

Görüntü işleme ile 3D Runner Oyun

Onur DOĞAN – Hasan ARNAVUTOĞLU – Ceren DEMİREZEN

Proje -1 / BİTİRME TEZİ

**Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Doç. Dr. Ahmet SAYGILI**

2023

T.C. TEKİRDAĞ NAMIK KEMAL ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ



PROJE-1 / BİTİRME TEZİ

Görüntü işleme ile 3D Runner Oyun

Onur DOĞAN – Ceren DEMİREZEN - Hasan ARNAVUTOĞLU
1190606901 1190606013 1200606049

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMAN: Doç. Dr. Ahmet SAYGILI

TEKİRDAĞ-2023

Her hakkı saklıdır.

İÇİNDEKİLER

A.GİRİŞ.....	8
B.Kullanılan Teknolojiler ve Versiyon.....	8
C.Proje Yol Haritası.....	15
D.Game Design ve Level Design	17
I.BÖLÜM	18
1.Karakter ve Karakterin Hareketi	19
1.1 Karakterin Eklenmesi	19
1.2 Karakterin Hareketi	22
1.3 Karakterin Hareketi ve Koşma Animasyonu	24
1.4 Karakterin Hızının Arttırılması	25
1.5 Karakterin Hareket Edeceği Zemin Sınırlarının Belirlenmesi	26
1.6 Karakterin Zıplaması	28
2.Kamera Nesnesi ve Kamera Takibi.....	31
3.Game Design ve Asset Kullanımı.....	34
3.1 Zemin Prefab	35
3.2 Çevre Prefab	37
3.3 Section Prefab.....	38
3.4 Engel Prefab.....	39
3.5 Elmas Prefab	40
4.Oluşturulan Prefabların Otomatik Üretilmesi ve Yok Edilmesi.....	42
4.1 Sectionların Oluşturulması	42
4.2 Engellerin Oluşturulması	47

4.3 Elmasların Oluşturulması	50
4.3.1 Elmas Prefab Rotasyon Bilgisi	53
5.Engel ile Çarpışma ve Oyun Sonu Ekranı.....	54
5.1.Engellerle Etkileşim	54
5.2. Oyun Sonu ekranı	55
6.Butonlar ve Sahne Geçişleri.....	56
6.1.Sahne 2 : Ana Menü	57
7.Level01 UI Sistemi	58
8.PlayerPrefs ile Skor Tutmak.....	59
9.Animator ve Animasyon Bağlantıları	60
10.Can Sistemi	61
11.Karakter ve Objelerin Efekt Ayarları	64
11.1.Karakter Efekt Ayarları	64
11.2. Elmas Efekt Ayarları ve Elmasların Toplanması.....	66
12.Müzik ve Ses Efektleri.....	67
12.1.Level01 : Müzik ve Ses Efektleri.....	67
12.2. Ana Menü : Müzik ve Ses Efektleri	69
II.BÖLÜM.....	70
1.Python OpenCV Mediapipe – CvZone Hand Detector	70
1.1 LandmarkList	71
2.Unity : Socket Üzerinden UDP Protokolüyle Verinin Alınması	76
3.Unity : Karakterin El Takibiyle Yönlendirilmesi	78
Kaynakça	81

A- GİRİŞ

Bir oyun geliştirme fikriyle yola çıktığımızda farklı bir soluk getirerek sivrilebileceğimiz bir fikir ortaya koymadık istedik. Son yıllarda gittikçe trend haline gelen yapay zeka uygulamalarına da bu projede yer vermek istedik. Unity oyun motorunun bize sunduğu fırsatları değerlendirerek geliştirilecek 3D bir oyunun standart olan giriş/çıkış birimlerinin yanına ek olarak kaynaktan alınacak bir görüntüye tepki vermesiyle oynanılabilirliği çeşitlendirebilir miyiz? Sorusuna yanıt aramak bu projedeki temel motivasyonumuz oldu.

Projenin geliştirme süreci meşakkatli olduğu kadar daha önce elde edilmemiş tecrübelerinde grup üyeleri arasında paylaşılmasına sebep oldu.

Oyun geliştirme süreci başlı başına bir süreç ve bu sürece dahil olmadan çok yüzeysel bir izlenimimiz vardı. Lakin sürecin içerisinde bulunduğunuzda çok fazla detayın içerisinde hedefinize ulaşmanız zorlaşıyor. Buna ek olarak bir de yapay zekanın projeye dahil olması işimizi zorlaştırmasına karşın kazanımlarımızı doğru orantıda arttırdı. Bunun için projeye başlamadan önce her şeyin **planlı** yürütülmesi bizim ilk hedeflerimizden biriydi. Sistematiik bir düzende çalışmayı yürütemezsek başarılı bir sonuç elde edilemeyeceğini biliyorduk. Buna müteakiben grup üyelerinin de üzerinde çalışmak istediğimiz konu hakkında teorik bilginin ötesinde çok fazla bir deneyime sahip olamayışımız bizi düzenli çalışmaya ve **yazılım geliştirme süreçlerine** riayet ederek hareket etmenin bizim için bu süreci yönetmek adına en makul seçenek olduğuna karar verdik.

Bu rapor projenin genel bir özeti ve yazılım geliştirme sürecinin son adımı, halkası diyebileceğimiz elde edilen birikimin dökümantasyonunu yapmamıza olanak sağladı. Proje iki ana bölümden oluşuyor. Ana bölümlerin altında birçok detay konu başlığı göreceksiniz. **Oyun geliştirme süreci** ve **Görüntü işleme** bu projenin orta noktada buluşturulması gereken iki ana başlığı.

B- Kullanılan Teknolojiler ve Versiyon

Proje içerisinde deneme yanılma yoluyla birçok farklı teknoloji ve teknik denenmiştir. Nihai olarak başarı elde edilen teknolojileri ve kullanılan kaynaklardan kısaca bahsetmek gerekirse ;

İki ana başlıktan bahsetmiş olsak da tek bir ana çerçevemiz var aslında o da ana ürünümüz olan oyunumuz. Tabi ki büyük bütçeli bir oyun şirketi olmamız sebebiyle bağımsız bir geliştirici grup olarak **oyun motoru** olmazsa olmazlardan ve bunu temin edebilmek için piyasada bulunan popüler oyun motorlarından birini tercih etmemiz gerekiyordu.

Oyun Motoru



Unity ve Unreal Engine iki büyük oyun motoru arasında bir tercih yapmamız gerekiyordu. Birbirine yakın işlevlere sahip olmasına rağmen forumlarda veya geliştirici topluluklarında sıkça tartışma konusu olan iyi büyük oyun motoru arasından **Unity'i** tercih ettik.

Unity'i tercih etmemizde etkin rol oynayan en büyük sebep ise geliştiricilerin unity'i daha çok tercih etmeleri ve bağımsız oyun geliştiricilerinin daha çok bu tarafa yönelmesi oldu. Unity seçildikten sonra ikinci ana başlığımız olan Görüntü işlemeyi içerisine entegre edip edemeyeceğimizi araştırmamız gerekti.

Unity ve Unreal Engine gibi oyun motorları kendine ait paketlerin olduğu ve dışarıdan fazla müdahale yapılmasına pek izin vermeyen platformlar. Yapılacak müdahalelerinde pek sağlıklı olacağını söyleyemeyiz.

Oyunumuzda bulunan grafik , ses , efekt gibi bileşenleri Unity Asset Store üzerinden temin ederek ilerledik.

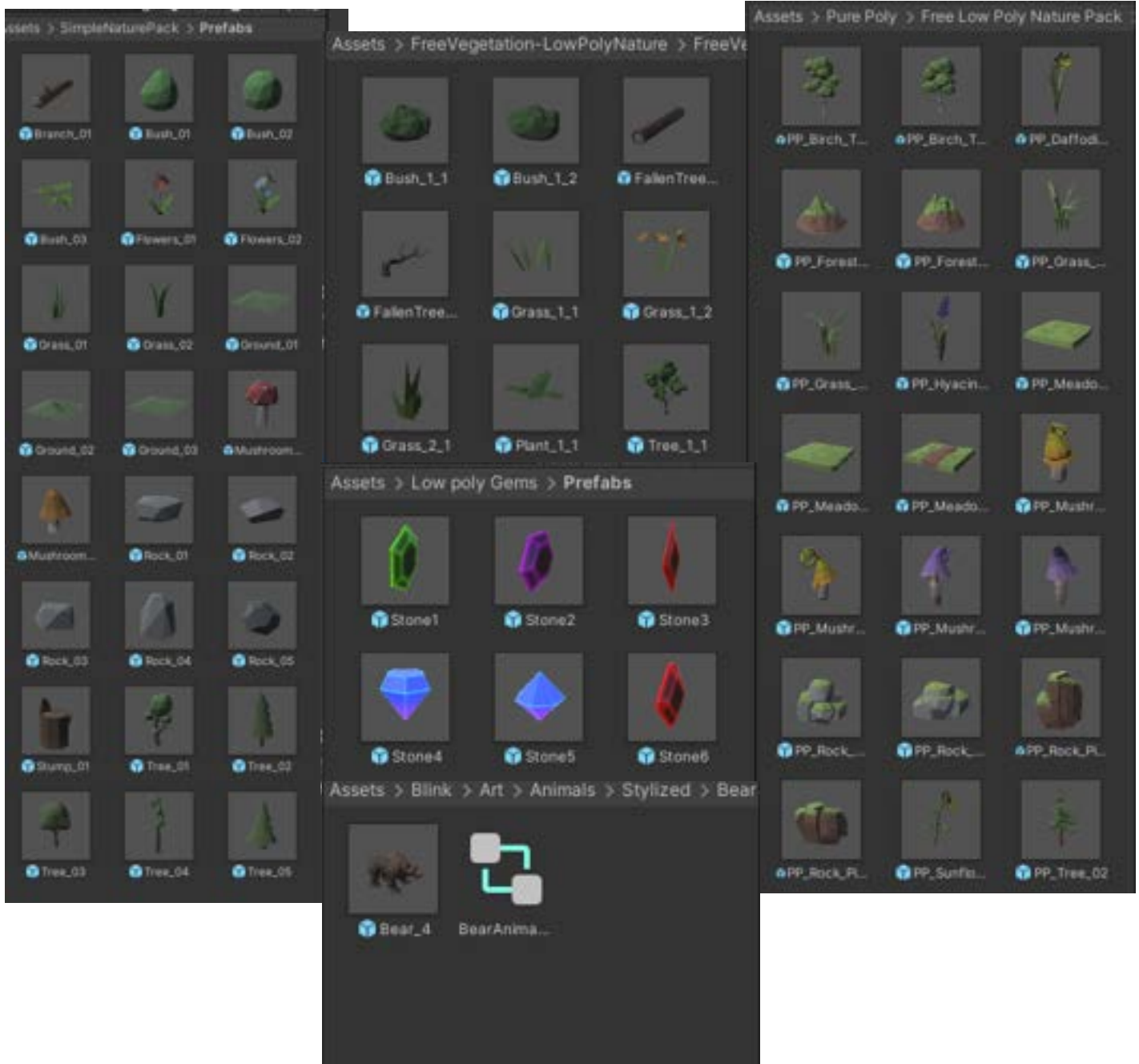
Unity versiyonlar arası çok fazla sıkıntılar çıkarabilen bir oyun motoru o sebeple grup üyelerinin üzerinde geliştirme yapacakları bu oyun motoru versiyonunu belirlemek ve sabit bir versiyon üzerinden ilerlemek çok önemliydi.

- **Unity Version : Unity 2022.3.11f1** tercih edilen Unity versiyonu.

Asset Store

Oyun dizayn edilirken birçok grafik unsuru bulunuyor ; kontrol ettiğiniz karakter , karakterin görüş açısında bulunan nesneler , etkileşime girebileceği nesneler gibi. Bu sebeple bunları temin etmek için Unity'nin sağlamış olduğu Asset Store üzerinden projeye dahil edilen ve kullanılmış olan paketler ;

- *Low Poly Simple Nature Pack*
- *Free Low Poly Nature Forest*
- *Low Poly Nature – Free Vegetation*
- *Low Poly Gems*
- *FREE Stylized Bear - RPG Forest Animal*



Müzik ve Ses Efektleri

Oyun atmosferini ve konseptini temsil edecek ve oyunla özdeşleşecek kaynaklardan biri de “*müzik ve ses efektleri*”. Unity Asset Store sadece grafik tabanlı kaynaklar değil , bunların yanında müzik ve ses efektleri için de paketler sunuyor. Aradığınızı bulamayabilirsiniz ama kapalı kutu olduğu için de oyun motoru tercihini yaptıktan sonra elimizde neler var , neyi kullanabiliriz gibi bir takım araştırmalar yapmamız gerekti.

- ***Free UI Click Sound Pack***

Ana menü ve oyun içerisinde bulunacak olan butonlar için ses efekti sunan bu pakette birçok farklı konsept için kategorilenmiş ses paketleri bulunuyor.

- ***Achievement SFX FREE***

Oyuncuya bir hedef olması için oyun içerisinde toplanabilir ödüllerin ve buna bağlı elde edilecek bir skor sisteminin olması gerekiyordu. Karakterin toplayacağı ödüller için kullanılan ses efekti paketi.

- ***Casual Jingles - 022420***

Karakterin oyun sonu özet ekranı için kısacası “game over” ekranı için özel olarak seçilmiş bir başka ses efekti paketi

- ***Free Music Tracks For Games***

Ana menü ve oyun içerisinde arka planda çalacak olan müzikler için seçilmiş olan paket. Oyun konseptiyle ortak noktada buluşabilecek arka plan müziği seçilmiştir.

Karakter ve Animasyon

Oyunun ana kahramanı tabi ki karakterimiz. Asset store üzerinde karakter modellemeleri bulunsa da bir dış kaynaktan karakter ve animasyonu temin ettik. Karakter seçimi de çevre ve müzik seçimleri kadar önemli hatta baş rol diyebiliriz. Adobe firmasının altında bulunan ***mixamo*** üzerinden karakter ve animasyonları da temin ettik.



Animasyonlar ;

- ***Standart Run***
- ***Stumble Backwards***
- ***Jumping***
- ***Happy Idle***

Ana animasyonumuz “Standart Run” olacak çünkü döngüye alarak sürekli işleteceğimiz bir animasyon olacak. Sürekli ileriye doğru hareket eden bir karakter nesnesinin haliyle koşma animasyonunda sürekli işletilmesi gerekiyor. Diğer animasyonlar ise bazı durumlarda aktif olacak animasyonlar ; Zıplama ya da Düşme animasyonu gibi.

Görüntü İşleme ve Python

Görüntü işleme konusu projemizin ikinci ana bölümlerinden biri. Bu sebeple oyun geliştirilmesiyle beraber ilerlemesi gereken ve aynı düzlemde sorunsuz bir şekilde buluşturmamız gerekiyor.

Geliştirme sürecinde birçok problemle karşılaştık. İlk aklımıza gelen çözümler sorunumuzu çözmedi. Unity'nin kendi Pluginlerini ve Asset Store'da bulunan OpenCV paketlerini kullanmaya çalıştığımızda versiyon olarak çok eski ve uzun süredir geliştirilme süreci kesilmiş paketlere denk geldik.

Güncel tutulan bir paket bulduk lakin bu da ücretli bir versiyondu. O sebeple başka bir çözüme yönelmemiz gerekti. Unity'nin dışına çıkarak Python ortamında OpenCV kütüphanesi ve frameworklerini kullanarak Hand Detection problemini çözebileceğimizi biliyorduk. Lakin bunu Unity ortamıyla nasıl buluşturabiliriz bunu araştırmamız gerekti.

Araştırmaların sonucunda UDP protokolünü kullanarak socket programlama ile port üzerinden Unity ile iletişim kurup kameradan alınan görüntünün koordinat bilgileri işleyip karakterimizin buna tepki vermesini amaçladık.

Teknolojiler :

- Python 3.11 versiyon
- OpenCV Mediapipe ve CvZone
- UDP Protokolü

Proje Takibi

Bir grup çalışması olması sebebiyle bir sürüm kontrol sistemi kullanmamız gerekiyordu. Bu sebeple bir **Github Reposu** oluşturuldu. Yapılan geliştirmelerin ve commitlerin takibi ise grup üyelerinin projedeki ilerleyişini görebilme ve herhangi bir grup üyesinin gelişim sürecinde geri kalmaması için faydalı bir çözüm oldu.

The screenshot displays a GitHub repository interface for a project named 'ForestRunner'. The top section, titled 'Commits', lists recent changes with details such as the commit message, the user who made the change, and the time elapsed since the commit. The commits listed include 'handtracking v2?', 'Rotation Modifikasyon', 'Update README.md', 'Main Menu - Buy Health Button SFX', 'Main Menu - Music ve Buttonclick SFX', 'Diamond collectable Music', and 'Level-01 Background ve Deathpanel Music'. Below the commits, a table provides a summary of the repository's files and their commit history. The bottom section shows the 'README' file, which contains information about the project, including the Unity version used (2022.3.11f1) and a list of assets used in the game. The README also lists the contributors to the project: kamahvari Onur Doğan, VersionTV Hasan Arasuloğlu, and CerenDemirhan.

File	Commit	Time
ForestRunner	handtracking v2?	16 hours ago
README.md	Update README.md	2 days ago
gameovermenu.psd	Homebutton	2 weeks ago
healthbutton.psd	HealthSystem UI	last week
main.py	handtracking v2?	16 hours ago
sondurum.py	medscape	2 weeks ago

ForestRun

Unity Version : Unity 2022.3.11f1

Asset Store

- 1- Low-Poly Simple Nature Pack
- 2- Free Low Poly Nature Forest
- 3- Low Poly Nature - Free Vegetation
- 4- Low Poly Gems
- 5 - Quantum UI
- 6- FREE Stylized Bear - RPG Forest Animal Animation and Character ([for Main Menu])

Contributors

- kamahvari Onur Doğan
- VersionTV Hasan Arasuloğlu
- CerenDemirhan

Languages

- C# 62.2%
- ShadersLab 22.2%
- HTML 4.7%
- VBScript 1.0%
- Python 1.1%

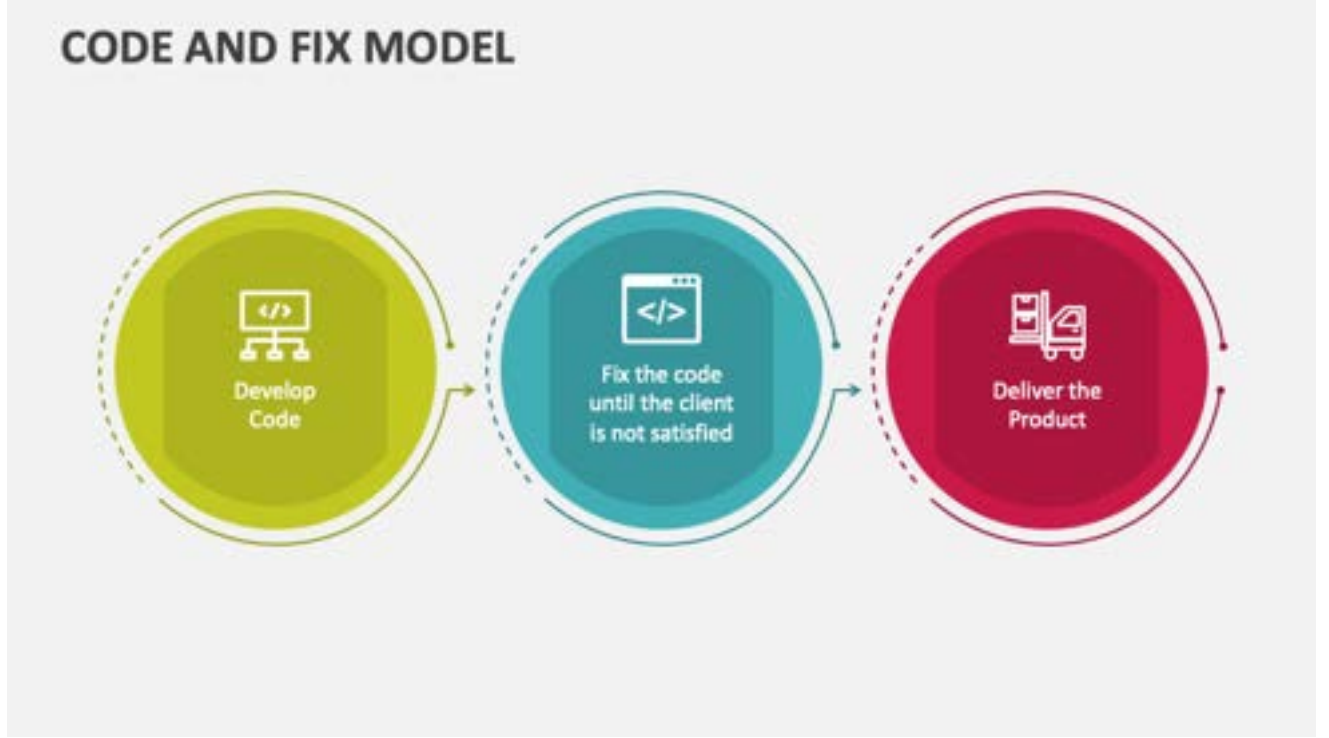
C- Proje Yol Haritası

Öncelikle hiçbir araştırmaya dayanmadan bir fikir olarak ortaya çıkan oyun geliştirme isteği üzerine bunu görüntü işlemeyle bağdaştırarak geliştirme kararı aldık. Lakin grup üyelerinin teorik bilginin ötesinde bir oyunun nasıl geliştirileceğine dair tecrübesi olmadığı için belli bir süreyi araştırmaya ayırmamız gerekti. Unity hakkında ve yapacağımız geliştireceğimiz oyun türünün muadillerini incelemeye başladık ve uzun süren araştırmalar gerçekleştirildi. Bunun yanında ikinci önemli başlığımız olan görüntü işlemeyi de Unity oyun motoruna nasıl entegre ederiz sorusuna cevap aramaya başladık.

Daha önceden tecrübemizin olmaması ve grup çalışmasının da yönetilebilir ve takip edilebilir olması açısından **Yazılım Geliştirme Süreci (Software Development Cycle)** adımlarına uyarak ilerlemeye karar verdik.

Bu yazılım geliştirme döngüsünün ilk aşaması **“Planlama”** evresi ve bu aşamada projenin genel hatlarıyla alakalı grup üyeleri arasında istişareler yapıldı. Önceliğimiz tabi ki konuya hakim olmaktı ilk adımda o yüzden belli bir zamanı araştırmayla geçirdikten sonra genel hatlarıyla hakim olduğumuzu düşündüğümüz anda **“Analiz”** aşamasına geçildi. Analiz aşamasında yapılacakları listelediğimiz ve elimizde bulunan kaynağı da efektif kullanmak adına ve fikirlerimizin oluşabilmesi adına Asset Store araştırmaları başladı. Bir oyunun konseptinin ortaya çıkabilmesi için elinizdeki grafik, ses gibi materyallerin önemi büyüktür. (Bunu game design başlığı altında irdelleyeceğiz.) Sonra ki aşama **“Tasarım”** aslında analiz aşamasında ihtiyaçlar kağıda dökülürken bir yandan mantıksal bir tasarımda ortaya koyuyorduk.

Daha sonra bunu fiziksel bir tasarım olarak ortaya koymamız gerekiyordu. Ama konuya hakim olmamamız sebebiyle öncelikle spesifik olarak yazılım geliştirme modellerinden birini benimsememiz gerekti **“Code and Fix”**. Biz bu yaklaşımı biraz daha kendimize göre uyarladık tabi ki. Kodla ve Düzelt yaklaşımı plan ve analiz sürecini uzun tutmadan hemen işe koyulmak için ideal ama biz aksine üzerinde fazla durduk. Kodla ve Düzelt ile bulduğumuz nokta geliştirme sürecinin tamamen uygulamaya dayalı olmasıydı. Proje geliştirme süreci boyunca da sadece uygulayarak ilerleme kaydedildi. Oyun geliştirmek gibi küçük detayların büyük hatalar oluşturabileceği bir projede kodlayıp sonra hataları düzeltmeye çalışmak elbette yorucuydu. Çok fazla forum , makale okumak zorunda kaldık. Lakin bu sürecin bizi sürekli araştırmaya teşvik etmesi yeni kavramlara hakim olmamızı ve Unity’e dair birçok şeyi öğrenebilmemize olanak sağladı.



Projenin belli bir bölümüne kadar sadece oyunun geliştirilme süreciyle ilgilenildi daha sonra Görüntü işleme sisteminin dahil edilmesi için taraftan araştırmalara başladık. Öncelikle Unity içerisinde bulunan OpenCV kütüphanesini kullanmaya izin veren Plugin bulunuyor lakin ücretli olması sebebiyle bu plan rafa kalktı. Buna bir çözüm ararken UDP protokolüyle Socket üzerinden geliştirilecek bir Python Cvzone-Mediapipe ile Hand Tracking yapabilir ve kameradan alacağımız el hareketi koordinat bilgilerini socket üzerinden Unity'e transfer edebileceğimizi keşfettik. Bunun üzerine araştırmalarımızı ve uygulamalarımızı devam ettirirken birçok sorunla karşılaştık çünkü donanıma bağlı olarak bir takım gecikmelere sebep oluyordu.

“Gerçekleştirme” aşamasında tamamen teoride planlanan işlerin gerçekleştirilmesiyle uğraştık. Tabi ki ne kadar kağıda teorik olarak problemi için bir çözüm geliştirdeniz de uygulamada işler değişebiliyor.

Bunun yanında proje geliştirilmeye başlandığında **“Divide and Conquerer”** yaklaşımından uzaklaşmamaya çalıştık. Çünkü projede ilerledikçe kod satırları çoğaldıkça yönetemediğimizi fark ettik bu sebeple tüm sınıfları ve fonksiyonları olabilecek en basit şekilde tutmaya çalıştık. **“Camel Case”** gibi değişken,fonksiyon isimlendirme gibi standartlara dikkat ederek proje içerisinde bir standart yakalamaya çalıştık.

D- Game Design ve Level Design

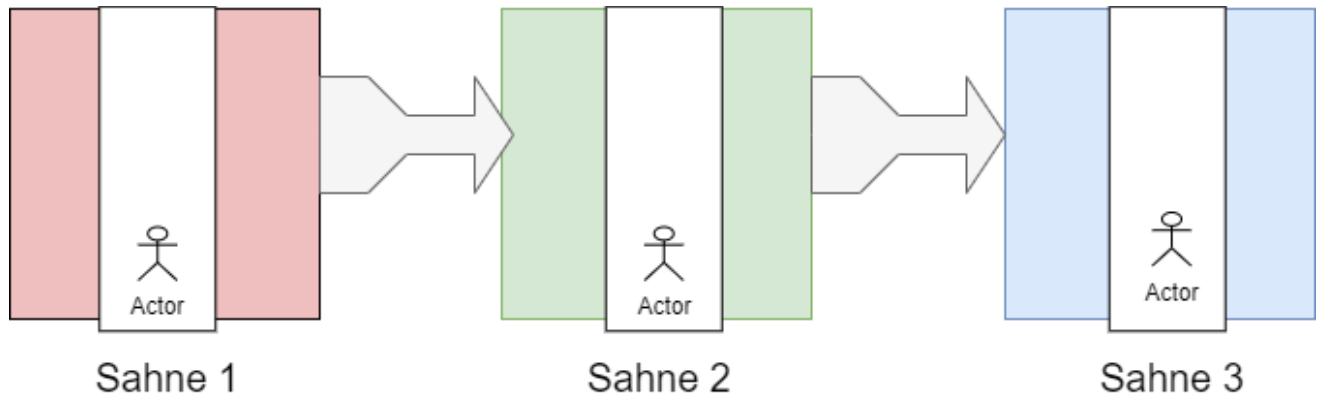
Oyun geliştirme dünyasına girdiğinizde karşınıza yeni terminolojiler çıkmaya başlıyor. **Game Design** ve **Level Design** da bunlardan biri. Aslında teorik olarak kalan ve analiz aşamasında aslında üzerinde çokça mesai harcamamız gereken kavramlar oldular.

Game Design aşamasını aslında oyunun göz önündeki bölümü olarak düşünebiliriz. Bunun için Asset Store'da harcadığımız mesaiye buna hizmet etti. Elimizdeki malzemenin ne olduğunu ve bununla ne ortaya koyabiliriz sorusuna yanıt bulmamız gerekiyordu. Oyunun konsepti ne olacak? Kontrolleri nasıl olacak , engel sistemi , ödül sistemi nasıl olacak? Oyuncuyu oyunda tutabilmek için neler yapılabilir gibi uygulamadan havada kalabilecek birçok soruya yanıt bulmanız gerekiyor. Oyunun mekanik, sistem ve kurallarını kağıt üzerinde oluşturduğumuz aşamaydı.

Oyun konseptini en başta belirlemiştik zaten **“3D Runner”** oyun türünde ve basit kontrolleri olan devamlılığı ve hızlı oynanabilirlik sunmak birinci önceliğimizdi. Oyun konseptine uygun assetleri ararken aklımıza yatan birkaç Asseti bulduktan sonra gözümüzde artık bir şeyler canlanmaya başlamıştı. Görerek ilerlemek her zaman daha etkili olmuştur o yüzden elimizde birkaç Assetin olması oyunun ismini de ortaya çıkarmıştı **“Forest Run”**.

Level Design ise ilerleyiş nasıl olacak ? Oyunu bir **Progress(süreç)** olarak düşünmemiz gerekirse bunun iki tarafı var. Geliştirici olarak iki farklı pencereden bakmanız gerekiyor. Bu oyunu deneyimleyecek kullanıcıların oyunda ilerlediklerini hissetmelerini sağlayacak unsurların dizayn edilmesi ve geliştirici tarafında bu çeşitliliği sağlayarak oyuncuya içerik sağlayabilmek.

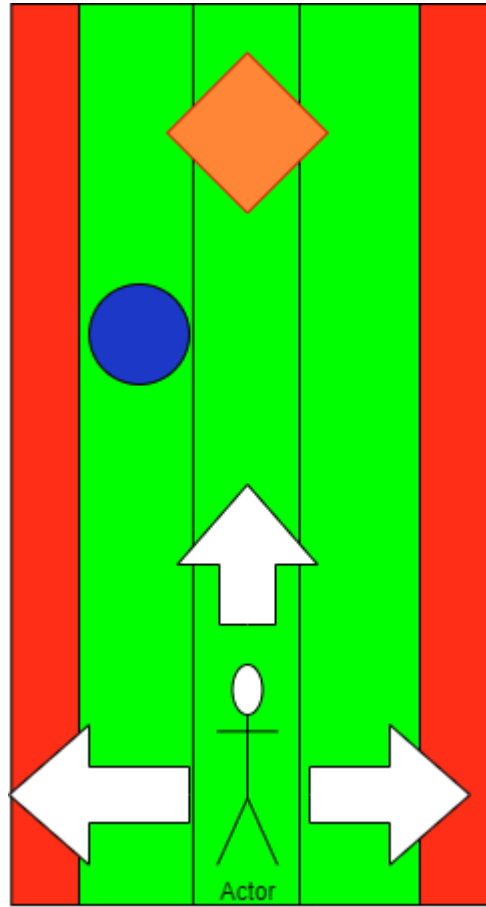
Bu aşamada bir level design ederken çevre etkenleri , mekanikler vs bunların biraz dışına çıkıp yukarıdan olaya bakmamız gerekirse bir oyunda birçok **sahne** bulunur.



Her bir sahne de ilerleyişi çeşitlendirmek adına farklı eventler de bulunmanız gerekmektedir. Çünkü kullanıcıyı ekran başında tutabilmek için merak uyandıran bir bölüm tasarımı yapılması gerekiyor.

Çoğu zaman gözden kaçan ama belki de bir oyunu oynanılabilir kılan parametrelerden biride mekanikler. Oyun mekaniklerini kompleks tutmaktansa oyuncu dostu olmasını sağlamak önceliğimiz olmalı. Daha yolun başındayken mekaniklerde zorlanan bir kullanıcıyı oyunda tutamazsınız. Bizim sıvrilmek istediğimiz noktada biraz bu sebeple ortaya çıkmıştı. Standart giriş/çıkış bir oyunda klavye ya da mousedan gelebilirken bunu daha interaktif tamamen kullanıcın görüntüsüyle ele almak istedik. Mekaniklere gelecek olursak karakterin iki farklı koordinat düzleminde ve üç farklı yönde hareket etmesine karar verdik.

X eksenini üzerinde sağa ve sola hareket eden karakterimiz ayrıca Y ekseninde zıplayarak ona zorluk çıkarmaya çalışan engellerden kaçınmaya çalışacak. Oyun mekanikleri gereği karakteri kontrol eden oyuncuya düşen sorumluluklar bundan ibaret. Lakin karakterimiz aralıksız koşan bir karakter bu da Z ekseninde sürekli pozisyonunun değişmesiyle gerçekleşiyor. Z eksenindeki hareketi kullanıcıdan bağımsız olarak gerçekleşen bir hareket yani kullanıcı müdahalesinin olamayacağı tek nokta.



I. BÖLÜM

Bu bölümde ilk ana başlığımız olan Oyun Geliştirme Süreci üzerinde durulacak. Süreç boyunca yapılan geliştirmelerin ve karşılaşılan problemlere bulunan çözümleri açıkladığımız bu bölümde adım adım alt başlıklarla nihai sonuç olan oyunumuzu ortaya çıkarmak üzerinde duracağız. Oyun geliştirme sürecinde kronolojik olarak ilerleyeceğiz.

1. Karakter ve Karakterin Hareketi

1.1 – Karakterin Eklenmesi

Öncelikle karakterimizi oluşturmamız gerekiyor ve adım adım üzerine inşa ederek ilerleyeceğiz her şeyi. Daha önceden bahsetmiş olduğum karakter ve animasyonları mixamo üzerinden temin ettikten sonra bunu projeye dahil ediyoruz. Animasyonlar olarak elimizde 4 farklı animasyon bulunuyor bunları zamanı geldikçe kullanacağız.

- Stumble Backward
- Standard Run
- Happy IDLE
- Jumping



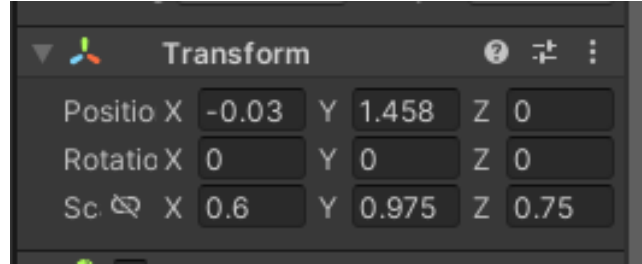
Tabi ki bu karakteri dahil etmeden önce bir Unity Objesi oluşturmamız gerekiyor ki üzerinde ayarlamaları rahatlıkla yapabilelim. Unity’de her şey Object ve Script ilişkileriyle ilerler. O sebeple Nesneye Yönelik Programlamanın çok iyi bir uygulaması diyebiliriz.



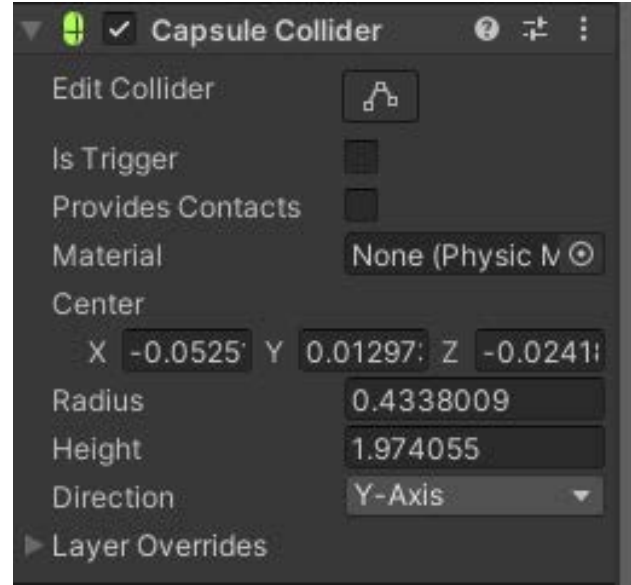
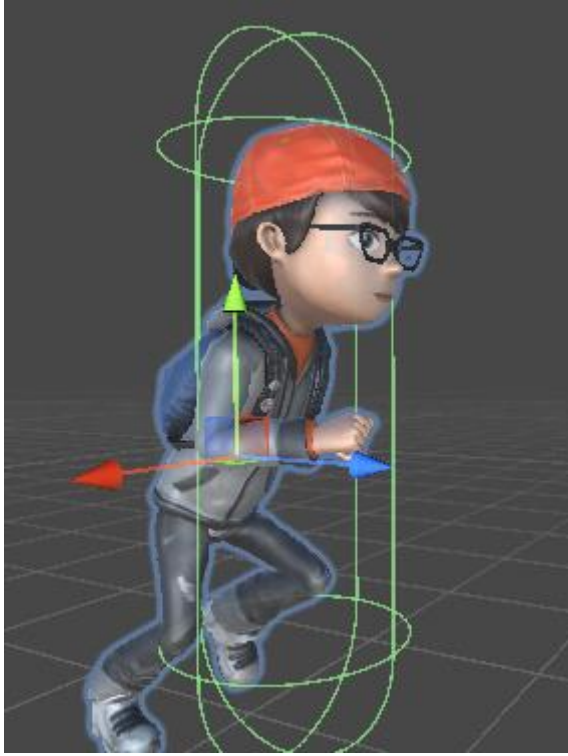
Unity’de iki önemli ana bölüm bulunuyor ; Hiyerarşi ve Inspector bölümleri. Hiyerarşi bölümünde sahneye sürüklediğiniz objelerinizi görebiliyorsunuz.

“Create Empty GameObject” diyerek bir GameObject oluşturduk. Bu GameObject nesnesi içerisine Prefab olarak indirdiğimiz karakterimizi **“Player”** Game objesi hiyerarşisi altında kalacak şekilde içerisine attık.

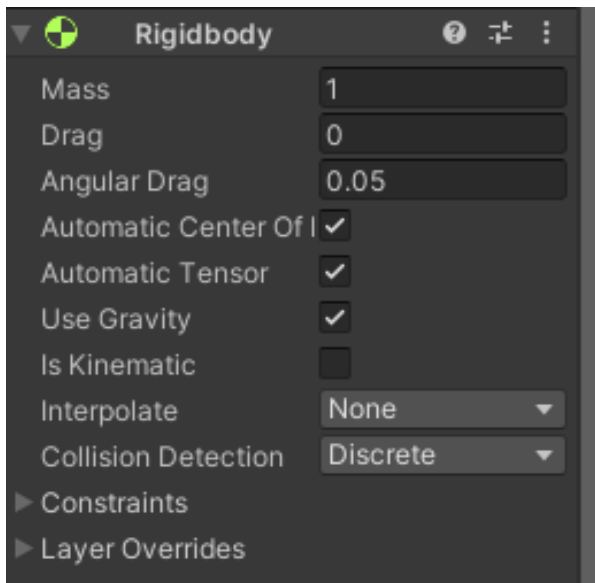
Diğer önemli bölümümüz olan **Inspector** bölümü ise objelerin detaylı ayarlamalarını yapabileceğiniz ve bu objelere “componentler” ekleyip bu componentleri üzerinde taşıyabileceğiniz objelerin özelliklerini ayarlamak için önemli bir sekmedir. Player objesine tıkladığımızda sağ tarafta gelen Inspector bölümünde her Game Objesinde olmazsa olmaz **“Transform”** bilgisini görebilirsiniz.



Transform bilgisi objenizin X-Y-Z eksenindeki pozisyonunu ayarlayabilmeyi, yine bu eksenlerde rotasyon yani açılarınızı ayarlayabilmeyi ve scale yani büyüklük gibi objenin temel ayarlamalarını yaptığınız önemli özelliklerden biridir. Birazdan bu özellik üzerinde bir takım işlemler yapmamız gerekecek çünkü karakterimizi hareket edecek daha doğrusu Player objesi hareket edecek ama onun altında olduğu için hiyerarşik olarak karakterin kendisinde hareket etmiş olacak.



İkinci önemli noktamız ise “**Collider**” kavramı. “Player” game objesi üzerine tıklayıp Inspector paneli üzerinden Component ekleyerek **Capsule Collider** ekliyoruz. Birçok collider türü bulunuyor ve adını şeklinden alıyor genelde. Fark ettiyseniz karakterimizin çevresini saran bir kafes görünüyor ve ona Collider diyoruz. Bunu eklememizin sebebi daha çok ileriye dönük yani karakterimiz engellerle ve toplanabilecek elmaslarla etkileşime girmesi gerekiyor ve bu Colliderlar “**Eventleri**” kullanabilmemizi sağlıyor. Şu olursa bunu tetikle gibi eventler tanımlamamıza olanak sağlıyor. Edit Collider diyerek colliderı daha nizami bir şekilde düzenleyebilirsiniz , karakterimiz sürekli ileri hareket ettiği için karşısına çıkacak engeller ve toplanabilir elmaslar ön tarafa bakan kısmından geleceği için colliderı ön tarafa daha yakın bir şekilde ayarladık.



Bir diğer önemli nokta ise “Fizik”. Oyun motorunun bize sağlamış olduğu componentlerden biri de “**Rigidbody**”. Karakterimize fizik elementlerini eklememiz gerekiyor. Çünkü gerçeklikten uzaklaşmayan ve bir zemin üzerinde durması gereken, üzerinde yer çekiminin etkisini de hissedecek bir karakter yaratmak istiyoruz.

1.2 – Karakterin Hareketi

İlk scriptimizi yazıp karakterimizi hareket ettirmenin vakti geldi.

```

PlayerMove.cs
Assembly-CSharp
1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Threading;
4  using UnityEngine;
5  using UnityEngine;
6  using UnityEngine.SceneManagement;
7  using UnityEngine.UI;
8
9  public class PlayerMove : MonoBehaviour
10 {
11
12     public static bool isAlive=true; //Karakter hayattamı
13     //Can Sayısı
14     public float ileriHiz = 3;
15     public float sagsagHiz = 4;
16

```

```

// Update is called once per frame
void Update()
{
    if (isAlive)
    {
        transform.Translate(Vector3.forward * Time.deltaTime * ileriHiz,Space.World); //Zamana bagli vector3 üzerinden hareket ; Spa

        if ((Input.GetKey(KeyCode.A)) || Input.GetKey(KeyCode.LeftArrow))
        {
            if (this.gameObject.transform.position.x > LevelSinir.solTrf) //levelsinirlari içerisinde kalmak şartıyla
            {
                transform.Translate(Vector3.left * Time.deltaTime * sagsagHiz);
            }
        }

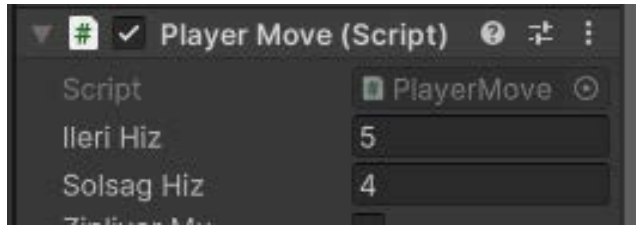
        if ((Input.GetKey(KeyCode.D)) || Input.GetKey(KeyCode.RightArrow))
        {
            if (this.gameObject.transform.position.x < LevelSinir.sagTrf)
            {
                transform.Translate(Vector3.left * Time.deltaTime * sagsagHiz * -1); //Tek değişken üzerinde sola-saga ivmelenmenin axi
            }
        }
    }
}

```

Öncelikle şundan bahsetmem gerekiyor **“MonoBehaviour”** . Unity’de sınıflar MonoBehaviour’dan türerler. Özellikle bahsetmiş olduğum Obje ve Script ilişkisi için MonoBehaviour’dan kalıtım yapmanız gerekir. Standart bir C# scripti olarakta kullanabilirsiniz tabi ki ama objelerle ilişki kurarak genelde ilerlenir Unity’de.

İlk olarak “**static**” bir değişken tanımladık “**isAlive =true**” başlangıç değerini verdik. Çünkü bu karakterimiz bir engele çarpıp oyun sonu gelecek o sebeple bazı hataların önüne geçmek için bu kontrolü yapmamız gerekiyor. Karakter hayatta mı ? Yani oyun içerisinde hareketini sürdürüyor mu ? Sorusunu bir nevi soruyoruz.

“**Public**” float türünde “**ileriHiz ve solsagHiz**” değerleri tanımlanıyor. Burada public olmasına dikkat edin. Bunun sebebi sadece sınıflardan ulaşmak için değil aynı zaman Inspector panelinden de bu değişkene ulaşmanıza imkan sağlıyor bu şekilde. Public olarak tanımladığım bu değişkenleri ben artık panel üzerinden istediğim gibi değiştirebilirim. Sürekli kodu açıp değiştirmeme gerek kalmıyor. Unity’de aynı şekilde [**SerializedField**] de bunun için kullanılıyor.



Buradaki **ileriHiz** değişkeni kesintisiz bir şekilde Z ekseninde hareket edeceğiz ve hareket ederken bir hız kat sayısı olarak kullanmak için tanımladık. **solsagHiz** değişkeninde X ekseninde karakterimizin sağa ve sola hareket ederken ki hareket hızları kat sayıları olarak kullanıyoruz.

MonoBehaviour’dan türeyen Unity Classları default olarak bir script açılırken ;

Start() ve **Update()** fonksiyonları oluşur. Kısaca bahsetmem gerekirse Start() fonksiyonu run time da çalıştırılacak kısım iken Update() fonksiyonu her “**frame**” de çalışacak fonksiyondur. Bunu bir objeyle ilişkilendirdiğinizde 30 FPS olarak düşünürseniz saniyede 30 kare için 30 kere çalışacak bir Update() fonksiyonu elinizde olmuş olur.

Update() fonksiyonu içerisinde ilk olarak “**isAlive=true**” kontrolü yapılıyor true ise içeri giriliyor. Daha önceden “**transform**” özelliğinden bahsetmiştik. Bu **PlayerMove.cs** scriptini Player objesine bağladık Component ekleyerek. O sebeple burdaki “**transform**” player objesinin transformunu belirtiyor.

Transform.Translate() ile karakterin değiştiriyoruz. “**Vector3**” , X-Y-Z’yi üzerinde tutan bir property “**forward**” yani ileri hareketi için Vector3 koordinatını değiştiriyoruz. “**Time.deltaTime**” zamana bağlı olarak hareketin devam edebilmesi için deltaTime ile çarpılır ve karakterin hız katsayısı olan “**ileriHiz**” ile çarpıyoruz. Karakterin hızı yavaş gelirse artık Inspector penceresinden direkt olarak değiştirebilirsin bu şekilde kolayca karakterin hızını ayarlayabiliriz.”**Space.World**” ifadesi ise default olarak 0 değeriyle sahne koordinatından başlar. Update() ile her framede ileri doğru hareket edecek bu karakter. Yani Z ekseni değişecek sadece. Burada kullanıcının bir müdahalesi olamayacak sabit olarak ileri sürekli hareket edecek.

Sağa ve Sola hareket etmek için ise iki adet IF kontrolü yaptık.

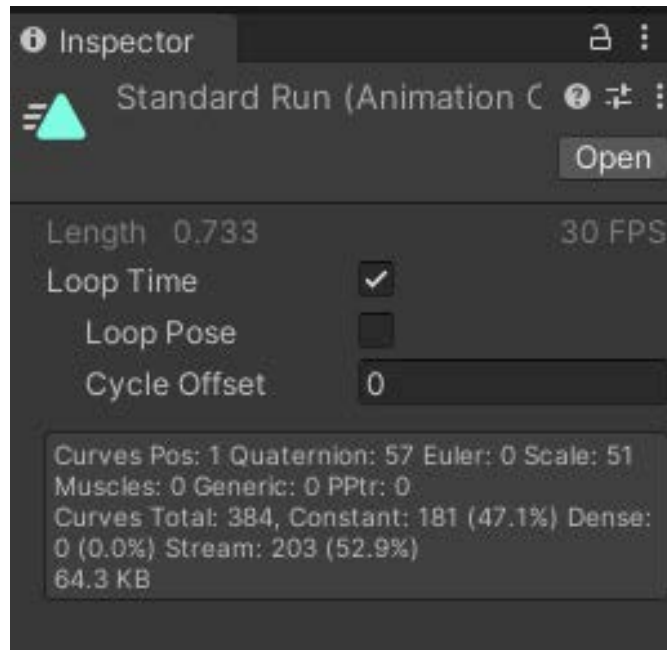
Input.GetKey() ile klavyeden basılan tuşu alıyoruz. A ve Sol Ok basınca sola hareket etmesini istiyoruz. Bir başka IF bloğu görüyoruz burada. **“this.gameObject.transform.position.x”** ifadesi “this” dediği scriptin bağlı olduğu Player obesi ve onun transform bilgisinden X eksenini alıyoruz ve **“LevelSinir.solTrf”** büyükse Sola hareket edeceğiz. LevelSinir diye bir script daha yarattık onun içerisinde ise karakterimizin bir zemin üzerinde hareket etmesini istiyoruz ama sağa ve sola giderken bir sınır belirlemezsek istediği kadar sağa ve sola gidebilir. Bu scriptte o sınırları belirliyoruz. İlerledikçe o sınıftan da bahsedeceğiz. Eğer bu iki if bloğunu geçerse karakterin transform pozisyonu değiştiriliyor ve “left” yani sola hareket ettiriyoruz ve bunu **“solSagHiz”** değişkenini kat sayı olarak kullanıyoruz.

Aynı işlemleri sağ tarafa hareket etmek için de yapıyoruz. D ve Sağ Ok tuşuna basınca sağa hareket edecek karakterimiz. Lakin burda tek farklı nokta transform pozisyonunun “-1” değeri ile çarpıldığını görüyorsunuz. solSagHiz olarak tek bir değişken tanımladık sağa ve sola hareket etmek için burda X ekseninde sağa ve sola giderken tam tersi olacağından -1 ile çarpıyoruz çarpmazsak eğer Sol tarafa hareket edecektir karakter sürekli.

1.3 – Karakterin hareketi ve Koşma Animasyonu

Karakterin “Standard Run” animasyonu otomatik olarak üzerinde çalışıyor peki nasıl? “Standard Run” animasyonunu indirirken karakter nesnesi + standard run bir arada eklentili bir şekilde “Collada” denilen formatta indirildi. “.dae” uzantılı bir format bu ve karakter objesinin içerisinde inmiş oldu. Sadece inspector panelinden “Standard Run” animasyonuna “Loop” etsin diye işaretlemeniz gerekecek. Bu sayede sürekli bu animasyonu tekrarlayacak.

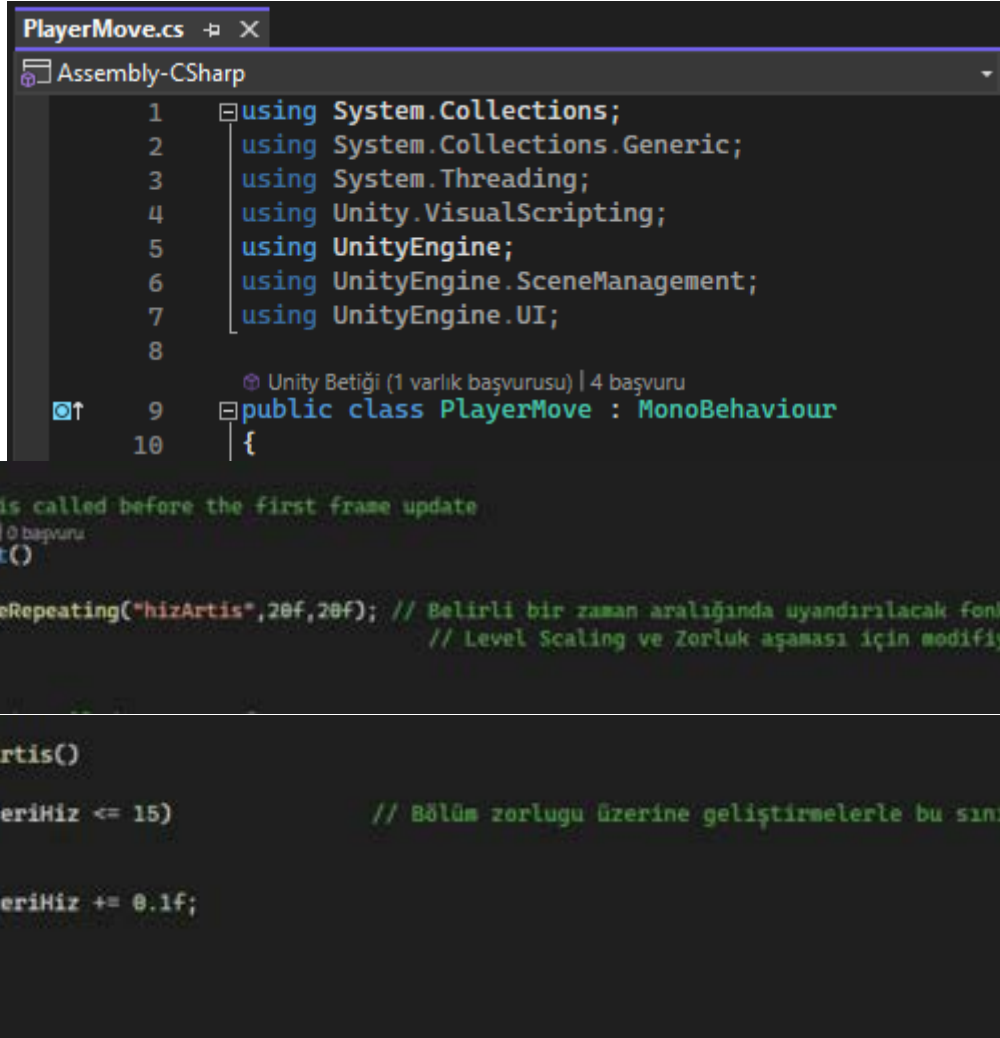
Daha sonra loop yapılmayacak ve herhangi bir olay da tetiklenecek animasyonlar için **“Animator”** denen özel bir sekmeyi kullanmamız gerekecek.



1.4 – Karakterin Hızının Arttırılması

Burada level design için ilk adımı atıyoruz. Karakterin “ileriHiz” diye belirlediğimiz sabit bir hızla ilerliyor. Bir süre sonra oyuncuyu zorlamamız gerekiyor ve bunu sadece karşısına engeller çıkarak yapamayız. Engellerde oyuncuya problem çıkaracak unsurlardan biri lakin hızı da arttırsak karakteri kontrol etmek zorlaşırken karşısına daha çok engel çıkmış olacak ve bunlardan kaçmak zorunda olacak.

“PlayerMove.cs” scriptine birkaç satır ekleyerek bunu halledebiliriz.



```

1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Threading;
4  using UnityEngine;
5  using UnityEngine.SceneManagement;
6  using UnityEngine.UI;
7
8  // Unity Betiği (1 varlık başvurusu) | 4 başvuru
9  public class PlayerMove : MonoBehaviour
10 {
11
12     // Start is called before the first frame update
13     // Unity İletisi | 0 başvuru
14     void Start()
15     {
16         InvokeRepeating("hizArtis", 20f, 20f); // Belirli bir zaman aralığında uyandırılacak fonksiyon ; 20sn sonra ilk uyandırma -
17                                                // Level Scaling ve Zorluk aşaması için modifiye edilebilir - geliştirilebilir.
18     }
19
20     // 0 başvuru
21     void hizArtis()
22     {
23         if (ilerihiz <= 15) // Bölüm zorluğu üzerine geliştirmelerle bu sınır da değiştirilebilir fakat oynan
24         {
25             ilerihiz += 0.1f;
26         }
27     }
28 }

```

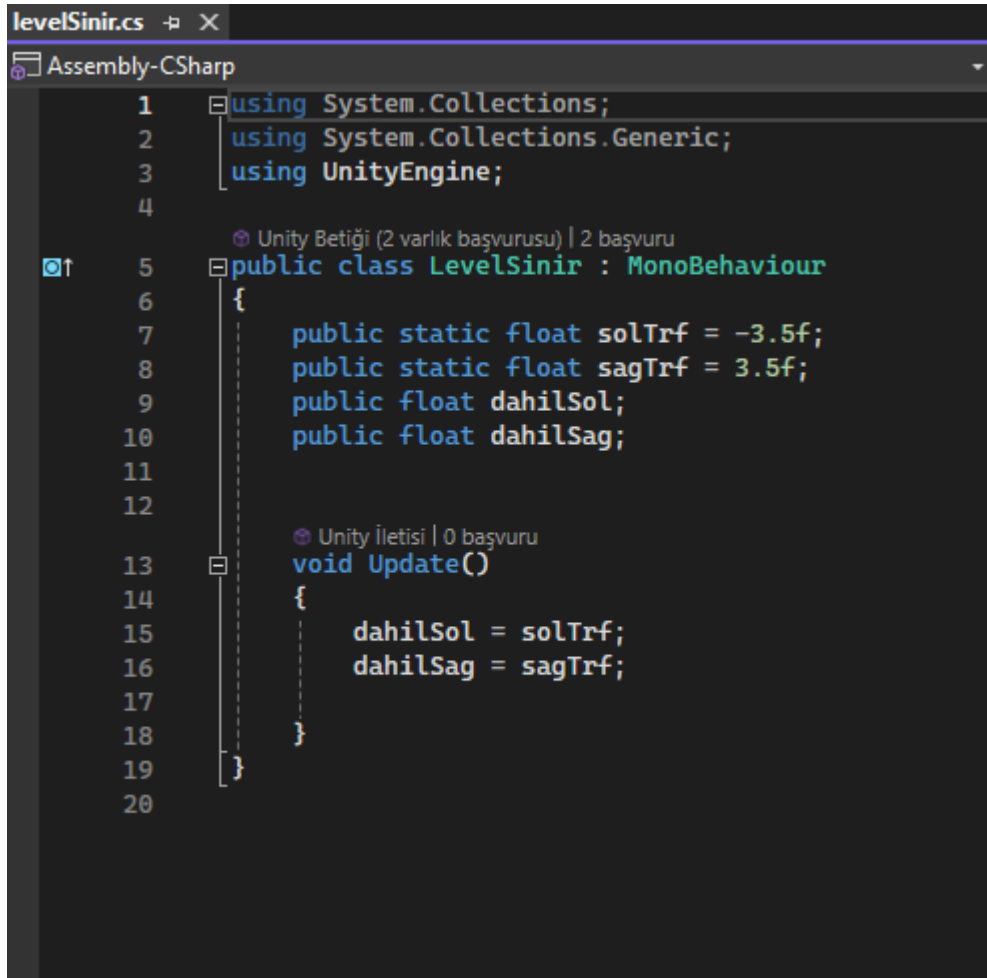
Start() fonksiyonunda daha önce bahsetmiştim. Proje run time geçtiğinde obje oluşturulduğunda ilk çalıştırılan fonksiyondur. Burada “InvokeRepeating()” fonksiyonunu kullanıyoruz. Bu fonksiyon “hizArtis()” fonksiyonunu çağırıyor. İkinci parametre 20f saniye sonra bu fonksiyon uyansın ve üçüncü parametrede 20f sonra tekrar çağırılsın bu fonksiyon anlamına geliyor. hizArtis() fonksiyonunda ise bir if bloğu oluşturduk çünkü oyun ilerledikçe karakterin hızının da sürekli artmasını istemiyoruz çünkü belli bir noktadan sonra hiç oynanamayacak bir hale gelebilir oyun. O sebeple birkaç test yaptıktan sonra en ideal hız sınırının 15f olduğunu tespit ettik. “ilerihiz +=0.1f” yani ilerihiz katsayısını 0.1f olarak arttırıyorum sürekli ta ki 15f’e gelene kadar ve 15f de sabitliyoruz.

Oyun başladıktan 20f saniye sonra ilk kez hizArtis() fonksiyonu çalışacak ve başlangıçta belirlediğimiz ileriHiz = 5f ; değeri → 5.1f olacak.

1.5 - Karakterin Hareket Edeceği Zemin Sınırlarının Belirlenmesi

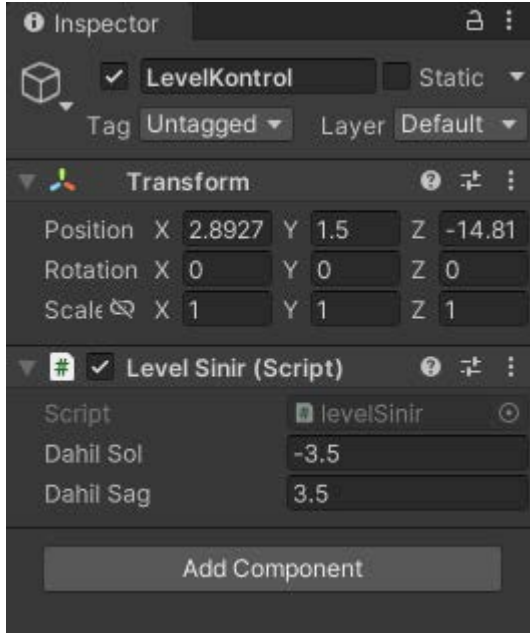
Daha önceden karakterin sola ve sağa hareket ederken if bloğu içerisinde kullandığımız “LevelSinir” denen scriptini oluşturacağız. Bu sayede karakterin sağa ve sola giderken istediği kadar kadar gidemeyecek sadece belirlediğimiz zemin içerisinde kalmak zorunda kalacaktır.

Öncelikle Hiyerarşi altında bir Empty GameObject oluşturduk ve ismini “LevelKontrol” yaptık. Bu LevelKontrol objesine Component ekledik. Component olarak “LevelSinir.cs” adını verdiğimiz script dosyasını ekledik. Bu script dosyasının içeriğini inceleyelim ;

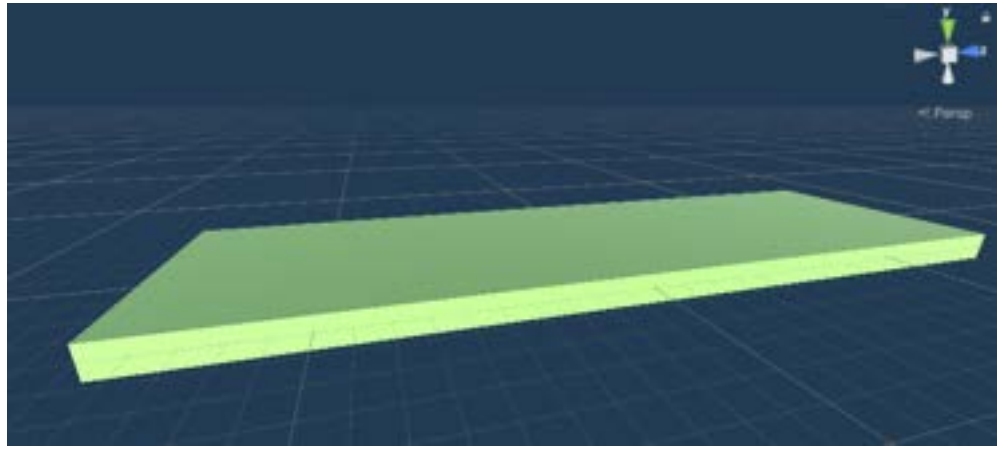


```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class LevelSinir : MonoBehaviour
6  {
7      public static float solTrf = -3.5f;
8      public static float sagTrf = 3.5f;
9      public float dahilSol;
10     public float dahilSag;
11
12
13     void Update()
14     {
15         dahilSol = solTrf;
16         dahilSag = sagTrf;
17     }
18 }
19
20
```

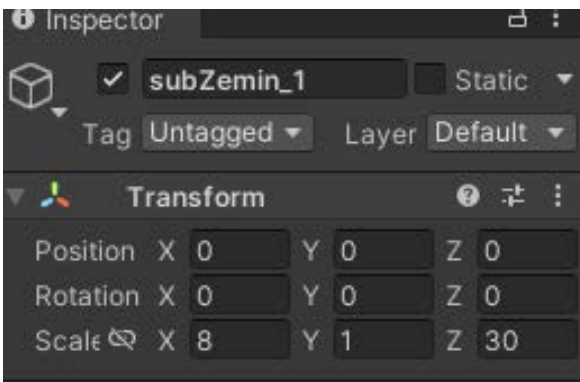
Öncelikle “**static**” **solTrf** ve **sagTrf** = $-3.5f / 3.5f$ değişkenlerini tanımladık. Burada “**public**” iki tane değişken daha tanımladık ki ve bu static olarak tanımladığımız değişkenlerin değerini bunlara atadık. **dahilSol** ve **dahilSag** değişkenlerine bu static değişkenlerin değerleri Update() fonksiyonunda aktarılıyor. “Static” değişkenler run time boyunca RAM’i işgal eden ve bu sayede bu değerlere müdahale edilmeden değiştirelemeyeceğini garanti altına alıyoruz.



LevelKontrol objesinin üzerine Component olarak eklediğimiz “LevelSinir.cs” scriptini ekledik ve Inspector panelinden de bunu görebiliyoruz.



Bu değerleri nasıl belirlediğimize gelirse karakterimizin hareket edeceği zemini belirledik tabi ki bu en sade hali. Bu zemin üzerinde birçok dizayn yapılacak. Bu zeminin inspector panelinden “**transform**” bilgisine bakarsak eğer ;



Zemin Scale bilgilerine bakacak olursanız $X = 8$ yani sağa ve sola doğru genişliği 8 birim , $Y=1$, $Z=30$ yani 30 birim uzunlukta olacak.

$dahilSol = -3.5f$ ve $dahilSag = 3.5f$ bunları mutlak değer içerisinde toplarsak eğer $7f$ yapıyor zeminin sınırlarına karakterin gelmesini istemiyorum o sebeple $0.5f$ bir mesafeyi kırpılmış olduk karakterin hareketinde.

1.6 - Karakterin Zıplaması

Karakterimizin son hareket eksenini olan Y yani karakterin zıplaması gerekiyor ve bunun yanında karakterin koşma animasyonu olarak “Standard Run” kullandık ve loopa aldığımız için onun üzerinde de işlem yapmamıza gerek yoktu. Şimdi “Jumping” animasyonunu da script içerisinde ulaşmayı ve zıplamak için klavyeden basılan tuşa göre tetiklenecek animasyon olarak kullanacağız.

```

PlayerMove.cs
Assembly-CSharp

1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Threading;
4  using UnityEngine;
5  using UnityEngine.SceneManagement;
6  using UnityEngine.UI;
7
8
9  public class PlayerMove : MonoBehaviour
10 {
    public bool zipliyorMu = false; //default -->> Zıplama 2 aşamalı ; yükselme ve düşme
    public bool dusuyorMu = false;

    [SerializeField] float ziplamaKuvveti = 3f;
    [SerializeField] float ziplamaAraligi = 0.45f;
    [SerializeField] Animator animator;

```

```

if ((Input.GetKey(KeyCode.W)) || Input.GetKey(KeyCode.UpArrow) || Input.GetKey(KeyCode.Space))
{
    if (zipliyorMu == false)
    {
        zipliyorMu = true;

        animator.SetTrigger("jump");//Jump Animasyonunu Tetikle
        zeminParticle.Pause(); // Zıplarken zeminparticle inaktif ediyoruz

        StartCoroutine(ZiplamaSirasi()); // Coroutine çalıştırarak belli bir süreyle tekrar tekrar çalışabilecek bir yapı
    }
}

```

PlayerMove.cs içerisinde öncelikle iki boolean değişken tanımlıyoruz. Tıpkı karakterin “isAlive” değişkeni gibi kontrol edilmesi gereken iki değişken tanımladık.

Zıplama işlevi aslında 2 fazlı bir işlemdir ; *Zıplama* ve *Düşüş*. Başlangıçta bu iki adet boolean değişkeni “**false**” tanımlıyoruz. Çünkü bir tetiklenme olmadan zıplamamızın bir anlamı yok zıplamıyorsa karakter düşmüyordur.

“**[SerializeField]**” tıpkı “public” gibi inspector penceresinden erişebilmemize olanak sağlayan “Attribute” tanımlanıyor.

“ **ZıplamaKuvveti**” tanımlıyoruz ki bir katsayı olsun. “**ZıplamaAraligi**” ise arka arkaya bir gecikme olmadan sürekli zıplama tuşuna basarsanız karakter hiç yere inmez o sebeple bir aralık belirlemek gerekiyor. “**Animator**” ise Unity sınıflarından biri ve bir animatör değişkeni tanımlandı ve buna inspector panelinden karakterimizi atamamız gerekiyor.

Update() fonksiyonu içerisine tıpkı Sağ ve Sol hareket için yaptığımız gibi klavye tuşlarını tanımlıyoruz. W-Yukarı Ok ve Space tuşuna basıldığında zıplasın karakterimiz. IF bloğundan içeri giriliyor ve o anda artık zıplıyoruz o sebeple “**zıpliyorMu = true**” yapılıyor.

“**animator.SetTrigger(“jump”)**” “ Animator sınıfından tanımlanmış animator değişkeni ile **SetTrigger()** fonksiyonu ile “**jump**” adındaki animasyonumuzu tetikliyoruz.

“**StartCoroutine(ZıplamaSirasi())**” → Burada öncelikle “Coroutine” bahsetmem gerekiyor. Coroutine belirli bir zaman aralığında çalışan fonksiyon olarak düşünebilirsiniz. Bu fonksiyonu başlattığımızı burda görebilirsiniz.

```
if (zıpliyorMu==true)
{
    if(dusuyorMu==false)
    {
        transform.Translate(Vector3.up * Time.deltaTime * zıplamaKuvveti, Space.World);
    }

    if(dusuyorMu==true)
    {
        transform.Translate(Vector3.up * Time.deltaTime * -zıplamaKuvveti, Space.World);
    }
}
```

Burda ise zıplama evresinin 2.fazı olan düşüş aşamasıyla ilgileniyoruz. “**Düşmek=false**” ise

“**Time.deltaTime * ziplamaKuvveti**” zıplama kuvveti katsayısıyla çarpılarak “**Vector3.up**” yani Y ekseninde yukarı doğru hareket ettireceğini gösteriyor.

“**Düşmek=true**” ise “**-ziplamaKuvveti**” (-) yapıyoruz çünkü düşerken tekrar zıplamaya basarsa yükselmesin karakter.

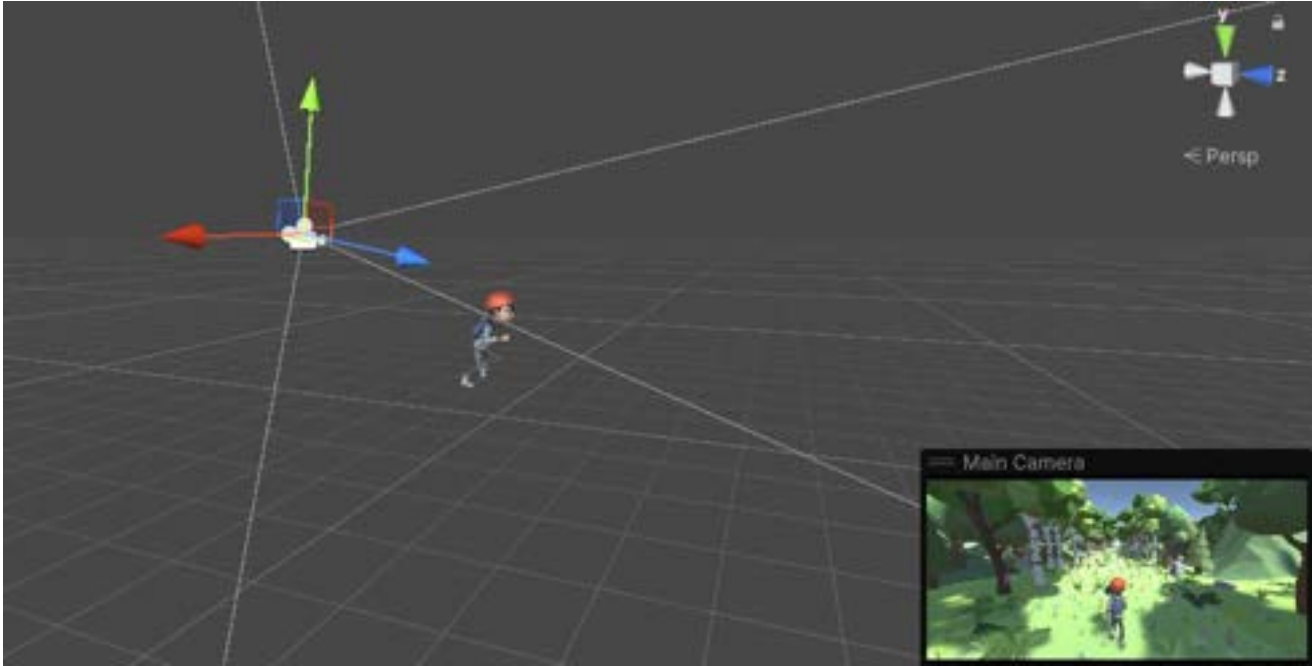
```
1 başvuru
IEnumerator ZiplamaSirasi()
{
    float ilkYukseklık = transform.position.y; //ilk yuksekligi almadığımızda karaktere ardarda z
    yield return new WaitForSeconds(ziplamaAraligi);
    dusuyorMu = true;
    yield return new WaitForSeconds(ziplamaAraligi); //belli bir süre beklemesini sağla
    zipliyorMu = false;
    dusuyorMu = false;

    transform.position = new Vector3(transform.position.x, ilkYukseklık, transform.position.z);
}
```

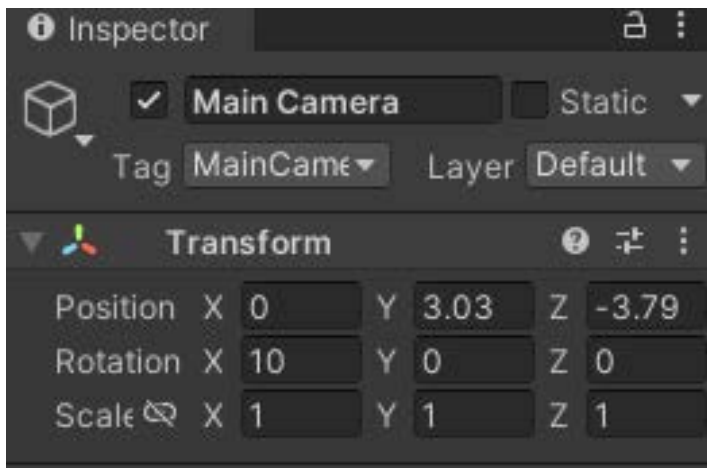
“**ZıplamaSirasi()**” denilen bu fonksiyon ise “**IEnumerator**” olan bir “**Interface**”. İlkYukseklık değerini tutmamız gerekiyor çünkü karakter tam zemine düşmeyebiliyor ve 0.45f aralıklarla zıplamak için tuşa bastığınızda kümülatif olarak kademe kademe karakter yükseliyor. Bu sebeple karakterin zıplamaya başlamadan önce ki Y koordinatını bir değişkende tutuyoruz. “**Yield return new WaitForSeconds()**” ile belli bir süre bekletiyoruz zıplamaAralığı kadar bekletmemizi sağlıyor. Daha sonra karakter zıplar ve düşmek = true olduğunda tekrar beklemesi için tekrar WaitForSeconds() fonksiyonunu kullanıyoruz ve başlangıçtaki ilk haline getiriyoruz bu iki boolean değişkeni = false yapıyoruz. Objenin pozisyonunu X ve Z’yi değiştirmiyoruz Player objesinin bilgileri neyse onu veriyoruz ama Y bilgisini “**ilkYukseklık**” yapıyoruz bu sayede Y pozisyonunda pozisyonunun değişmemesini sağlamış oluyoruz.

2. Kamera Nesnesi ve Kamera Takibi

Karakter hareket ediyor lakin Unity'nin default olarak oluşturduğu oyun içi kamera objesi sabit bir konumda duruyor bu nesneyi de karakter ile beraber hareket ettirmeniz gerekiyor. Kamera açılarına göre çeşitli tipleri bulunuyor. **FPS** ya da **TPS** dediğimiz ; “**First Person**” ya da “**Third Person**” kamera açılarıyla oyuncunun içinde bulunduğu dünyayı nasıl göreceğine karar vermek gerekiyor. Biz TPS kamera açısını tercih ettik ve karakterin arkasında belli bir yükseklikte karakteri takip etmemiz gerekiyor.

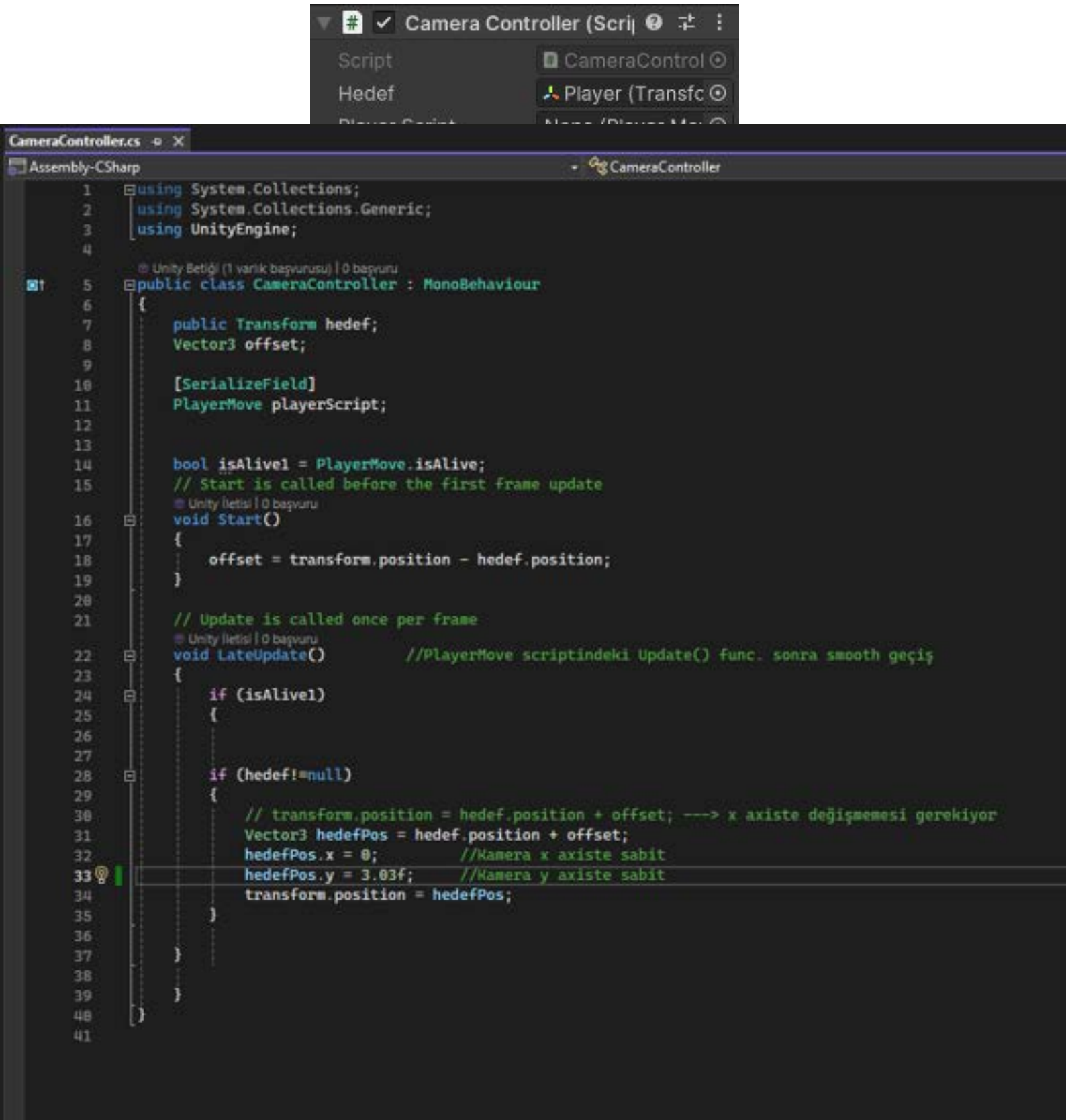


“Main Camera” adındaki kamera nesnesinin proje geliştirme ortamında ve oyun içinde kullanıcının oyunu nasıl deneyimleyeceğini sağ alt köşede görebiliyorsunuz. Bu kamera nesnesinin transform bilgilerini düzenleyerek ideal bir görüş açısı elde etmemiz gerekti.



Kamera nesnesinin bizim için ideal olan transform değerlerini görebilirsiniz. Lakin bu değerlerin korunması gerekiyor. Karakter ile Kamera objesi arasındaki mesafenin her bir framede korunması gerekiyor. Bu sebeple bir script oluşturmamız gerekiyor.

“CameraController.cs” diye bir script oluşturuyoruz. Bu script dosyasını “Main Camera” objesine component olarak ekliyoruz ve bu nesneyle ilişkilendiriyoruz.

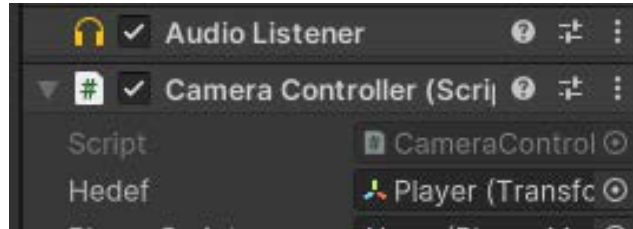


“**offset**” adında bir değişken tanımladık. Bu offset değeri (**Karakterin Konumu – Kamera Konumu**) iki obje arasındaki mesafeyi veriyor. Start() fonksiyonuyla birlikte kameranın pozisyon bilgisini Hedef olarak tanımladığımız ve inspector panelinden karakter objesini verdiğimiz karakterin pozisyon bilgisinden çıkartıyoruz ve aradaki mesafeyi bu offset içinde tutuyoruz.

“**LateUpdate()**” fonksiyonunda MonoBehaviour sınıfından türetilen classlar içinde gelen ve Update() fonksiyonunun bir çeşitidir. Update() fonksiyonundan sonra çalışacak bir fonksiyondur. Run time da işletilirken bu fonksiyonlar arasında böyle bir öncelik sırası bulunuyor. Update() ile karakter hareket ettikten sonra çalışsın ve pozisyon bilgilerini önce alsın sonra LateUpdate() içinde bu aradaki farkı kullanarak kullanmamız gerekiyor.

Fonksiyon içerisinde yine bir koşul oluşturuyoruz. Karakterin sahnede olup olmadığı kontrolünü yapıyoruz. **Vector3** hedefPos tanımlıyoruz ve “**hedef.position**” yani karakter nesnesinin pozisyonuna “**offset**” bilgisi ekleniyor. Burada bu bilgiler eklenirken **hedefPos.x** ve **y** bilgilerine ulaşarak 0 değerini atıyoruz. Çünkü kameranın X ve Y ekseninde pozisyonunun değişmesini istemiyoruz. Karakter sağa sola giderken kamerada onu takip etsin istemiyoruz çünkü pekte iyi bir görüntü elde edilmiyor. Ayrıca karakter zıplarken de kameranın karakteri takip etmesini istemiyoruz.

“**hedefPos.y**” = **3.03f** daha önceden inspector panelinde ayarlamalarla ideal bulduğumuz Y pozisyonunu aktarıyoruz ve değişmesini istemiyoruz. Kamera nesnesinin pozisyon bilgisine ekliyoruz daha sonra bu **hedefPos** üzerinde bulunan koordinat bilgilerini.



Bahsetmek için henüz erken ama kamera objesi üzerinde dururken bu objeye default olarak eklenmiş bir şekilde gelen “**Audio Listener**” componenti bulunuyor. Geliştirme sürecinin ileri aşamalarında müzik ve ses efektleri ekleyeceğiz. Tabi ki bu ses dosyalarını üzerinde taşıyacak **Audio Source** denilen bir component üzerinde taşıyacağız. Bu kaynakları dinleyecek bir dinleyici- listenera ihtiyacımız var haliyle.

3. Game Design ve Asset Kullanımı

Karakterin boş bir koordinat düzleminde koşması bir şey ifade etmiyor. O sebeple bölüm dizaynını yapmamız gerekiyor. Unity oyunlarının temelinde sahneler arası geçişler bulunur. İlk sahnemiz “**Level01**” bu sahneyi doldurmamız gerekiyor. Çevre, Engeller ve Toplanacak ödüller olmak üzere birçok unsur bulunuyor. Dizayn aşamasından sonra sahnede oluşturduğumuz bu objelerin karakter ile etkileşimiyle ilgileneceğiz.

Dizayn aşaması uzun bir süreç , bu sebeple iyi yönetilmesi gerekiyor. “**Prefab**” adında bir klasör oluşturduk proje dosyaları içerisinde. Oluşturacağımız tüm yapıları burda klasörleyerek düzenli bir şekilde tutacağız.

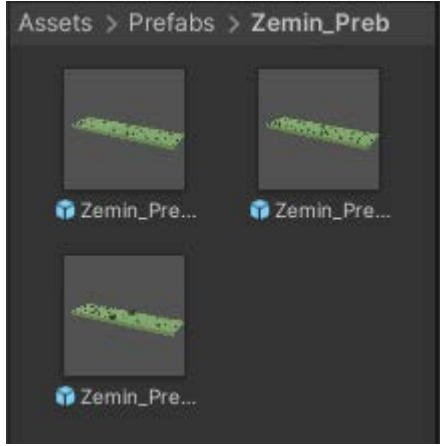
<Daha önce ilk bölümde bahsettiğimiz Assetlerin araştırılma sürecini geçtiğimiz ve artık bunların kullanılması evresine geldik. Basit bir noktadan başlamak gerekirse öncelikle karakterin boş ve düz yeşil bir **zemin** üzerinde karakterin hareket edeceğini söylemiştik. Şimdi bu zemini detaylandırmamız gerekiyor.



Projenin sonunda bu oluşturduğumuz Prefab klasöründe düzenli bir şekilde tutuyoruz tüm hazırladığımız prefabları.

Engeller, Çevre, Sectionlar, Zeminler , Diamond ve Particlelar için ayrı ayrı prefabları hazırlayıp bir noktada topladık.

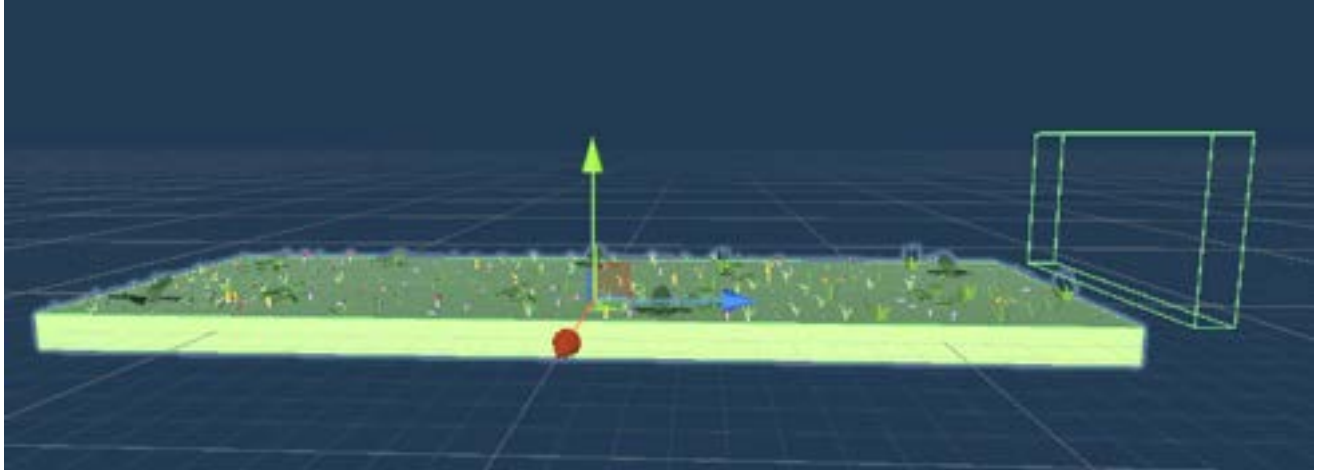
3.1 - Zemin Prefab



Birbirinden farklı birçok zemin prefabı hazırladık. Tabi ki bu süreci tarif etmek zor. Birçok assetle birleştirerek zeminlerimizi oluşturduk.

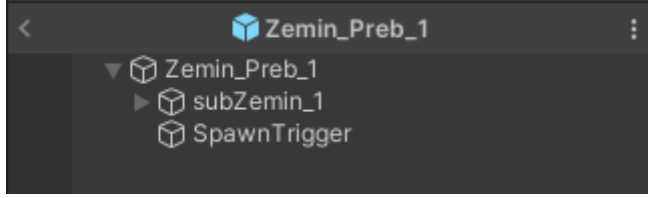
Bu zeminler üzerinde birden fazla Asset paketi içerisinde bulunan objeleri bu zemin üzerinde bir araya getirdik. İşin daha çok tasarım kısmı diyebiliriz.

Bu zemin örneklerinden sadece biri üzerinden gidecek olursak oyunun ilerleyişi için önemli noktaları burada inceleyelim.



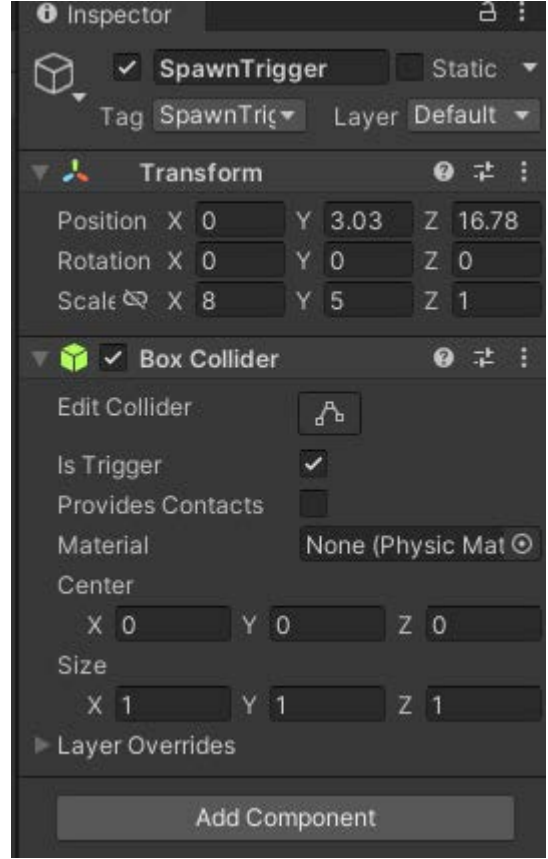
Zemin prefablarından birinin çalışma düzlemindeki halini görüyoruz. Birçok assetle bir araya getirerek tasarımlarını hazırladık. Burda dikkat edilmesi gereken zeminin sonunda bulunan “**box collider**”. Colliderı neden kullanıyoruz peki?

Sonsuza doğru koşan bir karakterimiz var ve hareket ettiği zemini sürekli sahneye sürükleyemeyiz. Bu sebeple bu objelerin otomatik yaratılması gerekiyor. Bunun içinde bu colliderdan geçen karakterimiz tetikleyecek ve yeni zemini spawnlayacak. Bu sayede sürekli arka arkaya zeminler peş peşe gelecek ve sonu gelmeyen bir oyun elde etmiş olacağız.



Zemin Prefab'ın içine bakacak olursanız “**SpawnTrigger**” adında bir GameObject yarattık. Bu GameObjenin inspector penceresindeki özelliklerini görüyoruz yan tarafta. “**Tag**” denilen takma bir isim olarak **SpawnTrigger** adını verdik bu collidera. Bu **etiketi** daha sonra scriptlerimiz içerisinde bu objenin tetiklenme mekanizmasını harekete geçirmek için ve karakterin buna karşı vereceği tepkileri yönetebilmek için kullanıyoruz.

“**Player**” nesnemizin inspector bilgilerine daha önceden bakmıştık. Orada colliderını hazırlamıştık. Collider özelliklerinden biri de “**isTrigger**” özelliği. Bunu daha önceden karakterimizin colliderında aktif etmemiştik.

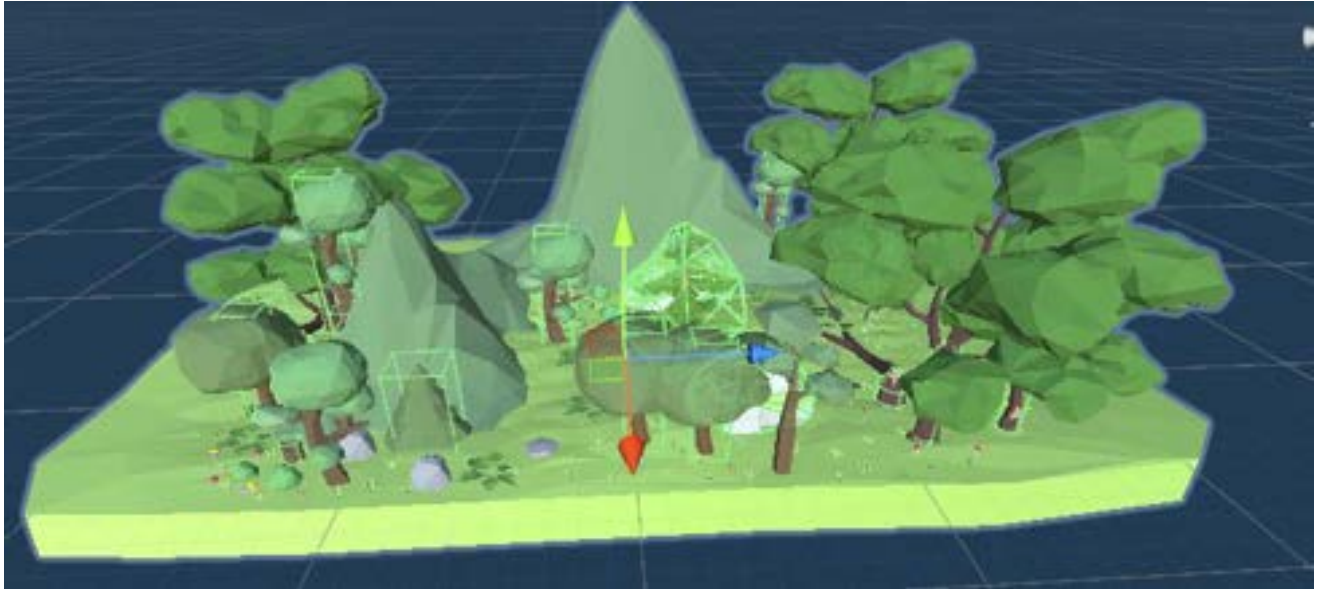
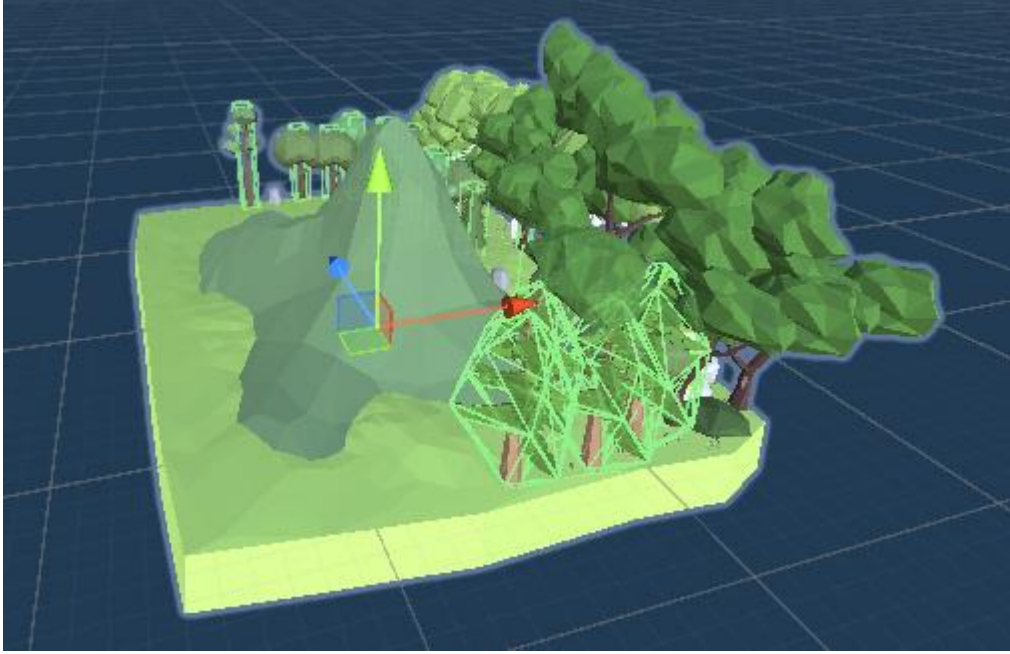


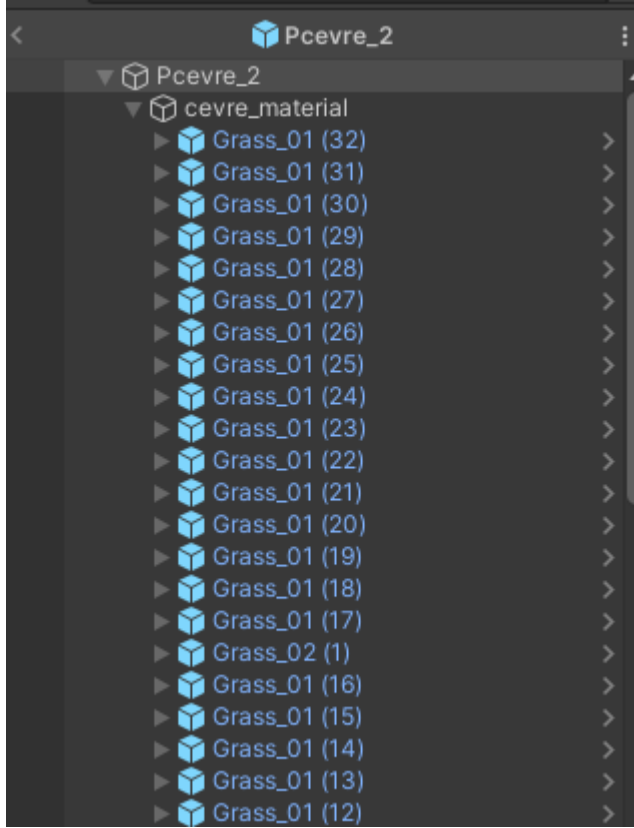
İki farklı colliderın birbirini tetikleyebilmesi için en az birinin “**isTrigger**” özelliğini aktif etmeniz gerekiyor. Birçok event tanımlayacağız o sebeple **isTrigger** aktif edildi.

Zemini oluşturduk ama karakterin görüş alanında sol ve sağ bulunuyor. Karakterin sol ve sağ tarafı bomboş bir şekilde duruyor bu sebeple çevreyide oluşturmamız gerekiyor. Bunları otomatize edecek scrippleri bir sonraki ana başlığımız altında inceleyeceğiz.

3.2 - Çevre Prefab

Şimdi ise çevre prefablerini oluşturuyoruz ve uzun süren bir tasarım aşamasında ilerliyoruz. Zemin üzerinde bulunan objelerden daha fazla objeyi bir çevre objesi üzerinde toplamamız gerekiyor. Zemin boyutu Z=30 boyutunda bir boyu bulunuyor hemen hemen örtüşecek şekilde bir boyut ayarlaması yaptık ve bu sayede her şey aynı anda spawn olacak ve yok olacak.





Hazırladığımız bir çevre prefabın içeriğine bakacak olursanız birçok nesne bulunuyor içerisinde. Tabi ki bu görüntü sonu gelmeyen bir liste gibi sadece küçük bir kısmını burada görüyoruz.

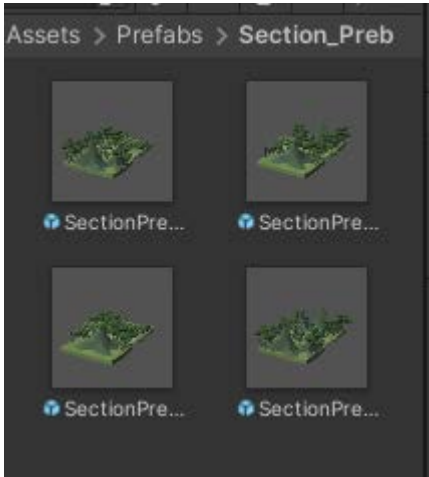
Buradaki tüm objelerin pozisyonlarını ayarlamak, karakterin arkasında hareket eden o kamera objesinin görüş açısının yakaladığı görüntüye kadar birçok detay işlemlerin gerçekleştirilmesi gerekti.

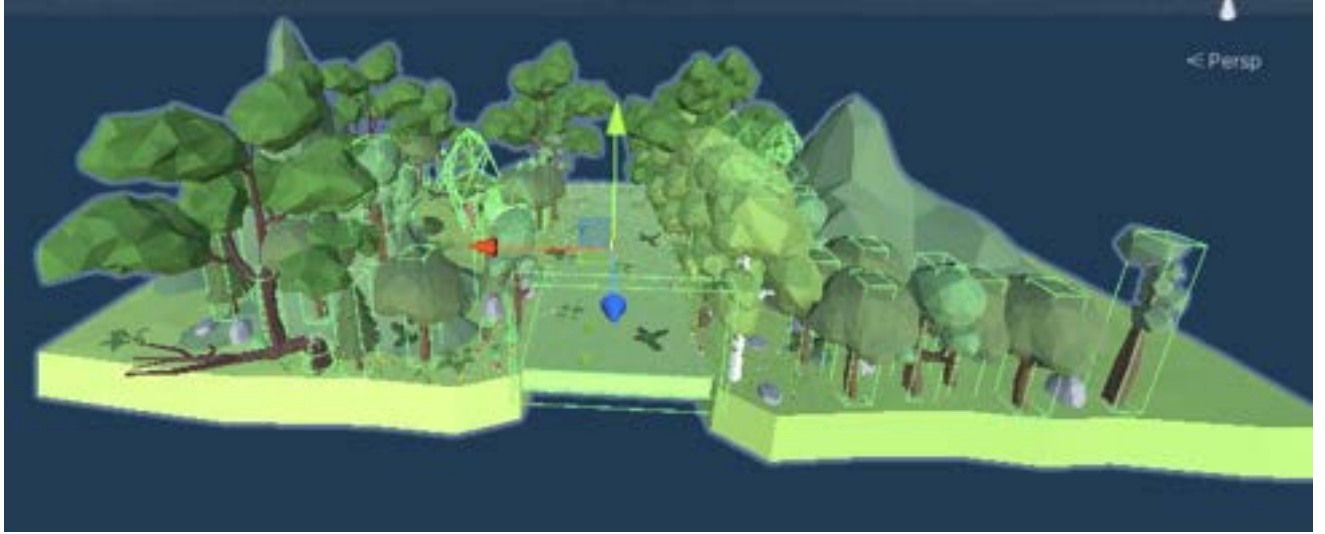
Oyun geliştirirken birçok objeyi sahneye bırakıyorsunuz ama işin **optimizasyon** tarafını da düşünmelisiniz. Günümüz teknolojisinde bellek miktarı yüksek olsa da geliştirici tekniklerine pek uyan bir yaklaşım değil. O sebeple bu objelerin yaratılmalarının yanında yok edilmeleri de gerekiyor. Buna zamanı gelince daha detaylı değineceğiz.

3.3 – Section Prefab

“**Divide and Conquerer**” yaklaşımından uzaklaşmak istemediğimizden birçok şeyi parçalara böldük ve kolay yönetilebilmesini sağladık. **Zemin** ve **Çevre** prefablerini oluşturduk ama bunları bir araya getirmemiz gerekiyor, ikisini birbirinden bağımsız düşünmemiz pek mümkün değil.

“**Section**” kavramı ise terminolojilerden biri. 360 derece karakterin gördüğü ve görmediği içinde bulunduğu ortam olarak tanımlayabiliriz. Bu sebeple Zemin ve Çevre prefablerini bir araya getirdik.

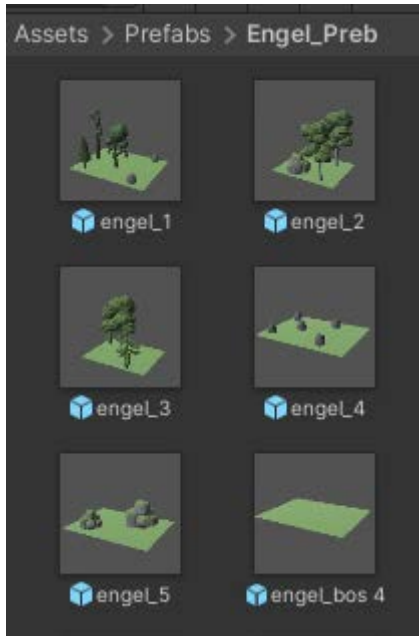




Tüm bu sectionları otomatik olarak üreteceğiz ve aynı zamanda yok edeceğiz. Yok etme işlemini yaparken yola çıkacağımız mentalite karakterin görüş açısından çıkan her nesneyi yok etmek olacak. Bu sayede kaynakları daha ideal kullanmış olacağız.

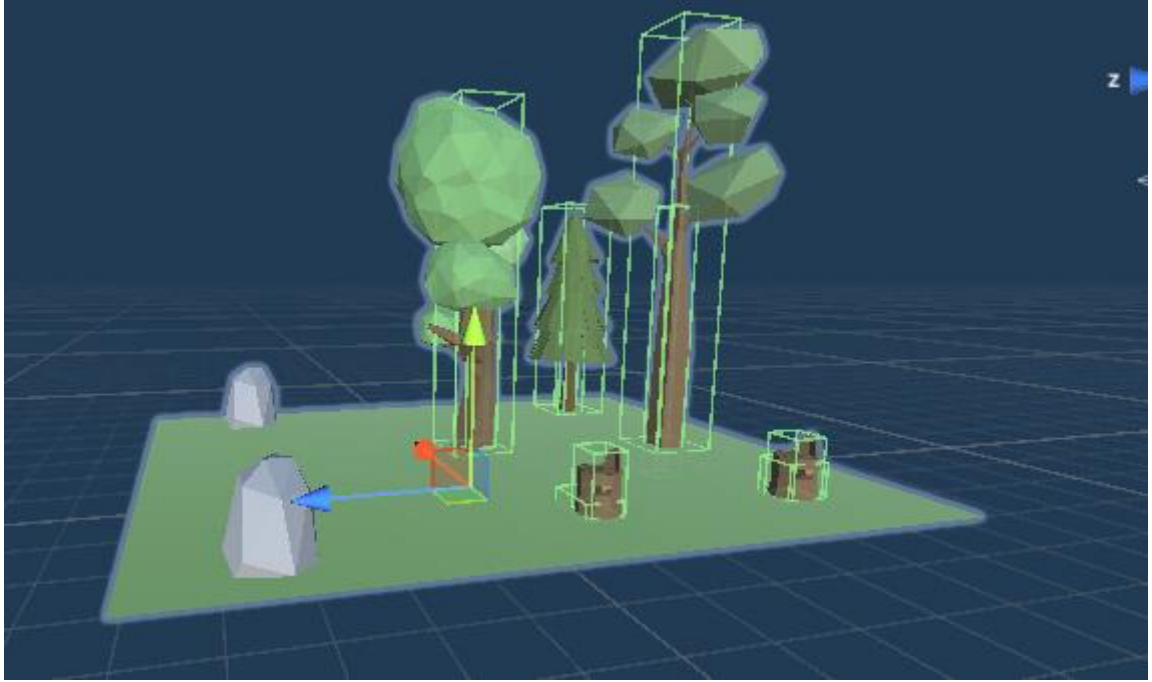
Zemin üzerinde tasarım için objeler bulunuyor ama karakterin ilerleyişini engellemeye çalışacak engellerinde olması gerekiyor. Bu engeller için prefablarda hazırlamamız gerekiyor.

3.4 – Engel Prefab



Biz engelleri rastgele oluşturmak istiyoruz. Çünkü oyunun tekrar oynanılabilirliğini artırmak istiyoruz. Sürekli aynı konumlarda aynı engellerin oluşmasını istemiyoruz. Prefab olarak hazırladığımızda sadece kendi dizayn ettiğimiz noktalar üzerinde çıksın ama spawn edilecek sırası tamamen rastgele olacak.

Rastgele zemin üzerinde belli bir sınır içerisinde bu hazırladığımız prefablar ortaya çıkacaklar. Bu sayede oyuncunun ilerleyişinin önünde durarak bir meydan okumanın içerisinde bulacağız kendimizi.

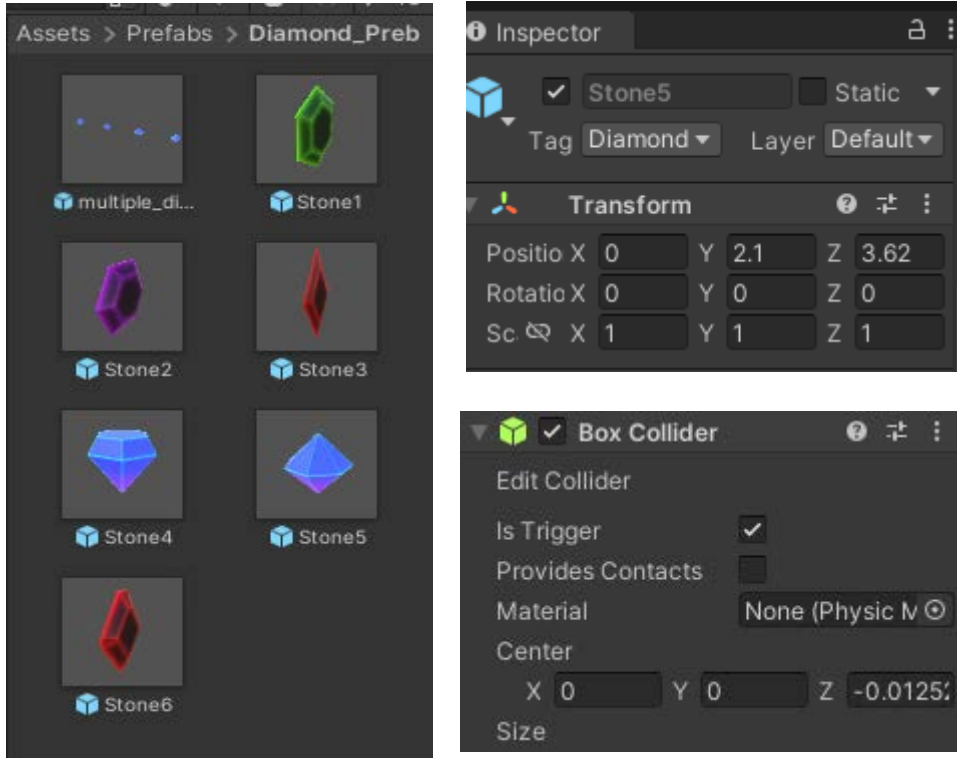


Zemin üzerinde ortaya çıkacak engel prefablerinden birini görüyoruz. Burada bulunan tüm objelere “**Tag**” atadık **Engel** adında hepsini aynı tag verdik ve her birine **collider** ekledik.

Bu tüm spawnlanma işlemlerini yönetebileceğimiz bir obje ve onun üzerinden birçok scripti component olarak ekleyerek tek bir noktadan parçalara ayırıp , basitleştirilen scriptleri bir kaynak üzerinden yöneteceğiz. Yönetilebilmesi bu sayede çok kolay olacak ve herhangi bir hata ile karşılaşmamız halinde takip edebilecek bir yapı sağlamış olacağız.

3.5 – Elmas Prefab

Oyuncunun odağını oyun için de tutabilmek için ödül sisteminin olması gerekiyor. Bu sebeple bunu **elmas** toplamak ve karakterin katettiği **mesafe** üzerinden oyuncunun başarısını derecelendireceğiz.



Diamond objelerimizin hepsine **box collider** ekliyoruz çünkü bunlara karakterimiz çarptığında bunları toplamış olacağız. Birbirinden farklı 6 adet diamond prefabı kullanıyoruz.

Bu diamond objesini otomatik X koordinatından rastgele oluşturacağız ama hiçbir efekti olmayan statik bir nesne olarak ortaya çıkacak. Diamond prefableri içinde script oluşturacağız ve efekt ayarlamalarını yapacağız. Elmas objesine de “**Diamond**” tagini verdik ve **isTrigger** özelliğın aktif ettik hepsinde. Scriptlerimiz de bu tagi kullanarak çeşitli eventleri tetiklememiz gerekecek.

Şu ana kadar işın daha çok tasarım kısmını gördük ve pozisyon bilgileriyle oynayarak ideal olanı bulmaya çalıştık. Şimdi bu işi otomatize etmemiz gerekiyor. Sırada ki bölümde üç ana başlığımız için rastgele ve belli bir sisteme bağlayarak oluşturduğumuz tüm prefableri oluşturacağız.

4. Oluşturulan Prefablerin Otomatik Üretilmesi ve Yok Edilmesi

Game design aşamasında birçok prefab tasarladık. Tabi ki geliştirme sürecinin bir bölümünü tamamen buna ayırmadık. Belli bir zamana yayarak yaptık bölümün dizayn edilmesini ve dizayn ettikçe scriptlerle bir araya getirdik.

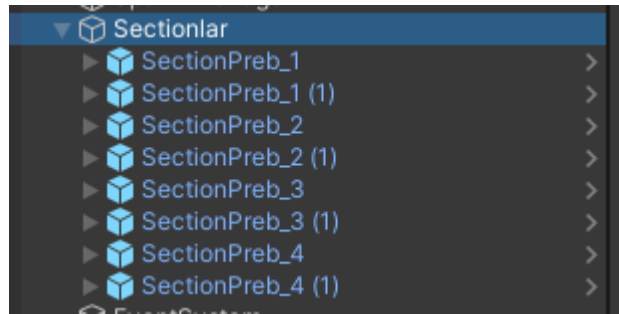
Otomatik olarak ortaya çıkaracağımız ve yok edeceğimiz bu prefablerin tek bir kaynaktan yönetilmesini istiyoruz. Bu sebeple bir obje yaratmamız gerekecek ama bu objeyi sahnede aktif olarak kullanmayacağız. Bu obje sadece bizim **spawn etme** görevlerimizi yürütecek.

Üç ana başlık altında inleyeceğiz bu otomatik oluşturma sürecini ;

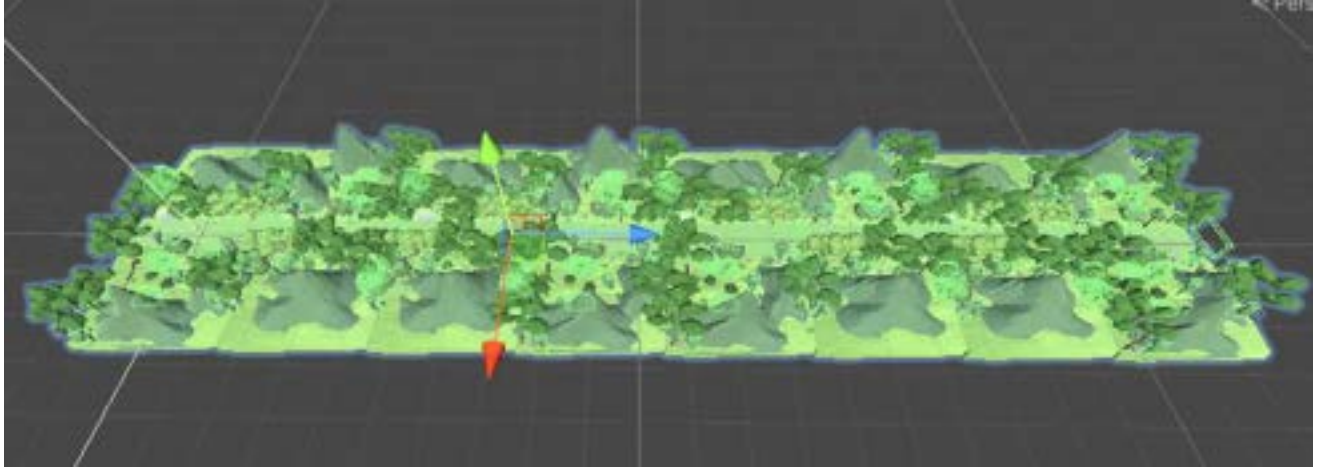
- **Sectionlar**
- **Engeller**
- **Elmaslar**

4.1 – Sectionların Oluşturulması

Daha önceden game design sürecinde zemin ve çevre prefablerini bir araya getirerek birçok “**section**” oluşturduk. Şimdi bu sectionları sahneye otomatik olarak ortaya çıkaracak scripti yazmamız gerekiyor. Otomatik üretilmesinden önce karakterin boş bir zemin üzerinde durmasını istemediğimiz için başlangıç olarak sahneye sekiz adet section’ı ekledik .



Sectionlar’ı bir arada tutmak için **parent** bir game object tanımladık ve onun altında bu sekiz adet hazırladığımız sectionı tutuyoruz. Tabi ki burda sayı kısıtlı sekizinci sectionın dışına çıktığında karakterin yine boşlukta hareket etmesini istemiyorsak bunu otomatize etmemiz gerekiyor.

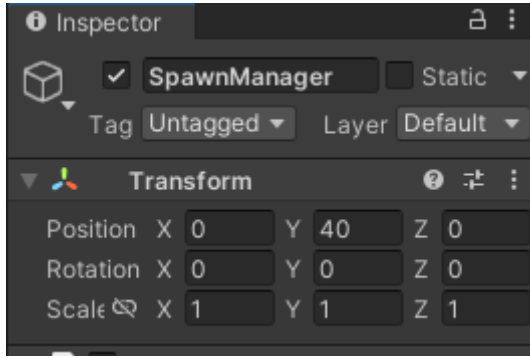


Artık boş bir zemin üzerinde durmuyoruz ve “**SpawnTrigger**” etiketiyle taglenmiş her bir zemin sonundaki collidera çarparak bir öncekini silip , yeni bir konumda bir başka sectionı yaratarak ilerleyeceğiz. Kamera açısından oyuncunun göreceği görüntü ise şöyle olacak ;



Artık karakterin hareket edebileceği ortamın temelini atmış olduk. Bunun üzerine daha birçok dizayn ve düzeltmeler yapıldı ama oyunun konseptinin oturduğu ve ortaya çıktığı ilk noktamız bu aşama. Burada **statik** olarak sahneye bıraktığımız sectionlar üzerinde hareket ediyoruz bunu **dinamik** bir hale nasıl getireceğiz sorusunun cevabını bulmak için script yazmamız gerekiyor.

Öncelikle **SpawnManager** adında bir game object yaratıyoruz. Bu game objesi üzerinde daha önce bahsettiğim tüm otomatik oluşturma işlemlerini bir obje üzerinde yöneteceğiz. Bu obje üzerinde üç farklı script yazıp component olarak ekleyeceğiz. Bütün scriptlerin inspector panelinden yönetilecek parametrelerini bu obje üzerinden kontrol edeceğiz.



Oluşturduğumuz SpawnManager objesinin inspector panelinde görüntüsünü görüyoruz. Geliştirdiğimiz scriptleri bu objenin üzerine component olarak ekleyeceğiz şimdi.

Öncelikle “ZeminSpawner.cs” scriptini oluşturuyoruz ve tüm sectionları artık otomatik bir şekilde ortaya çıkmasını sağlayacağız.

```

ZeminSpawner.cs X SpawnManager.cs
Assembly-CSharp
1 using System.Collections;
2 using System.Collections.Generic;
3 using System;
4 using UnityEngine;
5
6 public class ZeminSpawner : MonoBehaviour
7 {
8
9     public List<GameObject> zeminler; //Zemin prefablerini içerisinde tutacak GameObject türünde
10    private float offset = 30f;
11
12    // Start is called before the first frame update
13    void Start()
14    {
15
16        if(zeminler != null && zeminler.Count > 0)
17        {
18            zeminler = zeminler.OrderBy(z => z.transform.position.z).ToList(); //Listeye atılır
19        }
20
21
22
23
24
25
26    1 başvuru
27    public void MoveZemin()
28    {
29
30        GameObject moveZemin = zeminler[0]; //listenin ilk elemanını al
31        zeminler.Remove(moveZemin);
32        float yeniZ = zeminler[zeminler.Count - 1].transform.position.z + offset;
33        moveZemin.transform.position = new Vector3(0, 0, yeniZ);
34        zeminler.Add(moveZemin); //listenin başına tekrar eklemek zorunda kalmadan ekleyecektir.
35    }
36

```

Öncelikle **GameObject** türünde bir **List** tanımlıyoruz. Bu listede “**section**” prefablerimizi tutacağız. Bir offset değeri tanımlıyoruz ki bir sonra ki ortaya çıkacak section’ın aralığını belirliyoruz.

Start() fonksiyonu içerisinde önce bir kontrol yapıyoruz. Liste boş mu? Boş değilse devam eder. Z pozisyonunda tüm “zeminler” listesini sıralayacağız.

MoveZemin() fonksiyonu tanımlıyoruz ve öncelikle **zeminler[0]** diyerek listenin ilk elemanını bir Gameobject değişkeninde tutuyoruz çünkü “**Dairesel bir liste**” tasarlamayı planladık. Listenin son elemanı aynı zamanda bir sonra ki adımda listenin ilk elemanı olacaktır.

Tüm zeminler listesindeki section prefablerini Z ekseninde ortaya çıkaracağımız için bir sonra ki prefabin ortaya çıkacağı transform pozisyonuna **offset** bilgisinide ekliyoruz ve bu şekilde Z eksenindeki yeni değeri sürekli 30 birim artarak devam edecek ve sürekli sectionlar ortaya çıkacak. Ortaya çıkarken üst üste de çıkmamış olacak bu sayede. **Add()** fonksiyonu ile listedeki o sectionı tekrar listeye eklemiş oluyoruz. Bu sayede liste içinde bir döngü oluşturmuş oluyoruz.



Hazırladığımız bütün Sectionları bu listeye atıyoruz. Dairesel bir liste olduğu için kendi içinde çevirerek ortaya çıkartıyor ve sonsuz bir yapı sağlanmış oluyor.

Bu **MoveZemin()** fonksiyonunu çağırmak için daha önce hazırladığımız ve her zemin sonunda bulunan **collidera** giriş yapıldığında tetiklenerek yeni zemini oluşturacak ve bunu yapabilmek için **SpawnTrigger** diye adlandırdığımız o tagi kullanacağız.

```
SpawnManager.cs
Assembly-CSharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Unity Betiği (1 varlık başvurusu) | 1 başvuru
public class SpawnManager : MonoBehaviour
{
    ZeminSpawner zeminSpawner;

    // Start is called before the first frame update
    // Unity İletisi | 0 başvuru
    void Start()
    {
        zeminSpawner = GetComponent<ZeminSpawner>(); //Spawn yönetimi sınıfından nesneler çekilir.
    }

    // Update is called once per frame
    // Unity İletisi | 0 başvuru
    void Update()
    {
    }

    // 1 başvuru
    public void SpawnTriggerGiris()
    {
        zeminSpawner.MoveZemin();
    }
}
```

- **SpawnManager.cs** scripti çok basit bir script ve bunu da SpawnManager objesine component olarak ekliyoruz.
- Daha sonra burada **SpawnTriggerGiris()** adında bir başka fonksiyon içerisinde **zeminSpawner.cs** içerisinde oluşturduğumuz **MoveZemin()** fonksiyonunu çağırıyoruz.
- SpawnTriggerGiris() fonksiyonun diğer tarafını da görelim neler oluyor peki?

PlayerMove.cs içerisine bir Event ekliyoruz.

```
Unity İletisi | 0 başvuru
private void OnTriggerEnter(Collider other) //SpawnManager-ZeminSpawner
{
    if(other.tag == "SpawnTrigger") //player - isTrigger Spawn yeni tile
    {
        spawnManager.SpawnTriggerGiris(); // Collidera girişte tetiklenecek func.
    }
}
```

Karakterimiz eğer bir Collidera girerse tetiklenecek. **Other** burda collider içerisine giren objeyi belirtiyor. Eğer bu objenin tagi **SpawnTrigger** ise şunu yap diyoruz. SpawnManager classından bir nesne yaratıp **SpawnTriggerGiris()** fonksiyonuna erişiyoruz. O fonksiyonda hatırlayacağınız üzere **MoveZemin()** fonksiyonunu çalıştırarak sürekli yeni **Sectionlar** üretmiş olacağız.

4.2 – Engellerin Oluşturulması

Sectionlar otomatik olarak üretiliyor ama karakterimiz boş bir zeminde sadece koşuyor. Ona bir takım zorluklar çıkarmamız gerekiyor ve kullanıcıya ulaşması gereken bir hedef vermemiz gerekiyor.

Yine aynı mantıkla ilerleyerek engellerin oluşması için bir script oluşturacağız ve bu scripti SpawnManager objesine component olarak ekleyeceğiz. Bu component üzerinde prefab olarak vereceğimiz kaynakları **random** bir sırayla üreterek sahne üzerinde oluşmasını sağlayacağız.

Daha sonra bu engellere bir çarpışma triggerı ekleyerek karakterin oyun sonu ekranını göreceği ve durum değerlendirmesinin yapıldığı özet bir ekran görebilmesi sağlanacaktır.

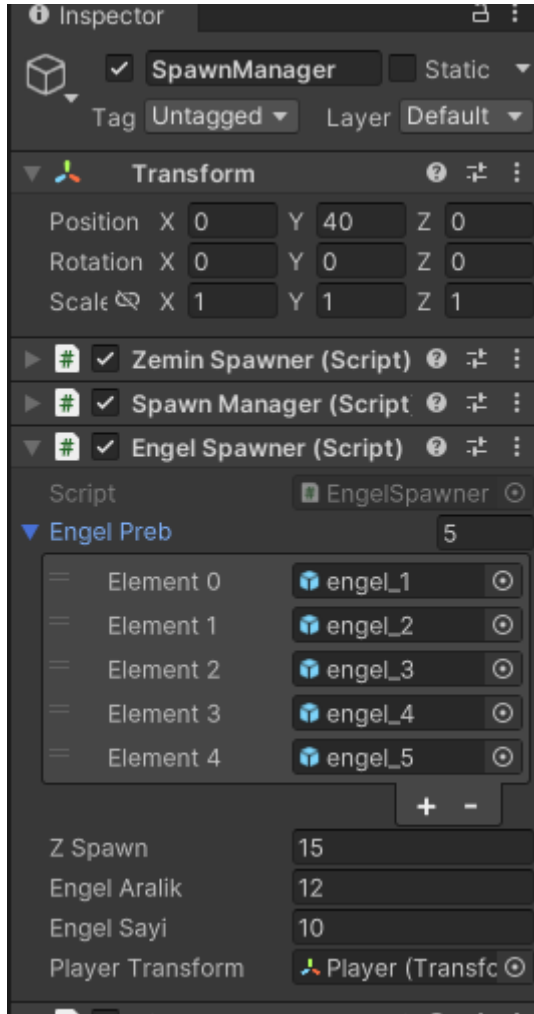
```

EngelSpawner.cs
Assembly-CSharp
EngelSpawner

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Unity Betiği (1 varlık başvurusu) 0 başvuru
6  public class EngelSpawner : MonoBehaviour
7  {
8      public GameObject[] engelPreb;
9      public float zSpawn = 15;
10     public float engelAralik = 12;
11     public int engelSayi = 5;
12
13     private List<GameObject> aktifEngel = new List<GameObject>();
14     public Transform playerTransform;
15
16
17     // Start is called before the first frame update
18     // Unity Betiği 0 başvuru
19     void Start()
20     {
21         for(int i = 0; i < engelSayi; i++)
22         {
23             if (i == 0)
24                 SpawnEngel(0);
25             else
26                 SpawnEngel(Random.Range(0, engelPreb.Length));
27         }
28     }
29
30     // Update is called once per frame
31     // Unity Betiği 0 başvuru
32     void Update()
33     {
34         if (playerTransform.position.z > zSpawn - (engelSayi * engelAralik)) // -35>
35         {
36             SpawnEngel(Random.Range(0, engelPreb.Length));
37             DeleteEngel();
38         }
39     }
40
41
42     // 3 başvuru
43     public void SpawnEngel(int engelIndex)
44     {
45         GameObject aktif = Instantiate(engelPreb[engelIndex], transform.forward * zSpawn, transform.rotation);
46         aktifEngel.Add(aktif);
47         zSpawn += engelAralik;
48
49     }
50
51     // 1 başvuru
52     private void DeleteEngel()
53     {
54         Destroy(aktifEngel[0], 2f); //gecikme
55         aktifEngel.RemoveAt(0);
56     }
57
58
59 }
60

```

- Öncelikle engel prefablerini tutacağımız game object türünde bir dizi tanımlıyoruz. Karakter ilk doğduğunda direkt olarak karşısına engel çıkmasını istemiyoruz bu sebeple $Z = 15f$ mesafeden ortaya çıkarmaya başla diyoruz.
- Bu engellerin sıklığıyla daha sonra oynayabiliriz oyun zorluğunu arttırmak için ama şu an **engelAralik = 12f** birim ötede bir engel ortaya çıkacak. Tek bir çevrimde 5 adet engel yaratsın sadece diyoruz. Bu 3 değişkende inspector panelinden ulaşabileceğimiz türden **public** olarak tanımladık ki daha sonra değiştirebileyim.
- **Start()** fonksiyonu içerisinde “**Random.Range()**” ile başlangıcımızı yapıyoruz ve 0 – dizi uzunluğu arasında bir engeli seçiyoruz **random** bir şekilde.
- Engelleri oluşturuyoruz ama onları yok etmezsek eğer karakterin arkasında kalır ve hiç silinmeyen nesneler olacaktır. Bu objelerin karakterin görüş açısından çıktıktan sonra silinmesini istiyoruz. **Update()** fonksiyonu içerisinde eğer karakterin Z pozisyon değeri $zSpawn - (engelSayi * engelAralik)$ büyükse **SpawnEngel()** fonksiyonundan rastgele bir engel daha oluştur ama aynı zamanda **DeleteEngel()** fonksiyonunu da çağır diyoruz.
- **SpawnEngel()** fonksiyonuna bakacak olursak ; integer bir parametre alıyor ve dizi elemanlarından rastgele seçilen elemanın indexini alıyoruz aslında.
- **Instantiate()** fonksiyonu burada seçilen objelerden **clone** yaratıyor. Objenin bir kopyasını yarattığı için Unity’de bu kopyalara müdahale diğer objelere müdahale etmekten biraz daha zor. O sebeple onları yok etmek içinde **Update()** fonksiyonunda karakterin pozisyonuna göre yok edilmesi için farklı bir kod yazıldı.
- **aktifEngel** diye tanımladığımız bir başka liste var ve bu liste üzerinde o an yaratılan engeli tutuyoruz çünkü onun yok edilmesi gerekecek. Bir yandan oluştururken bir yandan silmek için hazırlık yapıyoruz.
- **DeleteEngel()** fonksiyonunda ise **Destroy()** fonksiyonunu kullanıyoruz ve engel objesini yok ediyoruz. Bunu yaparken bu listeye alınan hep ilk eleman olacağı için 0.index veriliyor. **2f** ise bir gecikme süresidir. 2 saniye gecikmeyle yok et engeli gibi düşünebilirsiniz.
- Daha sonra **aktifEngel** listesini **RemoveAt()** ile boşaltıyoruz.



SpawnManager objesine component olarak ekliyoruz bu scriptimizi ve public olan değişkenlerimizi de bu inspector penceresi üzerinden erişebildiğimizi de buradan görebilirsiniz. Oyunun akışıyla alakalı beğenmediğimiz bir durum olursa buradan kolaylıkla müdahale edebilir hale getirdik.

Dizinin önce büyüklüğünü veriyorsunuz , önceden hazırlamış olduğumuz engel prefabları bu dizinin indexlerine atıyoruz.

Bu noktadan sonra çeşitliliği arttırmak ve yönetmek geliştirme sürecinde ufak bir adım haline geldi , ana yapıyı kurduğumuz için pek sorun olmayacak.

4.3 – Elmasların Oluşturulması

Engelleri oluşturduk şimdi de oyuncunun oyunda kendi ilerleyişini görebilmesi için bir ölçüye ihtiyacımız var. Bu da toplanabilir eşyalar, oyun süresi ya da katedilen mesafe oluyor. Burada oyuncunun toplayacağı nesneler olarak **elmas** kullanmaya karar verdik .

Bunun için bir **DiamondSpawner.cs** scripti oluşturuyoruz ve sürekli oluşturulmasını sağlayacağız. Temel mantık çok değişmiyor , neredeyse aynı diyebilirim.

```

DiamondSpawner.cs
Assembly-CSharp
DiamondSpawner

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class DiamondSpawner : MonoBehaviour
6  {
7      public GameObject[] diamondPreb;
8      public float zSpawn = 15;
9      public float diamondAralik = 5;
10     public int diamondSayi = 5;
11
12     private List<GameObject> aktifDiamond = new List<GameObject>();
13     public Transform playerTransform;
14
15     // Start is called before the first frame update
16     void Start()
17     {
18         for (int i = 0; i < diamondSayi; i++)
19         {
20             if (i == 0)
21                 SpawnDiamond(0);
22             else
23                 SpawnDiamond(Random.Range(0, diamondPreb.Length));
24         }
25     }
26
27     // Update is called once per frame
28     void Update()
29     {
30         if (playerTransform.position.z > zSpawn - (diamondSayi * diamondAralik))
31         {
32             SpawnDiamond(Random.Range(0, diamondPreb.Length));
33             DeleteDiamond();
34         }
35     }
36
37
38
39
40
3 başvuru

```



```

3 başlatıcı
public void SpawnDiamond(int diamIndex)
{
    float xPos = Random.Range(-3.4f, 3.5f);
    float yPos = 2.1f;

    GameObject aktif = Instantiate(diamondPreb[diamIndex], new Vector3(xPos,yPos,zSpawn), transform.rotation);
    aktifDiamond.Add(aktif);
    zSpawn += diamondAralik;
}

1 başlatıcı
private void DeleteDiamond()
{
    Destroy(aktifDiamond[0], 5f); //gecikme
    aktifDiamond.RemoveAt(0);
}

```

Daha önce engelleri oluştururken yine **Instantiate()** kullanıyorduk lakin burada bir değişiklik yapmamız gerekiyordu. Engeller bir zemin üzerinde hazırladığımız prefablerdi ve zemini tamamen kapladıkları için Z pozisyonunu sadece değiştirmeniz yeterli oluyordu **random** bir şekilde üretirken.

Elmaslar ise tek bir nesne olarak oluşmaları ve X ekseninin de rastgele bir noktasında oluşması gerekiyor. Bu X ekseninde oluşacak **Random pozisyon** bilgisi ise **xPos** değişkeni ile alınıyor. **Random.Range(-3.4f,3.5f)** aralığında bir random X koordinatı oluşturulmasını istiyoruz. **Random.Range()** fonksiyonu üst sınırları almayacağı için zemin sınırları içerisinde elmas prefablerinin rastgele oluştuğunu göreceğiz.

Instantiate() fonksiyonu içerisinde bir **Vector3** nesnesi yaratılıyor ve oluşacak bu **clone** örneklerin belli bir pozisyonda oluşmasını istiyorum. **xPos,yPos ve zSpawn** noktalarını vererek istediğimiz koordinatlarda **random** bir şekilde ortaya çıkacak bu elmaslar ve **5f** gecikmeyle **Destroy()** fonksiyonu yardımıyla yok edilecektir.

4.3.1 – Elmas Prefab Rotasyon Bilgisi

Oyunumuzu sadece statik nesnelerle üretmek sıkıcı bir yapıyı beraberinde getirecektir. Bunu biraz olsun kırmak için **Y=2.1f** ekseninde duran elmas prefabler sabit bir şekilde biraz hava da duruyorlar. Elmaslara basit bir script yazarak dinamik bir hale getirelim. Daha sonra ayrıca birçok efekt işlemleri yapacağız.

Objelerin transform bilgilerindeki parametrelerinden biri de **rotation** yani dönüş açısıdır. Bu dönüş açısını **Update()** ile her framede elimizde bulunan 6 adet elmas prefabına component olarak ekleyeceğimiz script ile rotasyon bilgisini sürekli değiştirebilir ve elmasların sürekli döndüğünü görebiliriz.

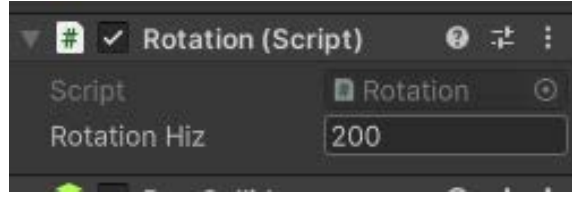
```

Rotation.cs
Assembly-CSharp
Rotation

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  Unity Betiği (6 varlık başvurusu) | 0 başvuru
6  public class Rotation : MonoBehaviour
7  {
8      [SerializeField] float rotationHiz = 200f;
9
10     // Start is called before the first frame update
11     void Start()
12     {
13     }
14
15     // Update is called once per frame
16     void Update()
17     {
18         this.transform.Rotate(Vector3.up * rotationHiz * Time.deltaTime);
19     }
20
21 }
22
23

```

[**SerializeField**] float rotationHiz değişkeni tanımlıyoruz ve bu sayede inspector panelinden değiştirebileceğimiz bu değeri. 200f'lik bir dönüş hızı olması için bir değişken tanımladık.



Daha sonra bunu **Update()** fonksiyonu içerisinde **transform** bilgisinin **rotate()** fonksiyonunu çağırarak zamana bağlı olarak dönüş açısını değiştiriyoruz. Bu da bize sürekli dönen bir elmas nesnesi elde etmemizi sağlıyor.

5. Engel ile Çarpışma ve Oyun Sonu Ekranı

5.1 – Engelle etkileşim

Bölümlerimizi ve engellerimizi ekledikten sonra sıra koşan karakterimizin bu engellerle etkileşimine geldi. Bu etkileşimin algılanması Player objemize eklediğimiz **Rigidbody** fizik komponenti sayesinde yapacağız. Bu obje eğer engele temas ettiği zamanda “**OnCollisionEnter**” çağrılır. Ve parametresi ile temas ettiği objenin “**tag**” ını alırız. Bu aldığımız tag daha önce engellerin oluşturulması sırasında eklediğimiz “**Engel**” tagı ise etkileşim gerçekleşmiş olur.

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Engel")//Engele temas ederse
    {

```

Ardından can sistemi kontrolleri başlar ancak buna sonraki bölümlerde değineceğiz. Kontroller tamamlandıktan sonra karakterimizin öldüğü anlaşılır. Ve **Death()** adı verdiğimiz ölüm işlemlerini gerçekleştiren fonksiyon çağrılır.

```
private void Death()
{
    isAlive = false;
    GameManager.SaveScore(GameManager.mesafe, GameManager.playerDiamonds);
    StartCoroutine(dieDelay());
    animator.SetTrigger("crash");//ölme animasyonu tetikle
    deathPanel.SetActive(true);
    GameManager.mesafe = 0;
    GameManager.playerDiamonds = 0;
    PlayerPrefs.SetInt("healthCount",3);

    AudioManager.DeathPlaySFX(audioManager.deathpanel); //Deathpanel , DeathPlaySFX fonksiyonuna parametre olarak geçir
}

IEnumerator dieDelay()
{
    yield return new WaitForSeconds(2f);
    playerObject.SetActive(false);
    //Animasyon bitince karakteri sil
}
```

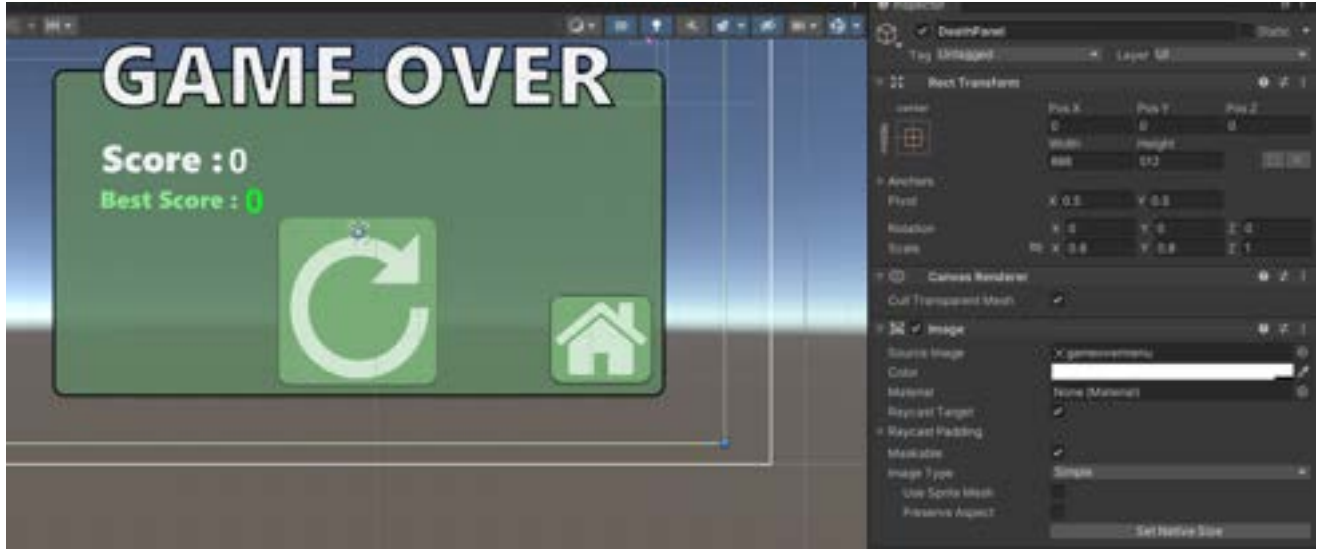
Burada karakterimizin canlılığını takip ettiğimiz static bir bool değişken olan `isAlive`'yi false olarak atadık. Ardından skor kaydetme işlemlerini gerçekleştirdik. Şimdi kullanıcıya oyunun bittiğini ifade eden bir ekran göstermeliyiz.

5.2 – Oyun Sonu Ekranı

Kullanıcıya topladığı elmasları, ve koştuğu mesafeyi göstermemiz için oyun tamamına uygun bir menü tasarlamamız gerekiyordu. Oyundaki renk paletimize ve modern UI ilkelerine mümkün olduğunca uyarak Adobe Photoshop uygulaması üzerinden bir menü oluşturduk.



Ardından Unity üzerinde 2D etkileşimli görsel öğeler eklememize izin veren **Canvas** objesinin `child`'i olacak şekilde oyunumuza ekledik. Ardından tasarımda boş bıraktığımız skor metin yerleri doldurduk. Ve kullanıcının yeniden oynaması veya Ana Menüye dönmesi için birer button ekledik. Ve Editor üzerinden **SetActive** özelliğini devredışı bıraktık. Bu sırada menü oyunu açar açmaz çıkmayacak. Ölüm fonksiyonu çağrıldığında `SetActive true` yapılacağı için gözükecektir.



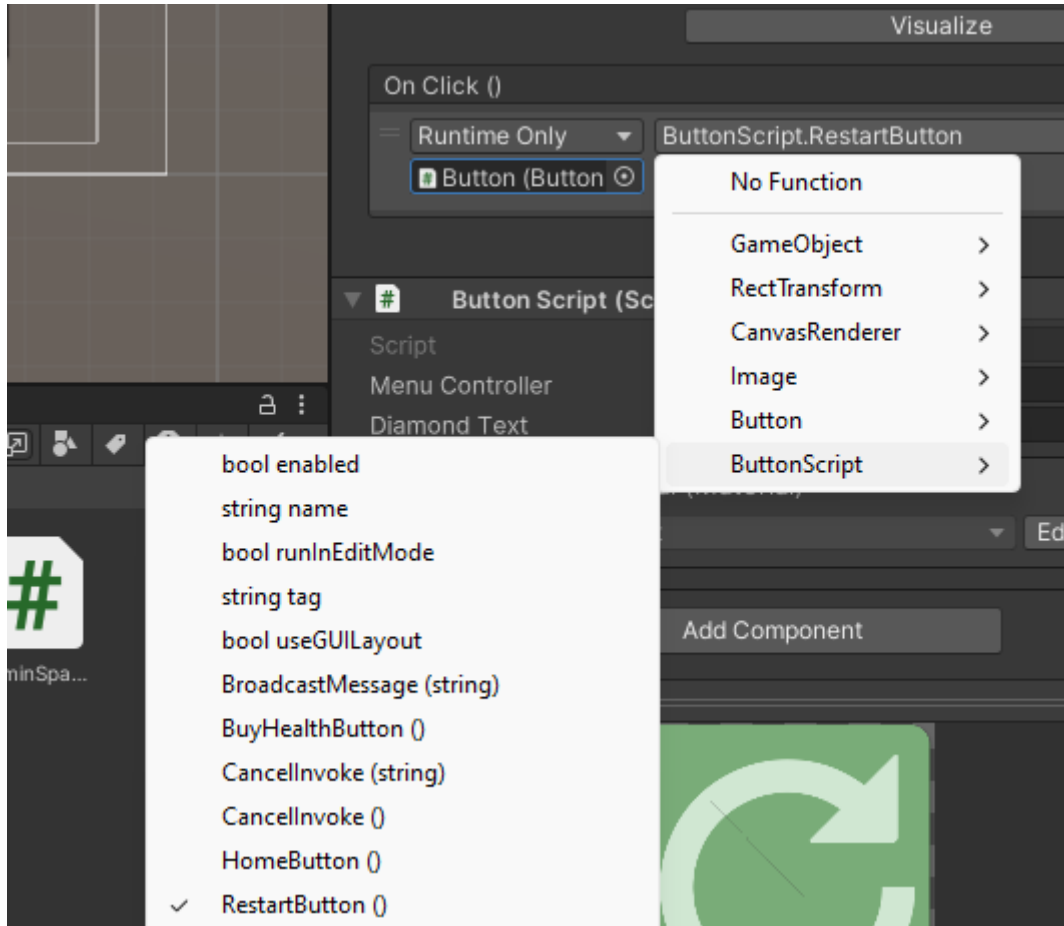
6. Butonlar ve Sahne Geçişleri

Canvas'ımıza butonlarımızı eklemiştik. Ardından butonlarımız için yeni bir script oluşturduk. Ve sahne geçişlerimiz için fonksiyonlar yazdık. Bu fonksiyonlar buton'un **onClick** özelliği sayesinde erişilebiliyor. Ancak fonksiyonların erişim belirteci **public** olmak zorunda.

```
0 başvuru
public void RestartButton()
{
    PlayerMove.isAlive = true;
    GameManager.healthcount = 3;
    PlayerPrefs.SetInt("healthCount", 3);
    SceneManager.LoadScene(SceneManager.GetActiveScene().name); //Sahneyi yeniden baslat
}

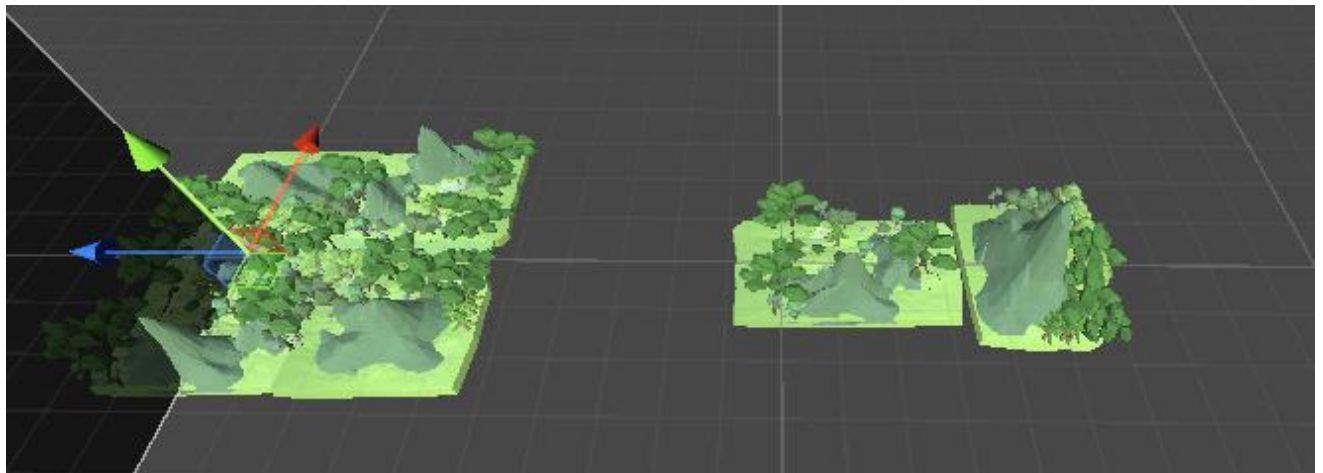
0 başvuru
public void HomeButton()
{
    SceneManager.LoadScene("Main Menu");
}
```

Scriptimizi ekledikten sonra Unity tarafında **onClick** atamalarını butonlarla fonksiyonları eşleştirdik. Sahne geçişlerimizi Unity'de sahneyi kontrol eden **SceneManager** üzerinden yaptık. Yeniden oyna düğmesi bulunduğu aktif sahnenin ismini alıp yeniden yüklerken, Home düğmemiz Main Menu'yü direkt olarak çağırır. Burada aktif sahnenin ismini almamızın sebebi ileride güncellemelerle getirebileceğimiz başka temalardaki başka sahneler için de aynı kodu kullanabilmektir.



6.1 – Sahne 2 : Ana Menü

Level01 sahnesi bizim ana sahnemiz ve oyunun geçtiği sahnenin kendisidir. Oyunu ilk açtığımızda karşımıza direkt olarak oyunun gelmesini elbette istemeyiz. O sebeple bir ana menü olması gerekiyordu. Ana menüyü sade bir tasarımla ayarladık ve oyunla hemen iç içe gibi bir görüntü elde etmek istedik.



Ana menü başka bir sahne olduğu için bu sahnenin tasarımını ayrıca yapmak gerekti. Objeleri değiştirip görmek istediğimiz görüntüyü yakalayana kadar tasarıma devam ettik. Üç farklı buton bulunuyor Ana Menü içerisinde. Ayrıca yüksek skoru tutması için PlayerPrefs çıktısı ve Can satın alma butonu bulunuyor.



Ayarlar butonu şu an için herhangi bir işlevi bulunmuyor. Bu projenin kapsamı dışında gereksiz detay olacağı için bu eklentiye eklemek için zaman harcamadık. Daha sonra bu projeyi geliştirmeyi düşünüyoruz. O sebeple ileriye dönük planlarımız ve geliştirmelerimiz için bir hazırlık olarak düşünebiliriz.

7. Level01 UI Sistemi

Sahnemiz için kullanıcının ilerlemesini anlık olarak görebileceği ve sahip olduğu can sayısını takip edebileceği görsel bir bölüme ihtiyaç vardı. Bunun için yine Canvas'ımıza text ve sprite eklemekle başladık. Bunlardan soldaki Can sayısını, sağdakiler ise sırasıyla; toplanan elmas sayısı, ilerlenen mesafe gösteriyordu.



Bunların çalışabilmesi için Unity için özel bir obje olan **GameManager** objesini ve Script'ini ekledik. GameManager oyun varolduğu sürece çalışacak ve komutlarını yerine getirecektir. Public olarak etiketlediğimiz text objelerimize editör yoluyla eriştikten sonra anlık olarak çağrılan Update() fonksiyonuna bu kodu yazdık.

```

public Text uiMesafe;
public Text uiDiamond;
public Text uiHealth;
public Text scoreText;
public Text BestScoreText;

```

📱 Unity İletisi | 0 başvuru

```

void Update()
{
    mesafe = Mathf.RoundToInt(player.transform.position.z);
    uiMesafe.text = mesafe.ToString() + " m";
}

```

1 başvuru

```

public void ToplananDiamond()
{
    playerDiamonds++;
    uiDiamond.text = playerDiamonds.ToString() + " x";
    //Debug.Log("Coin : " + playerDiamonds);
}

```

Bu kod aslında **float** bir değer olan transform'un "z" kordinatında anlık konumunu ifade eden position'unu int değere yuvarlıyor. Bu bizim mesafe diye ifade ettiğimiz değişken ardından kullanıcıya gösterilmek üzere **stringe** çevrilip, eklediğimiz Text elemanlara atanıyor. Ve sürekli olarak çağrıldığı için anlık gösteriliyor. Tabii sadece toplandığı zaman güncellenmesi gereken elmas sayısı göstergesi hariç.

8. PlayerPrefs ile Skor Tutmak

Kullanıcıya oyunu sevdirebilmek ve hırs hissi uyandırabilmek için oyundaki ilerlemeleri kaydetmemiz çok önemli. Bu sayede kullanıcı bir önceki rekorunu geçmeye heves edebilir. Ve oyun boyu topladığı elmasları her oyunu kapattığında kaybetmez. Bunun için Unity'nin içinde PlayerPrefs dediğimiz yöntemi kullanacağız. Bu yöntem sayesinde belirttiğimiz bilgiler kişinin bilgisayarında saklanacak. Bunun için oyunumuzu kontrol eden GameManager scriptimiz'in Start fonksiyonu yani ilk açılışında gerçekleşen işlemler kısmına bu kodu ekledik.


```
1 // Unity ile bir 3D oyunu  
2 void Start()  
3 {  
4     player = GameObject.Find("Player");  
5     bestScore = PlayerPrefs.GetInt("bestScore", 0); //PlayerPrefs yöntemiyle bestscoreyi kayıttan al yoksa sıfır al  
6     bestElmascore = PlayerPrefs.GetInt("bestElmas", 0);  
7     totalElmas = PlayerPrefs.GetInt("totalElmas", 0);  
8     healthcount = PlayerPrefs.GetInt("healthCount", 3); //Can sayısını al eğer yoksa 3 can ile başlat  
9     uiHealth.text = PlayerPrefs.GetInt("healthCount", 3).ToString() + " x";  
10 }
```

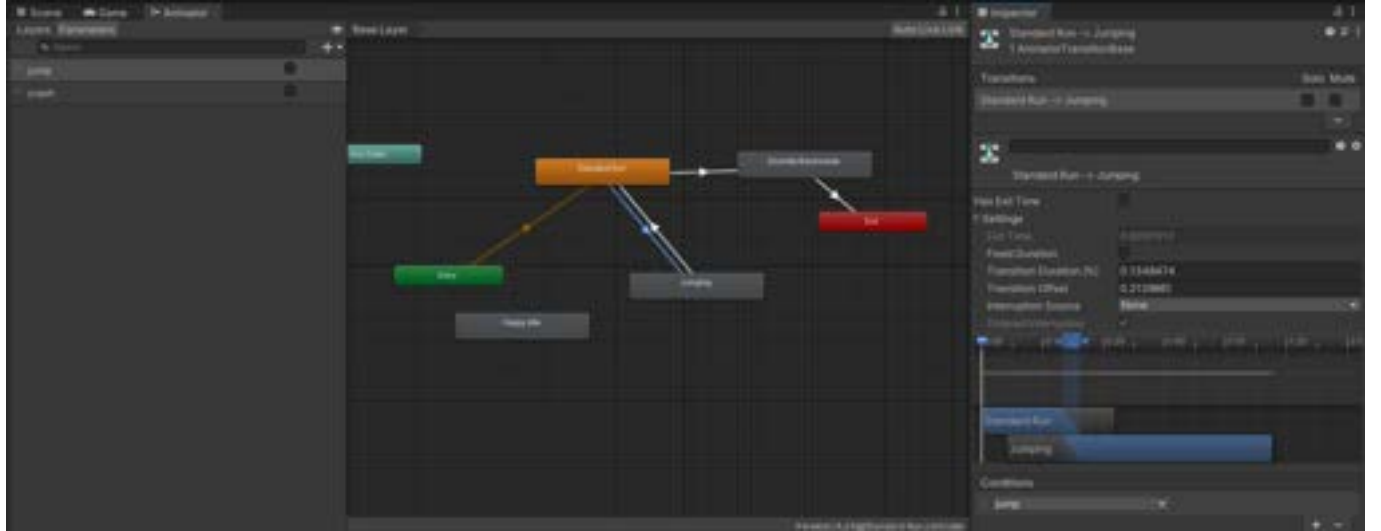
PlayerPrefs.GetInt ile daha önceden **SetInt** yaptığımız değerler varsa hafızadan oyunumuza yüklüyoruz, eğer yoksa ikinci parametresindeki değeri varsayılan olarak alıyoruz. Örneğin burada **"healthCount"** can sayısını belirten değer hafızadan yüklenecek ancak hafızada daha önceden bulunmuyorsa 3 canlı olarak karakterimiz başlatılacaktır.

```
1 //bunu  
2 public void SaveScore(int score, int elmasScore)  
3 {  
4     if (score > bestScore || elmasScore > bestElmascore) //eğer kullanıcı rekorunu geçtiyse yeni skor bestscore olur  
5     {  
6         bestScore = score;  
7         PlayerPrefs.SetInt("bestScore", bestScore); //playerprefs set etme işlemi  
8  
9         bestElmascore = elmasScore;  
10        PlayerPrefs.SetInt("bestElmas", bestElmascore);  
11    }  
12    BestScoreText.text = bestScore.ToString() + " = " + bestElmascore.ToString() + " E";  
13    scoreText.text = mesafe.ToString() + " = " + playerDiamonds.ToString() + " E"; //Menüye göster  
14    totalElmas += elmasScore;  
15    PlayerPrefs.SetInt("totalElmas", totalElmas);  
16 }
```

Ve **SaveScore** adında oluşturduğumuz bu iki parametrelili fonksiyonda ise **SetInt** işlemleri gerçekleştirilmekte. Burada önce anlık elde edilen mesafenin bir önceki rekordan daha fazla olup olmadığı bir if bloğunda test ediliyor eğer rekor kırıldıysa yeni **bestScore** olarak atanır. Ve **PlayerPrefs.SetInt** yöntemiyle hafızaya kaydedilir.

9 . Animator ve Animasyon Bağlantıları

Oyunumuzun daha güzel görünebilmesi için animasyonlarının olması ve bunların uyumlu bir şekilde çalışabilmesi oldukça önemli. Bu yüzden animasyon bağlantılarını iyi yapmalıyız. Unity’de animasyon işlemlerini kontrol etmekten **Animator komponenti** sorumludur. Ve bu komponent şemaya benzer bir yöntemle ayarlamaları yapılır.



Ortada gördüğümüz şema animasyon geçişlerini ayarladığımız şemayı ifade ediyor. Burada State dediğimiz durumlar önceden edindiğimiz animasyonları gösteriyor. Bizim oyunumuzda karakter koşarak başladığı için direkt olarak koşma animasyonu ile giriş yapılıyor. Ve bir tetik gelene kadar aynı durumda kalmaya devam ediyor. Solda gördüğünüz **“Parameters”** menüsüne eklediğimiz trigger parametreleri ile ok yönünde geçişleri tetikliyoruz.

Ardından stateleri oklar ile birbirine bağlıyoruz. Ve çağracağımız tetikleri sağ menüden **Conditions** sekmesinden ekliyoruz. Ardından üstteki zaman çizelgesini animasyon süremize göre kalibre ediyoruz. Animasyonlarımızdaki geçişin zamanında ve akıcı olarak yapılması bu menünün güzel yapılandırılmasıyla mümkün oluyor.

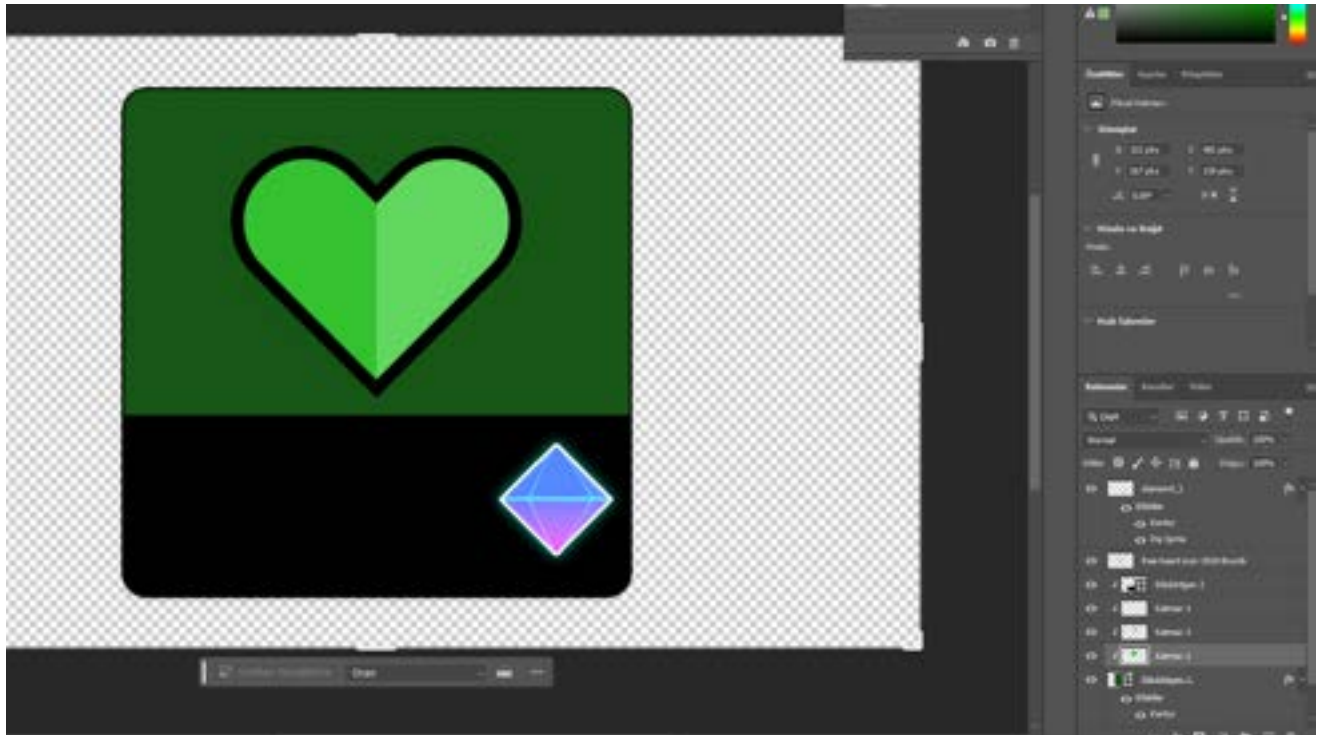
```
animator.SetTrigger("jump");//Jump Animasyonunu Tetikle
```

Örneğin karakter scriptimizdeki bu komut ile “jump” parametresini tetiklemiş oluyoruz bu sayede Animator aşağı ok yönünde geçiş yapıyor ve “jumping” state’ine geçiyor.

Ardından jump animasyonu gösteriliyor tamamlandıktan sonra başka parametre olmadığı için başlangıçtaki state’e geri dönüyor. Aynı şekilde **“StumbleBackwards”** yani nesneye çarpınca geriye düşme animasyonumuz da tetikleme yöntemiyle çalışıyor. Ancak bu oyunun bitmeden önceki son animasyon olduğu için direkt olarak **EXIT state** yani çıkış durumuna geçiyor. Ve animasyon sonlanıyor.

10 . Can Sistemi

Oyunumuzun türü gereği kullanıcıya yalnızca tek bir can hakkı vermek adil olmazdı. Ayrıca toplanan elmasların da boşuna gitmemesi bir amaç doğrultusunda kullanılması gerekiyordu. Bu yüzden oyuna bir can sistemi eklemeye karar verdik. Ve can için bir kalp tasarımı yaptıktan sonra Can alımı için button tasarımı yapmaya başladık.



Can alma buttonumuzu tasarladıktan sonra Ana Menüme button olarak ekledik. Ardından kod kısmına geçtik. Birkaç bölüm öncesinde de gösterdiğimiz gibi PlayerPrefs üzerinden kullanıcıya varsayılan önce 3 veya satın alım yapıldıysa hafızdaki can miktarına ulaşarak başladık.

```
healthcount = PlayerPrefs.GetInt("healthCount", 3); //Can sayisini al eger yoksa 3 can ile baslat
```

Ayrıca eğer kullanıcı daha öncesinde canlarını kullanmış olma ihtimaline karşı tabii ki bir kontrol işlemi de yapıyoruz.

```
if (healthcount < 3)
{
    healthcount = 3;
}
```

Ardından bir can azaltma fonksiyonu ekledik. Bu fonksiyon karakterimiz engellerle temas ederse çağrılacak ve direkt olarak ölme işlemleri yerine canı azaltılacak.

```
1 başvuru
public void HealthDecrease()
{
    GameManager.healthcount--; // Can'ı bir azalt
    uiHealth.text = GameManager.healthcount.ToString() + " x";
}
```

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Engel")//Engele temas ederse
    {
        if(GameManager.healthcount<= 0)//can sıfır değilse olmayacak
        {
            Death();//olmeyi çağır
        }
        else
        {
            gameManager.HealthDecrease(); //Can Azalt
        }
    }
}
```

Bir önceki bölümde anlattığımız engel etkileşiminde karakterimizin o andaki can sayısı **GameManager** üzerinden erişiliyor ve eğer sıfıra düşüyse ölme işlemleri başlıyor ancak hala geriye kalan canlar varsa yazdığımız can azaltma fonksiyonu devreye giriyor ve canı azalıyor.

Ardından sıra elmasları kullandığımız can satın aldığımız bölüme geldi. Bunun için ana menüye eklediğimiz Can Satın alma butonunun **onClick** özelliğine fonksiyon ekliyoruz.

```
0 başvuru
public void BuyHealthButton()
{
    AudioManager_m.PlaySFX(audioManager_m.buyHealth); // Can alırken farklı SFX
    GameManager.BuyHealth();
    diamondText.text = PlayerPrefs.GetInt("totalElmas", 0).ToString() + "x";
}
```

Sonrasında ise GameManager'a can satın alma kodumuzu yazıyoruz.

```

1 buyuru
public static void BuyHealth() {
    totalElmas=PlayerPrefs.GetInt("totalElmas", 0);
    if (totalElmas >= HealthPrice)
    {
        if (healthcount < 3)
        {
            healthcount = 4; //Onlem anaclı
        }
        else
        {
            healthcount++; //Can arttır
            Debug.Log("Can alındı");
        }

        totalElmas -= HealthPrice; //Sağlık fiyatı kadar elmas azalt
        PlayerPrefs.SetInt("totalElmas", totalElmas); //Bakiyeyi kaydet
        PlayerPrefs.SetInt("healthCount", healthcount); //Alınan canı kaydet
    }
}

```

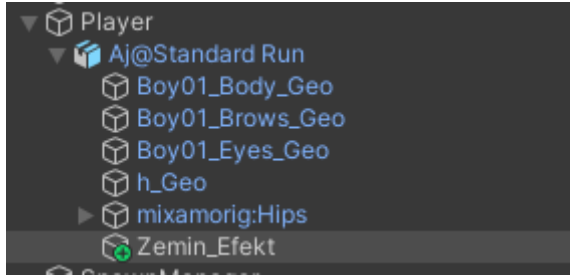
Burada elimizdeki totalElmas yani kullanıcının şuana kadar topladığı toplam elmas sayısı hafızadan elde edildikten sonra can satın alma işleminin fiyatı ile karşılaştırılıyor. Eğer kişinin can almak için yeterli elması var ise can sayısı artırılıyor. Ardından elmas bakiyesi ise düşürülüyor. Sonrada **SetInt** işlemleri ile yeni can sayısı ve yeni bakiye kayıt altına alınıyor. Bu sayede kullanıcı rekorunu geçmek için daha fazla hata yapma hakkına sahip oluyor. Ve elmas toplama işi önem kazanıyor.

11. Karakter ve Objelerin Efekt Ayarları

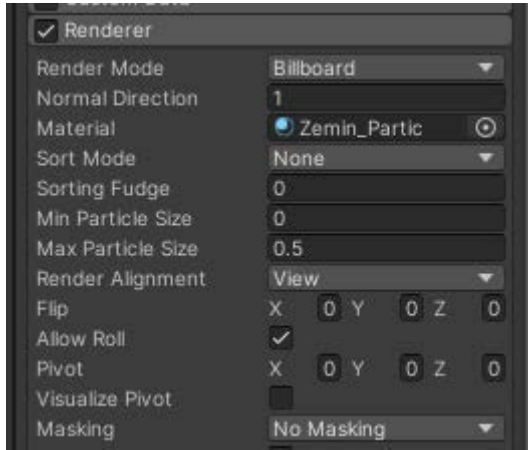
Daha önceden statik olan elmas prefablerimiz için rotasyon bilgisini değiştirerek dönmesini sağlamıştık ama bunu yeterli görmedik. Burdan yola çıkarak karakterimiz için de bir efekt eklemeye karar verdik. Elmaslar ve karakterimiz için efekt ayarlamaları yapacağız.

11.1 – Karakter Efekt Ayarları

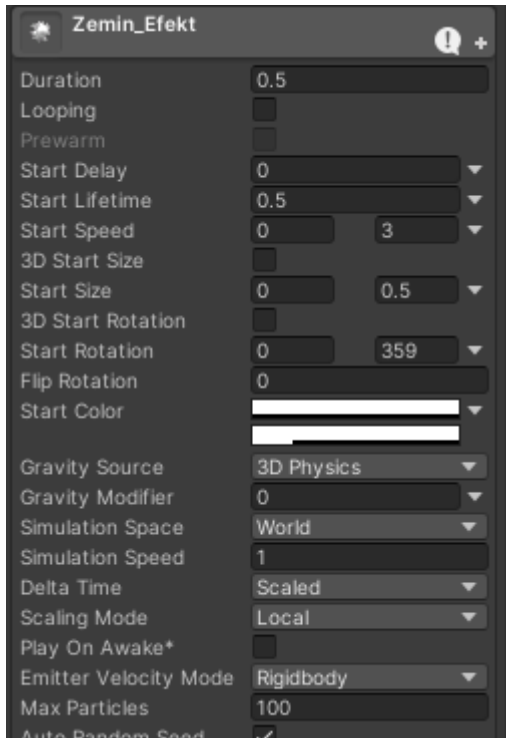
Karakterimiz için eklenecek efekt ; bir ormanın içerisinde sürekli koşmakta olan bir karakter için zeminde oluşacak efekt ayarlarını ekledik. Öncelikle “**player**” nesnesine sağ tıklayarak **Effects > Particle System** ekliyoruz. Bundan önce **Material** klasörümüzde koşma sırasında zeminde çıkacak efekt için daha önce png olarak indirdiğimiz duman.png resmini material üzerine attık ve hazır bir hale getirildi. Particle System üzerinde bir takım ayarlamalar yapmamız gerekiyor.



ParticleSystem çok detaylı efekt ayarlarının yapılabileceği içerisinde birçok alt başlığın bulunduğu bir özellik. O yüzden birçok detaylı ayarlamayı burada gerçekleştirdik.



ParticleSystem altında bulunan sekmelerden biri **Renderer** altında daha önce Material altında oluşturduğumuz ve üzerine png resim attığımız materialı çektik burada.



ParticleSystem'da en genel sekmelerden biri ve ilk kısım burası. Daha detaylı ayarlamalar yapmak için çeşitli sekmeler bulunuyor. Karşımıza çıkan bu ilk sekmeden bahsetmemiz gerekirse efektin Loop olup olmayacağını , ne kadar gecikmeyle başlatılacağını ve efekt partiküllerinin oluşuktan sonra aktif olarak kalma süresi gibi birçok detay ayarlamalar yapıldı.

StartColor kısmında ise iki renk belirledik ve bu iki renk arasında seçeceğiz. Bu renklerin ikisi de “beyaz”. Sadece transparant ayarı üzerinde oynamalar yaptık.

Bunun sebebi ise karakterin zemin üzerinde hareket etmesi sonucu zeminden çıkmasını planladığımız ve koşmanın şiddetiyle **toz efekti** diyebiliriz.

```
//----->> Efekt

public ParticleSystem diamondToplamaParticle; //Preb yapılan efekti toplanan diamond ile aktifleştirme
public ParticleSystem zeminParticle; // Karakterin zemin üzerinde koşarken çıkardığı efekt

void Update()
{
    if (isAlive)
    {
        transform.Translate(Vector3.forward *
        zeminParticle.Play(); //Zeminparticle
    }
}
```

PlayerMove.cs scripti içerisine bir takım eklemeler yapmamız gerekiyor. **Public** olarak tanımlanan bu particleSystem türündeki **zeminParticle** değişkenine inspectordan ulaştık ve Player objesi üzerinden daha önce oluşturduğumuz **Zemin_Efekt**'i göstermemiz gerekti.

Burada **diamondToplamaParticle** değişkenini de görüyoruz bu da bir sonra ki efektimiz olacak olan elmasların efektini işaret ediyor. **Update()** fonksiyonunda **Play()** yaparak efekt üzerinde yaptığımız birçok ayarlara göre her bir frame de yani karakterimiz her hareket ettiğinde ortaya çıkacak.

11.2 – Elmas Efekt Ayarları ve Elmasların Toplanması

```
if(other.tag == "Diamond") //player - isTrigger diamond
{
    gameManager.ToplananDiamond(); //playerdiamond++
    Instantiate(diamondToplamaParticle, other.transform.position + new Vector3(0, 0.497f, 0), other.transform.rotation); //
    Destroy(other.gameObject);
}
```

Aynı şekilde elmas içinde png resim üzerinden kendi hazırladığımız bir parıltı efekti bulunuyor. Bunu da çağırıyoruz **PlayerMove.cs** içerisinde ve **Instantiate** ile bir çok efekt klonu üretiliyor. Peki bu klonlar nasıl yok ediliyor? Normal de **Destroy()** ile yok ediyorduk ama inspector panelinde efektler için **StopAction** parametresini aktif ettiğinizde efekt yürütüldüğü anda sadece bir kere çalışır ve yok olur.

Bu arada **OnTriggerEnter** eventi içerisinde daha önceden hazırladığımız fonksiyonu çağırıyoruz ki **Diamond** tagi olan elmaslarımız ile etkileşime girdiğimizde **ToplananDiamond()** fonksiyonu çağırılır ve elmas sayacını artırır. Bunun sonucunda UI text üzerinde sonucu görüyoruz. Bu sırada yaratılan tüm elmas objelerini bu tetiklenmeyle beraber oluşturup aynı zamanda yok ediyoruz.

12. Müzik ve Ses Efektleri

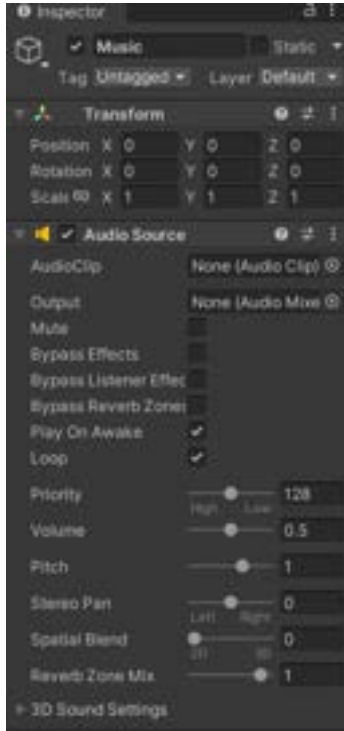
Müzik ve ses efektlerini düşünürken iki farklı script ve obje üzerinden yönetmek istedik. İki farklı sahnemiz bulunuyor ; **Ana Menü ve Level01**. Bu iki sahnenin müzik ve ses efektlerini farklı seçmek istedik. Buna göre de farklı kaynakları olsun ve yönetilsin , bu sayede işi de bölmüş oluruz diye düşündük. Herhangi bir güncelleme yapmak istediğimizde de bir sıkıntı yaşamadan gerçekleştirebiliyoruz.

12.1 – Level01 : Müzik ve Ses Efektleri

Daha önce Asset storedan beğenip projeye dahil ettiğimiz paketler içinden ; Level01 için arka planda çalacak bir müzik , oyun sonu paneli geldiğinde aktif olacak bir ses efekti ve her bir elmasın toplanmasıyla birlikte çalışacak ses efekti olarak üç farklı ses efektini seçtik bu sahne için.



Öncelikle boş bir gameobject nesnesi yaratıyoruz “**AudioManager**” adında ve tüm ses efektlerini bu obje üzerinden yöneteceğiz. **Music,SFX, DeathSFX** gibi child objeler tanımladık hemen altına. Bunun sebebi üç farklı ses kaynağı oluşturmak istememizden kaynaklanıyor. Kamera takip sisteminden bahsederken default olarak kamera objesinde bir **Audio Listener** olduğundan bahsetmiştik. Bu listenerın dinleyeceği kaynaklara ihtiyacımız var. Bu child objelerde 3 farklı **AudioSource** componentlerini tutuyorlar.



Sadece **Music** child objesine eklenmiş **AudioSource** componentini görüyoruz. Ama diğer iki child obje içerisinde de aynı şekilde bu component bulunuyor.

Burada tek farkı ise **Play on Awake ve Loop** aktive ettik. Arka planda sürekli çalacak bir arka plan müziği için bu ayarları yapmak gerekiyor.

Volume ise %50 oranında makul gördüğümüz için değiştirdik.

```

AudioManager.cs
Assembly-CSharp
AudioManager
musicSource

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class AudioManager : MonoBehaviour
6 {
7     [Header("----- Audio Source -----")]
8     [SerializeField] AudioSource musicSource;
9     [SerializeField] AudioSource SFXSource;
10    [SerializeField] AudioSource DeathSFXSource;
11
12    [Header("----- Audio Clip -----")]
13
14    public AudioClip background;
15    public AudioClip deathpanel;
16    public AudioClip diamond;
17
18
19
20
21
22    private void Start()
23    {
24        musicSource.clip = background;
25        musicSource.Play(); // inspectorda loop'a alınmış background music
26    }
27
28
29    public void PlaySFX(AudioClip clip)
30    {
31        SFXSource.PlayOneShot(clip); // tek seferde çalıştırılacak efekt
32    }
33
34
35    public void DeathPlaySFX(AudioClip clip) // Sadece Deathpanel'e özel fonksiyon background music disable etmesi için
36    {
37        DeathSFXSource.PlayOneShot(clip); // tek seferde çalıştırılacak efekt
38        musicSource.Stop();
39    }
40
41
42

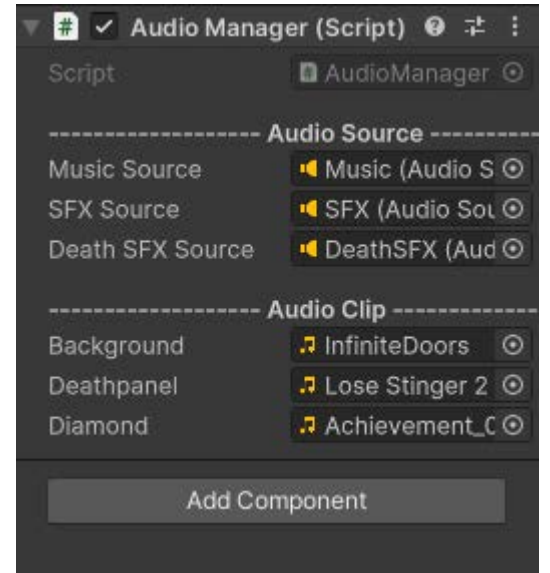
```

AudioManager.cs script dosyasını oluşturduk ve bunu AudioManager objesi componenti olarak ekleyeceğiz. **Start()** fonksiyonunda musicSource olarak tanımladığımız ve inspectordan ekleyeceğimiz arka plan müziği için çalışmasını başlattık. Zaten Loop olarak işaretlenmişti. **PlaySFX()** fonksiyonu ise çağrıldığı zaman çalışsın ve bu fonksiyon bir parametre alıyor. Bu parametreyi çağrıldığı sınıf içinde hangi ses efektini çalmak istiyorsa onu parametre olarak verebilecek.

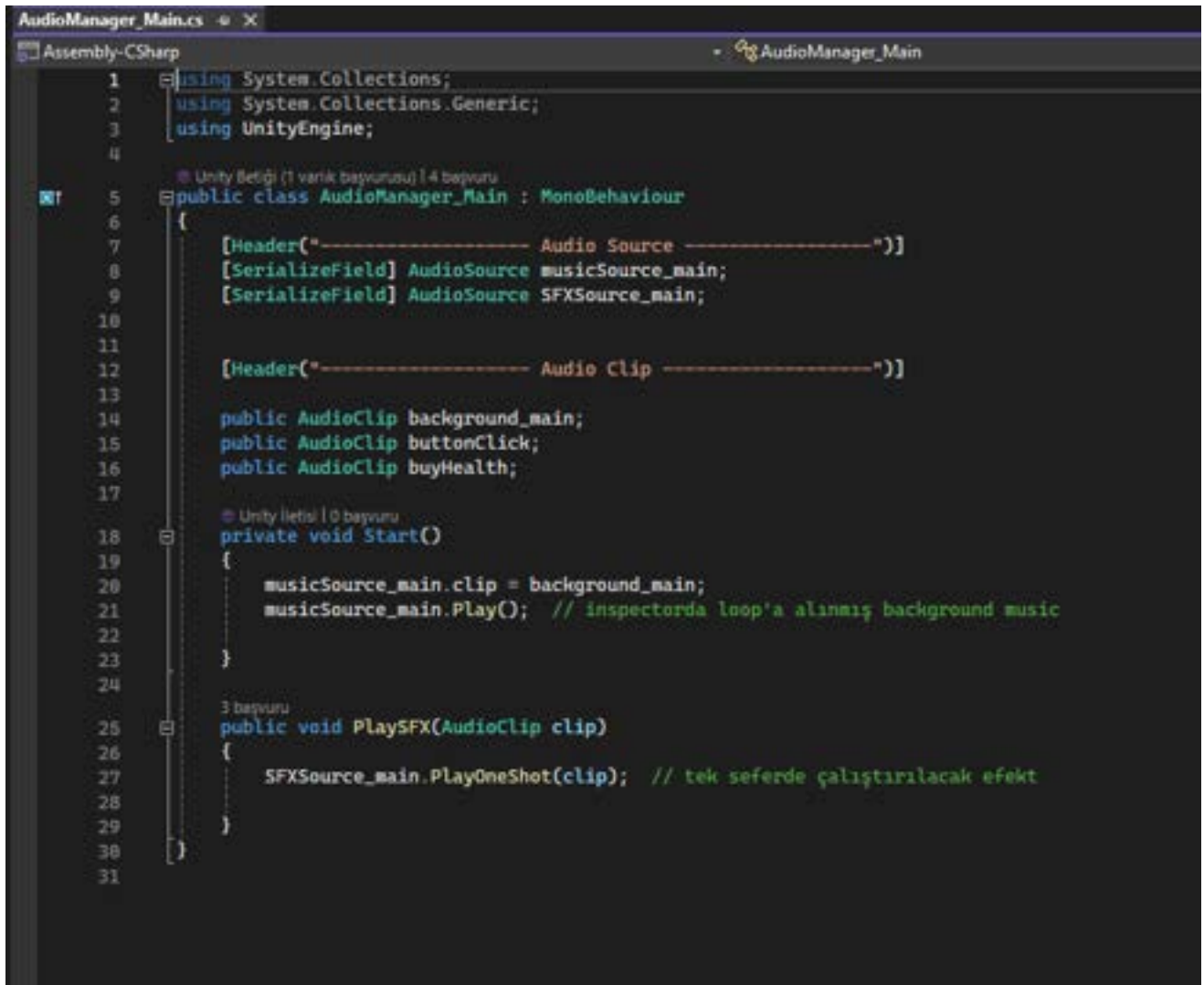
DeathPlaySFX() fonksiyon da oyun sonu ekranı için özel olarak bir kaynak oluşturmak istedik. Arka plan müziği volüme %50'de bırakırken oyun sonu ekranında çalışacak ses efektinin farklı ses seviyesinde olmasını istediğim için ekstra bir ses kaynağı belirledik ve farklı bir fonksiyon oluşturduk. Bu arada **PlayOneShot()** ile sadece bir kere çalıştırılması sağlanıyor parametre olarak alınan ses klibinin. Ayrıca **musicSource.Stop()** yaparak arka plan müziğini de durduruyoruz.

AudioManager objesi içerisine attığımız AudioManager scriptimizin [SerializeField] yapılmış değişkenlerine burada atamalarını yaptık.

Ayrıca child objelerinin de MusicSource , SFX Source ve Death SFX Source olarak bu scripte gösterdik.



12.2 – Ana Menü : Müzik ve Ses Efektleri



Ana menü için aynı ayarlamaları yapıyoruz sadece buttonlara özel button click ses efektleri eklendi. Aynı bir AudioManager objesi ve child objeleri oluşturuldu ve **AudioSource** componentleri oluşturuldu.

II. Bölüm

Bu bölümde oyunumuzu geliştirdikten sonra farklı bir oynanış deneyimi sağlamak için yapay zekanın bir alt dalı olan **görüntü işleme** yöntemlerinden yararlanarak webcam üzerinden kullanıcın el hareketlerinin koordinat bilgilerini çekip bunu Unity ortamına aktararak bunu sahne üzerinde hareket eden karakterimizin sağa , sola ve yukarı hareketleriyle senkronize etmek ana amacımız. Projemizin iki ana bölümden oluştuğunu söylemiştik. Bu iki ana başlığın bir araya getirildiği ve tüm çalışmanın sonucunu görebileceğimiz noktaya geldik.

Öncelikle hand tracking sürecinin de nasıl ilerlediğinden bahsetmek isteriz. Çünkü birden fazla yol denedik ve çok fazla hatayla, buglarla karşılaştık. İlk denediğimiz yöntem Unity Asset store da bulunan Hand Detection pluginiydi. Lakin bu plugine destek çok uzun süre kesilmişti ve bizim Unity versiyonumuz üzerinde stabil çalışmıyordu paket ve birçok hata veriyordu. Bu sebeple bu planı iptal etmek zorunda kaldık. Bir başka paket daha bulduk fakat bu da ücretli bir paketti. Hem bu ücreti karşılaştırmak istemiyorduk , hem de bunun daha kolay bir yolu olabileceğini düşünüyorduk.

Son olarak çözümü **OpenCV Mediapipe ve CvZone** kullanarak Unity'nin kapsamı dışına çıkmayı seçtik. Acaba Unity'e dışarıdan veri çekebilir miyiz? Sorusunu sormaya başladık ve bu mentaliteyle yola çıktık. Bunu **socket** bağlantısı sayesinde verileri çekebileceğimizi araştırmalarımızda gördük.

TCP ya da UDP hangi protokol ilk akla gelenlerdi lakin birçok teknolojiye desteklediğini birçok makaleyi incelediğimizde gördük. (Bluetooth vs)

TCP protokolü paketin karşı tarafı ulaşp ulaşmadığını kontrol etmekle çok vakit harcıyordu çünkü protokol paketi karşı tarafa ulaştırma garantisi veriyor. **UDP** protokolü ise daha hızlı çalışan bir protokol çünkü paketin ulaşp/ulaşmadığıyla ilgilenmiyordu. Hız burda birinci önceliğimiz oldu çünkü oyun içerisindeki gecikmeyi minimize etmek istiyorduk.

Hala gerçek zamanlı olarak en iyi haline ulaşamamış olsa da projemiz amacına ulaşmış oldu. Daha önce ki yöntemlerin hiçbirinden bu kadar ilerlememiştik.

1. Python : OpenCV Mediapipe – CvZone Hand Detector

Unity tarafının dışına çıkıp Python ortamında farklı kütüphaneleri ve frameworklerini kullanarak Hand Detection aşamasını çözmemiz gerekiyor öncelikle. İki farklı yöntem denedik Unity ortamına aktarılacak **veri** burada çok kritik bir rol oynuyordu. Çünkü sürekli akan bir veri olacak ve bunu işleyip Unity'nin işleyebileceği bir hale getirmek gerekiyor. Bunun oluşturduğu bazı sorunları göreceğiz şimdi.

1.1 Landmarklist

Projenin son halinde kullandığımız Python scriptini inceleyelim. Burada yorum satırlarına dikkat etmenizi istiyoruz çünkü deneyipte vazgeçtiğimiz ilk yöntem “**Landmarklist**” verilerini çekmek oldu.

```

1  import socket
2
3  import cv2
4  from cvzone.HandTrackingModule import HandDetector
5
6  #UDP server config
7  socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8  serverAddress = ("127.0.0.1", 5052) #localhost
9
10
11 #elde edilen frame boyutları gerekli
12 height = 480
13 width = 640
14
15
16 #capture nesnesi yaratılır
17 cap = cv2.VideoCapture(0)
18 #cap nesnesi 3-4.param
19 cap.set( propId: 3, width)
20 cap.set( propId: 4, height)
21
22 #tek el kontrolü
23 detector = HandDetector(maxHands=1)
24
25 #sürekli veri okunacak capture nesnesinden
26 while True:
27     #bool deger ve okunan frame dondurur
28     success, img = cap.read()
29     #bulunan el landmarklarını aynı frame üzerine yaz, buldugun elleri aynı sekilde döndür.
30     hands, img = detector.findHands(img)
31
32     data = []
33     if hands:
34         hand = hands[0]
35         centerpoint = hand['center']
36         #print(centerpoint)
37
38
39         #landmarkList = hand['lmList']
40         #print(landmarkList)
41

```

```
45     #box = hand['bbox']
46     #print(box)
47     #handtype = hand['type']
48     #print(handtype)
49
50     #list icinde list kurtul - veri isle -> unity gonder
51     # for lm in landmarklist:
52         #data.append(lm[0]) #data objeye ekle
53         #data.append(height-lm[1]) #opencv origin -> left top -- unity -> right top
54         #data.append(lm[2])
55         #socket.sendto(str.encode(str(data)),serverAddress)
56         socket.sendto(str.encode(str(centerpoint)),serverAddress)
57
58     #print(data)
59
60
61     #cv2.imshow("ForestRun_CAM",img)
62
63     flipped = cv2.flip(img, flipCode: 1)
64     cv2.imshow( winname: "ForestRun_cam",flipped)
65
66     # kapanmaması için
67     cv2.waitKey(1)
68
69
70
```

Önelikle Python scriptinden bahsedeceğiz , arka planda neler oluyor görelim. İlk olarak paketlerimizi import ediyoruz. **Socket** bağlantısı , **OpenCV** , **CvZone** Hand Tracking Modülü üzerinde Hand Detector eklendi.

Daha sonra **UDP** socket ayarlamaları **socket** isimli değişken üzerine yapılarak gönderildi. **serverAddress** değişkenimizi belirledik “127.0.0.1” yani **localhost** gösterdik ve **port: 5052** olarak gösterdik. Burada dikkat edilmesi gereken Unity tarafında da bu script ile haberleşecek bir script yazmamız gerekecek ve bu iki script **5052** portu üzerinden iletişim kuracak. Bu port sürekli akan kamera koordinatlarının aktarılacağı port olacak ve Unity tarafında gelen bu verileri dinleyeceğiz port üzerinden. Ayrıca buraya port atarken sisteminizin bu portu kullanmadığına emin olmanı gerekiyor yoksa hata alıyorsunuz.

Kameranın açıldığında çözünürlüğünü ayarladık 640x480 bize yetecektir. Bir capture nesnesi yaratıyoruz “**cap**”. VideoCapture(0) burada ki “0” değeri default olan ve cihazınızda tek bir kamera varsa ona bağlanacaktır. **Cap.set** diyerek capture nesnemizin 3 ve 4.parametrelerine bu yükseklik ve genişliği veriyoruz ki kamera açıldığında bu çözünürlüklerle açılsın.

Bir Detector tanımlıyoruz ve HandDetector() fonksiyonu çağırılıyor. Bu fonksiyon birçok parametre alıyor , bizim için tek bir parametre yeterli. **maxHands = 1** yakalanacak el sayısını parametre olarak verdik. İki elimizi kullanmamıza gerek yok. Tek bir elin sağa – sola hareketi bizim için oyunu kontrol etmek açısından yeterli olacaktır.

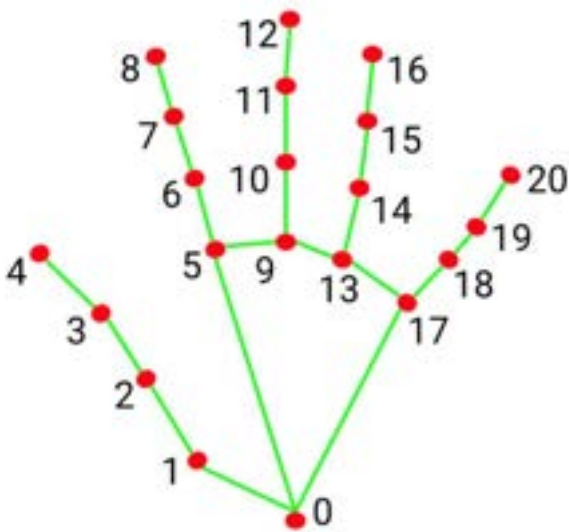
Sonsuz bir döngü açıyoruz çünkü sürekli kameradan pozisyon bilgisini yakalamamız gerekiyor. `İmg = cap.read()` ile görüntü okunuyor. Aynı zamanda Detector ile görüntüdeki eli yakalıyoruz.

Bir Dizi oluşturuyoruz ve verileri bunun üzerinde tutacağız. IF blogunda eğer el yakalanmışsa yakalanan el görüntüsünü **hand** değişkenine aktarılıyor. Burda birkaç satır atlayıp **landmarklist**'i anlatacağım bu script görüntüsü projenin son halinden bir görüntü, o sebeple asıl çözümü görüyoruz. Ama sebeplerine burada değinmek istedik. Socket üzerinden **sendto** diyerek veriyi gönderiyoruz.

Flipped() fonksiyonunu kullanmak çok önemli değil ama kameralardaki aynalama efektini kapatmak için kullandık. Görüntüyü çeviriyor ve **imshow()** diyerek gösteriyoruz. **waitKey(1)** diyerek kameranın hemen kapanmasını önlüyoruz.

Land Mark List yöntemini denediğimiz ve sonra iptal ettiğimiz çözüme gelecek olursak FOR döngüsü içerisinde **data.append()** diyerek 3 index değerine üç farklı değer ekliyoruz. Bunlar X-Y-Z koordinatları. Landmark ise kullanıcının elinin kamerada görünmesiyle birlikte detectorun elin farklı noktalarına işaretlediği 21 farklı noktayı temsil ediyor ve bu 21 farklı noktanın kendi X-Y-Z değerleri bulunuyor.

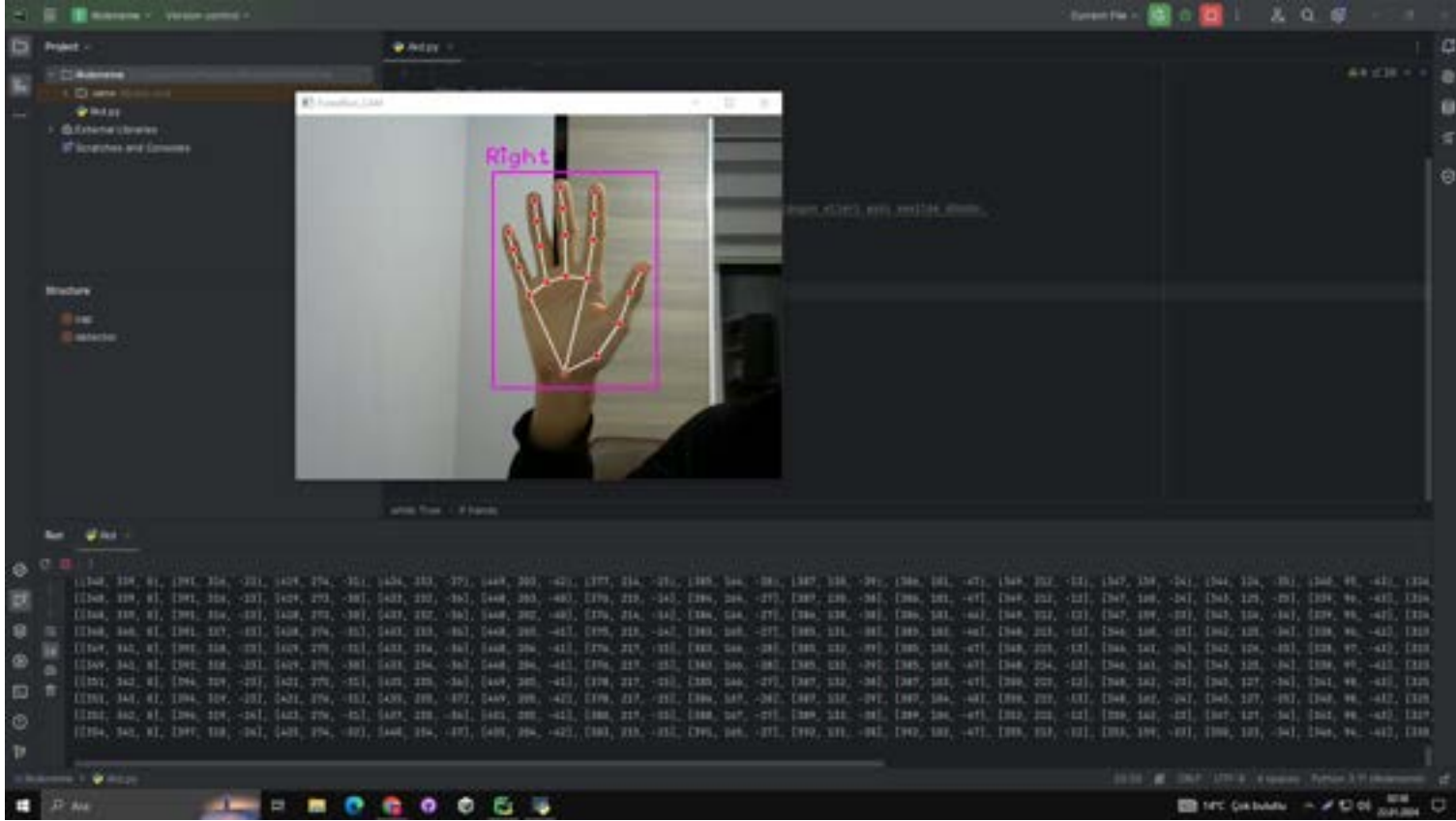
Burada karşılaştığımız sorunlardan biri landmarklist veriyi Unity'nin direkt olarak işleyebileceği şekilde üretmiyor. Liste içinde liste olarak çıktı üretiyor.



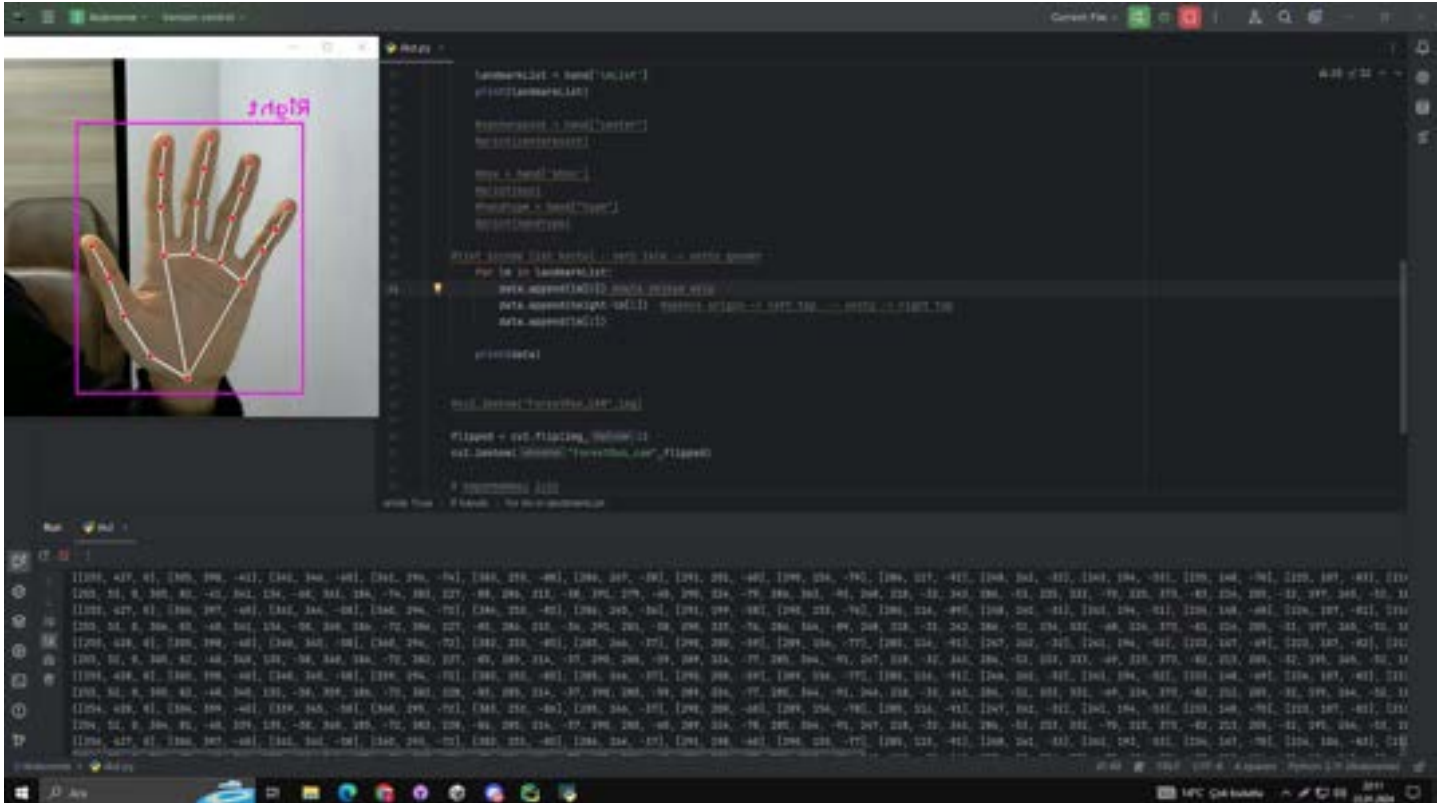
- 0. WRIST
- 1. THUMB_CMC
- 2. THUMB_MCP
- 3. THUMB_IP
- 4. THUMB_TIP
- 5. INDEX_FINGER_MCP
- 6. INDEX_FINGER_PIP
- 7. INDEX_FINGER_DIP
- 8. INDEX_FINGER_TIP
- 9. MIDDLE_FINGER_MCP
- 10. MIDDLE_FINGER_PIP

- 11. MIDDLE_FINGER_DIP
- 12. MIDDLE_FINGER_TIP
- 13. RING_FINGER_MCP
- 14. RING_FINGER_PIP
- 15. RING_FINGER_DIP
- 16. RING_FINGER_TIP
- 17. PINKY_MCP
- 18. PINKY_PIP
- 19. PINKY_DIP
- 20. PINKY_TIP

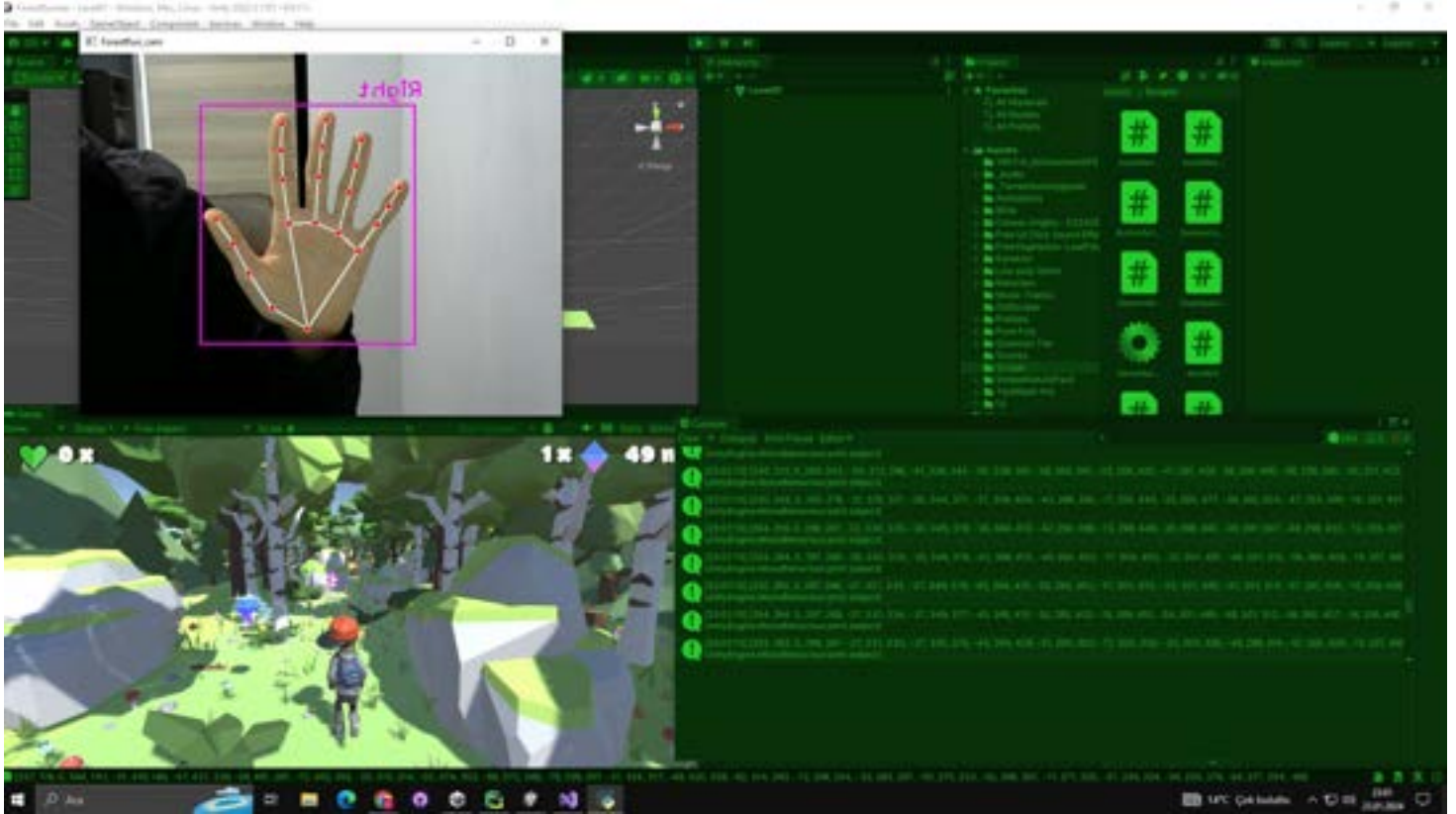
Görüntü işleme ile Unity 3D Runner Oyun



Her bir 21 landmark'ın [x,y,z] şeklinde bir de genel bir [[]] liste içinde tutulduğunu görebiliyoruz. Bunları console bastırdığımızda 21 landmarkın verisi anlık olarak akıyordu. Bunu işlenebilir hale getirmek gerekiyor.



21 landmarkın ayrı ayrı listelerde olmasını düzelttik ve genel bir liste içinde virgül ile ayrılmış bir şekilde veriyi elde edebildik. Bu veriyi Unity’e gönderdik ama bu kadar akan verinin kontrol edilmesi işlenmesi **gecikmeye** ve **performans kaybına** sebep oluyordu.



ÇÖZÜM :

O sebeple bu kadar veriyi çekmek istemiyorduk ve bizi başka bir yola sevk etti bu durum.

Hand["center"] gibi basit bir çözüm ürettik. Bu çözümün kısaca özeti $21 \times 3 = 63$ farklı X-Y-Z değeri yerine (345,245) gibi basit bir veri gönderiyor. Sadece X ve Y değerlerini gönderiyoruz ve bu **CENTER** ifadesi bir property ve yakalanan el objesinin merkez noktasını baz alıyor ve sürekli onu döndürüyor. Artık bu veriyi Unity’e aktarmak ve işlenebilir hale getirip kullanmak , performans testlerini yapmamız gerekiyor.

2. Unity : Socket Üzerinden UDP Protokolüyle Verinin Alınması

```

UDPReceive.cs
1  using UnityEngine;
2  using System;
3  using System.Text;
4  using System.Net;
5  using System.Net.Sockets;
6  using System.Threading;
7
8  public class UDPReceive : MonoBehaviour
9  {
10
11      Thread receiveThread;
12      UdpClient client;
13      public int port = 5052;
14      public bool startReceiving = true;
15      public bool printToConsole = false;
16      public string data;
17
18
19      public void Start()
20      {
21
22          receiveThread = new Thread(
23              new ThreadStart(ReceiveData));
24          receiveThread.IsBackground = true;
25          receiveThread.Start();
26      }
27

```

Öncelikle bu portu dinleyecek bir **Thread** kullanmamız gerekiyor. Bu Thread'i **Start()** fonksiyonunda başlatıyoruz. **IsBackground = true** ile arka planda çalışmaya devam etmesi için **iş parçacığını** başlatıyoruz.

Burada **ReceiveData()** fonksiyonu ile veriyi çekmeye başlayacağız. Lakin burada yapılan testlerde karşılaştığımız problemlerden biri de Thread'in sürekli veriyi dinlediği ve alıp getirdiği için Unity ortamına karakterimiz oyun sonu ekranına geldiğinde yani kaybettiğinde haliyle hareket etmiyor ve hala veriyi almaya devam ediyoruz. Bu sebeple threadin sonlandırılması gerekiyor karakter **isAlive=false** olduğunda.

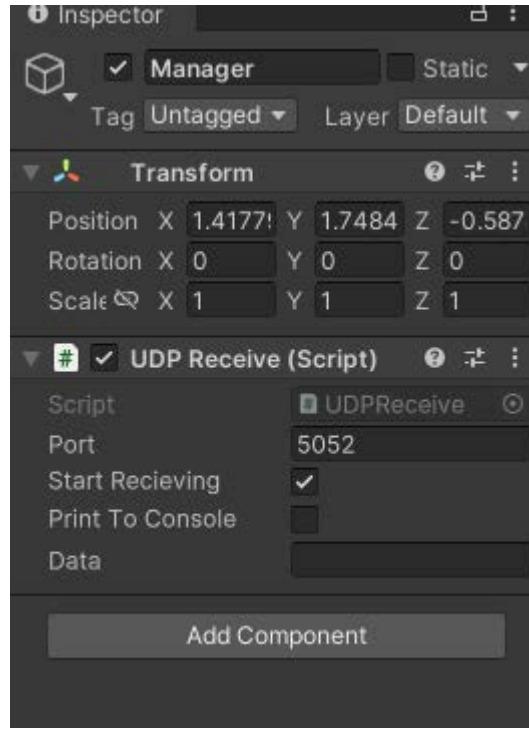
```

30 1 başvuru
31 private void ReceiveData()
32 {
33     client = new UdpClient(port);
34     while (startRecieving)
35     {
36
37         try
38         {
39             IPEndPoint anyIP = new IPEndPoint(IPAddress.Any, 0);
40             byte[] dataByte = client.Receive(ref anyIP);
41             data = Encoding.UTF8.GetString(dataByte);
42
43             if (printToConsole) { print(data); }
44         }
45         catch (Exception err)
46         {
47             print(err.ToString());
48         }
49     }
50 }
51 1 başvuru
52 void StopThread()
53 {
54
55     receiveThread.Abort();
56
57 }
58 Unity İletisi | 0 başvuru
59 private void FixedUpdate()
60 {
61     if (PlayerMove.isAlive == false)
62     {
63         StopThread();
64     }
65 }
66 }
67

```

Try-catch blokları arasında herhangi bir exception yakalama ihmalimiz yüksek olacağından özellikle Threadlerle çalışırken dikkatli davranmamız gerekiyor. Beklenmedik birçok bugla karşılaşıyoruz.

Ayrıca bu UDP scriptini bağlamak ve Python ile iletişimi sağlamak için bir obje oluşturduk ve scripti component olarak aktardık.



3 .Unity : Karakterin El Takibiyle Yönlendirilmesi

Karakterimizi yönlendirmek için yeni bir script oluşturduktan sonra **UDPReceive** sınıfımıza ulaşp verileri işlemeye başladık. Kodumuzun daha iyi çalışması için FixedUpdate bölümüne yazmayı tercih ettik. Anlık verilerle uğraşacağımız için de try/catch yöntemlerini de kullanmamız gerekiyor. Çünkü veri gelmezse veya hatalı bir veri gelirse kodumuz çökebilir. Bunun için veri işlemlerini **try** bloğuna yazmakla başladık.


```
private void FixedUpdate()
{
    try
    {
        string data = udpReceive.data;

        ;
        data = data.Replace('(', ' ');
        data = data.Replace(')', ' ');
        string[] points = data.Split(' ');
    }
}
```

Gelen veriyi data adında string'e aktardıktan sonra işlemeye başladık. Veriler parantez içerdiği için bunları değiştirerek başladık. Ardından gelen verileri **Split** özelliği yardımıyla **virgül** ile ayırıp **points** dizisine atadık.

```
//TryParse ile degerleri kontrol ediyoruz
if (float.TryParse(points[0], NumberStyles.Float, CultureInfo.InvariantCulture, out x))
{
    // Başarılı dönüşüm, x değişkenini kullanabilirsiniz.
    x = x / 10;
    if (float.TryParse(points[1], NumberStyles.Float, CultureInfo.InvariantCulture, out y))
    {
        y = y / 10;
    }
}
```

Sonrasında Parse işlemiyle **X ve Y** değerleri olacak şekilde atama gerçekleştirdik. Ancak veriler bazen hatalı geldiği için sık sık parse işleminde **hata** almaya başladık. Buna çözüm olarak bir kontrol işlemi eklemeye karar verdik. **TryParse** ile X ve Y değerlerinin eğer parse işlemi başarılı olursa gerçekleşecek şekilde ayarladık. Ardından sayılar çok büyük olduğu için 10'a böldük. Bu değerleri kullanarak karakterimizi yönlendireceğiz.


```

}
if (x > 35) //sağ sol
{
    if (this.gameObject.transform.position.x > LevelSinir.solTrf)
    {
        transform.localPosition = (transform.position + new Vector3(-0.02f, 0, 0));
    }
}
else if (x < 35)
{
    if (this.gameObject.transform.position.x < LevelSinir.sagTrf)
    {
        transform.localPosition = (transform.position + new Vector3(0.02f, 0, 0));
    }
}
}

```

Atadığımız X değerini test ederek optimal orta değerini “35” olarak seçtik. Buna göre değerin 35’ten küçük veya büyük olması karakterimizin sağa yada sola yönlendirileceği anlamına geliyor. Script karakterimizin içinde olduğu için direkt transform özelliğine ulaşp **localPosition** değerine ekleme yaparak sağ-sol hareketini yapabiliyoruz. Tabii ki level’imizdeki sınır değerlerini önceden kontrol ediyoruz ki karakterimiz harita dışına çıkmasın.

```

}
if (canJump && y < 8 && y > 0)
{
    StartCoroutine(JumpCorrutine());
}
}
catch (Exception e)
{
    Debug.LogError(e);
}
}

```

Zıplama işlemi için de optimal eşik değeri “8” olarak saptadık. Yine aynı yöntemle kontrol ediyoruz. Yeniden fazla zıplamalara engel olabilmek için **Corroutine** yoluyla yetki kontrolü yapıyoruz. Ve zıplamaya parametre göndererek zıplama işlemini gerçekleştiriyoruz. Ve son olarak eğer verilerle alakalı bir sorun olursa “catch” ile konsola hata olarak yazdırıyoruz.

KAYNAKÇA

- <https://docs.unity3d.com/ScriptReference/index.html>
- <https://libraries.io/pypi/cvzone>
- <https://assetstore.unity.com/>
- <https://kenney.nl/>
- <https://www.mixamo.com/#/>
- <https://app.diagrams.net/>
- <https://gamedesignskills.com/game-design/level-designer/>
- <https://book.leveldesignbook.com/introduction>
- <https://learn.unity.com/>
- http://repo.darmajaya.ac.id/5648/1/Beginning%203D%20Game%20Development%20with%20Unity%204_%20All-in-one%2C%20multi-platform%20game%20development.pdf
- https://luisnavarrete.com/design/pdf/unity_scripting.pdf