Yazılım Laboratuvarı Altın Toplama Oyunu

Özge POYRAZ
Kocaeli Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği
180202025@kocaeli.edu.tr

Burak Can TEMİZEL
Kocaeli Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği
180202024@kocaeli.edu.tr

***Özet*—Bu çalışmada, nesneye yönelik programlama paradigması, arayüz ögeleri, arama algoritmaları, yol bulma algoritması(a*) ve çeşitli veri yapıları ile bir altın toplama oyunu gerçekleştirdik. Oyun mantığı için thread tabanlı basit bir oyun döngüsü kullandık ve grafik fonksiyonları ile animasyonlar oluşturduk.**

Anahtar kelimeler—Nesneye yönelik programlama, arama algoritmaları, yol bulma algoritmaları, a* algoritması, veri yapıları, liste, matris, swing, grafik programlama

I. GIRIŞ

Gerçekleştirdiğimiz proje her ne kadar bir oyun tanımında olsa da oynanış kısmında kullanıcı ile herhangi bir etkileşim içermemekteydi. Burada asıl nokta oyun içerisindeki otonom karakterlerin birbirleri ile olan çok varyasyonlu etkileşimleriydi. Bu oynanış farklılığı haricinde baktığımızda ise diğer oyunlar gibi belli bir grafik arayüzüne sahip olan ve çeşitli olay örgülerinden sonra yeniden çizdirilen bir yapıda olması gerekiyordu. Bu oyunu gerçekleştirirken projeyi farklı kısımlara ayırmak faydalı olacaktı. Oyun döngüsü ve mantığı, kullanıcı arayüzü ve grafik programlama, dosya işlemleri, veri yapıları ve farklı kıstaslarda çalışan otonom oyuncular için algoritmalar bu kısımları oluşturmaktaydı. Raporun ilerleyen kısımlarında bu kısımların teknik özelliklerine ayrıntılı bir şekilde tekrar değineceğiz.

İlk etapta oynanışın temelini yol bulma algoritmaları oluşturmaktaydı. Karakterler birtakım şartlar altında belli bir veri yapısı içerisinden en uygun altını belirleyip, hareketini gerçekleştirmeliydi. Daha önceki projelerimizde dijkstra algoritmasını kullandığımız için yeni bir kazanım elde etmek amacıyla bu projede a* algoritmasını kullanmaya karar verdik. Başta tüm arama ve yol bulma sistemi, a* algoritması ile oluşturulsa da daha sonra bahsedeceğimiz bazı performans problemlerinden dolayı arama kısmında Manhattan, uzaklığı yol bulma kısmında a* kullanan karma bir implementasyona geçiş yaptık. Daha sonra bu temel üzerinden çeşitli algoritmalarla karakterlere farklı ölçütlerde otonomluk kazandırdık ve grafik programlama ile bunu görselleştirdik.

II. YÖNTEMLER, YAKLAŞIMLAR VE PROGRAM MIMARISI

Bu kısımda programın farklı özelliklerini oluşturmak için kullandığımız araçlar ve yöntemler üzerinde durularak ayrıntılı olarak bilgi verilecektir. Program mimarisi daha detaylı bir şekilde açıklanacaktır.

A. Oyun Döngüsü ve Mantığı

Oyunu gerçeklerken önceki oyun projelerimizden farklı olarak tek bir ana oyun döngüsü yazıp işlemleri gerçekleştirmek yerine bu noktada da karma bir yapı kullanmayı tercih ettik. Menü gibi kullanıcının sürekli etkileşimde bulunmadığı yerlerde event sistemini kullandık. Fiziksel bir etkileşimde bulunulmayan sadece görsellik amacıyla yapılmış menü animasyonlarında sabit kare hızına sahip bir tick ile animasyonları gerçekledik. Oyunun ana döngüsünü ise sürekli çalışan bir thread üzerinde gerçekledik. Bu noktada thread bir döngü içerisinde çalıştığı için gerekli tur kontrollerini gerçekleştirdik ve threadin uyku fonksiyonundan yararlanarak tur tabanlı oynanış sistemini oluşturduk. Bu şekilde threadin uyku hızını ayarlayarak oyunun hızına müdahil olma şansını da elde etmiş olduk.

Tur değişkenine bağlı olarak oyuncular sırası ile ana döngü içerisinde yazıldığı şekilde hareketlerini gerçekleştiriyorlar. Oyuncular farklı kıstaslara sahip benzer özellikleri taşımaktalar. Genel yapı itibarı ile tüm oyuncular ilk olarak oyun başlangıcının tespit edildiği özel bir şart ile ilk hedeflerini seçiyorlar. Ardından hamleye başlamadan önce gerekli kontroller ile hamle maliyetini karşılıyorlar. Hamleyi gerçekleştirmeden önce diğer turlar için hedefledikleri altının alınıp alınmadığını kontrol ederek gerekirse yeniden hedef belirleme işlemlerini gerçekleştiriyorlar. Adım sayısı kadar dönen bir döngü içerisinde a* algoritmasından elde ettiğimiz yol listesi üzerinde ilerliyorlar. Her adımdan sonra altını alıp almadıklarını kontrol ediyorlar. Altın aldıklarında yeniden hedef belirliyorlar. Ayrıca her hareketlerinden sonra bir gizli altının üzerinden geçtilerse onu açığa çıkartıyorlar. Genel olarak oyun mantığı bu şekilde olsa da oyuncuların farklı kıstaslarda çalıştığını söylemiştik. Burada temel farklılığı hedef belirleme fonksiyonları oluşturuyor. Proje içerisinde üç adet farklı hedef belirleme algoritması bulunmakta bunlara daha sonra detaylı bir şekilde değinilecektir. Ayrıca C oyuncusunun gizli altınlarla ilgili yeteneği yüzünden de ek bir metodu bulunuyor. Tüm bu bahsedilen olay örgüsü farklılaştırılmış metotlara sahip oyuncular tarafından tekrar tekrar gerçekleştiriliyor. Ne zaman tüm oyuncuların oyun sonu şartı geldiyse (altınları bittiyse ya da oyunda herhangi bir altın kalmadıysa) oyun sonlanıyor. Bu noktada projede bazı detaylar bize bırakılıdığından ve oyuncular gizli altını göremediği için oyun sonu şartı olarak görünür altınların bitmesini belirledik.

B. Kullanıcı Arayüzü ve Grafik Programlama

Her ne kadar projenin asıl odak noktası arama ve yol bulma algoritmaları olsa da oyun tanımını içerdiği için belli bir görsellik seviyesinde olması gerekiyordu. Biz de bu noktada çeşitli varyasyonlar ile gerekli görselliği yaratmaya çalıştık. Öncelikle Java'nın standart arayüzü olan Swing'in komponentlerinin görsellik konusunda biraz zayıf kaldığını düşündüğümüz için daha önceki projelerde de yaptığımız gibi darkLaf LookAndFeel kütüphanesiyle komponentlerin kalitesini daha modern bir seviyeye getirdik. Seçim ekranını oluştururken ilgili panel komponentlerinin paintComponent() metotlarını override ederek graphics sınıfı üzerinden çeşitli çizimler ile menüde altın yağması ve bulut animasyonları

oluşturduk. Buradaki nesneler için ise oluşturduğumuz MenuGrafigi adlı sınıfı kullandık.

Oyun ekranında grid sistemini koyu ve açık iki tip kare ile çizdirdik. A* algoritmasından elde ettiğimiz yol listesini kullanarak her oyuncunun henüz gerçekleştirmediği hamlesi boyunca gideceği yolu grafik arayüzde çizdirdik ve aynı şekilde hedef belirlediği kareyi kendi renginde çerçeveledik. Böylece oyuncuların hareketlerinin takip edilmesi daha da kolaylaştı. Tüm çizim işlemleri Graphics sınıfı üzerinde primitive çizim fonksiyonları ile gerçekleştirildi. Komponentlerin içine görsel eklenen bir yapı kullanılmadı. Sadece tek bir Panel üzerinde tüm oyun çizim işlemi gerçekleştiriliyor. Bu da hem bellekten çok büyük bir tasarruf hem de çizim hızında çok büyük bir performans sağlıyor.

Oyun hızını ayarlamak için bir adet slider komponenti sahneye eklendi. İzleyicinin oyuncuların altınını takip edebilmesi için ekran üzerinde her oyuncuya altını gösteren gui eklendi. Oyun ekranının tamamının dinamik olmasını istediğimiz için boyut fark etmeksizin tüm boyutlarda oyun alanını pencere yüksekliğine göre ölçekleyen ve ekranın ortasına tam olarak çizdiren ölçek sistemi gerçekleştirdik. Bu şekilde karelerin oranı bozulmadan oyun alanı pencere boyutlarına bağlı olarak dinamik bir şekilde kendisini yeniden boyutlandırıyor. Son olarak oyun sonunda kullanıcıların istatistiklerinin görülmesi bir tablo ekledik. Sabit arayüze sahip formları Netbeans'in Form Editor'ünde gerçekledik. Diğer kısımları el ile oluşturduk. Tüm bu bahsettiğimiz arayüz ve grafik işlemleriyle kullanıcı tarafından göze hoş gelen bir görüntü yakalamaya çalıştık.

C. Farklı Kıstaslarda Çalışan Otonom Oyuncular ve Algoritmaları

Daha önce de bahsettiğimiz gibi her ne kadar oyuncular ortak bir tur patternine göre hareket etseler de bazı metotları gereği farklılıklar bulunduruyorlar. A Oyuncusunun kullandığı, hedefBelirle() , B ve C Oyuncusunun kullandığı maliyetliHedefBelirle(), D Oyuncusunun kullandığı sezgiselMaliyetliHedefBelirle() ve C Oyuncusunun kullandığı gizliAltinlariAcigaCikart() metodları detaylı bir şekilde anlatılacaktır.

Başlangıçta komşu düğümler üzerinden yola çıkılan uzaklığın g olduğu, sezgisel girdi olarak Manhattan uzaklığının h olduğu f = g + h yapısına sahip a* algoritmasını kullanarak tüm işlemlerimizi gerçekleştirmiştik. Kodumuzu yazarken referans olarak rosettecode.org [1] sitesinden ve Daniel Shiffman'ın a* derslerinden [2] faydalandık. Bu sistemde oyuncu, tüm altın listesine tek tek a* algoritmasını uyguluyor daha sonra g uzaklıklarını karşılaştırarak en yakını belirliyordu. İmplementasyonu kolay ve mantıklı olan bu yöntem, sürekli olarak komşu listelerin oluşturulması işlemini tetiklediği için büyük haritalarda bize oldukça zaman kaybettiriyordu. Kompleksliği arttırıyordu. Bundan kaçınmak amacıyla en yakın altını bulma işleminde a* algoritmasını uygulamayı kaldırıp direkt olarak Manhattan uzaklığı ile tespiti gerçekleştirdik. Daha sonra sadece bu kareye a* uygulayarak yine hareket aşamasında kullanacağımız listeyi elde ederek karma bir yapı oluşturduk. Ayrıca AStar sınıfını tekrar oluşturmadan yeniden ilk konuma göre hesaplama yapabileceğimiz bir sifirla() metodu yazarak bellekten de biraz tasarruf sağladık.

A oyuncusu yukarıda bahsettiğimiz Manhattan uzaklığının kullanıldığı en küçüğü bulma döngüsü ve a* yol bulma algoritmasıyla kendisine en yakın olan kareyi bulabiliyor. Fakat B ve C oyuncularında bu durum biraz daha farklı gerçekleşmeli çünkü isterlere göre onlar en karlı oldukları kareye gitmeliler. Burada önemli bir husus ise oyunda adımlar üzerinden değil, hamle üzerinden ilerlememiz gerektiği çünkü bir oyuncunun bir hamlede atabileceği adım sayısı belirlidir. Yani varsayılan adım değerinin 3 olduğu bir senaryoda 2 adım içerisinde 4, 5 veya 6 birim uzaklıktaki karelere gidebiliyor. O yüzden kar/zarar hesabının adım üzerinden değil de hamle sayısı üzerinden gerçekleşmesi gerekiyor. Kaba kodu aşağıdaki gibi olan kazanç formülü ile B ve C oyuncusu için en karlı kare hesaplanıyor. A dan farklı olarak g'leri kıyaslamak yerine uzaklığı bu formüle yerleştirip kazanç miktarını kıyaslayarak en karlı kareyi seçiyorlar.

Math.ceil(manhattanUzakligi / adimSayisi) * hamleMaliyeti – hedeftekiAltinMiktari

Kar miktarının aynı olduğu zamanda B ve C oyuncusu ilgili karelerin Manhattan uzaklığını tekrar değerlendirerek daha yakın olan kareyi, avantajlı olduğu için hedefliyorlar.

D Oyuncusunun ise isterlere göre diğer oyunculardan önce gidemediği kareleri hedef dışı bırakması gerekiyordu. D'nin sezgiselMaliyetliHedefBelirle() fonksiyonu, B ve C'nin maliyetliHedefBelirle() metodunu alıp üzerine bazı özellikler ekliyor. Hedef belirlemeye başlamadan önce diğer oyuncuların mevcut hedeflerine a* algoritmasını uygulayarak yol bilgisini elde ediyor. Daha sonra buradan elde ettiği uzaklık değerini hamle üzerinden değerlendirip daha kısa hamlede gidip gidemediğine bakıyor. Eğer gidemiyorsa birazdan kareleri bulacağı listeden, bu işaretli kareleri çıkartıyor. Böylece kendi arama havuzunu oluşturmuş oluyor.

D. Kullanılan Temel Veri Yapıları

Harita sınıfı, oyunun çizdirilmesini sağlamanın yanında oyunla ilgili en önemli aktif birim olan karelerin de saklandığı bir sınıftır. Burada kareleri saklarken kullandığımız veri yapılarını listeler oluşturuyor. Kareler yapısal değişkenlerle birlikte aynı zamanda çizimle ilgili ögeleri de içlerinde barındırıyorlar. A* algoritmasında kareleri kullanmak çeşitli referans düğümlerin oluşturulması sırasında statik olmayan değişkenlerin tekrar tekrar oluşup bellekte artış yapmasını sağlayacağından harici bir mimari olarak ek bir düğüm sınıfı kullandık. Bu şekilde oldukça düşük boyutlu olan düğüm sınıfı üzerinden veri yapılarımızı kurup daha sonra onu kare nesneleriyle ortak özelliğine göre arama yaparak ilişkilendirebilecektik.

Oyun boyunca herhangi bir engel olmadığı için a* algoritmasını oyun alanıyla aynı büyüklükte maliyetsiz yani sıfır matrisiyle isleme sokmak, sadece yol listesini almak ve hareketi sağlamak bizim için oldukça mantıklı bir seçimdi.

Oyun içindeki tüm arama işlemlerine bütün kareleri sürekli sokmak çok fazla maliyetli olacağı için dinamik bir şekilde çeşitli listeler oluşturulup, koşullara göre içerikleri güncellenip aramalar bu kareler üzerinden yapılıyor. Böylece büyük bir performans artışı yaşanıyor. Örneğin gizli altınlara bakacağınız zaman tüm kareler içerisinden, gizli altın olanlara bakmaktansa kendisine özel referansların bulunduğu listede arama yapmak arama süresini oldukça düşürüyor.

E. Dosya İşlemleri

Projenin içerdiği diğer bir ister ise oyundaki tüm hamlelerin oyunculara özel dosyalar içerisinden verilen çıktılar aracılığıyla takip edilmesiydi. Bunu gerçekleştirirken Java'nın sahip olduğu FileWriter sınıfından yararlandık. Her oyuncu için bu sınıf aracılığıyla yeni bir cikti dosyasi yarattık. Oyun içerisinde oyuncuların gerçekleştirdiği ve takibi sağlayan önemli hamleleri her tur ilgili dosyaya yazdırdık. Daha sonra oyunun bitişiyle birlikte dosya kanallarını sonlandırarak çıktıları oluşturduk.

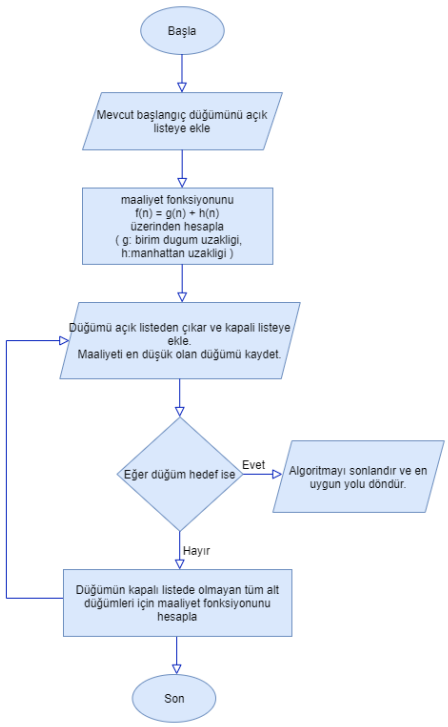
III. KAZANIMLAR

Sadece Manhattan uzaklığını kullanarak ta gerçekleştirebileceğimiz bir proje olmasına karşın, daha önceden deneyimlediğimiz Dijkstra yol bulma algoritmasının yanında bu projede de a* yol bulma algoritmasını inceleme ve anlama şansı elde ettik. Ayrıca oyun alanını dinamik hale getirmek için yazdığımız grafik fonksiyonlarıyla dinamik arayüzler elde etme konusunda daha tecrübelendik.

IV. EKSIKLIKLER

Proje dokümanında bazı noktalar yeteri kadar detaylandırılmadığı ve bize bırakıldığı için belli bir standart olmaksızın karar vermek bazı noktalarda bizi zorladı. Her ne kadar program içerisinde bir hata bırakmasak ta, proje içerisinde bazı şeyleri standartlaştırmak zorunda kaldık. Örneğin oyuncuların başlangıç konumları hep sabit olması, oyunumuz görünür altınlar bittiğinde sonlanması gibi.

V. AKIŞ DIYAGRAMLARI VE YALANCI KOD

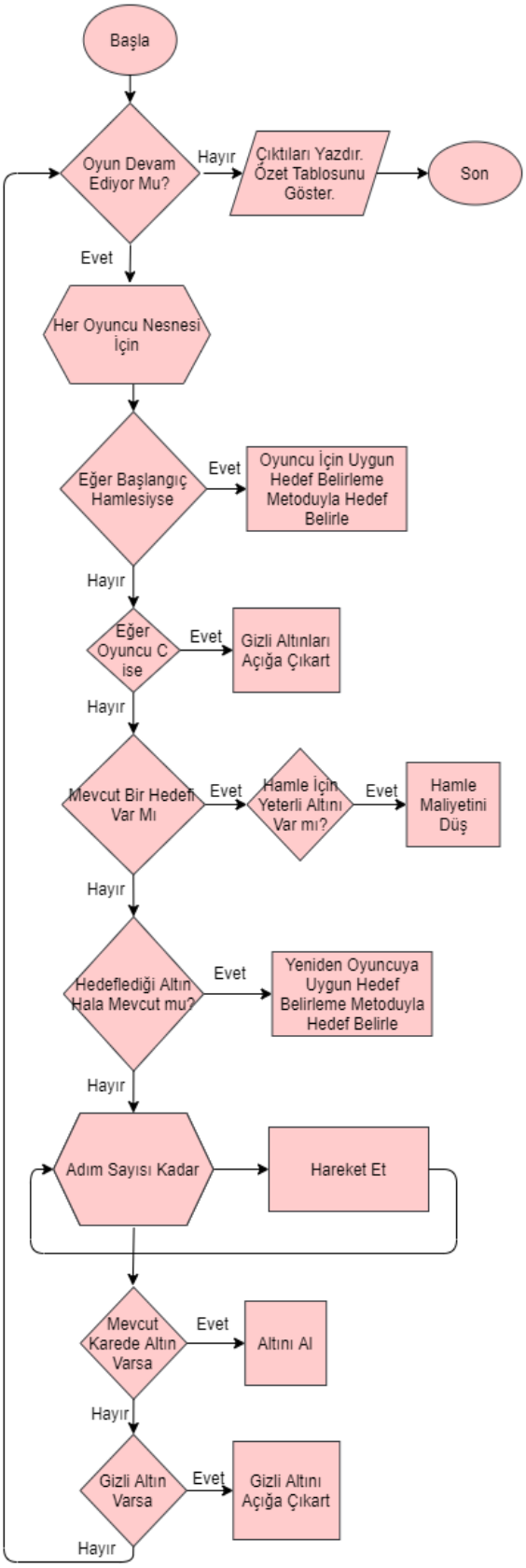


A* Algoritması Akış Şeması

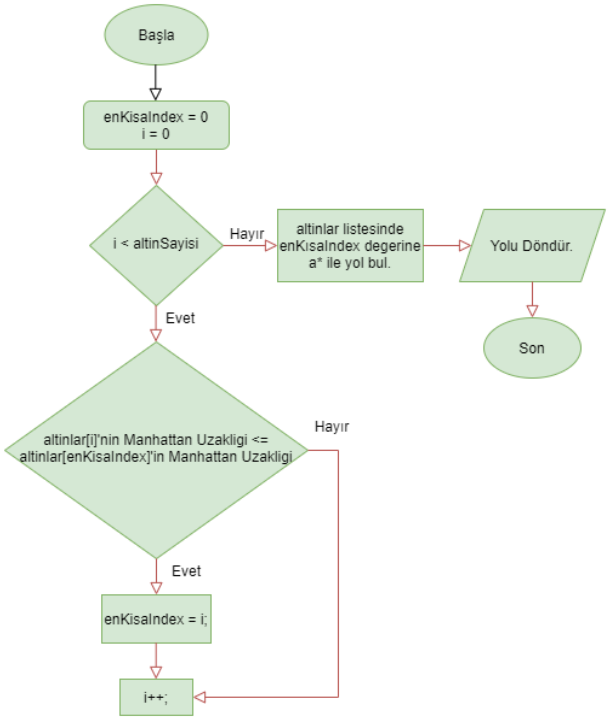
C Oyuncusunun Gizli Altınları Açığa Çıkartma Metodu

```
Açığa çıkacak altın sayısını al adet değişkenine ata.
acigaCikan = 0
Döngü(tekrar = 0 ve eğer tekrar adetten küçükse tekrarı arttır)
        gizliAltinVarMi = yanlış

        Döngü(GizliAltınOlanKareleri dolaş ve kare değişkenine aktar)

                eğer kare.gizliAltın = doğru ise
                        gizliAltinVarMi = doğru
                        Döngüyü Kır

        Eğer gizliAltinVarMi = yanlış ise
                Metodu Sonlandır

        enKisaGizliAltinKare = gizliAltinOlanKarelerin s ıfırıncı Indeksli elemanı

        enKisaGizliAltinYol = Boş Yol Objesi;
        enKisaGizliAltinIndeks = 0;

        Döngü(i = 0 ve i altınOlanKarelerin uzunluğundan küçükse i'yi arttır)

                Eğer altinOlanKarelerin i'nci Indeksinde gizliAltin == doğru ise
                        Eğer(altınOlanKarelerin i'nci Indeksinin Manhattan Uzakligi <= altınOlanKarelerin enKisaGizliAltınIndeks'nci Indeksinin Manhattan Uzakligi)
                                enKisaGizliAltınIndeks = i;
                                enKisaGizliAltınKare = gizliAltınların i'nci Indeksli elemanı

        Döngü(gizliAltinOlanKareleri dolaş ve kare değişkenine aktar)

                eğer kare.x == enKisaGizliAltinKare.x ve kare.y == enKisaGizliAltinKare.y ise
                        kare.gizliAltin = yanlış;
                        kare.altin = doğru;
                        altınOlanKarelere kare elemanını ekle.
                        Dosyaya gizli altının açığa çıkarıldığını yazdır.
                        gizliAltınOlanKarelerden kare elemanını çıkar
                        Döngüyü Kır
```

Yukarıdaki yalancı kod parçası C Oyuncusunun her turun başında gizli altınları açığa çıkarttığı metodun yalancı kodudur.



Oyunun Sadeleştirilmiş Genel İşleyiş Algoritması



A Oyuncusunun Hedef Belirleme Algoritması

VI. Geliştirme Ortamı ve Kullanılan Dil

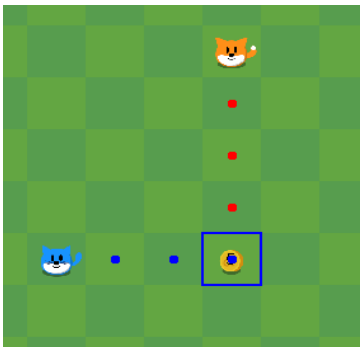
Projeyi Java programlama dilinde Windows işletim sistemi üzerinde gerçekleştirirken, geliştirme ortamı olarak 1.8 JDK konfigürasyonlu Netbeans ve Eclipse idelerini kullandık. Proje Java Maven projesi olup ilgili bağımlılıkları pom.xml dosyasında girilidir.

VII. Programın Genel Yapısı ve Tasarımı

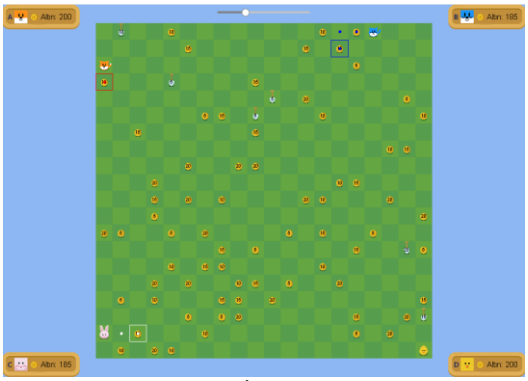
Program temel olarak seçim ekranı, oyun sahnesi ve özet tablo üzere üç adet sahne içeriyor. Her sahne farklı bir tasarıma sahip ve basit bir sahne yöneticisi aracılığıyla sahneler arası geçiş gerçekleştiriliyor.

.

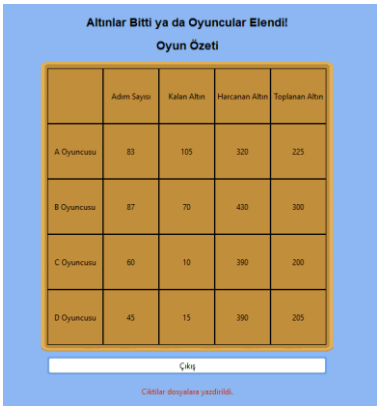
Animasyonlu Oyun Menüsünün Genel Tasarımı



Oyuncu Hareketinin Yakın Görüntüsü



Oyun İçi Tasarım



Oyun Sonu Ekranı

VIII. Deneysel sonuçlar

Daha önce de bahsettiğimiz, yaşadığımız performans problemleri yüzünden yol bulma algoritmalarının karmaşıklığ, bellek ve hız optimizasyonu konusunda uygulamalı olarak deneyim elde etme fırsatı bulduk. Sıkıntı yaşadığımız noktalarda bir sürü optimizasyon gerçekleştirmek zorunda kaldık. Burada elde ettiğimiz verileri karşılaştırıp uygun olan yöntemi kullandık. Uygun algoritmaları belirlerken, tüm algoritmaların performanslarını görselleştirme sitelerinden izleyerek ve diğer yardımcı sebeplerimizle birlikte son olarak uygun olanı seçtik. Bu noktada bol bol gözlem yapma fırsatı elde ettik ve verileri karşılaştırdık.

IX. Sonuç

Bu projeyi gerçekleştirerek etkileşimsiz oyuncuların otonom hareket ettiği bir oyun ortaya çıkartmayı başardık. Çeşitli arama, yol bulma algoritmalarını ve veri yapılarını kullandık, dosya işlemleri gerçekleştirdik ve grafik programlama yaptık.

X. Kaynaklar

(1) rosettacode.org/wiki/A*_search_algorithm (Erişim Tarihi: 20.11.2020)
(2) thecodingtrain.com/CodingChallenges/051.1-astar.html (Erişim Tarihi: 20.11.2020)