SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5495

Određivanje putanje D* algoritmom u virtualnom okolišu

Filip Pavletić

Zagreb, lipanj 2018.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5495

Određivanje putanje D* algoritmom u virtualnom okolišu

Filip Pavletić

Zagreb, lipanj 2018.

Sadržaj

1. Uvod

Određivanje putanje je važan problem u više područja računarstva. Koristi se za mnoge stvari od robotike do video igara. Kada se spomene određivanje puta, većina će se prvo sjetiti A* algoritma. A* je statički algoritam. Jednom kada izračuna optimalni put on je gotov i zaboravlja sve što je do tada radio. Ako je graf na kojem tražimo putanju nepromjenjiv, ovo ponašanje nam odgovara. Problem se javlja kada radimo s promjenjivim grafovima. Primjerice, robot koji se kreće nepoznatim terenom pa na početku nema informacije o cijelom grafu već samo malom podskupu njegovih čvorova (engl. node) ili lik u igri čiji se teren može mijenjati. Kada bismo u tim slučajevima koristili A* morali bi ispočetka računati cijelu putanju svaki put kada se graf promijeni što se može događati vrlo često. Rješenje ovog problema su dinamički algoritmi. Njihov rad ne prestaje nakon što izračunaju početnu putanju već su prisutni dok god agent ne stigne do cilja te su, zbog zadržavanja informacija izračunatih pri određivanju početne optimalne putanje, sposobni relativno brzo reagirati na promjene u grafu. U ovom ćemo radu pogledati nekoliko dinamičkih algoritama te u konačnici implementirati određivanje puta u mreži D* Lite algoritmom.

2. Bitni algoritmi

2.1 Algoritam A*

A* su 1968. godine stvorili Nils Nilsson i Bertram Raphael. Algoritam je dokazano optimalan za nalaženje najkraćeg puta u grafu uz uvjet da je korištena heuristika dopustiva.

```
1
2   A* algoritam
3
4   function AStarSearch(start, successors, goal, heuristic)
5       openList <- [(start, 0)]
6       closedList <- []
7       while openList != [] do
8           n <- removeHead(openList)
9           if ( goal(n[0]) )
10              return true;
11          closedList <- closedList U [n]
12          for m ∈ successors ( n[0] ) do
13              if m' ∈ closedList such that m'[0] == m then
14                  if n[1] + c(n, m) > m'[1] then
15                      continue
16                  else
17                      remove(m', closedList U openList)
18              insertSortedBy((m, n[1] + c(n, m), openList, cost(m) + heuristic(m))
19      return false;
20
```

Slika 2.1: Pseudokod A* algoritma

Kao što možemo vidjeti na slici 2.1 A* koristi kombinaciju cijene prijelaza i heuristike kako bi odredio koje će čvorove istražiti. Algoritam radi tako da dok god nije obradio ciljni čvor uzima najbolji čvor iz liste otvorenih čvorova te iterira po njegovoj listi nasljednika. Ukoliko se nasljednik već nalazi u otvorenoj ili zatvorenoj listi provjeravamo je li nova putanja bolje od one koju imamo spremljenu. Ako je brišemo ga iz lista te s novom cijenom putanje ubacujemo u otvorenu listu. Ukoliko je ovo prvo put da vidimo neki čvor jednostavno ga ubacimo u otvorenu listu.

2.2 LPA*

Lifelong Planning A* je dinamički algoritam. [1]

```
function calculateKey(s)
    return ( min ( cost(s), rhs(s) ) + heuristic(s), min ( cost(s), rhs(s) )

function init()
    openList <- []
    for s ∈ states
        rhs(s) = inf
        g(s) = inf
    rhs(s0) = 0
    insertSortedBy(s0, openList, calculateKey(s0))

function calculateRhs(u)
    rhs = -inf
    for s ∈ predecessors(u)
        if ( cost(s) + costTransition(s, u) < rhs )
            rhs = cost(s) + costTransition(s, u)
    return rhs

function updateState(u)
    if ( u != s0 )
        rhs(u) = calculateRhs(u)
    if ( u ∈ openList )
        remove(u, openList)
    if ( cost(u) != rhs(u) )
        insertSortedBy(u ,openList, calculateKey(u))

function computeShortestPath()
    while ( bestKey(openList) < calculateKey(goal) or rhs(goal) != cost(goal))
        u = removeHead(openList)
        if ( cost(u) > rhs(u) )
            cost(u) = rhs(u)
            for s ∈ successors(u)
                updateState(s)
        else
            g(u) = inf
            for s ∈ successors(u) U [u]
                updateState(s)

function main()
    init()
    computeShortestPath()
    while not reached goal
        move()
        for u, v, newValue ∈ affectedEdges
            updateEdgeCost(u, v, newValue)
            updateState(v)
```

Slika 2.2: Pseudokod LPA* algoritma

Uvodi se nova vrijednost “desna strana” (eng. right hand side) koju ćemo ubuduće zvati rhs. Ta je vrijednost jednaka zbroju cijene prethodnog čvora i cijene prijelaza iz tog čvora u trenutni. Uspoređujući ovu vrijednost s cijenom trenutnog čvora možemo prepoznati promjene u grafu, slika 2.2.

rhs == cijena

Čvor je lokalno konzistentan. U grafu se nisu dogodile promjene koje utječu na ovaj čvor.

rhs > cijena

Čvor je podkonzistentan. U grafu su se dogodile promjene koje otežavaju dolazak do ovog čvora. Čvor je potrebno resetirati i put treba ponovno izračunati.

rhs < cijena

Čvor je nadkonzistentan. U grafu su se dogodile promjene koje olakšavaju dolazak do ovog čvora. Nadkonzistentnost implicira da je nađen najkraći put do trenutnog čvora. Tu ćemo činjenicu koristiti pri implementaciji LPA*.

Na slici 2.2 možemo uočiti kako LPA* otvorenu listu ne sortira direktno na temelju heuristike i cijene čvora već na temelju ključa. Ključ je tuple kojemu su vrijednosti sljedeće:

key[0] = min (cijena, rhs) + heuristika

key[1] = min (cijena, rhs)

Ključevi se uspoređuju tako da se prvo usporede njihovi prvi članovi pa tek ukoliko su oni jednaki uspoređujemo druge.

Zadnja važna napomena vezana uz LPA* tiče se smjera potrage. Za razliku od A*-a LPA* put traži od ciljnog prema početnom čvoru. Time više nema potrebe za držanjem liste zatvorenih čvorova.

3. D* algoritam

3.1 Originalni D*

Originalni D* algoritam smislio je Anthony Stentz 1994. godine. D* je također dinamički algoritam [2]. Omogućava promjenu izračunatog puta ovisno o promjenama u grafu. Ovaj algoritam ne koristi rhs kao parametar pomoću kojega bi mogao prepoznati promjene u grafu, već to radi pomoću ključa i heuristike. Ključ je minimum između heuristike prije modifikacije grafa i svih vrijednosti koje je heuristika poprimila od dodavanja tog čvora u otvorenu listu.

```
D*

function processState()
    u = removeHead(openList)
    if u == null
        return -1
    kold = calculateKey(u)
    if kold < heuristic(u)
        for s ∈ successors(u)
            if heuristic(s) < kold and heuristic(u) > heuristic(s) + cost(u, s)
                backpointer(u) = s
                heuristic(u) = heuristics(s) + cost(u, s)
    if kold == heuristic(u)
        for s ∈ successors(u)
            if tag(s) == NEW or
                    ( backpointer(s) == u and heuristic(s) != heuristic(u) + cost(u, s) ) or
                    ( backpointer(s) != u and heuristic(s) > heuristic(u) + cost(u, s))
                backpointer(s) = u
                insertSortedBy(s, openList, heuristic(s) + cost(u, s))
    else
        for s ∈ successors(u)
            if tag(s) == NEW or
                    ( backpointer(s) == u and heuristic(s) != heuristic(u) + cost(u, s) )
                backpointer(s) = u
                insertSortedBy(s, openList, heuristic(s) + cost(u, s))
            else
                if backpointer(s) != u and heuristic(s) > heuristic(u) + cost(u, s)
                    insertSortedBy(u, openList, heuristic(u))
                else
                    if backpointer(s) != u and heuristic(u) > heuristic(s) + cost(u, s) and
                            tag(s) == CLOSED and heuristic(s) > kold
                        insertSortedBy(s, openList, heuristic(s))
    return getKmin()

function modifyCost(u, s, eval)
    cost(u, s) = eval
    if tag(u) == CLOSED
        insertSortedBy(u, openList, heuristic(u))
    return getKmin()

```

Slika 3.1: Pseudokod D* algoritma

Uspoređujući trenutnu vrijednost heuristike i ključa, slika 3.1, možemo prepoznati čvorove na koje je promjena grafa utjecaja direktno pa tu promjenu propagirati kroz ostale čvorove.

heuristika > ključ

U grafu se dogodila promjena koja je negativno utjecala na dobrotu ovog čvora, klasificiramo ga kao RAISE.

heuristika == ključ

U grafu se dogodila promjena koja je pozitivno utjecala na dobrotu ovog čvora, klasificiramo ga kao LOWER.

3.2 Algoritam D* Lite

D* Lite je moderna interpretacija D* algoritma koja se umjesto na A*-u temelji na LPA* algoritmu [3]. Proširuje ga tako da uz promjene u grafu prati i trenutnu poziciju agenta. To mu omogućuje da, pri promjeni u grafu, novi optimalni put računa iz trenutne pozicije agenta umjesto iz početnog čvora. Da bi cijeli proces funkcionirao potrebno je promijeniti način na koji se u LPA* algoritmu računaju ključevi.

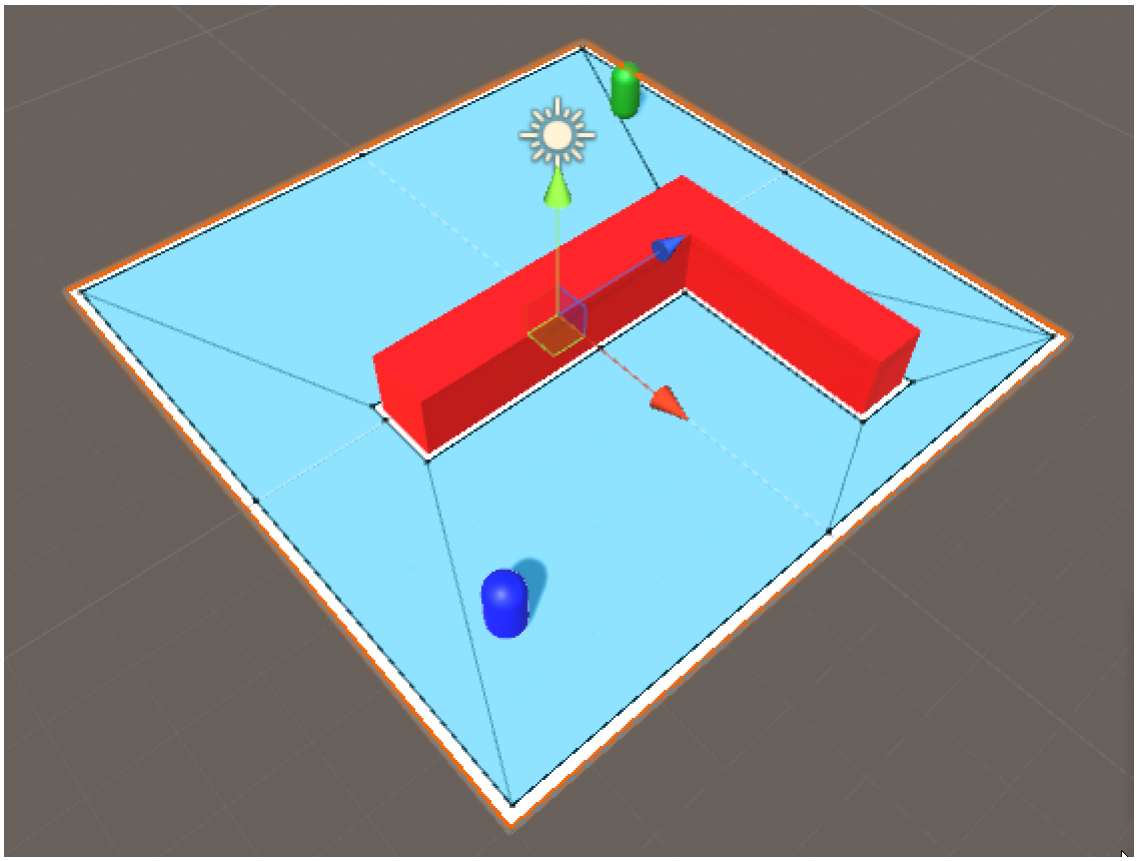
Razlika je u tome što heuristiku više ne računamo između početnog i ciljnog čvora, već između trenutnog i ciljnog čvora. Ovo je moguće implementirati tako da svim čvorovima otvorene liste oduzmemo heurističku procjenu ili istu dodamo udaljenosti između početnog i trenutnog stanja svakom novom članu otvorene liste. Kada bismo odabrali oduzimanje morali bi iterirati po cijeloj otvorenoj listi pri svakoj promjeni u grafu. Stoga biramo dodavanje vrijednosti te heuristike svakom novom čvoru.

D* Lite je uvijek barem jednako brz kao D* te je uz to jednostavniji. Zbog tih razloga D* Lite je skoro u potpunosti zamijenio D* te ćemo u ostatku rada raditi s D* Lite algoritmom.

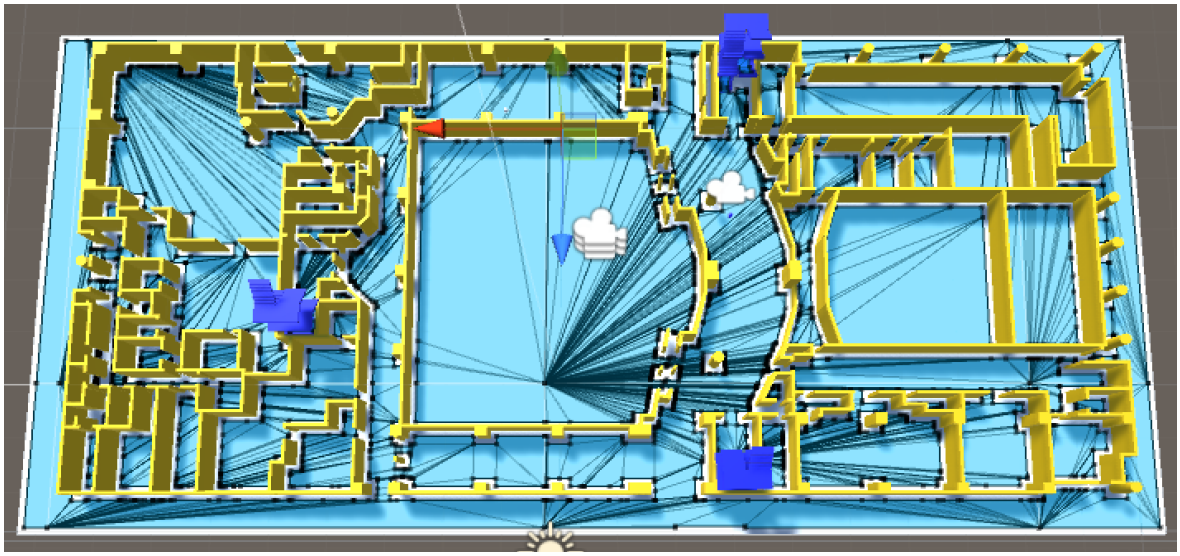
4. Unity komponente

4.1. Komponenta NavMesh

NavMesh je Unityeva komponenta koja omogućuje automatsku generaciju navigacijske mreže iz mreže scene [4]. Podržava izgradnju specifične mreže ovisno o sposobnostima i veličini agenta za kojeg se mreža gradi. Nakon izgradnje, podatci o mreži se mogu izvući u obliku polja vrhova (engl. vertex) i polja poligona (engl. polygon). To omogućuje relativno jednostavnu izgradnju grafa koji će biti korišten u određivanju putanje.



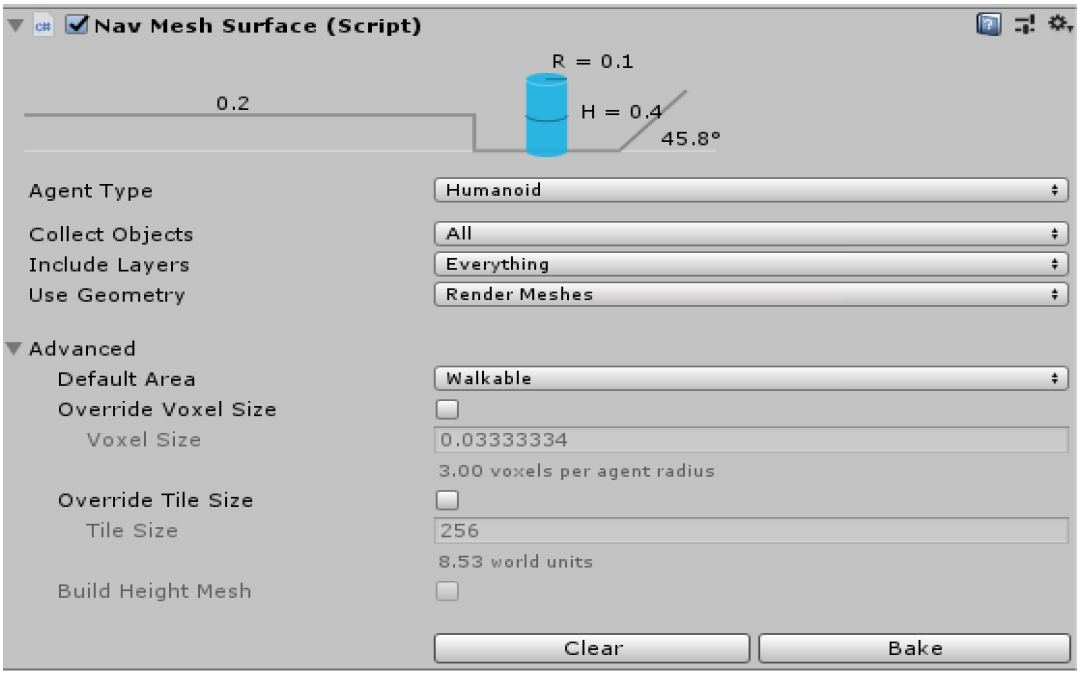
Slika 4.1: Jednostavni NavMesh



Slika 4.2: Složeni NavMesh

Na slikama 4.1 i 4.2 uočiti plavičaste plohe koje prekrivaju mapu, radi se o indikatorima izgleda NavMesha. Agent je sposoban doći do svake točke koja je plavom bojom povezana s trenutnom pozicijom agenta.

4.2. Komponenta NavMeshSurface



Slika 4.3: NavMeshSurface komponenta

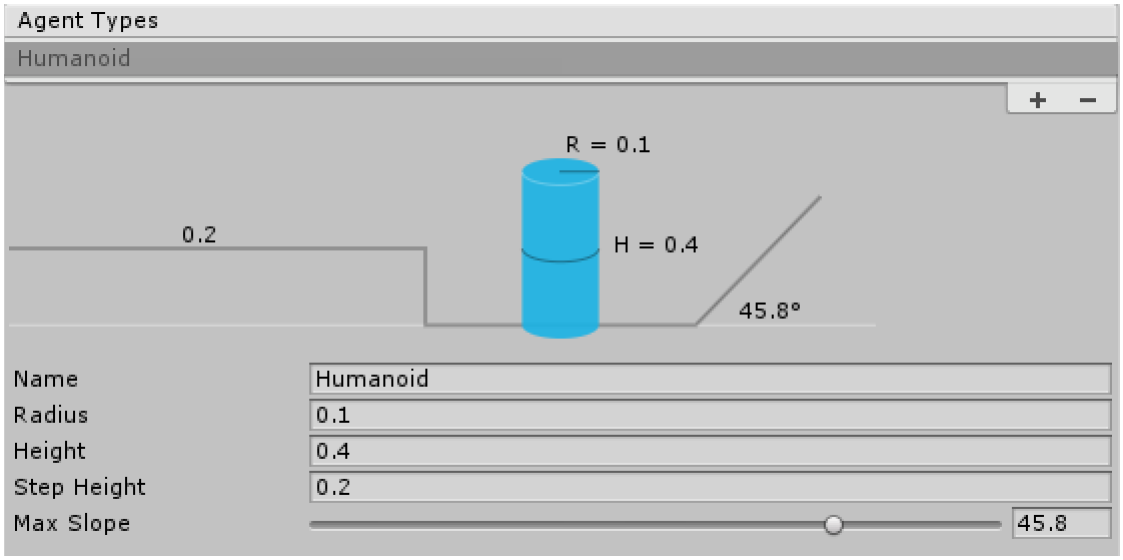
NavMeshSurface, slika 4.3, koristimo za generiranje NavMesha [4]. Kako bi se kreirala što bolja mreža pri generaciji koriste se svojstva agenta.

4.3. Komponenta NavMeshObstacle

NavMeshObstacle je komponenta koja se koristi da bi objekt označili kao prepreku [4]. Statičkim preprekama nije potrebno priključiti NavMeshObstacle, već je NavMesh sposoban sam, koristeći parametre zadanog agenta stvoriti dobru navigacijsku mrežu. NavMeshObstacle je nužno dodati dinamičkim preprekama, to jest preprekama koje se miču, naknadno uklanjaju ili dodaju.

4.4. Komponenta NavMeshAgent

NavMeshAgent je komponenta koja se pridružuje objektima koji moraju imati sposobnost navigacije u navigacijskoj mreži [4]. Mi nećemo koristiti NavMeshAgent-ovu implementaciju određivanja putanje u navigacijskoj mreži već ćemo ga samo koristiti kako bi si olakšali kretanje.



Slika 4.4: Svojstva agenta

Svaki tip agenta ima svoja svojstva [4], slika 4.4. Radius je radijus, a Height visina valjka koji određuje koliko veliki moraju biti prolazi da bi ovaj tip agenta kroz njih uspješno prošao. Step Height je maksimalna visina okomite prepreke na koju se ovaj tip agenta sposoban popeti. Max Slope je najveća kosina po kojoj se tip agenta može kretati.

5. Implementacija

5.1. Mogućnosti implementacije

Prvotna ideja bila je jednostavno koristiti Unityev NavMesh i klasu koja nasljeđuje NavMeshAgent te nadjačava metode vezane uz računanje puta i obradu promjena u grafu. Ovo nažalost nije moguće implementirati zbog modifikatora vidljivosti pridijeljenih tim metodama u NavMeshAgentu.

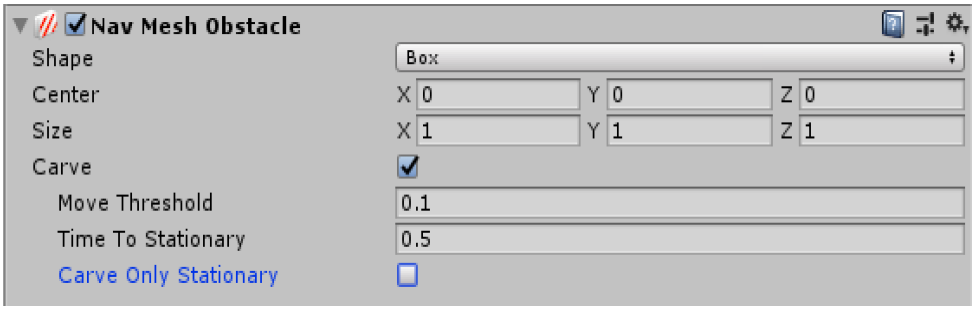
Druga ideja bila je iz NavMesha povući podatke o grafu, u zasebnoj klasi odrediti putanju te ju proslijediti NavMeshAgentu koristeći svojstvo *path*. Ovo također nije moguće zbog limitacija u NavMeshevom API-u.

Treća I konačna ideja bila je iz NavMesha povući podatke o vertexima I poligonima te iz njih izgraditi vlastiti graf. Nakon toga ostatak druge ideje postaje izvediv.

5.2 Konačno rješenje

5.2.1. Stvaranje NavMesha

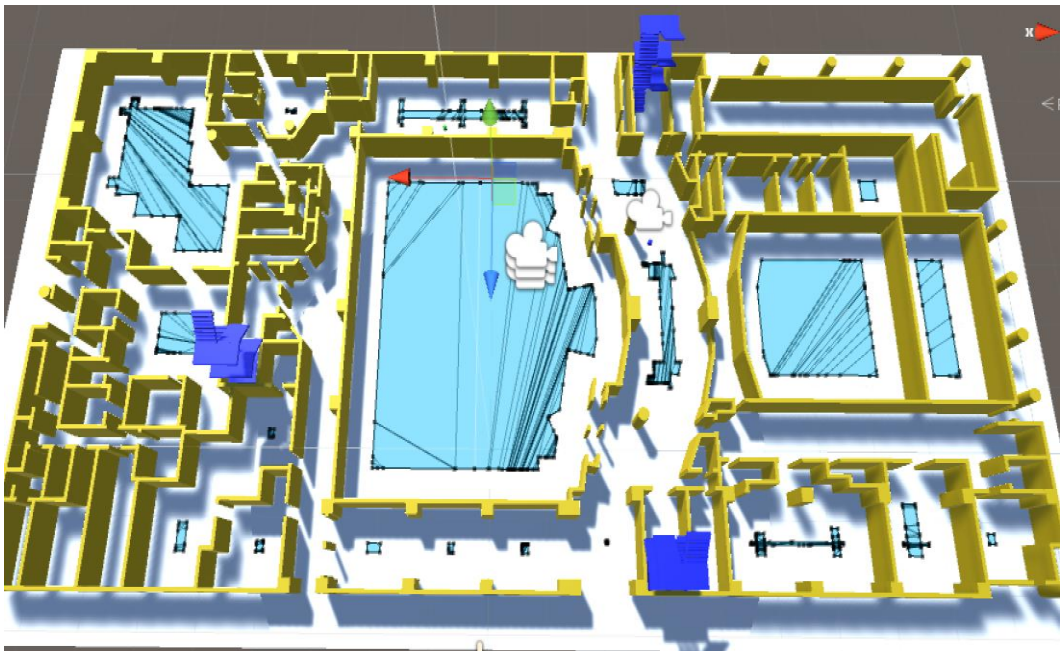
Stvaramo prazni GameObject i dodajemo mu komponentu NavMeshSurface. Ovaj će nam objekt služiti za generaciju NavMesha. Nakon toga stvaramo objekt s modelom koji će nam služiti kao teren.



Slika 5.1: NavMeshObstacle komponenta

Ukoliko na terenu postoje dinamički dijelovi, njima pridodajemo komponentu NavMeshObstacle konfiguriranu kao na slici 5.1. Primijetimo kako je Carve kućica označena. Kada te kvačice ne bi bilo NavMeshSurface bi pri generiranju NavMesha ovu prepreku ignorirao.

Vrijeme je za prvi pokušaj generiranja NavMesha. To radimo tako da u Inspectoru otvorimo GameObject kojemu smo ranije pridružili NavMeshSurface i kliknemo Bake. NavMesh je dobar ukoliko sve površine namijenjene kretanju agenta iznad sebe imaju plavičasti poluprozirni sloj te su povezane. Najčešći problem pri generaciji NavMesha je inkompatibilnost svojstava agenta i modela kojeg koristimo kao teren.



Slika 5.2: Neispravno generirani NavMesh

Na slici 5.2 vidimo situaciju u kojoj je agent preširok da bi se normalno kretao hodnicima našeg modela. Ovaj se problem rješava podešavanjem svojstava agenta dok ne dobijemo traženi NavMesh.

5.2.2. Generiranje grafa iz NavMesha

Graf se generira pri pokretanju programa. NavMesh nam nudi metodu CalculateTriangulation(). Ta metoda vraća NavMeshTriangulation iz kojega možemo dobiti polje vrhova i polje poligona u kojemu su poligoni zadani kao kombinacija indeksa 3 vrha.

```
1
2
3       IList<Node> nodes = new List<Node>();
4       IDictionary<Vector3, Node> vertexToNode = new Dictionary<Vector3, Node>();
5       Node[,] polygons = new Node[indices.Length / 3, 3];
6       IDictionary<Node, IList<int>> nodeToPolygonIndices = new Dictionary<Node, IList<int>>();
7
8       foreach (Vector3 vertex in vertices){
9           Node tmp = new Node(vertex);
10          nodes.Add(tmp);
11          vertexToNode[vertex] = tmp;
12      }
13
14      //populate polygons and vertexToPolygonIndices
15      for (int i = 0; i < indices.Length; i += 3){
16          for (int j = 0; j < 3; j++){
17              Node polygonNode;
18              vertexToNode.TryGetValue(vertices[indices[i + j]], out polygonNode);
19              polygons[i / 3, j] = polygonNode;
20              if (!nodeToPolygonIndices.ContainsKey(polygonNode)){
21                  nodeToPolygonIndices.Add(polygonNode, new List<int>());
22              }
23              IList<int> polygonNodePolygonIndices;
24              nodeToPolygonIndices.TryGetValue(polygonNode, out polygonNodePolygonIndices);
25              polygonNodePolygonIndices.Add(i / 3);
26          }
27      }
28
29      //connect nodes
30      foreach (Node node in nodes){
31          IList<int> polygonIndices;
32          nodeToPolygonIndices.TryGetValue(node, out polygonIndices);
33          foreach (int polygonIndex in polygonIndices){
34              for (int i = 0; i < 3; i++){
35                  Node connectedNode = polygons[polygonIndex, i];
36                  if (node.Equals(connectedNode)){
37                      continue;
38                  }
39                  node.connectedNodes.Add(connectedNode);
40              }
41          }
42      }
43
44      //calculate travel cost
45      foreach (Node node in nodes){
46          foreach (Node connectedNode in nodes){
47              node.SetCost(connectedNode, CalculateTransitionCost(node, connectedNode));
48          }
49
50      }
```

Slika 5.3: Izvorni kod za generiranje grafa

Prvi je korak pretvaranje polja vrhova u listu čvorova, slika 5.3. Čvorovi u ovom trenutku nemaju nikakve vrijednosti osim pozicije vrha iz kojeg su stvoreni. Sljedeći je korak izgradnja poligona. Uzimamo polje indeksa i dijelimo ga u dijelove veličine tri. Iteriramo po svakom novostvorenom dijelu i

povezujemo indeks poligona s poljem njegovih čvorova. U ovom ćemo koraku također izgraditi mapu koja povezuje čvor sa svim poligonima čiji je on dio. Ta nam je mapa potrebna kasnije. U trećem koraku povezujemo čvorove tako da međusobno povežemo čvorove svakog ranije stvorenog poligona. Preostaje nam samo dobiti cijene prijelaza koje su u našem slučaju euklidske udaljenosti.

Klasu koja obavlja stvaranje grafa moramo proširiti s još dvije funkcionalnosti.

Prvo, klasa mora podržavati dodavanje novih čvorova u graf. Taj zahtjev postoji zbog mogućnosti da su agentov početni i ciljni vrh van skupa vrhova koje smo primili od NavMesha.

```
1
2   Node node = new Node(position);
3           IList<int> polygonIndices = nodeToPolygonIndices[getClosestNode(node)];
4
5           int closestPolygonIndex = -1;
6           double closestPolygonDistance = double.PositiveInfinity;
7           foreach ( int polygonIndex in polygonIndices)
8           {
9               double testPolygonDistance = 0;
10              for ( int i = 0; i < 3; i++)
11              {
12                  testPolygonDistance += CalculateDistance(
13                      node.position, polygons[polygonIndex, i].position);
14              }
15              if ( testPolygonDistance < closestPolygonDistance)
16              {
17                  closestPolygonIndex = polygonIndex;
18                  closestPolygonDistance = testPolygonDistance;
19              }
20          }
21
22          for ( int i = 0; i < 3; i++)
23          {
24              Node connectedNode = polygons[closestPolygonIndex, i];
25              double cost = CalculateTransitionCost(node, connectedNode);
26              node.connectedNodes.Add(connectedNode);
27              node.SetCost(connectedNode, cost);
28              connectedNode.connectedNodes.Add(node);
29              connectedNode.SetCost(node, cost);
30          }
31
32          nodes.Add(node);
33          return node;
```

Slika 5.4: Izvorni kod za dodavanje novog čvora u graf

Na slici 5.4 vidimo kod koji pronalazi poligon najbliži vrhu u koji želimo dodati čvor. Udaljenost poligona određujemo kao zbroj udaljenosti njegovih čvorova. Novi čvor povežemo s čvorovima pronađenog poligona.

Drugo, klasa mora podržavati promjene cijena prijelaza u grafu.

```
public void ObstacleAdded(Vector3 position, double size)
    {
        IList<Node> possiblyBlockedNodes = new List<Node>();
        IList<Node[]> blockedNodePairs = new List<Node[]>();
        IList<Node> checkedNodes = new List<Node>();
        possiblyBlockedNodes.Add(getClosestNode(position));
        while ( possiblyBlockedNodes.Count != 0)
        {
            Node possiblyBlockedNode = possiblyBlockedNodes[0];
            possiblyBlockedNodes.RemoveAt(0);
            checkedNodes.Add(possiblyBlockedNode);
            if ( IsInObstacle(position, size, possiblyBlockedNode.position))
            {
                foreach ( Node tmp in possiblyBlockedNode.connectedNodes)
                {
                    blockedNodePairs.Add(new Node[] { possiblyBlockedNode, tmp });
                    tmp.SetCost(possiblyBlockedNode, double.PositiveInfinity);
                    if ( checkedNodes.Contains(tmp))
                    {
                        continue;
                    }
                    possiblyBlockedNodes.Add(tmp);
                }

            }
        }

        foreach ( IGraphChangeListener l in listeners)
        {
            l.OnGraphChange(blockedNodePairs);
        }
    }
```

Slika 5.5: Izvorni kod za osvježavanje podataka o grafu

Metoda, slika 5.5, se poziva iz generatora prepreka i prima parametre o poziciji i veličini stvorene prepreke. Lociramo čvor najbliži postavljenoj prepreci. Dodamo ga u listu čvorova koji bi mogli biti blokirani. Provjerimo je li blokiran preprekom. Ukoliko je, ubacimo sve čvorove povezane s njim u listu čvorova koji bi mogli biti blokirani, a njega prebacimo u listu blokiranih čvorova. Ukoliko nije uklanjamo ga iz liste čvorova koji bi mogli biti blokirani. Ponavljamo dok god lista možda blokiranih čvorova nije prazna. Kada ta petlja završi, pomoću listenera agentima javljamo da je došlo do promjene u grafu te točno na koje je čvorove promjena utjecala.

5.2.3. Implementacija D* Lite algoritma

Implementacija D* Lite algoritma korištena u ovom radu vrlo je slična originalnoj implementaciji [3]. Jedina razlike je što je u ovom radu Main funkcija, zbog pojednostavljenja izvedbe, podijeljena na dva dijela, Move koji određuje putanju i OnGraphChange koji se brine o osvježavanju podataka.

```
1
2   internal double Heuristic(Node start, Node finish)
3       {
4           // simple Euclidian distance boosted in case of different floors
5           if ( start.position.y.Equals(finish.position.y))
6           {
7               return Math.Sqrt(
8                   Math.Pow(start.position.x - finish.position.x, 2) +
9                   Math.Pow(start.position.z - finish.position.z, 2));
10          } else
11          {
12              Vector3 closestStairs = graphGenerator.getClosestStairs(
13                  start.position.x, start.position.z);
14              double distanceToStairs = Math.Sqrt(
15                  Math.Pow(start.position.x - closestStairs.x, 2) +
16                  Math.Pow(start.position.z - closestStairs.z, 2));
17              double distanceFromStairs = Math.Sqrt(
18                  Math.Pow(finish.position.x - closestStairs.x, 2) +
19                  Math.Pow(finish.position.z - closestStairs.z, 2));
20              return distanceToStairs + distanceFromStairs +
21                  climbCoefficient * Math.Abs(start.position.y - finish.position.y);
22          }
23
24      }
```

Slika 5.6: Izvorni kod metode za izračun heuristike

U implementaciji koristimo jednostavnu heuristiku baziranu na Euklidskoj udaljenosti, slika 5.6. Kada su ciljno i trenutno stanje na istom katu, računa se međusobna udaljenost na x i z osima. Kada su na različitim katovima nalazimo stubište najbliže početnom čvoru te računamo udaljenosti od tog stubišta do početnog i ciljnog čvora te njihovom zbroju dodajemo umnožak razlike u visini i koeficijenta težine penjanja. Koeficijent težine penjanja je polje otvoreno Unityu.

```
internal void UpdateNode(ExtendedNode node)
{
    if (!node.node.Equals(start.node))
    {
        double minRhs = double.PositiveInfinity;
        foreach (Node succ in node.node.connectedNodes)
        {
            minRhs = nodeToExtendedNode[succ].cost + node.node.GetCost(succ);
        }
        node.rhs = minRhs;
    }

    int nodeIndex = openList.IndexOfValue(node);
    if (nodeIndex != -1)
    {
        openList.RemoveAt(nodeIndex);
    }

    if (node.cost != node.rhs)
    {
        openList.Add(GetKey(node), node);
    }
}
```

Slika 5.7: Izvorni kod UpdateNode metode

Metoda UpdateNode, slika 5.7, se koristi kako bi nakon promjena u grafu osvježile rhs te ukoliko je nova vrijednost rhs-a različita od stare dodala čvor u listu. Nova rhs vrijednost jednaka je najmanjem zbroju cijene čvora i cijene tranzicije u trenutni čvor među čvorovima nasljednicima trenutnog čvora.

```
1
2       internal void ComputeShortestPath()
3       {
4           while ( keyComparer.Compare(openList.Keys[0], this.GetKey(start)) > 0
5               || start.rhs != start.cost)
6           {
7               double[] bestKey = openList.Keys[0];
8               ExtendedNode bestNode = openList[bestKey];
9               openList.RemoveAt(0);
10
11              double[] updatedKey = GetKey(bestNode);
12
13              if ( keyComparer.Compare(bestKey, updatedKey) < 0)
14              {
15                  openList.Add(updatedKey, bestNode);
16              } else if ( bestNode.cost > bestNode.rhs)
17              {
18                  bestNode.cost = bestNode.rhs;
19                  foreach (ExtendedNode pred in bestNode.GetPredecessors(start, this))
20                  {
21                      UpdateNode(pred);
22                  }
23              } else
24              {
25                  bestNode.cost = double.PositiveInfinity;
26                  UpdateNode(bestNode);
27                  foreach (ExtendedNode pred in bestNode.GetPredecessors(start, this))
28                  {
29                      UpdateNode(pred);
30                  }
31              }
32          }
33      }
```

Slika 5.8: Izvorni kod ComputeShortestPath metode

Na slici 5.8 prikazana je metoda ComputeShortestPath. Njena svrha je izračun cijene čvorova iz perspektive cilja. U otvorenoj listi se na početku nalaze čvorovi promijenjeni u zadnjoj promjeni grafa, ili ako je ovo prvi puta da pozivamo ovu metodu ciljni čvor. Metoda izvlači čvorove s vrha liste sortirane po ključevima dok ne procesira čvor u kojem se agent trenutno nalazi. Akcije obavljene u petlji ovise o svojstvima čvorova. Ukoliko je ključ spremljen u listi manji od stvarnog ključa, čvor vraćamo u listu s novim ključem. Inače provjeravamo je li cijena veća od rhs vrijednosti. Ukoliko je, cijenu čvora postavljamo na rhs vrijednost te pozivamo UpdateNode metodu za svakog od predaka trenutnog čvora. Inače cijenu trenutnog čvora postavljamo na beskonačno, zovemo UpdateNode za trenutni čvor te zovemo UpdateNode za pretke. Pozivi UpdateNode metode nad predcima nam omogućavaju propagaciju promjena u grafu.

```
internal Node Move()
{
    ExtendedNode lastNode = start;
    if (start.rhs.Equals(double.PositiveInfinity))
    {
        return null;
    }
    ExtendedNode bestNode = null;
    double bestNodeValue = double.PositiveInfinity;
    foreach ( ExtendedNode n in start.GetSuccessors(start, this))
    {
        double nodeValue = n.node.GetCost(start.node) + n.cost;
        if ( nodeValue < bestNodeValue)
        {
            bestNode = n;
            bestNodeValue = nodeValue;
        }
    }
    start = bestNode;
    return start.node;
}
```

Slika 5.9: Izvorni kod Move metode

Move metoda, prikazana na slici 5.9, vraća čvor koji bi agenta najbolje pomakao prema cilju te osvježava trenutnu poziciju agenta u memoriji D* Litea. Metoda pretpostavlja da između zadnjeg poziva ComputeShortestPath i poziva ove metode nije došlo do promjena u grafu. Metodu nikada ne pozivamo iz algoritma već ju poziva agent kada želi odrediti koji je sljedeći vrh u koji treba otići.

```
public void OnGraphChange(IList<Node[]> affectedNodePairs)
{
    km += Heuristic(start.node, goal.node);
    ExtendedNode lastNode = start;
    foreach (Node[] affectedNodePair in affectedNodePairs)
    {
        UpdateNode(nodeToExtendedNode[affectedNodePair[0]]);
    }
    ComputeShortestPath();
}
```

Slika 5.10: Izvorni kod OnGraphChange metode

Na slici 5.10 vidimo OnGraphChange metodu. Ta je metoda implementacija OnGraphChange metode listenera spomenutog u poglavlju 5.2.2. OnGraphChange iterira kroz promijenjene bridove (eng. edge) prikazane u obliku para čvorova te nad prvim čvorom poziva UpdateNode. Ovime se osvježavaju rhs vrijednosti svih čvorova na koje je promjena grafa direktno ili indirektno utjecaja. Konačno, poziva se ComputeShortestPath kako bi koristeći nove rhs vrijednosti osvježili cijene čvorova.

5.2.4. AgentController

Zadnji dio implementacije je AgentController. On kontrolira interakciju agenta s ostatkom svijeta.

```
1   void Start () {
2       graphGenerator = GameObject.FindGameObjectWithTag("GraphGenerator").GetComponent<GraphController>();
3       dStarPathfinder = new DStarPathfinder(graphGenerator, transform.position, destination);
4   }
5
6   void Update () {
7       if (hasTarget && navMeshAgent.isStopped)
8       {
9           Node next = dStarPathfinder.Move();
10          if (next == null)
11          {
12              hasTarget = false;
13          } else
14          {
15              navMeshAgent.SetDestination(next.position);
16          }
17      }
18  }
```

Slika 5.11: Izvorni kod metoda AgentControllera

Specifičnije, kao što možemo vidjeti na slici 5.11, pri pokretanju programa brine se o inicijalizaciji servisa za određivanje puta te se nakon što je agentu zadan cilj brine o tome da do tog cilja dođe.

6. Rezultati

Za svrhu testiranja implementacije koristili smo dva modela terena. Testovi su se izvodili na računalu sa procesorom Intel Core i5 4210H, grafičkim procesorom Nvidia GTX 860m i 8 GB RAM-a.

Jednostavni model, slika 4.1, ima svega nekoliko poligona i funkcionirao je kao osnovni test na kojemu je bilo lako pratiti rad algoritma. Izvođenje algoritma za izgradnju grafa na ovom primjeru traje manje od sekunde. Vrijeme trajanja izvođenja D* Lite algoritma teže je pratiti zbog toga što nakon slanja podataka o sljedećem čvoru do kojega se agent treba kretati algoritam čeka da agent dođe do novog čvora prije no nastavi svoj rad. Iako nemamo točne brojke, bitno je napomenuti da algoritam radi u stvarnom vremenu.

Složeni model, slika 4.2, im vrlo velik broj poligona. Neefikasnosti trenutne implementacije postaju očite pri stvaranju grafa. Taj proces traje više od 2 minute. Promjena na vremenu rada D* Lite algoritma nije ni približno toliko drastična. Algoritmu za izračun putanje na početku treba malo više od sekunde. Izmjena putanje zbog promjena u grafu i dalje radi u stvarnom vremenu.

7. Zaključak

D* algoritmi omogućavaju značajno ubrzanje određivanja optimalnih putanja nakon promjene prostora po kojem se krećemo. Unatoč tome, u slučajevima gdje se radi s prostorima koji se ne mijenjaju često, povećani opći troškovi u odnosu na A* algoritam mogu značiti ukupno usporenje sustava. D* algoritmi najbolje dolaze do izražaja kada ih se upari s prostorima čije nam karakteristike nisu poznate ili se učestalo mijenjaju.

Najveći izazov pri implementaciji D* sustava je točno i brzo praćenje promjena u prostoru. To vrijedi i za ovu implementaciju. Cilj budućeg razvoja bio bi poboljšanje načina otkrivanja novih prepreka i omogućavanje otkrivanja prečaca nastalih uklanjanjem prepreka postavljenih tijekom rada algoritma. Uz to bilo bi interesantno razmotriti mogućnost zamjene Unityevog NavMesha vlastitom implementacijom. Time bi mogli pojednostaviti proces generiranja grafa i osjetljivo smanjiti vrijeme potrebno za inicijalizaciju sustava.

8. Literatura

[1] Sven Koenig, Maxim Likhachev, and David Furcy, »Lifelong Planning A*« [Mrežno]. Available:

https://www.cs.cmu.edu/~maxim/files/aij04.pdf

[2] Anthony Stentz, »Optimal and Efficient Path Planning for Partially-Known Environments« [Mrežno]. Available:

http://www.frc.ri.cmu.edu/~axs/doc/icra94.pdf

[3] Sven Koenig and Maxim Likhachev, »D* Lite« [Mrežno]. Available:

http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

[4] »Unity - Scripting API - Manual [Mrežno]«. Available:

https://docs.unity3d.com/ScriptReference/

9. Sažetak

Određivanje putanje D* algoritmom

u virtualnom prostoru

Ovaj rad obrađuje određivanje putanje u dinamičkom virtualnom prostoru koristeći D* algoritam. Objašnjen je način rada D* i D* Lite algoritma i dana je usporedba s tradicionalnim statičkim algoritmima. Ukratko su opisane prednosti dinamičkih pri radu s dinamičkim prostorom. Ponuđen je način implementacije D* Lite algoritma u Unity grafičkom pogonu. Na kraju su iskazane situacije u kojima je poželjno koristiti D*, mane trenutne implementacije i načini na koje bi se te mane u daljnjem razvoju mogle ispraviti.

Ključne riječi: određivanje putanje, D*, D* Lite, NavMesh, Unity, C#

Pathfinding in virtual environments using D*

This paper deals with pathfinding in dynamic virtual environments using the D* algorithm. We take a look at how D* and D* Lite work and compare them to traditional static algorithms. A quick summary of the advantages of dynamic algorithms when dealing with dynamic environments is provided. A Unity based D* Lite implementation is included. Finally, we discuss situations where D* should be used, assess the flaws of our implementation and provide suggestions on how to improve it in future development.

Keywords: pathfinding, D*, D* Lite, NavMesh, Unity, C#