İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

OYUN MOTORLARINDA GÜZERGAH BELİRLEME

YÜKSEK LİSANS TEZİ

Abdulkadir BAYTİMUR

Bilişim Uygulamaları Anabilim Dalı

Coğrafi Bilgi Teknolojileri Programı

TEMMUZ 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

OYUN MOTORLARINDA GÜZERGAH BELİRLEME

YÜKSEK LİSANS TEZİ

Abdulkadir BAYTİMUR
(706181001)

Bilişim Uygulamaları Anabilim Dalı

Coğrafi Bilgi Teknolojileri Programı

Tez Danışmanı: Doç. Dr. Caner GÜNEY

TEMMUZ 2021

İTÜ, Lisansüstü Eğitim Enstitüsü'nün 706181001 numaralı Yüksek Lisans Öğrencisi Abdulkadir BAYTİMUR, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı "OYUN MOTORLARINDA GÜZERGAH BELİRLEME" başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Doç. Dr. Caner GÜNEY** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Prof. Dr. Rahmi Nurhan ÇELİK** İstanbul Teknik Üniversitesi	.....
	Doç. Dr. Özgün AKÇAY Çanakkale Üniversitesi	.....

Teslim Tarihi : 11 Haziran 2021
Savunma Tarihi : 08 Temmuz 2021

Aileme,

ÖNSÖZ

Bu yüksek lisans tezinin konusunun belirlenmesi, gerekli araştırmaların yapılması ve projenin tamamlanması aşamalarında verdiği katkılardan ötürü, tez danışmanım sayın Doç. Dr. Caner GÜNEY'e yardımları, teşviki ve rehberliği için teşekkür ederim.

Ayrıca bu yüksek lisans tezinin hazırlanması sırasında sabır gösteren aileme ve arkadaşlarıma da teşekkür ederim.

Haziran 2021

Abdulkadir BAYTİMUR
(Geomatik Mühendisi)

İÇİNDEKİLER

Sayfa

KISALTMALAR

AR **:** Augmented Reality
CBS **:** Coğragi Bilgi Sistemi
CPU **:** Central Processing Unit
DEM **:** Digital Elevation Model
DOTS **:** Data-Oriented Technology Stack
DTM **:** Digital Terrain Model
ECS **:** Entity-Component-System
GIS **:** Geographic Information System
GUI **:** Graphical User Interface
NavMesh **:** Navigation Mesh
PC **:** Personal Computer
TIN **:** Triangulated Irregular Network
VR **:** Virtual Reality
XR **:** Extended Reality

SEMBOLLER

C#	**:** Programlama dili
C++	**:** Programlama dili
F	**:** En uygun yolu bulmak için kullanılan G + H değeridir
G	**:** Seçilen hücre dizisi boyunca başlangıç konumundan hedef konumuna giden yolun maliyetidir
H	**:** Hedef konumuna ulaşması için gereken sezgisel maliyettir

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

OYUN MOTORLARINDA GÜZERGAH BELİRLEME

ÖZET

Geçmişte oyun motorları sadece oyun yapımında kullanılan programlar olarak kabul edilirdi. Günümüzde ise oyun motorları sağladıkları özgür üç boyutlu ortam, çalışma anındaki anlık etkileşim olanağı ve kişiselleştirilebilir araçlarının olması gibi nedenlerle; oyun endüstrisi dışında da birçok sektörde kullanım alanı bulmuştur. Bunlardan biri de farklı tematik alanlarda mekansal veri kümelerinin analizi ve görselleştirmesi üzerinde çalışan Coğrafi Bilgi Sistemleri (CBS) sektörüdür. CBS uygulamalarında hali hazırda kullanılan bir çok açık kaynaklı ve ticari yazılım bulunsa da, bunlar genel olarak üç boyutlu ortamda çalışmak için gerekli grafik hesaplama yetisine sahip olmadıklarından bu tür işler için hantal kalabilmektedir. Özetle CBS yazılımları oyun motorları kadar performans sunamamaktadır. Bu tez çalışmasında, oyun, oyun motoru ve CBS arasındaki ilişkiden bahsedilmeye çalışılmıştır. Çalışma kapsamında NavMesh sistemi ve yapay zeka karakterlerinin otonom güzergah bulması konusu açıklanmıştır. Unity oyun motoru kullanılarak güzergah belirleme uygulaması geliştirilmiş ve daha sonra Data-Oriented Technology Stack gibi oyun motorundaki farklı yaklaşımlar kullanılarak güzergah belirleme uygulamasının nasıl daha performanslı hale getirildiği açıklanmıştır.

Günümüzde güzergah belirleme pek çok farklı alanla kullanılmaktadır. Amacı ne olursa olsun güzergah belirlerken temel olarak bir başlangıç ve bir hedef noktası olması gerekmektedir. Bu iki nokta arasında farklı gereksinimler ve parametrelerle (en az maliyetli, en verimli, en kısa yol vb.) güzergah belirlenebilmesi için Djikstra ve A* gibi güzergah belirleme algoritmaları kullanılmaktadır. Tez çalışması kapsamında Unity oyun motorunda nesne odaklı (object oriented) ve veri odaklı (data oriented) olmak üzere A* algoritması kullanılarak bir çok güzergah belirleme uygulaması geliştirilmiştir. Nesne odaklı yaklaşımda Unity oyun motoru doğrudan kullanılmıştır. Veri odaklı yaklaşımda Unity oyun motorunda henüz yeni bir teknoloji olan ve farklı bir bakış açısı getiren Data-Oriented Technology Stack (DOTS) kütüphanesi eklenerek geliştirme yapılmıştır. DOTS; C# İş Sistemi, Entity-Component-System, Burst Derleyicisi olmak üzere temel olarak 3 farklı bileşeni içermekte ve hepsinin ortak amacı performans artışı sağlamaktır. DOTS ile bu performans artışı; üç boyutlu nesnelerin, nesne yerine veri olarak işlenmesi, C# kodunun daha optimize yazımı ve kodların derlenmesindeki yöntem farkı ile sağlanmaktadır.

Bu çalışma için, Unity'de birim karelerden oluşan ve 20*20 ve 100*100 boyutlarında olan, kare şeklinde 2 farklı grid ağ oluşturulmuştur. Daha sonra A* güzergah belirleme algoritması simüle edilmiştir. Kare gridin en sol alt köşesinden (başlangıç noktası) en sağ üst köşesine (hedef noktası); eş zamanlı olarak 5'er ve 50'şer güzergah belirleme işleri aynı anda çalıştırılmıştır. Bu test nesne odaklı sistem, C# İş Sistemi, C# İş Sistemi + çoklu işlem parçacığı metodu ve son olarak tüm C# İş Sistemi + çoklu işlem parçacığı metodu + Burst Derleyicisi birlikte kullanımı biçiminde 4 farklı yöntem için denenmiştir. Nesne odaklı sistemde 20*20'lik gridde 5'er güzergah hesaplama eş

zamanlı yapıldığında 330-400 milisaniye arasında, sadece C# İş Sistemi 1.6 milisaniyede, C# İş Sistemi + çoklu işlem parçacığı metodu ile 0.11 milisaniyede ve C# İş Sistemi + çoklu işlem parçacığı + Burst Derleyicisi birlikte kullanıldığında 0.04 milisaniyede güzergah bulunmuştur. 100*100'lük gridde 50'şer güzergah hesaplama eş zamanlı yapıldığında ise; nesne odaklı sistem çökmüş ve çalışmamıştır, C# İş Sistemi 46 milisaniyede, C# İş Sistemi + çoklu işlem parçacığı 4.5 milisaniyede, C# İş Sistemi + çoklu işlem parçacığı metodu + Burst Derleyicisi birlikte 1.5 milisaniyede hesaplamayı tamamlamıştır. Çalışmada beklenildiği gibi veri odaklı yöntemin nesne odaklı yönteme göre çok daha performanslı çalıştığı belirlenmiştir.

Sonraki aşamada bu uygulamaya ECS sistemi entegre edilmiştir. Böylece ECS ile birlikte DOTS'un tüm elemanları kullanılmıştır. Hareketli ajanlar oyun nesnesinden, varlıklara dönüştürülmüştür. Kullanılan iş sistemleri tarafından, bu varlıklar için güzergah bulma algoritmaları çalıştırılmıştır. 100 hareketli ajanın güzergah belirleme işi Burst Derleyicisi kapalıyken 9.76 milisaniyede, Burst Derleyicisi açıkken 1.01 milisaniyede hesaplanmıştır. Daha sonra, ECS'nin sınırlarını zorlamak adına Burst Derleyicisi açıkken, 20000 ajan kullanılmış ve güzergah bulma işi 236.04 milisaniyede hesaplanmıştır.

ROUTING IN GAME ENGINES

SUMMARY

In the past, game engines were considered only as programs used in game production. Today, game engines have found use in many sectors outside the game industry; due to reasons such as free three-dimensional environment they provide, the opportunity for instant interaction at run-time, and the fact that they have customizable tools. One of these sectors is the Geographic Information Systems (GIS) field, which works on the analysis and visualization of spatial datasets in different thematic areas. Although there are many open source and commercial softwares currently used in GIS applications, they can be cumbersome for this type of work as they generally do not have the necessary graphical computation ability to work in a three-dimensional environment. In summary, GIS software cannot offer as much performance as game engines.

In this thesis, the relationship between game, game engines and GIS were tried to be mentioned. It was tried to explain with several examples that how some video games benefit from real world spatial data. While the game industry makes great use of the real world; it has been mentioned that the GIS industry does not sufficiently benefit from games and game engines. Although GIS softwares are generally considered sufficient in 2D studies, it cannot quickly apply the technological possibilities brought by game engines in 3D systems and interactive applications. Game engines are an advanced development environment for building GIS applications and running simulations. They offer a high-level visualization system, include pyhsics engine that allows realistic simulations. Also they provide an animation system, audio engine, cross-platform software support. Game engines have advanced script structure that allows users to build their own tools. In addition, game engines have an extremely large developer community and is constantly developing new tools and features.

Storing and using location data is critical in the GIS approach. For this reason, GIS softwares work integrated with databases. The game development environment supports the storage and use of datasets from various sources, just like a GIS software does. Since there is an advanced script structure in game engines, it is possible to find a library suitable for all kinds of database needs. By adding these libraries to the game engine, data sets can be worked on. Terrains can be brought into the game engine, models of pyhsical objects (building, road, etc.) can be added. In addition, the information of each object (meta-data) can be stored in databases and used by the game engine.

Within the scope of the study, the subject of autonomous path finding system and artificial intelligence (AI) characters were explained. In this thesis, two different AI navigation method were explained and used in different applications. These are grid-based system and navigation mesh system. In grid-based system, the world is divided up into an even grid and A* search algorithm is used to find the shortest path in this grid. A navigation mesh system divedes the world surface into polygons and calculates

shortest path according to these areas. Unity game engine's own navigation mesh system (NavMesh) were explained and a pathfinding application was develeoped by using the Unity game engine. After that a grid-base pathfinding application were developed by using a basic A* algorithm. This application was later used for performance comparison with subsequent applications.

Today, pathfinding is used in many different fields. Regardless of its purpose, it is essential to have a starting point and a destination while determining the route. Pathfinding algorithms such as Djikstra and A* are used to determine routes between these two points with different requirements and parameters (least cost, most efficient, shortest route, etc.).Within the scope of the thesis, many pathfinding applications have been developed using the A* algorithm for object-oriented and data-oriented approaches in the Unity game engine environment. Unity game engine is used directly in the object-oriented approach. In the data-oriented approach, the Unity game engine has been developed by adding the Data-Oriented Technology Stack (DOTS) library, which is a new technology and brings a different perspective. DOTS includes 3 different components such as C# Job System, Entity-Component-System and Burst Compiler and the common purpose of all of them is to increase performance. With DOTS, this performance increase is achieved by processing three-dimensional objects as data instead of objects, more optimized writing of C# code, and the difference in the way the codes are compiled.

C# Job System makes use of multi-core processors to run several jobs at the same time. One of the main problems with Unity game engine performance has always been that the main-thread is single-threaded. With C# Job System, many paths can be calculated at the same time and this leads to an increase in performance. ECS writes code from a different perspective; it requires dividing the code into two, as logic and data. Instead of standard game objects, there are entities, components and systems that converts game objects to data which also increases performance. And lastly, the Burst Compiler takes all written C# code and turns it into highly optimized machine code. It automatically makes the code being compiled benefit from certain optimizations for certain platforms.

For this study, 2 different grid networks in the form of squares, consisting of unit squares and having dimensions of 20*20 and 100*100, were created in Unity. Then the A* pathfinding algorithm was simulated from the bottom left corner (starting point) to the top right corner (destination point) of the square grid. 5 and 50 pathfinding jobs were run simultaneously. This test has been tested for 4 different methods such as object-oriented system, C# Job System, C# Job System + multi-threading method and finally all C# Job System + multi-threading method + Burst Compiler together. When calculating 5 routes on a 20*20 grid simultaneously; the object-oriented system took between 330-400 milliseconds, C# Job System took 1.6 milliseconds, C# Job System + multi-threading took 0.11 milliseconds and when C# Job System + multi-threading + Burst Compiler used together, the paths were found in 0.04 milliseconds. When calculating 50 routes simultaneously on a 100*100 grid; the object-based system crashed and did not work, C# Job System completed the calculations in 46 milliseconds, C# Job System + multi-threading comleted the calculations in 4.5 milliseconds, C# Job System + multi-threading + Burst Compiler together completed the calculations in 1.5 milliseconds. As expected in the study, it was determined that the data-oriented method works much more efficiently than the object-oriented method. In the next stage, the ECS was integrated into this application. Thus, all elements of DOTS were used. Moving agents have been converted from game objects

to entities. Pathfinding algorithms for these entities were run by the job systems. Pathfinding job of the 100 agents was calculated in 9.76 milliseconds while Burst Compiler turned off, and calculated in 1.01 milliseconds while Burst Compiler turned on. Then, to push the limits of ECS, with the Burst Compiler turned on, 20000 agents were used and the pathfinding job was calculated in 236.04 milliseconds.

1. GİRİŞ

Günümüzde Coğrafi Bilgi Sistemleri (CBS) için yapılan bir çok farklı tanım bulunmaktadır. Bir tanıma göre CBS, grafik ve grafik olmayan veriler ile bütün içerisinde çalışan bir bilgi sistemidir (Yomralıoğlu & Çelik, 1994). Başka bir tanımda CBS, belirli bir amaç doğrultusunda karmaşık sorunları çözmek için konumsal verilerin yönetimini, manipülasyonunu, analizini, modellenmesini, temsil edilmesini ve görüntülenmesini kolaylaştıran bir donanım, yazılım ve prosedürler sistemidir (Escobar, Hunter, Bishop, & Zerger, 2005). Daha basit bir tanıma göre CBS, Dünya yüzeyindeki yerleri tanımlayan, verileri tutabilen ve kullanabilen bir bilgisayar sistemidir (Environmental Systems Research Institute (ESRI), 1992).

Yukarıdaki tanımlarda da görüldüğü üzere aslında CBS teknolojisi, ana konusu konum verisi olan ve mekansal veriyi yapılacak iş doğrultusunda kullanabilen bir sistem ve bir mekansal karar destek aracı olarak düşünülebilir. Merkezinde veri olduğu için; bu veriyi, günümüzde yetenekleri giderek artan oyun motorlarıyla birlikte kullanmak, klasik CBS yazılımlarıyla yapılabilecek işlerden çok daha fazlasının yapılmasına olanak tanıyacaktır.

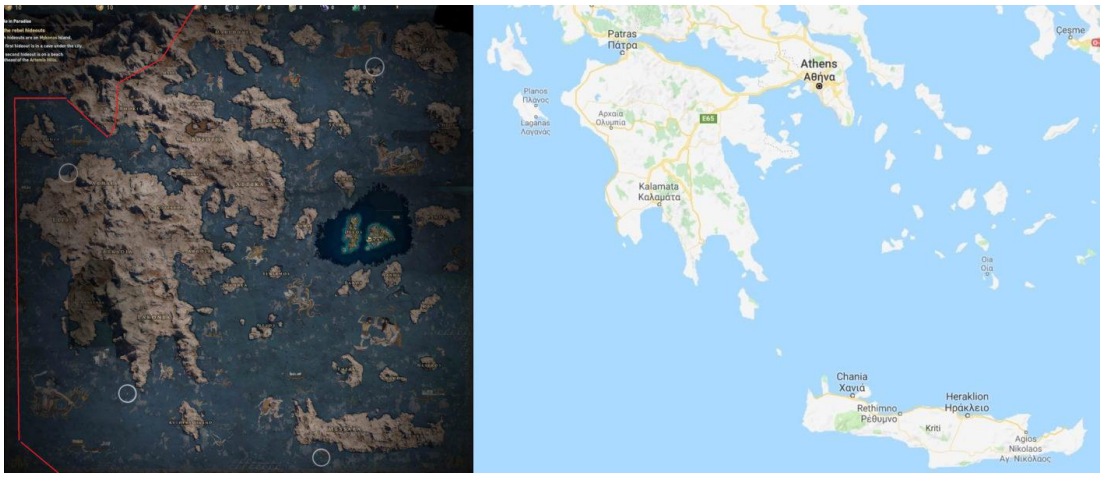
Peki CBS ile oyunların birbiriyle nasıl bir ilişkisi olabilir? Günümüzde birçok video oyunu, geliştiricileri tarafından gerçek dünya coğrafyası dikkate alınarak tasarlanmaktadır. Nehirler, dağlar, vadiler, binalar, ağaçlar ve diğer tüm yapılar oyun motorunun kullandığı fizik motoru, render sistemi, ses sistemi, animasyon sistemi vb. bileşenleri sayesinde gerçekçi bir şekilde sanal ortama aktarılır ve simüle edilir. Oyunlar genellikle fantezi dünyalara sahip olsalarda, dayanak noktaları gerçek dünyadır. Örneğin, "SimCity" ve "Cities Skylines" gibi oyunlar ile şehirler kurup bunu yönetmek mümkündür (Şekil 1.1). Bu tür oyunlarda binalar, yollar, köprüler, parklar vb. nesneler kullanarak şehir geliştirilmektedir. Ayrıca su şebekesi, elektrik dağıtım yolları, hava kirliliği vb. farklı görsel katmanları kullanmak ve bunlara göre şehrin arz talep durumu yönetmek mümkündür. Cities Skylines oyununda, gerçek dünya arazi verisini içe aktarıp istenilen 3 boyutlu arazide oynanabilmektedir.

Şekil 1.1: “Cities Skylines” oyunundan bir şehir görseli.

Bir diğer örnek, “World of Warcraft” oyunundaki gibi devasa 3 boyutlu arazilere ve şehirlere sahip oyunlarda, yüzlerce farklı bölgenin farklı ölçeklerdeki haritalarını ve karakterin o anki konumunu gösteren mini haritayı (mini-map) kullanmak mümkündür. Şekilde 1.2’de görüldüğü gibi, bu haritalar farklı detay seviyelerine sahip olup, modellenen 3 boyutlu dünyayı bire bir betimlemektedir. Şekil 1.2’de, 1 numaralı harita tüm kıtaları içeren oyun dünyasının, 2 numaralı harita bir kıtanın tamamının, 3 numaralı harita bir bölgenin haritasıdır ve 4 numara harita ise bir mini haritadır. Mini harita, oyunlarda kontrol edilen karakteri oyun dünyasında yönlendirmeye yardımcı olmak için genellikle ekranın bir köşesine yerleştirilen minyatür bir haritadır. Bu haritada karakterin hareketleri takip edilerek bulunduğu anlık konum bilgisi sürekli güncellenmektedir. Yaygın olarak mini haritaya dahil edilen özellikler arasında; oyuncunun anlık konumu, ulaşılması gereken hedefler ve arazi bilgisi bulunmaktadır. Evreninin genişliği sebebiyle, bu tür oyunlarda haritaların kullanımı zorunludur. Bu tür oyunlarda, oluşturulan dünyadaki bir konumdan başka bir konuma gidebilmek için kullanılan bu haritalar bir koordinat sistemine sahiptir ve dolayısıyla her noktanın koordinat bilgisi bulunmaktadır. Bir başka deyişle, oyunlar sanal bir dünya olmalarına rağmen konum bilgisi etkin olarak kullanılmaktadır.

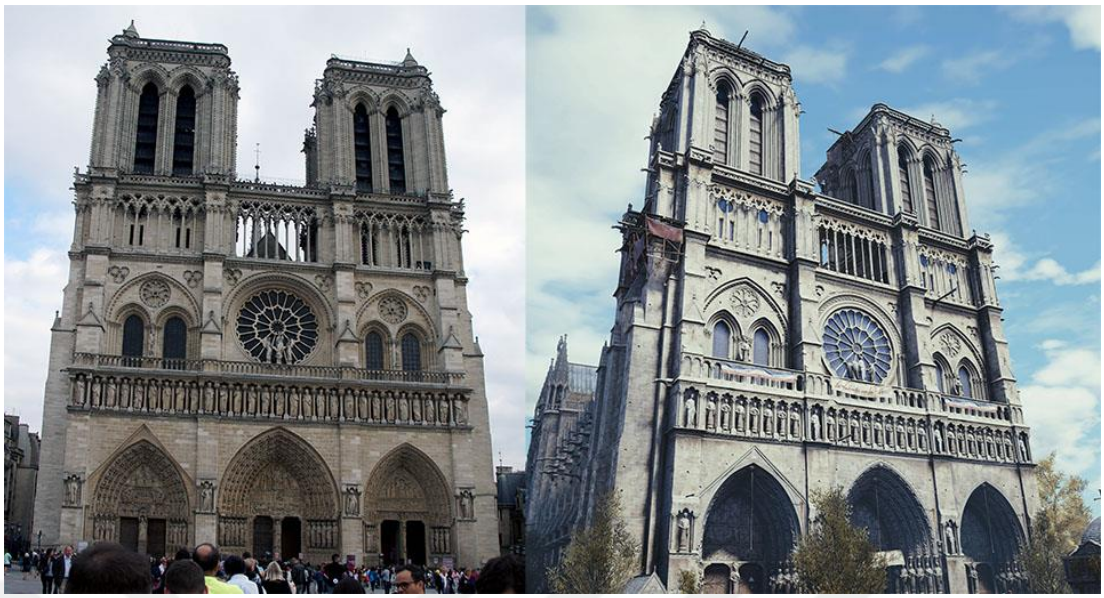
Şekil 1.2: "World of Warcraft" oyununda farklı detay seviyelerindeki haritalar.

Bunun dışında gerçek dünyayı grafik olarak modelleyen oyunların sayısı giderek artmaktadır. Kimi oyunlar tarihi yapıları, kimi oyunlar arazileri modellemektedirler. Bu tür oyunlar eğlence, eğitim ve turizm amaçlı kullanılabilmektedirler. Örneğin, Antik Yunanistan'da geçen "Assassin's Creed Odyssey" oyununda bir çok tarihi eser ve tarihi yapı, alanında uzman danışmanlar ile çalışılarak orjinaline uygun şekilde modellenmiştir. Aynı zamanda oyunda yaratılan devasa 3 boyutlu dünyanın haritasının üretilmesi de söz konusudur. Şekil 1.3'te görüldüğü üzere, oyundaki harita günümüz Yunanistan'a oldukça benzemektedir. Ancak bu bir video oyun olduğu için orjinaline bire bir uymasını beklemekten çok, oyuncuya sağlayacağı eğlence düşünülerek 3 boyutlu dünyası modellenip geliştirilmiştir. Bu nedenle bazı farklılıkların bulunması normal kabul edilmelidir (Gibson, 2018). Bununla birlikte gerek görülmesi durumunda oyun ortamı gerçek dünyaya bire bir uygun olacak biçimde modellenebilir. Şekil 1.3'teki oyun haritasındaki her ada 3 boyutlu modellenmiş olup, içerisinde gezilip farklı oyun karakterleriyle etkileşime geçilebilinmektedir.



Şekil 1.3: "Assassin's Creed Odyssey" oyunundaki Antik Yunanistan haritası (solda) ve günümüz Yunanistan haritası (sağda).

Fransız ihtilali yıllarındaki Paris şehrininde geçen "Assassin's Creed Unity" oyununda, şehrin o zamanki kültürü, tarihi eserleri, sokakları ve binaları uzun bir geliştirme süreci sonucunda tarihi açıdan doğru biçimde modellenmiştir (Arif, 2014). Şekil 1.4'te de görüldüğü gibi, oyunun en dikkat çeken yanı, orjinali ile bire bir modellenen Notre Dame Katedrali'dir. Tarihçiler, sanatçılar ve mühendisler ile birlikte bu katedralin gerçeğe uygun modellenmesi yaklaşık 14 ay sürmüştür (Lowrie, 2019). Oyunun yapımcısı olan Ubisoft şirketi, lazer tarayıcılar ile katedreli baştan sona tarayıp 3 boyutlu lazer nokta bulutlarını oluşturmuştur (Woods, 2019). Bu sayede katedralin sadece dışı değil, iç kısmı da detaylı bir şekilde modellenmiştir. 2019 yılında meydana gelen Notre Dame Katedrali yangınında harap olan tarihi eserin restorasyonu için; oyunun yapımcısı, katedral ile ilgili ellerindeki tüm lazer nokta bulutlarını ve 3 boyutlu modelleri, istenilmesi durumunda ilgililerle paylaşabileceğini ifade etmiştir (Holt, 2019). Görüldüğü gibi mekansal veri ve oyunlar arasında bir ilişki bulunmaktadır. Bu ilişki; oyun motorlarının kabiliyetlerinin artması ve oyunları çalıştıran cihazların donanımsal özelliklerinin gelişmesiyle, sanal dünyanın giderek gerçek dünyaya benzemesine sebep olmaktadır. Oyun motorunun sahip olduğu teknolojiler ile bu kadar detaylı şehirler yüksek poligon sayılı modeller ile hazırlanabilmektedir ve bunun yanında kullanıcılar bu şehirlerle anlık olarak etkileşime girebilmektedir. Klasik CBS yazılımlarının ise henüz bu tür yetileri bulunmamaktadır. CBS teknolojisinden faydalanarak, oyun motoru ile Paris şehri modellenmiş; daha sonra bir afet anında ise oyundaki modelden yararlanabilmek mümkün olmuştur.



Şekil 1.4: Notre Dame Katedrali'nin gerçek (solda) ve oyun içindeki 3 boyutlu modellenmiş (sağda) hali.

Bunlardan başka Londra'yı 3 boyutlu modellemiş olan ve yakın gelecekte (2030 yılının başlarında) geçen "Watch Dogs: Legion" oyunu (O'Malley, 2019), bazı ilkokullarda şehirlerin modellenip eğlenceli bir şekilde öğretilmesi için öğrencilere oynatılan "Minecraft" oyunu (Macgregor, 2019), Antik Maya, Aztek ve İnka gibi medeniyetlerinin tapınaklarının ve eşyalarının modellendiği "Tomb Raider" oyunları (Power, 2018) mekansal bilişim ve oyun alanları arasındaki ilişkiye örnek olarak gösterilebilir.

Eğer bu kapsam biraz daha genişletilmek isternirse, güneş sistemi de dahil olmak üzere Samanyolu Galaksisi'nin modellendiği ve birçok astronomik simülasyonun yapılmasına olanak sağlayan, fizik-tabanlı uzay simülasyon oyunu olan "Universe Sandbox" örnek olarak verilebilir (Şekil 1.5). Bu oyunda; yer çekimi, çarpışma, iklim vb. simülasyonlar yapılabilmekte ve gök cisimlerinin kütle, çap, hız, sıcaklık vb. nitelikleri ayarlanabilmektedir (Giant Army, 2020).

Şekil 1.5: "Universe Sandbox" oyununun görseli

Yukarıda bahsedilen tüm bu örneklerin odağında mekansal/coğrafi veri bulunmaktadır. İster gerçek dünyadan modellensin, ister kurgu dünyası olsun bu oyunlarda kullanılan mekansal veri kendi içinde tutarlıdır ve kendi haritaları vardır. Diğer bir ifadeyle oyun motorlarında her bir nesnenin mekansal verisi saklanmaktadır. Oyun endüstrisi gerçek dünyadan bu kadar beslenirken, benzer yaklaşımla CBS sektörünün de oyun motorlarını daha çok kullanması gerektiği düşünülebilir. Ayrıca istisnalar olmakla birlikte, örneklerde verilen çoğu çalışma mühendislik kaygısından çok, sanatsal kaygıyla yapılmıştır. Bu oyunların bazı kısımları, gerçek dünyadaki karşılığına uygun olsa bile diğer kısımlarında sanatsal yaklaşımlar ön plandadır ve sanatçı yorumu bulunur. Bunun sebebi, son ürünün insanlara eğlence vadeden video oyunları olmasıdır. Eğer oyun motorları ve CBS kullanılarak baştan sona bir mühendislik çalışması yapılmak istenirse, bunun için bir engel bulunmamaktadır.

1.1 CBS ve Oyun Motorları İlişkisi

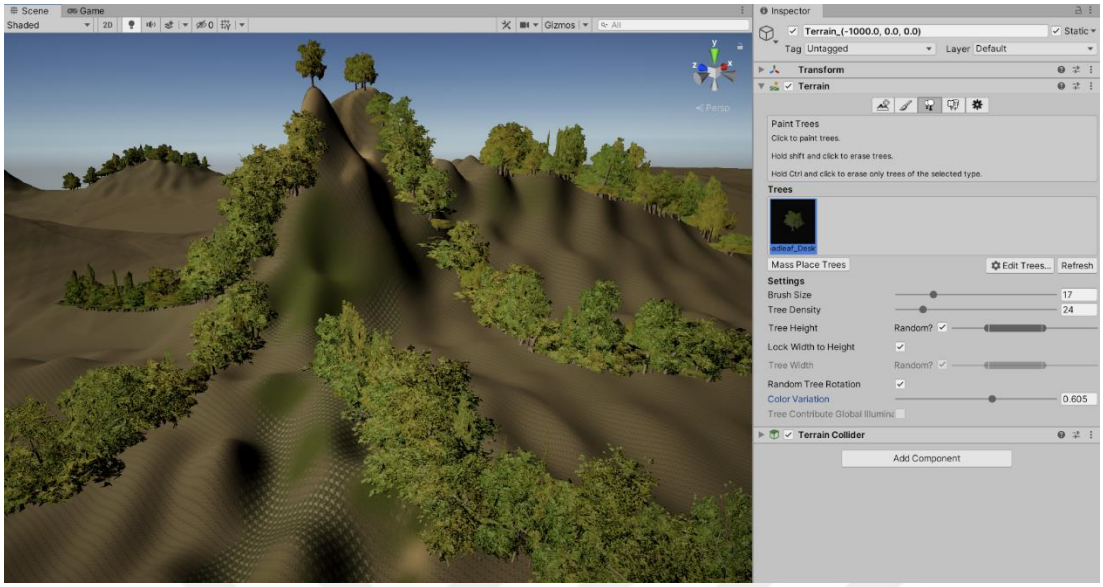
Günümüzde oyun motorları oyun geliştirmenin yanında birçok farklı alanda da kullanılmaktadır. CBS sektöründe ise henüz istenilen kullanım oranlarına çıkmamasına rağmen, bu sektörde de ilgi başlamıştır. Klasik CBS yazılımları 2 boyutlu çalışma alanında genel olarak yeterli görülse de, 3 boyutlu sistemlerde ve etkileşimli uygulamalar yapımında oyun motorlarının getirdiği teknolojik olanakları hızlıca uygulayamamaktadır. Klasik CBS yazılımları oyun motorlarıyla karşılaştırıldığında özellikle grafik açısından hantal ve genellikle geliştirmeye pek

elverişli olmayan bir yapıda bulunmaktadır. Oyun motorları; içersindeki fizik ve grafik motoru, projeye özel araçların yapımını sağlayan script yapısı, gelişmiş GUI sistemi, kolay kullanımı, uygulamaya çalışma anında dinamik olarak etkileşimde bulunabilmek gibi özellikleri sebebiyle CBS yazılımlarına birer alternatif olma potansiyeni taşımaktadır. Ayrıca genel olarak ücretsiz olmaları ve kolayca öğrenilebiliyor olmaları da üstünlükleri arasında kabul edilebilir.

Oyun motorları, CBS uygulamaları oluşturmak ve simülasyonlar yapmak için ileri düzey bir geliştirme ortamıdır. Bunun bazı temel sebepleri vardır. Üst düzey bir görselleştirme sistemi sunmakta ve bunun optimizasyonunu yapabilmektedir. Bunun yanında oyun motorunun kendine has sunduğu diğer özellikler arasında gerçekçi simülasyonların yapılmasına olanak tanıyan fizik motoru, her türlü animasyon için animasyon sistemi, düşük hesaplama maliyetiyle performanslı ve kaliteli görsel efektler sunan partikül sistemi bulunmaktadır. Oyun motorları, CBS yazılımlarının genellikle sahip olmadığı, farklı platformlar arası yazılım desteğine sahiptir. Bu sayede tek bir ortamda geliştirilen uygulama kolaylıkla mobil, PC, sanal gerçeklik ve oyun konsolu gibi farklı platformlara aktarılabilir. CBS topluluğuna kıyasla, son derece geniş bir geliştirici topluluğu vardır ve bu topluluk oyun motorları için sürekli yeni araçlar ve özellikler geliştirmektedir. Ayrıca oyun geliştiriciler, uygulamalarda kullanmak üzere CBS içeriklerine gün geçtikçe daha fazla ilgi göstermektedir.

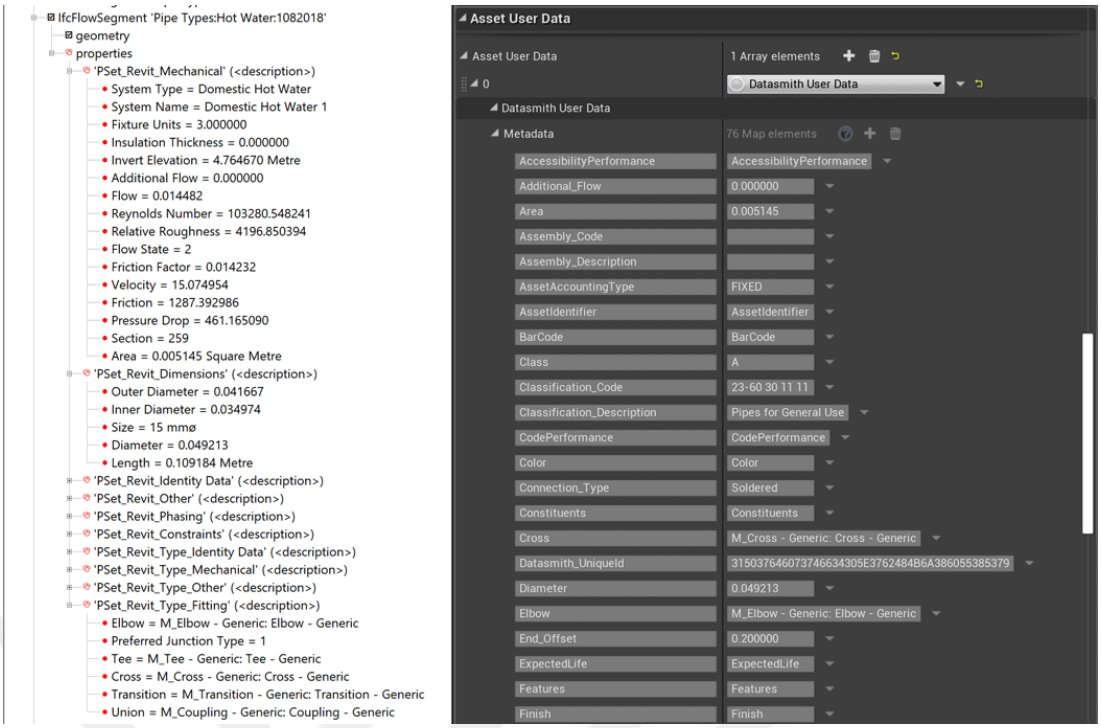
Sayısal yükseklik modeli (DEM), üçgenlenmiş düzensiz ağ (TIN) ve sayısal arazi modeli (DTM) 3 boyutlu arazileri işleme ve görüntüleme yeteneğine sahip yazılımlarda sıklıkla kullanılır (Mat, Shariff, Zulkifli, Rahim, & Mahayudin, 2014). Bu gibi arazi verileri ile çalışabilmek oyun motorlarıyla da mümkündür. Günümüzde, artık çoğu oyun motorunda kendi arazi editörü bulunmaktadır (Şekil 1.6). Bu editörler gerçekçi arazi modellemek için özelleşmiş araçlardır. Geliştiriciler tarafından hazırlanan kütüphaneler ile, DEM, TIN gibi verileri kullanılarak da 3 boyutlu araziler oluşturulabilmektedir. Ayrıca arazi modellemenin yanında farklı türdeki ve detay seviyesindeki 3 boyutlu ağaç nesneleri de araziye eklenebilmektedir. Arazi modeli hazırlandıktan sonra ayrıntı düzeyi (level of detail) denilen, araziye bakarken farklı yakınlık seviyeleri için farklı nesne detayları gösteren sistem de kullanılabilmektedir. Bu sayede kameranın araziye uzaklığına göre, arazinin ve üzerindeki nesnelerin poligon sayıları otomatik olarak artıp azalabilmektedir. Geniş bir açıyla uzaktan arazi

gözlemleniyorsa, düşük poligonlu modeller yüklenir ve sistem performansı, kullanıcı tarafından görülebilen alan sınırları artmış olmasına rağmen azalmamaktadır.



Şekil 1.6: Unity'de 3 boyutlu arazi editörü.

CBS yaklaşımında konum verisinin saklanıp, kullanılması kritik öneme sahiptir. Bu nedenle CBS yazılımları veri tabanlarıyla entegre bir şekilde çalışmaktadır. Oyun geliştirme ortamı, tıpkı bir CBS yazılımının yaptığı gibi, çeşitli kaynaklardan veri kümelerinin saklanmasını ve kullanılmasını desteklemektedir. Oyun motorlarında gelişmiş bir script yapısı olduğu için, her türlü veri tabanı ihtiyacına uygun bir kütüphane bulmak mümkündür. Bu kütüphaneler oyun motoruna eklenerek veri kümeleri üzerinde çalışılabilinir. Bunun dışında motorun desteklediği script diliyle kişisel gereksinimlere göre özel bir araç da hazırlamak mümkündür. Araziler oyun motoruna getirilebilir, fiziksel nesnelerin modelleri (bina, yol vb.) eklenebilir ve değiştirilebilir. Ayrıca her nesnenin bilgileri veri tabanlarında saklanıp, oyun motoru tarafından kullanılabilinir. Şekil 1.7'de görüldüğü üzere, Unreal oyun motorunda bulunan Datasmith aracı ile mimarlık veya mühendislik yazılımlarından elde edilen nesnelerin meta verileri (metadata) oyun motorunda ilgili nesneye aktarılmaktadır. Bu sayede meta verilere oyun motoru ulaşır ve bu veriler istenildiği gibi kullanılır.



Şekil 1.7: Mühendislik yazılımındaki meta veri (solda), Datasmith ile Unreal oyun motoruna aktarılmış meta veri (sağda).

Günümüzde, oyun motorları ile sanal gerçeklik (VR) ve arttırılmış gerçeklik (AR) uygulamaları geliştirilebilmektedir. En iyi sanal gerçeklik simülasyonlarını üretebilmek için artık yakın geçmişte olduğu gibi yüksek maliyetli iş istasyonlarına ve özel yazılımlara gereksinim kalmamıştır. Geçmişte; sanal gerçeklik, arttırılmış gerçeklik ve yüksek kaliteli fiziksel simülasyon, yalnız yüksek bütçeli fonlara sahip olabilen araştırmacılar tarafından kullanılabilmekteydi (Lewis & Jacobson, 2002). Bugün, en hızlı, en gerçekçi simülasyonlara ve karmaşık grafiklere sahip olmanın yolu, oyun yazılımı çalıştıran standart bilgisayarlardır. En gelişmiş görselleştirme sistemleri (rendering pipelines) artık özel bilimsel makinelerde değil, düşük maliyetli ekran kartlarındadır. En gelişmiş etkileşimli simülasyonlar artık oyunlara güç vermek için tasarlanmış motorlardadır. Sanal gerçeklikte özel bir kask ile görselleştirilen yapılar (arazi, şehir, bina vb.) birinci şahıs görüş açısıyla (first person view) deneyimlenebilmekte, arttırılmış gerçeklikte ise gerçek dünyanın üzerinde sanal yapılar görselleştirilmektedir. Şekil 1.8'de gösterilen örnek çalışmada, arttırılmış gerçeklik kullanılarak yeraltı altyapıları görüntülenmektedir (Rees, 2018).

Şekil 1.8: Arttırılmış gerçeklik kullanılarak yapılmış bir çalışma örneği.

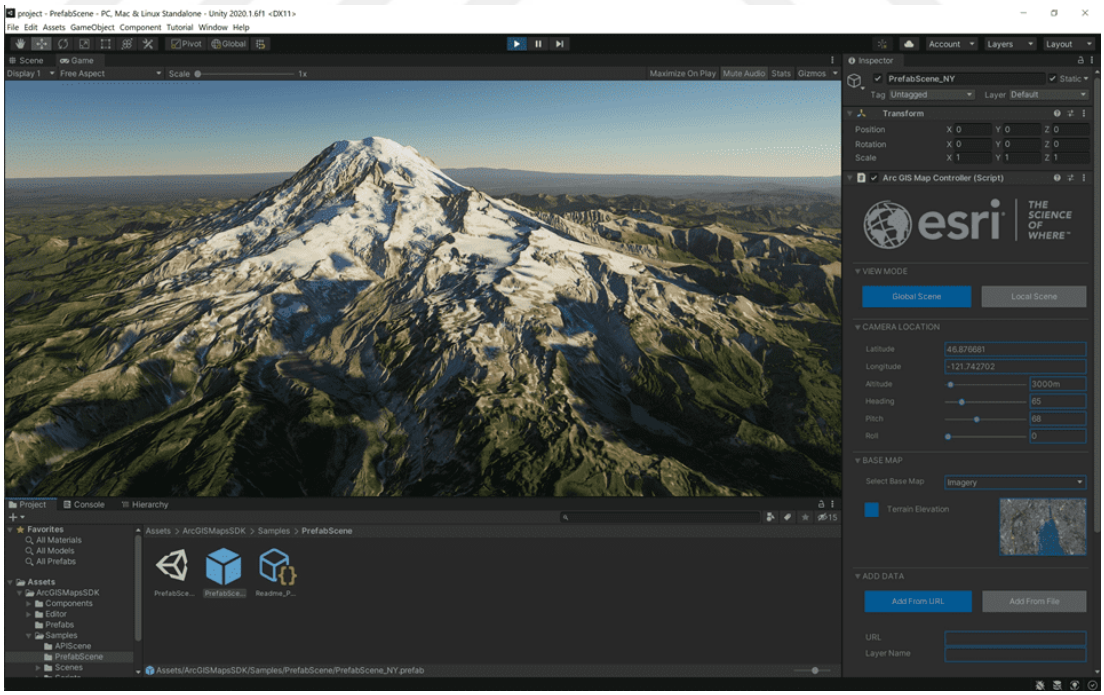
1.2 Oyun motorlarında kullanılan CBS kütüphaneleri

Oyun motorlarında CBS verilerini kullanmak için hazırlanmış bazı özel kütüphaneler bulunmaktadır. Bunların en popülerleri Unity oyun motoru için geliştirilmiş olan ArcGIS Maps ve Mapbox kütüphaneleri ve Unreal oyun motoru için geliştirilmiş olan CesiumJS kütüphanesidir. Bu kütüphaneler sayesinde bir CBS yazılımıyla yapılan çalışmalar oyun motoruna aktarılabilmekte veya oyun motoru üzerinde gerçek dünya verileri kullanılarak yeni bir çalışma yapılabilmektedir. Bunun yanında 3 boyutlu dünya modeli, şehirlerin binalarla birlikte modelleri, yükseklik verilerinin olduğu arazi modelleri gibi farklı özellikleri de bulunabilmektedir. Ayrıca oyun motorlarının sunduğu sanal ve arttılırmış gerçeklik desteği ile bu kütüphaneler kullanılarak daha etkileşimli çalışmalar da hazırlamak mümkündür.

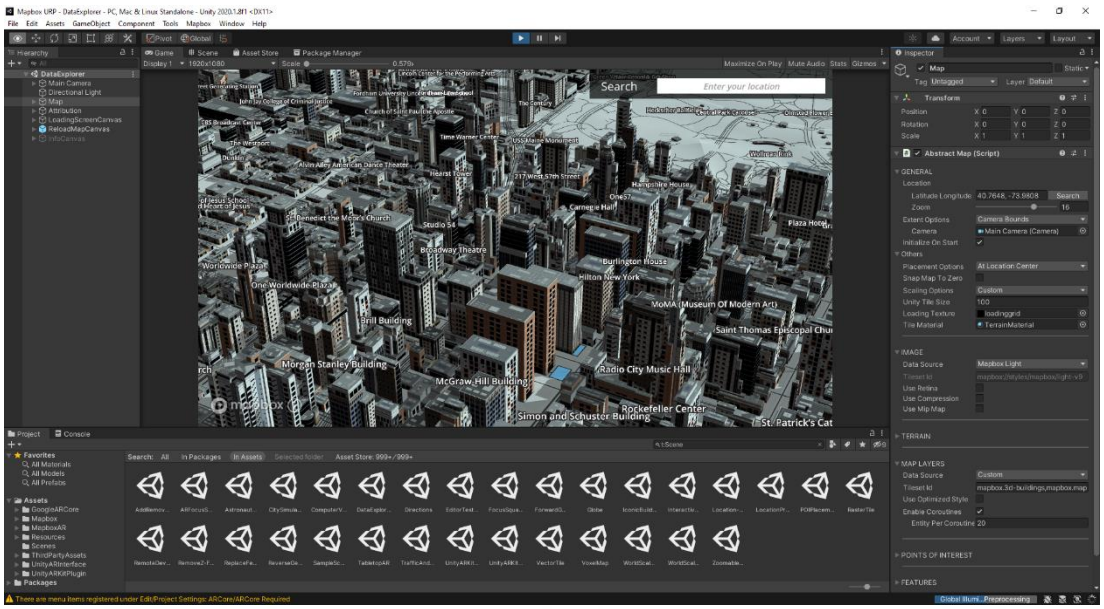
ArcGIS Maps kütüphanesi (Şekil 1.9), Esri firması tarafından geliştirilen, gerçek dünya haritalarına ve 3 boyutlu içeriğe erişim sağlayan; Unity oyun motoru için hazırlanmış bir eklentidir. Mapbox kütüphanesi (Şekil 1.10), Unity'de mekansal verileri işleyebilen bir kütüphanedir. Mapbox, sunucusundan aldığı haritalar ile Unity'de 3 boyutlu arazinin ayarlanıp görselleştirilmesini sağlar. CesiumJS kütüphanesi (Şekil 1.11), Epic Games ile AGI şirketlerinin iş birliği ile hazırlanan; 3

boyutlu mekansal veri teknolojisini ücretsiz ve açık kaynak kodlu bir şekilde Unreal oyun motoruna getiren bir eklentidir. Bu üç kütüphanenin bazı farklılıkları olsa da hemen hemen benzer özellikler sunmaktadırlar.

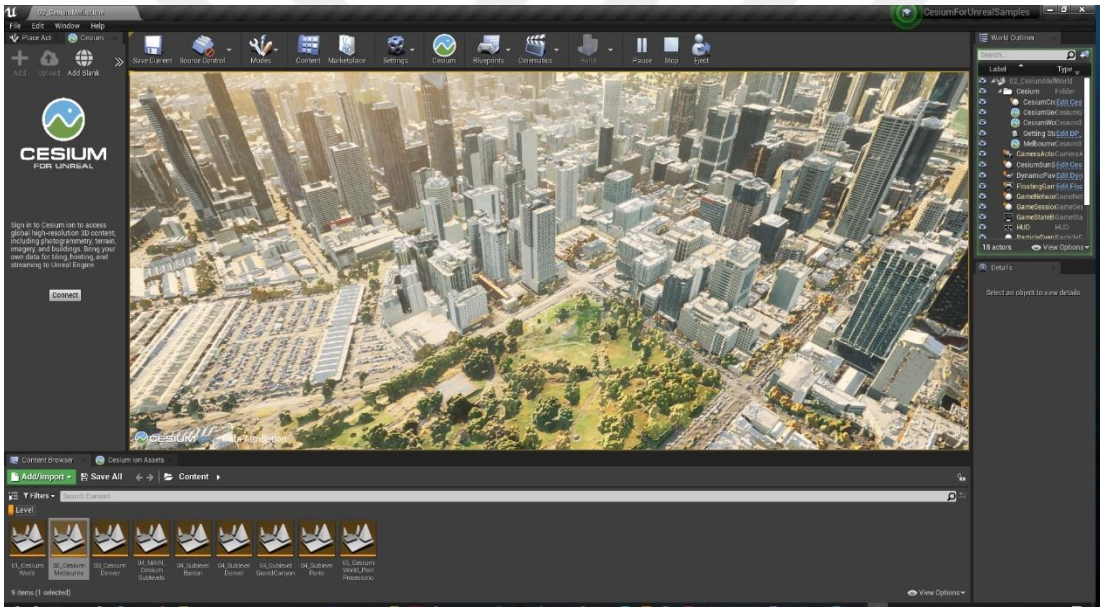
Bu kütüphaneler, mekansal verilerle etkileşimli, 3 boyutlu fotogerçekçi ve genişletilmiş gerçeklik (XR) deneyimleri oluşturmak için ortam sağlamaktadır. Herhangi bir coğrafi referanslı veri, gerçek dünya koordinatlarına dayalı olarak oyun motoru üzerinde kaplanabilir ve bu sanal dünyada istenilen çalışmalar gerçekleştirilebilir. Hem lokal hem de küresel sahneler mevcuttur. Lokal sahnelerle, coğrafi verilerin düzlemsel bir yüzey üzerinde görüntülenmesi sağlanır ve genellikle daha küçük, yerel alanlar için uygun olan farklı koordinat sistemlerinde ve farklı projeksiyonda çalışabilir. Küresel sahnelerle, genellikle geniş alanlar için uygun olan 3 boyutlu bir dünya üzerinde coğrafi verilerin görüntülenmesi sağlanabilir. Bu kütüphaneler ile dünya üzerinde gözlemlenmek istenilen noktanın enlem ve boylam değerleri girilerek oraya gidilir ve kütüphanenin sunduğu farklı türde temel haritalar (basemap) ile görüntülenebilir. Bazı büyük şehirlerde binaların 3 boyutlu modelleri de hazır olarak mevcuttur. Bunun yanında, yükseklik verilerinin kullanılarak, yeryüzünün yükseltileri gerçekçi bir 3 boyutlu model üzerinde gösterilebilinmektedir.



Şekil 1.9: Unity oyun motorunda ArcGIS Maps kütüphanesi.



Şekil 1.10: Unity oyun motorunda Mapbox kütüphanesi.



Şekil 1.11: Unreal oyun motorunda Cesium kütüphanesi.

1.3 Çalışmanın Amacı

Bu tez çalışmasında; Coğrafi Bilgi Sistemi (CBS), oyun ve oyun motoru ilişkisi kurulmaya çalışılacak, oyun geliştirme farklı oyun motorları, ve özellikle Unity oyun motoru bileşenleri üzerinden açıklanacaktır. Ayrıca Unity oyun motorunun kendi navigasyon sistemi olan NavMesh ve yapay zeka ile birlikte nasıl kullanıldığı açıklanacaktır. Son olarak da kullanılan oyun motorunun sunduğu veri odaklı yeni teknoloji ve A* algoritması ile güzergah belirleme uygulamaları geliştirilecektir.

Tez kapsamında yanıt aranan sorulara örnek olarak aşağıdaki sorular gösterilebilir:

- CBS teknolojisi neden oyun motorlarına gereksinim duymaktadır?
- Oyun alanının (game domain), CBS alanına (GIS domain) katkıları neler olabilir?
- Oyun motorlarında güzergah belirleme nasıl gerçekleştirilir?
- Oyun motorunda, nesne odaklı ve veri odaklı sistemlerin farklılıkları nelerdir?
- Oyun motorlarında güzergah belirleme uygulamasının başarımı nasıl arttırılabilir?

2. OYUN MOTORU NEDİR?

Oyun motoru, video oyunlarının ve simülasyon yazılımlarının hızlı ve etkili bir şekilde geliştirilmesi için kullanılan yazılımlardır (Doğan, 2019). Oyun motorunun kullandığı bazı temel bileşenler; grafik motoru (rendering engine/renderer), fizik motoru, ses motoru, script yapısı, animasyon, yapay zeka (AI) ve networking olarak örneklendirilebilir.

2.1 Oyun Motoru Bileşenleri

2.1.1 Grafik motoru

Grafik motoru, 2 ve 3 boyutlu görüntülerin oluşturulmasını sağlamaktadır. Oyun nesnelerinin gerçek zamanlı olarak işlemesi gerekmektedir. Grafik motoru, grafik kartı ile iletişim kurtmaktadır. Kullanılan 2 ve 3 boyutlu modellerin materyalleri ve bu materyallerin farklı dokuları bulunmaktadır. Sahnedeki ışıklandırma, partikül efektleri, gölgeler ve en önemlisi sahnenin görüntülenmesini sağlayan kamera nesnesi grafik motoru ile görselleştirilmektedir. İyi bir grafik motoru farklı formatlardaki oyun nesneleriyle uyumlu çalışabilmelidir.

Grafik motoru oyun motorunun önemli parçalarından birisidir. Oyun çalışırken saniye başına yapılan hesaplamaların yaklaşık %90'ı görselleştirmeye (rendering'e) harcanmaktadır. Oyun mantığı, script'ler, fizik ve sesler geriye kalan diğer hesaplamaları oluşturur. Kullanıcılar tarafından talep edilen görsel kalite her geçen sene artmaktadır. Ancak uygulamaların düşük donanımlı cihazlarda da çalışmasını sağlamak için, geliştiriciler grafik motorunda çeşitli ayarlar yaparak optimize edebilmekte ve kullanıcıya farklı grafik seçenekleri sunabilmektedirler.

Sahnede belirli konumlarda nesneler ve kamera bulunmaktadır. Kamera, gerçek dünya kamera lensi gibi bir konfigürasyona sahiptir. En önemli parametre görüş açısıdır ve genellikle 60 ile 90 derece arasında ayarlanmaktadır. Görüş açısının gösterdiği alan, uygulama çalıştırıldığında kullanıcının gördüğü sahnedir. Görüş açısının büyüklüğüne göre uygulamanın performansı etkilenmektedir.

Nesne oluşturmak için farklı yaklaşımlar bulunmaktadır. Geleneksel metod olan ileri işleme (forward rendering), görünür her geometri için ışıkları ve malzemeleri hesaplamakta ve sonra bunlardan hangisinin kameraya en yakın olduğunu çözüp ve görüntülemektedir. Unity ve Unreal oyun motorlarının şu an hali hazırda kullandığı yaklaşım ise deferred rendering'tir (Akenine-Moller, Haines, & Hoffman, 2008). Geometrinin çoklu geçişleri işlenmektedir (derinlik, normal vektörü, renk vb.). Gölgeleme bu geçişlere göre hesaplanmaktadır. Yalnızca görülen parçalar gölgelenmektedir. Deferred rendering, büyük miktarda ışığı verimli bir şekilde işleyebilir; ancak gölgelerin oluşturulmasına yardımcı olmaz, kenar yumuşatması yoktur ve bu yöntem ile saydam nesneler oluşturulamaz (Unity Technologies, 2020).

Oyun motorları, genellikle bunları oluşturmak ve yapılandırmak için materyeller veya yollar sağlar. Materyeller; ışıksız (unlit), aydınlık (lit), transparan veya yarı saydam olabilmektedir. Dokular (textures), materyali tanımlamaya yardımcı olan görüntülerdir. Materyalin rengini, pürüzlülüğünü, yansıtıcılığını veya yer değiştirmesini (displacement) tanımlayan farklı doku türleri bulunmaktadır.

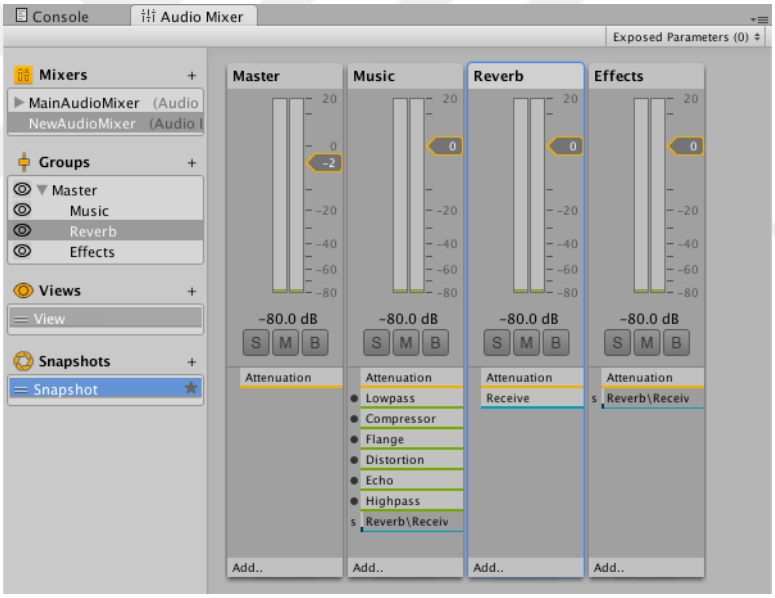
Sahne işlendiğinde, grafik motoru işlem sonrası efektlerini uygular. Bu efektler, nihai sonucun görsel kalitesini iyileştirmektedir. En yaygın kullanılan efektler; kenar yumuşatma (anti-aliasing), ortam örtme (ambient occlusion, kenar ve köşe gölgelendirilmesi) ve renk düzeltmeleridir.

2.1.2 Fizik motoru

Fizik motoru, özellikle de simülasyon uygulamaları için oyunun motorunun en önemli bileşenlerinden birisidir. Gerçek hayat aktivitelerinin, hareketlerinin ve reaksiyonlarının simüle edilmesini sağlamaktadır. Nesnelerde collider (nesnenin fizik hesaplamaları için kullanılan 3 boyutlu sınırı), kütle, sürtünme katsayısı vb. özellikler bulunmaktadır. Motor, nesneler arasındaki hareket vektörlerini ve çarpışmaları hesaplamaktadır. Yer gibi hiç hareket etmeyen statik nesneler ile kütlesi olan, sürtünmeli, kuvvetlere tepki veren, yerçekimi ile düşen katı cisimler fizik motoru sayesinde etkileşime girebilmektedir. Ek olarak fizik motoru, dış kuvvetlerden etkilenip şeklini değiştiren, kumaş gibi yumuşak cisimleri de simüle edebilmektedir. Bazı oyun motorları kendi fizik motorlarını geliştirilmiştir ve bazı oyun motorları da Havok, PhysX gibi harici fizik kütüphanelerini kullanmaktadır (Šmíd, 2017).

2.1.3 Ses motoru

Ses motoru, oyun içi kullanılan tüm ses efektlerini kontrol eder. Oyun nesnelerine, ses dinleyicisi (audio listener) bileşeni eklendiğinde; kullanıcı, çevrede oluşan seslerin hangi yönden geldiğini ve yaklaşık olarak ne kadar uzakta olduğunu duyabilmektedir (spatial audio/3 boyutlu ses). Hızla hareket eden bir ses kaynağı (örneğin bir ambulans sireni) dinlenildiği zaman Doppler efektinin (Doppler effect) bir sonucu olarak ses perdesinde değişme olmaktadır. Bu sayade gerçekçi sesler duyulmaktadır. Ayrıca bulunulan ortam sesin yankılanmasını da etkilemektedir. Mağara içinde ses yankılanırken, açık havada ise yankılanmayacaktır. Unity oyun motorunda eklenen her sese, ses motorunun sunduğu bir çok efektten istenilenler eklenebilmektedir. Şekil 2.1'de de görüldüğü gibi, eğer çok fazla ses kaynağıyla çalışılıyorsa ses mikseri (audio mixer) kullanılarak bunlar gruplandırılmakta ve ses seviyeleri ve efektler rahatça ayarlanmaktadır.

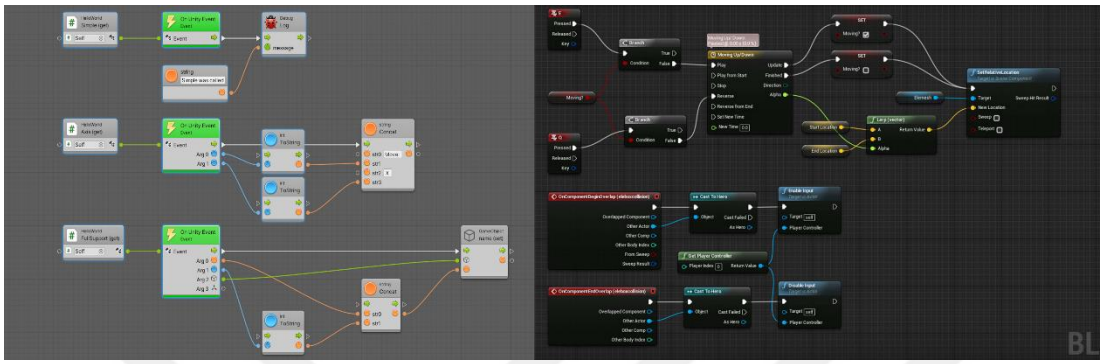


Şekil 2.1: Unity oyun motorunda ses mikseri.

2.1.4 Script yapısı

Script yapısı, motorun kullandığı script dilini veya varsa görsel script yapısını kullanarak, oyun döngüsünü (oyun mantığını), her türlü etkileşimi ve oyun motorunda bulunmayan araçları üretebilmeyi sağlayan yapıdır. Yazılan script'ler oyun nesnelerine bileşen olarak eklenmektedir. Bu sayede nesneler kullanıcıların tepkilerine karşılık verir duruma gelmekte ve ayrıca birbirleriyle de etkileşime girebilmektedirler. Unity oyun motoru script dili olarak C#, Unreal oyun motoru ise C++ programlama dillerini kullanmaktadır. Bunun yanında tek satır kod yazmadan karmaşık mantık

oluşturulmasına olanak tanıyan görsel script yapısını kullanan oyun motorları da bulunmaktadır. Şekil 2.2'de görüldüğü üzere, Unreal oyun motorunda BluePrint ve Unity oyun motorunda Bolt kullanılan görsel script sistemleridir. Görsel script sistemlerinde node yapısı bulunur ve bu node'lar birbirlerine bağlanarak oyun mantığını oluşturmaktadır.



Şekil 2.2: Unity'de Bolt (solda), Unreal'da BluePrint (sağda).

2.1.5 Animasyon sistemi

Buna ek olarak, oyun motorlarında animasyon sistemi bulunmakta ve motor içerisinde animasyon hazırlanmasına ve/veya başka uygulamalarla hazırlanan animasyonların motora aktarılıp çalıştırılmasına olanak tanımaktadır.

Sahnedeki her nesne statik değildir. Nesneler, fizik motoru veya animasyon ile hareket ettirilebilmektedir. Hemen hemen her uygulamada biraz animasyona ihtiyaç vardır. Oyun motorlarının çalıştırdığı animasyonlar farklı türlerde olabilmektedir. Bunlar:

- zaman içinde oyun nesnelerinin basit parametre değişimleri (konum, açı, renk, şeffaflık vb.),
- iskelet animasyonları (kemikler ile hareket eden karakterler) ve
- geçiş animasyonları (nesne mesh'lerinin içindeki noktaların konumlarını değiştirme) olabilir (Šmíd, 2017).

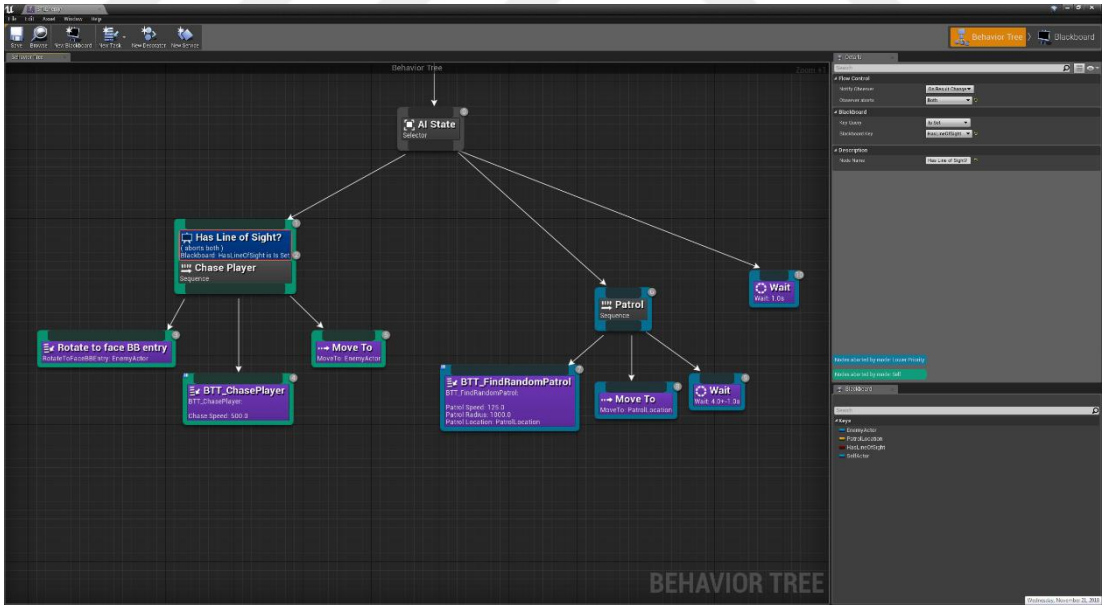
2.1.6 Yapay zeka (AI)

Kullanıcının kontrolü dışındaki karakterlerin/nesnelerin makul bir şekilde davranması için belirli bir miktarda zekaya ihtiyacı bulunmaktadır. Oyun motorlarında yapay zekanın ana görevlerinden birisi, A noktasından B noktasına bir yol bulmak ve bu yol boyunca karakterle yürümektir. Genellikle motor, sahnedeki statik nesnelere dayanan NavMesh adlı bir navigasyon yüzeyi oluşturmaktadır. Bu ağ, oyun sırasında

karakterlerin hareketlerine veya katı vücut (rigidbody) fiziğine göre dinamik olarak değiştirebilmektedir. Yapay zeka karakterleri bu ağ üzerinde yürüyebilmekte ve yolunu bulabilmektedir. Kullanılan ağ; örneğin yol, bataklık, su gibi farklı hareket maliyetlerine sahip alanlara ayrılabilmektedir. Yapay zeka bu maliyetleri göz önünde bulundurarak en uygun yolu, üzerinden geçtiği alana özel bir animasyon, ses efekti, partikül efekti ile seçebilmektedir.

Bunun yanında yapay zeka karakterleri; durum makinelerine (state machines), denetleyicilere (controllers) sahiptirler ve belirli kurallara göre hareket etmektedirler. Şekil 2.3'teki gibi, Unreal oyun motorunda davranış ağaçları (behavior tree) bulunmaktadır. Bu ağaçlar, karakterin hangi görevi yapması gerektiğine denetleyici ile karar verilmesini sağlayan sistemdir. Karakter belirli bir noktaya geldiği zaman, belirli bir görüş açısını sağladığında veya bambaşka bir koşul yerine geldiğinde yapay zekanın davranışı belirlenmektedir (Epic Games, 2018).

Oyun motorları yapay zekayı programlamazlar ve geliştirmezler. Bunun yerine geliştiricilere, yapay zekayı geliştirebilmek için ihtiyaç duydukları araç ve sistemleri sunmaktadırlar.



Şekil 2.3: Unreal oyun motorunda davranış ağacı (behavior tree) örneği.

2.2 Oyun Motorları

Piyasada çeşitli ihtiyaçlardan dolayı geliştirilmiş bir çok farklı oyun motoru bulunmaktadır. Oyun motoru geliştirmek zorlu bir teknik çalışma ve yoğun emek isteyen bir süreçtir. Yapılmak istenilen oyunun/uygulamanın geliştirme süresine bir de

oyun motorunun geliştirme süresinin eklenmesi çoğu zaman mantıklı değildir. Bu yüzden oyun motorunu kullanacak olan firmalar veya ekipler piyasada hali hazırda bulunan motorları tercih etmektedirler. Kimi oyun firması da yapacakları projelere özel oyun motorları geliştirmektedir. Şirket politikalarına göre, bu motorların şirket bünyesi dışında kullanılmasına izin verilmemektedir. Alternatif olarak, şirketler, oyun motorlarını pazarlayarak bir gelir elde edebilmektedir. Bunun yanında gelişmiş özelliklere sahip tamamiyle ücretsiz olan bir çok oyun motoru da bulunmaktadır. En popüler ücretsiz oyun motorlarının başında gelen Unity oyun motoru, bu tezdeki uygulamalarda da kullanılmıştır.

Sıradaki bölümde, Unity oyun motoru detaylı bir şekilde anlatılmış, bileşenleri açıklanmıştır. Unity dışındaki diğer diğer popüler oyun motorlarının bazılarından ise kısaca bahsedilmiştir.

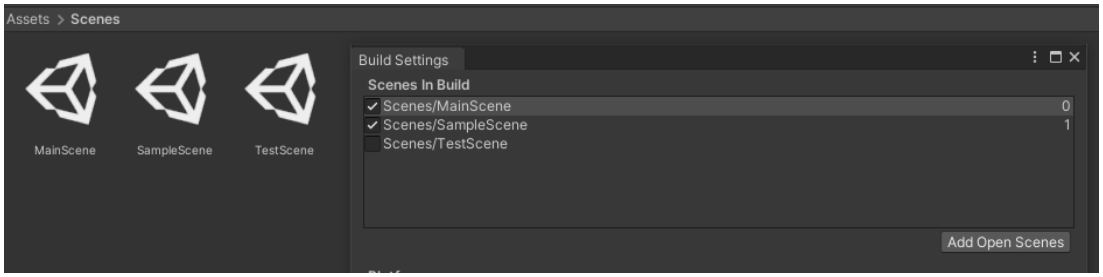
2.2.1 Unity oyun motoru

Unity, genellikle video oyunları gibi etkileşimli medya oluşturmak amacıyla kullanılan bir oyun motoru ve entegre geliştirme ortamıdır (Haas, 2014). Unity CEO'su David Helgason'ın dediği gibi, "Unity, oyun oluşturmak için kullanılan bir araç setidir ve grafikleri, sesi, fiziği, etkileşimleri ve ağları çalıştıran teknolojidir." (Brodkin, 2013). Unity, hızlı prototip oluşturma yetenekleri ve oluşturulan ürünlerin kolayca yayınlayabilmesiyle ünlüdür. Çapraz platform bir oyun motoru olduğu için, Unity'de yapılan bir uygulamanın farklı platformlar için çıktısı (Örneğin, WebGL, Mac, Android, iOS vb.) minimum eforla hazırlanabilmektedir. Unity'nin ücretsiz ve kolay kullanılabilir olması, öğrencilerin ve yeni geliştiricilerin ilgisini çekmektedir. Unity'nin Eylül 2019 verilerine göre, piyasadaki en iyi 1000 mobil oyunun %52'si ve Augmented Reality/Virtual Reality (AR/VR) içeriğinin %60'ı Unity sayesinde marketteki yerini almıştır (Dealessandri, 2020).

2.2.1.1 Sahne yapısı ve nesneler

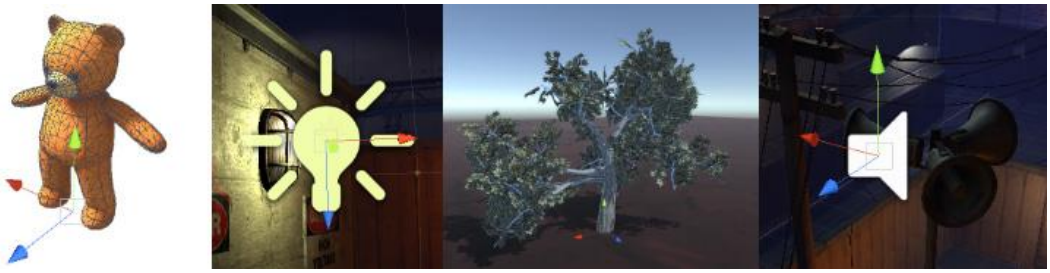
Unity'de sahne (scene), içerikle çalışılan yerdir. Bir oyunun veya uygulamanın tamamını veya bir kısmını içeren varlıklardır. Örneğin tek bir sahnede basit bir oyun oluşturabilirken, daha karmaşık bir oyun için her biri kendi ortamları, karakterleri, engelleri, dekorasyonları ve kullanıcı ara yüzleri olan seviye başına bir sahne kullanılabilmektedir (Şekil 2.4). Bir projede istenilen miktarda sahne oluşturulabilmektedir. Sahnelerin içinde oyun nesneleri bulunmaktadır (kamera, ışık,

3 boyutlu nesneler vb.). Sahneler arası geçişler manuel veya kod ile otomatik sağlanabilmektedir.



Şekil 2.4: Unity oyun motorunda sahneler.

Her bir Unity projesi en az bir sahneden oluşmak zorundadır. Sahnelerin içinde yer alan nesnelere oyun nesneleri denir. Bu nesneler farklı bileşenler (transform, rigidbody, material vb.) barındırmakta ve geliştirici tarafından değiştirilebilmektedirler. Oyun içerisinde küp, küre, silindir, kapsül, yüzey gibi temel oyun nesneleri bulunmaktadır. Şekil 2.5'teki gibi, ihtiyaç duyulduğunda daha kompleks 3 boyutlu modeller projeye dışarıdan eklenebilmektedir. Oyundan görüntü alabilmek için kamera nesnesi kullanılmaktadır. Bunun dışında oyuna ses eklenmek istendiğinde uygun oyun nesnesine "Audio Source" bileşeni veya ışık eklenmek istendiğinde "Light" bileşeni eklenmektedir. Görüldüğü üzere oyun nesnelerine farklı bileşenler eklenip çıkartılarak ihtiyaçlar giderilmektedir.

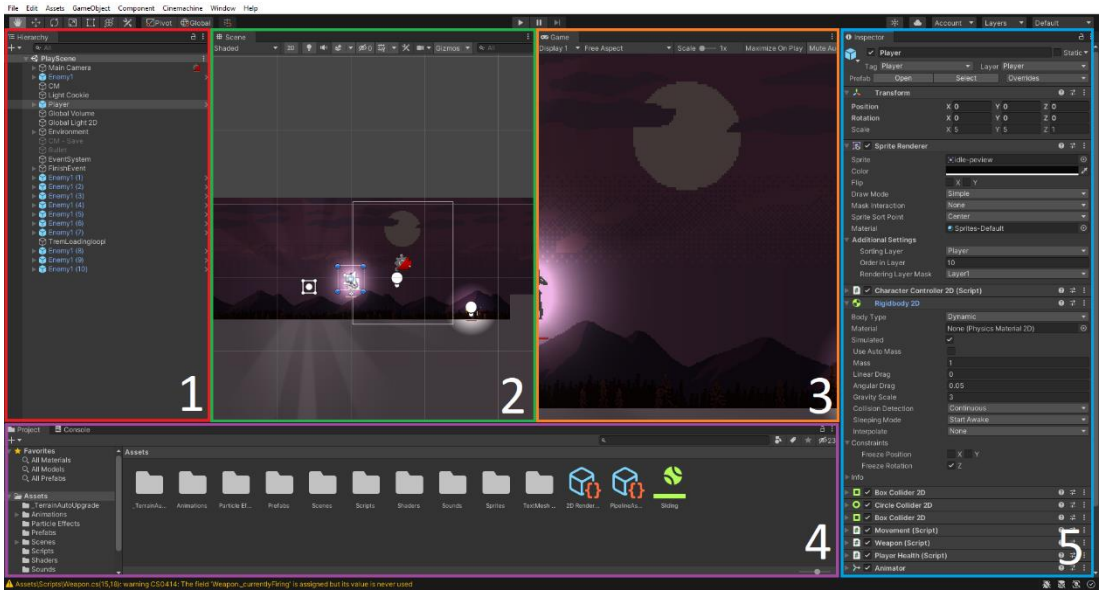


Şekil 2.5: Unity oyun motorunda oyun nesneleri.

Unity'deki tüm oyun nesneleri 3 boyutlu koordinat sistemi üzerindedir. Nesnelerin, x, y ve z eksenleri üzerinde bir konumu bulunmaktadır. Bu koordinat değerleri değiştirilerek nesnelerin konumları değiştirilebilmektedir. X ekseni sağ-sol yönünde hareketi, y ekseni yukarı-aşağı yönünde hareketi ve z ekseni ileri-geri yönde hareketi temsil etmektedir. Şekil 2.5'teki mavi renkli ok z eksenini, kırmızı renkli ok x eksenini ve yeşil renkli ok y eksenini göstermektedir. Bir oyun nesnesi her zaman bir "Transform" bileşenine (koordinat bilgilerinin saklandığı bileşen) sahiptir ve bunu kaldırmak mümkün değildir.

2.2.1.2 Unity editörü

Bu bölümde Unity'nin ara yüzü, kullanılan panellerin işlevleri ve bazı araçların görevleri tanıtılmıştır. Şekil 2.6'da görüldüğü gibi Unity toplam 5 ana panelden oluşmaktadır. Bunlar istenildiği zaman kapatılıp açılabilmektedir. Şekildeki 1 numaralı alan Hierarchy paneli, 2 numaralı alan Scene paneli, 3 numaralı alan Game paneli, 4 numaralı alan Project paneli ve 5 numaralı alan Inspector panelidir. Bunların dışında zaman zaman kullanılan ve isteğe bağlı açılıp kapatılan ve bu projede kullanılan; Console paneli, Profiler paneli ve Entity Debugger paneli de bulunmaktadır.



Şekil 2.6: Unity oyun motorunun arayüzü.

- Hierarchy paneli, geçerli sahnedeki tüm oyun nesnelerinin bir listesini içermektedir. Bu liste, sahneye bir nesne getirildiği zaman otomatik güncellenmektedir. Bir nesne diğerinin altına sürüklenip bırakılırsa aralarında ebeveyn, çocuk ilişkisi kurulmaktadır.
- Scene paneli, oyunun oluşturulduğu yerdir. 2 veya 3 boyutlu nesneler burada istenildiği gibi ayarlanmaktadır.
- Game panelinde, Scene panelinde hazırlanan oyun, çalıştırıldığında oluşan ön izlemesi gösterilmektedir. Projenin derlenmesini ve hedef platformlara dağıtılmasını beklemek zorunda kalmadan değişikliklerin test edilmesine izin vermektedir. Buraya yansıyan görüntü, kamera oyun nesnesinden gelen görüntüdür. Kamera olmazsa bu panelden görüntü alınamamaktadır.

- Project paneli, Unity'e aktarılan ve kullanıma hazır tüm varlıkları (görseller, kodlar, ses dosyaları vb.) içeren penceredir. Klasör yapısı bulunmakta ve organize edilebilmektedir.
- Inspector paneli, her bir oyun nesnesinin ayrıntılarının görüntülendiği ve değiştirildiği yerdir. Geliştirici, bu panelden yararlanarak oyun nesnelerinin parametrelerini arzu ettiği şekle getirebilmektedir.
- Console paneli, uygulama çalışırken meydana gelen hataların, uyarıların ve genel mesajların gösterildiği bir paneldir. Ayrıca kod kullanarak istenilen parametreler ve mesajlar da yazdırılabilmektedir.
- Profiler paneli, çalışan uygulama hakkında teknik performans bilgilerinin alındığı paneldir. Profiler; CPU, bellek, render ve ses gibi alanlarda ilgili bilgileri hem sayısal hem de görsel olarak sunmaktadır. İşlem süresi, bellek kullanımını hesaplamakta ve bu bilgiler sayesinde geliştirici, uygulamanın optimize olmayan kısımları gözden geçirebilmektedir.
- Entity Debugger paneli, varlıkların, bileşenlerin ve sistemlerin (ECS) görüntülenmesini sağlayan paneldir.

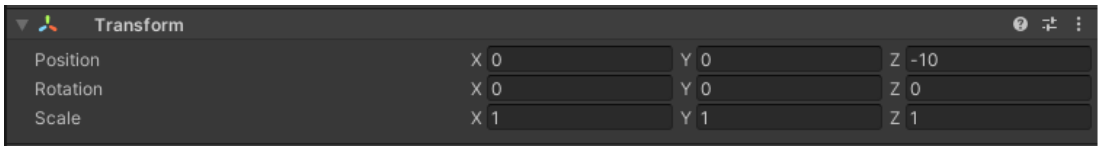
2.2.1.3 Bileşenler

Bileşenler, oyun nesnesine eklenip çıkartılabilen ve eklendiği nesneye belirli özellikler katan birimlerdir. Inspector panelinde bulunmaktadırlar.

Transform

Transform bileşeni sahnedeki nesnenin konumunu, rotasyonunu ve ölçeğini belirlemekte ve değiştirilmesini sağlamaktadır. Buradaki değerler değiştirildiği an sahnedeki nesnenin konumu, açısı, ölçeği anlık olarak değişmektedir. Oyun nesnesini hareket ettirmenin yollarından birisi transform bileşenindeki konum (position) değerini, yazılan script ile değiştirmektir. Şekil 2.7'de gösterilen, transform bileşenindeki x, y veya z ölçeklerinden herhangi birisinin değeri 0 olarak girilir ve diğerleri de 0'dan büyük olursa oyun nesnesi 2 boyutlu olmaktadır.

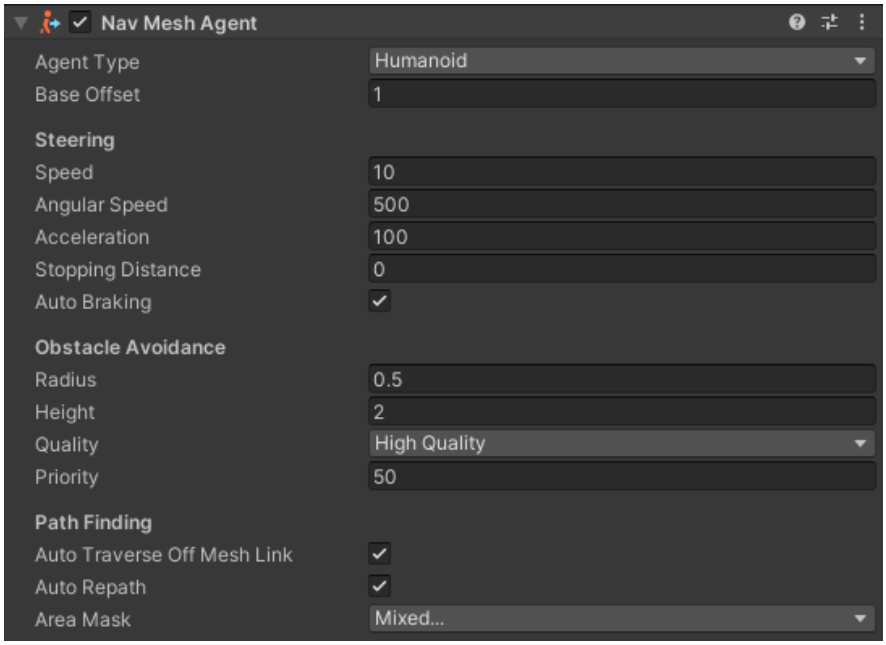
Bu bileşendeki değerler metre ve derece cinsindendir. Transform bileşeni her bir oyun nesnesinde bulunmak zorundadır; diğer bileşenler ise isteğe göre kullanılabilmektedir.



Şekil 2.7: Transform bileşeni.

NavMesh Agent

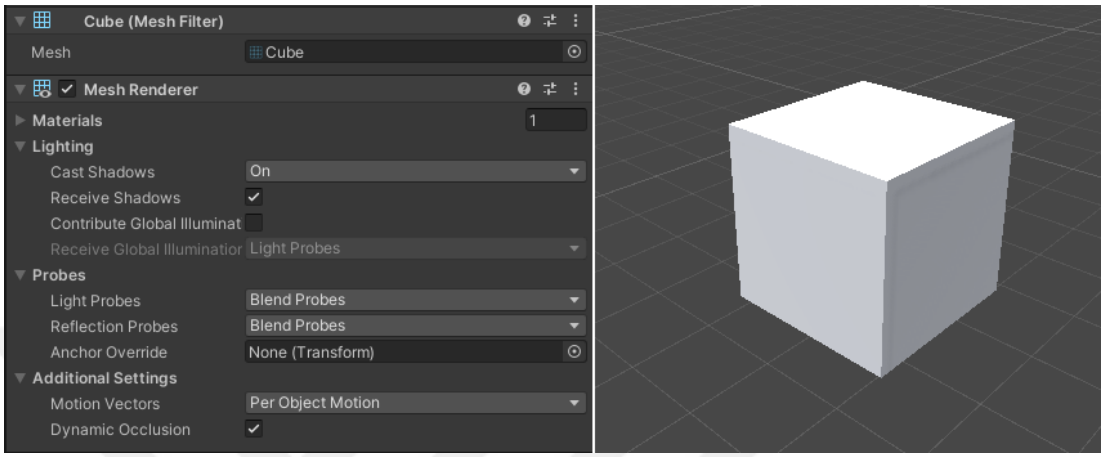
Güzergah belirlemede kullanılan en temel bileşendir. Bir oyun nesnesinin yapay zeka karakteri (ajanı) haline gelip otonom hareket edebilmesi ve engellerden kaçınabilmesi için NavMesh Agent bileşenine ihtiyacı bulunmaktadır. Belirlenen navigasyon ağı üzerinde, bu ajanlar otonom hareket edebilmektedirler. Şekilde 2.8'deki NavMesh Agent bileşeni, ajana çizgisel hız, açısal hız, ivme, duruş mesafesi, otomatik durma gibi parametreler sunmaktadır. Çizgisel hız, açısal hız ve ivmelenme parametreleriyle ajanın hızı ayarlanmaktadır. Birimleri metre ve saniye cinsindendir. Duruş mesafesi, ajanın hedeften kaç metre önce durması gerektiğini ayarlamaktadır. Örneğin bu değer 0 girilirse hedefin tam üstünde, 5 girilirse hedeften 5 metre önce durmaktadır. Otomatik durma, ajan hedefe ulaştığında hedefi geçip gitmesini önleyerek tam olarak hedef noktası üzerinde durmasını sağlamaktadır. Bunun yanında ajanın boyu ve çapı metre cinsinden girilmekte ve ajan çevredeki engellerle bu fiziksel özelliklerine göre etkileşime geçmektedir. Şekil 2.8'deki "auto repath" özelliği, eğer ajan hedefe giderken kullanmakta olduğu rota geçersiz olursa, hemen yeni rota bulup hareketine devam etmesini sağlamaktadır.



Şekil 2.8: NavMesh Agent bileşeni.

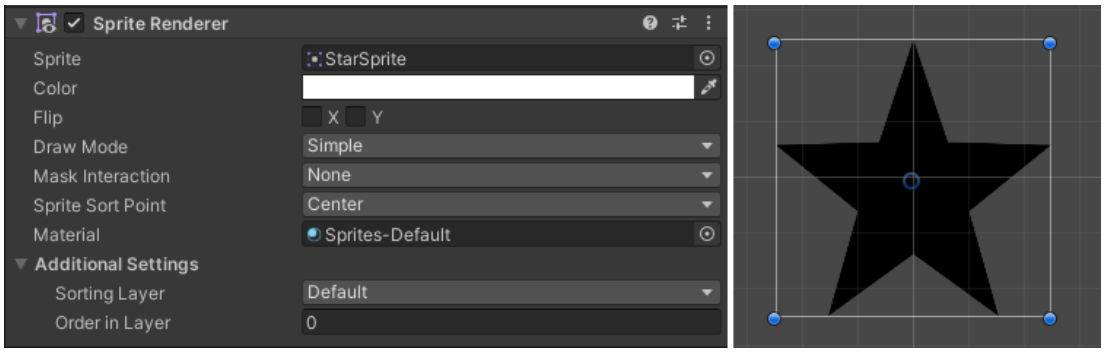
Renderer

Renderer bileşeni, nesnenin geometrisini almakta ve transform bileşenindeki konumunda işleyip görselleştirmektedir. Bu sayede sahnede, bu nesne 2 veya 3 boyutlu olarak görüntülenmektedir. 3 boyutlu nesneler için "mesh renderer" (Şekil 2.9), 2 boyutlu nesneler için "sprite renderer" (Şekil 2.10) bileşeni kullanılmaktadır.



Şekil 2.9: Küp nesnesinin Mesh Renderer bileşeni.

Sprite renderer ile projeye eklenen görüntü dosyaları sahnede görselleştirilmektedir. Eğer görüntüde transparanlık özelliği varsa sahnedeki görüntüsünde de aynı şekilde transparanlık görülmektedir.



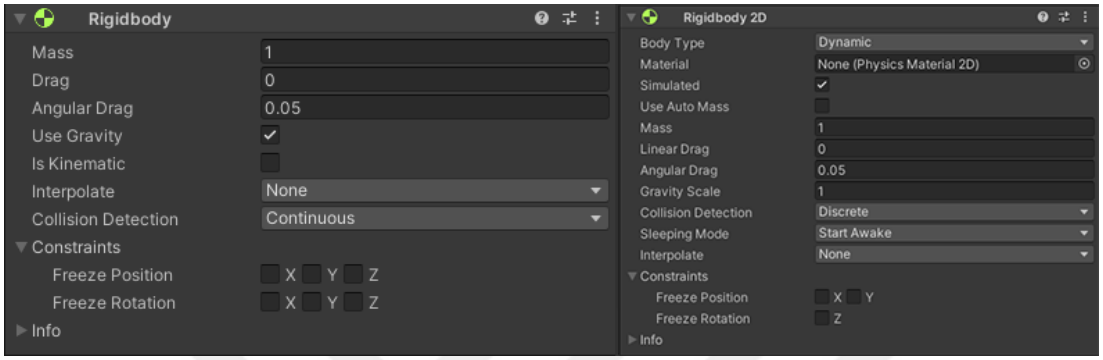
Şekil 2.10: Sprite Renderer bileşeni.

Rigidbody

Rigidbody; oyun nesnelerinin fiziksel hareketlerini gerçek dünyada olduğu gibi simüle etmeyi sağlamaktadır. Örneğin, yerçekimi yüzünden nesnelerin aşağı düşmesi; eğimli yüzeyde küresel nesnelerin yuvarlanırken köşeli nesnelerin sabit kalması gibi hareketler düşünülebilir. Bunların yanında nesneleri gerçekçi bir şekilde hareket ettirmek için kuvvet ve tork uygulayabilmektedir. Herhangi bir oyun nesnesinin yerçekiminden etkilenmesi, yazılan script'ler ile kuvvet uygulanması veya diğer

nesneler ile etkileşime geçmesi (örneğin çarpışma) için rigidbody bileşeni içermesi gerekmetidir.

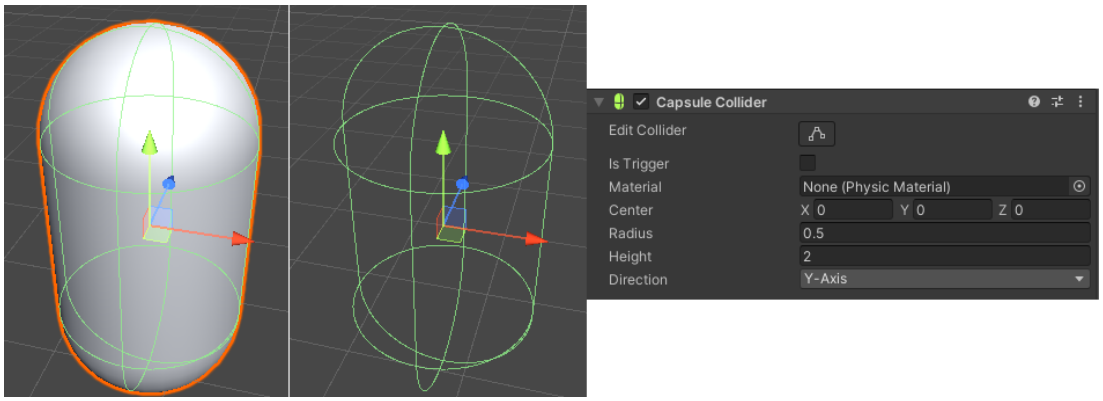
Şekil 2.11'de görüldüğü gibi, Unity'de 2 ve 3 boyutlu nesneler için farklı rigidbody bileşenleri bulunmaktadır. Buradan nesnenin kilogram biriminde kütlesi, sürtünmesi, açısal sürtünmesi, yer çekimi kullanıp kullanmaması, istenilen eksenlerde dönüşünün kilitlenmesi gibi bir çok özellik kontrol edilebilmektedir.



Şekil 2.11: Rigidbody (solda) ve Rigidbody 2D (sağda) bileşeni.

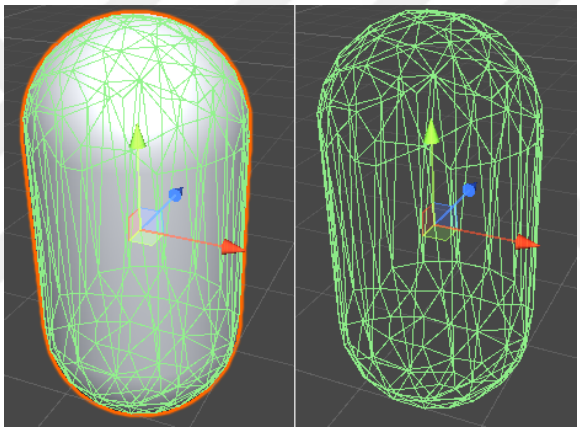
Collider

Collider, oyun nesnelerin fiziksel olarak birbiriyle etkileşime geçebilmelerini sağlayan bileşendir. Eğer oyun nesnelerinde collider yoksa birbirleri içinden geçmekte ve temas etmemektedirler. Bir zemin üzerinde oyun nesnelerinin durabilmesi için hem zeminde hem de üstündeki nesnelerde collider olmalıdır. Şekil 2.12'de görüldüğü üzere, collider'ın sınırı yeşil çizgilerle gösterilmekte ve nesnenin mesh'i kapatılsa bile nesne fiziksel etkileşimlere devam edip çevresini etkileyebilmektedir. Nesnenin mesh'iyle collider'ının boyutlarının aynı olması zorunlu değildir. Collider sınırları 3 boyutlu olarak ölçeklendirilebilmektedir.



Şekil 2.12: Kapsül collider bileşeni ve oyun nesnesi üzerinde sınırları.

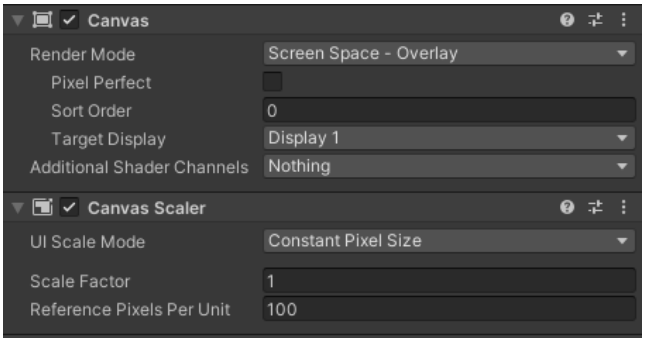
Eğer büyük bir projede çalışılıyorsa collider'ların optimizasyonu önem kazanmaktadır. Şekil 2.12'de görünen kapsül collider gibi; Unity, basit nesneler için (küp, küre, kapsül vb.) fiziksel hesaplamaları kolaylaştırmak adına temel collider'lar sunmaktadır. Kullanılan nesneler insan modeli, araç, ağaç gibi daha kompleks yapıdaysa "mesh collider" denilen ve nesnenin her poligonunu tek tek kaplayan collider kullanılmaktadır (Şekil 2.13). Ancak bu mesh collider'ların sayısı çok fazlaysa fiziksel hesaplamalarda performans kaybına sebep olmaktadırlar. Bu yüzden oyun motorlarında genellikle mesh collider kullanımından kaçınılmaktadır. Bunun yerine 3 boyutlu bir insan modelininde kapsül collider, bir araba veya bina modelinde küp collider kullanılması daha doğru olabilmektedir. Eğer bu kompleks modelleri kaplamaya tek bir temel collider yeterli olmuyorsa, oyun nesnesinde birden fazla collider kullanılarak istenilen çarpışma sınırları elde edilebilmektedir.



Şekil 2.13: Mesh collider örneği.

Canvas

Canvas, sahnedeki tüm arayüz elemanlarının (yazı, 2 boyutlu şekil vb.) içinde bulunduğu alandır ve bunlar Canvas'ın alt nesnesi olarak çizilmektedir (Şekil 2.14). Ayrıca Canvas, farklı ekran çözünürlüklerine göre arayüz elemanlarının doğru konumlarda çizilmesini sağlamaktadır.



Şekil 2.14: Canvas bileşeni.

Image

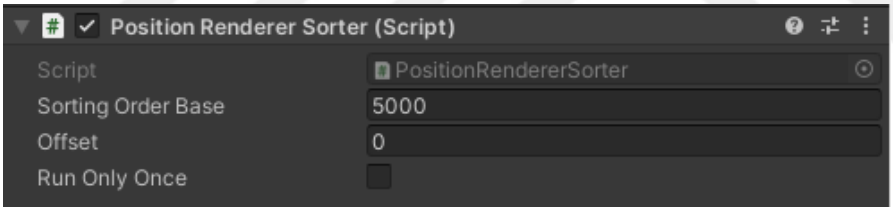
Farklı formatlardaki görüntü dosyalarının sahnede gözükmesini sağlamaktadır.

Text

Sahnede gösterilmek istenilen yazı ve sayıların; font, renk ve büyüklük gibi ayarlarının yapılmasını sağlamaktadır.

Script

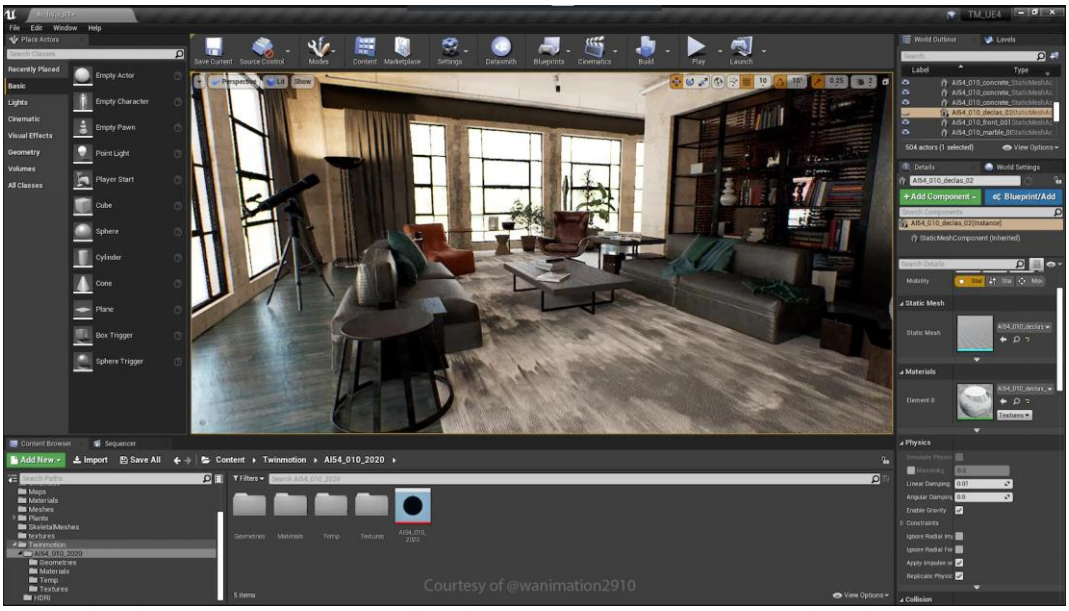
Unity tarafından hazır olarak sunulan bileşenler yeterli gelmediği durumda; C# programlama dili kullanılarak, belirli kurallar çerçevesinde kişisel bileşenler oluşturulabilmektedir. Unity'de, MonoBehaviour denilen ana C# sınıfı bulunmaktadır ve tüm Unity Scriptleri otomatik olarak bu sınıfı miras almaktadır. MonoBehaviour ile hazır gelen belirli fonksiyonlar bulunmaktadır. Bunlar ile oyun başlatıldığında, nesne oluşturulduğunda veya yok edildiğinde, nesneler arası çarpışma yaşandığında veya ekrana saniye başına çizilen her kare için fonksiyonlar çalıştırılabilmektedir. Şekil 2.15'teki gibi, ihtiyaç olduğu durumda, Script bileşeninde belirlenen parametreler, Inspector panelinde gösterilmekte, oyun sırasında hızlıca farklı durumlar için değiştirilmesi sağlanmaktadır.



Şekil 2.15: Script bileşeni örneği.

2.2.2 Unreal oyun motoru

Unreal oyun motoru, Epic Games şirketi tarafından geliştirilen ve 1998 yılında yayınlanan bir motordur (Şekil 2.16). Yeni başlayanlar için öğrenmesi diğer motorlara kıyasla daha zordur. Unreal oyun motoru büyük projeler ve ekipler için uygun bir motordur. Mobil platformları desteklemesin rağmen, küçük mobil oyunlar geliştirmek için pek de uygun değildir. Bilgisayar ve mobil platformları dışında, konsol ve sanal gerçeklik platformlarını da desteklemektedir. Motor C++ programlama diliyle yazılmıştır ve script dili olarak da bunu kullanmaktadır. Sunduğu görsel kalite diğer oyun motorlarına kıyasla çok daha iyidir. Kullanımı ücretsizdir ve piyasadaki en gelişmiş oyun motorlarından birisidir.

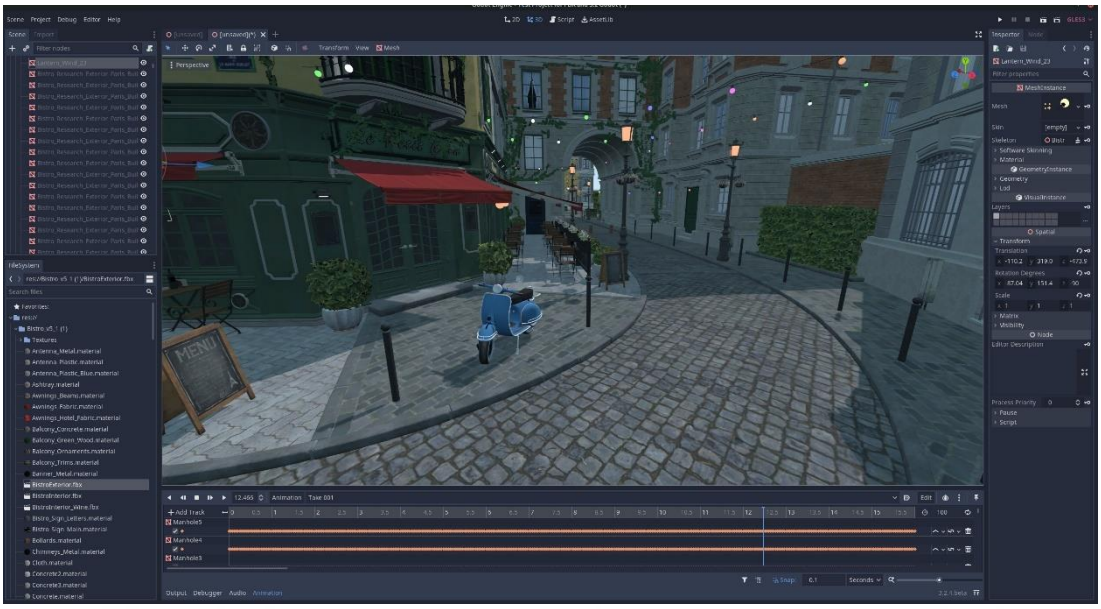


Şekil 2.16: Unreal oyun motoru.

2.2.3 Godot oyun motoru

Godot oyun motoru, çoklu platform desteğine sahip olan, tamamen ücretsiz ve açık kaynak kodlu bir oyun motorudur (Şekil 2.17). Unreal ve Unity'den sonra kullanıcıların en çok tercih ettiği motorlardan birisidir. 2 ve 3 boyutlu oyunlar oluşturmak için tasarlanmış olup; PC, mobil ve web platformlarını desteklemektedir. Açık kaynak kodlu olduğu için kullanıcılar tarafından rahatlıkla özel araçlar geliştirilebilmekte veya direkt motora müdahale edilebilmektedir. 2 boyutlu uygulamalarda rakiplerine kıyasla çok başarılıdır. Ayrıca çok hafif bir motordur, düşük sistemli bilgisayarlarda rahatlıkla çalışabilmektedir.

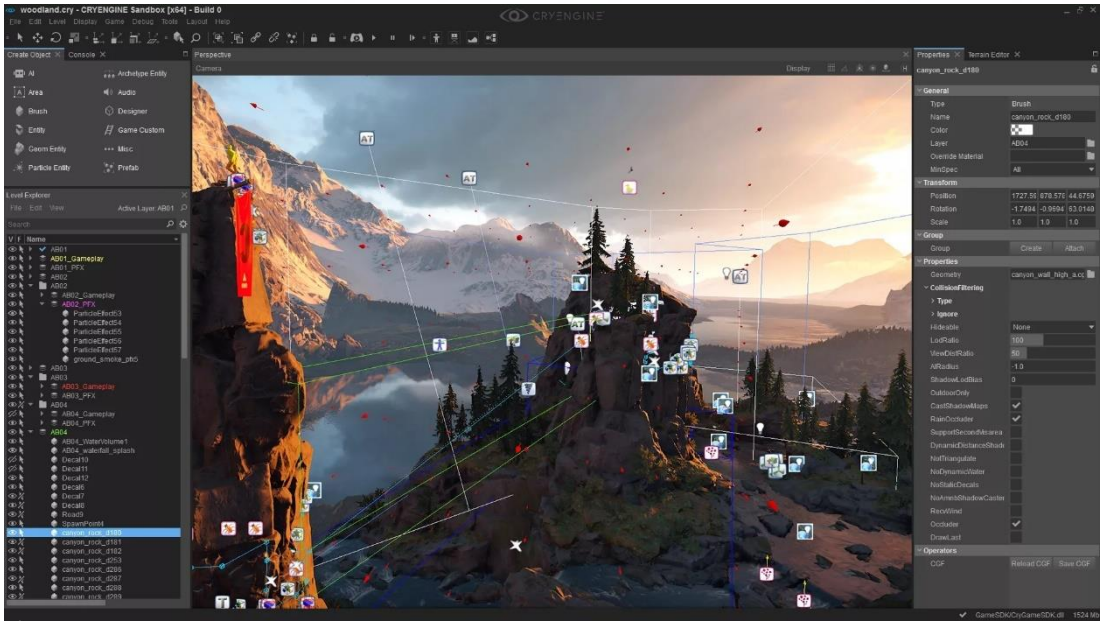
Script dili olarak Godot'a özel olarak geliştirilen GDScript dilini kullanmaktadır. Bu dilin yapısı çok sade olup öğrenmesi kolaydır. Bunların yanında C++ ve C#'da kullanılabilmektedir.



Şekil 2.17: Godot oyun motoru.

2.2.4 CryEngine oyun motoru

CryEngine, CryTek firması tarafından geliştirilen ve 2002 yılında piyasa sürülmüş bir oyun motorudur (Şekil 2.18). Görsel olarak başarılı bir 3 boyut sunan motor, PC, konsol ve sanal gerçeklik platformlarını desteklemektedir. Kullanımı ücretsizdir. Script dili olarak LUA'yı kullanmaktadır.



Şekil 2.18: CryEngine oyun motoru.

3. OYUN MOTORLARINDA GÜZERGAH BELİRLEME

Güzergah belirleme, belirli bir haritada belirtilen başlangıç ve hedef düğümleri arasında en uygun yolu hesaplamayı ifade eder. Güzergah belirleme uygulamalarında kullanılan Dijkstra, A* gibi pek çok farklı algoritma bulunmaktadır (Sidhu, 2020). Oyun motorlarında genellikle yapay zeka tarafından kontrol edilen karakterler için navigasyon uygulamaları yapılmaktadır. İlerleyen bölümlerde NavMesh ve Unity oyun motorunda kullanımı açıklanacaktır.

3.1 Yapay Zeka Navigasyonu

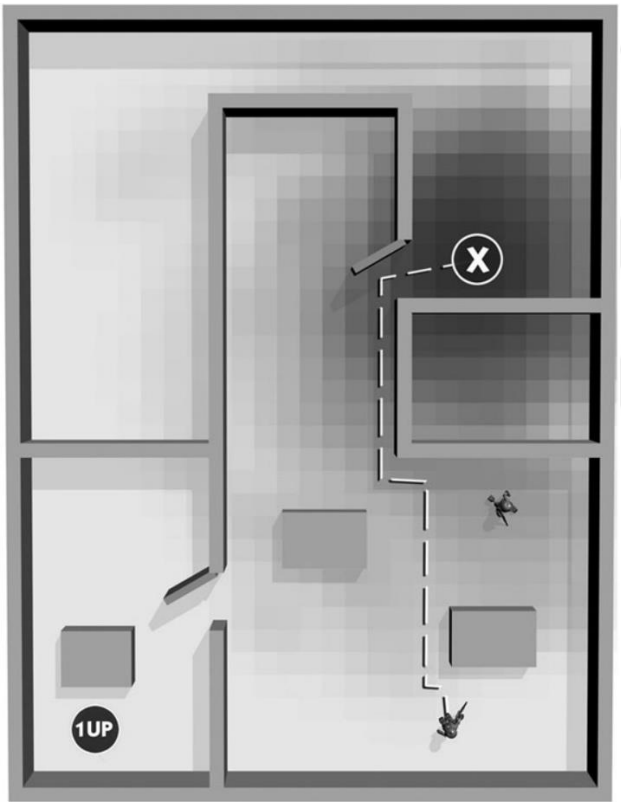
Yapay zeka navigasyonu, A noktasından B noktasına gitme sanatıdır. Daha gerçekçi simülasyonlar arayışındaki modern oyunların/uygulamaların dünyaları genellikle çeşitli araziler, engeller ve hareketli nesneler ile doludur. Navigasyon sırasında bunun gibi sorunları çözmek için gelişmiş AI algoritmalarına sahip olmak gerekmektedir. Navigasyon tipik olarak iki ana göreve ayrılır; güzergah belirleme ve engellerden kaçınma.

3.1.1 Güzergah belirleme (path finding)

Güzergah belirleme, ilginç ve karmaşık bir sorundur. Piyasaya çıkan ilk oyunlarda güzergah belirleme neredeyse yok gibidir. Ortamlar basit veya tamamen açık olduğundan dolayı, yapay zeka doğrudan kullanıcı karakterinin konumuna yönelirdi veya tamamen rastgele dolaşırlardı. Oyunların daha gerçekçi dünyalara sahip olmasıyla birlikte tüm bunlar değişim göstermiştir.

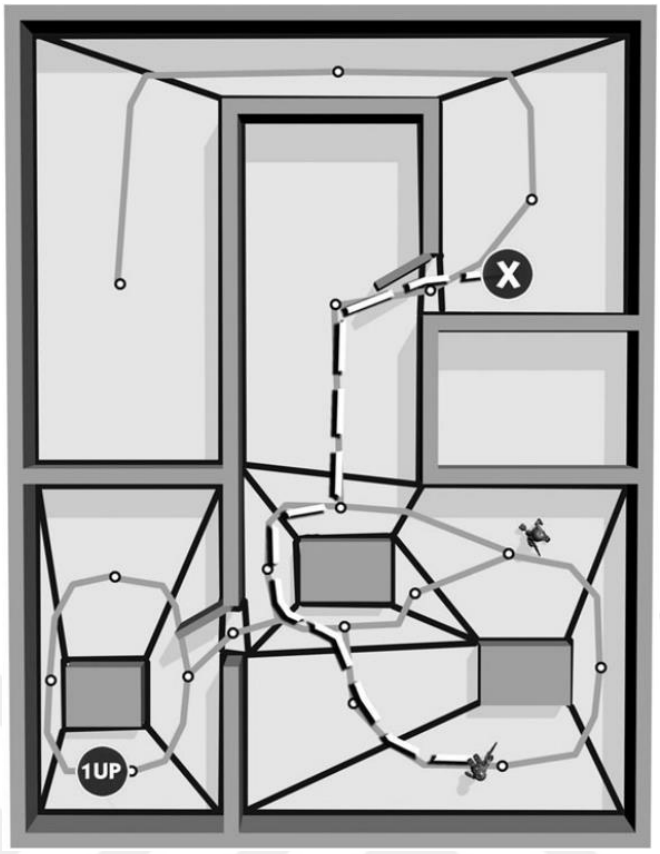
Bir yapay zeka karakterinin, dünyadaki A noktasından B noktasına akıllıca hareket etmesi için, yolu bulmasına yardımcı olacak özel bir yönteme gereksinim duymaktadır. Bu yöntemler arasında: grid tabanlı yöntem (gridbase), navigasyon ağları (navigation meshes veya NavMesh), basit kaçınma ve potansiyel alanlar, harita düğüm ağları (map-node networks) ve kombinasyon sistemleri örnek olarak gösterilebilir (Schwab, 2004). Bu tezdeki uygulamalarda da kullanılan ızgara tabanlı yöntem ve navigasyon ağları aşağıdaki gibi açıklanabilir.

Grid tabanlı yöntemde, dünya genellikle kare veya altıgen olmak üzere eşit gridlere bölünür ve gridleri kullanarak en kısa yolu bulmak için A* güzergah belirleme algoritması veya benzer yöntemler kullanılır (Şekil 3.1). Her grid, genellikle 0'dan (hiç geçemez) 1'e (seyehat için tamamen açık) bir geçiş olasılığı değerine sahiptir. Basit sistemler, grid için yalnızca ikili değerleri kullanabilir. Daha karmaşık sistemler ise gridte ara değerleri de kullanır. Bu yöntemin olumsuz yanı, gridin bellek boyutu ve sistem en kısa yolu bulduğunda geçici verilerin depolanmasıdır. Yüksek çözünürlüklü gridler, arama algoritmasının yapması gereken iş miktarı önemli ölçüde artacağı için maliyet açısından çok engelleyici hale gelebilir. Bu tezde yapılan uygulamada grid tabanlı yöntem ile A* algoritması birlikte kullanılmıştır.



Şekil 3.1: Izgara tabanlı yöntem ile güzergah belirleme örneği (Schwab, 2004).

Navigasyon ağları yöntemi (NavMesh), haritayı/araziyi oluşturmak için kullanılan yüzey poligonları kullanarak, yapay zekanın kullanabileceği bir yol düğüm ağını algoritmik olarak oluşturur (Şekil 3.2). Bu çok daha güçlü bir sistemdir, ancak eğer kullanılan bu yönteminin kendisi oldukça akıllı değilse veya sahneler bu işlemin gerçekleştirileceği bilgisi ile oluşturulmadıysa bazı garip görünen yollara neden olabilir.



Şekil 3.2: Navigasyon ağları yöntemi ile güzergah belirleme örneği (Schwab, 2004).

3.1.2 Engellerden kaçınma

Dinamik olarak engellerden kaçınma ise çok daha basit bir navigasyon görevidir. Bir karakterin rotasında ilerlerken güzergahındaki nesnelerin etrafından dolaşmayı içerir. Güzergah belirleme sistemi, oyuncuya hedef konumuna ulaşmak için bir güzergah bulmuştur, ancak yoluna beklenmedik bir nesne çıktığı anda güzergahını tekrar ayarlaması gerekmektedir.

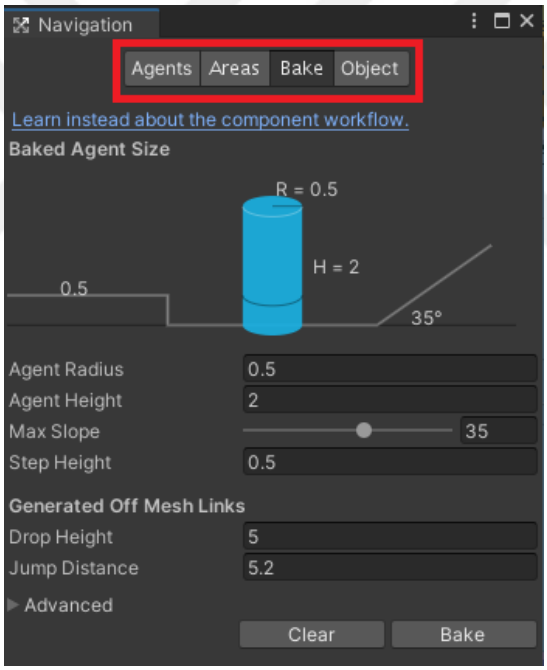
3.2 Unity Oyun Motorunda NavMesh

NavMesh, Unity'e dahil olan bir güzergah belirleme aracıdır. Bu araç, sahne geometrisinde seçilen yüzeyleri işler ve buna göre otomatik olarak bir navigasyon ağı oluşturur. Yapay zeka karakterleri (ajanlar), bu ağ üzerinde akıllıca hareket edebilir. Hem oluşturulan navigasyon ağı için, hem de ajanlar için farklı parametreler girilerek sistemin istenildiği gibi çalışması sağlanır. Uygulama çalışırken, ajanın önüne çıkan dinamik engeller ajanın rotasını anında tekrar ayarlamasına neden olur.

Şekil 3.3'te görüldüğü üzere Unity'nin navigasyon panelinde "agents," "areas", "bake" ve "object" olmak üzere 4 temel araç vardır. "Agents" aracında, farklı türdeki

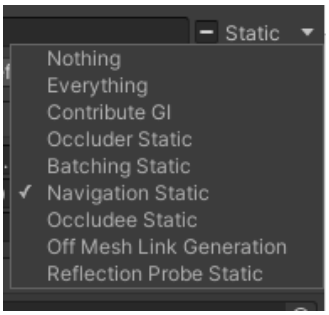
ajanlar tanımlanabilir. Örneğin bir insan modeli ve araba modeli için 2 farklı profil oluşturulur ve bu sayede ajanları kullanırken hepsinin aynı davranışı sergilemesi yerine daha gerçekçi ve özelleşmiş hareketler gözlemlenir.

"Bake" aracıyla navigasyon ağı oluşturulurken, ajanların tırmanabileceği maksimum açı, adım yüksekliği, ajanların çapı ve boyu gibi parametreler Şekil 3.3'te gösterildiği gibi girilir. Eğer belirlenen açıdan daha dik bir yokuş varsa, adım yüksekliğini aşan bir yükselti varsa veya ajanın boyu bir nesnenin altından geçmek için fazla uzunsa ajan buralardan geçemez. Aynı zamanda ajanın çapı değiştirilerek nesnelerin ne kadar yanından geçeceği de ayarlanabilmektedir. Bunların yanında ajanın, iki yüzey arasındaki en fazla kaç metrelik boşluktan zıplayabileceğini ayarlayan "jump distance" ve yüksek bir yüzeyden alçak bir yüzeye düşmek için gereken maksimum uzunluğu gösteren "drop height" değerleri de vardır.



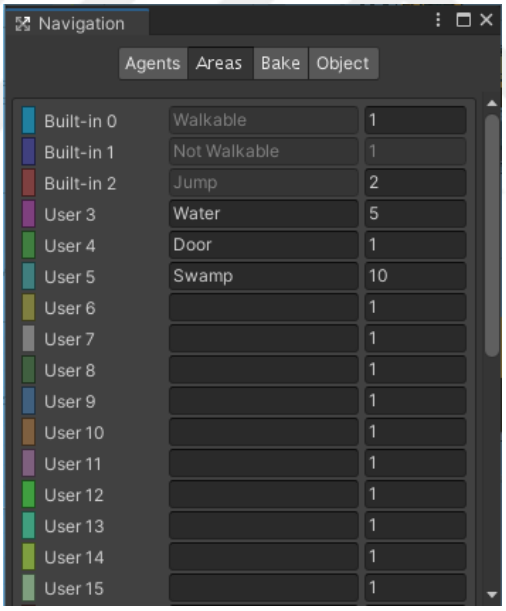
Şekil 3.3: Unity'de navigasyon paneli ve NavMesh parametreleri.

Bir nesnenin veya yüzeyin, navigasyon ağına dahil edilip hesaplamalarda kullanılabilmesi için nesne özelliklerinden Şekil 3.4'te gösterildiği gibi "navigation static" özelliğinin açılması gerekmektedir. Unity, navigasyon ağını oluştururken sadece "navigation static" özelliği olan nesneleri kullanır.



Şekil 3.4: Nesnenin "navigation static" özelliğinin açılması.

"Areas" aracı ile hazırlanacak olan navigasyon ağında farklı bölgeler oluşturmak Şekil 3.5'te görüldüğü üzere mümkündür. Navigasyon aracında kullanıma hazır yürünebilir, yürünemez ve zıplanabilir türünde alanlar bulunmaktadır. Bunun yanında geliştirici, gereksinim duyduğu zaman yeni bir alan tanımlayabilir. Ayrıca tüm alanların maliyet değeri de bulunmaktadır. Ajanlar gidecekleri yolu bu maliyet değerlerini hesaba katarak belirlerler. Maliyet değeri ne kadar yüksekse ajanın o bölgeden geçme olasılığı o kadar azalır. Alanlar ile hesaplama yapılırken A* algoritması kullanılmaktadır.



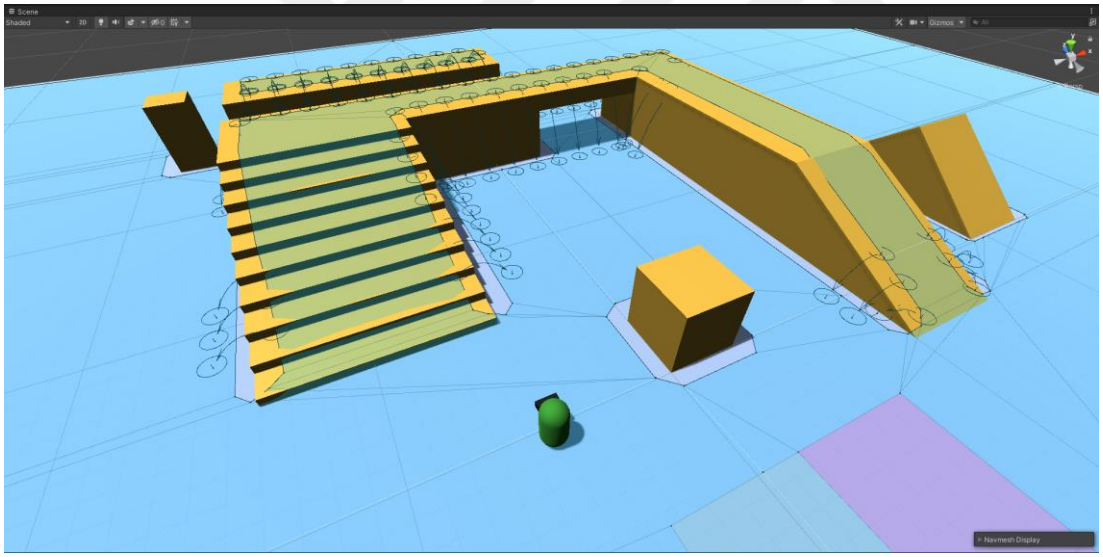
Şekil 3.5: Unity'de navigasyon panelinde "areas" aracı.

3.2.1 NavMesh uygulaması

Şekil 3.6'da görüldüğü üzere, Unity'de NavMesh kullanılarak örnek bir çalışma yapılmıştır. Bu çalışmada, önce ilgili alanlar (zemin ve turuncu platformlar) "navigation static" yapılıp gerekli parametreler girildikten sonra, navigasyon ağı oluşturulmuştur. Zemini ve platformları kaplayan koyu mavi renkli alan, navigasyon ağ alanının sınırlarını göstermektedir. Ajan bu sınırlar içinde dolaşabilmektedir. Ajanın çapı ve boyu ayarlanarak navigasyon yapılabilecek bu alan

değiştirilebilmektedir. Örneğin, ajan şu an platformda bulunan alt geçitten geçebilmektedir. Eğer ajanın boyu uzatılırsa, NavMesh sistemi, ajanın o bölgeden geçemeyeceğini fark edip alt geçitin bulunduğu bölgeyi navigasyon ağına katmayacaktır.

Yeşil renkli karaktere "NavMesh Agent" bileşeni eklenerek yapay zeka karakteri haline getirilmiştir. Şekil 3.6'da görülen iki adet farklı açılı rampadan sağdaki rampanın açısı ajanın tırmanabileceği en büyük açıdan fazla olacak şekilde ayarlandığından ajan daha düşük açılı soldaki rampadan çıkabilmektedir. Bunun yanında platform üzerinden zemine doğru giden siyah oklar bulunmaktadır. Bu oklar, platformdaki o noktalardan, ajanın aşağıya düşebileceğini göstermektedir. "Drop height" değeri değiştirilerek en fazla kaç metreden düşülebileceği ayarlanabilmektedir. Buna benzer bir başka durum ise aralarında boşluk olan 2 platform arasında da siyah çizgilerin bulunmasıdır. Burada ajanın yeterli zıplama mesafesini sağlayabildiği görülmektedir.



Şekil 3.6: Unity'de NavMesh ile hazırlanmış navigasyon ağı.

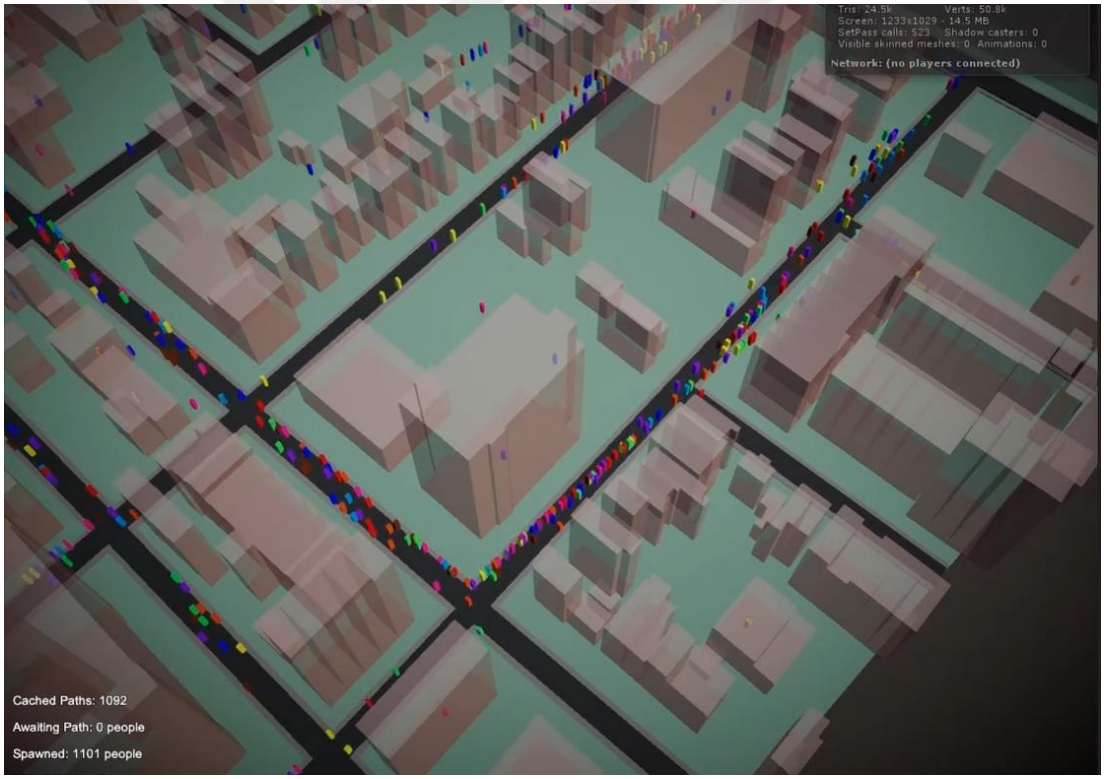
Ajanın hemen yanındaki "navigation static" özellikli kübün zeminle olan bağlantısı navigasyon ağına dahil edilmiş ancak üst kısmı navigasyon ağına dahil edilmemiştir. Bunun nedeni, navigasyon ağı oluşturulurken belirli bir alandan küçük alanların hesaplamaya dahil edilmemesi biçimindeki ayarlamadır. Bu sayede istenmeyen bölgeler yol bulma işlemine dahil edilmez.

Yazılan script ile, uygulama çalıştırıldığında ekranda tıklanılan noktaya ajan otomatik olarak gitmekte ve hedefe vardığında durup yeni komut beklemektedir. Ajanın hemen

sağında farklı renklerde ve navigasyon ağına dahil olan iki alan bulunmaktadır. Bu alanlar NavMesh'de su ve bataklık olarak tanımlanmış ve geçiş maliyetleri yürünebilir zeminden daha yüksek girilmiştir. Bu yüzden ajanın rotası bu alanlarla kesiştiğinde ajan genellikle bu alanlardan geçmemeyi tercih etmektedir. Bu temel NavMesh uygulamasına, tezin sonunda verilen github bağlantısı üzerinden erişim sağlanabilinir.

3.2.2 Örnek çalışmalar

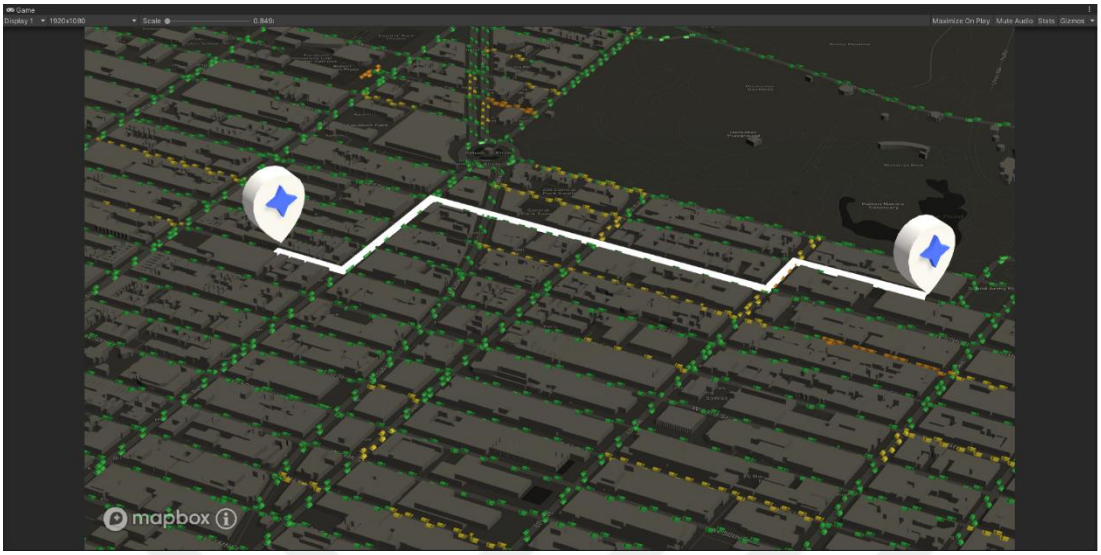
Unity oyun motorunun NavMesh sistemi ve CBS kütüphaneleri kullanılarak yapılmış bazı örnek güzergah belirleme çalışmaları bulunmaktadır. Şekil 3.7'de gösterilen örnek çalışmada Unity'de Mapbox kütüphanesi kullanılarak şehir ve 3 boyutlu bina modelleri alınmış ve daha sonra binalar arasındaki yollar navigasyon ağına dahil edilmiştir. Ajanlar bina kapılarından yola çıkmakta ve şehirde otonom olarak dolaşabilmektedir (Juniadi, 2018).



Şekil 3.7: Unity'de NavMesh ile yapılan yaya trafiği uygulaması.

Şekil 3.8'de görülen örnekte ise, Mapbox firmasının Unity ile kendi yaptığı bir çalışma bulunmaktadır. Bu çalışmada başlangıç ve hedef noktaları sürükle bırak yapılarak şehir üzerinde istenilen yerlere taşınmakta ve NavMesh sistemi otomatik olarak en kısa ve maliyetsiz yolu bulmaktadır. Bunun yanında şehirdeki araç trafiği gösterilmekte ve

trafik yoğunluğu yeşil, turuncu ve kırmızı olarak renklendirilmektedir (MapBox, tarih yok).



Şekil 3.8: Unity'de NavMesh ile yapılan güzergah belirleme uygulaması.

4. GÜZERGAH BELİRLEME ALGORİTMALARI

Tezin bu bölümünde A* algoritmasıyla, temel bir güzergah belirleme uygulaması yapılarak, sonraki bölümlerde yapılacak uygulamalara bir basamak oluşturulmuştur. Unity'nin kendi NavMesh sistemini kullanmak yerine, A* algoritması ile bu işlemler gerçekleştirilmiştir. Bunun en önemli sebebi NavMesh sistemine script'ler ile erişimin istenilen seviyede olmaması ve veri odaklı yöntemle yapılan uygulama için yetersiz kalmasıdır. Öncelikle güzergah belirleme ve A* algoritması tanıtılacak, sonra da Unity oyun motoruyla temel bir yol bulma uygulaması yapılacaktır. Sonraki bölümlerde, Unity'nin geliştirdiği ve yeni bir teknoloji olan Data-Oriented Technology Stack (DOTS) açıklanacak ve DOTS bileşenleri kullanılarak bu güzergah belirleme uygulaması tekrarlanacaktır.

Güzergah belirleme (pathfinding) genellikle iki uç nokta arasındaki en kısa yolu bulmayı ifade eder. Kullanım alanına örnek olarak transit planlama, telefon trafiği yönlendirme, labirent navigasyonu, robot yol bulması verilebilir (Cui & Shi, 2011). Oyun endüstrisinin giderek büyümesiyle yol bulma daha da önemli bir hal almaya başlamıştır. Oyunlarda veya 3 boyutlu diğer uygulamalarda; önceden belirlenmiş veya kullanıcı tarafından belirlenen hedef bir konuma gitmek veya bu güzergahın bilgisini almak sıklıkla gerekmektedir. Engellerden kaçma, farklı arazi üzerinde en verimli yolu bulma gibi problemlerde çalışma alanının dinamik olarak görülebilmesi nedeniyle oyun motorları yol bulma için güzel bir çalışma alanı oluşturmaktadır.

4.1 A* Algoritması

Bu tezde yapılan çalışmadaki amaç, değişik algoritmaların farklarını gözlemlemekten ziyade, seçilen bir algoritmanın farklı koşullarda nasıl sonuç verdiğini incelemektir. Bu yüzden bu tezde, diğer pek çok çalışmada da sıklıkla kullanılan algoritmalardan birisi olan A* güzergah belirleme algoritması kullanılmıştır.

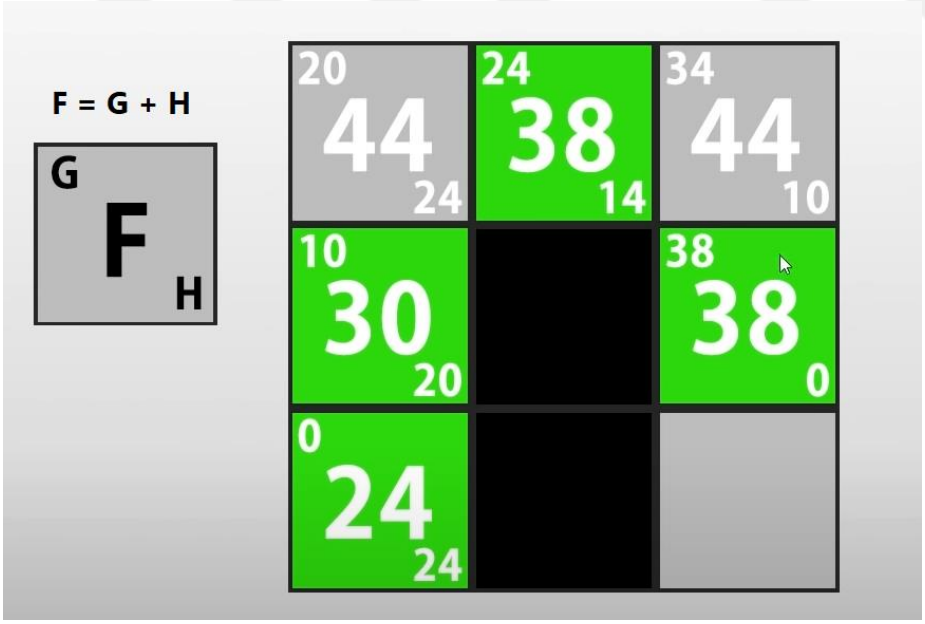
A* algoritması, içinde yol bulmanın da olduğu, birçok soruna çözüm bulmak için kullanılabilen genel bir arama algoritmasıdır. A* güzergah belirleme algoritması, gördüğü en olası keşfedilmemiş konumları tekrar tekrar inceler. Bir konum

araştırıldığında, eğer orasının hedef olduğu bulunursa algoritma tamamlanır. Aksi takdirde, daha fazla keşif için tüm komşularını not eder. A* algoritmasının matematiksel formülü denklem 4.1'de gösterilmiştir.

$$F = G + H \tag{4.1}$$

Burada H, bulunulan hücrenin hedef konumuna ulaşması için gereken sezgisel maliyettir. G, seçilen hücre dizisi boyunca başlangıç konumundan hedef konumuna giden yolun maliyetidir. Bu dizi en son değerlendirilen hücrede biter. Ulaşılan hücrenin, her bitişik hücresi F değeri ile değerlendirilir. En düşük F değerine sahip hücre, dizide bir sonraki hücre olarak seçilir. Bu algoritmanın avantajı kriter olarak kullanılan mesafelerin belirlenebilmesi, değiştirilebilmesi veya başka bir mesafenin eklenebilmesidir (DuchoĖ, ve diğerleri, 2014).

Şekil 4.1'de gösterildiği gibi, sol alttaki 24 numaralı F değerine sahip hücre başlangıç noktasıdır. Sağ ortadaki 38 F değerine sahip hücre hedef konumudur. Ortadaki siyah kutular üzerinden geçilemez engelleri göstermektedir. A* algoritması çevresinde bulunan komşu karelere gideceği noktaya göre belirli maliyet değerleri vermektedir. En az maliyetli yol A* algoritması tarafından yeşil hücreler olarak seçilmiştir.



Şekil 4.1: A* algoritması örneği.

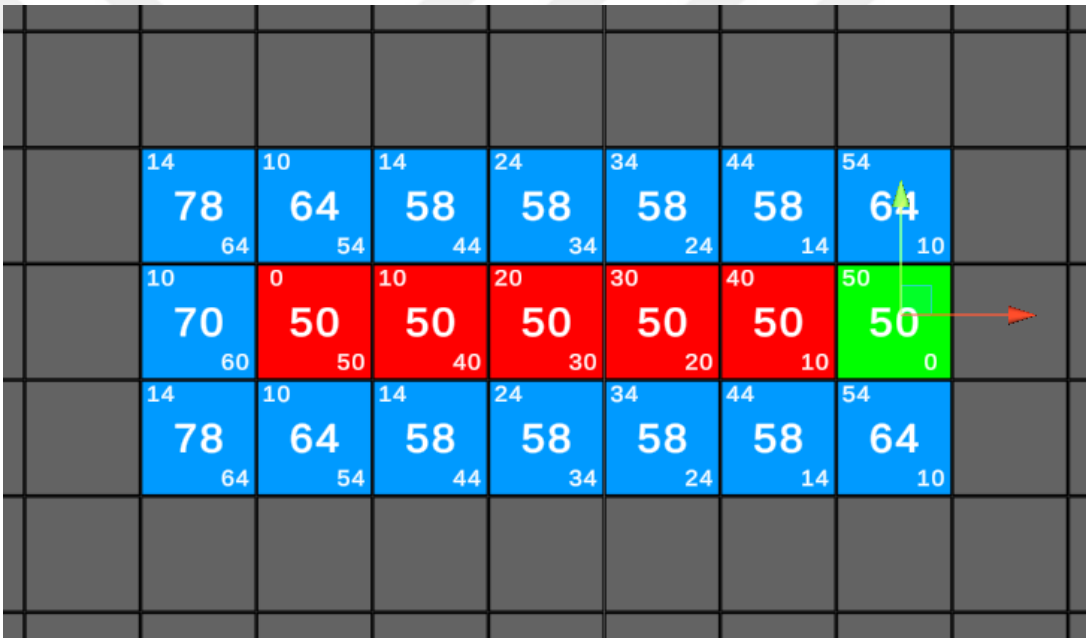
4.1.1 Unity'de A* algoritması uygulaması

Hazırlanan Unity uygulaması başlatıldığında; sahnede, kare şeklinde olan ve gidilebilecek noktaları temsil eden hücreler belirmektedir. A* algoritması kullanılarak

gidilmek istenilen hücre seçildiğinde, ajan otomatik olarak seçilen kareye ilerlemektedir. Çalışma anında, istenilen hücrelere engeller (siyah hücreler) koyulup işlemler tekrarlanabilmektedir. Algoritmanın rotayı bulmak için yaptığı aramayı detaylı görebilmek için, her bir hücre araması adım adım takip edilebilmektedir.

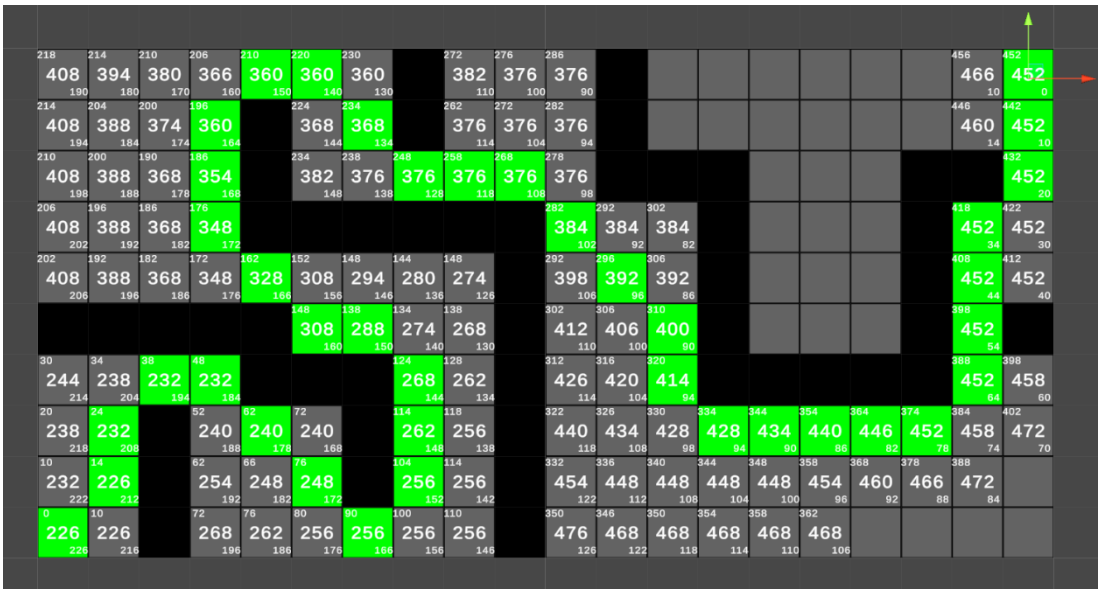
Şekil 4.2'de görüldüğü gibi, Unity'de yapılan uygulamada, kullanılan hücrelere A* algoritması adım adım uygulanırken, hücreler ilgili renklerle gösterilmektedir. Bu farklı renklerdeki hücreler için C# ile yazılan kodda 2 liste kullanılmaktadır. Bunlar açık liste ve kapalı listedir.

- Açık liste, arama için kuyruğa alınmış tüm hücrelerin bulunduğu listedir.
- Kapalı liste, önceden aranmış olan tüm hücrelerdir.



Şekil 4.2: Unity'de A* algoritması uygulanırken kullanılan açık-kapalı liste mantığı.

Aramaya açık listede mevcut hücre bulununa veya açık liste boş olana kadar devam edilir. Eğer arayacak hiçbir yer kalmadıysa, güzergah oluşturulamamıştır. Eğer arama sırasında üzerinde geçilemeyen bir engelle karşılaşılırsa, bu hücre kapalı listeye eklenmektedir. Şekil 4.2'de, kırmızı hücreler kapalı listeye girdiğini, mavi hücreler arandığını ve yeşil hücre ise seçilen yolu göstermektedir. Hedefe ulaştıktan sonra, Şekil 4.3'teki gibi diğer renkler kapatılıp rotayı gösteren yeşil renkli hücreler etkin olmaktadır.



Şekil 4.3: Unity'de A* algoritmasının kullanım örneği.

Bu uygulamadaki asıl amaç temel bir A* algoritması ile güzergah belirleme sistemini kurmaktır. Bu sayede sonraki bölümlerde yapılacak uygulamalara bir temel oluşturmaktadır. Bir kontrol grubu olarak da düşünülebilinir. DOTS ile yapılacak uygulamalarla optimizasyon sağlanacağından kullanılan A* algoritmasının optimizasyonu önemsenmemektedir.

5. UNITY'DE DOTS

Data-Oriented Technology Stack (DOTS), Unity'de kod yazmanın, oyun nesnelerinin mantığını düşünmenin ve verileri ayarlamanın farklı bir yoludur. Bu paradigma değişiminin temel amacı, uygulamanın varsayılan olarak yüksek performanslı çalışmasıdır. Yani Unity'deki uygulama DOTS ile yapılırsa, daha temiz koda ve çok iş parçacıklı (multi-threaded) performansa sahip olacaktır. Bu sayede daha fazla birime (oyun nesnesine), daha fazla görsel efekte, daha iyi görsellere ve daha karmaşık oyunlara olanak sağlanacaktır. Unity'deki paket yöneticisinden DOTS ile ilgili paketleri indirerek DOTS kullanımına başlanabilir.

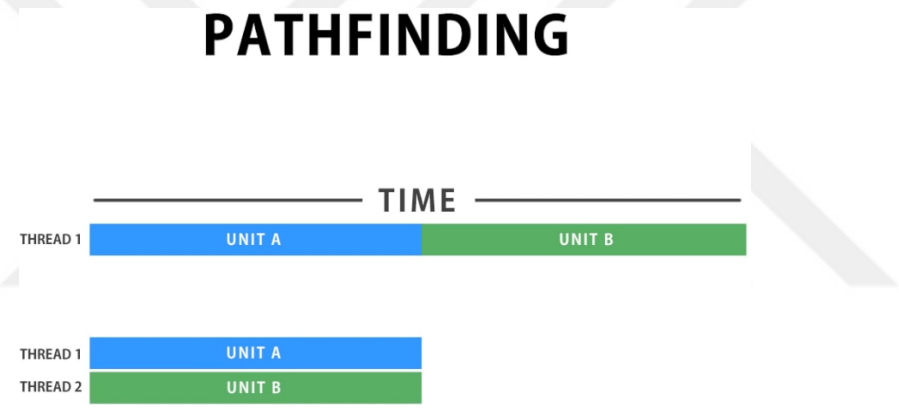
Yeni bir teknoloji olduğu için geliştirme süreci devam etmektedir. Unreal oyun motoru da kendi veri odaklı sistemini geliştirmeyi sürdürmektedir. Yakın gelecekte, oyun motorlarının bu veri odaklı sistemleri, standart sistemleri haline getirecekleri ve şu an kullandığımız nesne odaklı sistemin pek kullanılmayacağı ifade edilmektedir. Bu durumun en büyük sebebi sağladığı yüksek performanstır. Örneğin, nesne odaklı sistemde binlerce nesneyle çalışabilen bir bilgisayar; veri odaklı sistemle milyarlarca nesneyle çalışabilecektir. Bu durumdan en çok da cep telefonları ve mobil uygulamalar yararlanacaktır. Çünkü donanımsal olarak zaten bilgisayarların çok gerisinde olan mobil cihazların aynı zamanda ısınma ve hızlı pil tüketme sorunları bulunmaktadır. Veri odaklı sistem ile hazırlanan bir uygulamada aynı anda milyarlarca nesne görselleştirilmeyecek bile olsa, telefonun daha az kaynağını kullanarak pilinin daha az tükenmesini sağlayacak ve daha az ısındıracaktır. Bu sayede cep telefonunun ömrü de kısalmayacaktır.

Ancak veri odaklı sistemlerin, nesne odaklı sistemlerin yerine geçmesinin önünde bazı engeller bulunmaktadır. En büyük engel, veri odaklı sistemleri öğrenmenin çok zor olması ve karmaşık yapıdaki script kodlarıyla geliştirilmeleridir. Bunun dışında henüz oyun motoruna entegre değildirler. Dışarıdan kütüphaneler yükleyerek kullanılabilmektedir. Eğer planlandığı gibi script dili kolaylaştırılır ve kullanıcıların kolayca öğreneceği bir yapıya bürünürse geleceğin teknolojisi olması beklenmektedir.

DOTS, üç ana bileşenden oluşur: C# İş Sistemi (C# Job System), ECS (Entity-Component-System), Burst Derleyicisi (Burst Compiler).

5.1 C# İş Sistemi

C# İş Sistemi, aynı anda birkaç işi yürütmek için çok çekirdekli işlemcilerden yararlanılmasını sağlar. Unity performansıyla ilgili ana sorunlardan biri, her zaman ana iş parçacığının (main-thread), tek iş parçacıklı olması olmuştur. Şekil 5.1'deki görüldüğü üzere, oyundaki ajanlar için güzergah belirleme işlemimiz varsa, tek iş parçacıklı kodda önce ilk ajanın yolu hesaplanır. Sonra bu bittiğinde, ikinci ajanın yolu hesaplanır ve bu böyle devam eder. Çok iş parçacıklı kod ile rota sayısına bağlı olarak aynı anda birçok yol hesaplanabilir. Oyunda çalışan çok sayıda küçük bağımsız görev varsa, performans bu sayede büyük ölçüde artırılabilir.



Şekil 5.1: Tek ve çok iş parçacıklı kodun zamana bağlı çalışma durumu.

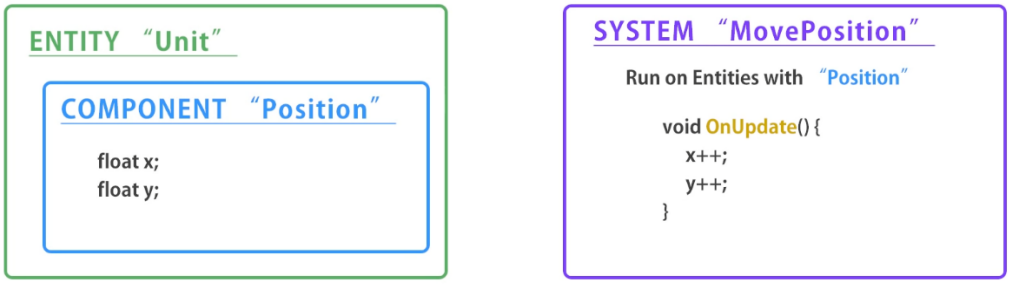
Dikkat edilmesi gereken birçok durum olduğundan dolayı, C#'da çok iş parçacıklı kod yazmak pek de kolay değildir. Çok iş parçacıklı kod yazmak; aynı anda kaç tane iş parçacığı olduğu, kaç tanesini kullanmak gerektiği ve her birinin ne yaptığını yönetmek gerektiği anlamına gelmektedir. C# İş Sistemi, tüm bunları yönetir ve iş parçacığı yerine işler (jobs) oluşturulmasını sağlayarak daha basit kod yazımına yardımcı olur. Daha sonra bu işler, iş parçacıkları üzerinde çalıştırılır ve her şey C# İş Sistemi tarafından yönetilir. Geliştirilen bu çözüm, karmaşıklığı azaltmakta ve çoklu iş parçacıklı kod yazmayı kolaylaştırmaktadır.

5.2 ECS

ECS, farklı bir bakışla kod yazmayı; kodu mantık ve veri olmak üzere ikiye ayırmayı gerektirmektedir. Unity'nin standart MonoBehaviour C# sınıfını içeren oyun nesneleri

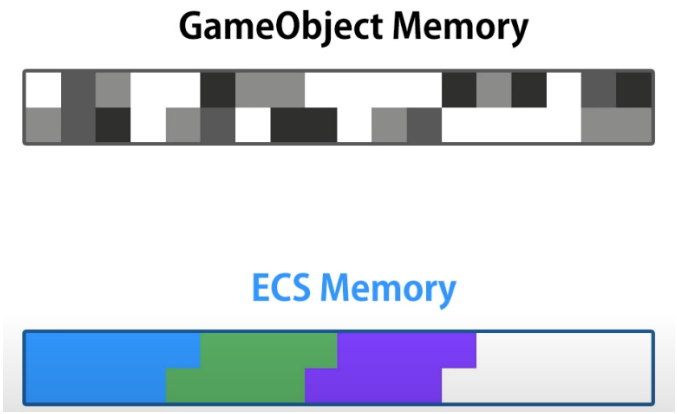
yerine; varlıklar (entities), bileşenler (components) ve sistemler vardır. Başka bir ifadeyle bir varlık var, bu varlık verileri tutan bileşenleri içermekte ve sistemler bu veriler üzerinde mantıksal çıkarımlar yapabilmektedir.

Şekil 5.2'deki örnekte görüldüğü üzere, "Unit" adında bir varlık bulunmaktadır. Bu varlığın "Position" olarak adlandırılan bir bileşeni bulunmakta ve x, y koordinat değerlerini içermektedir. "Position" bileşenine sahip her varlık üzerinde çalışan ve x, y değerleriyle hareket eden "MovePosition" isimli bir sistem bulunmaktadır.



Şekil 5.2: ECS için entity, component ve system örnekleri.

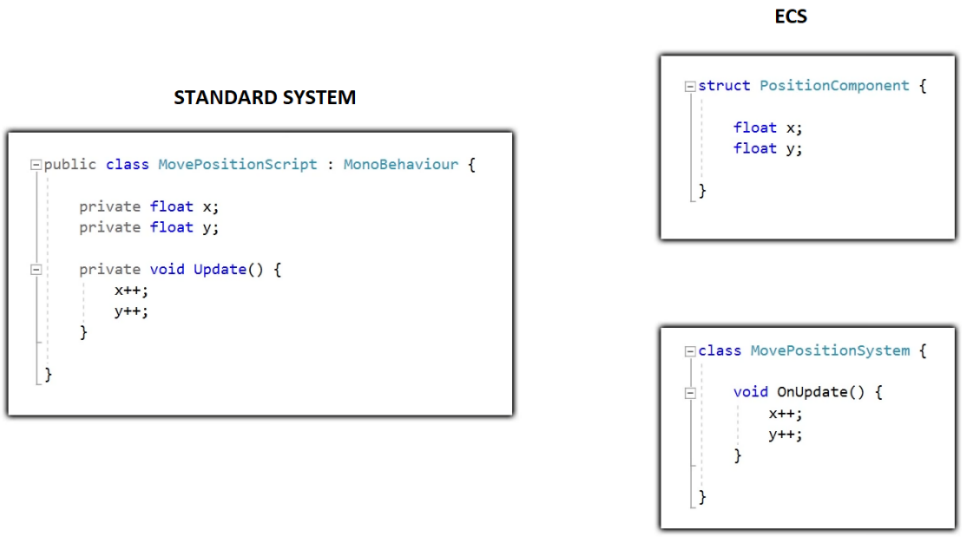
Şekil 5.3'te görüldüğü gibi, ECS'in performanslı çalışmasını sağlayan şey, belleğin nasıl kullanıldığıyla ilişkilidir. Bir işlemci için, bellekte sürekli gezinmek oldukça maliyetlidir. Oyun nesneleri ile normal şekilde çalışırken, bu nesneler çok ağırdır ve bellekte rastgele bir yerde bulunurlar. İş parçacığı çalışırken, işlemci gereksinim duyduğu belleği bulmak için karmaşık bir arama yapar. Öte yandan ECS ile bellek, belirli bileşen gruplarında çok daha fazla paketlenir ve bu da işlemcinin gereksinim duyduğu bir sonraki nesneyi bulmak için artık bellekte gezinmesi gerekmediğinden uygulamanın performansını önemli ölçüde artırır.



Şekil 5.3: Standart bir oyun nesnesi belleği ile ECS belleğinin temsili karşılaştırması.

Bileşenler verileri tutar, sistemler bu verileri işler ve varlıklar bileşen verilerinin tek tek örneklerine atıfta bulunur. Böylece standart oyun nesnesi sistemiyle nasıl bir

ilişkisi olduğu karşılaştırılabilir. Bir varlık, belirli bir oyun nesnesidir. Yani sahnede iki oyun nesnesi varsa, her biri varlık olacaktır. Bileşenler varlıklar tarafından tutulur ve verilere sahiptir. Yani standart sistemde olduğu gibi, ECS'de de her oyun nesnesine bağlı bileşenler bulunmaktadır. Son olarak temel fark, bileşen verileri üzerinde hareket eden sistemlerdir. Standart sistemde, MonoBehaviour C# sınıfı, verileri değişkenler içinde tutar ve bu verileri değiştirmek için fonksiyonlara sahiptir. Şekil 5.4'te görüldüğü gibi, ECS'de, bunlar net bir şekilde ayrılır. Bir varlık, belirli bir bileşen örneğine atıfta bulunur. Bileşenler verileri tutar ve sistemler bu verileri değiştirir. ECS modelini takip etmek daha performanslı kod yazmaya olanak tanır.



Şekil 5.4: C# kodlamada standart sistemle (MonoBehaviour), ECS karşılaştırması.

5.3 Burst Derleyicisi

Burst Derleyicisi, yazılan tüm C# kodunu alır ve yüksek düzeyde optimize edilmiş makine koduna dönüştürür. Bu derleyici özellikle C# İş Sistemi tarafından oluşturulan kodu, yüksek optimize edilmiş koda dönüştürme konusunda iyidir. Derlenmekte olan kodu, belirli platformlar için belirli optimizasyonlardan otomatik olarak yararlandırır. ECS ve C# İş Sistemi'nde kodlar ne kadar iyi yazılırsa, Burst Derleyicisi ile performans artışı o kadar çok olur.

6. DOTS UYGULAMALARI

Tezin bu bölümünde, Unity'nin standart sistemiyle yapılmış olan A* güzergah belirleme uygulaması, C# İş Sistemi ve Burst Derleyicisi kullanılarak yapılacak ve performans farkı gözlemlenecektir. Daha sonraki aşamada ECS de kullanılacak ve böylelikle tüm DOTS elemanları uygulanmış olacaktır.

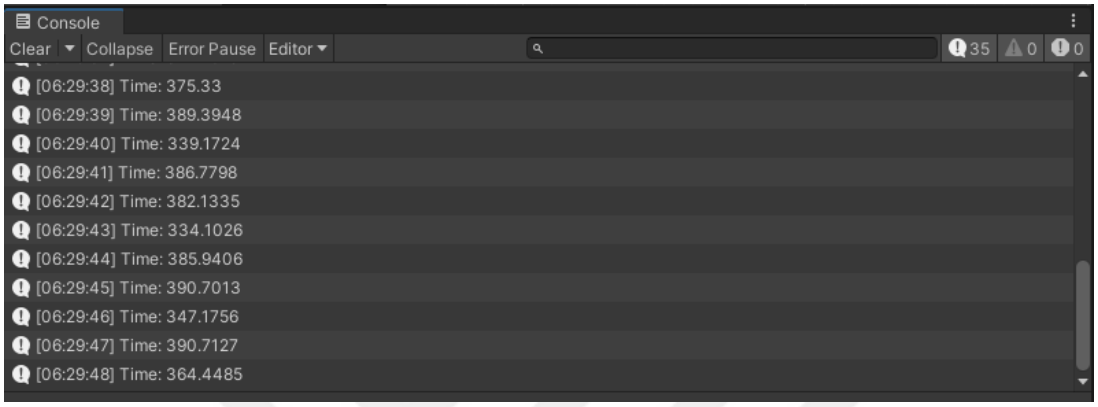
6.1 C# İş Sistemi ve Burst Derleyicisi Uygulamaları

Öncelikle Unity'nin standart sistemiyle hazırlanan algoritmanın, yapılacak diğer uygulamalar için referans olması için performansı ölçülmüştür. Bunun için algoritmanın otomatik çalıştığı bir simülasyon ortamı hazırlanmıştır. Şekil 6.1'deki kod parçasında görüldüğü gibi, kare şeklinde hücrelerden oluşan, 20*20 boyutlarında bir grid ağında; ajanın algoritmaya başlangıç noktası en sol alt ve bitiş noktası en sağ yukarı olacak şekilde ayarlanmıştır. Ajan durmaksızın, başlangıç noktasından hedefe doğru A* güzergah belirleme algoritmasıyla belirlenen parametreler doğrultusunda güzergahı oluşturmaktadır. Bu sırada bulunan güzergahların kaç milisaniyede hesaplandığı Unity'nin konsol paneline yansıtılmaktadır. Şekil 6.1'de gösterilen "testPathCount" sayısı, kaç tane güzergah belirleme işinin tam olarak aynı karede çalıştırılmasını ayarlayan parametredir. Yani her seferinde 5 güzergah birden hesaplanmaktadır. Bu sayı arttırılısa sisteme binen yük de artacaktır.

```
Event function
private void Start() {
    pathfinding = new Pathfinding( width: 20,  height: 20);
    FunctionPeriodic.Create( action: () =>
    {
        float startTime = Time.realtimeSinceStartup;

        int testPathCount = 5;

        for (int i = 0; i < testPathCount; i++)
        {
            pathfinding.FindPath( startX: 0,  startY: 0,  endX: 19,  endY: 19);
        }

        Debug.Log( message: "Time: " + ((Time.realtimeSinceStartup - startTime) * 1000f));

    },  timer: 1f);
    pathfindingDebugStepVisual.Setup(pathfinding.GetGrid());
    pathfindingVisual.SetGrid(pathfinding.GetGrid());
}
```

Şekil 6.1: Standart sistemle hazırlanan A* güzergah belirleme algoritması parametreleri.

Önceki bölümlerde de bahsedildiği gibi, kullanılan A* algoritması optimize edildiği takdirde performansı bir miktar daha iyileşecektir. Ancak bu haliyle elde edilen sonuçlara Şekil 6.2'de bakıldığında, standart sistemle ajanın güzergahı oluşturması yaklaşık 330 ile 400 milisaniye arasındadır. Standart sistem performans bakımından yetersiz olduğu için grid ağını büyütünce veya hesaplanan güzergah sayısını arttırınca; donanımın gücü yetersiz kalıp, Unity'de çökmeler meydana gelmektedir.



Şekil 6.2: Standart sistemde 20*20'lik gridin milisaniye bazında performansı.

Kodda değişikler yapılarak temel bir DOTS kullanımı hazırlanmıştır. Bu ilk DOTS uygulamasında sadece C# İş Sistemi kullanılmıştır. A* algoritmasında öncelikle, standart metoddaki parametrelerin aynıları kullanılmıştır (20*20 grid ve 5'er kere rota hesaplanması). Şekild 6.4'te görüldüğü gibi, ikinci denemede grid boyutu 100*100'e çıkartılmış ve 50'li şekilde rota hesaplanması sağlanmıştır. Sonuçlar Şekil 6.3'te görüldüğü üzere standart sistemin çok altında kalmıştır. 20*20'lik gridde yaklaşık 1.6 milisaniyedir. Şekil 6.5'teki gibi, 100*100'lük gridde yaklaşık 46 milisaniyelik bir sonuç elde edilmiştir.

Şekil 6.3: Sadece C# İş Sistemi ile 20*20'lik gridin milisaniye bazında performansı.

```
private void Start()
{
    FunctionPeriodic.Create( action: () => {
        float startTime = Time.realtimeSinceStartup;

        for (int i = 0; i < 50; i++)
        {
            FindPath(new int2( x: 0,  y: 0), new int2( x: 99,  y: 99));
        }

        Debug.Log( message: "Time: " + ((Time.realtimeSinceStartup - startTime) * 1000f));

    }, timer: 1f);
}
```

Şekil 6.4: Sadece C# İş Sistemi ile hazırlanan A* güzergah belirleme algoritması parametreleri.

Şekil 6.5: Sadece C# İş Sistemi ile 100*100'lük gridin milisaniye bazında performansı.

Yukarıdaki C# İş Sistemi ile hazırlanan uygulamada, sadece ana iş parçacığı kullanılarak rotalar sırayla hesaplanmıştır. Yani çoklu bir iş parçacığı kullanımı olmamıştır. Uygulamanın bu kısmında aynı algoritma bu sefer, çoklu iş parçacıkları kullanılarak tekrar çalıştırılmıştır. Aynı şekilde önce 20*20'lik grid ile ve sonra 100*100'lük grid ile alınan sonuçlar performansın daha da iyileştiğini göstermektedir. Şekil 6.6'da görüldüğü gibi 20*20'lik gridde ortalama 0.11 milisaniyeye ulaşılmıştır. Şekil 6.7'de 100*100'lük grid için kullanılan kod ve parametre değerleri gösterilmektedir. Şekil 6.8'de görüldüğü üzere 100*100'lük gridde yaklaşık 4.5 milisaniyelik bir sürede hesaplamalar yapılmaktadır.

Şekil 6.6: C# İş Sistemi ve çoklu iş parçacığı birlikte kullanıldığında, 20*20'lik gridin milisaniye bazında performansı.

```
Event function
private void Start() {
    FunctionPeriodic.Create(action: () => {
        float startTime = Time.realtimeSinceStartup;

        int findPathJobCount = 50;
        NativeArray<JobHandle> jobHandleArray = new NativeArray<JobHandle>(findPathJobCount, Allocator.TempJob);

        for (int i = 0; i < findPathJobCount; i++) {
            FindPathJob findPathJob = new FindPathJob {
                startPosition = new int2(x: 0, y: 0),
                endPosition = new int2(x: 99, y: 99)
            };
            jobHandleArray[i] = findPathJob.Schedule();
        }

        JobHandle.CompleteAll(jobHandleArray);
        jobHandleArray.Dispose();

        Debug.Log(message: "Time: " + ((Time.realtimeSinceStartup - startTime) * 1000f));
    }, timer: 1f);
}
```

Şekil 6.7: C# İş Sistemi ve çoklu iş parçacığı birlikte çalışırken kullanılan A* güzergah belirleme algoritmasının parametreleri.

Şekil 6.8: C# İş Sistemi ve çoklu iş parçacığı birlikte kullanıldığında, 100*100'lük gridin milisaniye bazında performansı.

Bu aşamadan sonra daha önceki uygulamalarda kullanılmayan, Burst Derleyicisi'ni devreye sokulmuş ve uygulama tekrarlanmıştır. Burst Derleyicisi'ni devreye sokmak için koda ilgili kütüphaneyi ekleyip, Şekil 6.9'da gösterilen "[BurstCompile]" satırının eklenmesi yeterlidir.

```
[BurstCompile]
2 usages
private struct FindPathJob : IJob {

    public int2 startPosition;
    public int2 endPosition;
```

Şekil 6.9: Burst Derleyicisi'ni çalıştırmak için koda eklenen satır, "[BurstCompile]"

Şekilde 6.10'da ve 6.11'de görüldüğü üzere, Burst Derleyicisi algoritmaya dahil edildiğinde elde edilen sonuç 20*20'lik gridde yaklaşık 0.04 milisaniye ve 100*100'lük gridde yaklaşık 1.5 milisaniye olarak kaydedilmiştir. Bu sonuçlar bu çalışmada elde edilen en hızlı değerlerdir. Çizelge 6.1'de tüm sonuçlar gösterilmiştir.

Şekil 6.10: Burst Derleyicisi kullanıma dahil edildiğinde, 20*20'lik gridin milisaniye bazında performansı.

Şekil 6.11: Burst Derleyicisi kullanıma dahil edildiğinde, 100*100'lük gridin milisaniye bazında performansı.

Çizelge 6.1: A* algoritması simülasyonunun farklı yöntemlerle sonuçları.

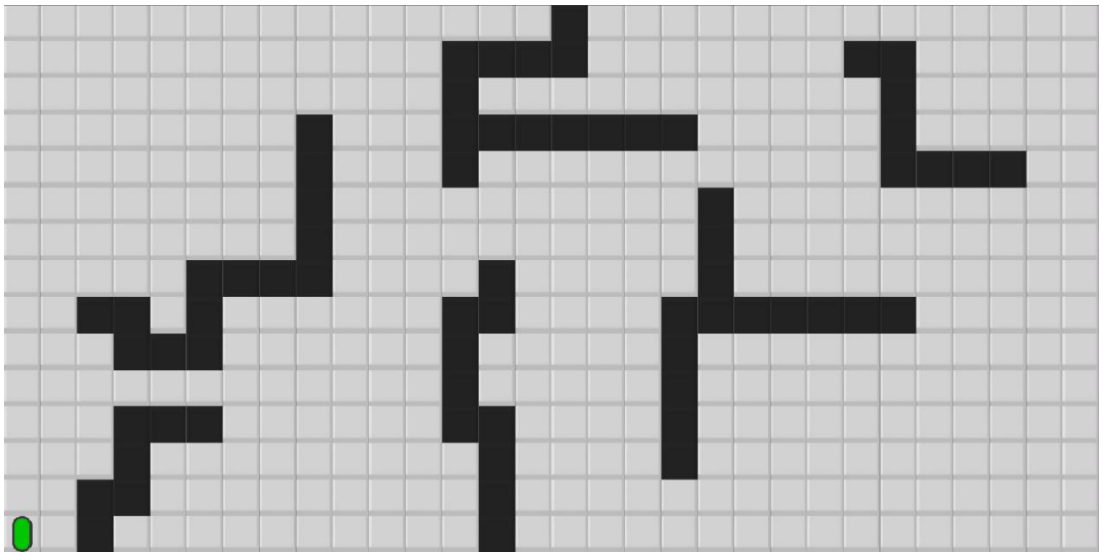
Kullanılan yöntem	20*20'lik grid ve 5'li iş yapımı (milisaniye)	100*100'lük grid ve 50'li iş yapımı (milisaniye)
Standart sistem	330-400	Çalışmadı
C# İş Sistemi	1.6	46
C# İş Sistemi + çoklu işlem parçacığı	0.11	4.5
C# İş Sistemi + çoklu işlem parçacığı + Burst Derleyicisi	0.04	1.5

6.2 ECS Uygulaması

Tezin bu bölümünde, bir önceki bölümde yapılan C# İş Sistemi ve Burst Derleyicisi uygulamasına, ECS entegre edilmiştir. Bu sayede DOTS elemanlarının tümü ile A* güzergah belirleme algoritması gerçekleştirilmiştir. DOTS'da:

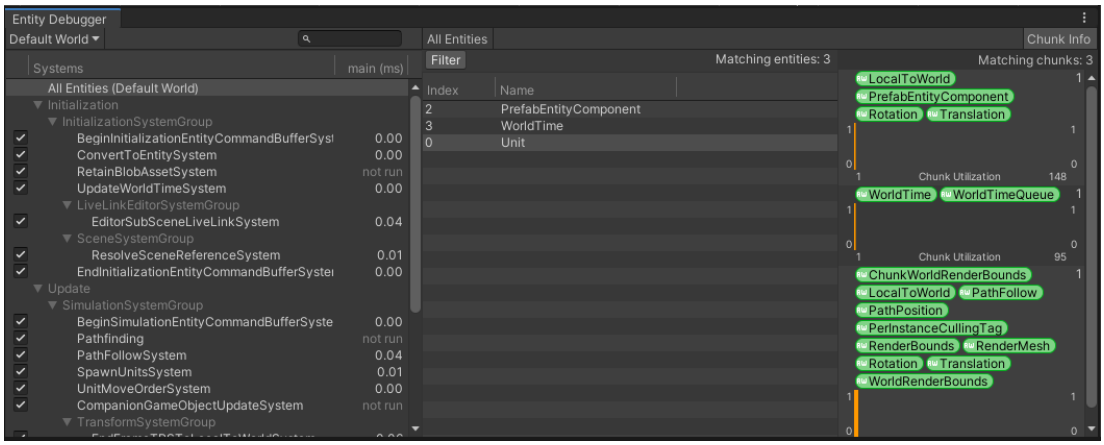
- C# İş Sistemi, bulunacak yolları paralel hesaplar.
- Burst Derleyicisi, kullanılan tüm kodları hızlı makine kodlarına çevirir.
- ECS, veri odaklı varlıklar ile algoritmayı yüksek performansla çalıştırır.

Şekil 6.12'de görüldüğü üzere, yapılan ECS uygulaması çalıştırıldığında, hareketli ajan, başlangıç noktası olarak belirlenen grid yüzeyin en sol alt noktasında ((0,0) koordinatlı hücrede) oluşturulmuştur. Grid yüzeyde istenilen hücreler seçilerek engel haline getirilebilmektedir (siyah renkli hücreler). Ajan hareket halindeyken bile engeller istenildiği gibi eklenip kaldırılabilir. Ajana herhangi bir hücre hedef olarak seçilir ve ajan A* güzergah belirleme algoritmasıyla hesapladığı güzergaha uygun bir şekilde hedefe doğru ilerler. Hedefe vardığında, o anki konumu başlangıç noktası ve yeni seçilecek nokta yeni hedef olur ve seçilen yeni rotalara hareket etmeye devam eder. Tüm bu işlemler sırasında Şekil 6.12'de görülen yeşil renkli ajan, oyun nesnesi değil, bir varlıktır. Yani veri-odaklı bir yapıdadır. Bu da ECS kullanımının bir sonucudur.



Şekil 6.12: Güzergah belirleme algoritması için oluşturulan grid yüzey, engeller ve hareketli ajan.

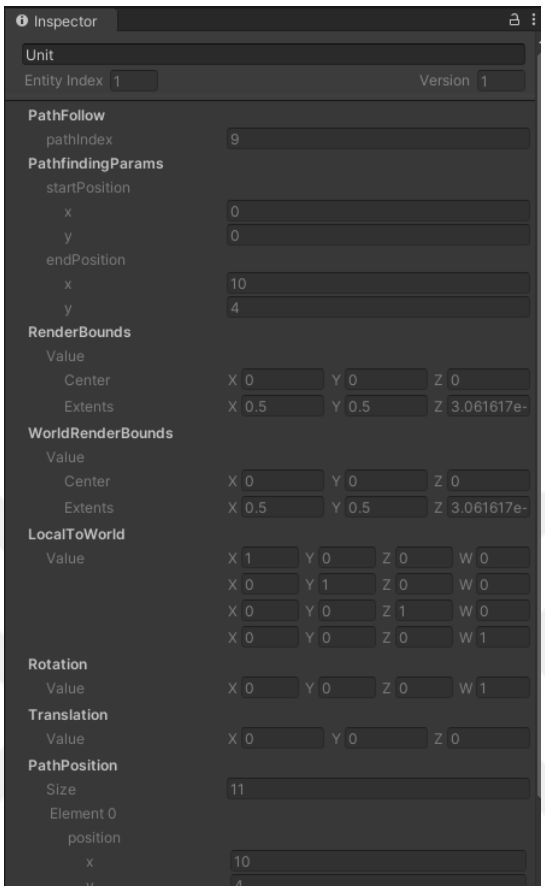
Unity'de varlıklar, nesne olmadıkları için yer kaplamazlar ve Hierarchy panelinde gözükmezler. Bu sayede bu tip varlıklardan binlercesi aynı simülasyonda çok az performans kaybıyla kullanılabilir. Varlıklar sadece Entity Debugger panelinden gözlemlenebilmektedir (Şekil 6.13). Bu panelde filtreleme yapılıp, istenilen varlıklar, bileşenler ve sistemler seçilerek; Inspector panelinden hakkında detaylı bilgiler elde edilir. Var olan oyun nesnelerine DOTS kütüphanesindeki ilgili script'i ekleyip varlığa dönüştürerek veya C# koduyla direkt olarak varlık oluşturarak Unity'de iki farklı şekilde varlık oluşturulabilir.



Şekil 6.13: Entity Debugger paneli.

Bu uygulamada öncelikle ajan, oyun nesnesinden varlığa çevrilmiştir. Bu sayede nesne olmaktan çıkıp, veri haline gelmiştir. Şekil 6.14'te hareketli ajanın Inspector panelindeki, güzergah istemek ve onu takip etmek için kullandığı çeşitli yol bulma

bileşenleri bulunmaktadır. Burada gözüken PathFollow, PathfindingParams ve PathPosition bu bileşenlere örnektir.



Şekil 6.14: Inspector panelinde, hareketli ajanın varlık bileşenleri.

Uygulama çalıştırılıp, ajanın gitmesi için bir hedef hücre seçildiğinde, hareketli ajana bu sırada PathfindingParams bileşeni otomatik olarak eklenir. Bu bileşen yol bulma algoritmasında kullanılan, başlangıç ve hedef konumlarının x ve y koordinatlarını içerir (Şekil 6.15). Yol bulma sistemi (pathfinding system), PathfindingParams bileşenlerine sahip varlıkları aramaktadır. PathfindingParams bileşenine sahip bir varlık bulduğunda bu parametrelerle yol bulma algoritmasını çalıştırır ve hesapladığı rotayı PathPosition (Şekil 6.14) bileşenine gönderir. PathPosition'da, hedef hücreye ulaşmak için hareketli ajanın üzerinden geçmesi gereken tüm hücrelerin sırayla x ve y koordinat bilgisi bulunmaktadır. Ajan, bu hücrelerden PathPosition'daki sıraya göre tek tek geçerek, A* algoritmasının bulduğu hedefe ulaşır. Ajan hedefe ulaştıktan sonra PathfindingParams bileşeni üzerinden silinir. Eğer yeni hedef belirlenirse işlemler tekrarlanarak ajan hareket eder.

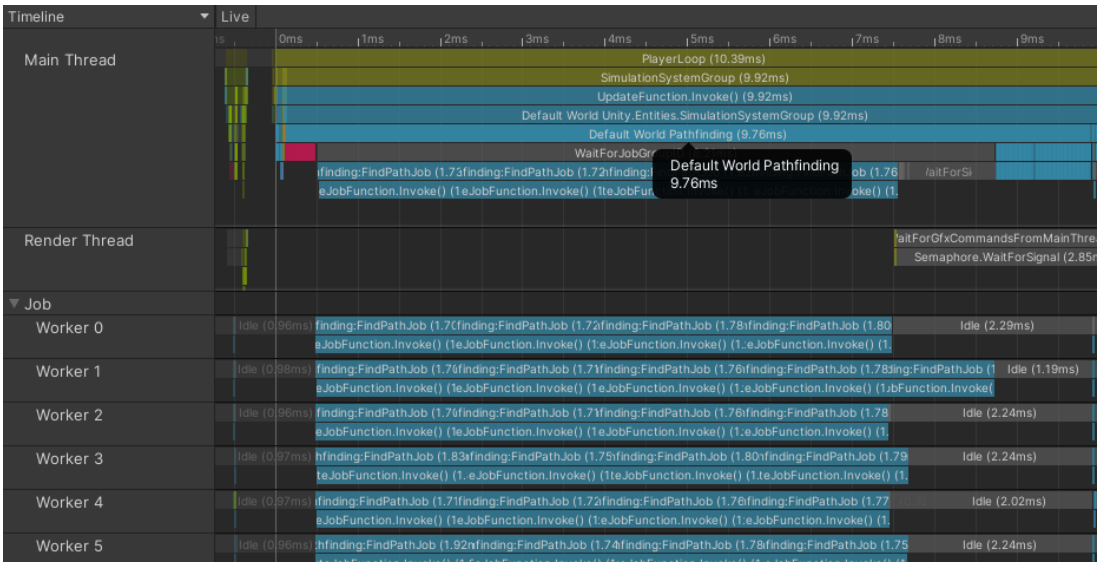
```
8 usages
public struct PathfindingParams : IComponentData {

    public int2 startPosition;
    public int2 endPosition;

}
```

Şekil 6.15: Sadece başlangıç ve hedef konum bilgisini tutan PathfindingParams bileşeni.

Özetlemek gerekirse, ajana bir hedef seçilip yol bulma parametreleri otomatik olarak eklenir. Yol bulma sistemi bu sırada, diğer varlıkları dinlemektedir. Eğer yol bulma parametresine sahip bir bileşen bulursa, ilgili varlık için yol bulma işini çalıştırıp güzergahı hesaplar. Güzergahı hesaplayan sistem, varlığa güzergah bilgisini bileşen olarak ekler (PathPosition). Son olarak da ajan bu güzergaha göre hedefe doğru hareket eder.

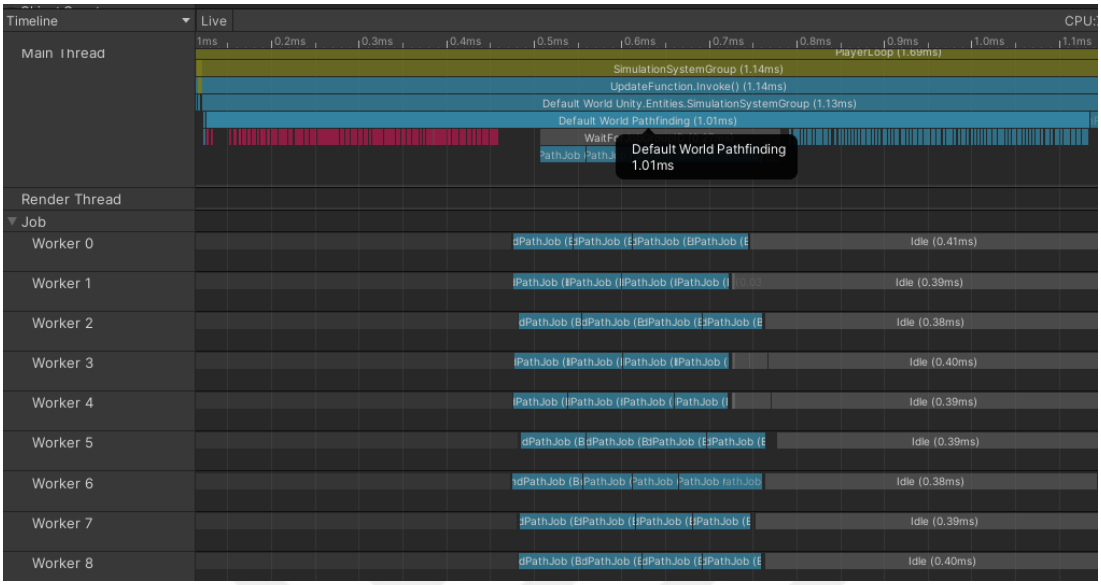
Uygulama bu şekilde çalışmaktadır. Ancak bu noktaya kadar sadece bir ajan kullanarak çalıştırılmıştır. Bundan sonra güzergah belirleme algoritması birden çok ajan ile paralel hesaplama yaparak çalıştırılmıştır. Bunun için kod C# İş Sistemi'ne uygun olarak düzenlenmiştir. Bu sayede çoklu işlem parçacıkları kullanılarak performans artışı sağlanmıştır.

Bu uygulamada Öncelikle 100 tane hareketli ajan varlığı, yol bulma işini aynı anda Burst Derleyicisi kapalıyken yapmıştır. Şekil 6.16'daki Entitiy Debugger panelinde görüldüğü gibi yol bulma işi 9.76 milisaniye sürmüştür.



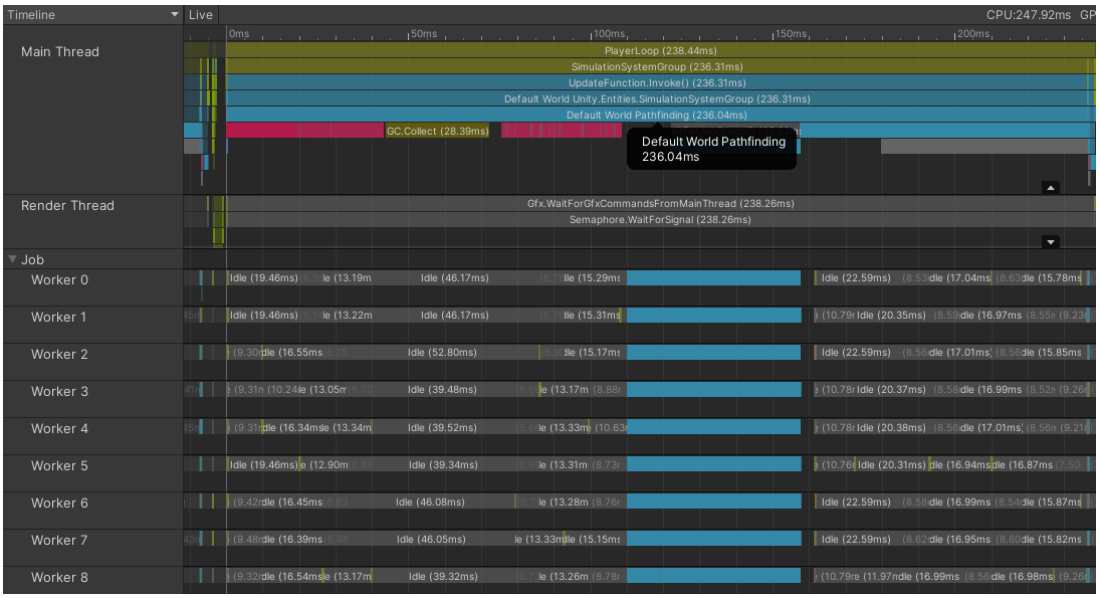
Şekil 6.16: Burst Derleyicisi kapalıyken, 100 hareketli ajan varlığının yol bulma işi 9.76 milisaniyede hesaplandı.

Daha sonra yine 100 hareketli ajan varlığı, yol bulma işini aynı anda Burst Derleyicisi açıkken yapmıştır. Şekil 6.17'deki Entitiy Debugger paneline göre 1.01 milisaniyede tamamlamıştır.



Şekil 6.17: Burst Derleyicisi açıkken, 100 hareketli ajan varlığının yol bulma işi 1.01 milisaniyede hesaplandı.

Son olarak, DOTS'un sağladığı performansın sınırlarını zorlamak adına Burst Derleyicisi etkinken 20000 hareketli ajan varlığının aynı anda A* algoritması ile yol bulma işi yaptırılmıştır. Şekil 6.18'deki Entitiy Debugger panelinde görüldüğü gibi yol bulma işi 236.04 milisaniyede tamamlanmıştır. Bu türde yüksek işlem gücü isteyen bir hesaplamayı DOTS'un sunduğu teknoloji olmadan çalıştırmak mümkün olmayacaktır. Aynı çalışmanın 20000 varlık yerine 20000 oyun nesnesi kullanılarak çalıştırılması olası olamayacaktı.



Şekil 6.18: Burst Derleyicisi açıkken, 20000 hareketli ajan varlığının yol bulma işi 236.04 milisaniyede hesaplandı.

7. SONUÇ VE ÇIKARIMLAR

Son birkaç yılda oyun motoru teknolojisinin hızlı gelişimi ile her CBS kullanıcısı, oyun teknolojilerinin gelecekte CBS deneyimini temelden değiştireceğinin farkında olmalıdır. Çünkü oyun motorunun sunduğu teknolojiler CBS yazılımlarınnın çok ilerisindedir. Oyun motorlarında gelişmiş görselleştirme, fizik ve ses motorları bulunmaktadır. Bunun yanında animasyon, yapay zeka gibi sistemleri de vardır. En önemlisi de, ihtiyaç duyulan herhangi bir aracın veya özelliğin script yapısı kullanılarak motora eklenebilmesidir. Oyun motorlarının çok geniş geliştirici topluluğu olduğu için, kullanıma hazır bir çok kütüphanesi bulunmaktadır.

NavMesh, Unity oyun motorunun kendi güzergah belirleme sistemidir ve geliştiriciler tarafından sıklıkla kullanılmaktadır. Otonom hareket eden yapay zeka ajanlarının gidecekleri yollar NavMesh ile belirlenmektedir. İstenilen yüzeyler "navigation static" özelliği açılarak, navigasyon ağına dahil edilebilmektedir. Bu sayede engeller ve yollar oluşturulmaktadır. NavMesh sistemi bazı CBS uygulamaları tarafından da kullanılmaktadır.

Bu tez çalışmasında, Unity oyun motorunda A* güzergah belirleme algoritması kullanılarak uygulamalar geliştirilmiştir. Çalışmanın amacı kullanılan A* algoritmasının iyileştirilmesine ilişkin olmayıp, bunun yerine aynı algoritmanın farklı yaklaşımlarla nasıl sonuç vereceğini ölçmek olduğu için; temel bir A* yol bulma algoritması kullanılmıştır. Geliştirilen uygulamalar nesne ve veri odaklı olmak üzere ikiye ayrılmıştır. Nesne odaklı yöntem için Unity'nin standart sürümü doğrudan kullanılırken, veri odaklı yöntem için DOTS kütüphanesinden faydanılmıştır. Unity'de DOTS'un temel amacı nesne odaklı sisteme alternatif getirip daha fazla performans sunmak olması dolayısıyla uygulama sonuçlarında da bu başarım artışı gözlemlenmiştir. 2 farklı boyuttaki birim karelerden oluşan bir grid ağ üzerinde, başlangıç ve bitiş noktaları belirlenip nesne ve veri odaklı yöntemlerin performansları test edilmiştir. Veri odaklı sistem açık ara fark ile nesne odaklı sistemi güzergah belirleme hesaplamalarında geçmiştir. Veri odaklı sistem ise kendi içinde 3'e ayrılmaktadır. Bunlar, en performanslı yöntemden en az performanslı olana şu şekilde

sıralanmıştır: C# İş Sistemi + çoklu işlem parçacığı metodu + Burst Derleyicisi'nin birlikte kullanımı, C# İş Sistemi + çoklu işlem parçacığı metodunun birlikte kullanımı ve sadece C# İş Sistemi kullanımıdır.

ECS'de varlıklar çeşitli bileşenleri barındırmaktadır. Sistemler ise bu bileşenleri kullanarak varlıklara etki etmektedir. ECS, oyun nesnelerini varlıklara dönüştürerek nesne odaklı sistemin hantal yapısından kurtulmasını sağlamıştır. Böylece normalde sahnede aynı anda birden fazla ajan çalıştırıp hesaplama yapılabileceğinden çok daha fazla nesneyi çalıştırmak mümkün olmuştur. En son uygulamada 20000 hareketli ajan aynı anda çalıştırılmıştır.

Sonuç olarak, veri odaklı çalışma sistemi oyun motorları için yeni bir bakış ve vizyondur. Bu teknoloji henüz çok yeni olmasına rağmen şimdiden eski yönteme göre çok daha fazla performans sunabilmektedir. Ancak kod yazımının zorluğu, projeyi DOTS'a göre baştan konfigüre etmenin gerekliliği, veri odaklı sisteme alışmanın zorluğu kullanıcıların bu sisteme geçmesini engelleyen unsurlar arasındadır. DOTS hala geliştirme aşamasında olup bir gün oyun motorlarında standart yöntem olarak kullanılacağı düşünülmektedir.

Bu tezde yer alan uygulamalara erişmek için aşağıdaki web adresi kullanılabilinir: https://github.com/abaytimur/OyunMotorlarindaGuzergahBelirleme

KAYNAKLAR

Akenine-Moller, T., Haines, E., & Hoffman, N. (2008). Real-Time Rendering. *3rd Edition* (s. 279-283). içinde

Arif, F. (2014, Ekim 7). *Assassin's Creed Unity In-Game Paris vs Real Life Paris Screenshot Comparison Shows Dynamic Structural Visuals*. WccfTech: https://wccftech.com/assassins-creed-unity-ingame-paris-real-life-comparison/ adresinden alındı

Brodkin, J. (2013, Haziran 3). *Dice*. https://insights.dice.com/2013/06/03/how-unity 3d-become-a-game-development-beast/ adresinden alındı

Cui, X., & Shi, H. (2011, Ocak). A*-based Pathfinding in Modern Computer Games. *IJCSNS International Journal of Computer Science and Network Security*, 125.

Dealessandri, M. (2020, Ocak 16). *GameIndustry*. https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you adresinden alındı

Doğan, C. (2019, Mayıs). Unity Oyun Motoru ile Simülatör Tasarımı. *İstanbul Teknik Üniversitesi*.

DuchoĖ, F., Babinec, A., Kajan, M., BeĖo, P., Florek, M., Ficoa, T., & Jurišica, L. (2014). Path planning with modified A star algorithm for a mobile robot. *ScienceDirect*, 59–69.

Environmental Systems Research Institute (ESRI). (1992). *Understanding GIS: The ARC/INFO Method.* http://www.ciesin.org/docs/005-331/005-331.html adresinden alındı

Epic Games. (2018). *Behavior Trees*. Unreal Engine Documents: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/ adresinden alındı

Escobar, F., Hunter, G., Bishop, I., & Zerger, A. (2005). *Introduction to GIS.* The University of Melbourne. http://www.geogra.uah.es/patxi/gisweb/GISModule/GISTheory.pdf adresinden alındı

Geig, M. (2019). Converting your game to DOTS. Copenhagen. https://www.youtube.com/watch?v=BNMrevfB6Q0 adresinden alındı

Giant Army. (2020). *What is Universe Sandbox ?* Universe Sandbox: http://universesandbox.com/ adresinden alındı

Gibson, A. (2018, Ekim 2). *Here's How Big Assassin's Creed Odyssey's Map is Compared to Greece.* Twinfinite: https://twinfinite.net/2018/10/how-big-assassins-creed-odysseys-map-compared-greece/ adresinden alındı

Haas, J. K. (2014). A History of the Unity Game Engine. *Interactive Qualifying Project*. https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/ adresinden alındı

Holt, K. (2019, Nisan 17). *Engadget*. Ubisoft is donating $564,000 to help rebuild Notre-Dame: https://www.engadget.com/2019-04-17-notre-dame-fire-apple-ubisoft-assassins-creed-unity.html adresinden alındı

Juniadi, Z. (2018, Temmuz 24). *Unity ECS with Navmesh and MapBox*. GitHub: https://github.com/zulfajuniadi/unity-ecs-navmesh adresinden alındı

Lester, P. (2005, Temmuz 18). A* Pathfinding for Beginners. http://www.gamedev.net/reference/articles/article2003.asp adresinden alındı

Lewis, M., & Jacobson, J. (2002). Game Engines in Scientific Research. *Communications of the ACM*, 27-31. https://www.cse.unr.edu/~sushil/class/gas/papers/GameAIp27-lewis.pdf adresinden alındı

Lowrie, M. (2019, Nisan 17). *Ubisoft historian says company's 3D Notre Dame Cathedral a reminder of beauty amid rebuild*. GlobalNews: https://globalnews.ca/news/5177299/ubisoft-notre-dame-cathedral-assassins-creed/ adresinden alındı

Macgregor, J. (2019, Ağustos 22). *How teachers are using Minecraft in the classroom*. PC Gamer: https://www.pcgamer.com/how-teachers-are-using-minecraft-in-the-classroom/ adresinden alındı

MapBox. (tarih yok). *Traffic and directions*. MapBox: https://docs.mapbox.com/unity/maps/examples/traffic-and-directions/ adresinden alındı

Mat, R., Shariff, A., Zulkifli, A., Rahim, M., & Mahayudin, M. (2014). Using game engine for 3D terrain visualisation of GIS data: A review. *IOP Conference Series: Earth and Environmental Science*. https://iopscience.iop.org/article/10.1088/1755-1315/20/1/012037 adresinden alındı

O'Malley, J. (2019, Haziran 12). *How accurate is Watch Dogs Legion's London? We break down the trailer*. Tech Radar: https://www.techradar.com/news/how-accurate-is-watch-dogs-legions-london-we-break-down-the-trailer adresinden alındı

Patel, A. (2019). *Amit's Thought on Pathfinding*. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html adresinden alındı

Power, T. (2018, Eylül 18). *Is Shadow of the Tomb Raider Exploitation?* The Escapist: https://www.escapistmagazine.com/v2/opinion-is-shadow-of-the-tomb-raiders-mesoamerica-an-homage-or-an-exploitation/ adresinden alındı

Rees, E. V. (2018, Şubay 19). *Enhancing GIS with augmented reality*. Geo Week News: https://www.geoweeknews.com/blogs/enhancing-gis-augmented-reality adresinden alındı

Şahin, İ. (2020, July). Unity 3d Oyun Ortamında Akıllı Ajanlar ile Kaçma-Kovalama Oyun Tasarımı. *İstanbul Teknik Üniversitesi*.

Schwab, B. (2004). *AI Game Engine Programming.* Charles River Media. https://scholar.google.com/scholar_lookup?title=AI%20game%20engine%20programming&publication_year=2004&author=Schwab%2CB adresinden alındı

Sidhu, H. (2020). Performance Evaluation of Pathfinding. https://scholar.uwindsor.ca/etd/8178/?utm_source=scholar.uwindsor.ca%2Fetd%2F8178&utm_medium=PDF&utm_campaign=PDFCoverPages adresinden alındı

Šmíd, A. (2017, Mayıs). *Comparison of Unity and Unreal Engine.* Prague: Czech Technical University. https://core.ac.uk/download/pdf/84832291.pdf adresinden alındı

Unity Technologies. (2020). *Deferred Shading rendering path*. Unity3D: https://docs.unity3d.com/Manual/RenderTech-DeferredShading.html adresinden alındı

Unity Technologies. (2020). *Unity Manual*. https://docs.unity3d.com/Manual/index.html adresinden alındı

Wang, X., Lin, W., & Zhao, J. (2020). DOTS in Unity3d Game. *ICVRV.*

Woods, A. (2019, Nisan 17). *Video game could be key to rebuilding Notre Dame*. NewYorkPost: https://nypost.com/2019/04/17/video-game-could-be-key-to-rebuilding-notre-dame/ adresinden alındı

Yomralıoğlu, T., & Çelik, K. (1994). GIS ? *1.Ulusal Coğrafi Bilgi Sistemleri Sempozyumu* (s. 21-32). Trabzon: Karadeniz Teknik Üniversitesi.

ÖZGEÇMİŞ

Ad-Soyad **:** Abdulkadir BAYTİMUR

ÖĞRENİM DURUMU:

- **Lisans** **:** 2017, İstanbul Teknik Üniversitesi, İnşaat Fakültesi, Geomatik Mühendisliği

MESLEKİ DENEYİM VE ÖDÜLLER:

- 2020- Soft Towel Games'de oyun geliştiricisi.
- 2018-2019 Infotech Bilişim ve İletişim Teknolojileri'nde CBS uzman yardımcısı.