Универзитет у Београду

Математички факултет

МАСТЕР РАД

Симулација развоја града

Аутор:
Дејан Живковић,
1032/2013

Ментор:
Миодраг Живковић

април, 2015.

Садржај

1 Увод

Циљ овог рада је симулација развоја града, са инфраструктуром, зградама и становницима. Инспирација потиче од игре Симсити[1] (енг. Simcity) у којој корисник има могућност да направи и усмерава развој читавих градова. Градови су данас веома сложени системи са великим бројем становника и симулација реалистичног живота у њима је немогућа, стога је потребно одредити границу комплексности система. Реализован градски систем је доста једноставнији од градског система у Симситију, управо због преобимности. Један од главних циљева је био ефикасна реализација кретања многобројних становника по градској саобраћајној мрежи. Коришћен је један прецизан и брз алгоритам за проналажење путева по графу, модификован за потребе овог пројекта. Рад истражује како се сложеност израчунавања путања повећава са растом димензија града. Велики број становника се креће по граду током читавог трајања симулације, па би добар распоред зграда смањио дужину њихових путања. На основу неких индикатора које нуде становници симулације може се приметити да ли је распоред зграда лош. Још један аспекат на којем се рад базира је тродимензинални графички приказ који омогућава интеракцију са градом и приказ свих неопходних података. Програм нуди више различитих приказа који користе разне методе постпроцесирања слике.

Рад може да се примени као подлога за проучавање алгоритама за кретање по мрежи. Може, уз одређене модификације, да служи и као алат за пројектовање распореда зграда по граду и за тестирање ефикасности распореда.

У поглављу OpenGL је описан графички систем који је коришћен за исцртавање симулације. Наредно поглавље описује два алгоритма за проналажење путева по правоугаоној мрежи. Након тога следи приказ класне структуре програма и водич за употребу програма. На крају је извршена анализа перформанси алгоритма A*.

[1] http://www.simcity.com/en_US/game/info/what-is-simcity

2 OpenGL

2.1 Увод

OpenGL је програмски интерфејс за приступање својствима графичког хардвера.[9] Силикон графикс (енг. Silicon graphics) је направио прву верзију 1994. године. Од тада је настало много верзија, као и много библиотека за лакше креирање апликација заснованих на њему. Верзија 2.1, која се овде користи, садржи преко 500 команди помоћу којих се праве објекти, слике и операције неопходне за функционисање интерактивних тродимензионалних апликација. OpenGL је независан од хардвера, и имплементиран је за многе различите хардверске системе. Не представља библиотеку за директно моделовање тродимензионалних објеката, већ је неопходан додатни рад да се то реализује. Комплексне тродимензионалне слике се генеришу дефинисањем једноставних геометријских примитива - тачака, линија, троуглова и одређивањем њихових карактеристика и веза. OpenGL је дизајниран као клијент-сервер систем, где се апликација коју корисник пише сматра клијентом, а сервером имплементација коју је направио произвођач графичког хардвера.

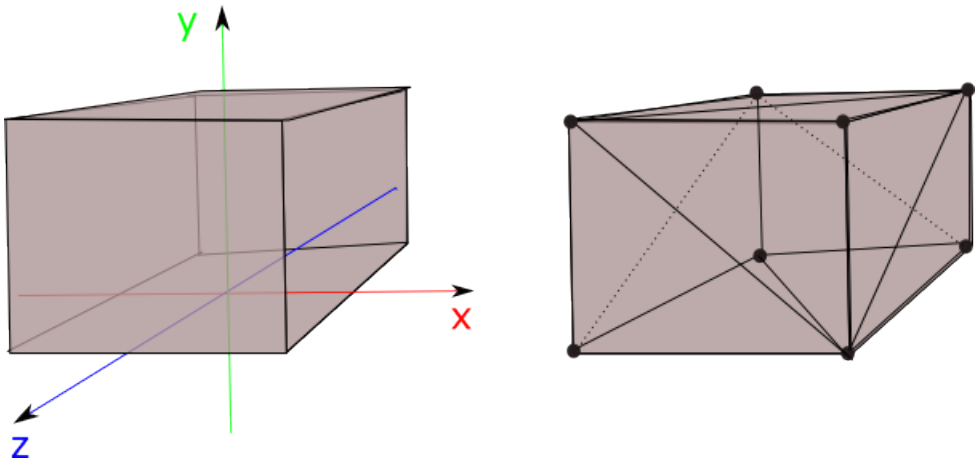
Како OpenGL формира дводимензионални приказ тродимензионалне сцене? На Слици 1 је приказ процеса.

Слика 1: Процес креирања слике кроз OpenGL

Клијент не може да програмира све фазе. Неке фазе, као што је растеризација, OpenGL обавља сам. У ранијим верзијама (до верзије 2.0), добар део овог процеса је био фиксиран, тј. програмиран од стране сервера. Међутим, са новијим верзијама, више етапа процеса је постало програмабилно, што је довело до веће флексибилности (али и сложености) читавог процеса за клијента. У наставку је објашњен процес са Слике 1 кроз исцртавање коцке.

2.2 Подаци о теменима

Да би се потпуно специфицирали улазни подаци о теменима, потребно је дефинисати *позиције темена*, *нормале*[2], *боје* и евентуално позиције *текстура*[3] везане за та темена. На Слици 2 лево је приказана коцка која би требало да се исцрта.



Слика 2: лево: коцка која се црта, десно: коцка након подељене површине

Постоји више начина да се представе улазни подаци. Овде је описан један од њих. Користе се OpenGL низови који чувају податке о позицијама, нормалама, и бојама темена (за сваку од ових категорија по један). Нека је низ позиција темена P, низ нормала N, а низ боја C. Површина изабраног објекта (коцке) се подели на суседне троуглове (Слика 2 десно). Подаци о сваком темену сваког троугла се надовезују на горе наведене низове на следећи начин. Нека је T једно од тих темена. Нека је његова позиција у симулацији вектор (x, y, z), вектор нормале[4] (n_x, n_y, n_z), а интензитет боје вектор (r, g, b, a). Нека су n, m, k текући индекси низова P, N, C. Тада је

$$P_{n+1} = x, \quad P_{n+2} = y, \quad P_{n+3} = z$$

$$N_{m+1} = n_x, \quad N_{m+2} = n_y, \quad N_{m+3} = n_z$$

$$C_{k+1} = r, \quad C_{k+2} = g, \quad C_{k+3} = b, \quad C_{k+4} = a$$

[2] Дефинисање нормала је неопходно због рачунања коначне боје објекта, под утицајем осветљења. Ово је детаљније објашњено касније.

[3] Текстуре су објекти који садрже слике.

[4] Вектор нормале за теме се бира тако да буде уперен ка спољашњој страни троугла коме теме припада.

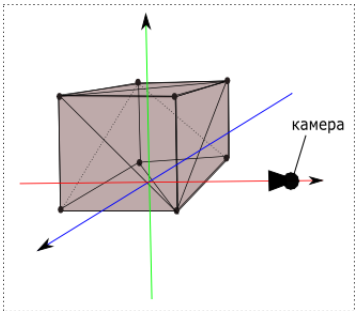
Након овога су подаци о теменима спремни и чувају се у низовима P, N, C. Ови низови се користе у даљем процесу исцртавања преко посебних променљивих (*handle*) које је серверски део издвојио за њих.

2.3 Процесор темена

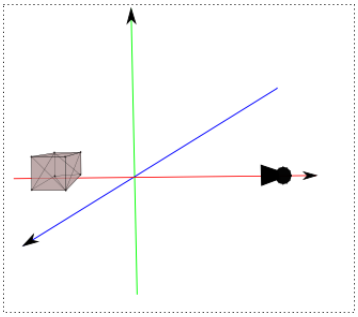
За процесор темена (енг. *vertex shader*) се пишу мали мали програми у језику *GLSL (GL shading language)*. Овај језик има синтаксу сличну језику C++, а омогућава једноставан рад са матрицама и векторима, јер је то у овој фази кључно. Кôд написан у GLSL-у покреће процесор темена на графичкој процесоркој јединици, која је специјализована за брзе операције са матрицама и векторима реалних бројева. У овој фази могуће је урадити додатне трансформације са теменима, бојама и нормалама. Конкретно, у апликацији се рачунају коначне позиције темена у координатном систему камере, као и боје тих темена под утицајем осветљења.

2.3.1 Рачунање коначних позиција темена

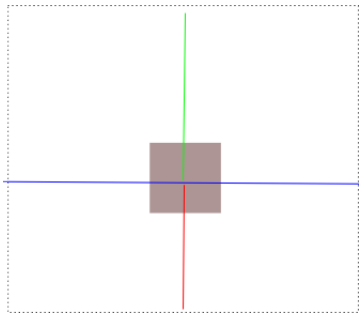
Координате темена дефинисане у првој фази (подаци о теменима) биле су везане за објекат, али у његовом координатном систему. Рачунање коначних позиција темена у координатном систему камере састоји се из три корака. Први корак је примена трансформација (транслација, ротација, скалирања) на темена, да би се пребациле на одговарајуће место на сцени. Ове трансформације су дефинисане ван процесора темена, у кôду апликације, пре фазе исцртавања. Други корак је промена досадашњег координатног система објекта у координатни систем у ком је камера у центру. Позиција камере се такође дефинише у кôду апликације, пре ове фазе. Трећи корак је примена матрице пројекције на добијене координате, након чега се објекат види из ока посматрача (камере), са реалистичним ефектом пројекције. Дакле, стање темена пре ове фазе је описано на Слици 3, након првог корака на Слици 4, а након другог и трећег на Слици 5.



Слика 3: коцка након прве фазе



Слика 4: коцка након померања објекта унутар света

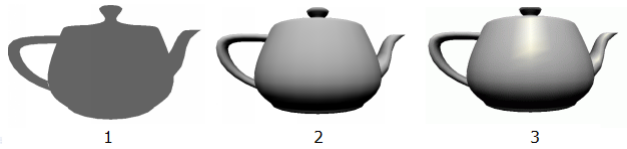


Слика 5: коцка након промене координатног система у координатни систем камере и након примене матрице пројекције

2.3.2 Рачунање боја темена на основу осветљења

Процесор темена може да израчуна и коначне боје темена, укључујући неке информације које у првој фази нису искоришћене, као што је осветљење. Предност коришћења процесора темена је то што се његов кôд извршава на графичкој процесорској јединици, која је специјализована за такве операције, а тиме се растерећује процесор.
OpenGL апроксимира реално светло помоћу три боје: боја *амбијенталног, дифузног и спекуларног* осветљења. *Амбијентално* осветљење представља генерални ниво осветљења на сцени и његова позиција не утиче на коначну боју објеката. *Дифузно* осветљење истиче облик објеката, а коначна боја објеката зависи од позиције извора светлости. *Спекуларно* осветљење представља светлост која се одбија од објекта и долази до камере. Коначна боја објекта произведена учинком спекуларног осветљења зависи од позиције светла, а и од позиције камере. На Слици 6 се види како изгледа чајник под утицајем ових компоненти светлости.

Како се рачуна коначна боја објекта? У првом кораку (подаци о теменима) дефинисана је основна боја објекта. За конкретно теме, боја



Слика 6: 1) амбијентално осветљење 2) амбијентално и дифузно осветљење 3) амбијентално, дифузно и спекуларно осветљење

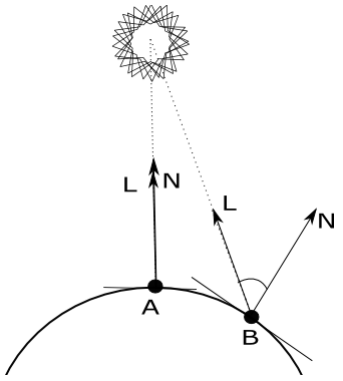
је дефинисана као вектор од три вредности у интервалу [0-1] редом: црвена (r), зелена (g), плава (b). Раније је помињана и четврта вредност, али то је прозирност. Она је у имплементацији постављена на 1.0 за све боје и не учествује у овом рачуну. Први корак је да се на основну боју дода амбијентална боја светла[5]. Други корак је да се дода допринос дифузне боје светла. Тај допринос је у имплементацији израчунат према Ламбертовом светлосном моделу[1]. Он зависи од растојања извора светлости од темена и од угла под којим светлост пада на теме. Нека је L вектор правца од темена до извора светлости, L_0 нормализован вектор L, $d = |L|$, N вектор нормале на троугао коме припада теме, N_0 нормализован N, а D вектор дифузне боје светлости. Тада је укупна боја која се додаје на основну боју у овом кораку једнака

$$D \cdot \frac{N_0 \cdot L_0}{a + bd + cd^2}$$

јер што је веће растојање темена од извора светлости, именилац је мањи, а самим тим је и мањи интензитет осветљења. Веза између растојања и интензитета осветљености није линеарна у природи, а по овом моделу је квадратна. Коефицијенти a, b, c могу да варирају, а у имплементацији су коришћени $a = 0, b = 0, c = 0.001$. У Ламбертовом моделу интензитет осветљености зависи од угла L_0 и N_0. На пример, ако су N_0 и L_0 нормални вектори, тада је $N_0 \cdot L_0 = cos(\angle(N_0, L_0) = cos(\frac{\pi}{2}) = 0$, што значи да дифузно осветљење у тој тачки нема никакав ефекат. Ако су, пак N_0 и L_0 у истом смеру, тада је $N_0 \cdot L_0 = cos(\angle(N_0, L_0) = cos(0) = 1$, што је максимална вредност скаларног производа, па онда осветљење у тој тачки има највећу могућу вредност бројиоца. На Слици 7 је тачка A боље осветљена од B, јер је ближа извору светлости, а и угао између нормале и вектора ка светлу је мањи него у случају тачке B.

Трећи корак је додавање доприноса спекуларног осветљења. Осветљење настало спекуларном светлошћу се добија множењем спекуларне боје светла коефицијентом који се добија на следећи начин. Одреди се нормализован вектор R који настаје рефлектовањем вектора светлости од површине (Слика 8). N_0 и L_0 су нормализовани вектори N и L, а θ је угао између N_0 и L_0. Нека је:

[5] Амбијентална боја је углавном слабог интензитета



Слика 7: Демонстрација дифузног осветљења

$$L' = \frac{N_0}{cos(\theta)},\ M = N_0 - L'$$

Тада се R рачуна на основу израза:

$$R = N_0 + M = N_0 + N_0 - L' = 2N_0 - \frac{N_0}{cos(\theta)} = \frac{2N_0(N_0 \cdot L_0) - N_0}{N_0 \cdot L_0}.$$

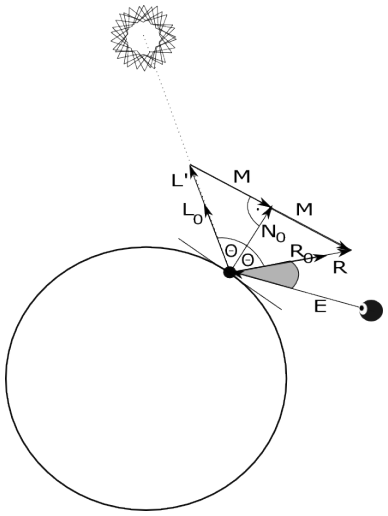
Даље се користи R_0, нормализован R. Након тога се израчуна угао између вектора R_0 и нормализованог вектора који показује правац теме - камера (нека то буде вектор E_0). Ако спекуларну боју светлости обележимо са S, а коефицијент сјаја објекта са s, онда је укупан учинак спекуларне светлости

$$S * (R_0 \cdot E_0)^s$$

Тиме је завршено рачунање боја. У наредним фазама се врши интерполација којом се на основу боја придружених теменима троугла добијају и боје свих осталих тачака у троуглу.

2.4 Теселација, геометријски процесор, склапање примитива

Теселација је процес дељења скупова темена из претходне фазе на подскупове, додавањем нових темена. Један део теселације је програмабилан и ту се дефинише како се тумаче темена паковања у низове (у имплементацији се пакују као троуглови). Остатак сервер обавља сам. Геометријски процесор није коришћен у овом пројекту. Он пружа могућност да се на основу постојећих темена дефинишу нова, пре наредног корака. Након ових фаза долази финално организовање темена у примитиве.



Слика 8: Демонстрација спекуларног осветљења

2.5 Одсецање

У клијенту се дефинише и оквир кроз који се гледа сцена. На основу тога сервер, користећи алгоритам за одсецање скраћује добијене примитиве тако да не превазилазе границе оквира. Ова фаза је такође имплементирана на серверској страни, стога није даље разматрана.

2.6 Растеризација

Растеризација је процес формирања фрагмената на основу примитива. Прво се одреде *пиксели* које покрива свака примитива. Након тога се за све те пикселе рачунају вредности (боје, дубине) као интерполисане вредности темена које су дошле из претходне фазе. На основу ових података се формирају фрагменти, који се могу описати као потенцијални пиксели. Неки од тих фрагмената се одбацују у овој фази, на основу одређених фактора, као што је, на пример, дубина.

2.7 Процесор фрагмената

Процесор фрагмената (*fragment shader*) омогућава да се додатно обраде видљиви фрагменти. Ова фаза може да се окарактерише као постпроцесирање. Након ове фазе пиксели добијају своје коначне боје. У имплементацији процесор фрагмената не ради ништа значајно него само прихвата боје фрагмената који су доспели до њега и њима боји пикселе.

3 Алгоритми за проналажење најјефтинијег пута

Алгоритми за проналажење најјефтинијег пута у графу имају за задатак да пронађу оптималну путању између две тачке у графу. Нека је са $G(V, E, F)$ дефинисан неусмерен граф G, са скупом темена V, скупом грана E и тежинском функцијом $F : E \to R$ која гранама графа додељује тежине. Претпоставља се да су вредности F позитивне. За дата два чвора s и e потребно је пронаћи путању

$$v_1 = s, v_2, ..., v_{n-1}, v_n = e \quad (v_i, v_{i+1}) \in E, \ i = 1, 2, ..., n-1$$

тако да не постоје

$$v'_1 = s, v'_2, ..., v'_{n'-1}, v'_{n'} = e \quad (v'_i, v'_{i+1}) \in E, \ i = 1, 2, ..., n'-1$$

тако да

$$\sum_{i=1}^{n'-1} F(v'_i, v'_{i+1}) > \sum_{i=1}^{n-1} F(v_i, v_{i+1})$$

Ово је општа дефиниција. Рад се бави тражењем најкраћих путања по градској мрежи, стога граф може да се конкретизује матрицом. Чворови графа су поља матрице, а гране су ивице између поља матрице. Свако поље има дефинисану вредност која означава цену проласка кроз то поље. Тежинска функција овде има мало другачији облик. То је сада функција $F : R^2 \to R, F : (i, j) \mapsto c$. Дакле, она пољима додељује цену. Поља која имају бесконачно велику цену називамо препрекама. Приликом тражења пута, из поља (i, j) могуће је посетити сва околна поља, $((i-1, j-1), (i, j-1), (i+1, j-1), (i-1, j), (i+1, j), (i-1, j+1), (i, j+1), (i+1, j+1))$, осим ако садрже препреку, или ако не постоје, кад је поље (i, j) на ивици мапе. Задатак проналаска оптималног пута кроз овако дефинисану матрицу гласи: За два дата поља $u = (u_i, u_j)$ и $v = (v_i, v_j)$, таква да $F(u) < \infty$ и $F(v) < \infty$ пронаћи низ поља

$$(p_k) = (i_k, j_k), k = 1, ..., n$$

тако да је

$$p_0 = u, p_1 = v \quad |i_{k+1} - i_k| \in \{0, 1\}, |j_{k+1} - j_k| \in \{0, 1\}, (i_k, j_k) \neq (i_{k+1}, j_{k+1})$$

$$k = 1, ..., n-1$$

и не постоји низ $(p'_k) = (i'_k, j'_k), k = 1, ..., n'$ различит од (p_k), тако да испуњава исте услове и да

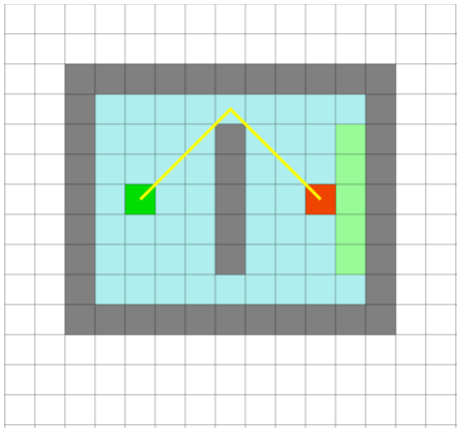
$$\sum_{k=1}^{n'-1} F(i'_k, j'_k) > \sum_{k=1}^{n-1} F(i_k, j_k)$$

.

3.1 Дејкстрин алгоритам

Једно од решења је Дејкстрин алгоритам за графове, модификован за случај горе дефинисане матрице. Поље се дефинише као структура која има четри вредности: i за врсту, j за колону, *cena* за цену (представља вредност функције F) и логичку променљиву *markirano*, која је за сва поља иницијализована на 0. Доле је описан псеудо кôд Дејкстриног алгоритма. Улаз чине матрица A са m редова и n колона, која чува сва поља, затим почетно поље s, и крајње поље k. Излаз је низ поља R који чине пронађену путању. Помоћне променљиве су мапа претходника M (кључ је поље, а вредност је његов претходник у оптималној путањи), матрица P која чува цену од s до датог поља и низ Q који чува активна поља.

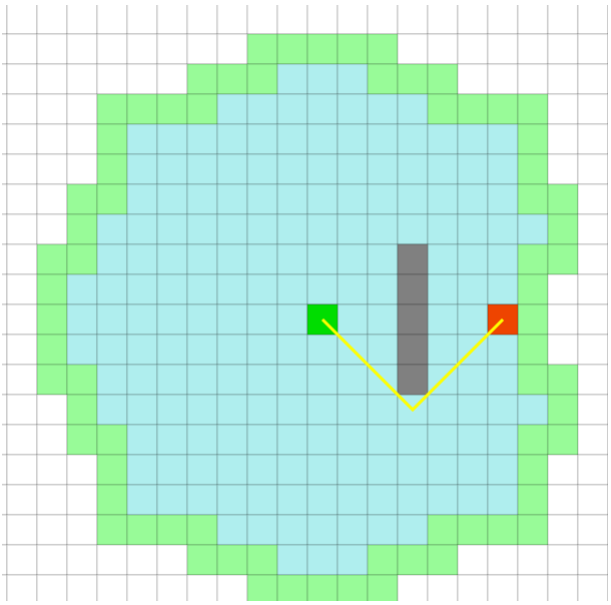
Први корак алгоритма је иницијализација. У другом кораку се пролази кроз матрицу модификованим BFS алгоритмом, са специфичним начином одабира наредног темена и чувањем претходника и укупне удаљености од почетка за свако теме. На улазу у трећи корак попуњене су матрица P и мапа M. На основу M може да се пронађе оптимални пут, полазећи од крајњег темена и пратећи мапу. Како се прелази мапа, убацују се поља путање у низ R. Добија се низ поља оптималне путање од k ка s, па је стога неопходно обрнути овај низ и тиме се добија тражени резултат. Овај алгоритам ради добро за уске површине ограничене препрекама, али за отворене, простране површине прави доста сувишних итерација.
Резултати симулације Дејкстриног алгоритма помоћу једне интернет апликације[10] могу се видети на Слици 9 и 10.



Слика 9: Демонстрација Дејкстриног алгоритма у ограниченој области

Algoritam 1 Dejsktra

```
1:  procedure DEJKSTRA
2:      // I korak
3:      P[s] ← 0
4:      M[s] ← NULL
5:      Q ← Q ∪ {s}
6:      for p ∈ A do
7:          if p ≠ s then
8:              P[p] ← ∞
9:              M[p] ← NULL
10:             A[p].markirano ← 0
11:     // II korak
12:     while Q ≠ {} do
13:         u je polje u Q sa najmanjim P[u]
14:         A[u].markirano ← 1
15:         Q ← Q \ {u}
16:         for w ∈ susedi(u) do
17:             if A[w] = INF then
18:                 continue
19:             if A[w].markirano = 1 then
20:                 d = P[u] + A[w].cena
21:                 if d < P[w] then
22:                     P[w] = d
23:                     M[w] = u
24:             else
25:                 P[w] = P[u] + A[w].cena
26:                 M[w] = u
27:     // III korak
28:     R ← {k}
29:     while M[k] ≠ NULL do
30:         k = M[k]
31:         R ← R ∪ {k}
32:     R ← reverse(R)
33:     return R
```



Слика 10: Демонстрација Дејкстриног алгоритма у отвореној области

На две наведене слике зеленом бојом је обојено почетно поље, црвеном крајње, сивом зидови (препреке), плавом и светлозеленом поља која је алгоритам посећивао[6] и коначно, жутом путању коју је алгоритам пронашао. На слици 10 се види колико непотребних посета поља је направио алгоритам у отвореној области. У наставку је описан бољи алгоритам: A*.

3.2 Алгоритам A*

Алгоритам A* за тражење најкраће путање између два чвора графа (у нашем случају мреже) је један од најпопуларнијих алгоритама вештачке интелигенције[6]. Дејкстрин алгоритам узима у обзир само цену од почетног до текућег чвора, док A* користи и информације о циљу. A* имплементира функције f, g, h дефинисане на следећи начин:

$$f(P) = g(P) + h(P)$$

Вредност $g(P)$ је цена од полазног поља до P, а $h(P)$ процењена цена оптималног пута од P до циља. Функција h је функција *процене* и она мора да буде *конзистентна*, тј. да за свака два суседна поља P, Q важи:

$$h(Q) \leq h(P) + F(Q)$$

[6]Плавом, односно светлозеленом бојом су представљена поља затворене, односно отворене листе, које су описане касније.

Вредност $F(Q)$ је цена проласка кроз поље Q. Одавде следи да h неће прецењивати цену стизања до циља. Вредност функција g и h за поља се чувају у матрицама истих димензија као што је матрица коју претражујемо. Користе се и три додатне структуре, *отворена* и *затворена* листа, које су дефинисане у даљем тексту, и *мапа предака* која за поље чува његовог претка у покушају тражења пута. Опис алгоритма[8][7]:

[7] Напомена: Ово је алгоритам за случај када је дозвољено и дијагонално кретање по матрици. Мало је општији него алгоритам у имплементацији.

Algoritam 2 : A*

```
1:  procedure ASTAR
2:      O = O ∪ {s}
3:      g[s] ← 0
4:      h[s] ← izracunajH(s)
5:      ishod ← USPEH
6:      while true do
7:          P je polje u O sa najmanjim f(P)
8:          C ← C ∪ {P}
9:          O ← O \ {P}
10:         for Q ∈ susedi(P) do
11:             if f[Q] = ∞ or Q ∈ C then
12:                 continue
13:             if Q ∉ O then
14:                 O ← O ∪ {Q}
15:                 M[Q] ← P
16:                 g[Q] ← izracunajG(P, Q)
17:                 h[Q] ← izracunajH(Q)
18:             else
19:                 cena ← cena(P, Q)
20:                 if g[P] + cena < g[Q] then
21:                     g[Q] = g[P] + cena
22:                     M[Q] ← P
23:         if k ∈ C then
24:             ishod ← USPEH
25:             break
26:         if O = {} then
27:             ishod ← NEUSPEH
28:             break
29:     if ishod = NEUSPEH then
30:         return NULL
31:     else
32:         R = {k}
33:         while M[k] ≠ NULL do
34:             k ← M[k]
35:             R ← R ∪ {k}
36:         R ← reverse(R)
37:         return R
```

Објашњење функција:

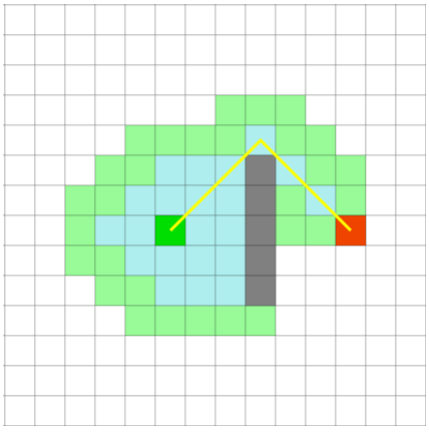
1. *izracunajH*(P): Вредност h за поље може да се рачуна на више начина, али једнако за сва поља. Један од начина да се мери h је квадрат еуклидског растојања поља од циља: $h(P) = (k_i - P_i)^2 + (k_j - P_j)^2$. Ипак, одлучено је да се користи следећа *хеуристика*:

$h(P) = 10(|k_i - P_i| + |k_j - P_j|)$. Једноставнија је за израчунавање и прилагођена потребама апликације.

2. *izracunajG*(P, Q): Како се рачуна вредност $g(Q)$, када долазимо до Q преко поља P? Вредност функције g за суседа Q поља P зависи од њихових позиција у матрици. Ако су P и Q у истој врсти, или истој колони матрице, тада је $g(Q) = g(P)+F(Q)+10$. У супротном (постављени су дијагонално), $g(Q) = g(P) + F(Q) + 14$. Циљ ових константи (10 и 14) је да направе разлику између путовања по врстама и колонама и путовања дијагонално. Константа $14 \approx 10\sqrt{2}$ се користи управо због одлуке да путовање по дијагонали буде $\sqrt{2}$ пута скупље него путовање по врстама/колонама.

3. *cena*(P, Q): Вредност коју враћа ова функција је једнака 10, односно 14 ако су P и Q у истој хоризонтали/вертикали, односно ако су постављени дијагонално.

Као што је напоменуто, ово је случај када је дозвољено дијагонално кретање и то је изворни алгоритам A*. У апликацији је забрањено дијагонално кретање, па се функција g за све суседе P, Q дефинише са $g(Q) = g(P) + F(Q) + 10$. Поред тога, када се посматрају суседи, не гледају се дијагонални, па се узимају у обзир само 4 суседа (горе, доле, лево, десно). На Слици 11 се види како A* скраћује путању у односу на Дејкстрин алгоритам на отвореној области (Упоредити са Сликом 10).



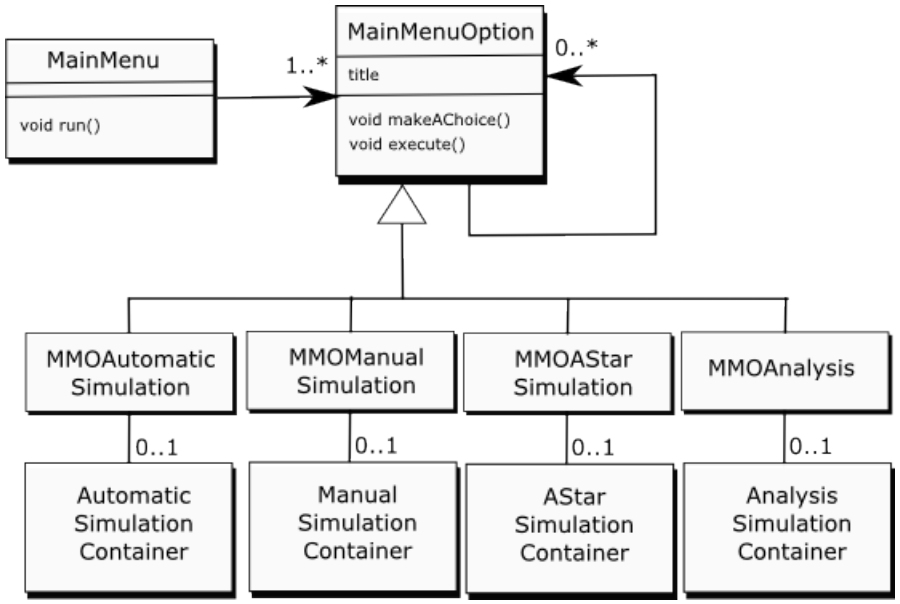
Слика 11: Демонстрација рада A* алгоритма

4 Програм за симулацију развоја града

4.1 Приказ структуре програма

4.1.1 Мени

Приликом покретања програма покреће се *мени* којим се бира врста симулације. Мени се састоји од листе опција, које могу да садрже и подопције, тако да скуп опција чини *стабло*. Покретањем менија извршава се метода *makeAChoice()* која проверава да ли је листа опција празна. Ако јесте, то значи да се ради о листу стабла и тада се позива *execute()* метода опције, којом се покрећу функционалности програма којима та опција одговара. У супротном, ако листа опција није празна, уноси се број којим се бира индекс подопције и поступак избора се понавља за изабрану подопцију. Дијаграм класа[8] менија представљен је на Слици 12. Сваки лист мени-стабла садржи кôд којим се креира одређени оквир симулације, који у себи има симулацију и окружење погодно за њену презентацију. Постоје 4 оквира за симулацију: мануелна симулација, аутоматизована симулација, симулација A* алгоритма и анализа.

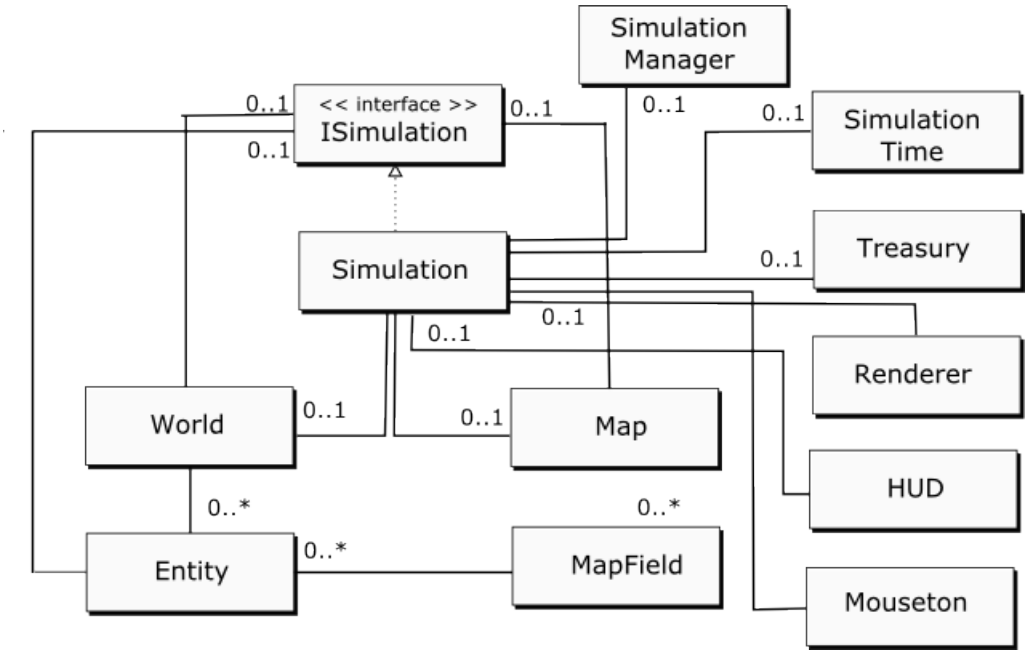


Слика 12: Дијаграм класа - мени

[8] Дијаграми представљени у овом раду су упрошћени УМЛ дијаграми.

4.1.2 Симулација и компоненте

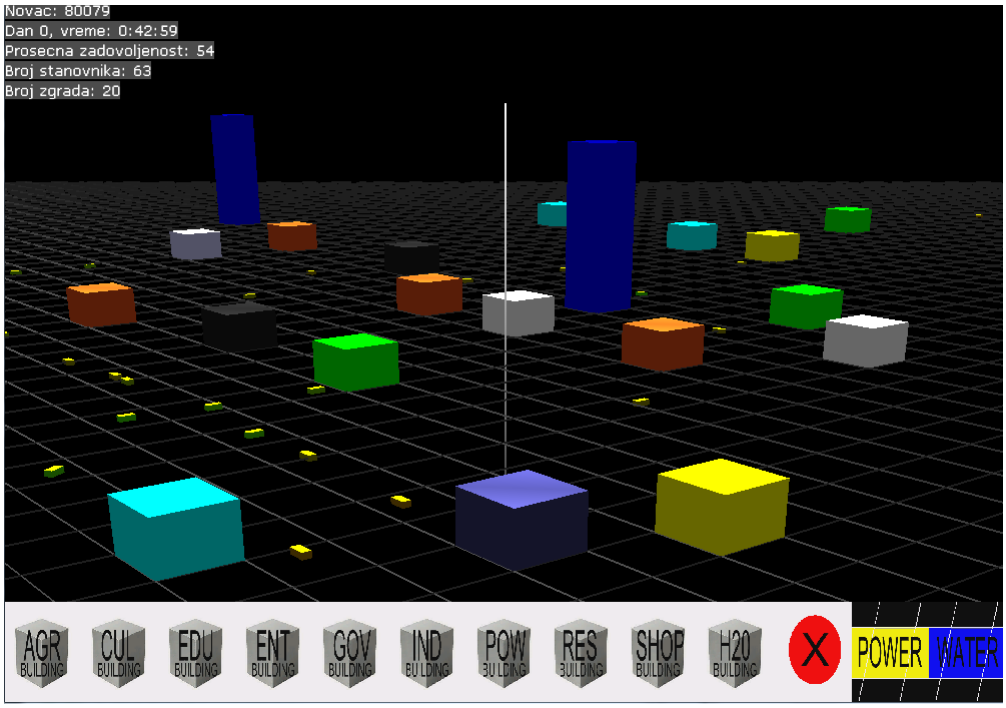
На Слици 13 је описана структура симулације, а на Слици 14 је графички приказ. Постоји интерфејс за приступање симулацији од стране њених компоненти, да би се избегла *циркуларна зависност* класа. Класа *SimulationManager* се разликује у наведена 4 оквира симулације и његова сврха је описана касније. Основне компоненте које чувају ентитете симулације су свет (*World*) и мапа (*Map*). Оне представљају две различите репрезентације скупа ентитета. Када се нова инстанца неког ентитета додаје у симулацију, показивач се додаје и свету и мапи. Свет садржи низ показивача на ентитете, док се мапа састоји из матрице поља(*MapField*) која садрже кратке низове показивача на ентитете. Позитивна последица (у односу на случај са само једном структуром) је лако манипулисање ентитетима, док је негативна последица додатно коришћење меморије за смештање показивача.



Слика 13: Дијаграм класа - симулација

Координате ентитета у свету и у мапи су различите, и постоје посебне методе које конвертују координате из једног система у други, и обрнуто. У симулацији су дефинисане полу-дужина (hw) и полуширина света (hh). Координатни системи света и мапе су оба Декартови правоугли координатни системи. Тачке у свету дефинисане су са:

$$\{(x,y)|\ -hw \leq x \leq hw, x \neq 0, -hh \leq y \leq hh, y \neq 0\}$$



Слика 14: Графички приказ симулације

Поља у мапи су дефинисана скупом:

$$\{(i,j)|\ 0 \leq i < 2hh, 0 \leq y < 2hw\}$$

Мапа служи за проучавање позиције ентитета у односу на остале, док се свет користи за брзи пролаз кроз низ ентитета.

Класа *SimulationTime* генерише време у симулацији и користе је сви ентитети преко објекта симулације. Уведена је првенствено да се надомести недостатак дефинисања јединичног временског интервала у OpenGL-у. Наиме, средиште сваког OpenGL програма је функција за *исцртавање* садржаја која се извршава у кратким временским интервалима, али њихова дужина варира у зависности од комплексности садржаја корака. Одатле може да се закључи да једна итерација те функције не може да буде мерило за време. Класа која мери време симулације не мери време на основу броја итерација функција за исцртавање, већ на основу мерења разлике у откуцајима системског бројача времена. Такође, даје могућност да се догађаји дефинишу у зависности од времена (нпр. уради нешто на сваких 10секунди).
Класа *Mouseton* је класа за интеракцију са симулацијом преко миша. Садржи важне методе којима се на основу координата курсора миша

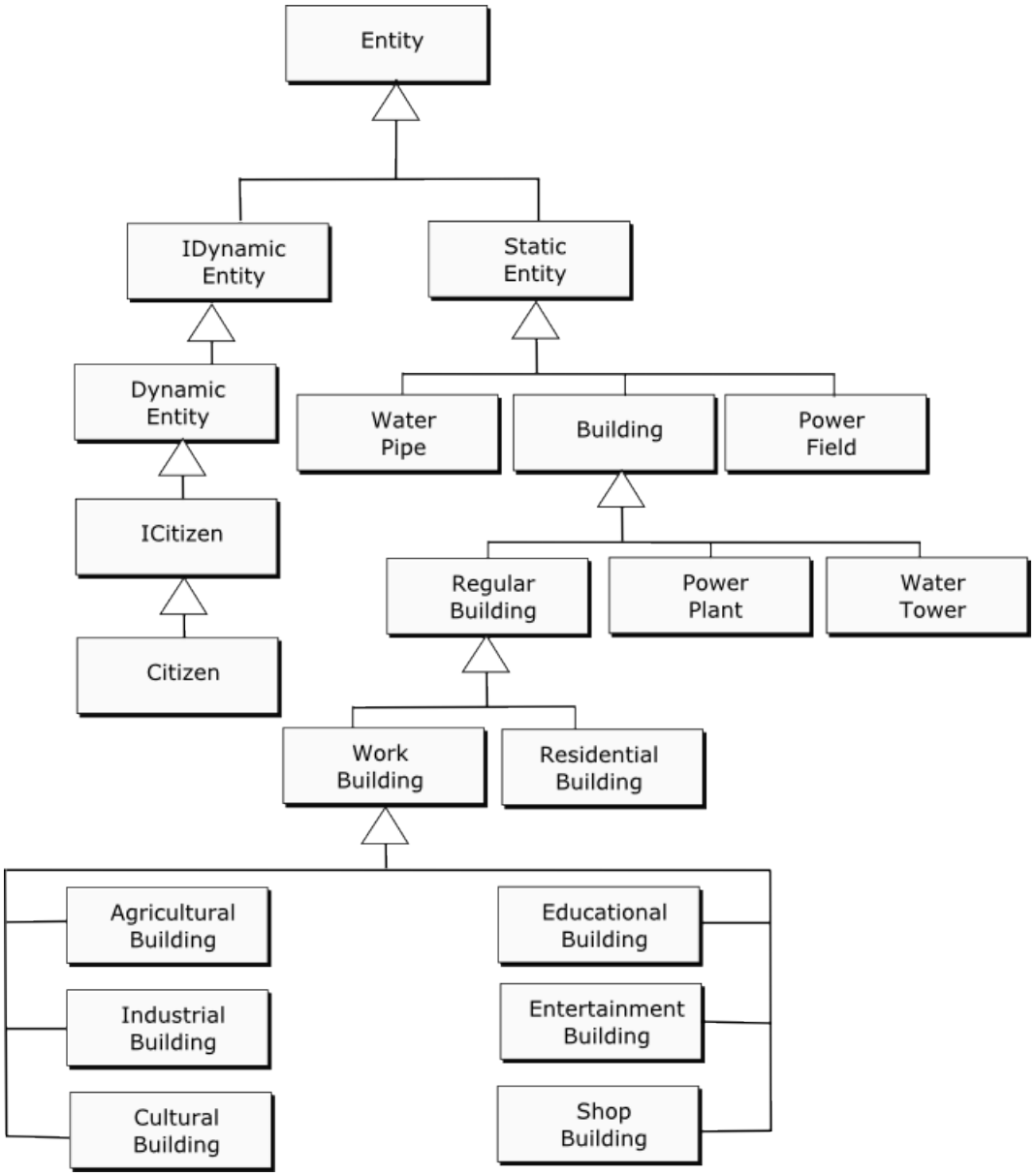
на екрану проналазе координате поља у свету изнад кога се налази курсор. OpenGL чува матрице којима се тродимензионалне тачке конвертују у дводимензионалне тачке на екрану. На основу тих матрица и дате дводимензионалне тачке могуће је пронаћи праву у тродимензионалном свету, која одговара положају курсора. Након тога се рачунањем пресека те праве и жељене равни (у овом случају равни на којој стоји град) добија тачка у свету коју показује курсор. Ова класа такође регулише притисак, пуштање и померање левог и десног курсора миша и позива одговарајуће акције. Остале класе са Слике 13 су објашњене касније, након приказа ентитета.

4.1.3 Ентитети

Сваки *ентитет* има *ID*, који је јединствени идентификатор, а класа *Entity* статичким методама и променљивама брине о томе да идентификатори заиста јединствено одређују ентитет. Сваки ентитет има податке о теменима и нормалама (класа $MeshData$) и податке о бојама тих темена(класа $ColorData$). Ентитети се на основу ових података исцртавају у простору, како је описано раније. Показивач на симулацију је такође присутан у ентитету, да би ентитет могао да комуницира са осталим члановима симулације. Методи које имплементирају сви ентитети су методи за освежавање стања, исцртавање итд. Дијаграм ентитета и поткласа је приказан на Слици 15.

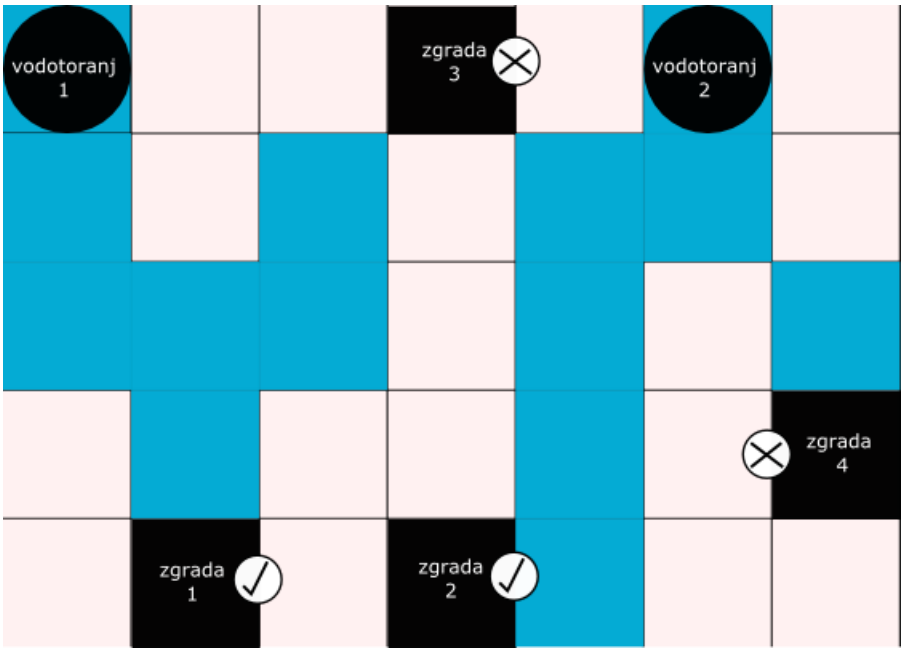
Статични ентитети се не померају, па није неопходно освежавати њихово стање и положај током рада апликације, док динамични ентитети имају дефинисано понашање. Ентитет *водоводно поље*($WaterPipe$) и *електрично поље*($PowerField$) учествују у креирању градске водоводне и електричне мреже. *Регуларна* зграда може да се повеже на воду или струју, ако на мапи постоји пут од ње до *водоторња*($WaterTower$) или до електране($PowerPlant$). На Слици 16 су приказана 2 водоторња, 4 зграде и водоводне цеви.

Зграда 1 је директно повезана на водоторањ 1, зграда 2 је директно повезана на водоторањ 2. Зграда 3 није повезана ни на један водоторањ јер се ниједна путања водовода не завршава у њој. Зграда 4 има водоводно поље до себе, али је путања од водоторња 2 прекинута, тако да ни она не добија воду. У позадини се, након сваког додавања водоводног поља, регуларне зграде и водоторња одради једна претрага и освежавање стања, након кога свака зграда "зна" да ли је повезана на водовод. Наиме, за сваки водоторањ се позива претрага у ширину која обилази сва водоводна поља везана за њега и проверава да ли је нека зграда повезана на водовод тим пољима. Исто тако функционишу поља за струју и електране. Сврха специфичних регуларних зграда је описана касније.
Што се тиче динамичних ентитета, ситуација је компликованија. На Слици 17 је приказан дијаграм класа за динамичне ентитете.



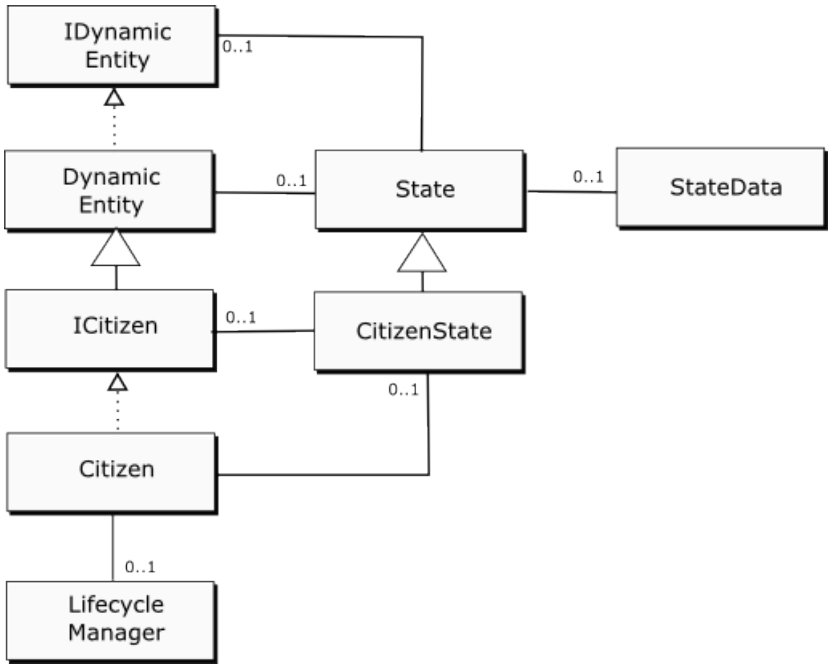
Слика 15: Дијаграм класа - ентитети

Динамични ентитети се активно крећу по свету. Једини динамични ентитети који постоје су грађани, мада је у овако дизајнираном систему лако дефинисати нове динамичне ентитете. Сваки динамични ентитет има сачувану тренутну путању по којој се креће, показивач на



Слика 16: водовод

тренутно стање кретања и показивач на објекат класе *Movement*, који чува податке о кретању и позицији. Путања по којој се ентитет креће је скуп суседних поља по свету. Динамични ентитет функционише као коначни аутомат, а показивач на стање које садржи је управно његово тренутно стање у том аутомату. Идеја о понашању као скупу стања је преузета из књиге Мета Бакленда[2]. Он је описао ентитете као аутомате са коначним бројем стања и представио имплементацију у језику C++ . Нека од могућих стања у имплементацији су: почетак праволинијског кретања, акција на центру поља, акција на крају поља, почетак кружног кретања, стање мировања итд. Класа *State* је надкласа свих стања и она нуди 3 методе: *enter*, *execute* и *exit*, које као аргумент узимају динамични ентитет. Ове методе се наслеђују у потклasама и свака потклаcа њима у потпуности дефинише понашање динамичног ентитета у том стању. Динамични ентитет имплементира методу *changeState* која извршава *exit* методу претходног стања, брише претходно стање (и ослободи сву меморију које је стање држало), покреће *enter* методу новог стања и поставља ново стање као тренутно. Овако дефинисаним стањима се лако описује понашање динамичког ентитета. Стања су уско повезана са објектом кретања (*Movement*) који има сваки динамични ентитет. Објекат кретања садржи податке о брзини ентитета и о његовим унутрашњим координатама у пољу. Свако поље света има свој координатни систем ($-0.5 \leq x \leq 0.5, -0.5 \leq$



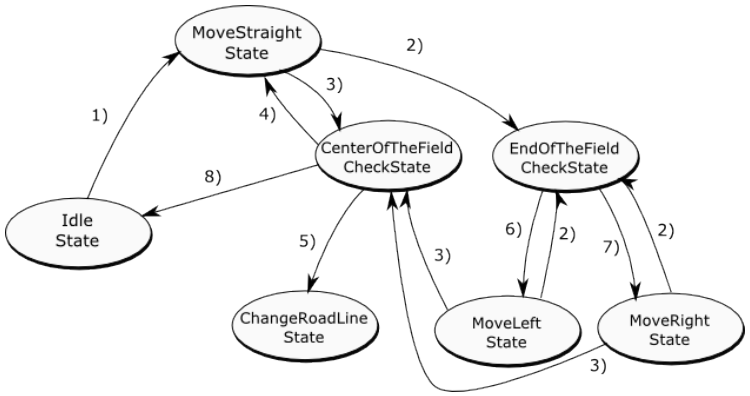
Слика 17: Дијаграм класа - динамични ентитети

$y \leq 0.5$) и динамични ентитет може да се нађе у некој од тих тачака. Положај ентитета у пољу зависи од његовог објекта кретања и од путне траке у којој се налази. Наиме, свако поље се сматра парчетом пута које има 6 паралелних трака у правцу кретања ентитета и ентитет може да буде у једној од тих трака. На Слици 18 дат је приказ аутомата који користи динамични ентитет.

Свако од ових стања, док се извршава, помера динамични ентитет и проверава да ли је неопходан прелазак у наредно стање. Свако стање има свој начин на који рачуна *усмереље*, вектор смера кретања ентитета. На основу израчунатог усмерења се врши померај у текућем стању, а усмерење се касније користи и за исцртавање ентитета[9]. Имплементација стања је захтевала примену аналитичке геометрије, а опис је сложен, стога није наведен у овом раду.

Грађанин (класа *Citizen*) наслеђује динамични ентитет и самим тим има у себи аутомат који служи да га усмерава у кретању по мапи. На Слици 17 се види да грађани садржи још један аутомат, који је његов главни покретач (класа *CitizenState*). Аутомат динамичног ентитета је угњежден у грађанинов аутомат и користи се само када грађанин путује по мапи, јер само тада постоји кретање. Да би се разумео

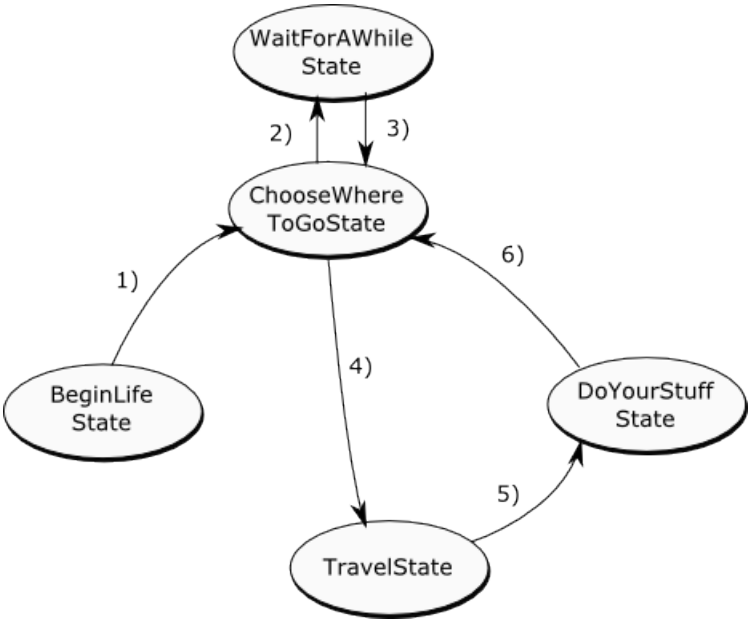
[9]Грађанин је представљен као квадар чија је најдужа ивица увек у правцу вектора усмерења.



Слика 18: Кретање динамичног ентитета: 1) почетак кретања, 2) на крају поља, 3) на средини поља, 4) ако не мора да мења траку, 5) ако је неопходно променити траку пре скретања 6) у наредном пољу се ротира у лево, 7) у наредном пољу се ротира у десно, 8) крај стазе

грађанинов аутоматат, неопходно је објаснити још једну компоненту класе грађанин - организатор(класа $Lifecycle Manager$). Организатор води рачуна о *потребама* грађанина, које га воде по свету. Потребе су изражене у виду *атрибута* (класа *Atribut*). У овој имплементацији апликације постоји 6 врста атрибута: енергија, посао, куповина, култура, забава и едукација. Сваки грађанин у свом организатору има листу објеката који чине агрегацију атрибута и додељене зграде. На почетку *животног циклуса*(при креирању грађанина), у тој листи су зграде недефинисане. Одмах након тога грађанинов аутомат упада у стање у коме се бирају зграде које ће бити додељене атрибутима. Зграде се бирају тако да буду најближе почетној позицији грађанина. Сада долази на ред и прича о врстама зграда која је одложена раније. Одређени атрибут може да буде додељен одређеним зградама. На пример, атрибут за енергију може да буде додељен само стамбеној згради, док атрибут за посао може да буде додељен свим регуларним зградама осим стамбених. Након "бирања" животног циклуса, грађанин почиње да циркулише по одабраним зградама да би задовољио своје потребе(атрибуте). Шта значи да је атрибут задовољен? Атрибут је бројач који има дефинисан *капацитет*, *брзину раста*, *брзину опадања* и користи *текућу вредност*. Стање атрибута се освежава сваке секунде (у времену симулације) и дизајнирано је да се понаша приближно као потреба у стварности. На пример, атрибут енергије има капацитет $57600 = 16 \cdot 3600$, брзину опадања 1 и брзину раста од $\frac{16}{9} \approx 1.77$ до $\frac{16}{5} \approx 3.2$. Када је атрибут активан (тј. када је грађанин у одговарајућој згради), сваке секунде се тренутна вредност атрибута увећава за вредност брзине раста, а кад није активан, вредност опада брзином опадања. У овом примеру се целокупна грађанинова енергија "истроши" за 16

симулираних сати, а напуни се од нуле до максимума за 5-9 симулационих сати, што је приближно ситуацији у стварности. Грађанинов аутомат је описан на Слици 19.



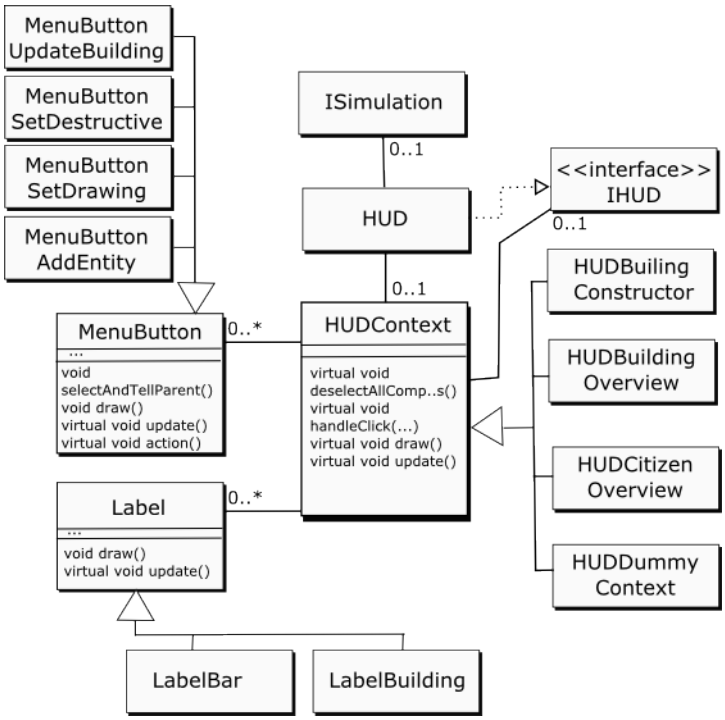
Слика 19: Стања грађанина: 1) након иницијалног избора зграда 2) ако не постоји ниједна додељена зграда 3) ако се појавила доступна зграда 4) ако је изабран наредни пар атрибут-зграда 5) ако је путовање завршено 6) ако је вредност тренутног атрибута дошла до максимума

Овим аутоматом се грађанин креће по граду са циљем да вредности својих атрибута приближи максимумима. Што су изабране зграде на већем растојању, више времена проводи на путу где вредност ниједног атрибута не расте, што не одговара његовом циљу. Постоји још детаља који се не виде на дијаграмима. Ако се новом зградом препречи грађанинов пут од старта до циља, његова путања се брише и поново се израчунава, да би се заобишла препрека. Такође, ако се обрише нека од зграда коју грађанин посећује, он покушава да пронађе замену за ту зграду.

4.1.4 Приказ детаља

Класа *HUD* (од *head – up display*) је задужена за приказ детаља. При доњој ивици екрана стоји главни део приказа. Интеракцијом са симулацијом преко курсора миша приказује се различит садржај. Кликом на одређене ентитете приказују се додатни подаци о њима, а постоји и

посебан приказ за креирање нових зграда. Овај део апликације обилује сликама и текстом. За претварање слика *png* формата у текстуре коришћена је библиотека *SOIL*[10]. Она садржи метод који креира текстуру на основу слике и враћа OpenGL идентификатор којом се користи текстура. Текст се исцртава коришћењем библиотеке *glFont*. Преко ње је могуће дефинисање *фонта* који се помоћу библиотечких метода користи за исцртавање текста, дефинисаног као низ карактера. Дијаграм приказа може се видети на Слици 20. Сваки приказ се састоји од низа *дугмића* и *лабели*.



Слика 20: Приказ података

При креирању приказа дефинише се како су његови елементи распоређени. У класи *HUDContext* која дефинише садржај приказа постоје методе којима се каскадно освежавају и исцртавају елементи. Дугме које је последње одабрано у неком од приказа се посебно третира (истакнуто је, светлије).
Свако дугме носи акцију којом испољава своје дејство на симулацију. Приказ за прављење зграда садржи две групе дугмића - за одабир врсте зграде која се прави и дугмиће и за промену начина исцртавања симулације (*DrawingMode*). Приказ грађана се активира кликом

[10]SOIL - http://lonesock.net/soil.html

на грађанина и у њему се виде статуси његових атрибута. Приказ зграде се активира кликом на регуларну зграду и у њему су дугме за подизање зграде на виши ниво, и информације о згради. Споредни део приказа се налази у горњем левом углу екрана. Ту се налазе додатни подаци везани за саму симулацију.

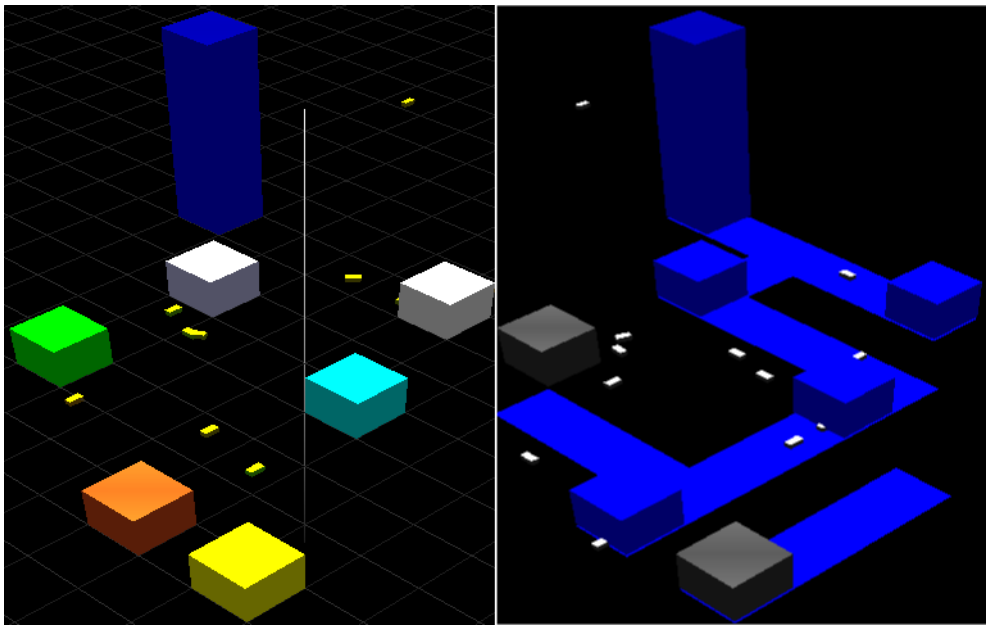
4.1.5 Исцртавање ентитета

Класа *Renderer* служи за исцртавање свих ентитета. Поред тога, она чува податке о светлима и камери. На основу методе $preRenderTransformations()$ сваки ентитет се поставља на одговарајућу позицију у свету, а на основу података о боји и облику (нормале, темена) се исцртава на тој позицији. Постоје 5 типа *погледа на симулацију*: обичан поглед, поглед за додавање зграда, поглед водоводне мреже, поглед електричне мреже и деструктивни поглед. *Renderer* на основу тренутног погледа исцртава сцену на различите начине. Обичан поглед представља све зграде и грађане. Поглед за додавање зграда представља све што и обичан, али и додатни привремени ентитет. Док се привремени ентитет креће по месту у свету где то није дозвољено, он бива обележен као неприкладно постављен и то се види променом његове боје. Поглед водоводне мреже приказује зграде неутралном (сивом бојом), водоторањ плавом бојом, а показује и где се налазе водоводне цеви. У овом погледу могуће је додавати нова водоводна поља. Поглед електричне мреже приказује зграде, такође неутралном бојом, електране приказује жутом бојом, показује и где су електрична поља. У овом погледу могуће је додавати нова електрична поља. Деструктивни поглед показује све ентитете и у њему се они могу брисати. Кликом на поље у том погледу брише се један од ентитета, по посебно дефинисаном приоритету. На Слици 21 је приказана разлика у приказу између обичног погледа и погледа водоводне мреже.

Камера (класа *Camera*) утиче на исцртавање и она је омотач за функцију $gluLookAt(...)$ која као аргументе прима координате ока, координате тачке у коју се гледа и вектор вертикале камере. Атрибути камере су тачка $T = (t_x, t_y, t_z)$, полупречник r и углови ϕ и θ. Они представљају сферне координате камере. Координате ока (камере) у свету се рачунају на следећи начин:

$$x = t_x + r \cdot cos(\theta) \cdot cos(\phi)$$
$$y = t_y + r \cdot cos(\theta) \cdot sin(\phi)$$
$$z = t_z + r \cdot sin(\theta)$$

Координата тачке у коју се гледа су координате тачке T, а вектор вертикале камере је у апликацији постављен на (0,0,1). Мењањем углова ϕ и θ камера се креће по сфери полупречника r са центром у T. У апликацији је могуће и мењање r и T.



Слика 21: лево - обичан поглед, десно - поглед водоводне мреже

Систем за учитавање процесора темена и фрагмената је дефинисан у класи *Shader*. Могуће је лако учитавање из текстуалног фајла, уз управљање грешкама.

4.1.6 Додатни алати

У апликацији је имплементирано неколико *фабрика* (*Factories*) за креирање објеката. Фабрика је интерфејс за креирање фамилије зависних или асоцираних објеката без знања о њиховим конкретним класама[4]. Постоје фабрике за креирање засебних: текстура, боја, ентитета, фонтова, и облика ентитета. Неке фабрике чувају листу коначног скупа инстанци који се често користи у апликацији. На пример, фабрика за облике ентитета садржи по једну инстанцу коцке и квадрата. Ту једну инстанцу коцке користе све зграде (чак и грађани), наравно, уз додатне трансформације (скалирање, транслације, ротације).

Дефинисане су класе *Point* (тачке са целобројним координатама) и *PointF* (тачке са реалним координатама) у којима постоје разне методе за олакшан рад са дводимензионалним тачкама. Осим стандардних метода, постоје и методе за рад са дводимензионалним векторима (рачунање вектора који гледа лево/десно од датог вектора. Постоји и класа за неопходне методе за рад са тродимензионалним векторима - *Vector3D*.

Класа *Tools* нуди много статичких метода које се често користе. У

њој су имплементиране методе за разне конверзије.

4.2 Водич за употребу програма

Приликом покретања апликације, приказује се мени са 4 опције. Да би се одабрала опција, откуца се број који стоји уз опцију и притисне се тастер *enter*. Следи опис прве три врсте симулације, а четврта је описана у наредној глави.

4.2.1 Аутоматизована симулација

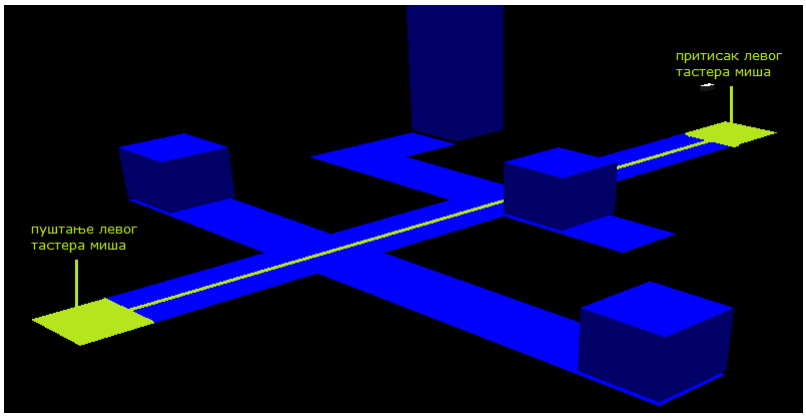
У аутоматизованој симулацији је број могућих радњи доста мањи у односу на мануелну симулацију, мада има доста заједничких елемената. Камера се контролише тастерима тастатуре:

- W: удаљавање од мете
- S: приближавање мети
- A: померај у смеру казаљке на сату
- D: померај у математичком смеру
- Q: померај навише
- E: померај наниже

Кликом на зграду се у приказу на дну екрана види број спратова, број људи везаних за њу и максималан број људи који може да буде везан за зграду тог нивоа. Кад број људи у згради дође до максимума, не може више људи да се веже за њу, осим ако се не унапреди, што захтева додатан новац. Цена унапређења опада са растом броја спратова и дефинисана је са $cena(n) = \frac{i}{n+1}$, где је i иницијална цена зграде, а n број новог спрата. Грађани сваким пролазом кроз своју зграду за посао пуне благајну и одатле долази новац. $SimulationManager$ покушава да дода нову зграду у симулацију, ако у благајни ($Treasury$) има довољно новца. Ако нема, покушава да унапреди неку зграду. У сваком случају, у кратким временским интервалима додаје нове грађане симулацији, а они сами иницирају своје аутомате. Кликом на грађанина се виде подаци о његовим атрибутима. У зависности од просечне задовољености грађанин се боји неком бојом од црвене (просечна задовољеност је минимална) до зелене (просечна задовољеност је максимална). Боја се дефинише реалним вредностима r, g, b у интервалу [0-1]. Ако је просечна задовољеност грађанина $p \in [0, 1]$, тада је његова боја $(1 - p, p, 0)$. Овако ће, на пример, за просечну задовољеност ($p = 0.5$) боја бити жута $(0.5, 0.5, 0)$, за $p = 0$ црвена $(1, 0, 0)$, а за $p = 1$ зелена $(0, 1, 0)$. У горњем левом углу екрана су корисни подаци: тренутна количина новца у благајни, време симулације, просечна задовољеност грађана, број становника и број регуларних зграда.

4.2.2 Мануелна симулација

Мануелна симулација се разликује по томе што *SimulationManager* не прави нове зграде и не покушава да их надогради. Приказ садржаја је богатији за мени за креирање зграда. Првих 10 дугмића је намењено за прављење зграда. Левим кликом на једно од њих формира се привремени ентитет који још не припада свету, али може да се види, како се помера курсор миша по свету. Ако у благајни има довољно новца за куповину те зграде и ако је курсор миша изнад поља на коме је дозвољено прављење зграде, левим кликом миша се креира зграда на том месту. Зграда не може да се направи на пољу на ком већ постоји зграда, или на пољу по коме се тренутно креће грађанин. Привремена зграда је обојена црвеном бојом ако је у недозвољеној позицији. Наредно дугме у менију мења поглед симулација у деструктивни поглед. Кликом на поље врши се брисање по следећем редоследу: Ако постоји зграда, брише се зграда. У супротном, ако постоји електрично поље, оно се брише. Затим, ако постоји водоводно поље, оно се брише. Последња два дугмета служе за пребацивање симулације у поглед водовода и електричне енергије. Следи опис дугмета за водовод (за струју се поступа аналогно). Притиском левог тастера миша изнад неког поља света дефинише се почетак линије водовода. Пуштањем миша изнад неког поља дефинише се крај линије и исцртава се водоводна линија. Почетно и крајње поље морају да имају барем једну исту координату да би се линија исцртала (Слика 22).



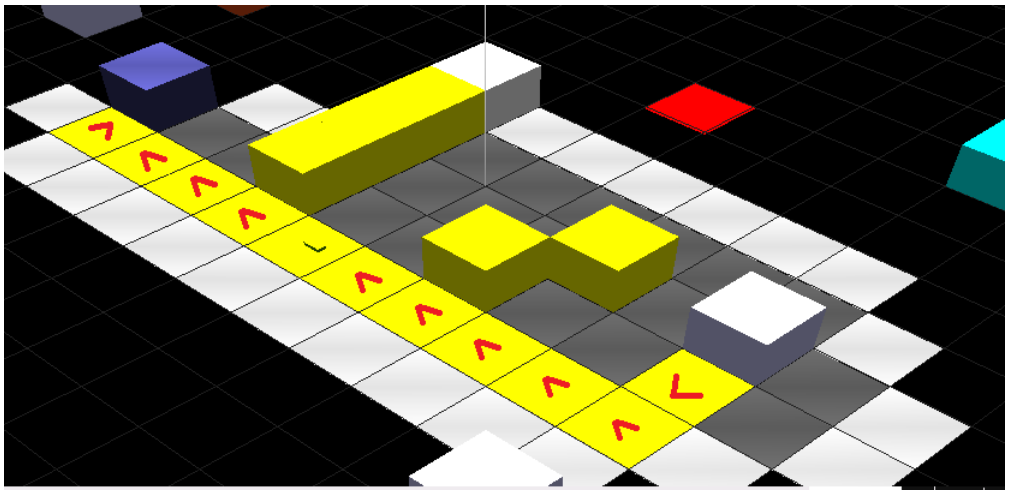
Слика 22: Цртање водовода

Зграда је повезана на водовод ако постоји путања сачињена од водоводних поља од ње до водоторња, а индикатор да је повезана је њена плава боја. Још једна разлика између мануелне и аутоматизоване симулације је то што у мануелној могу да се ручно унапреде зграде. У приказу зграде види се дугме за њено унапређење. Оно је могуће само ако је број људи везаних за зграду достигао максималну

вредност за тај број спратова и ако има довољно новца у благајни.

4.2.3 Симулација A*

У овој симулацији на почетку постоји један грађанин и неколико зграда неопходних за његово кретање. Могуће је додавање нових и брисање постојећих зграда. Приликом сваког прорачуна путање, генеришу се поља која показују како је алгоритам пронашао путању. Тамна поља су поља коначне затворене листе, а светла поља припадају отвореној листи. Жутом бојом је представљена путања коју је алгоритам пронашао. Црвене стрелице показују смер кретања становника. Алгоритам посећује само та поља, и ту може да се види његова ефикасност. У *конзолном прозору* се за сваки прорачун пута исписују подаци о резултату претраге. На Слици 23 је приказан графички приказ ове симулације.



Слика 23: Графички приказ симулације A*

5 Анализа перформанси алгоритма A*

Постоји посебна класа за рад са A* алгоритмом. Она укључује методу за тражење најкраћег пута по мапи, неколико помоћних метода и *структуру* за чување информација о претрази. Приликом сваког позива методе за тражење пута, структура се попуни следећим подацима: почетна и крајња тачка, удаљеност (у броју поља) и број операција. *Операција* може да се дефинише на више начина и овде је извршена анализа за 2 случаја:

- Операција је дефинисана као посета пољу (*Површно мерење*).
- Операција је дефинисана као најјефтинија операција у алгоритму (*Детаљно мерење*).

Тестирање T се састоји од више *тестова* T_n. Тест T_n је просечан број операција који алгоритам изврши на мапи величине $n \times n$, за једног становника. Процес рачунања вредности T_n:

- Генерише се мапа димензија $n \times n$.
- Мапа се попуни зградама. Могуће је дефинисати густину зграда за T и тако број зграда расте са порастом димензија мапе (n), а слике се не мењају драстично. Зграда се поставља тако што јој генератор случајних бројева додели координате у оквиру мапе. Ако је додељено место заузето, поступак се понавља док генератор не произведе координате које су слободне.
- На мапу се дода 200 грађана.
- Сваки грађанин тражи пут до најдаље зграде на мапи и овде се мери број операција претраге.
- Просечан број операција за свих тих 200 путева се додаје у низ $vremena_n$.

Овај поступак се понавља више пута (5-10) да рачун за димензију T_n не би зависио само од једне конфигурације мапе. Коначно T_n је просек вредности низа $vremena_n$.

5.1 Површно мерење

На Слици 24 се види коначна табела добијена површним мерењем. Рађени су тестови за мапе димензија од 4×4, све до 40×40, са једнаком густином распореда зграда и са 200 становника у сваком мерењу.

Коришћењем алата за проналажење најбоље средњеквадратне праве[11] добија се права $y = 91.367 \cdot x - 793.792$ (x представља дужину ивице мапе, n, а y представља $T(n)$).

[11] http://www.mathportal.org/calculators/statistics-calculator/correlation-and-regression-calculator.php

n	2n*n	T(n)
4	32	23.6667
6	72	51.3864
8	128	104.326
10	200	186.61
12	288	247.321
14	392	353.776
16	512	512.762
18	648	609.423
20	800	762.273
22	968	952.497
24	1152	1182.04
26	1352	1304.02
28	1568	1543.08
30	1800	1784.51
32	2048	2138.56
34	2312	2417.68
36	2592	2704.14
38	2888	2963.16
40	3200	3268.11

Слика 24: Површно мерење, табела

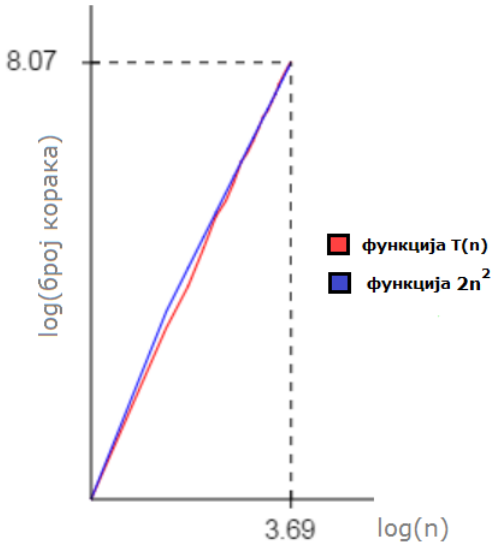
Површно мерење је уведено због поређења са имплементацијом у *јаваскрипт* библиотеци *PathFinding.js*[10]. Поређењем броја корака за исте случајеве, добијају се резултати који се мало разликују.
Поред тога, посматрањем броја операција у упрошћеном мерењу, може се запазити да је функција $T(n)$ приближно једнака функцији $2n^2$ за мапе димензија до 40×40, што се може видети на Слици 25.

5.2 Детаљно мерење

Површно мерење које броји посете пољима није довољно меродавно, јер занемарује доста рачунских операција, упоређивања, а и неких детаља специфичних за имплементацију, као што су пролази кроз дуге низове. Детаљно мерење операцијама сматра: сабирање, множење, упоређивање, доделу. На Слици 26 се види табеларна функција $T(n)$ добијена тестирањем са детаљним мерењем Једначина најбоље средњеквадратне праве је сада $y = 27495.995 \cdot x - 330563.33$.[12] Дакле, нагиб је доста већи него у првом случају, мада је мањи од нагиба најбоље средњеквадратне праве за график функције n^4.

Тачна процена сложености алгоритма је нетривијалан проблем, јер не зависи само од удаљености почетка и циља, него и од распореда препрека које се налазе на путу. Алгоритам се ослања на отворену и

[12] Ознаке су исте као код површног мерења



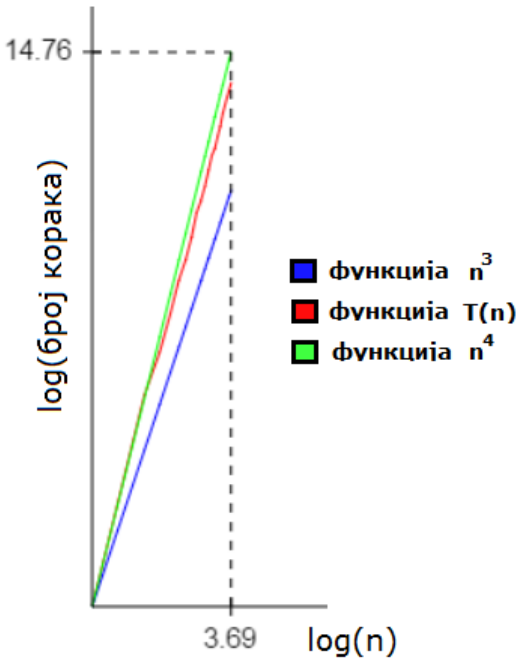
Слика 25: Површно мерење, поређење $T(n)$ са $2n^2$

n	n^3	n^4	T(n)
4	64	256.00	277.24
6	216	1296.00	860.96
8	512	4096.00	2436.78
10	1000	10000.00	5926.82
12	1728	20736.00	10238.50
14	2744	38416.00	18462.60
16	4096	65536.00	35491.60
18	5832	104976.00	49117.80
20	8000	160000.00	72504.30
22	10648	234256.00	110044.00
24	13824	331776.00	163138.00
26	17576	456976.00	196641.00
28	21952	614656.00	270942.00
30	27000	810000.00	350866.00
32	32768	1048576.00	491634.00
34	39304	1336336.00	619688.00
36	46656	1679616.00	772059.00
38	54872	2085136.00	924374.00
40	64000	2560000.00	1117920.00

Слика 26: Детаљно мерење - табела

затворену листу, чији садржај није тако лако одредити за дати корак. На Слици 27 је упоређен раст функције $T(n)$ са функцијама n^3 и n^4 и

ту може да се види понашање за мапе до димензија 40×40.



Слика 27: Детално мерење, поређење $T(n)$ са n^3 и n^4

6 Закључак

Основни циљ рада је био прављење подлоге за креирање модела града и симулације кретања у њему. Направљена је велика хијерархија класа, довољно добро организована да пројекат може да се прошири без већих проблема. Програм је често *рефакторисан*, под утицајем књиге о рефакторисању, Мартина Фаулера[3]. Књига је помогла и у дубљем разумевању објектно-оријентисане парадигме. Велики број класа и веза захтевао је добро познавање програмског језика $C++$, стога су коришћене књига Ласла Крауса[7] и приручник за разумевање $C++$[5].

Коришћен је један од актуелних концепата у програмирању графике - програмабилан ток исцртавања. Потребно је било савладати OpenGL, који је један веома сложен систем. Систем за исцртавање може да се побољша, и то је један интересантан проблем на коме би могло да се ради.

Алгоритам A* се показао као довољно ефикасан за потребе рада. На мапама великих димензија (у зависности од јачине процесора) понекад се примећују кратки застоји у исцртавању сцене, због великог броја операција које A* алгоритам извршава за грађане. Могуће је убрзати алгоритам коришћењем адекватнијих структура за смештање привремених података.

Ентитети представљени у овом раду могу да буду основа за прављење нових ентитета. Захваљујући фабрикама лако се дефинишу облик и боја ентитета, а понашање може да се испрограмира аутоматима. Дефинисањем нових врста зграда и изменом функција којима се грађанима додељују зграде, може се дефинисати потпуно другачији живот у граду. Богатије понашање грађана може да се реализује већим бројем стања у њиховим аутоматима. Систем је за сада заокружен и може да се развија самостално, што може да се види у аутоматизованој симулацији. Додаци би превазишли временске оквире за рад. Коначно, на основу боја грађана види се да ли превише путују између зграда, што може бити индикатор да је распоред зграда лош. Тиме би овај рад могао да се употреби као основа за планирање неког градског система.

7 Литература

[1] E. Angel, D. Shreiner, *Interactive computer graphics: A top-down approach with shader-based OpenGL, 6th edition*, Addison-Wesley, United States of America, 2012.

[2] M. Buckland, *Game AI by Example*, Wordware Publishing, Inc. United States, 2005.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code* Addison-Wesley, United States of America, 2000.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, United States of America, 1997.

[5] J. Gray, *C++: Under the Hood*, march 1994.

[6] P. Janicic, M. Nikolic, *Vestacka inteligencija*, Beograd, 2010.

[7] L. Kraus, *Programski jezik C++ sa reshenim zadacima* Akademska misao, Beograd 2003.

[8] Patrick Lester, A* tutorial, *http://www.policyalmanac.org/games/ aStarTutorial.htm*, 2005

[9] D. Shreiner, G. Sellers, J. M. Kessenich, B. M. Licea-Kanem, *OpenGL Programming Guide, 8th Edition*, The Khronos OpenGL ARB Working Group, United States of America, 2013.

[10] Pathfinding.js, *http://qiao.github.io/PathFinding.js/visual/*