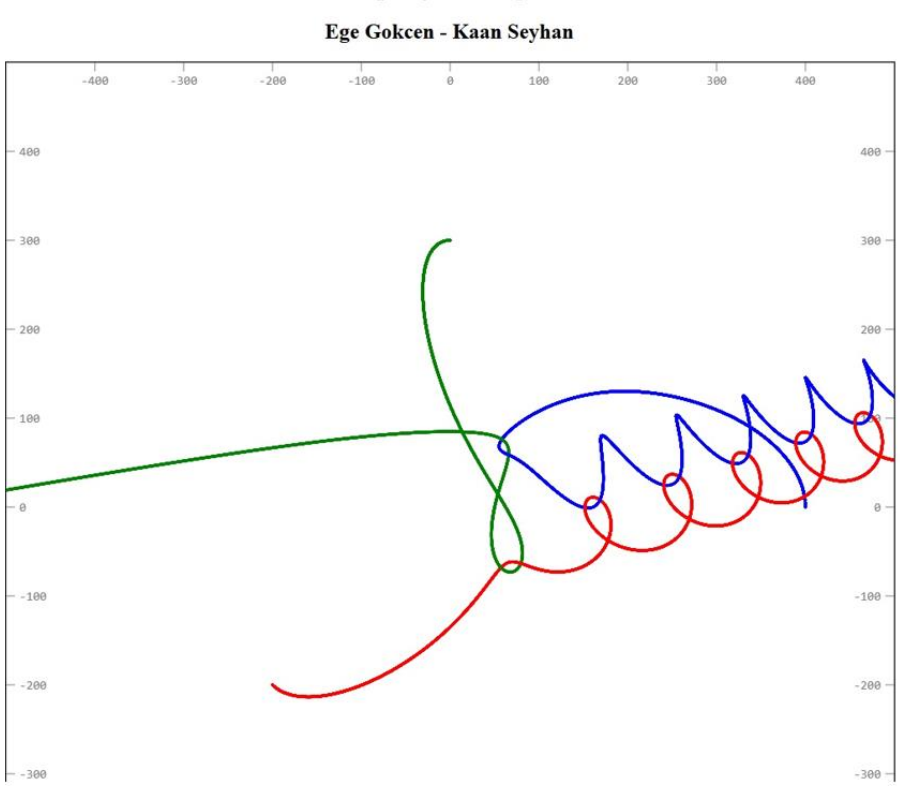


Resim 4.1.3: Roket ve Cisimlerin Terminal Ekranından Oluşturulması

Resim 4.1.3'ten görüleceği üzere roket ve cisim için nesneler gerekli parametre değerleri ile terminal ekranı üzerinden oluşturulabilmektedir. Bu nesneler için istenilen değerler girildikten sonra ek analiz olarak oluşturulan kara delik özelliğinin kullanılması için bir soru ekrana yansımaktadır. Eğer kara delik kullanılmak isteniyorsa öncelikle kara delik için önerilen değerlerin kullanıp kullanılmayacağı sorusu ekranda belirmektedir. Eğer karadeliğin önerilen özelliklerinin kullanılması istenmiyorsa karadelik için istenilen parametrelerin girilmesi gerekilmektedir.

```
evren.insertCisim(150, vector2d(400, 0), vector2d(0, 0.2));
evren.insertRoket(70,  vector2d(0, 300), vector2d(-0.15, 0),vector2d(0,0),0);
evren.insertRoket(150, vector2d(-200, -200), vector2d(0.1, -0.1),vector2d(0, 0),0);
```

Resim 4.1.4: Roket'in Parametrelerine 0 Değeri Girilerek Cisime Benzetimi

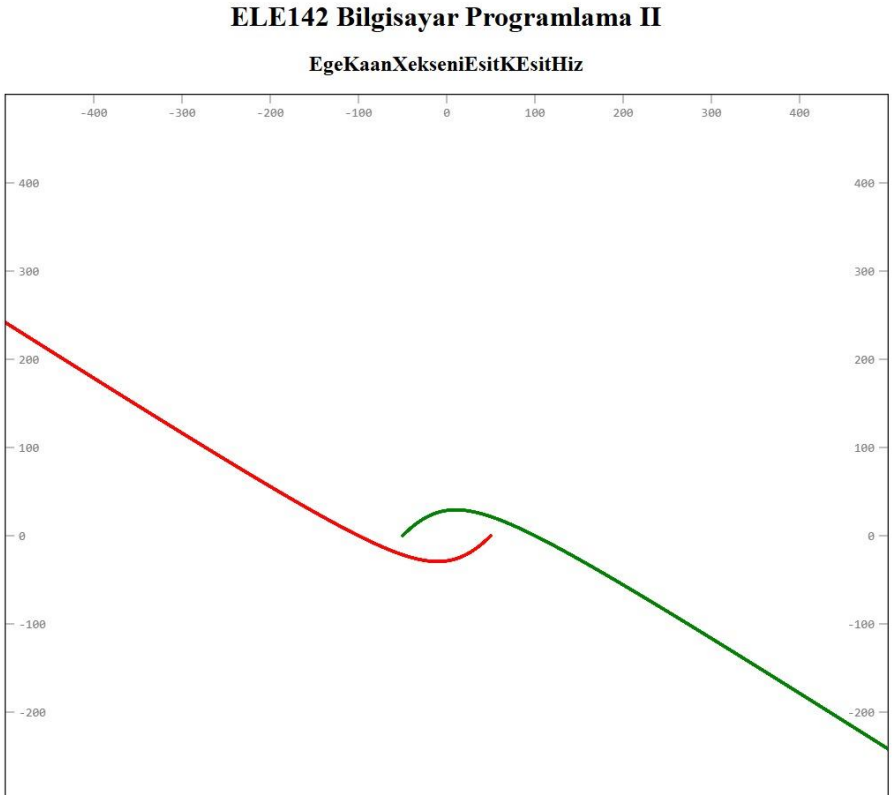


Resim 4.1.5 : Roket Sınıfı Kullanılarak Cisime Benzetimin Grafiksel Çıktısı

Resim 4.1.5'te nesneler roket sınıfı üzerinden çağırılmıştır. Ancak püskürtme hızı ve kütlesi değerleri olarak 0 değeri tercih edilmiştir. Bunun sonucunda bir roket ve bir cisim arasında herhangi bir fark kalmamıştır. Bu durumda roketler bir cisim gibi davranmış ve Resim 4.1.2'deki çıktının aynısı elde edilmiştir. Bu test mühendisliği durumunda roket simülasyon çıktılarının doğru olduğunu gösterdiğinden projenin önemli bulgularından biridir.

```
evren.insertCisim(150, vector2d(-50, 0), vector2d(1, 1));
evren.insertCisim(150,  vector2d(50, 0), vector2d(-1, -1));
```

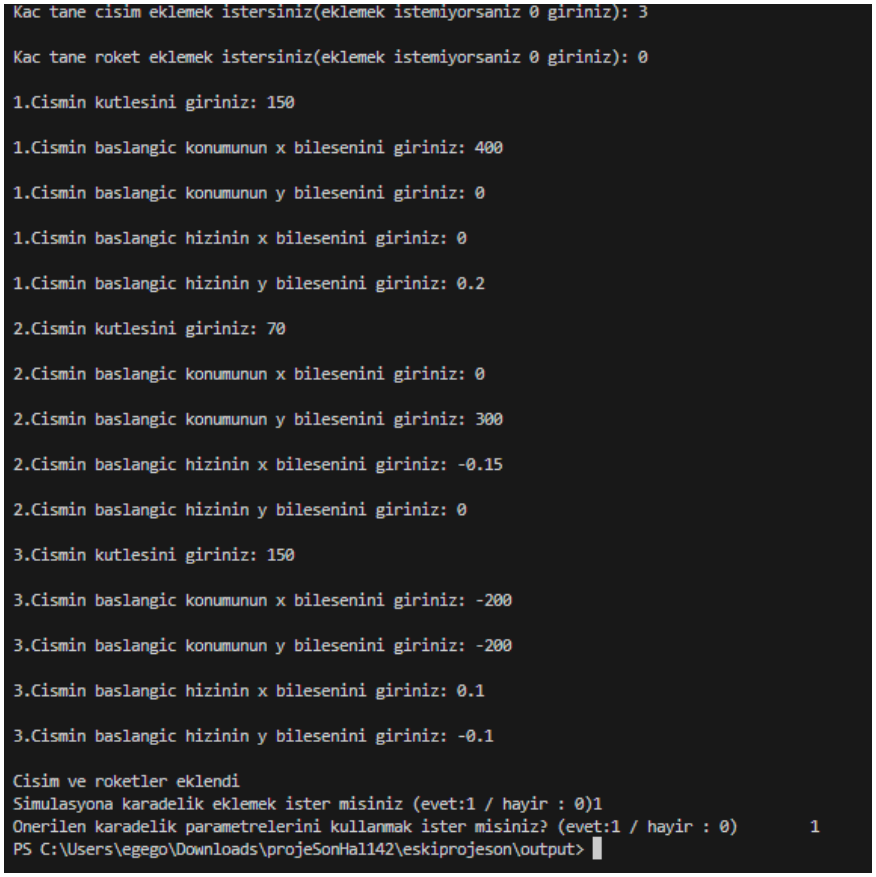
Resim 4.1.6: Eşit Kütle ve Hız Sahip Cisimlerin X Ekseni Üzerinde Oluşturumu



Resim 4.1.7: X Ekseninde Oluşturulmuş Eşit Kütle ve Hıza Sahip İki Cismin Çıktısı

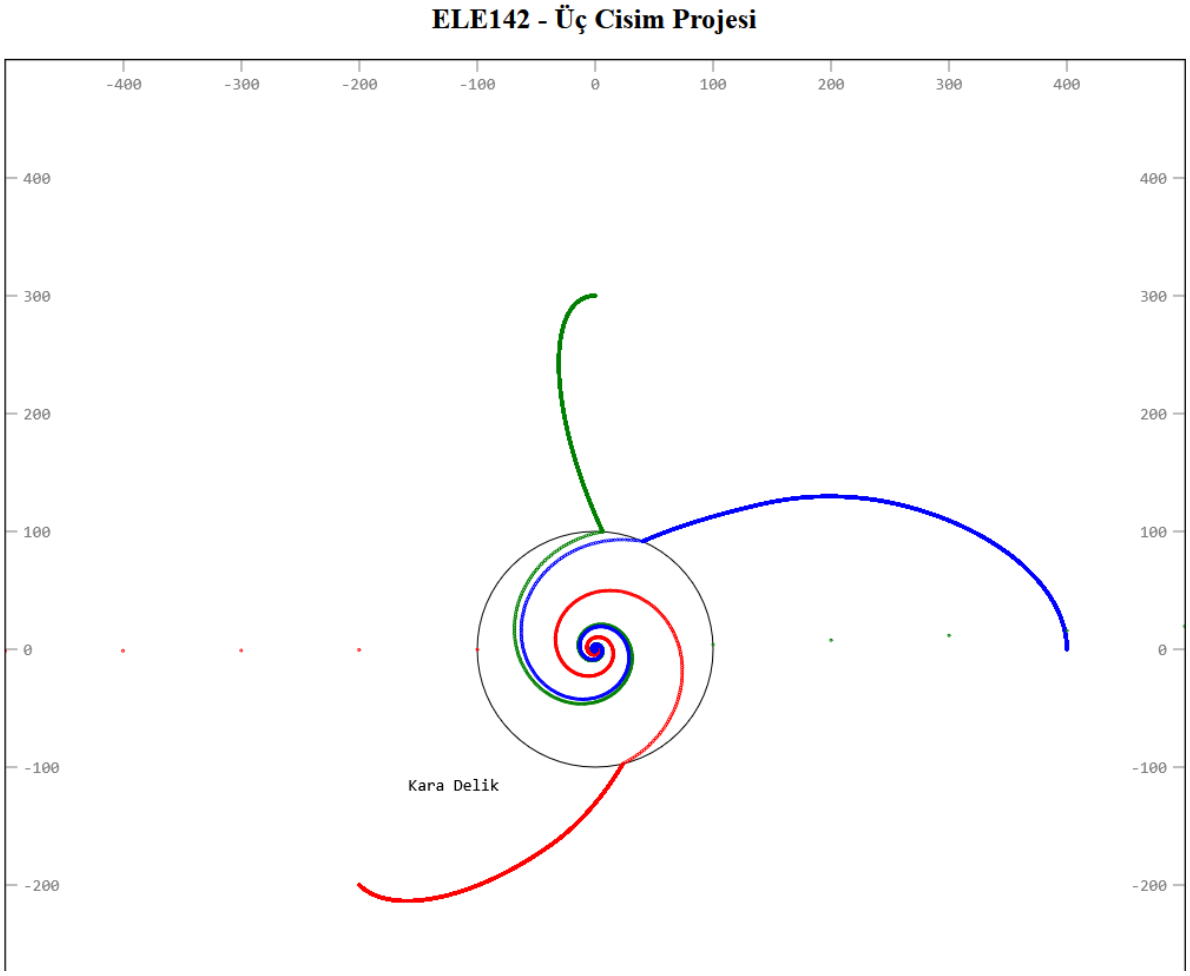
Resim 4.1.7'de, x ekseni üzerinde simetrik konumlarda büyüklüğü eşit fakat yönleri zıt olan iki cisim başlatılmıştır. Momentum, kapalı sistemde dış bir kuvvetin etkisi yoksa korunur. Burada, sistemin momentumu her iki cismin momentumlarının toplamına eşittir. Başlangıçta kütlelerinin aynı olması ve hızlarının zıt yönlü olması nedeniyle başlangıçta sistemin toplam momentumu 0'dır. İki cisim başlatıldığında, cisimlerin birbirine doğru hareket etmeleri beklenir. Bu resmi incelediğimizde, iki cismin birbirine doğru hareket ettiklerini, sonrasında zıt yönlerde momentum korunacak şekilde hareketlerine devam ettikleri yorumu yapılabilir. Bu da, sistemin toplam momentumunun başlangıçtaki sıfır değerini koruduğunu ve momentumu koruduğunu gösterir.

Enerji, kapalı bir sistemde, korunur. Bu durumda, kinetik enerji ve potansiyel enerji arasında bir dönüşüm yaşanacaktır. Cisimler birbirlerine yaklaştıkça potansiyel enerji artacak ve kinetik enerji azalacaktır, ancak toplam enerji sabit kalmalıdır. Resim 4.1.7'ye bakıldığında, eşit kütleli cisimlerin, eşit büyüklükteki hızlarla hareketlerine devam ettikleri görülmektedir. Bu da kapalı bir sistemde enerjinin korunduğunun göstergesidir. Yani, cisimlerin enerjileri zaman içinde dönüşüm yapsa da toplam enerji değişmeden korunmaktadır. Bu gözlemler, sistemin hem momentumunun hem de enerjisinin korunduğunu doğrular niteliktedir.



Resim 4.1.8 : Ek Analiz Olarak Oluşturulan Kara Delik Özelliğinin Önerilen Girdileri

Resim 4.1.8'de proje evrağında örnek olarak verilen 3 farklı cisimin kütle, başlangıç konumu ve hız değerleri girilmiştir. Ardından projeye ek analiz olarak eklenen kara delik özelliği aktif edilmiş ve kara delik özelliğinin varsayılan değerleri tercih edilmiştir. Tercih edilmesi durumunda kara delik özelliğinin başlangıç konumu ve yarıçapı değerleri kullanıcı tarafından seçilebilmektedir.



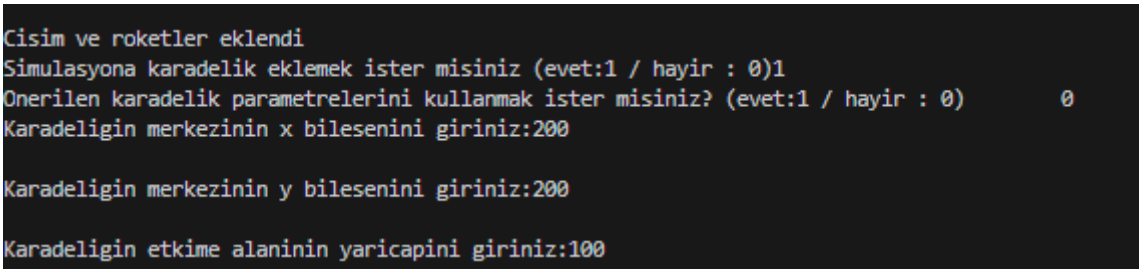
Resim 4.1.9: Proje Evrağındaki Başlangıç Değerleri ile Oluşturulmuş 3 Cismin Kara Delik Simülasyonu

Kara delikler, uzay-zamanın bir noktada son derece yoğunlaşarak, çekim gücünün öyle bir seviyeye geldiği gök cisimleridir ki, bu bölgeden ışık dahi kaçamaz. Kara delikler, Einstein'ın genel görelilik teorisiyle açıklanan, zaman ve mekânın bükülmesiyle oluşan ve kütlesi yoğunlaşmış olan bölgelerdir. Bir kara delik, çevresindeki her şeyi, ışık dâhil, güçlü çekim gücüyle içine çeker. Bir cisim kara deliğin olay ufkuna (çekim alanının geri dönüşsüz sınırı) yaklaşırken, kara delikten kaçabilmesi imkansız hale gelir. İçerisine düşen maddeler, zamanla birlikte spiraller çizerek kara deliğe doğru hızla ilerler, bu süreç evrendeki en ilginç ve gizemli olaylardan biridir.

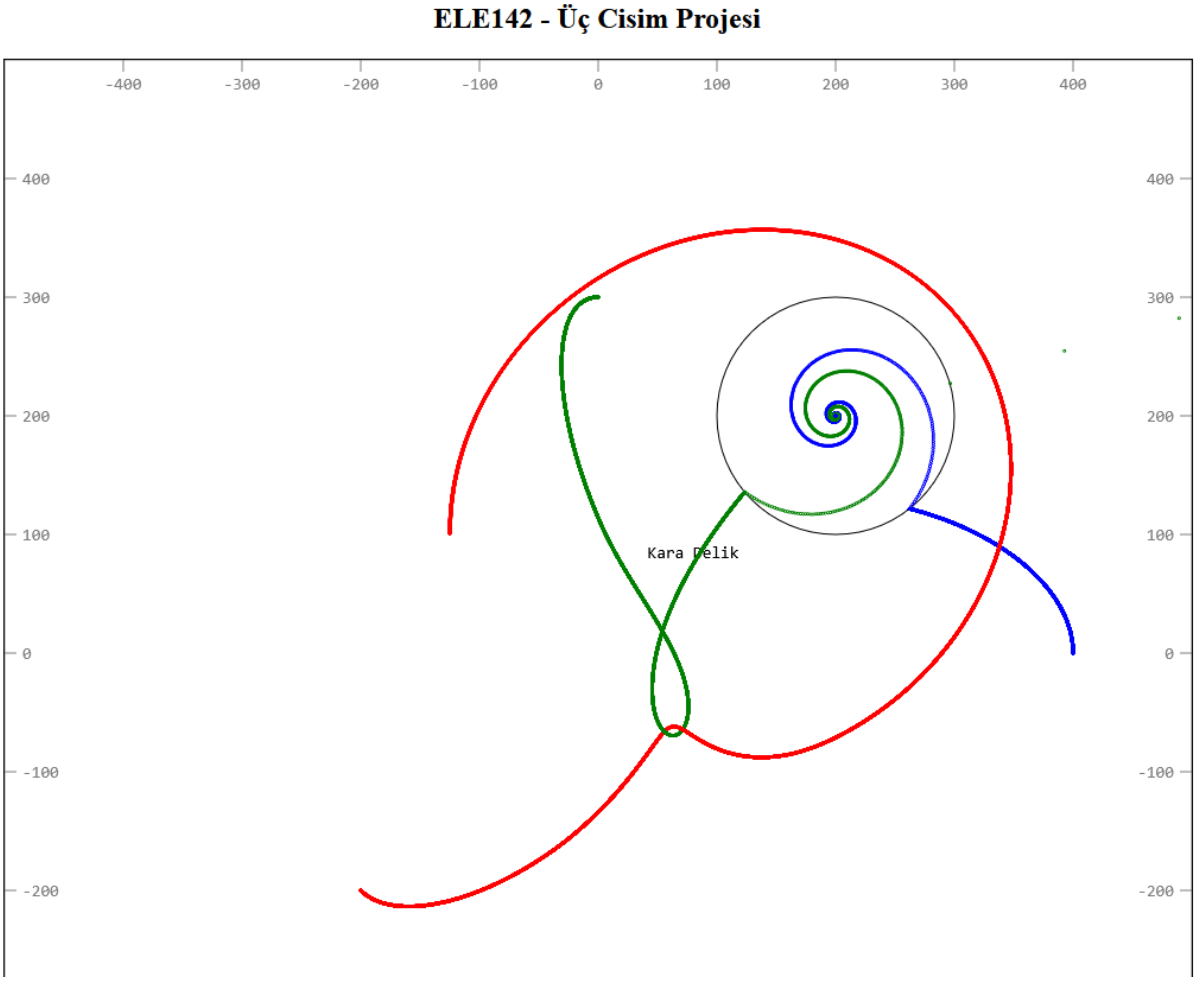
Projede kara delik özelliği, bu kuvvetli çekim alanının etkisini modelleyerek, içine giren cisimlerin hareketlerini simüle etmeyi amaçlamaktadır. Cisimler, başlangıçta karadelik

etrafında serbestçe hareket ederken, zamanla kara deliğin çekim gücüne kapılarak yörüngelerinin şekli değişir. Bu cisimler, kara deliğe doğru spiral bir yol izlerler. Logaritmik spiral hareketi, kara deliğin merkezine doğru daha yakın bir yörünge izleyen her cisim için geçerlidir. Yani, bir cisim kara deliğe yaklaştıkça, uzaklık ve hız arasında belirli bir ilişki kurularak, ivmesi değişir ve cismin yörüngesi sürekli olarak daralır. Zamanla bu hareketin ivmesi artar ve cisim, kara deliğin içine doğru daha hızlı bir şekilde hareket etmeye başlar.

Bu süreçte, cisimler kara delik etrafında sürekli bir dönüş yaparak, spiral bir yörünge izlerler. Bu hareket, uzayda ve zamanın bükülmesinde meydana gelen değişikliklerin somut bir örneğidir. Resim 4.1.9'daki simülasyon, bu fiziksel olayları daha iyi anlamamıza ve kara deliklerin çevresindeki dinamikleri gözlemlememize olanak tanır.

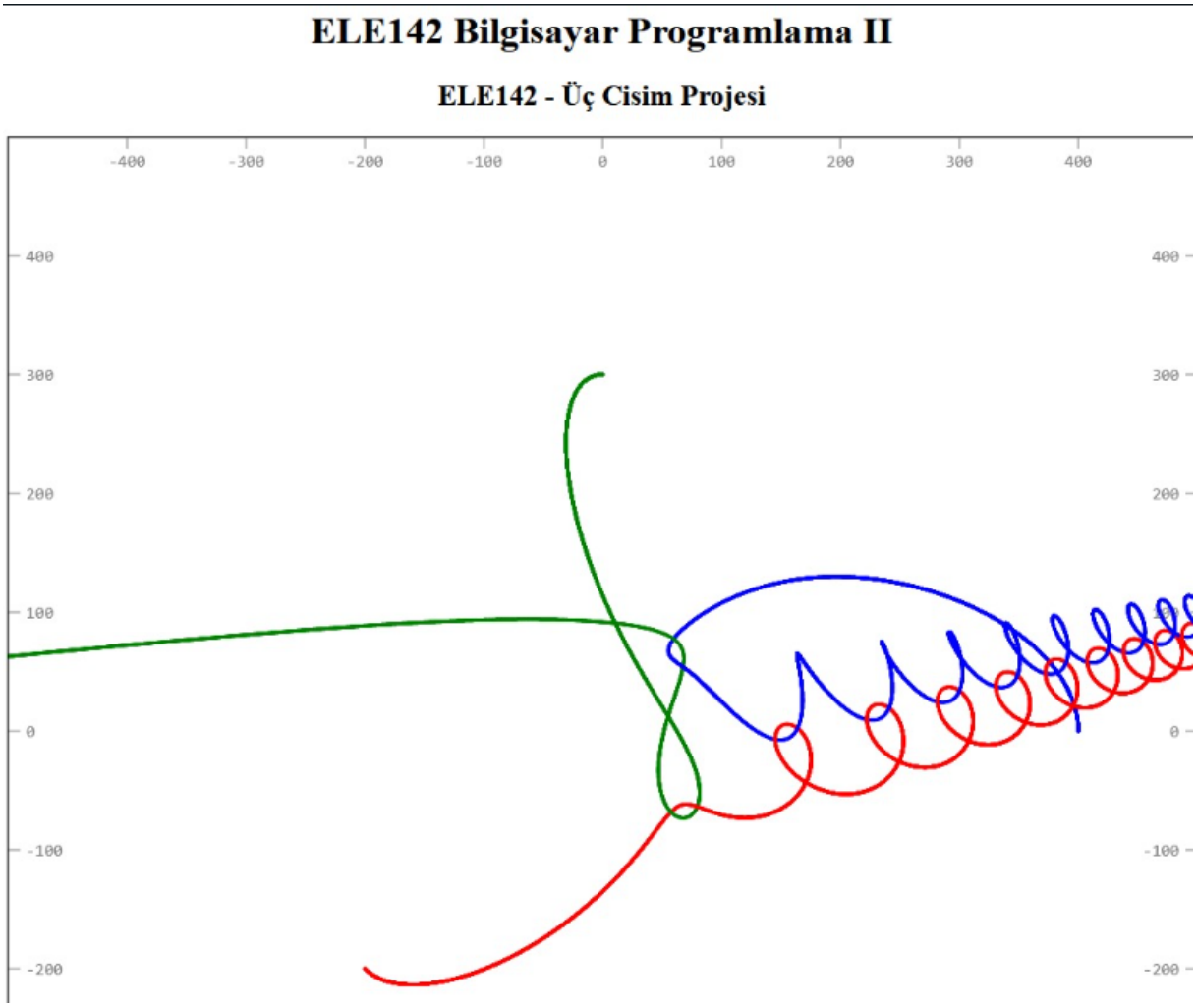


Resim 4.1.10: Ek Analiz Olarak Oluşturulan Kara Delik Simülasyonunun Değerlerinin Kullanıcıdan Alınması



Resim 4.1.11: Kullanıcı Girdileri ile Oluşturulmuş Kara Deliğin Simülasyonu

Resim 4.1.10'dan da görülebileceği üzere karadelik istenen konumlarda ve istenen yarıçap değerlerine oluşturulabilmektedir. Ancak simülasyonun daha net ve anlaşılır şekilde gözlemlenmesi için kullanıcıya varsayılan değerler sunulmaktadır. Kullanıcının varsayılan değerleri tercih etmemesi sonuç terminalde kullanıcıya konum ve yarıçap değerleri sorulmaktadır. Resim 4.1.11'de varsayılan değerlerin aksine terminal ekranından veri alınarak başka bir konumda bir kara delik oluşturulmuştur. Kara deliğin simülasyonu sonucu aynı bir önceki simülasyonda olduğu gibi kara deliğin içine giren cisimlerin spiral bir hareket yaparak merkeze yakınsadığı gözlemlenmiştir.

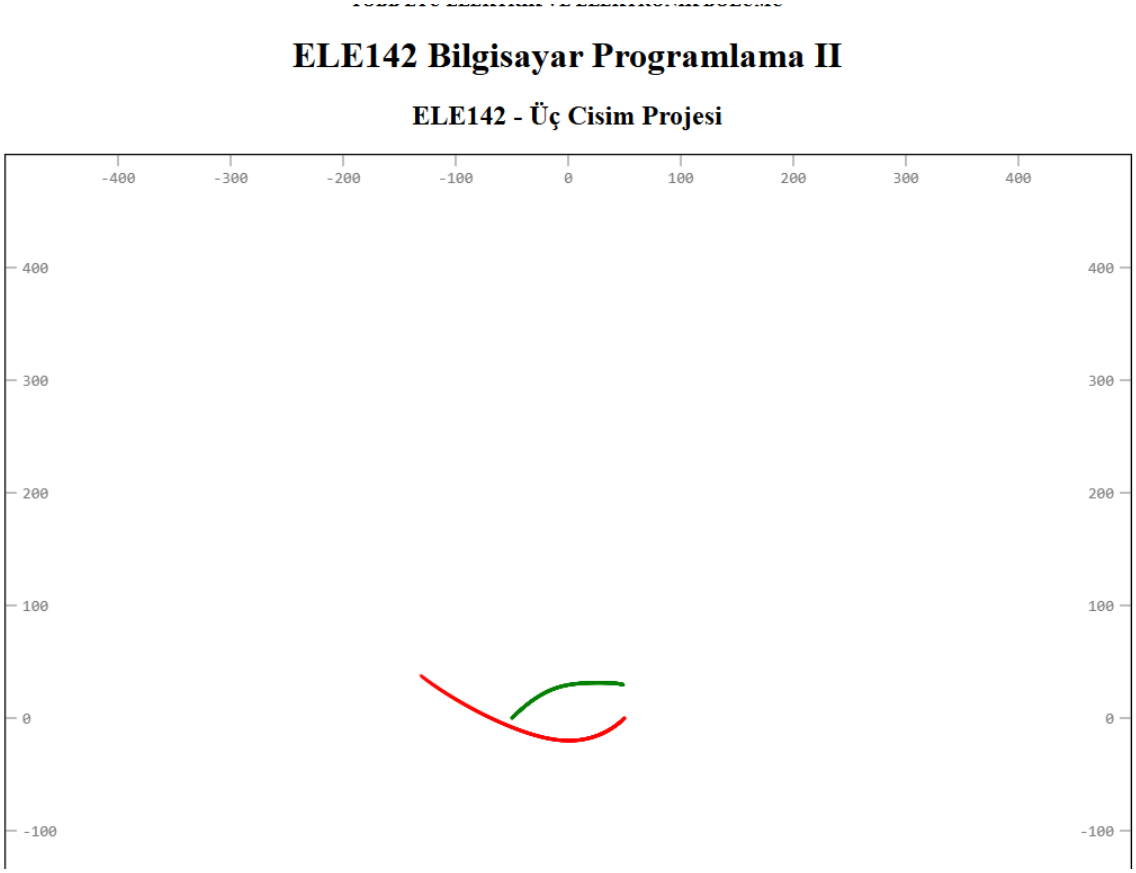


Resim 4.1.12: “dt” Parametresinin Artırılmasıyla Elde Edilen Üç Cisimli Simülasyon Çıktısı

Projede “dt” (delta t) değerinin arttırılması (0.1 idi 0.5 değerine yükseltildi), Resim 4.1.12’den görülebileceği üzere simülasyonun doğruluğunu ve dinamiklerini doğrudan etkileyen bir faktördür. Delta t, zaman adımını temsil eder ve simülasyonun her adımında sistemin durumunu güncellemek için kullanılır. Delta t değeri arttıkça, her adımda yapılan değişikliklerin büyüklüğü de artar. Bu durum, özellikle hızlı değişen sistemlerde, simülasyonun sonucunun daha hassas ve doğru olmasını engelleyebilir. “dt” değerinin arttırılmasıyla elde edilen çıktı Resim 4.1.2’deki çıktıyla kıyaslandığında farklılıklar gözlemlenmektedir. Çünkü zaman adımının büyümesi, fiziksel hareketlerin daha kaba bir şekilde hesaplanmasına yol açar, bu da özellikle hareketin doğruluğunu ve sırasını etkileyebilir.

Bunun sonucunda, çıktılar daha belirgin farklılıklar gösterebilir. Küçük “dt” değerleri, simülasyonun daha hassas olmasını sağlarken, büyük “dt” değerleri, hesaplama süresini kısaltabilir ancak doğruluk kayıplarına neden olabilir. Bu, özellikle hız, ivme ve konum gibi parametrelerin zamanla

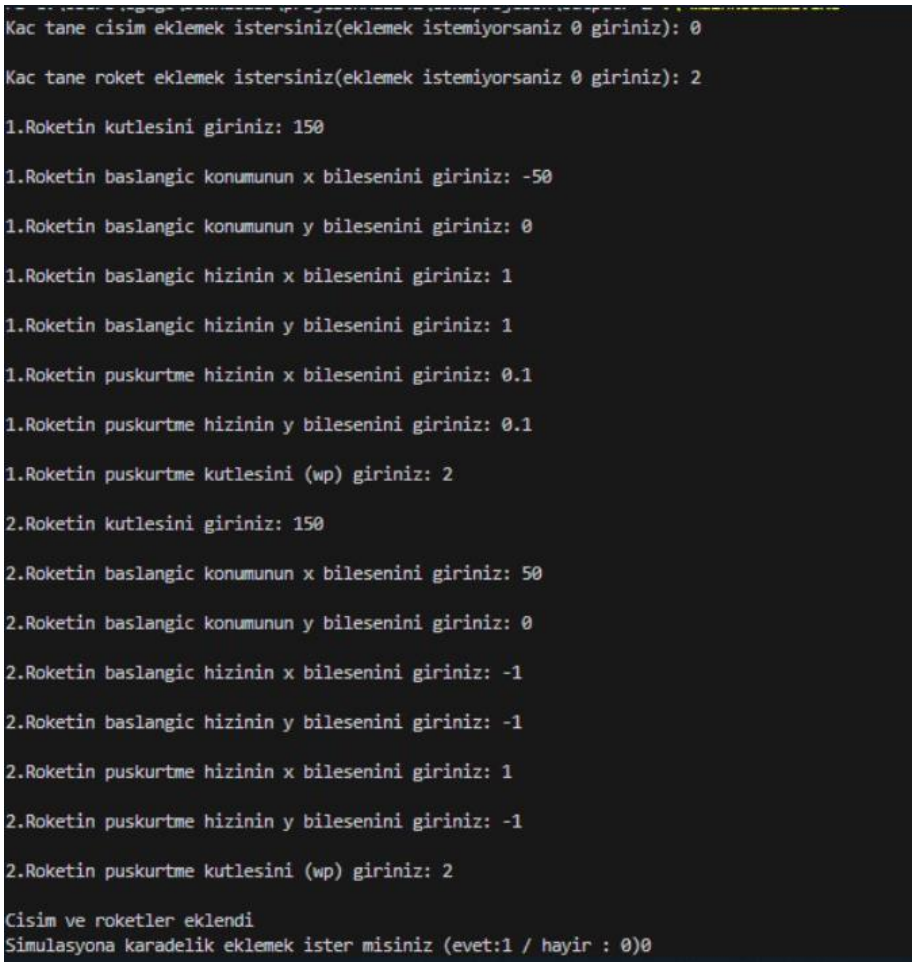
nasıl değiştiğini izleyen hesaplamalarda daha belirgin hale gelir. Örneğin, roketin veya cismin hareketi, zaman adımları arttıkça daha az düzgün ve daha tahmin edilemez hale gelebilir. Bu nedenle, “dt” değeri, doğru sonuçlar almak için dikkatlice seçilmeli ve simülasyonun doğruluğu ile hesaplama verimliliği arasında bir denge kurulmalıdır. Bu dengeyi sağlamak, projede elde edilen sonuçların fiziksel doğruluğu açısından kritik öneme sahiptir.



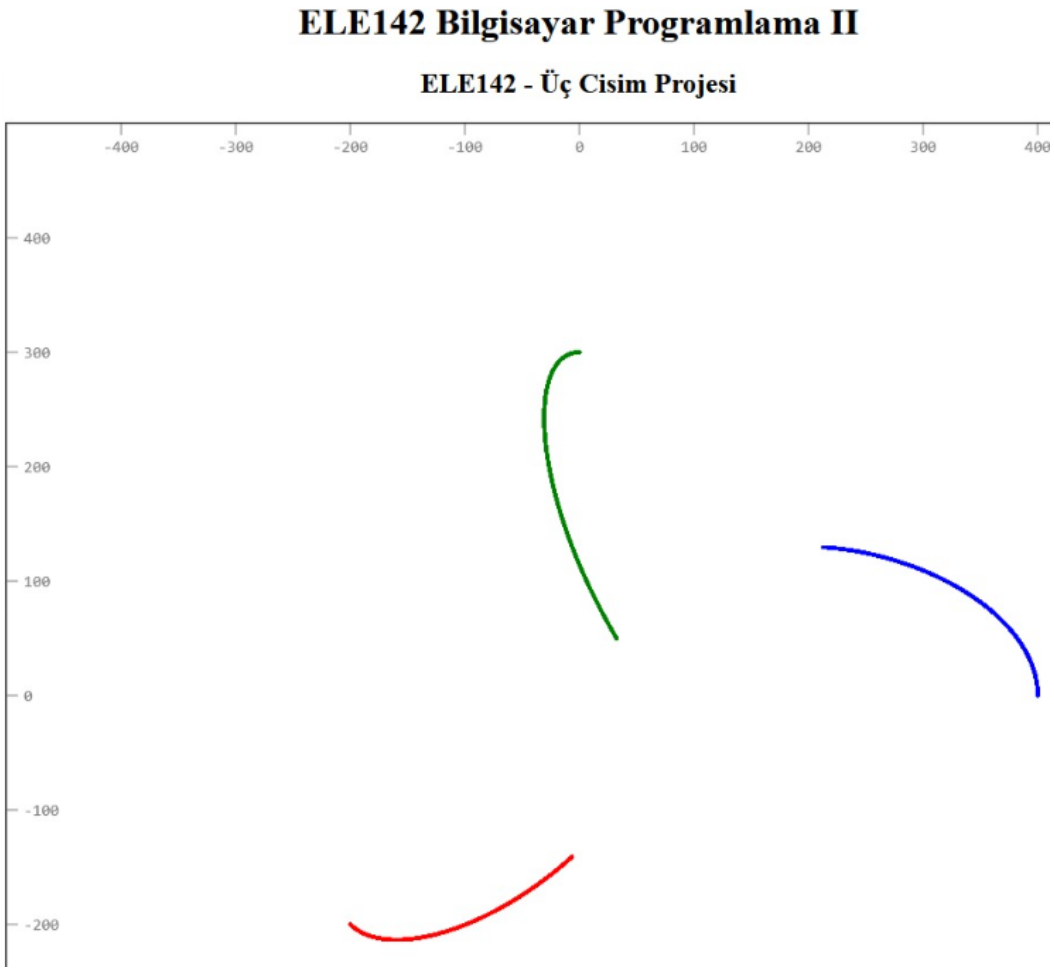
Resim 4.1.13: Resim 4.1.7’de Yapılan X-Eksen Analizinin Roketlere Genelleştirilmesi

Resim 4.1.13’de verilen grafikte, roket sınıfı kullanılarak yapılan simülasyon sonucunda, roketin hareketi gösterilmektedir. Kırmızı renkli roket, diğer cisme kıyasla daha düşük bir itme kuvvetine sahip olup, dolayısıyla yakıt tüketimi daha azdır. Bu nedenle, roketin kütlesi zamanla 0’a yaklaşırken, diğer roketle kıyaslandığında daha uzun bir süre boyunca hareket etmektedir. Kırmızı roketin kütlesi yavaş bir şekilde azalırken, yeşil cisim daha hızlı bir şekilde hareket eder çünkü kütlesinin azalması daha hızlı gerçekleşmektedir. Bu fark, kırmızı roketin yakıt tüketiminin daha düşük olmasından kaynaklanmaktadır.

Sonuç olarak, roketin kütlesi sıfırlanana kadar geçen süre, yakıt tüketim hızı ve itme kuvveti ile doğrudan ilişkilidir. Kırmızı roketin daha düşük bir yakıt tüketimine sahip olması, onun daha uzun süre hareket etmesine olanak tanır. Buna karşın, yeşil cisim daha hızlı bir şekilde kütlesini kaybeder ve bu da hareketinin erken sona ermesine yol açar. Bu farklar, projenin simülasyon kısmında, roketin fiziksel özelliklerine dayalı olarak ortaya çıkan farklı dinamikleri göstermektedir.



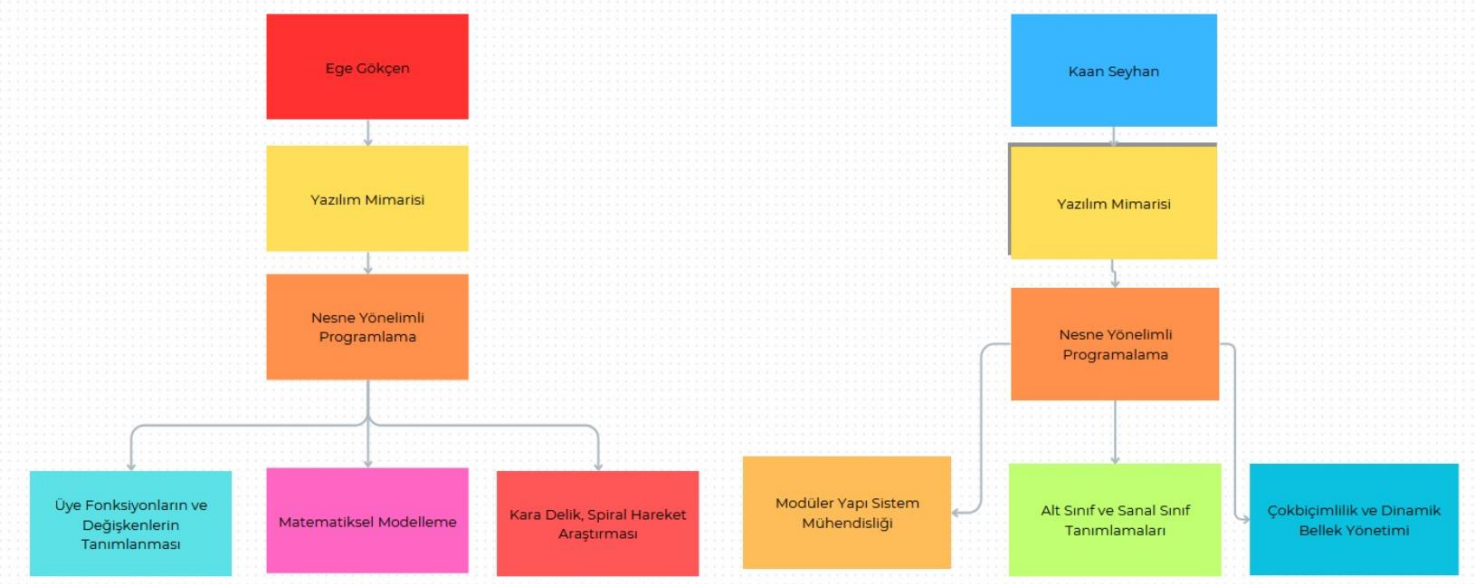
Resim 4.1.14 Resim 4.1.13'te Yapılan Simülasyonun Oluşturumu



Resim 4.1.15: Toplam Örnek Sayısının Azaltıldığı Üç Cisim Çıktısı

Resim 4.1.15'te bulunan çıktı için verilen girdiler, Resim 4.1.1 ile aynıdır ancak çıktı Resim 4.1.2 ile farklıdır ve simülasyon tamamlanamamıştır. Bunun sebebi, Resim 4.1.2'nin 60000 örnek için çizdirilirken Resim 4.1.15'in 20000 örnek için çizdirilmesidir. Analizde iki girdi için de "dt" parametresi sabit tutulmuştur ve bir kontrollü deney gerçekleştirilmiştir. Kontrollü deney sonrası gözlemlenen bu durum, "dt" ile tanımlanan birim zaman adımının değişimiyle belirli bir hassasiyete göre tolere edilebilir. Ancak "dt" nin belirli bir eşiğin üstüne çıkması gözlemlenmesi istenen noktaların gereğinden fazla anlık değişime uğramasına sebep olur bunun yanında örnek sayısının artırılması kodun çalışma süresi ve performansını zayıflatır. Bu noktada kullanıcı bir mühendislik ödünleşmesiyle karşı karşıya kalır. Proje dosyası olarak yüklenen kod, göz kararı ile optimize ödün değerlerine yakın olduğu tespit edilmiştir.

5. Görev Dağılımı



Resim 5.1: Projede Kişi Bazında Düşen Görev Dağılımı

Proje içerisinde görev dağılımı, ilgi alanlarına göre yapılmıştır. Üstteki görselde de görüldüğü gibi, proje iki ana bölüme ayrılmaktadır: Ege Gökçen ve Kaan Seyhan her biri belirli görevleri üstlenmiştir. Ege Gökçen, yazılım mimarisinin oluşturulmasında ve nesne yönelimli programlama tekniklerinin uygulanmasında liderlik etmiştir. Özellikle üye fonksiyonları ve değişkenlerin tanımlanması ile ilgili olarak, simülasyona dahil edilen tüm nesnelerin ve fonksiyonların doğru bir şekilde yapılandırılmasını sağlamıştır. Ayrıca, matematiksel modelleme kısmında da önemli bir rol oynamış ve simülasyonun fiziksel doğruluğunu sağlamak adına gerekli hesaplamaları ve formülleri uygulamıştır. Kara delik ve spiral hareket araştırması da Ege Gökçen'in sorumluluğunda olmuş, simülasyona bu konseptin doğru bir şekilde entegre edilmesi sağlanmıştır. Bu araştırma, özellikle kara deliklerin etki alanında gerçekleşen hareketlerin doğru bir şekilde simüle edilmesi için büyük önem taşımaktadır.

Kaan Seyhan ise yazılım mimarisinin ve nesne yönelimli programlama çerçevesinin diğer önemli parçalarını geliştirmiştir. Alt sınıf ve sanal sınıf tanımlamaları Kaan Seyhan'ın

sorumluluğunda yapılmış, böylece her nesne ve fonksiyonun doğru şekilde organize edilmesi sağlanmıştır. Ayrıca, çokbiçimlilik (polimorfizm) ve dinamik bellek yönetimi konularında Kaan Seyhan, önemli katkılarda bulunmuştur. Bu sayede simülasyonun verimli bir şekilde çalışabilmesi ve bellek yönetiminin doğru yapılabilmesi sağlanmıştır. Projeye genel olarak bakıldığında, her iki grup üyesi de birbirlerini tamamlayıcı bir şekilde çalışmış ve projenin her aşamasında eşit sorumluluklar almışlardır. Bu şekilde yapılan iş bölümü, projede verimliliği artırmış ve projeye katılan herkesin kendi güçlü yönlerini sergileyebilmesini sağlamıştır.

6. Bulgu Yorumları ve Sonuçlar

Bu proje sonucunda karmaşık fiziksel dinamikleri simüle eden bir sistem tasarlanmıştır ve. Üç cisim problemi, nesne yönelimli programlama prensiplerine dayalı olarak, C++ kullanılarak modellenmiştir. Proje, nesne yönelimli programlama (OOP) ilkelerini başarılı bir şekilde uygulayarak, simülasyonun her aşamasını modüler ve esnek bir yapıda inşa etmeyi başarmıştır. Bu bağlamda, C++ dilinin sunduğu avantajlar, özellikle virtual fonksiyonlar, kalıtım ve operatör aşırı yükleme (overloading) gibi özellikler, projenin temel yapı taşlarını oluşturmuş ve her bir fonksiyonel bileşenin bağımsız olarak işlev görmesini sağlamıştır.

Öncelikle, virtual fonksiyonlar ile ilgili olarak, projenin tüm sınıflarında fonksiyonların yeniden tanımlanabilir olmasını sağlayan bir yapı kullanılmıştır. Bu sayede, purevirtual sınıfı üzerinden türetilen alt sınıflar, kendilerine özgü işlevsellikleri sağlamak amacıyla fonksiyonları özelleştirebilmiştir. Bu sayede, tüm sistemdeki nesneler birbirinden bağımsız şekilde özelleştirilmiş ve belirli bir işlevsellik kazandırılmıştır. Bu yaklaşım, kodun esnekliğini artırmış ve projenin ölçeklenebilirliğine katkı sağlamıştır. Örneğin, cisim sınıfı ve ondan türetilen diğer sınıflar, her bir fiziksel nesnenin hareketini ve etkileşimini doğru bir şekilde simüle edebilmek için özelleştirilmiş fonksiyonlarla donatılmıştır. virtual anahtar kelimesiyle deklare edilmiş fonksiyonlar, bu fonksiyonların türetilen sınıflarda geçersiz kılınabileceğini belirtirken, yazılımda polimorfizmi uygulamıştır.

Kalıtım (inheritance) ise sınıflar arasında güçlü bir bağ kurmuş ve kodun yeniden kullanılabilirliğini artırmıştır. purevirtual sınıfından türetilen cisim sınıfı, temel özelliklerin ve fonksiyonların alt sınıflarda kullanılmasına olanak sağlamıştır. Cisim sınıfı, genel fiziksel nesnelerin özelliklerini taşırken, bu sınıftan türetilen roket gibi özel sınıflar, kendi özgün işlevlerini ekleyebilmiştir. Bu kalıtım yapısı, kodun daha verimli ve sürdürülebilir olmasına katkı sağlamıştır, çünkü her bir alt sınıf yalnızca gerekli özellikleri ve işlevsellikleri eklemiştir.

Operatör aşırı yükleme, matematiksel işlemleri daha basit hale getirerek (bazı durumlarda bir zorunluluk) C++ dilinde vektörel hesaplamaları kolaylaştırmıştır. "vector2d" sınıfında uygulanan aşırı yükleme sayesinde, toplama, çıkarma, çarpma ve bölme işlemleri doğrudan operatörler aracılığıyla gerçekleştirilebilmiş, böylece kod daha anlaşılır ve işlevsel hale gelmiştir. Bu özellik, simülasyondaki nesnelerin hareketlerini ifade etmeyi kolaylaştırarak işlem süreçlerini optimize etmiştir.

Projede ayrıca bellek yönetimi ve dinamik bellek kullanımı üzerinde de durulmuştur. C++ dilinin sunduğu new ve delete komutları ile bellek dinamik olarak yönetilmiş ve simülasyonun her adımında kullanılan veriler doğru bir şekilde işlenmiştir. Bellek sızıntılarının önlenmesi ve verilerin etkin bir şekilde saklanması için özel önlemler alınmış ve sistemin verimliliği artırılmıştır. Önemli bir gerçek olarak sınıfların üye değişkenlerinin bir gösterici yardımıyla heap bellekte tutulması için gösterici oluşturulmasından bahsedilebilir.

Son olarak, grafiksel çıktılar ve veri görselleştirme projede önemli bir yer tutmuştur. Her bir nesnenin hareketi, canvas sınıfı yardımıyla görselleştirilmiş ve simülasyonun sonuçları, görsel bir şekilde kullanıcıya sunulmuştur. Bu görselleştirme, kullanıcıların simülasyonun her aşamasını daha rahat bir şekilde takip etmelerine olanak tanımıştır.

Ek Analiz: Kara Delik ve Spiral Hareket

Projeye eklenen kara delik özelliği, simülasyona derinlik katmıştır ve proje isterlerinden biri olan ek özellik koşulunu gidermiştir. Kara delikler, uzay-zamanın bükülmesiyle oluşan ve çekim gücü güçlü olan nesnelerdir ki hiçbir cisim kara delikten kaçamaz. Bu özellik, kara delik etrafındaki cisimlerin hareketlerini simüle ederken dikkate alınmıştır. Bu projede, kara deliğin etki alanına giren cisimlerin logaritmik spiral hareketle kara delik merkezine doğru yaklaşması simüle edilmiştir. Bu hareket, cisimlerin hızlanarak ve daha dar yörüngelerde dönerek kara deliğe doğru ilerlemelerini göstermektedir.

Cisimlerin kara deliğe doğru yaklaşırken izlediği spiral hareket, kara deliğin çekim gücünün arttığı bölgelerde gerçekleşir. Başlangıçta cisimler, kara delikten uzakta yer alan sabit bir yörüngede hareket ederken, zamanla kara deliğin çekim alanına girerler ve hızları artar. Bu süreçte, radyal güncellemeler ve açısal hareketler ile cisimlerin yörüngeleri daralır. Logaritmik spiral formülü, bu hareketi matematiksel olarak tanımlar. Kara deliğin çekim gücü, cisimlerin hareketini dönüştürür ve onları daha hızlı bir şekilde kara deliğe çeker. Bu simülasyon, uzay-zamanın bükülmesi ve kara deliklerin çevresindeki fiziksel dinamiklerin gözlemlenmesine olanak tanır.

7. Kaynakça

https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples *[1]*

https://www.mdpi.com/2227-7390/11/17/3744 [2*]*

B. Özcan - Spagettileştiren Kara Delikler [3]

O. Arık, Grafik Çizim Kütüphanesi, TOBB Ekonomi ve Teknoloji Üniversitesi, Elektrik ve Elektronik Mühendisliği Bölümü, 2025. [4]