VIŠJA STROKOVNA ŠOLA ACADEMIA,
MARIBOR

DIPLOMSKO DELO

INTERAKTIVNI PRIKAZ PRAVILNE VOŽNJE V KROŽIŠČU V ADOBE FLASH CS4

Kandidat: Matej Zavernik
Študent študija ob delu
Številka indeksa: 11190122764
Program: Multimediji
Mentor: Veronika Saje, univ. dipl. inž. arh.
Mentor v podjetju: Matej Butala

Maribor, februar 2011

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani Matej Zavernik, št. indeksa 11190122764, sem avtor diplomske naloge z naslovom

INTERAKTIVNI PRIKAZ PRAVILNE VOŽNJE
V KROŽIŠČU V ADOBE FLASH CS4,

ki sem jo napisal pod mentorstvom Veronike Saje, univ. dipl. inž. arh.

S svojim podpisom zagotavljam, da:

- je predložena diplomska naloga izključno rezultat mojega dela;
- sem poskrbel, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia;
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli kot moje lastne – kaznivo po Zakonu o avtorskih in sorodnih pravicah (UL št. 16/2007 – v nadaljevanju ZASP), prekršek pa podleže tudi ukrepom VSŠ Academia, skladno z njenimi pravili;
- skladno z 32. členom ZASP dovoljujem VSŠ Academia objavo diplomske naloge na spletnem portalu šole.

Maribor, februar 2011 Podpis študenta:

ZAHVALA

Ob zaključku diplomskega dela se zahvaljujem svoji družini, prijateljem in dekletu Mateji ter njeni družini, da so me podpirali in mi omogočali študij.

Zahvaljujem se mentorici Veroniki Saje, univ. dipl. inž. arh., ter mentorju v podjetju Mateju Butali za čas, ki sta mi ga namenila, da sta mi svetovala ter me vodila skozi pisanje diplomske naloge.

Zahvala velja tudi vsem profesorjem, ki so skozi moje šolanje ustvarjali prijeten prostor za sprejemanje znanja.

Posebna zahvala gre gospe Silvi Požlep, inž. multimedije, ki mi je v 1. in 2. letniku študija omogočila opravljanje obvezne delovne prakse, ter

Hvala lektorici Kseniji Pečnik, prof. slov., za slovnični pregled naloge.

POVZETEK

Diplomsko delo opisuje proces ustvarjanja aplikacije *Interaktivni prikaz vožnje v krožišču* od ideje do končnega izdelka. V začetnem delu je prikazan program Adobe Flash, njegova zgodovina ter opis bistvenih orodij, ki sem jih uporabil. Sledi jedro naloge, kjer sem postavil temelje za uspešen potek dela in nadaljeval z razlago delovanja posameznih elementov, ki sestavljajo krožišče. Zaključek predstavlja moje mnenje o končnem izdelku ter doseženem rezultatu nasploh.

Ključne besede:

Adobe Flash, programiranje logike, animacija, interaktivnost, pravilna vožnja v krožišču

ABSTRACT

This diploma thesis describes the process of creating the application *Interactive presentation of correct driving in the roundabout*, from the initial idea to the finished product. At the beginning I describe Adobe Flash, its history and the essential tools I used. What follows is the core of the thesis, where I lay down the foundation for a successful workflow and proceed to explain the function of the individual components that make up the roundabout. The conclusion presents my opinion on the final product and the achieved result in general.

Key words:

Adobe Flash, logic programming, animation, interactivity, roundabout, correct driving in a roundabout.

KAZALO

1 UVOD 11

1.1 NAMEN, CILJI IN OSNOVNE TRDITVE DIPLOMSKEGA DELA 12

1.1.1 Glavni cilji 12

1.1.2 Dodatni cilji 12

1.2 PREDPOSTAVKE IN OMEJITVE 13

1.3 PREDVIDENE METODE DELA 13

1.4 UPORABLJENE KRATICE 13

2 ADOBE FLASH CS4 14

2.1 KRATKA ZGODOVINA PROGRAMA FLASH 15

2.2 ACTIONSCRIPT 17

2.3 ORODJA 18

2.4 OBJEKTI 21

2.4.1 Risalni objekt (Drawing object) 21

2.4.2 Grafični simboli (Graphic symbols) 21

2.5 NAČINI ANIMACIJE 23

3 INTERAKTIVNO KROŽIŠČE 25

3.1 TEMELJI 25

3.1.1 Vektorske oblike ali bitne slike 25

3.1.2 Dimenzije v slikovnih pikah 26

3.1.3 Hitrost predvajanja animacije ali število slik na sekundo 26

3.1.4 ActionScript 27

3.1.5 Pravila vožnje v krožišču 27

3.2 KROŽIŠČE 28

3.2.1 Skica 28

3.2.2 Videz krožišča 29

3.2.3 Odvijanje v krožišču 30

3.2.4 Logika 31

3.3 AVTOMOBIL 33

3.3.1 Načrtovanje logike 33

3.3.2 Videz avtomobila 34

3.3.3 Različne preobleke 34

3.3.4 Animacija 36

3.3.5 Nastanek avtomobilov na različnih krakih *37*
3.3.6 Preverjanje trčenja (Collision Detection) *38*
3.3.7 Seznam avtomobilov *39*
3.3.8 Kolone *40*
3.3.9 Poenostavljen prikaz logike *42*
3.3.10 Interakcija *44*
3.3.11 Zajem gibov *45*
3.3.12 Dodaten avtomobil *46*
3.4 PEŠEC 47
3.4.1 Logika peščevih poti *47*
3.4.2 Podrobneje o peščevi logiki *49*
3.4.3 Hoja proti cilju *51*
3.4.4 Prehod za pešce *52*
3.4.5 Poenostavljen prikaz logike *55*
3.5 MAČKA 57
3.5.1 Izmikanje *58*
3.5.2 Sledenje miškinemu kazalcu *58*
3.5.3 Težava z globino *63*
3.5.4 Logika *64*
3.6 UPORABNIŠKI VMESNIK 66
3.6.1 Meni in gumbi *66*
3.6.2 Menjava jezika *67*
3.6.3 Seštevanje prometa *67*
3.6.4 Izbira avtomobilove smeri z gumbi *67*
3.7 ZVOK 69
3.7.1 Zvok krožišča *70*
3.7.2 Zvok mačke *72*
3.8 OPTIMIZACIJA 73
3.8.1 Počasna grafika *73*
3.8.2 Lov za hrošči *74*
4 ZAKLJUČEK 75
5 LITERATURA IN VIRI 77
5.1 LITERATURA 77
5.2 SPLETNI VIRI 77

6 SLOVAR TUJIH BESED 78

7 PRILOGE 80

KAZALO SLIK

Slika 1: Adobe Flash CS4 14
Slika 2: Časovna linija (*Timeline*) 18
Slika 3: Orodja (*Tools*) 18
Slika 4: Napake prevajalnika (*Compiler Errors*) 18
Slika 5: Lastnosti (*Properties*) 19
Slika 6: Knjižnica (*Library*) 19
Slika 7: Barva (*Color*) 19
Slika 8: Odtenki (*Swatches*) 19
Slika 9: Akcije (*Actions*) 20
Slika 10: Izpis (*Output*) 20
Slika 11: Poravnava (*Align*) 20
Slika 12: Informacije (*Info)* 20
Slika 13: Nenavadna časovna linija simbola *Button* 22
Slika 14: Prikaz pretvorbe vektorja s pomočjo *Shape Tween* 23
Slika 15: Prikaz animacije *Classic Tween* 23
Slika 16: *Motion Tween* animiramo z orodjem *Motion Editor* 24
Slika 17: Povečeva vektorske oblike 25
Slika 18: Povečava bitne slike 25
Slika 19: Glavne nastavitve dokumenta 26
Slika 20: Podrobnejša skica krožišča 28
Slika 21: Oblikovanje krožišča skozi čas 29
Slika 22: Različne barve rož 30
Slika 23: Senca letala 30
Slika 24: Statičen promet je vsakič enak 31
Slika 25: Dinamičen promet je vsakič drugačen 31
Slika 26: Kraki krožišča (0,1,2,3) 32
Slika 27: Razvoj videza avtomobila 34
Slika 28: Vse preobleke in dodatni vzorci 35
Slika 29: Razvoj smerokaza skozi čas od leve proti desni 35
Slika 30: Drobovje MC-a *CarSkin0* 35

Slika 31: »Tirnice«, po katerih je animiran avto 36
Slika 32: Različne rotacije avtomobila glede na krak nastanka 37
Slika 33: Mesta za preverjanje (0,1,2,3) 38
Slika 34: Zanka nastajanja in življenjske dobe avtomobila 40
Slika 35: Sistem sestavljene animacije vožnje v koloni 41
Slika 36: Vse kombinacije vožnje v koloni 42
Slika 37: Poenostavljen graf poteka avtomobilove logike v koloni 43
Slika 38: Smeri, ki jih lahko določimo avtu 44
Slika 39: Različno dojemanje smeri glede na izvorni krak avtomobila 44
Slika 40: Kako v Flashu merimo slikovne pike 45
Slika 41: Kako premik miške pretvorimo v smer 46
Slika 42: Videz pešca 47
Slika 43: Mreža poti 47
Slika 44: Logika peščevih ciljev za krak 0 48
Slika 45: Celoten prikaz logike peščevih ciljev 50
Slika 46: Kot med pešcem in ciljem 51
Slika 47: Kot iz slike 41 prenesen v krog 51
Slika 48: Prostor, kjer pešci prečkajo cestišče 53
Slika 49: Postavitev in poimenovanje prehodov za pešce 53
Slika 50: Peščevi tipali 54
Slika 51: Kako pešec prečka cestišče 55
Slika 52: Poenostavljen graf poteka peščeve logike 56
Slika 53: Videz mačke v teku 57
Slika 54: Mačkin brlog 57
Slika 55: Tipali mački sporočita, od kod prihaja nevarnost 58
Slika 56: Prikaz enakomernega in neenakomernega 180 ° obrata mačke 59
Slika 57: Težave pri računskem iskanju pravilnega obrata 60
Slika 58: Primer računanja pravilnega obrata, kjer sta *dir* in *dirNew* pozitivna (a in b) ali negativna (c in č) 61
Slika 59: Primer računanja pravilnega obrata, kjer je *dir* pozitiven in *dirNew* negativen 61
Slika 60: Primer računanja pravilnega obrata, kjer je *dir* negativen in *dirNew* pozitiven 61
Slika 61: Poenostavljen graf poteka logike za najdbo ustrezne prilagojene rotacije *dirTarget* 63
Slika 62: Težava z globino mačke 64
Slika 63: Poenostavljen graf poteka mačkine logike 65
Slika 64: MC *AboutScreen* vsebuje poleg vseh gumbov tudi meni ter prometne znake 66

Slika 65: Seštevanja prometa na kraku 67
Slika 66: Grafični prikaz izbrane poti – *naključno*, *prvi*, *drugi*, *tretji* in *četrti izhod* 68
Slika 67: Adobe Audition 1.5 69
Slika 68: Graf glasnosti glede na število vozil 71
Slika 69: Graf glasnosti glede na število vozil v obliki krivulje 71
Slika 70: Formula za glasnost 71
Slika 71: Povečan prikaz razlike v kakovosti izrisa – *high*, *medium* in *low* 73

KAZALO TABEL

Tabela 1: Formule za izračun peščevih ciljev 49
Tabela 2: Prikaz vrednosti funkcij sinus in kosinus glede na različne smeri v krogu 52
Tabela 3: Formuli za izračun gladkega premika oziroma obrata 59
Tabela 4: Izračun primerov b in c 60
Tabela 5: Dokazi za primere s slik 58, 59 in 60 62
Tabela 6: Seznam vseh zvokov za mačko 72
Tabela 7: Datoteke na priloženem DVD-ju 80

1 UVOD

Projekt, o katerem govori ta diplomska naloga, je zaživel kot obvezna naloga pri predmetu glavni multimedijski programi. Izdelati je bilo treba kratko animacijo v programu Adobe Flash CS4. Ker se s tem programom dokaj aktivno ukvarjam že od srednje šole naprej, sem tu videl priložnost, da bi namesto osnovne animacije naredil nekaj zahtevnejšega.

Pri iskanju ideje nisem imel sreče, zato sem za nasvet vprašal svoje dekle. Predlagala mi je prikaz pravilne vožnje v krožišču. Zdelo se mi je, da v obliki animacije takšno krožišče ne bi bilo zanimivo, saj bi se vožnja avtomobilov kaj hitro začela ponavljati. Da bi bila animacija zanimiva, bi moral vsak avtomobil voziti in izbirati pot skozi krožišče sam, in to po predpisih. Hkrati bi se to moralo dogajati vsakič drugače, torej naključno – nekakšen lahkoten približek simulacije vožnje avtomobilov v krožišču. Bolj kot sem o tem razmišljal, bolj navdušen sem postajal nad zamislijo. Vsekakor se je slišalo kot izziv, zato sem se lotil ustvarjanja.

Jasno je, da sem dobro poznal osnove programa Flash in njegovega skriptnega jezika ActionScript (v nadaljevanju AS), ki sem jih osvojil s pomočjo informacij na spletu, toda to so bile le osnove. Da bi lahko uspešno realiziral zamišljeno, bi se moral naučiti še veliko. Poleg tega pa sem s tem želel sporočiti tudi, da se v današnjem času da ogromno naučiti kar od doma. Za to sem potreboval verodostojen vir informacij, iz katerih sem lahko črpal potrebno znanje. Najboljši vir informacij o programu je gotovo priročnik, ki ga izda podjetje, ki je program naredilo. Za Flash ta priročnik obstaja v spletni obliki, in sicer kot center za pomoč, ki je za aktualno doslednost informacij posodobljen iz dneva v dan. Poleg tega pa so na spletu gradiva za učenje malodane vsepovsod, le voljo in motivacijo potrebujemo, da jih najdemo. In ker sem od srednje šole naprej srečal ogromno ljudi, ki se sami niso bili pripravljeni naučiti ničesar, da bi dosegli kaj več, sem bil še toliko bolj motiviran ustvariti nekaj zanimivega. Sledile so ideje, ki so bile osnova za cilje kot so zapisani v nadaljevanju.

Poudariti želim, da je krožišče, ki sem ga prikazal v svoji diplomski nalogi, posodobljena različica krožišča, ki sem ga oddal pri predmetu glavni multimedijski programi. Prav tako želim opomniti, da so vse slike in skice v diplomskem delu moje lastno delo.

Za lažje razumevanje diplomske naloge priporočam predhoden ogled izdelka, ki se nahaja na priloženem DVD-ju.

1.1 Namen, cilji in osnovne trditve diplomskega dela

Namen mojega diplomskega dela je ustvariti aplikacijo s programom Adobe Flash CS4, ki prikazuje pravilno vožnjo avtomobilov v krožišču ter vsebuje osnovno interaktivnost – uporabnik lahko posredno manipulira s potjo avtomobilov. Gre za celoten postopek izdelave multimedijskega projekta od načrtovanja do realizacije, v katerega so vključeni programiranje, grafika in zvok. Za stvarjenje vseh segmentov projekta sem poskrbel popolnoma sam, prav tako sem projekt tudi sam vodil, kar pomeni, da je moj vpliv prisoten v celotnem obsegu projekta.

Vnaprej napisani cilji so ključna opora, kamor se lahko vrnemo, ko ne vemo več, kaj smo želeli doseči. Zajemati morajo bistvene dele našega projekta, saj se po njih orientiramo.

1.1.1 Glavni cilji

Gre za cilje, ki sem jih dorekel še pred razvojem projekta. Ti elementi so pomembni za osnovno delovanje krožišča in predstavljajo bistvo tega projekta:

- ustvariti krožišče s štirimi kraki, od katerih ima vsak dva vozna pasova,
- ustvariti avtomobile, ki bodo znali pravilno voziti v krožišču, se postavljati v kolono ter paziti na druge avtomobile in pešce – njihov nastanek mora biti do neke mere naključen,
- ustvariti pešce, ki se bodo naključno sprehajali ob cestišču ter uporabljali prehod za pešce, s čimer bodo ovirali avtomobile – njihov nastanek mora biti prav tako do neke mere naključen,
- vključiti možnost avtomobilom dodeliti smer z uporabo miške in/ali tipkovnice,
- ustvariti meni oz. gumbe, ki bodo služili spreminjanju določenih nastavitev krožišča,
- ustvariti všečno grafiko in animacijo ter vključiti zvok v aplikacijo,
- optimizirati delovanje aplikacije.

1.1.2 Dodatni cilji

Ti cilji niso bili v mojem izvirnem načrtu, temveč so se mi porajali sproti. Gre za dodatne elemente, ki ne vplivajo na osnovno delovanje krožišča. Ti cilji so:

- ustvariti mačko, ki bo nastala na ukaz uporabnika in bo bežala pred avtomobili,
- vključiti možnost vodenja mačke z uporabo miške,
- ustvariti letalo, ki bo v naključnih intervalih preletelo krožišče,
- vključiti možnost spreminjanja jezika aplikacije.

1.2 Predpostavke in omejitve

Predvidevam, da s samo uporabo programa Adobe Flash CS4 ne bom imel težav, saj imam z njim že veliko izkušenj, prav tako pa je literatura na internetu prosto dostopna. Tako se bo največji izziv pokazal v samem oblikovanju aplikacije, oblikovanju algoritmov delovanja posameznih delov aplikacije ter smiselni uporabi skriptnega jezika AS, ki je za takšen podvig nuja. Največji izziv pa pričakujem v upodabljanju celotnega postopka izdelave aplikacije v besedilo.

1.3 Predvidene metode dela

Za teoretični del diplomskega dela bom uporabil:

- knjižne vire,
- podatke z interneta in
- lastne izkušnje.

Raziskovalni del moje diplomske naloge pa bo potekal v naslednjih korakih:

- načrt,
- izdelava,
- končni izdelek,
- analiza celotnega postopka izdelave in
- predstavitev izdelka.

1.4 Uporabljene kratice

AS – ActionScript

DVD – Digital Versatile Disc

FPS – Frames per second[1]

HZ – Hertz

MC – MovieClip

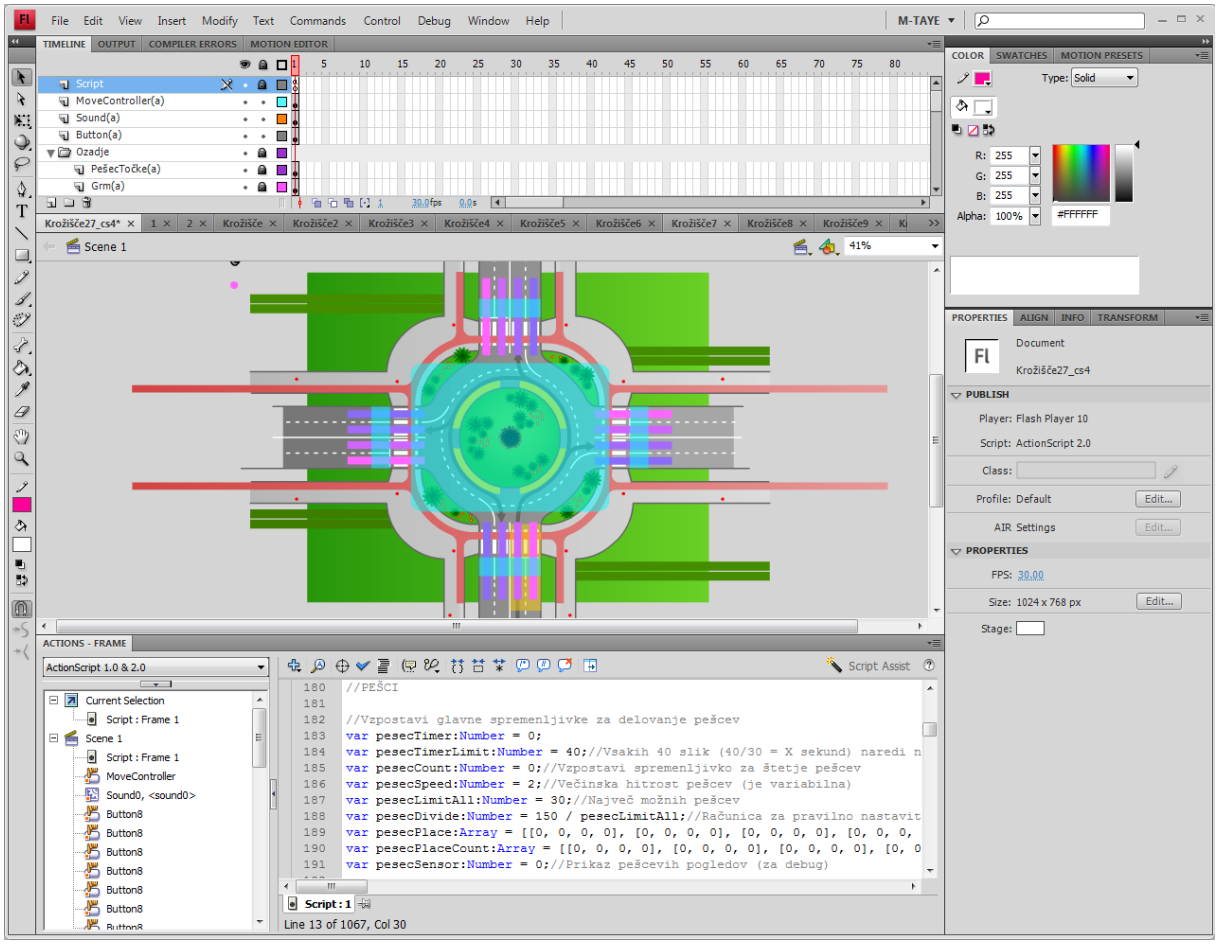
[1] Pove, koliko slik se izmenja v eni sekundi.

2 ADOBE FLASH CS4

Program Flash podjetja Adobe je namenjen ustvarjanju vsebine za splet. Z njegovo pomočjo lahko spletnim stranem dodamo animacije, video in interaktivnost. Gre za dandanes izredno priljubljeno spletno aplikacijo, ki jo glede na meritve, opravljene julija 2010, uporablja 99 % vseh spletnih deskarjev. Poleg samih spletnih strani se Flash v veliki meri uporablja tudi za oglaševanje in spletne igre (http://en.wikipedia.org/wiki/Adobe_Flash/, dne 15. 12. 2010).

Flash z manipulacijo vektorske in rastrske grafike omogoča animacijo teksta, risb in statičnih slik. Podpira dvostransko pretakanje zvoka in videa ter komunikacijo z uporabnikom preko miške, tipkovnice, mikrofona in kamere. Flash vsebuje objektno usmerjen programski jezik AS.

Več o programu Adobe Flash se da prebrati na:
http://www.adobe.com/products/flash/, dne 15.12.2010



Slika 1: Adobe Flash CS4

Vir: Lasten

2.1 Kratka zgodovina programa Flash

Predhodnik programa Flash je bila risalna aplikacija SmartSketch za operacijski sistem PenPoint OS. Po začetnem neuspehu na trgu je bil program preveden tudi za operacijska sistema Windows in Mac. S priljubljenostjo interneta je bil program SmartSketch posodobljen in izdan kot FutureSplash – vektorsko orodje za spletno animacijo, neposreden konkurent takratnemu Macromedia Shockwave. Leta 1995 je izšel FutureSplash Animator – posodobljena različica programa SmartSketch z izboljšavami pri delu z animacijo. Leto dni kasneje je FutureSplash pridobilo podjetje Macromedia in na njegovi podlagi izdalo Flash. Sledi seznam vseh inačic programa Flash do sedaj, in sicer s kratkim opisom sprememb, ki sem ga povzel po Wikipediji (http://en.wikipedia.org/wiki/Adobe_Flash/, dne 15. 12. 2010).

FutureSplash Animator (1996) - prvotna različica programa Flash s časovno linijo in osnovnimi orodji za urejanje.

Macromedia Flash 1 (1996) - program FutureSplash Animator s spremenjeno blagovno znamko.

Macromedia Flash 2 (1997) - izdan s Flash Player 2. Nove možnosti: zbirka objektov.

Macromedia Flash 3 (1998) - izdan s Flash Player 3. Nove možnosti: element MovieClip, vtična integracija jezika JavaScript, prozornost in zunanji samostojni predvajalnik vsebine.

Macromedia Flash 4 (1999) - izdan s Flash Player 4. Nove možnosti: notranje spremenljivke, polje za vnos teksta, napreden AS in možnost pretoka MP3.

Macromedia Flash 5 (2000) - izdan s Flash Player 5. Nove možnosti: AS 1.0 (zaradi zasnove na ECMAScript v skladnji zelo podoben jeziku JavaScript), podpora XML, objekti Smartclip (predhodniki Flashevih komponent), dodano oblikovanje besedila na osnovi HTML za dinamičen tekst.

Macromedia Flash MX (2002) - izdan s Flash Player 6. Nove možnosti: video kodek (Sorenson Spark), Unicode, nova različica komponent za uporabniški vmesnik (v1), kompresija, vmesnik za programiranje aplikacij za risanje vektorjev v AS.

Macromedia Flash MX 2004 (2003) - izdan s Flash Player 7. Nove možnosti: AS 2.0 (omogoči objektno usmerjen model programiranja v Flashu, toda le ob ročnem vnosu ukazov AS, saj ni bil združljiv z uporabo podpore programiranja), vedenja, sloj za razširitve (JSAPI), podpora tekstu, učinki časovne linije. Poleg vseh možnosti iz Flash MX 2004 je Macromedia Flash MX Professional 2004 vseboval še: zaslone (forme za nelinearno izdelavo glede na stanje ter diapozitive za organizacijo vsebin v linearni obliki kot pri programu PowerPoint), integracija spletnih storitev, čarovnik za uvoz videa, komponente za predvajanje medijev (vsebujejo celoten predvajalnik MP3 ali FLV, odet v obliko komponente, ki se vstavi v SWF), komponente za podatke (DataSet, XMLConnector, WebServicesConnector, XupdateResolver itd.) ter vmesnike za programiranje aplikacij, ki jih povezujejo, zavihek za projekte, novo različico komponent za uporabniški vmesnik (v2) in zbirko prehodov.

Macromedia Flash 8 (2005) - Macromedia Flash Professional 8 s Flash Player 8 vsebuje dodatke, osredotočene na izrazitost, kakovost, video in mobilno založništvo. Dodani so bili filtri in načini prelivanja med objekti, nova orodja za animacijo, način risanja objektov, dodatne lastnosti pri risanju, izboljšano mehčanje robov teksta, On2 VP6 napreden video kodek, podpora za prosojnost v videu, samostojni enkoder in napredni uvoznik videa, napredno komponento za predvajanje videa, interaktivni emulator mobilnih naprav ter še vrsto drugih novosti ter izboljšav.
Hkrati je izšla tudi različica Macromedia Flash Basic 8, namenjena novim uporabnikom, ki želijo uporabljati le osnove risanja, animacije in interaktivnosti. Ta različica je omejena, zato ji manjka določena podpora za video in napredne grafične ter animacijske učinke.

Adobe Flash CS3 Professional (2007) - Flash CS3 je bila prva različica Flasha izdana pod blagovno znamko Adobe. CS3 omogoča pretvorbo celotnih aplikacij v AS, vsebuje polno podporo AS 3.0, izboljšano integracijo z ostalimi programi Adobe ter posodobljeno risanje vektorjev.

Adobe Flash CS4 Professional (2008) - vsebuje inverzno kinematiko (*Bones*), osnovno manipulacijo 3D-objektov, animacijo na osnovi objektov, nov pogon za tekst in dodatne širitve k AS 3.0. Dodana so bila tudi nova orodja za uporabo pri ustvarjanju animacij.

Adobe Flash CS5 Professional (2010) - vsebuje podporo za založništvo aplikacij za iPhone. Med drugimi novostmi je tu nov pogon za delo s tekstom, dodatne izboljšave k inverzni kinematiki ter nov zavihek koščki kode.

2.2 ActionScript

AS je skriptni programski jezik, ki se v glavnem uporablja za razvoj spletnih strani in programske opreme za platformo Adobe Flash Player. V originalu ga je razvijalo podjetje Macromedia, a si ga zdaj lasti podjetje Adobe (ki je pridobilo podjetje Macromedia v letu 2005).

ActionScript 1.0

Začetna inačica jezika AS je bila načrtovana za Macromedia Flash in je bila zelo omejena, saj se je uporabljala za vodenje preprostih 2D-vektorskih animacij. Posodobitve so v kasnejših različicah omogočile izdelavo spletnih iger ter bogatih spletnih aplikacij s pretakajočim se videom in zvokom.

ActionScript 2.0

Flash MX 2004 je prinesel skriptni programski jezik AS 2.0, ki je bil bolje prilagojen razvoju aplikacij v Flashu. Velikokrat poskušamo nekaj animirati na roko, a se izkaže, da je to v resnici laže sprogramirati z uporabo AS-a. Tak način pri urejanju večino časa nudi tudi večjo fleksibilnost.

ActionScript 3.0

Leta 2006 je s prihodom platforme Flash Player 9 alpha izšla tudi nova inačica skriptnega programskega jezika AS. AS 3.0 je objektno usmerjen programski jezik, ki ponuja precej večji nadzor in ponovno rabo kode, s čimer se da ustvariti zapletene Flash aplikacije. V AS-u 3.0 se koda izvrši tudi do desetkrat hitreje kot v prejšnjih različicah.

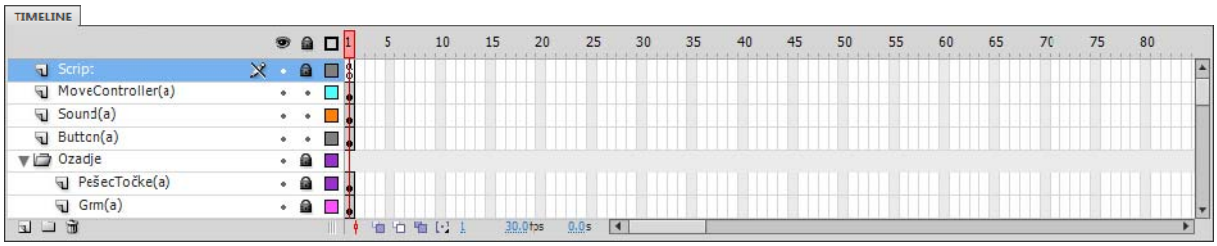
Več o AS-u in njegovem delovanju se da izvedeti na:
http://livedocs.adobe.com/flash/9.0/main/flash_as2_learning.pdf, dne 15. 12. 2010.

2.3 *Orodja*

V tem poglavju so omenjena in opisana samo tista orodja, ki sem jih uporabil pri izdelavi krožišča.

Časovna linija (*Timeline*)

Flash se veliko uporablja za ustvarjanje animacij, zato ni čudno, da imamo možnost premika skozi čas. Gre za dodatno dimenzijo, na katero je treba misliti ob ustvarjanju projekta, in časovna linija je orodje, ki nam jo pomaga vizualizirati.



Slika 2: Časovna linija (*Timeline*)

Vir: Lasten

Orodja (*Tools*)

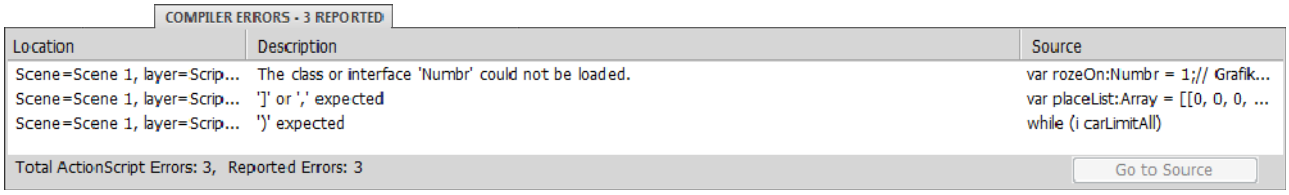
Na tem mestu se nahajajo orodja, ki jih potrebujemo za delo z grafiko.

Slika 3: Orodja (*Tools*)

Vir: Lasten

Napake prevajalnika (*Compiler Errors*)

Tukaj nam Flash javi, kje v kodi so se pojavile napake. Te napake so največkrat rezultat površnosti pri pisanju kode, velikokrat pa nam tudi sporočajo, da stvari, ki smo si jih zamislili, enostavno ne morejo tako delovati.

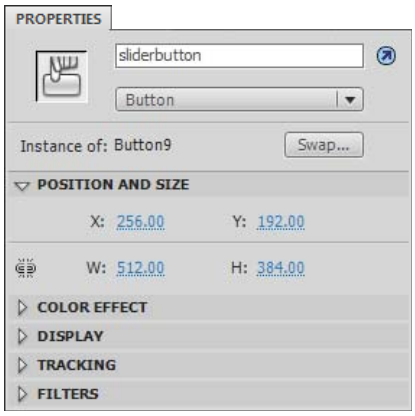


Slika 4: Napake prevajalnika (*Compiler Errors*)

Vir: Lasten

Lastnosti (*Properties*)

V tem oknu lahko upravljamo z lastnostmi označenih objektov.

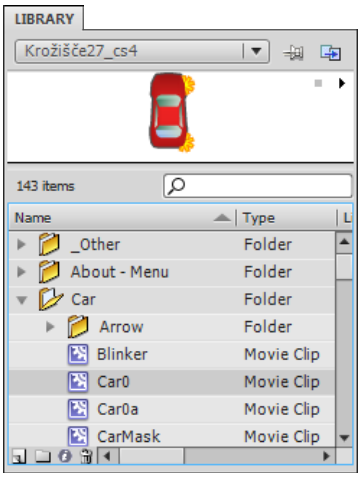


Slika 5: Lastnosti (*Properties*)

Vir: Lasten

Knjižnica (*Library*)

V knjižnico se shranjujejo ustvarjeni simboli.

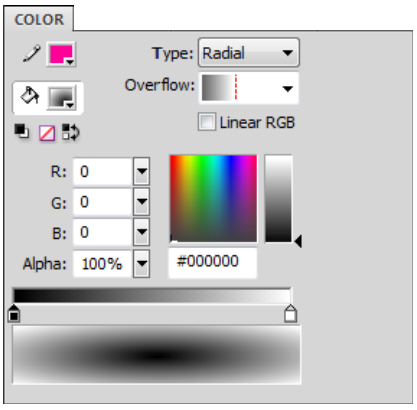


Slika 6: Knjižnica (*Library*)

Vir: Lasten

Barva (*Color*)

Vsa izbira barv in njihovo mešanje se dogaja tukaj. Obroba (S*troke)* in polnilo (*Fill)* imata ločene nastavitve.

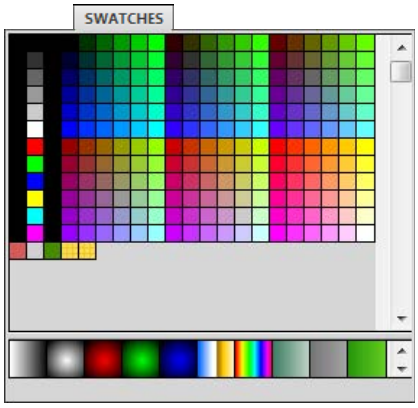


Slika 7: Barva (*Color*)

Vir: Lasten

Odtenki (*Swatches*)

To so že obstoječe mešanice barv, kjer lahko ustvarimo tudi lastne odtenke.

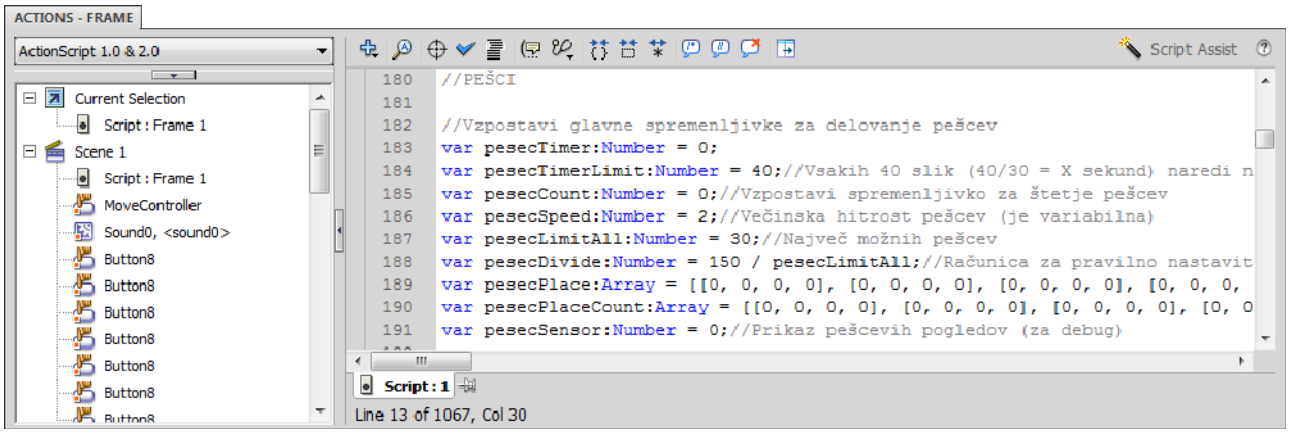


Slika 8: Odtenki (*Swatches*)

Vir: Lasten

Akcije (*Actions*)

To je mesto, kjer pišemo kodo za AS. Za boljšo preglednost se napisan tekst obarva glede na pomen.

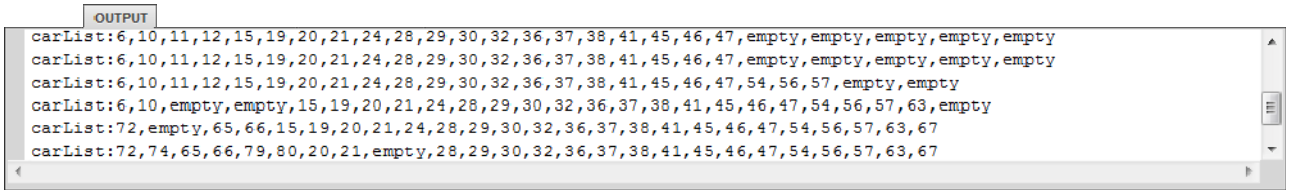


Slika 9: Akcije (*Actions*)

Vir: Lasten

Izpis (*Output*)

S pomočjo funkcije `trace()` lahko na preprost način preverimo, kaj se dogaja v naši kodi. Pričakovane informacije se zapišejo v tem oknu.

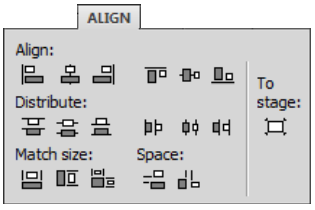


Slika 10: Izpis (*Output*)

Vir: Lasten

Poravnava (*Align*)

Priročno orodje za enostavno razporeditev označenih objektov glede na druge objekte.

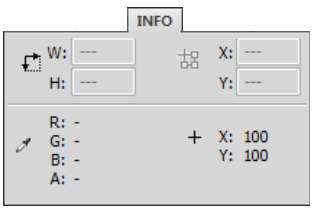


Slika 11: Poravnava (*Align*)

Vir: Lasten

Informacije (*Info*)

Ko potrebujemo natančnost pri delu s postavitvijo in velikostjo objekta, nam te informacije pridejo zelo prav.



Slika 12: Informacije (*Info*)

Vir: Lasten

2.4 Objekti

Vektorje, ki jih narišemo v Flashu, lahko pretvorimo v objekte. Ti nam nudijo naprednejše funkcije urejanja in animacije. Teh objektov je več vrst, in čeprav se med seboj razlikujejo, so nekatere lastnosti, ki jih imajo, enake. Uporabnejše med njimi so:

- **Položaj** *(Position)* je izjemno uporabna lastnost, še posebej če želimo biti pri postavitvi objektov karseda natančni. Nastavljamo lahko vrednosti za obe osi (x, y) in to v slikovnih pikah (*pixels*).
- **Širina** in **višina** *(Width and Height)* objekta se prav tako nastavljata v slikovnih pikah.
- **Rotacija** (Rotation) je izražena v stopinjah. Z njeno pomočjo lahko objekt vrtimo okoli svoje osi oziroma nastavljene točke.

2.4.1 Risalni objekt (Drawing object)

Flashev risalni model navadno deluje na principu spojitve – vsaka nova črta, ki jo narišemo, se spoji s prejšnjo in skupaj postaneta en grafični element. Risalni objekt je alternativa, ki zagotovi, da je vsaka nova poteza risanja svoj lasten objekt.

2.4.2 Grafični simboli (Graphic symbols)

Grafični simbol je t. i. zabojnik, v katerega lahko vstavimo grafiko, sliko ali celo animacijo, saj vsak od teh objektov do neke mere poseduje svojo časovno linijo. Največja prednost simbola je dejstvo, da ga lahko uporabimo ponovno na več koncih naše animacije. Priročnost se tako pokaže v tem, da nam potem, ko želimo narediti spremembe, ni treba spremeniti vsakega simbola posebej, temveč naredimo spremembe samo v enem in avtomatsko se pokažejo na vseh drugih. Vsak od teh objektov poseduje svojo časovno linijo, ki pa od simbola do simbola deluje drugače.

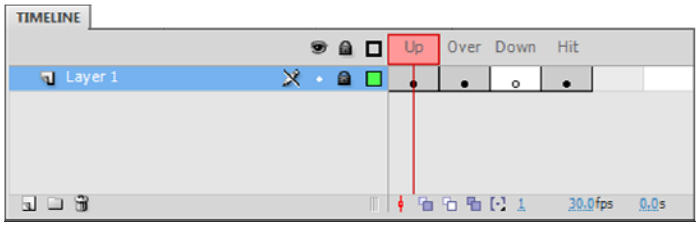
V Flashu lahko ustvarimo tri vrste grafičnih simbolov, od katerih ima vsak svoje prednosti. To so *Graphic*, *MovieClip* (v nadaljevanju MC) in *Button*.

Graphic je edini simbol, ki mu ne moramo dati imena, saj po njem ni potrebe. *Graphic* za delovanje namreč ne koristi skriptnega jezika AS, tako da mu ne moremo spreminjati lastnosti s pomočjo kode. Časovna linija simbola *Graphic* je enaka kot glavna časovna linija dokumenta z eno pomembno spremembo. Če se odločimo na glavni časovni liniji ustaviti animacijo, se bo ustavila tudi animacija vseh naših objektov *Graphic*. Če tega ne želimo, lahko uporabimo MC, katerega animacija se privzeto predvaja v nedogled, saj se ponavlja.

Čeprav kot rečeno sam *Graphic* ne podpira AS, ga pa lahko vnesemo na njegovo časovno linijo, kar je dobro vedeti.

MovieClip je med simboli tisti, ki ima največjo funkcionalnost. Tako kot *Graphic* tudi MC poseduje sebi lastno časovno linijo, ki pa se privzeto predvaja znova in znova, tudi če nam glavna časovna linija miruje. Ker je popolnoma integriran v AS, lahko s pomočjo kode z njim počnemo stvari, ki jim z ročno animacijo ne bi bili kos. Pomembni dejavnik pri njegovi učinkovitosti je tako tudi možnost interakcije oziroma vplivanja na njegovo delovanje v realnem času. Na tak način lahko naredimo dinamične in vedno znova spreminjajoče se vsebine, ki omogočijo vključitev samega uporabnika. Določenih skriptnih ukazov, ki jih poseduje *Button*, MC ne poseduje, toda z malo razmišljanja se dajo doseči enaki, če ne boljši rezultati.

Button ima časovno linijo, ki je popolnoma drugačna od časovne linije v zgoraj omenjenih simbolih in to je pravzaprav njegova prednost. Njegova časovna linija ima namreč samo štiri mesta, kar ga naredi izredno preprostega za uporabo. Kot simbol lahko sprejema ukaze AS-a, toda ne moramo jih vpisati neposredno na njegovo časovno linijo.



Slika 13: Nenavadna časovna linija simbola *Button*

Vir: Lasten

2.5 *Načini animacije*

Okvir ob okvirju (*Frame by frame*)

To je način, kjer vsako sliko animacije izrišemo ročno, tako kot klasične Disneyeve risanke. Ta način nudi največji nadzor nad animacijo, a hkrati vzame največ časa. Uporabimo lahko tako čiste vektorje kot grafične simbole.

Animacija oblike (*Shape Tween*)

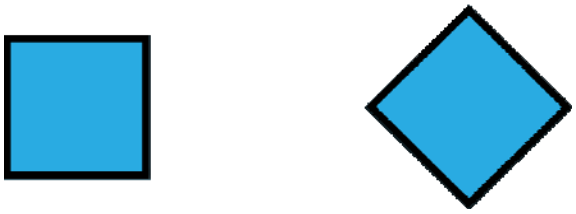
Gre za način animacije, kjer vektorje skozi čas pretvorimo v druge vektorje. Tehnika je precej podobna tehniki *Morphing*. S tem načinom ne moremo animirati grafičnih simbolov, temveč le čiste vektorje.

Slika 14: Prikaz pretvorbe vektorja s pomočjo *Shape Tween*

Vir: Lasten

Klasična animacija gibanja (*Classic Tween*)

To je način animacije, pri katerem z malo dela dosežemo veliko. Deluje po principu ključnih slik, ki jih postavimo in animiramo ročno, nakar Flash vmesno animacijo ustvari sam. Z njim lahko animiramo samo grafične simbole.

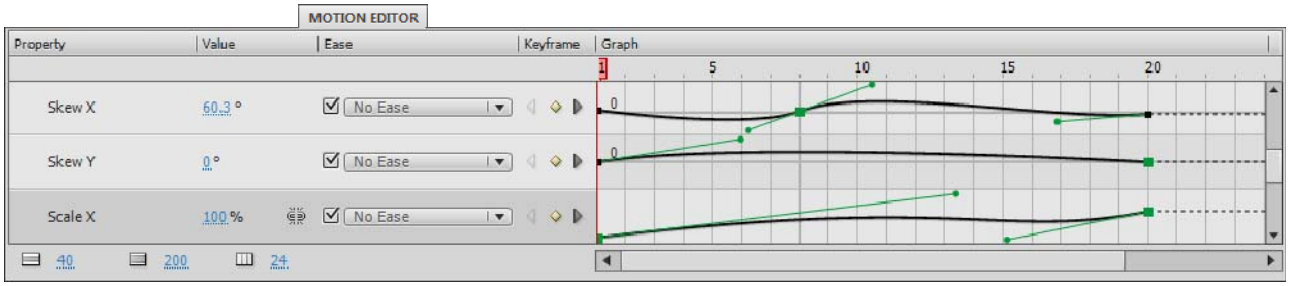


Slika 15: Prikaz animacije *Classic Tween*

Vir: Lasten

Animacija gibanja (*Motion Tween*)

V primerjavi s *Classic Tween* nudi naprednejši sistem animacije s krivuljami, ki določajo hitrost in manipulacijo posameznih delov objekta, ločeno od nastavitev na časovni liniji. Ima pa v času pisanja *Motion Tween* eno nevšečnost. Ko z njim animiramo premikanje objekta, ga ni možno mehko ustaviti oziroma pospešiti, vendar lahko kot pri *Classic Tween* to storimo neposredno na časovni liniji. Tudi tukaj lahko animiramo samo grafične simbole.



Slika 16: *Motion Tween* animiramo z orodjem *Motion Editor*

Vir: Lasten

Pot gibanja (*Motion Guide*)

Tukaj ne gre za način animacije, temveč za pripomoček, s katerim določimo pot animacije objekta. Pot ustvarimo tako, da jo narišemo z uporabo Flashevih risalnih sredstev. Uporabimo lahko tako krivulje kot tudi vektorje, narisane s prosto roko. Objekt se te poti oprime in se po njej vozi kot po tirnicah. Na tak način lahko ustvarimo privlačnejše animacije.

3 INTERAKTIVNO KROŽIŠČE

Preden sem začel delati v Flashu, sem moral najprej narediti načrt. Ta načrt je vključeval splošne cilje, skice posameznih delov projekta in zapis drugih pomembnih odločitev, ki se skozi ustvarjanje ne bi spreminjale.

3.1 Temelji

Flash dokument vsebuje pomembne lastnosti, ki se skozi nadaljnje ustvarjanje naj ne bi več spreminjale. S tem, ko jih dorečem, ustvarim temelje, s katerimi se že na začetku obvarujem pred morebitnimi slabimi odločitvami, ki bi me lahko kasneje ovirale pri izdelavi projekta. Te lastnosti so na primer: ali za grafiko uporabiti vektorje ali bitne slike, kakšni bosta resolucija in hitrost predvajanja animacije itd.

3.1.1 Vektorske oblike ali bitne slike

Bitne slike in vektorske oblike sta glavni vrsti grafike v programu Adobe Flash. Bitna slika (*Bitmap*) opredeljuje sliko kot mreža barvnih točk in shrani barvo za vsako slikovno piko na sliki. Vektorska oblika je matematični opis geometrijske oblike.

Bitne slike lahko vsebujejo veliko podrobnosti, vendar se ob njihovi povečavi opazijo nazobčani robovi. Vektorji, čeprav običajno vključujejo manj podrobnosti kot bitne slike, te težave ne poznajo.

Slika 17: Povečeva vektorske oblike

Vir: Lasten

Slika 18: Povečava bitne slike

Vir: Lasten

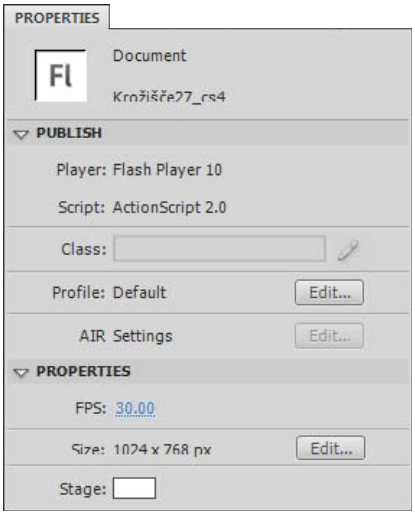
Izbral sem vektorje, saj sem si želel čisto in ostro grafiko, ne glede na resolucijo. Slabost tega pristopa je, da je treba paziti na zapletenost vektorskih oblik, saj njih izris zahteva večjo količino procesorske moči kot pa izris bitnih slik.

3.1.2 Dimenzije v slikovnih pikah

Flash omogoča prikaz animacije v kakršni koli resoluciji ne glede na izvorno dimenzijo. Sam sem se odločil za resolucijo 1024 x 768 slikovnih pik, saj je pri večji resoluciji prikaz grafike ostrejši. Za izris višje resolucije prav tako potrebujemo več procesorske moči.

3.1.3 Hitrost predvajanja animacije ali število slik na sekundo

V kolikor delamo z animacijo, je to verjetno najpomembnejša odločitev, saj diktira njeno dolžino. Če želimo, da animacija traja eno sekundo, zanjo pri 30 fps potrebujemo 30 posameznih sličic (*frames*). Če bi v tem primeru animirali 60 sličic, bi dobili dve sekundi animacije. Če pa se odločimo spremeniti hitrost predvajanja animacije s 30 fps na 60 fps, ponovno dobimo eno sekundo animacije. Takšna sprememba lahko razdre delovanje projekta, zato moramo to odločitev doreči že na začetku projekta.



Slika 19: Glavne nastavitve dokumenta

Vir: Lasten

Z namenom, da sem prišel do dobre odločitve, sem moral vzeti v zakup gladkost animacije glede na osvežitev zaslona. Večina LCD-monitorjev privzeto osvežuje oziroma izrisuje sliko s frekvenco 60 hz. Za gladek prikaz animacije brez cukanja je potrebno imeti takšno hitrost izrisa, da dobimo celo število, če z njo delimo 60 (zaradi 60 hz). Števila, ki so tako primerna za monitorje s frekvenco 60 hz, so 60, 30, 20, 15, 12, 10, 5, 3, 2, 1. V primeru 30 fps (60/30 = 2) je animacija prikazana vsak drugi izris slike na monitorju. Poskusil sem s števili manjšimi od 30, vendar se mi potem animacija ni zdela dovolj lepa oziroma gladka. Poskusil sem tudi 60 fps, kar je najbolj gladko, saj se animacija osveži vsak izris slike na monitorju, vendar bi glede na 30 fps tako moral animirati preveliko količino sličic. Poleg omenjenega pa sem

gledal še na količino podrobnosti, ki jih lahko računalnik izriše v sekundi. Enaka količina podrobnosti na ekranu namreč pri 60 fps za gladek izris zahteva močnejši računalnik kot pri 30 fps, in ker je risanje zapletenih vektorskih oblik procesorsko precej požrešno, sem zato moral gledati tudi na to. Tako sem se odločil za hitrost 30 fps.

3.1.4 ActionScript

V času ustvarjanja projekta z AS 3.0 še nisem bil seznanjen, zato sem se raje odločil uporabiti različico 2.0, s katero sem takrat že imel nekaj izkušenj. Namreč, obtičati v luknji neznanja brez možnosti nadaljevanja je bila moja zadnja želja.

3.1.5 Pravila vožnje v krožišču

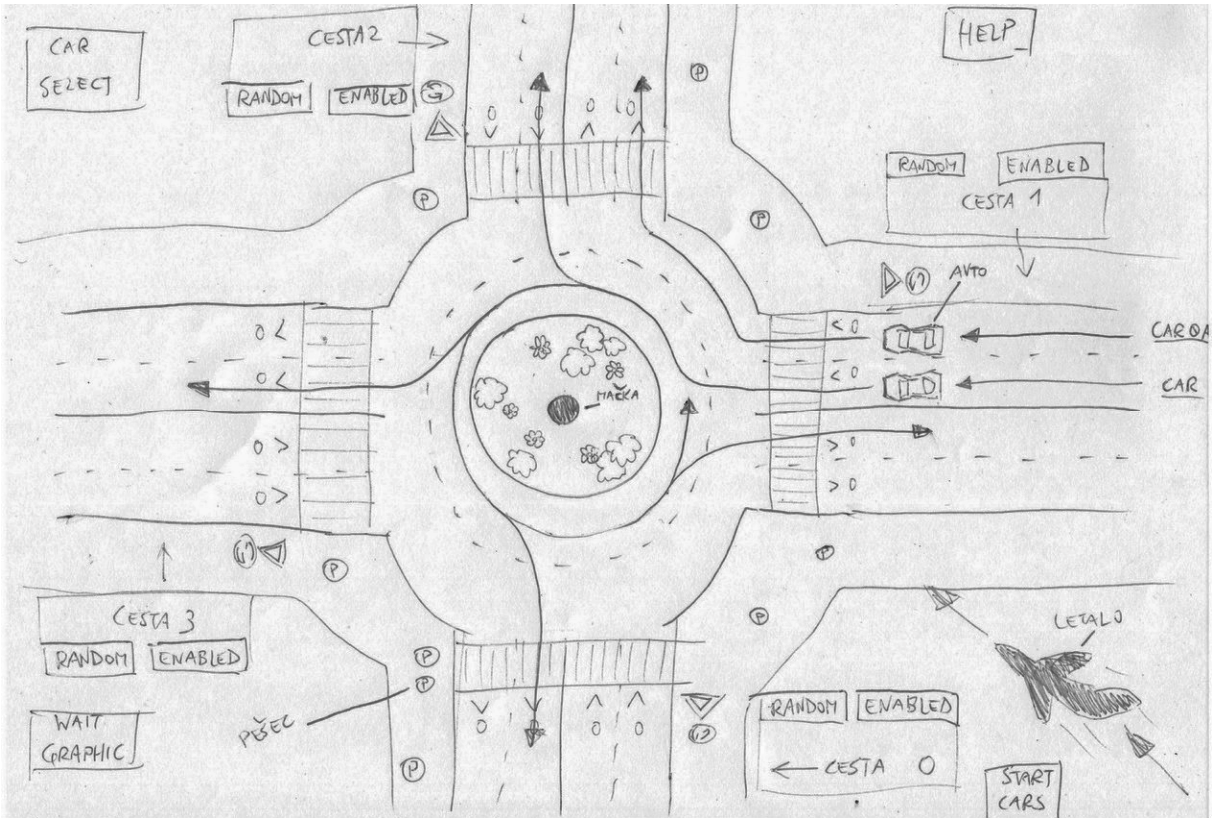
Čeprav smo vozniki s pravili v prometu dobro seznanjeni, sem za vsak slučaj preveril zakonitosti vožnje v krožišču. Za tiste, ki teh zakonitosti ne poznajo oziroma so jih pozabili, predlagam branje Zakona o pravilih cestnega prometa, natančneje, člena o vožnji v krožišču: http://www.mzp.gov.si/fileadmin/mzp.gov.si/pageuploads/DPR/Predlogi_predpisov_DPR/10_01_20-Zakon_o_pravilih_cestnega_prometa.pdf, dne 15. 12. 2010.

3.2 Križišče

Premišljeval sem o tem, kakšno naj bo križišče, ki ga bom upodobil in skozi katerega se bo odvijal promet. Spraševal sem se, koliko krakov naj ima križišče in ali naj bo cesta enopasovna ali dvopasovna. Po razmisleku sem izbral križišče s štirimi kraki, od katerih ima vsak dva vozna pasova. S štirimi kraki je križišče lepo simetrično, iz česar sem sklepal, da mi bo olajšalo ne samo risanje križišča, temveč tudi ustvarjanje vožnje avtomobilov skozenj. Dva vozna pasova pa sem izbral zaradi težnje po zanimivem dogajanju. Izmed teh voznih pasov bi namreč vsak potreboval malce drugačno logiko.

3.2.1 Skica

Že od začetka sem imel v mislih, da bo križišče vidno iz ptičje perspektive, saj se mi je ta zorni kot na dogajanje zdel najbolj pregleden.



Slika 20: Podrobnejša skica križišča

Vir: Lasten

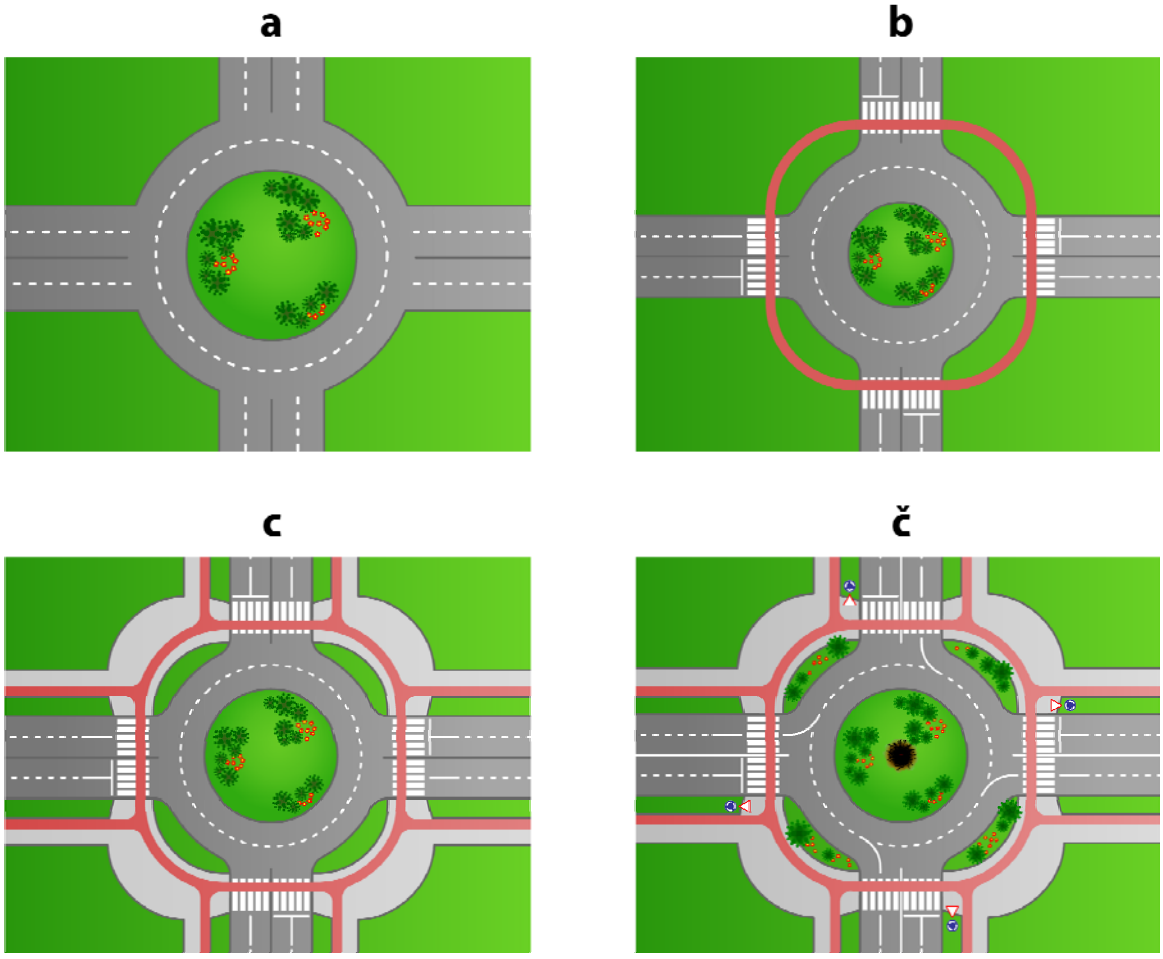
Za začetek sem potreboval le grob tloris križišča, zato se pri risanju skice nisem obremenjeval s podrobnostmi. Ker sem želel, da je skica jasna, sem moral ob vsaki spremembi križišče skicirati znova, zato sem v tem trenutku skušal upodobiti le najnujnejše

elemente. Vključil sem tudi nekatere predvidene interaktivne elemente, kot so gumbi in meniji.

3.2.2 Videz krožišča

Pri ustvarjanju grafike sem se trudil, da bi bil videz krožišča čim bolj jasen. Z vektorji sem oblikoval čiste poteze, pri katerih sem si pomagal z načeli simetrije. Videza nisem dokončal takoj, temveč sem ga neprestano izpopolnjeval.

V prvi izvedbi (*slika 21a*) sem upodobil cestišče in rastlinje ter ustvaril temeljno grafično vzdušje. Kasneje (*slika 21b*) sem dodal prehod za pešce ter zametke kolesarske poti. Zgladil sem tudi ostre robove med krožiščem in cesto, ki pelje vanj. Naslednja različica (*slika 21c*) je imela dodan pločnik ter kolesarsko pot. Za konec (*slika 21č*) so mi poleg dodatnega rastlinja ostale samo še malenkosti v obliki prometnih znakov in črt, ki avtomobile na desnem pasu prisilijo, da obvezno zapustijo krožišče. Prav tako sem bil proti koncu prepričan, da bom vključil mačko, zato sem ji na sredi krožišča ustvaril brlog.



Slika 21: Oblikovanje krožišča skozi čas

Vir: Lasten

Zaradi natančne izdelave grafike sem doživel številne nevšečnosti s postavitvijo elementov. Kaj hitro se mi je zgodilo, da elementi več niso bili pravokotni eden na drugega, s čimer sem ogrozil simetričnost izdelka. Ker je v samem krožišču prevladovala simetrija, sem si dal toliko več duška pri asimetrični postavitvi rastlinja.

Rože

Pri tem so izstopale rože, ki so ob vsakem zagonu aplikacije dobile drugo barvo. Dejansko posedujejo pet različnih preoblek, ki se izbirajo naključno. Poleg omenjenega sem jih tudi animiral, in sicer tako, da so videti kot da veselo skačejo.

Slika 22: Različne barve rož

Vir: Lasten

Letalo

Ob tej priliki lahko omenim še letalo, ki je v resnici samo senca, ki vsake toliko časa preleti ekran. Njegovo nastajanje je do neke mere naključno.

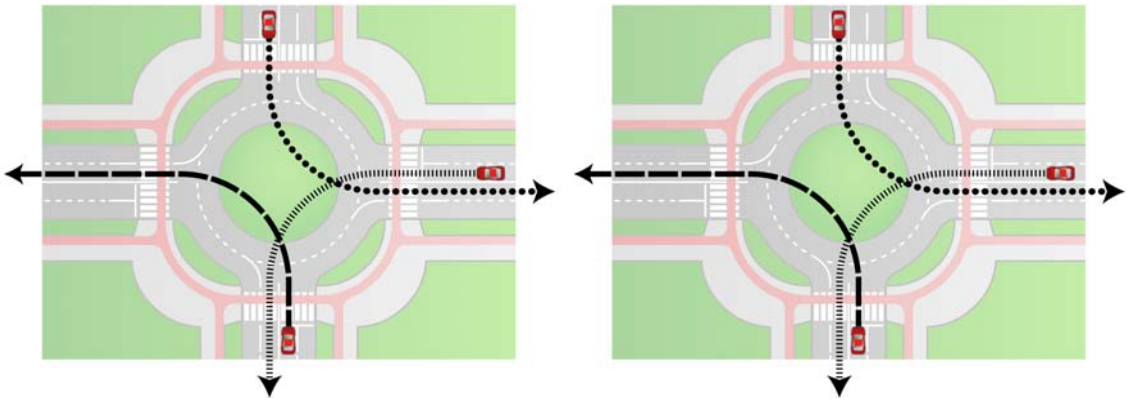
Slika 23: Senca letala

Vir: Lasten

3.2.3 Odvijanje v krožišču

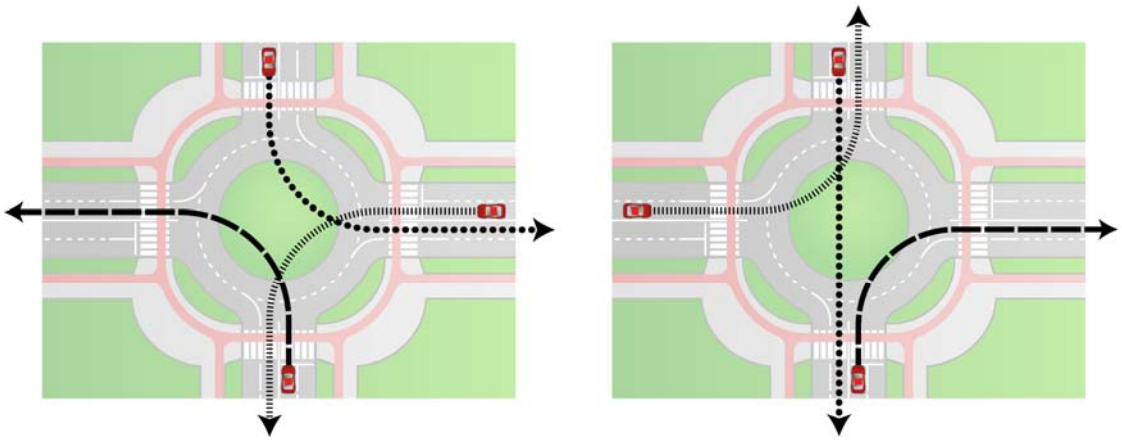
Ali bi se dalo krožišče narediti kot eno dolgo animacijo? Seveda, toda slej ko prej bi se začela ponavljati in ko bi v isti minuti že tretjič videli popolnoma enako kolono avtomobilov, ki ponovno uberejo iste izhode, to enostavno ne bi bilo več zanimivo. S stališča načrtovanja bi bil ta način relativno enostaven, le nekaj zbode v oči – vožnjo posameznega avtomobila in pešca bi bilo treba animirati na roko, kar je v tem primeru izredno neučinkovito, saj je krožišče zanimivo prav z velikim številom avtomobilov. Poleg tega pa je animirati veliko število premikajočih se objektov ročno herkulski podvig.

Nasprotje je dinamičen promet, kjer bi bil nastanek avtomobilov in pešcev povsem naključen. Ustvarjal jih torej ne bi ročno, temveč bi obstajal sistem, ki bi jih ustvarjal na določene intervale. Animacija avtomobilov bi sicer bila pripravljena vnaprej, toda avtomobil bi vsakič ubral drug pristop skozi krožišče. To daje dobre temelje za vključitev interakcije, saj nismo vezani na vedno isto animirano vožnjo vseh avtomobilov.



Slika 24: Statičen promet je vsakič enak

Vir: Lasten



Slika 25: Dinamičen promet je vsakič drugačen

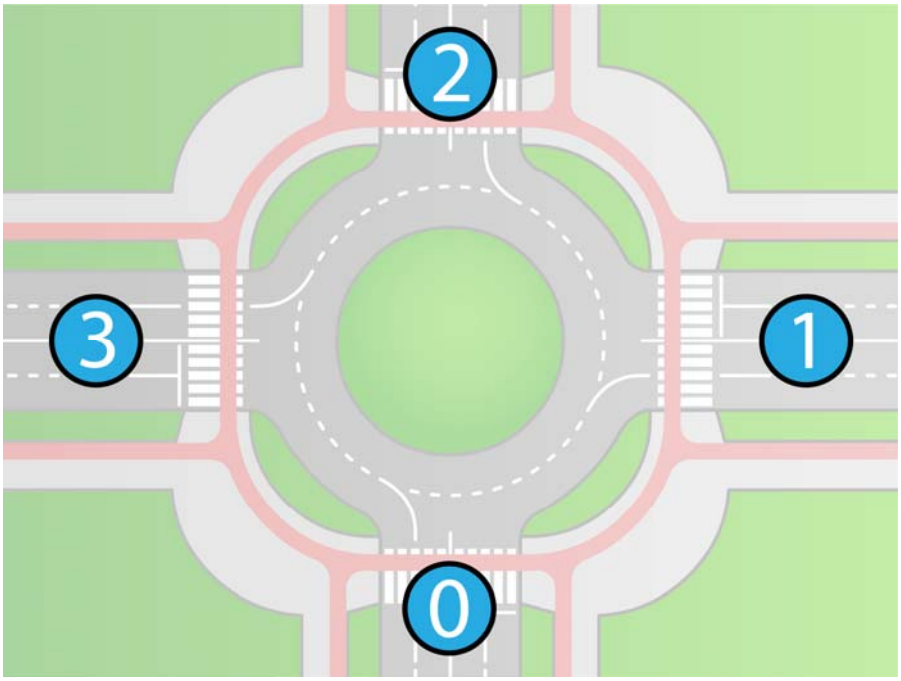
Vir: Lasten

Za izvedbo takšnega pristopa potrebujemo večji premislek kot za prvo rešitev, kajti naključnost v logiki lahko povzroči veliko glavobolov, saj se stvari ne zgodijo vedno enako. Pristop, ki sem ga izbral sam, je dinamičen pristop, kar pomeni, da sem si moral izmisliti logiko, ki bi ga vodila.

3.2.4 Logika

Pri programiranju logike nekako ni najboljših rešitev, še posebej če se programiranja šele učimo. Vsak problem se da namreč rešiti na vrsto načinov in ko se nam zdi, da smo našli najboljšega, se ob kasnejšem vnovičnem pregledu kode zavemo, da bi se to dalo narediti še bolje. Temu postopku tako imenovanega prečiščevanja kode nikdar ni konca, saj znova in znova najdemo dele, kjer bi se dalo kaj izboljšati. Seveda to ne pomeni, da to, kar naredimo, ni učinkovito. Kar želim povedati, je, da si moramo sami postavljati meje, tako da nam določen del, ki mogoče sploh ni tako pomemben, ne odvzame prevelike količine časa, ki bi ga lahko uporabili za kaj drugega.

Načrtovanje logike in raznih algoritmov mi je bilo že od nekdaj zanimivo in ker sem s programiranjem že imel nekaj izkušenj, sem vedel, da če skušamo nekaj zapletenega spraviti v kodo, se je tega najbolje lotiti s poenostavitvijo. Na tak način iz osnove izluščimo elemente, ki so ključni za delovanje naše logike. V primeru krožišča sem skušal nekaj, kar je analogno, ročno prenesti v digitalno obliko. Pri tem sem stremel k jasnosti, preprostosti in učinkovitosti. Krožišče sem si tako razdelil na mesta, kjer bi nastajali avtomobili. Ta mesta so bila števila, saj se mi je to zdelo najučinkovitejše, čeprav mogoče malce nepregledno (*slika 26*). Torej, avtomobili bodo nastajali sproti, in to naključno.



Slika 26: Kraki krožišča (0,1,2,3)

Vir: Lasten

3.3 Avtomobil

Preden govorimo o naključnem nastanku avtomobilov, moramo najprej vedeti tudi nekaj o samih avtomobilih, in sicer, kako bo delovala njihova logika.

3.3.1 Načrtovanje logike

V programiranju obstaja praksa – ponavljajoče se kode ne pišemo večkrat, vendar jo poskušamo napisati tako, da se jo da uporabiti znova in znova. Upoštevajoč ta pristop in željo, da bi v kratkem času ustvaril kar največ, sem se odločil za samo en model avtomobila. Vsi avtomobili, ne glede na to iz katere smeri pridejo v krožišče, se v osnovi držijo enakih pravil in imajo v zvezi s prometom enako funkcionalnost, zato bi bilo lepo imeti eno univerzalno vozilo, ki bi ga lahko pomnožil in tako dobil celo armado vozil, od katerih bi vsako peljalo po svoje. Dobiti idejo je seveda preprosto, težji del je realizacija.

Za začetek se mi je zdelo najbolje poenostaviti vožnjo avtomobila skozi krožišče:

1. Avto se pripelje in se ustavi pred krožiščem.
2. Na tej točki preveri, ali se lahko vključi v krožišče. Če se ne more vključiti, počaka, dokler nima proste poti, nakar zapelje v krožišče.
3. Avto vozi skozi krožišče, dokler ne najde želenega izhoda.
4. Ko ga najde, mu preostane le, da odpelje iz krožišča.

Za nas je zgornji tekst razumljiv, toda ne za Flash. To, kar sem napisal, je treba pretvoriti v nekaj, kar je bolj podobno nečemu, kar se da narediti v Flashu, s čimer si precej olajšamo delo:

1. Avto nastane izven vidnega območja ekrana – avto moramo najprej ustvariti.
2. Vozilo začne voziti in vozi, dokler se ne ustavi pred krožiščem. To se da doseči z animacijo.
3. Na tej točki avto preveri, ali se lahko vključi v krožišče. Tukaj bo moral nekako izvedeti, ali je v krožišču kak avto, ki mu onemogoča vključevanje. V kolikor ga ni, nadaljuje z animacijo vključevanja, v drugem primeru nadaljuje s ponovnim preverjanjem.
4. Animacija vključevanja.
5. Avto vozi skozi krožišče. V tej animaciji obstajajo štiri točke, kjer ima možnost ubrati izhod. Ko doseže izbrano točko, se sproži izbrana animacija vožnje iz krožišča. Če avto ne izbere izhoda, ponovi vožnjo v krožišču.

6. Izbrana animacija vožnje iz krožišča, kjer avto odpelje izven vidnega območja ekrana, kjer se tudi izbriše.

To je že bolje, saj se da videti, kako bi stvar lahko delovala v Flashu. Takšno realizacijo avtomobila se da doseči le z uporabo MC-a, saj je izmed grafičnih simbolov ta edini, ki nudi vse potrebne funkcije.

3.3.2 Videz avtomobila

Najprej sem ustvaril podolgovat lik v obliki pravokotnika, ki sem ga razrezal na kup navpičnih pravokotnikov (*slika 27a*). Na tak način sem avto razdelil na dele, ki sem jih nato lahko oblikoval brez nevšečnosti (*slika 27b*). Ustvaril sem karoserijo, ki mi je bila všečna, in nadaljeval z oblikovanjem stekel (*slika 27c*).

Slika 27: Razvoj videza avtomobila

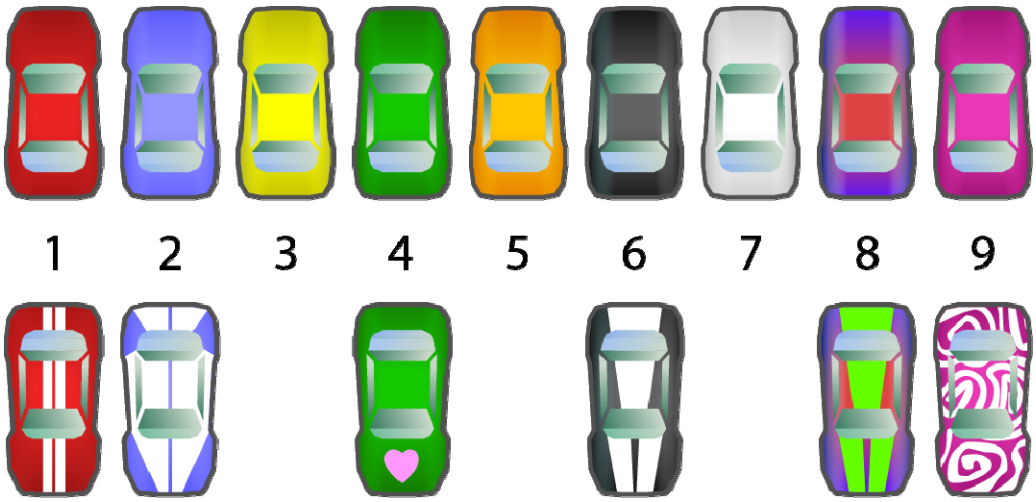
Vir: Lasten

Kot je razvidno iz slike, sem uporabil kar nekaj barvnih prelivov, za katere izris Flash porabi veliko procesorske moči (*slika 27č*). Vseeno se mi je videz avtomobila tako prikupil, da sem se odločil, da v tem primeru trenutno ne bom iskal kompromisov glede zmanjšanja zahtevnosti grafičnega prikaza, ampak bom raje počakal na končno fazo projekta, saj bo komaj takrat jasno, kolikšno optimizacijo ta del krožišča resnično potrebuje.

3.3.3 Različne preobleke

Za večjo razgibanost sem avtu ustvaril več videzov, natančneje devet. Zamislil sem si tudi dodatne vzorce, ki jih avto lahko uporabi pri določenih preoblekah. Ob vsakem nastanku si avto izbere eno izmed preoblek, in če ta podpira vzorce, si včasih izbere tudi enega izmed njih.

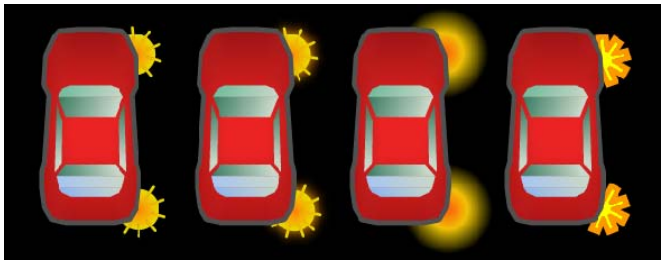
Način, s katerim sem to dosegel, je, da sem v glavni MC avtomobila namesto končne risbe dodal MC *CarSkin0*, ki je imel na vsakem *frame*-u drugo preobleko, izmed teh pa je za določene (1, 2, 4, 6, 8 in 9) vseboval še dodaten MC z vzorci. Ta *CarSkin0* je bil torej grafični videz avtomobila.



Slika 28: Vse preobleke in dodatni vzorci

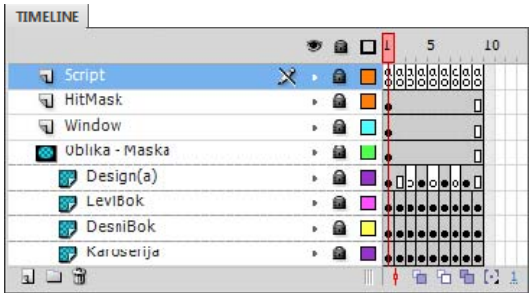
Vir: Lasten

Pri lepotičenju avtomobilov se nisem ustavil le pri preoblekah. Dodal sem še smerokaze, ki se vklopijo v trenutku, ko avto naleti na izbran izhod iz krožišča. Kar se tiče videza, so ti do zadnje različice doživeli vrsto sprememb. Glavni razlog za to je bilo dejstvo, da se enostavno niso dovolj dobro videli. Všeč mi je, da je končna, tj. najboljša inačica, hkrati tudi najbolje optimizirana. Čim sem napisal še kodo za delovanje omenjenega, je promet postal bolj pisan in bolj všečen za oko. Z videzom avtomobila sem zelo zadovoljen, saj ne glede na zgradbo iz čistih vektorjev daje bežen občutek tridimenzionalnosti.



Slika 29: Razvoj smerokaza skozi čas od leve proti desni

Vir: Lasten

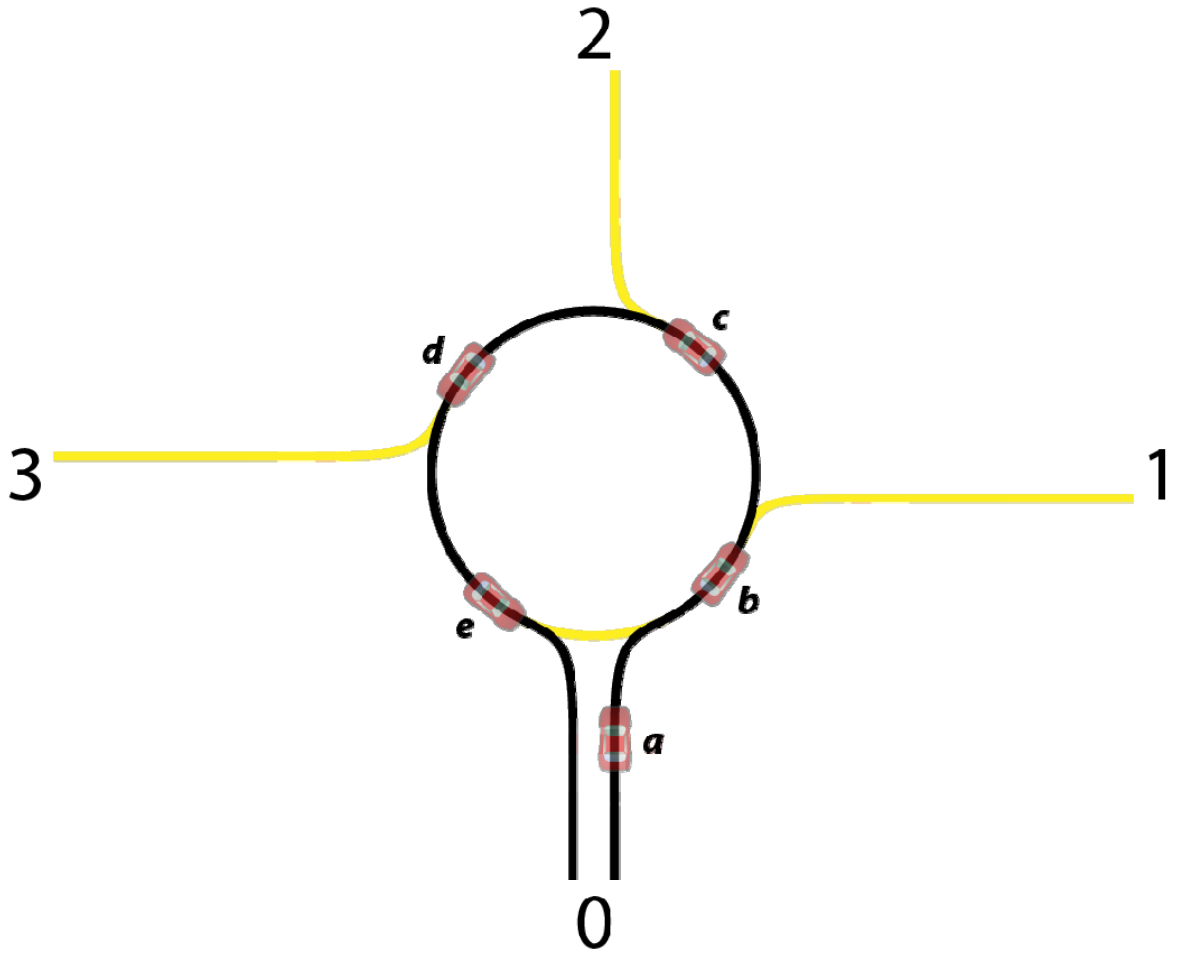


Slika 30: Drobovje MC-a *CarSkin0*

Vir: Lasten

3.3.4 Animacija

Želel sem, da bi bila vožnja avtomobilov natančna, in to brez odstopanj. Zaradi tega se mi je zdelo najbolje narediti vso animacijo avtomobila vnaprej. Avto bi se dejansko obnašal kot vlak. Peljal bi po »tirnicah« in na določenih točkah oziroma »postajah« bi imel izbiro, kaj narediti. Te postaje bi bile ključne točke, kjer se sprožijo nadaljnje animacije.



Slika 31: »Tirnice«, po katerih je animiran avto

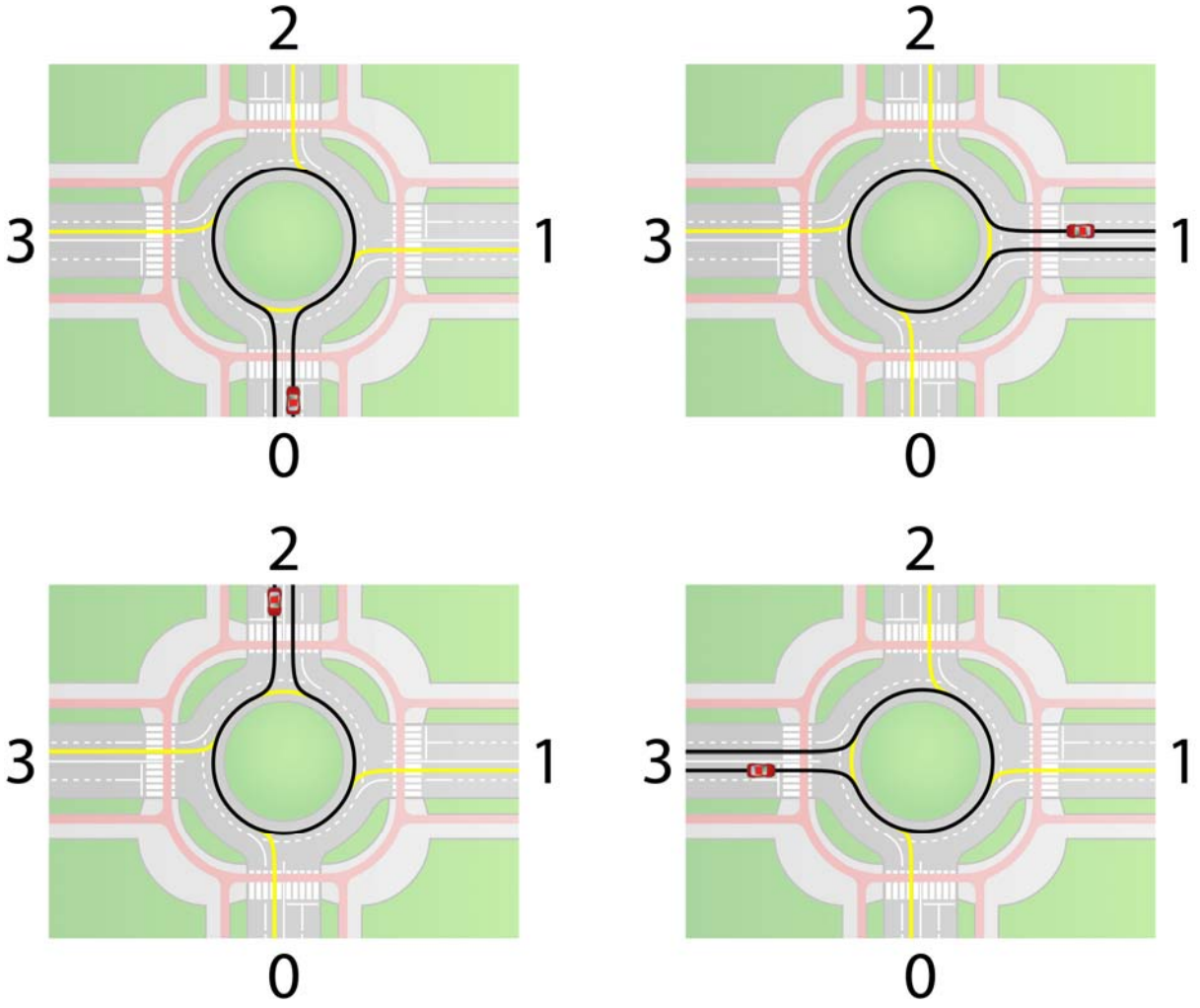
Vir: Lasten

Vožnjo avtomobila sem skušal animirati tako, da bi se odvijala čim bolj gladko, zato sem se odločil za uporabo načina animacije *Classic Tween*. S pomočjo krivulj sem narisal poti avtomobila in jih pretvoril v *Motion Guide*, nakar sem začel z animacijo.

Flash pri animaciji z *Motion Guide* ne podpira vejitve poti, zato sem moral narisati več različnih poti, ki so se dopolnjevale. Izziv je bil narediti animacijo, ki deluje gladko in kjer se ne občuti, da je v resnici razdeljena na dele, ki so med seboj povezani. Temu sem namenil precejšen del časa, vendar se je izplačalo.

3.3.5 Nastanek avtomobilov na različnih krakih

V tej točki razvoja krožišča sem imel univerzalen avtomobil in njegova celotna animacija vožnje je bila prilagojena začetku na kraku 0 (*slika 32 – levo zgoraj*). Če bi ta avto postavili na drugi začetni krak, bi še vedno začel voziti navzgor, kar ni dobro, saj bi namesto tega moral peljati proti krožišču. Moral bi biti ne samo na pravi poziciji, temveč glede na začetni krak tudi pravilno obrnjen.



Slika 32: Različne rotacije avtomobila glede na krak nastanka

Vir: Lasten

To sem rešil tako, da sem pravilni položaj in rotacijo MC-a avtomobila za vsako začetno pozicijo krožišča (0–3) shranil vnaprej.

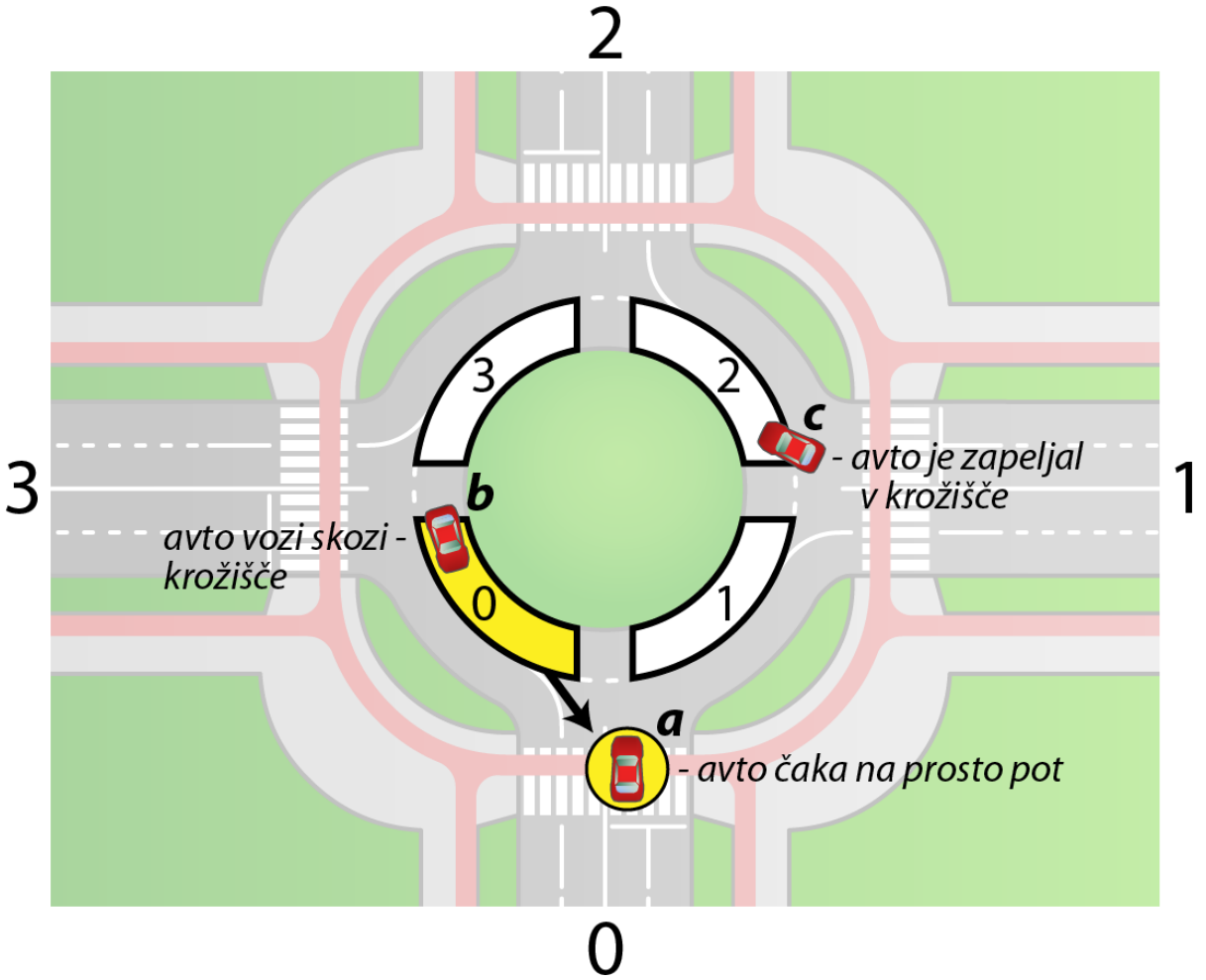
```
var carSpawnRot:Array = [0,-90,180,90];
var carSpawnX:Array = [506,506.5,516.2,514.5];
var carSpawnY:Array = [386,392,391.3,385.3];
```

Do teh vrednosti sem se dokopal z ročnim postavljanjem avtomobila na sceno. Rotacija je izražena v stopinjah, osi x in y pa v slikovnih pikah z izvorom v levem zgornjem kotu

projekta. Ti podatki so dovolj, da lahko avto pravilno obrnemo ter ga umestimo v krožišče. Poskrbeti sem moral le še, da je avto ob svojem nastanku te shranjene podatke prebral in jih uporabil kot lasten položaj in rotacijo. Na tak način sem dosegel pravilno vožnjo avtomobila, ne glede na njegov začetni položaj.

3.3.6 Preverjanje trčenja (Collision Detection)

Ker sem izbral naključni nastanek avtomobilov, se med njihovo vožnjo skozi krožišče zna zgoditi, da se srečajo, kar lahko povzroči, da en avto dobesedno pelje čez drugega. Tega si pri pravilni vožnji ne želimo, zato bi bilo dobro na točki, ko se avto ustavi pred krožiščem *(slika 33a*), preveriti, če mu kateri avto preprečuje vključevanje (*slika 33b*). Preveriti je torej bilo treba trk med dvema objektoma.



Slika 33: Mesta za preverjanje (0,1,2,3)

Vir: Lasten

S tem trkom ne mislim na trčenje dveh avtomobilov, temveč na trčenje med avtomobilom in delom krožišča, ki mora biti prazno, da se določen avto lahko vključi. Ob pogledu na krožišče se pokažejo štiri mesta, kjer bi bilo pametno preverjati prisotnost avtomobilov (*slika 33*).

Vsako mesto preverjanja je vezano na svoj krak, saj ni potrebe, da vsak avto, ki se vključuje, preveri vse štiri možnosti.

Za preverjanje trka dveh objektov obstaja v AS-u funkcija `hitTest()`, ki nam pove, ali sta dva MC-a trčila. Seveda potrebujemo njuno ime, saj ju v AS-u drugače ne moremo najti. V tem primeru bi en objekt bil mesto v krožišču, kjer se preverja, drugi pa avtomobil. Da najdemo, ali je mesto zasedeno, ga je treba preveriti za vse možne avtomobile v krožišču. Najlažji način se mi je zdel sledeč. Vsako izmed teh štirih mest bi bilo MC z unikatnim imenom (*piece0*, *piece1*, *piece2*, *piece3*) in avtomobil, ki čaka na prosto pot, bi preverjal, ali je vanj trčil kateri koli avto oziroma MC z imenom *car*. Vsi avtomobili, ki bi nastali, bi torej imeli enako ime, saj jih je tako laže preveriti vse.

Čeprav se je v teoriji slišalo delujoče, se je v realnem poskusu izkazalo, da ni tako. Avtomobili so peljali v krožišče ne gleda na to, ali je bilo mesto za preverjanje prazno ali polno. Napravil sem naslednji poskus. Ustvaril sem okolje, kjer avtomobili nenehno vozijo čez mesto krožišča, kjer se preverja njihova prisotnost. In najbolj zanimivo ter hkrati nadležno je bilo to, da je mesto zaznalo trk samo z avtomobilom, ki je nastal prvi, ne pa tudi z vsemi ostalimi, ki so nastali kasneje. Na tem primeru sem spoznal, da če v Flashu ustvarimo več objektov z enakim imenom (v tem primeru *car*), bo za AS viden samo tisti, ki je nastal prvi. Zato sem moral najti novo rešitev, kjer bi vsak avtomobil imel sebi lastno ime, ki mora biti unikat.

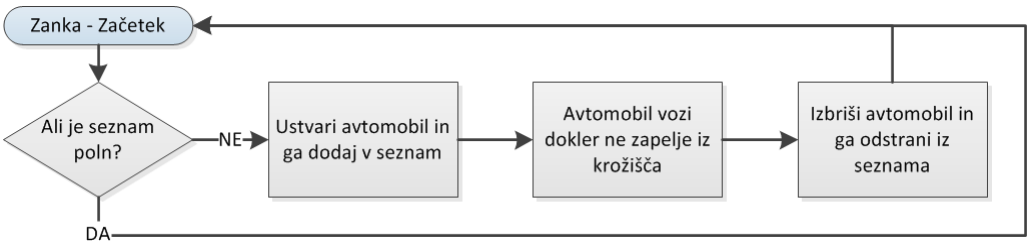
3.3.7 Seznam avtomobilov

Ko s pomočjo AS-a ustvarimo MC, se njegova globina zapiše v Flashev seznam prikaznih objektov. Gre za seznam globine, kjer večje število pomeni, da je objekt prikazan bližje v ospredju kot pa objekt z manjšim številom. V tem seznamu je nemogoče, da bi dva objekta imela isto število, zato sem se za ime avtomobilov odločil uporabiti prav to število globine. S tem, ko sem dobil edinstveno ime za posamezen avto, pa sem potreboval še dodaten seznam, kjer bi se ta imena hranila.

Seznam je dobra izbira prav zaradi dejstva, da lahko na učinkovit način potujemo skozenj in tako nenehno preverjamo, ali se je objektu, katerega ime najdemo v seznamu, spremenila kakšna lastnost. Ob nastanku posameznega avtomobila ta dobi sebi lastno ime iz števila globine, ki se zapiše v seznam. Medtem ko avto vozi, je na voljo za preverjanje. Ko se avto ob

koncu svoje poti odstrani, se iz seznama odstrani tudi ime. S tem, ko sem v logiko uvedel sezname, sem ustvaril red.

Beleženje trkov med mesti preverjanja in vsemi avtomobili v seznamu je z uporabo najdenih rešitev končno delovalo. Spoznal sem tudi, da ni potrebe po nenehnem preverjanju zasedenosti krožišča. Preverjati jo je treba le, kadar se posamezen avtomobil želi vključiti v krožišče. S tem zmanjšamo obremenitev računalnika z nepotrebnimi ukazi.



Slika 34: Zanka nastajanja in življenjske dobe avtomobila

Vir: Lasten

3.3.8 Kolone

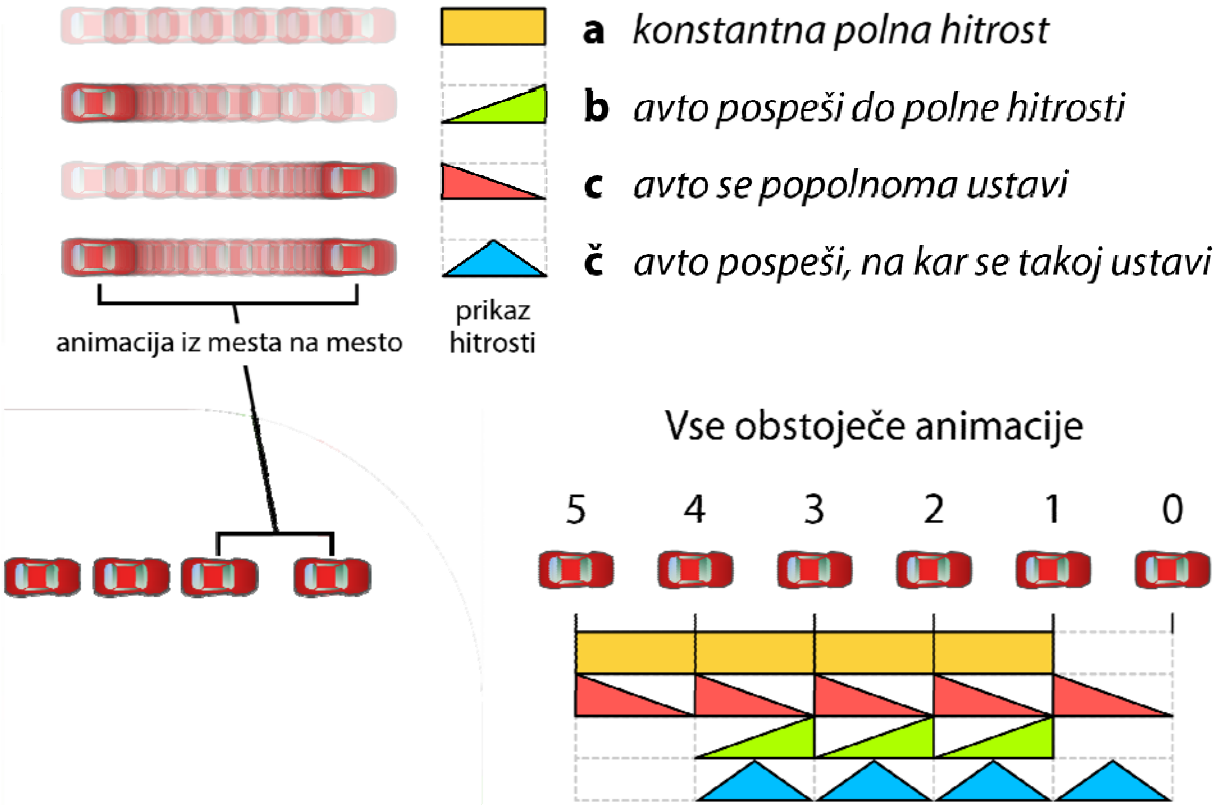
Kolone nastanejo, ko določen avto zasede pot drugim avtom, ki so za njim. Vzel sem si čas in razmislil o tem, kako se avtomobili obnašajo v kolonah. Logika bi šla nekako takole – vsak avtomobil preverja prostor pred seboj, in če naleti na avto ali pešca, se mora ustaviti. Tu so se pojavile težave. Vsak avto bi moral to preverjati nenehno, česar pa si nisem želel, saj se mi je to glede uporabe računalniške moči zazdelo potratno. Toda hujše je bilo spoznanje, da sem morda naredil več škode kot koristi, ko sem celotno vožnjo avtomobila animiral vnaprej, kar pomeni, da ga med vožnjo ni mogoče kar tako ustaviti, saj bi moral vsak njegov postanek predvideti in ga vnesti v animacijo.

Pri kolonah se postanki zgodijo kadar koli avtomobil naleti na oviro, kar zna biti kadar koli, še posebej s pešci, ki se odločijo prečkati cestišče, kot se jim zazdi. Kako torej animirati kaj takega? Če sem avtomobile ustvaril tako, da se obnašajo kot vlak, potem bi si jih moral tudi predstavljati kot vlake. Pa vendar, še noben vlak se ni ustavil pešcu, ki mu je v hipu prekrižal pot, saj enostavno niso tako načrtovani. Vlaki se praviloma ustavijo le na postajah, kar me je privedlo do spoznanja, da bi si mesta v koloni lahko predstavljal kot postaje. Kolono bi tako razdelil na več mest, in ker bi ta mesta bila statična, bi za vožnjo po njih lahko brez težav ustvaril animacijo. Seveda gre za kompromis, saj avtomobili v kolonah ne vozijo vedno le z mesta na mesto – v kolonah namreč ni vnaprej označenih mest. Velikokrat tudi drsijo, ne da bi se ustavili, kar v tem primeru ne bi bilo mogoče.

Začel sem razmišljati o načinu, s katerim bi animiral vse možnosti vožnje v koloni. Prišel sem do spoznanja, da bi vožnjo lahko razdelil na dele, ki bi jih nato sestavil kot neke vrste sestavljanko in tako dobil celotno vožnjo avtomobila. Na koncu vsakega »kosa« animacije bi avtomobil preveril, kakšno je stanje zasedenosti mest pred njim, in glede na rezultat bi se odločil, katero animacijo izbrati za nadaljnjo pot. V kolikor bi bila vsa mesta zasedena, bi počakal, dokler se mesto pred njim ne izprazni.

Vsak del animacije vožnje v koloni bi potekal z izbranega mesta na sosednjo mesto in bi bil eden izmed štirih različnih vrst, ki so predstavljene na sliki 35:

a) Avto pelje s konstantno polno hitrostjo. Za tem delom lahko nadaljuje z **a** ali s **c**.
b) Avto pospeši do polne hitrosti. Za tem delom lahko nadaljuje z **a** ali s **c**.
c) Avto se popolnoma ustavi. Za tem delom lahko nadaljuje z **b** ali s **č**.
č) Avto pospeši, nakar se takoj ustavi. Za tem delom lahko nadaljuje z **b** ali s **č**.



Slika 35: Sistem sestavljene animacije vožnje v koloni

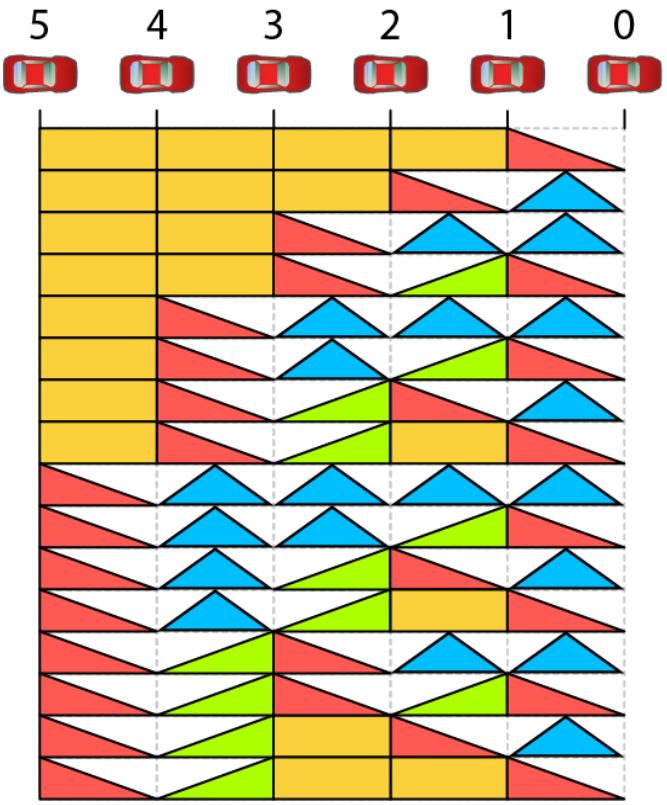
Vir: Lasten

Najprej nisem bil prepričan glede dolžine kolone, a sem spoznal, da je komaj 6. mesto (označeno s številom 5) stalo dovolj izven vidnega območja ekrana, kjer je varno, da nastane avto. Prikazal sem vse možne dele animacije, ki so potrebni za pravilno delovanje kolone. Za

razumevanje je treba vedeti, da avto na mestu, označenim s številom 5, začne s polno hitrostjo, na mestu 0 pa se popolnoma ustavi, saj tu doseže trenutek, ko preveri, ali se lahko vključi v promet. Z uporabo te metode sem moral ustvariti le štiri različne animacije, ki sem jih nato enostavno skopiral za različna mesta v koloni.

Slika 36 prikazuje vse možne kombinacije animacije vožnje, ki se jih da na ta način doseči za toliko mest kolone.

Najtežje pri tem je bilo programiranje logike, ki upošteva zgoraj opisane principe. Prav to mi je vzelo največ časa, saj se je pojavilo toliko stvari, na katere sem moral biti pozoren. Zaradi moje površnosti pri vodenju velike količine spremenljivk v kodi sem imel tukaj kup težav, katerih razlog ni bil takoj očiten, zato sem veliko časa porabil z iskanjem napak. Ne glede na natančnost, s katero delamo, se temu pri programiranju ne moremo izogniti. Ostalo je samo še, da sem funkciji za nastanek avtomobila dodal kodo, ki je v primeru, da je kolona polna, poskrbela, da avto ni smel nastati. S tem sem preprečil, da bi nastali avtomobili začeli voziti eden preko drugega, kar se je pred tem dogajalo.



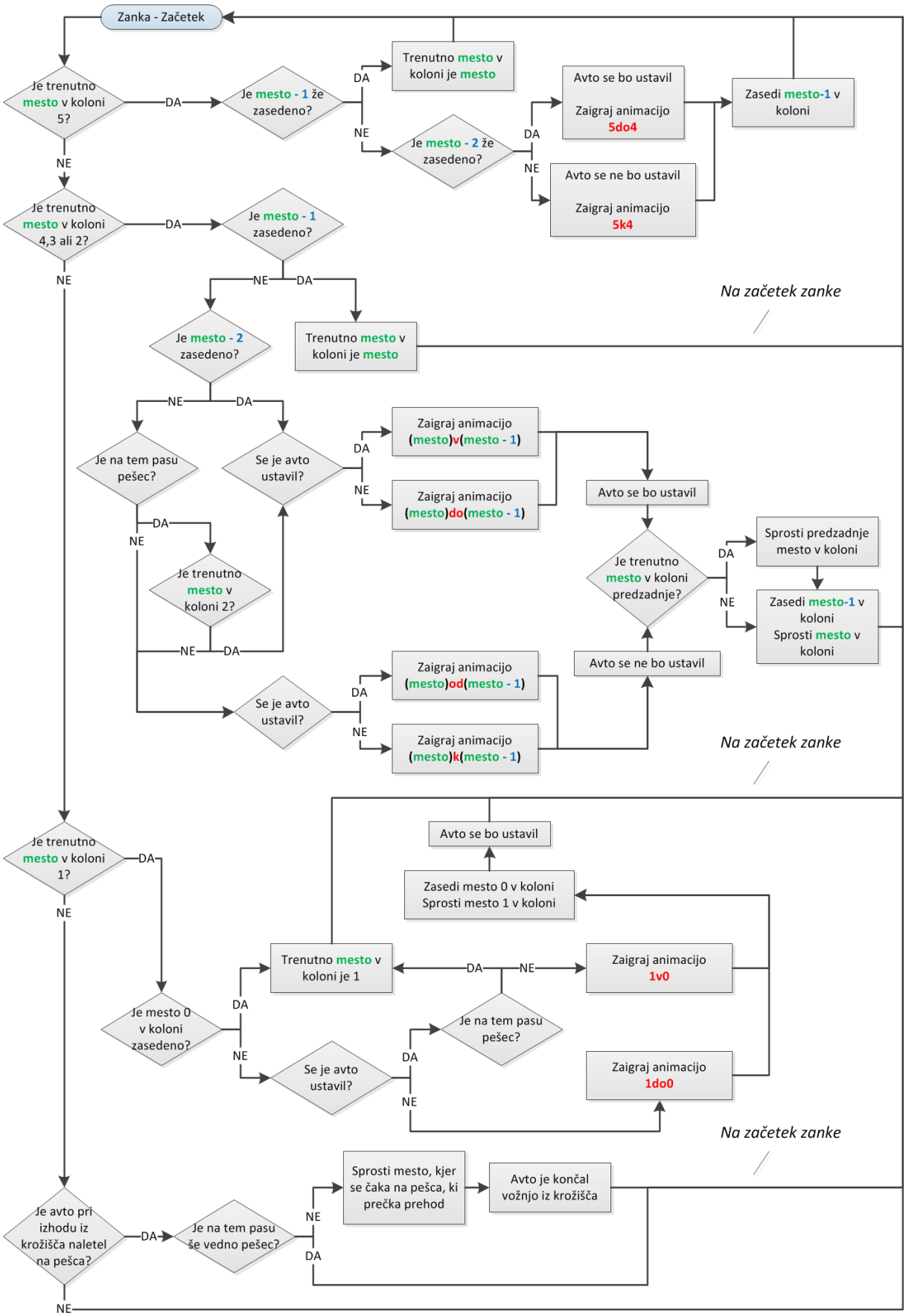
Slika 36: Vse kombinacije vožnje v koloni

Vir: Lasten

Ko je kolona končno delovala tako, kot sem si zamislil, sem bil navdušen, saj nekako nisem pričakoval, da bo delovala tako dobro. Velik razlog za to je dejstvo, da med delom vidiš le nedelujoče delčke, ki jih moraš povezati v delujočo celoto, tudi če še ne veš točno, kako.

3.3.9 Poenostavljen prikaz logike

Slika 37 kot zanimivost prikazuje poenostavljeno različico logike, ki sem jo zasnoval za pravilno delovanje kolon. Sproži se vsako osvežitev aplikacije, kar je v tem primeru 30-krat na sekundo. Poudariti moram, da gre tu za končno različico, zato je vanjo vnesena tudi interakcija s pešci, o katerih bom govoril kasneje.



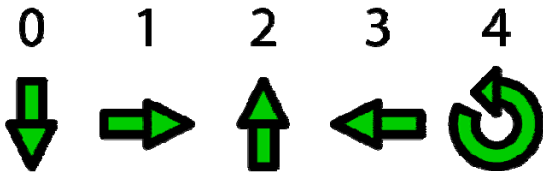
Slika 37: Poenostavljen graf poteka avtomobilove logike v koloni

Vir: Lasten

3.3.10 Interakcija

Pri načrtovanju elementov, ki sem jih želel vnesti v ta projekt, se mi je zdelo izredno zanimivo, da bi imel uporabnik možnost avtomobilom določiti, kam naj peljejo v krožišču. Premišljeval sem o načinu interakcije – kaj klikniti, kakšen bo odziv in podobno. Vedel sem, da bom za to moral poseči v AS. Domislil sem se naslednjega postopka, kako avtu določiti smer.

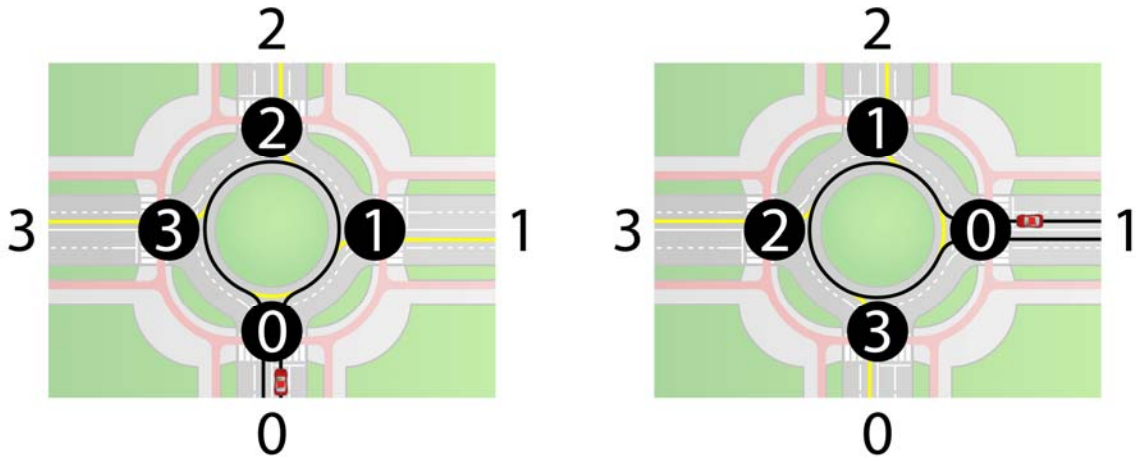
Najprej bi se z miško premaknil na avto, kateremu bi želel spremeniti pot. Pritisnil bi levi miškin gumb in ga držal, nakar bi s premikom miške določil, kateri izhod iz krožišča naj avto uporabi. Pojavila bi se puščica, ki bi nakazala izbrano smer. Ko bi miškin gumb spustil, bi avtu ostala nazadnje izbrana smer. Prepoznati bi bilo treba samo osnovne gibe – gor, dol, levo in desno.



Slika 38: Smeri, ki jih lahko določimo avtu

Vir: Lasten

Smeri, ki jih z gibi lahko določimo, so prikazane na sliki 38. Kot vidimo, se razen vožnje v krogu (4) vse smeri nanašajo na krake krožišča (0, 1, 2, 3)[2]. Premik miške navzgor na primer pomeni, da smo izbrali krak 2. Toda tu se je pojavila težava, ki jo prikazuje slika 39.



Slika 39: Različno dojemanje smeri glede na izvorni krak avtomobila

Vir: Lasten

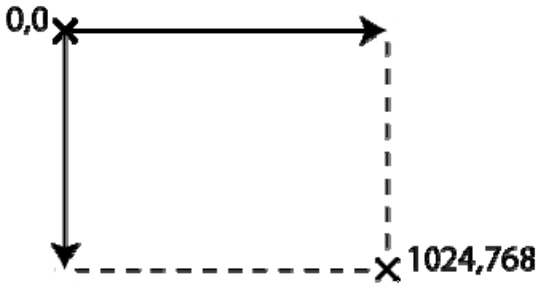
[2] Za boljše razumevanje krakov krožišča si oglejte na sliko 20.

Na vsaki skici imamo napisana števila krakov in smeri, kakor jih dojema avto. Prva so napisana ob robu vsake skice, medtem ko so slednja napisana v črnih krogih na sredini. Kadar avto nastane na kraku 0, deluje vse normalno, saj je njegovo dojemanje smeri enako krakom krožišča, kar pomeni, da če mu določimo smer desno (1), bo vzel izhod desno (krak 1). Poglejmo naslednjo sliko, kjer je avto nastal na kraku 1. Ker gre za isti avtomobil iz kraka 0, le da je obrnjen tako, da namesto navzgor pelje levo, se glede na krake spremeni njegovo dojemanje smeri. Če bomo temu avtu določili smer desno (1), bo vzel izhod navzgor (krak 2), saj je to njegova desna, kar si seveda nisem želel. Vidimo torej, da avtomobil dojema smer relativno na svoj izvor. Da sem prišel do rešitve, sem moral narediti, da se je smer, ki smo jo izbrali s premikom miške, avtu posredovala v obliki, ki je ustrezala njegovemu kraku izvora. S tem sem dosegel, da se je avto pravilno odzval na ukazano smer ne glede na krak nastanka.

Potreboval sem še en dodaten gib, ki bi označil peto možnost – vožnjo v krogu (*slika 38,4*). Ta bi se sprožil, kadar uporabnik za določen čas ne bi naredil nobenega izmed prej naštetih gibov. Gib sem usposobil tako, da sem ob izbiri avtomobila začel odštevati čas. Ko ga je preteklo zadostno število, se je aktivirala smer. V kolikor pa bi uporabnik med tem odštevanjem naredil nek drug gib, se bi čas, ki ga je bilo treba odšteti do aktivacije, povečal.

3.3.11 Zajem gibov

Kar se miške tiče, lahko v AS-u dobimo samo njeno pozicijo na x in y osi, zato sem za prepoznavo gibov moral napisati svoj algoritem. Toda preden se v to poglobimo, je dobro vedeti, da se slikovne pike v Flashu začnejo meriti od levega zgornjega roba proti desnemu spodnjemu. To pomeni, da so vrednosti na osi x pozitivne desno od izhodišča, za y os pa navzdol. Primer na sliki 40 prikazuje dimenzije krožišča.

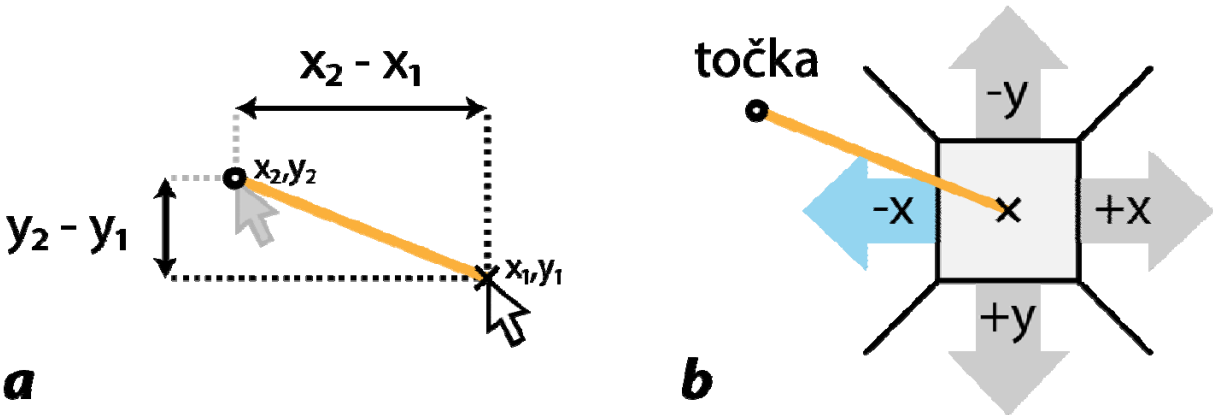


Slika 40: Kako v Flashu merimo slikovne pike

Vir: Lasten

Na sliki 41a se vidi, da sem vzel trenutno pozicijo miške (beli kazalec) in od nje odštel prejšnjo pozicijo miške (sivi kazalec). Na tak način sem dobil pot (oranžna črta), ki jo je miš

opravila v enem *frame*-u. Čeprav sem pravkar omenil pot, tu v resnici ne gre za vektor, temveč za razliko med točkama na x in y osi – delta x in delta y. Slika 41b prikazuje, kako sem »pot«, ki sem jo dobil, pretvoril v želeno smer. Da uporabnik giba ne bi naredil pomotoma, sem vnesel toleranco, ki je na sliki prikazana kot bel kvadrat, od koder se meri pot. Kadar je bila delta x ali y večja od te tolerance, je bil premik dovolj velik, da sem ga lahko obravnaval kot smer. Nato sem izmeril, katera izmed vrednosti je daljša, delta x ali delta y. S tem sem dobil os, na kateri se je zgodil večji premik. Preostalo mi je le, da sem preveril, ali je vrednost pozitivna ali negativna. Tako sem dobil najverjetnejšo smer. Za pravilno odzivnost se je morala koda za ta algoritem izvršiti za vsak *frame* delovanja aplikacije.



Slika 41: Kako premik miške pretvorimo v smer

Vir: Lasten

Spoznal sem, da je zelo pomembno, da ti gibi delujejo gladko in brez težav, saj je bilo prvih nekaj različic prav nerodnih. Na primer težava, kjer je bilo prelahko nehote spremeniti smer, je botrovala zamisli o že prej omenjeni toleranci.

3.3.12 Dodaten avtomobil

Poudariti moram, da sem ves čas govoril o MC-u avtomobila *car0*, ki vozi na levem pasu cestišča, nisem pa omenil obstoja še enega MC-a avtomobila, ki vozi po desnem – *car0a*. Ta se od *car0* razlikuje v tem, da je njegova pot vožnje vedno enaka, zato ne poseduje zgoraj omenjene zmožnosti spreminjanja smeri z miško. V drugih pogledih sta MC-a identična.

3.4 Pešec

Ena težjih nalog pri delu na tem projektu je bila zasnova pešcev. Pri ustvarjanju peščevega videza se nisem želel prenagliti, zato sem jih naredil karseda enostavne. Razlog za to je bil podroben videz avtomobilov, ki so že sami bili procesorsko zelo požrešni.

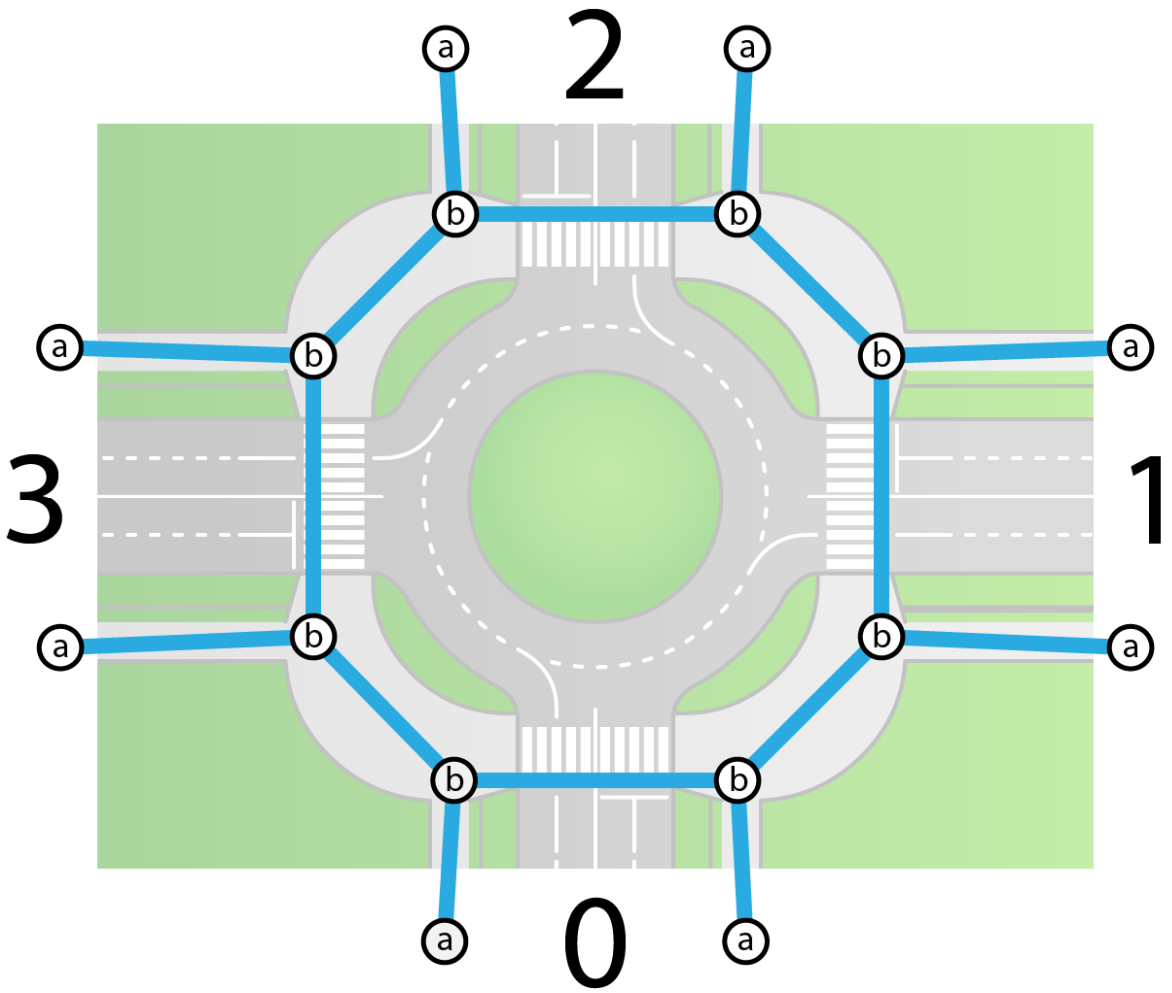
Preprostost videza me je najprej motila, vendar se mi je skozi tok projekta priljubila. Tako je prikazana različica ostala nespremenjena do konca projekta.

Slika 42: Videz pešca

Vir: Lasten

3.4.1 Logika peščevih poti

Prva stvar, ki sem jo naredil, je bila, da sem si narisal skico, na kateri sem označil vse poti, ki bi jih naj pešci lahko naredili. Ta skica je bila temeljnega pomena, saj sem se pri iskanju optimalnih rešitev orientiral prav po njej.



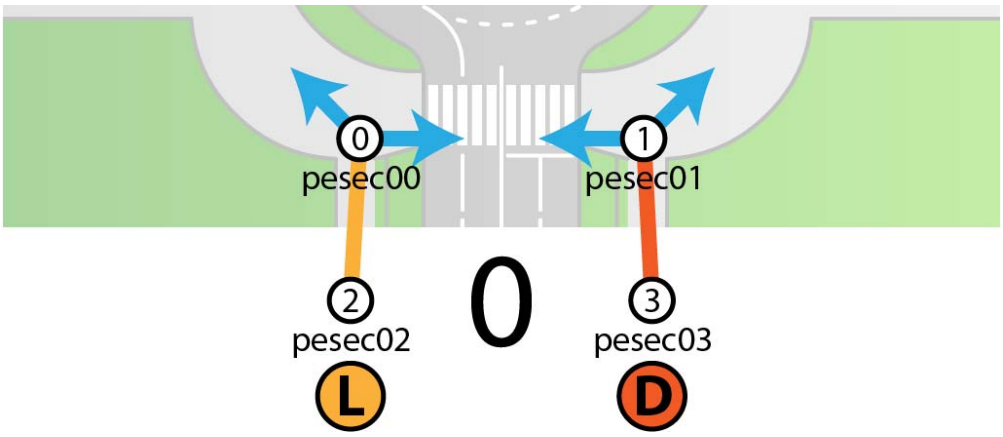
Slika 43: Mreža poti

Vir: Lasten

Tako kot univerzalen avtomobil bi pešci nastali, se premikali, nakar bi se izbrisali. Toda ob pogledu na številčnost poti (*slika 43*) sem se odločil, da za razliko od avtov pešci ne bi bili animirani po tirnicah, temveč bi se orientirali na podlagi točk, ki sem jih poimenoval cilji. Ob nastanku bi pešec namreč dobil nalogo slediti enemu izmed teh ciljev. Ko bi se ga dotaknil oziroma vanj trčil, bi dobil nalogo slediti drugemu cilju in tako naprej.

Vzel sem si čas in razmislil o načinu poenostavitve prikaza skice. Prišel sem do spoznanja, da obstajata dve vrsti točk oziroma ciljev. Na sliki 43 sem ju označil z *a* in *b*. Pri točki *a* gre za mesto, kjer bi pešec nastal ter končal svojo pot. Pri točki *b* pa gre za križišče s tremi kraki. Na tem cilju bi pešec imel možnost izbrati eno izmed preostalih dveh poti, saj je iz ene strani ravnokar prišel. Vsak cilj bi lahko prehodil le enkrat, zatem tja več ne bi smel iti. S tem bi ga prisilil, da bi mu slej ko prej zmanjkalo postaj in bi bil primoran oditi iz krožišča.

Torej, če predpostavimo, da pešci nastanejo na kateri koli točki *a*, potem je jasno, da se morajo začeti premikati proti najustreznejši točki *b*. Ampak kako vedeti, katera točka *b* je najustreznejša?



Slika 44: Logika peščevih ciljev za krak 0

Vir: Lasten

Z namenom, da sem prišel do pravega odgovora, sem moral logiko za vsak krak ločiti na levo in desno stran cestišča, saj lahko pot iz točke *a* vodi samo do tiste točke *b*, ki se nahaja na isti strani (*slika 44*). Enako velja v obratni smeri. Nato sem vsaki točki dodelil sebi lastno število in jih razporedil tako, da se je dalo kar iz njih razločiti, za katero točko gre. Na primer, če je število sodo, gre za levo stran cestišča, če pa je liho, pa za desno. Prav tako se da razbrati, ali gre za točko *b* (število je manjše od 2) ali za točko *a* (število je večje od 2). Očitno je tudi, da

sem logiko, tako kot pri avtomobilu, razdelil na krake. S tem sem si pripravil jasno osnovo za pisanje kode, ki bo skrbela za vodenje pešcev ter njihove odločitve.

Medtem ko bi pešci nastajali preko AS-a, sem cilje, po katerih se bi orientirali, ustvaril ročno kot MC-je. Vsak je dobil svoje ime, ki je vsebovalo besedo *pesec*, število kraka ter število cilja na tem kraku. Ta imena so na sliki zapisana pod točkami. To je bilo pomembno, saj se peščeva logika izbiranja novih ciljev nanaša prav na ta imena.

3.4.2 Podrobneje o peščevi logiki

Najprej sem spisal logiko za nastajanje pešcev, ki jih je ob nastanku postavila na točko 2 ali 3 na poljubnem kraku. Pešec si je zapisal, da je tukaj že bil, nakar je dobil svoj prvi cilj, ki je bil izračunan po formuli a. Pri tem cilju gre za točko *b* na isti strani cestišča. Od tod naprej bi pešec postal samostojen.

	Formula	Primer za L	Primer za D
a	trenutni cilj **– 2**	**2 – 2 = 0**	**3 – 2 = 1**
b	trenutni cilj **+ 2**	**0 + 2 = 2**	**1 + 2 = 3**
c	| trenutni cilj **– 1** |	| **0 – 1** | **= 1**	| **1 – 1** | **= 0**
č	trenutni krak **– 1**		
d	trenutni krak **+ 1**		

Tabela 1: Formule za izračun peščevih ciljev

Vir: Lasten

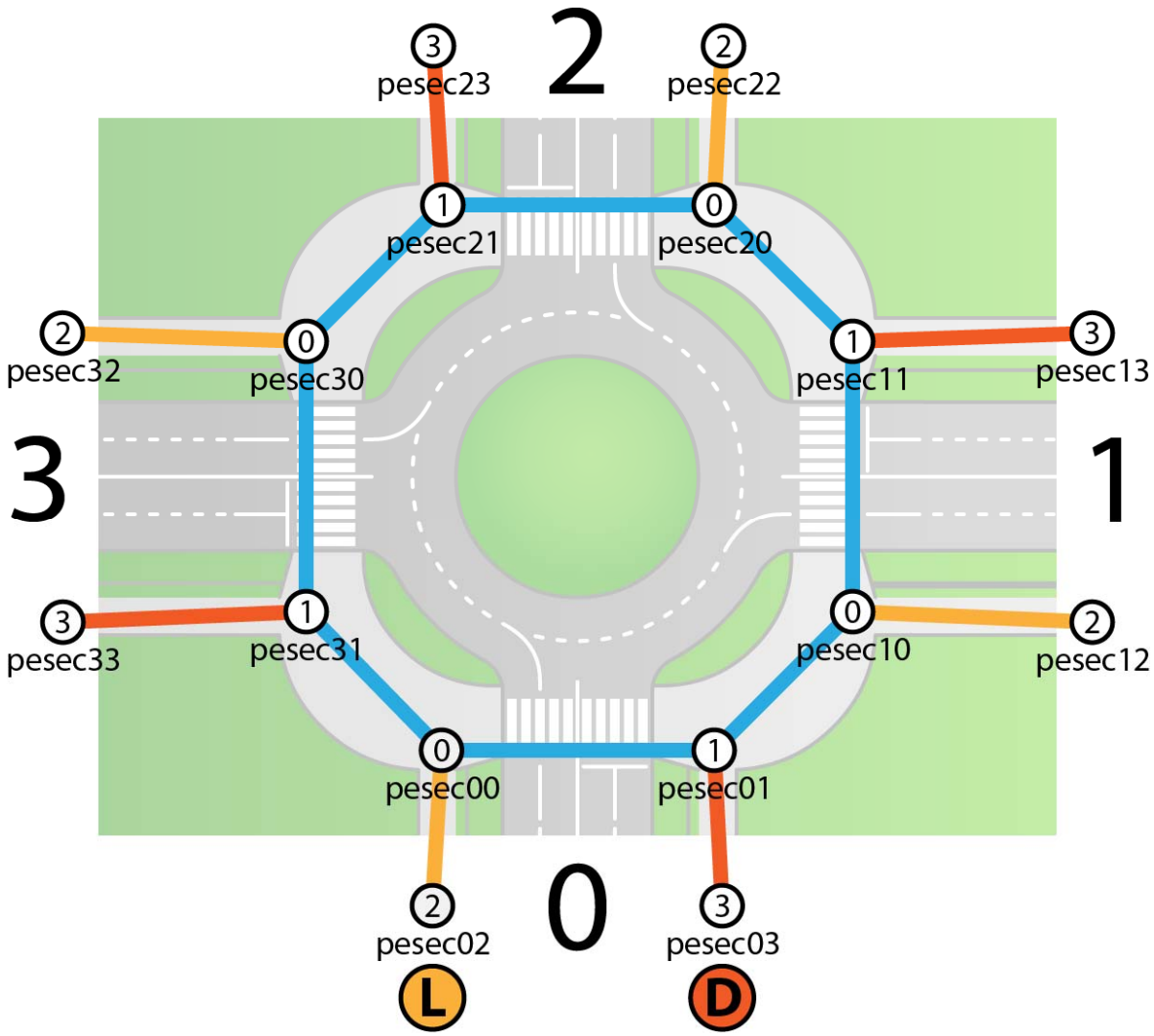
Pešec bi se premikal do svojega cilja in ko bi ga zadel, bi se glede na to, za katero vrsto točke gre (*a* ali *b*), odločil, kako reagirati. Ob srečanju s točko *b* bi pešec dobil nov cilj, kamor bi se začel premikati. V primeru srečanja s točko *a* pa bi se izbrisal, saj točka *a* pomeni konec njegove poti. Ne glede na vrsto točke bi si za vsak cilj, ki bi ga obiskal, zapisal, da je tam že bil. Oborožen s tem znanjem sem se odpravil pisati kodo, ki bi upoštevala vse omenjene zapovedi.

Poglejmo natančneje, kaj se zgodi, ko pešec naleti na točko *b*. Ko jo zadene, naključno izbere eno izmed naslednjih možnosti:

1. Nov cilj je točka *a* v tem kraku. Dobimo jo po formuli b (*tabela 1b*).
2. Nov cilj je točka *b* v tem kraku. Dobimo jo po formuli c (*tabela 1c*).
3. Nov cilj je točka *b* v sosednjem kraku. Dobimo jo po formuli c (*tabela 1c*), toda krak cilja se spremeni glede na naslednji princip. Če smo na levi strani cestišča in nismo na

kraku 0, nov krak cilja dobimo po formuli č (*tabela 1č*). V primeru, da smo na kraku 0, je nov krak cilja krak 3. Če pa smo na desni strani cestišča in nismo na kraku 3, nov krak cilja dobimo po formuli d (*tabela 1d*). V primeru, da smo na kraku 3, je nov krak cilja krak 0.

V kolikor je pešec na izbranem cilju že bil, bi ponovno izbral eno od naštetih možnosti. Za boljše razumevanje postopka izbire ciljev si je najbolje pomagati s sliko 45.

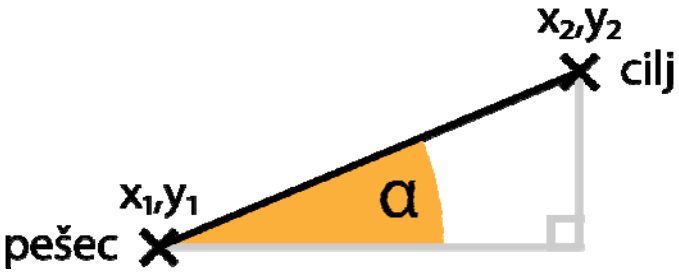


Slika 45: Celoten prikaz logike peščevih ciljev

Vir: Lasten

3.4.3 Hoja proti cilju

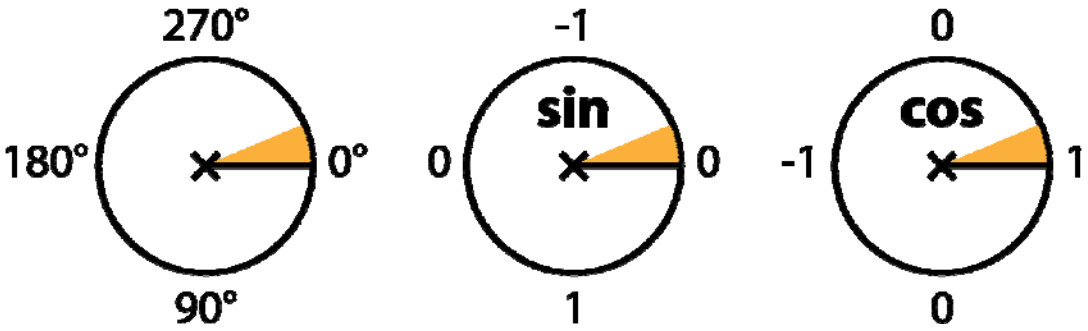
Zanimiv je tudi način, s katerim sem dosegel, da so pešci znali hoditi proti izbranemu cilju. Kot vemo, je pozicija vseh MC-jev v Flashu izražena v dveh dimenzijah, x in y. Če začnemo katero izmed teh vrednosti spreminjati vsako sliko na sekundo, dobimo gibanje. Gibanje je seveda odvisno od tega, na kak način to vrednost spreminjamo. Če na x-osi vrednost večamo, se začne objekt premikati v desno, če jo manjšamo, v levo. Manjšanje vrednosti na y-osi ga premakne navzgor, večanje navzdol. To je vse odlično, toda jaz sem potreboval gibanje proti neki izbrani točki.



Slika 46: Kot med pešcem in ciljem

Vir: Lasten

Za lažje razumevanje sem moral najprej vedeti, kakšno pot bi pešec sploh moral opraviti. Gledal sem skico (*slika 46*) in razmišljal, kako izračunati premik pešca za vsako os posebej. Skiciral sem kup skic, dokler nisem ugotovil, da bi lahko uporabil funkciji sinus in kosinus (*slika 47*).



Slika 47: Kot iz slike 41 prenesen v krog

Vir: Lasten

V tabeli 2 sem prikazal nekaj primerov, ki dokazujejo, da je sinus idealna funkcija za izračun premika po osi y, medtem ko za kosinus velja enako kot za os x.

Smer	Kot	Sinus	Kosinus
Desno	0°	0	1
Dol	90°	1	0
Levo	180°	0	-1
Gor	270°	-1	0
Desno-dol	45°	0,71	0,71
Levo-dol	135°	0,71	-0,71
Levo-gor	225°	-0,71	-0,71
Desno-gor	315°	-0,71	0,71
Smer iz primera	22,8°	0,39	0,92

Tabela 2: Prikaz vrednosti funkcij sinus in kosinus glede na različne smeri v krogu

Vir: Lasten

Da bi lahko napisal formulo, sem potreboval le še način, kako pridobiti kot med pešcem in ciljem. Na spletu sem našel informacijo o AS-funkciji `atan2()`, ki je že vgrajena v Flash.

```
Radians = Math.atan2(destName._y - this._y, destName._x - this._x);
```

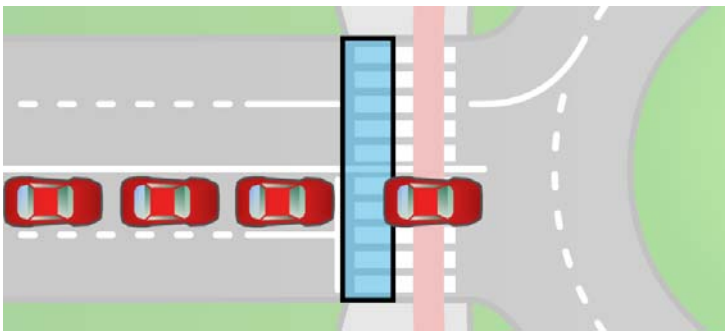
Ta funkcija vzame razliko med točkama na osi x ter osi y in vrne kot med njima v radianih. S pomočjo tega kota sem končno lahko sestavil funkciji, ki sta izračunali takšno velikost premika za vsako os, da se je pešec pravilno premaknil v smeri točke. Za prilagoditev velikosti tega premika sem dobljeno vrednost pomnožil s številko, ki je predstavljala hitrost pešca. Tako se je pešec premaknil proti cilju za pot, ki je bila enaka temu številu hitrosti.

```
this._x = _x + Math.cos(Radians) * speed;
this._y = _y + Math.sin(Radians) * speed;
```

Za nenehno gibanje pešca se je ta koda morala zagnati vsako sliko na sekundo.

3.4.4 Prehod za pešce

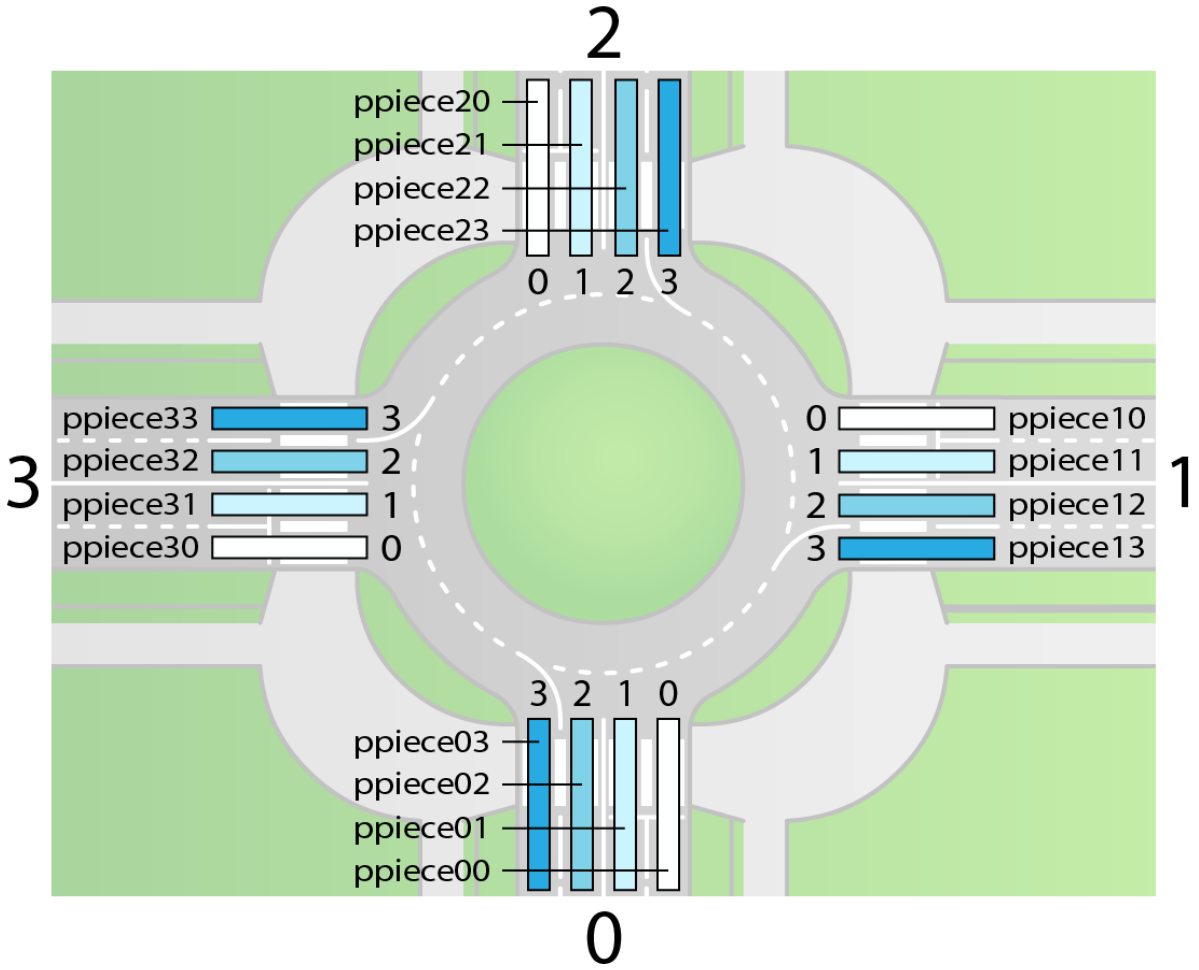
Hojo pešcev skozi krožišče sem do sem imel že vzpostavljeno in delujočo. Želel sem samo še, da avtomobili ne bi vozili čez njih. Tako je, vozilom je bilo vseeno za pešce. Prav tako teh ob prečkanju cestišča ni motilo, da je čez njih pravkar peljala vrsta vozil. Temu je bilo treba narediti konec. Moj namen je bil prikazati pravilno vožnjo v krožišču, ne pa kup nedeljskih voznikov, ki niso sposobni spremljati pešcev, in če bi bil voznik vzoren, se pešcu ob prečkanju cestišča sploh ne bi bilo treba ustaviti. Svojo pot čez prehod bi lahko nadaljeval brez kakršnega koli postanka. Na srečo sem kolone, v katere se formirajo avtomobili, načrtoval tako, da je za pešce ostalo nekaj prostora. Avtomobilom je bilo treba sporočiti le še, kdaj naj se ustavijo.



Slika 48: Prostor, kjer pešci prečkajo cestišče

Vir: Lasten

Ko je pešec stopil na prehod, je bilo za ustavitev avtomobila že prepozno. Prav tako si nisem mogel pomagati s spremljanjem peščeve odločitve o tem, ali bo prečkal cestišče ali ne, saj je tudi ta prišla prepozno. Zakaj prepozno? Ker se avto ni imel časa tako hitro ustaviti in je vseeno peljal čez pešca. Nekako bi moral že vnaprej vedeti, ali bo pešec izbral pot čez prehod, in to sporočiti avtomobilu.

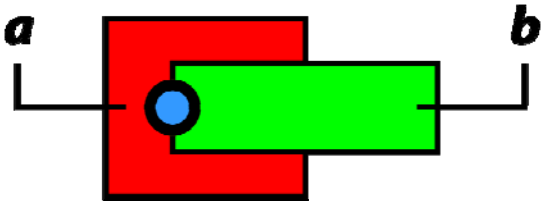


Slika 49: Postavitev in poimenovanje prehodov za pešce

Vir: Lasten

Težavo sem rešil tako, da sem najprej na vsakem pasu krožišča ustvaril mesto za preverjanje prisotnosti pešca (*slika 49*). Ko se je pešec katerega od njih dotaknil ali se ga nehal dotikati, se je to zapisalo in avto je lahko reagiral pravilno, čeprav še vedno prepozno.

Rešitvi sem bil bliže v trenutku, ko sem si izmislil, da bi pešec imel tipalo (*slika 50b*), s katerim bi lahko pot mestom za preverjanje prisotnosti sporočil nekoliko vnaprej. Ko sem zadevo preizkusil, se je pokazalo, da je avtomobil za reakcijo tokrat imel dovolj časa, s čimer je lahko bolje predvidel peščev namen.



Slika 50: Peščevi tipali

Vir: Lasten

Zaradi boljše učinkovitosti prometa, sem pešcu ustvaril še dodatno tipalo (*slika 50a*). Kot pri tipalu b se je tudi to aktiviralo komaj takrat, kadar je zadelo katero od mest za preverjanje prisotnosti pešca, in se onemogočilo takoj, ko se je katerega izmed njih nehalo dotikati. Njegova naloga pa ni imela zveze z avtomobili, temveč s hitrostjo peščeve hoje. Pokazalo se je namreč, da so bili pešci ob prečkanju cestišča prepočasni, kar je povzročilo, da je bila vožnja avtomobilov v krožišču manj odzivna, kot sem želel. S pospešitvijo hitrosti pešca ob prečkanju cestišča sem zmanjšal čas čakanja avtomobilov, s čimer je promet postal bolj živ.

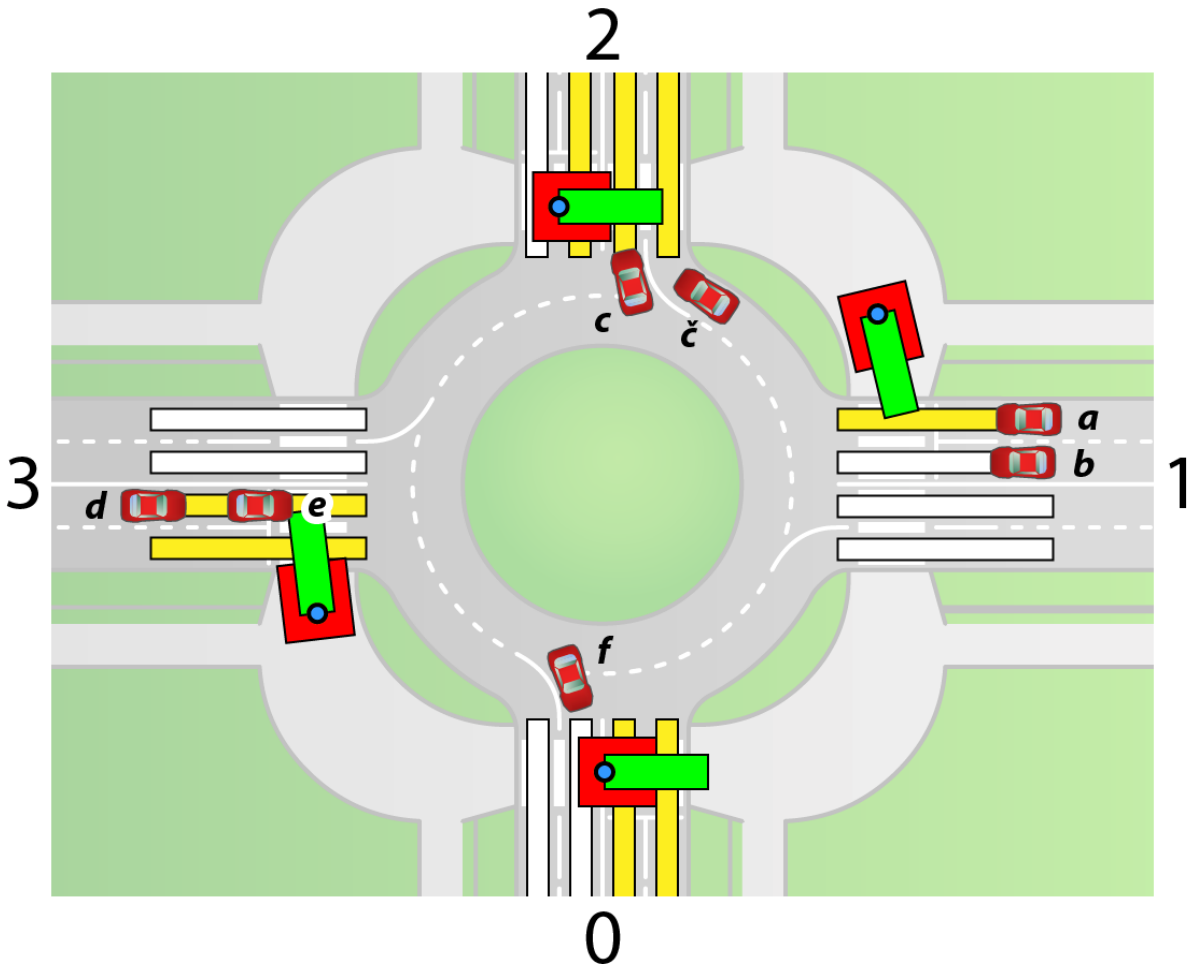
Preostalo mi je samo še, da sem celotno logiko nekoliko izpopolnil in v kratkem so avtomobili zanesljivo pazili na pešce. Za lažje razumevanje sem priložil sliko 51, ki prikazuje vrsto primerov te interakcije med pešci in avtomobili:

Krak 1 Pešec je s tipalom zadel v pas cestišča, ki je ponazorjen z rumeno barvo. S tem je povzročil, da se bo avtomobil **a** moral ustaviti. Avtomobilu **b** se ne bo treba ustaviti, saj pešec ni bil dovolj hiter, da bi mu preprečil pot.

Krak 2 Avtomobila **c** in **č** se bosta morala ustaviti pešcu, saj je s svojim tipalom še pravi čas naznanil svoj prihod.

Krak 3 Vozilo **d** se bo moralo ustaviti pešcu, medtem ko je vozilo **e** že prečkalo točko, po kateri več ni možnosti ustavitve. Avto se bo tako pešcu umaknil še pravi čas.

Krak 0 Kot vidimo, peščevo drugo tipalo ne vpliva na to, ali se bo avto moral ustaviti ali ne, zato bo avto **f** lahko vožnjo nadaljeval brez težav.



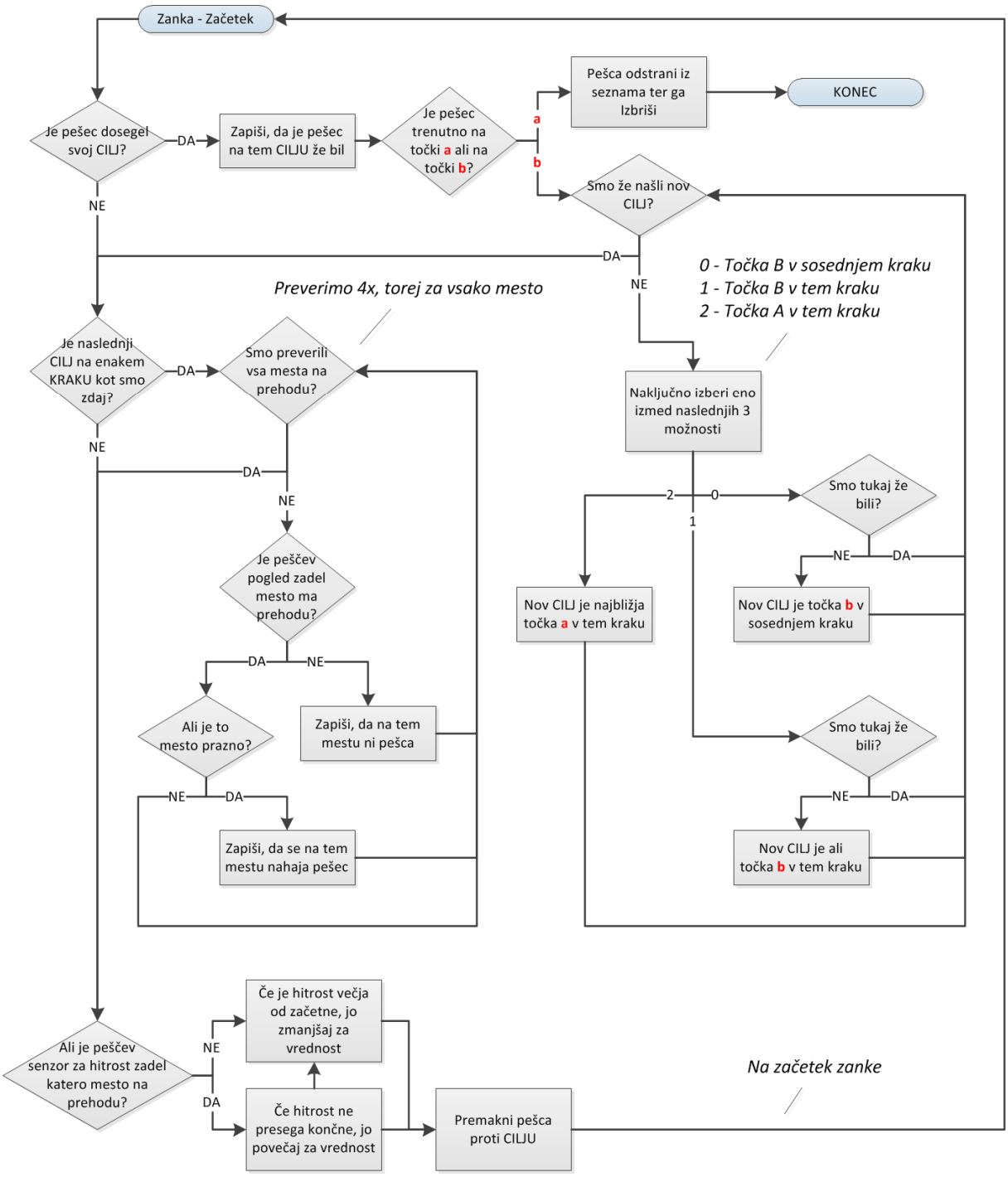
Slika 51: Kako pešec prečka cestišče

Vir: Lasten

Ker sem na pešce in dejstvo, da bodo s prečkanjem cestišča motili avtomobile, mislil že od začetka, pri ustvarjanju prehodov nisem imel znatnih težav.

3.4.5 Poenostavljen prikaz logike

Kot zanimivost prilagam poenostavljen prikaz peščeve logike (*slika 52*), ki vsebuje tudi logiko prečkanja cestišča.



Slika 52: Poenostavljen graf poteka peščeve logike

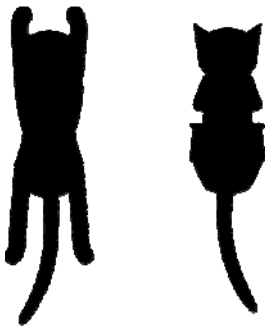
Vir: Lasten

3.5 Mačka

Vozniki poznamo občutek, ko nam v soju nočnih luči na cesto skoči mačka. Zaradi njene neslavne povezave z avtomobili, sem se odločil, da jo vključim v krožišče.

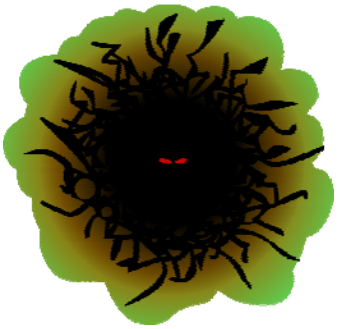
Mačka bi se brezbrižno sprehajala po krožišču, dokler ne bi naletela na nevarnost v obliki avtomobila, ki bi jo lahko povozil. Mačka bi zato morala biti dovolj pametna, da bi se avtu izognila. Najbolj zanimivo mesto za nastanek mačke je bila sredina krožišča, saj mačka tam nima druge izbire, kot da prečka cestišče, kjer vozijo avtomobili. Za to mesto sem imel najprej namen uporabiti grm, a sem se raje odločil za brlog, ki je nenavadnejši (*slika 54*). Šlo je za MC, ki je nenehno preverjal, ali smo ga z miško kliknili.

Sama mačka je bila prav tako MC, saj je morala tako kot pešec sama zaganjati svojo logiko. Z njenim videzom se nisem pretirano obremenjeval, saj majhnost njene postave ne bi omogočila prikaza podrobnosti. Njeno telo sem razdelil na kose, saj sem ga tako laže animiral. Črno barvo sem uporabil zato, ker se je med zmedo v krožišču najbolje videla.



Slika 53: Videz mačke v teku

Vir: Lasten



Slika 54: Mačkin brlog

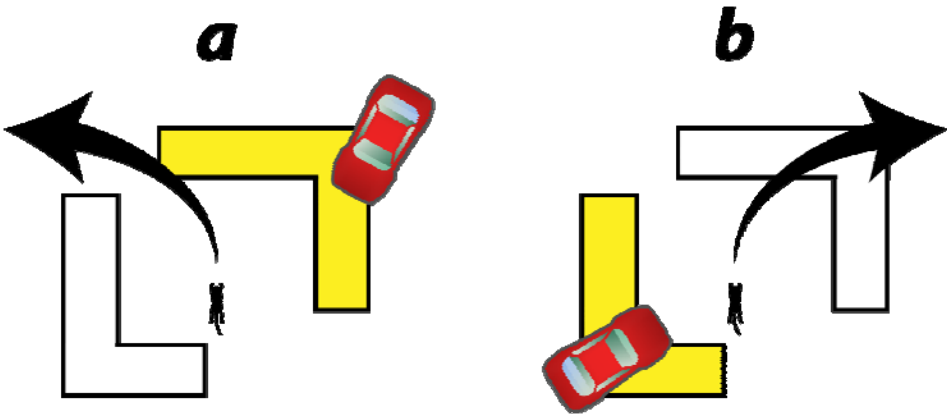
Vir: Lasten

Brlog je ob kliku ustvaril mačko, ki se je začela sprehajati. Pri tem sem uporabil isti način premikanja kot pri pešcu. Toda za razliko od pešca se mačka ni premikala proti določeni točki, temveč tja, kamor je bila obrnjena. Njena smer je bila torej odvisna od njene rotacije, katere smer se je izbrala naključno ob mačkinem nastanku. Tu sem vnesel tudi, da je mačka ob pobegu iz vidnega območja izginila.

Ker je bilo nenavadno, da je mačka tekla v matematično popolnoma ravni liniji, sem v njeno logiko vnesel naključno spreminjanje smeri. Ta vedenjski vzorec mačke sem poimenoval »neinteligentni«, saj se med njenim tekanjem po krožišču ni zmenila za noben drug objekt.

3.5.1 Izmikanje

Ko sem mački vnesel posledice ob morebitnem stiku z avtomobili, sem vedno bolj začel razmišljati o tem, kako bi se slednjim izognila. Načrtovati sem začel nov vedenjski vzorec, kjer bi mačka morala biti pametna, vendar ne prepametna, saj bi jo vozila tu in tam morala povoziti, vendar redko. Odločil sem se za inteligenco, ki je osnovana na uporabi dveh tipal. Z njuno pomočjo bi mačka prepoznala, iz katere smeri prihaja nevarnost, in se pravilno umaknila. Če bi nevarnost zaznalo tipalo a, bi mačka zavila v levo, v primeru, da pa bi se aktiviralo tipalo b, bi zavila desno. Dolžino in obliko tipal sem tekom testiranja uspešnosti izmikanja spreminjal in tako prišel do končnih različic (*slika 55*).



Slika 55: Tipali mački sporočita, od kod prihaja nevarnost

Vir: Lasten

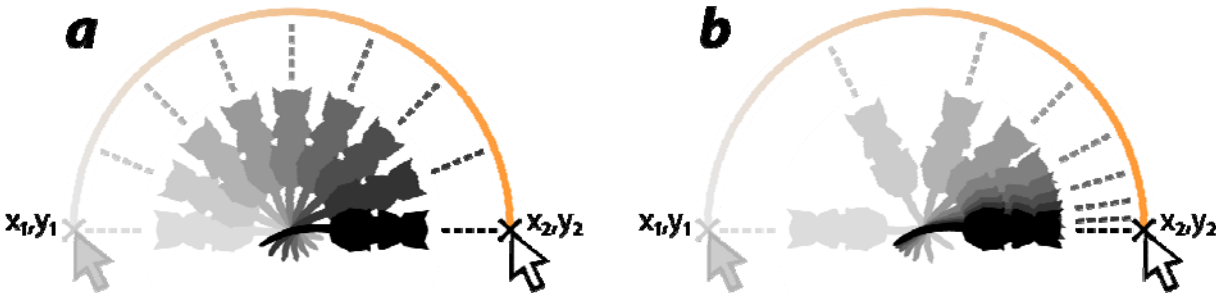
Logika je bila zelo preprosta, zato s pisanjem kode nisem imel težav. Zagotoviti sem moral le, da se mačka v zmedi, ko sta avto zaznali obe tipali, ni odzvala narobe. Ko je nastala zmeda, je izmed dveh možnosti morala izbrati najbolj gotovo. To sem dosegel tako, da sem za vsak avtomobil v krožišču preveril, ali je zadel v katero izmed tipal. Vsakič, ko je bilo določeno tipalo izbrano, se je to prištelo k rezultatom. Na koncu sem moral preveriti le še to, katero izmed tipal je bilo izbrano večkrat. Ta primitivni način zaznavanja je mački na krožišču, polnem nevarnosti, omogočil preživetje.

3.5.2 Sledenje miškinemu kazalcu

Prišel je čas, ko se mi je začelo dozdevati, da avtomobili mačke niso povozili, čeprav so več kot očitno peljali čeznjo. Pri iskanju hroščev, ki so bili odgovorni za to nenavadno dogajanje, mi obstoječa vedenjska vzorca mačke nista prav veliko koristila, kajti potreboval sem več kontrole glede narave trka. Tako se je rodila ideja o novem vedenjskem vzorcu, s katerim bi

mačka sledila kazalcu miške. S tem bi jo lahko vodil in natančno preučil morebitne anomalije v trku z avtomobili.

Da bi mačka pravilno tekla proti miškinem kazalcu, je morala uporabljati logiko, podobno tisti, ki sem jo vnesel pešcu, da je lahko hodil proti ciljem. S to kodo bi se obračanje mačke začelo prav tako sunkovito, kot bi se končalo, čemur sem rekel enakomerna rotacija (*slika 56a*). Med takšnim obračanjem je vsak zasuk enako velik, zato se mačka ob koncu obračanja ne bi zaustavila gladko, temveč v trenutku. Za razliko od mačke je pri pešcu to bilo videti solidno, saj je grafično precej preprostejši, poleg tega pa se dejansko ne vidi, kam je zasukan. Zaradi vsega omenjenega sem si za mačko želel obračanje, ki bi vsebovalo sunkovit začetek in bolj gladek konec, kar sem poimenoval neenakomerna rotacija (*slika 56b*). Za učinkovito obračanje po primeru b moramo imeti na razpolago tako začetno kot končno rotacijo, torej obe hkrati.



Slika 56: Prikaz enakomernega in neenakomernega 180 ° obrata mačke

Vir: Lasten

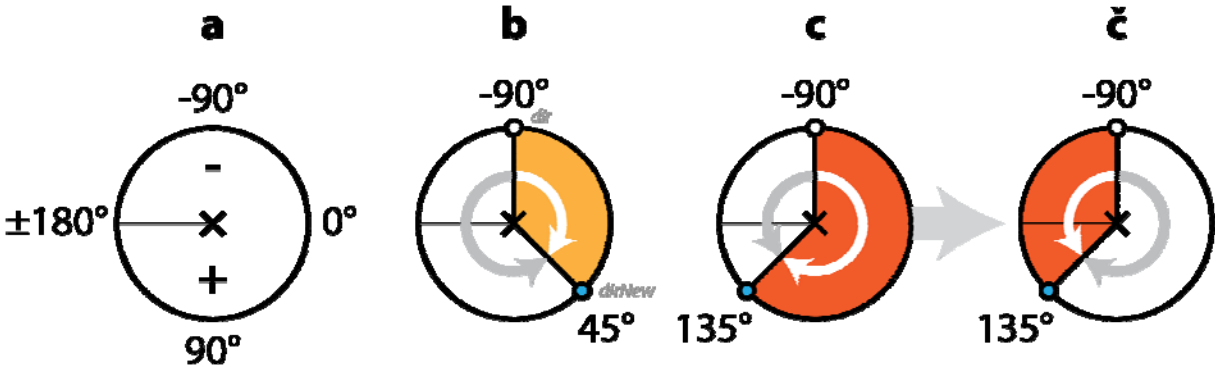
Z neenakomernim načinom premikanja sem v času dela na projektu že bil seznanjen, toda le v navadnem dvodimenzionalnem koordinatnem sistemu, kjer ga dosežemo s pomočjo preproste računske operacije (*tabela 3a*). To logiko sem pretvoril za uporabo pri neenakomerni rotaciji (*tabela 3b*). Napisal sem tudi različico te formule za enakomerno rotacijo (*tabela 3c*). Prav to izvedenko sem uporabljal, dokler nisem bil prepričan, da sistem deluje.

	Formula
a	$x_1 + (x_2 - x_1) * vrednost$
b	dir + (dirNew – dir) * *vrednost*
c	dirNew – dir
d	dir + (dirTarget – dir) * *vrednost*

Tabela 3: Formuli za izračun gladkega premika oziroma obrata

Vir: Lasten

Tu je *dir* trenutna rotacija, medtem ko je *dirNew* nova rotacija, torej kamor želimo, da se mačka obrne. Pri *vrednosti* gre za število, ki regulira hitrost premikanja po tej osi in mora biti manjše od 1. Če je enako 1, se objektu rotacija sploh ne spremeni. Za razliko od človeka računalnik ne zna intuitivno prepoznati prave smeri zasuka, zato je bilo treba rešitev poiskati s pomočjo računskih in logičnih operacij. S pomočjo formule c sem izračunal najkrajšo možno rotacijo, ki pa včasih ni bila pravilna. Zakaj se je to dogajalo, bom pokazal na naslednjem primeru, toda preden nadaljujem, moram omeniti, da funkcija za iskanje kota med mačko in kazalcem[3] vrne vrednosti, ki se gibljejo od -180 ° do 180 °, kar je razlog za nenavaden prikaz stopinj v krogu (*slika 57a*).



Slika 57: Težave pri računskem iskanju pravilnega obrata

Vir: Lasten

Če s formulo c preverimo primer b (*slika 57*), vidimo, da je pravilna dolžina zasuka najdena korektno *(tabela 4a, primer b*) in s tem tudi smer. Če pa isto formulo uporabimo za primer c (*slika 57*), pa rezultat (*tabela 4a, primer c*) ni to, kar želimo, saj nismo dobili najkrajšega možnega obrata. Najkrajši je v nasprotno smer urinega kazalca in je prikazan v primeru č (*slika 57*), kjer se vidi, da bi dejansko moral iti preko točke, kjer se zgodi preskok od -180 ° do 180 °.

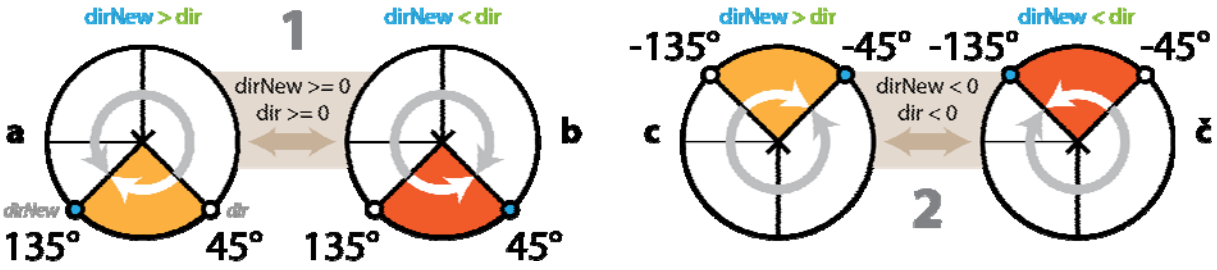
	Formula	Primer **b**	Primer **c**
a	| dirNew – dir |	| 45 – (-90) | = 135	| 135 – (-90) | = 225
b	| dirNew – (360 + dir) |	| 45 – (360 + (-90) | = 225	| 135 – (360 + (-90) | = 135

Tabela 4: Izračun primerov b in c

Vir: Lasten

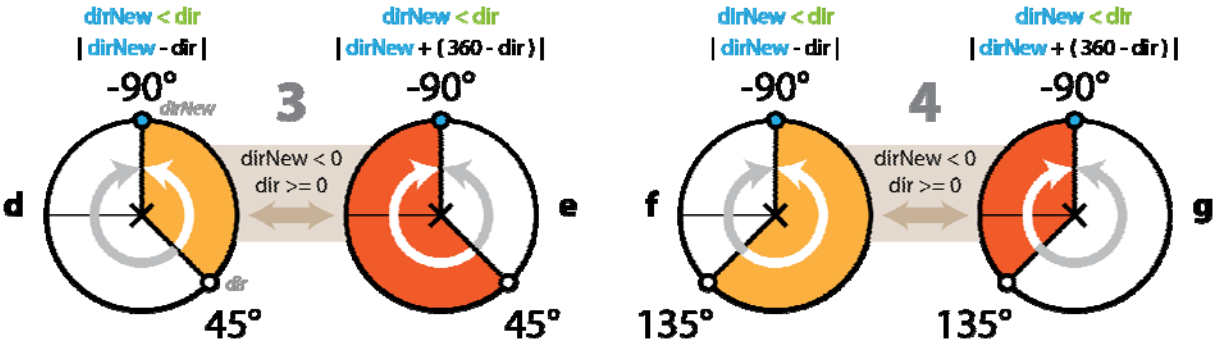
[3] To je tista, ki je bila uporabljena pri pešcu, kadar je iskal kot glede na cilj.

Da sem to težavo rešil, sem moral uporabiti formulo, ki bi število, ki je v minusu, seštela s 360 (*tabela 4b*). S tem sem -90 spremenil v 270, kar je še vedno isti kot, le prirejen. Ta kot sem lahko brez težav vstavil v obstoječo formulo b (*tabela 3*) namesto *dirNew* in dobil pravilen rezultat[4].



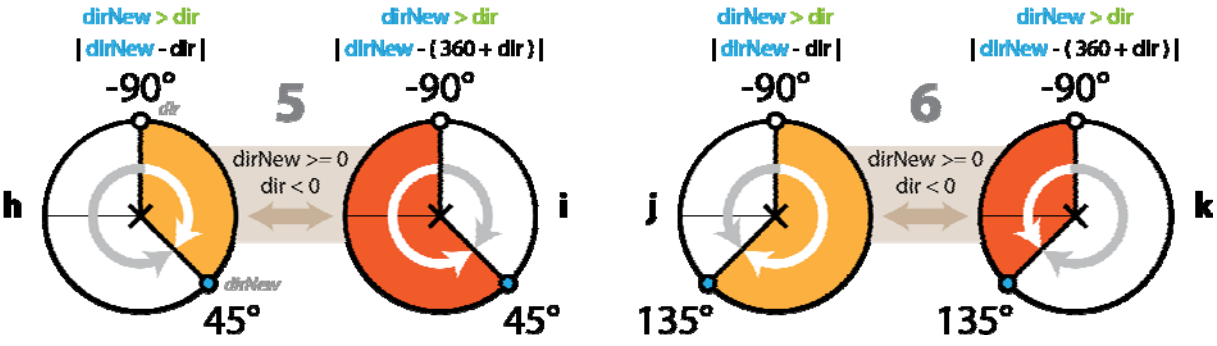
Slika 58: Primer računanja pravilnega obrata, kjer sta *dir* in *dirNew* pozitivna (a in b) ali negativna (c in č)

Vir: Lasten



Slika 59: Primer računanja pravilnega obrata, kjer je *dir* pozitiven in *dirNew* negativen

Vir: Lasten



Slika 60: Primer računanja pravilnega obrata, kjer je *dir* negativen in *dirNew* pozitiven

Vir: Lasten

[4] V tem primeru dobimo pravilen rezultat samo, če odmislimo atribut *vrednost*.

Bolj ko sem preizkušal primere, prej sem opazil, da mi te formule ne bi bilo več treba spreminjati, samo pravilno prirejen kot, ki sem ga poimenoval *dirTarget*, bi bilo treba najti. S tem sem dobil novo formulo (*tabela 3d*), ki je od tod naprej ostala nespremenjena. Spreminjal se je samo še *dirTarget*, kar mi je precej olajšalo delo. Zanj sem moral napisati kup formul, ki bi ga izračunale. Poleg tega sem moral zasnovati še logični sistem, ki bi izmed omenjenih formul vedno izbral najustreznejšo. Z intenzivnim primerjanjem različnih primerov sem tako prišel do formul, ki jih prikazujejo slike 58, 59 ter 60 in ki so dokazane v tabeli 5.

	Primer	Pogoj	Formula	Račun	Smer
a	1	dirNew >= 0 dir >= 0	dirNew > dir	135 > 45 = da	✓
b			dirNew < dir	135 < 45 = ne	
c	2	dirNew < 0 dir < 0	dirNew > dir	-45 > -135 = da	✓
č			dirNew < dir	-45 < -135 = ne	
d	3	dirNew < 0 dir >= 0	| dirNew – dir |	| –90 – 45 | = 135	✓
e			| dirNew + (360 – dir) |	| –90 + (360 – 45) | = 225	
f	4	dirNew < 0 dir >= 0	| dirNew – dir |	| –90 – 135 | = 225	
g			| dirNew + (360 – dir) |	| –90 + (360 – 135) | = 135	✓
h	5	dirNew >= 0 dir < 0	| dirNew – dir |	| 45 – (–90) | = 135	✓
i			| dirNew - (360 + dir) |	| 45 – (360 + (–90) | = -225	
j	6	dirNew >= 0 dir < 0	| dirNew – dir |	| 135 – (–90) | = 225	
k			| dirNew - (360 + dir) |	| 135 – (360 + (–90) | = 135	✓

Tabela 5: Dokazi za primere s slik 58, 59 in 60

Vir: Lasten

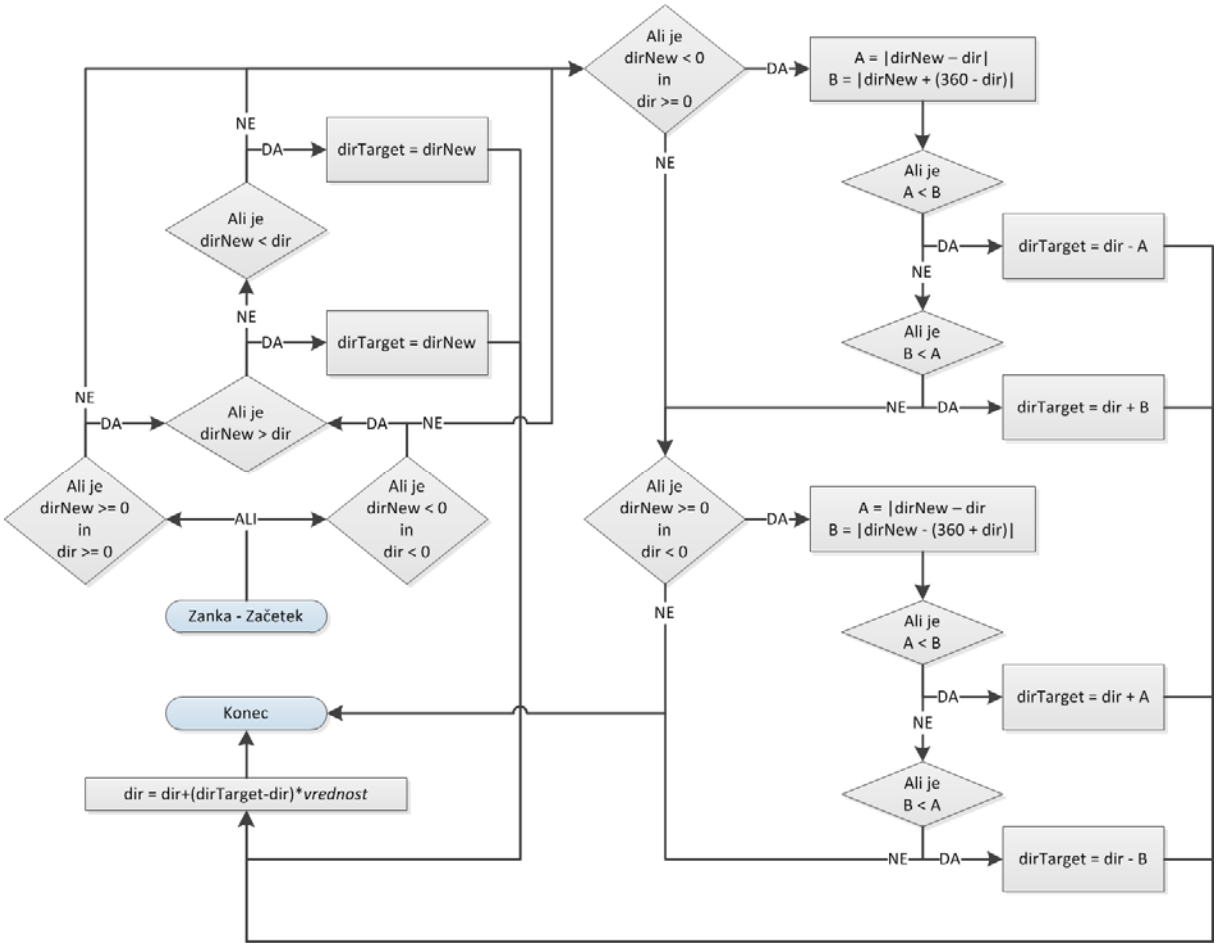
Kot vidimo iz primerov, sem ustvarjene formule razdelil na tri kategorije, ki so ločene po tem, ali so vrednosti pozitivne ali negativne.

1. kategorija – *dir* in *dirNew* sta bodisi pozitivna (a in b) bodisi negativna (c in č).
2. kategorija – *dir* je pozitiven, *dirNew* je negativen (d, e, f in g).
3. kategorija – *dir* je negativen, *dirNew* je pozitiven (h, j, i in k).

Čeprav je črk dvanajst, gre tu v resnici za šest primerov. Vsi prikazani primeri namreč delujejo v parih, kar pomeni, da je v izbranem primeru na koncu izmed dveh rezultatov pravilen samo eden. Ta je v tabeli zapisan pod »Smer«. Pri primerih a–č se izbere pravilen odgovor, medtem ko se pri d–k izbere tisto število, ki je manjše, saj označuje najkrajši obrat.

Z namenom, da se vidi, kako sem te formule implementiral v logični sistem, v katerem s pomočjo zgoraj omenjenih formul za računanje trenutne (*dir*) in nove rotacije (*dirNew*)

iščemo prilagojeno rotacijo (*dirTarget*), sem kot zanimivost priložil sliko 61, ki prikazuje poenostavljen graf poteka te logike.



Slika 61: Poenostavljen graf poteka logike za najdbo ustrezne prilagojene rotacije *dirTarget*

Vir: Lasten

Z omenjenimi načini ter minimalnim igranjem z vrednostmi spremenljivk, ki so bile zadolžene za občutek neenakomernega zasuka, mi je uspelo ustvariti gladko obračanje mačke proti miškinemu kazalcu. Narediti sem moral samo še to, da se je mačka začela premikati, kar pa je bilo izredno enostavno, saj sem lahko uporabil enak način kot pri pešcu.

3.5.3 Težava z globino

Pojavila se je zanimiva težava, ki je bila rezultat Flashevega načina upravljana z globino objektov. Spomnimo se, da objekti, ki nastanejo kasneje, v Flashevem seznamu globine končajo bolj v ospredju kot objekti, ki nastanejo prej. Lahko rečemo, da se kot plasti (*layers*) nalagajo eden na drugega. Spomnimo se tudi, da dva objekta ne moreta imeti enakega števila globine.

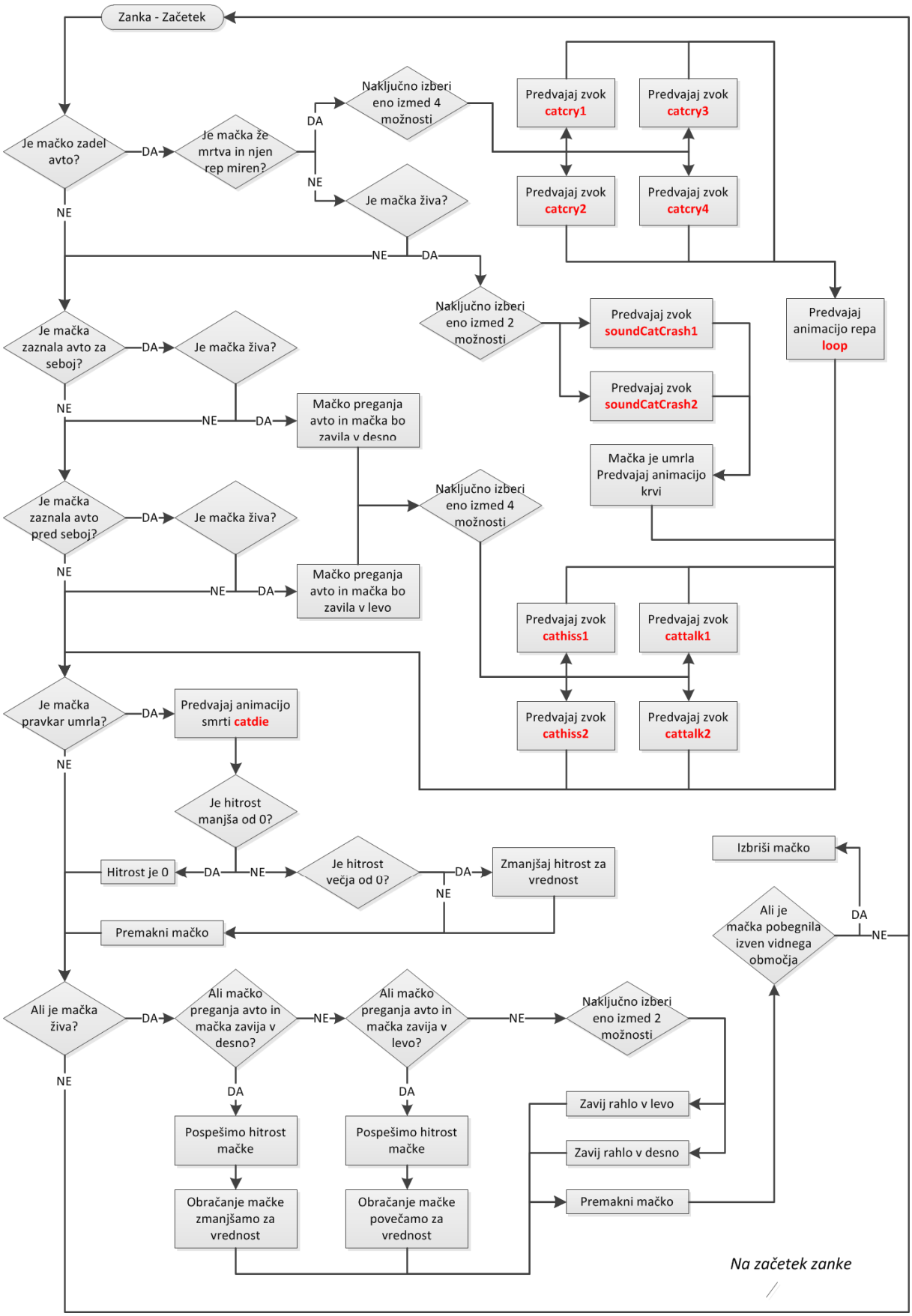
Dokler so zadnji nastajali avtomobili, ni bilo težav, saj ni bilo nič narobe, če je bila njihova grafika po globini pred drugimi objekti. Težave so se pojavile pri nastanku mačke, ki je hodila po rastlinah in avtomobilih, namesto da bi tekla pod njimi (*slika 62*). Edino, kar sem v tem primeru lahko naredil, je, da sem vnaprej ustvaril prazen MC, ki je v globini stal med tistimi objekti, ki morajo biti pred mačko, in tistimi, ki morajo biti za njo. Edini način manipulacije globine, ki jo Flash omogoča, je namreč ta, da jo izmenjamo med dvema objektoma. Zato sem ukaz spremenil tako, da je ob nastanku mačka globino tega objekta prevzela, ob smrti oziroma izbrisu pa mu jo vrnila. To izmenjavo globine sem dosegel z uporabo funkcije `swapDepths()`.

Slika 62: Težava z globino mačke

Vir: Lasten

3.5.4 Logika

Težko si je predstavljati, kaj vse se mora sprožiti, da mačka dela to, kar pač dela. Zato sem za boljšo predstavo o tem, kaj se dejansko dogaja v drobovju mačke, priložil graf poteka, ki poenostavljeno prikazuje njeno logiko (*slika 63*).



Slika 63: Poenostavljen graf poteka mačkine logike

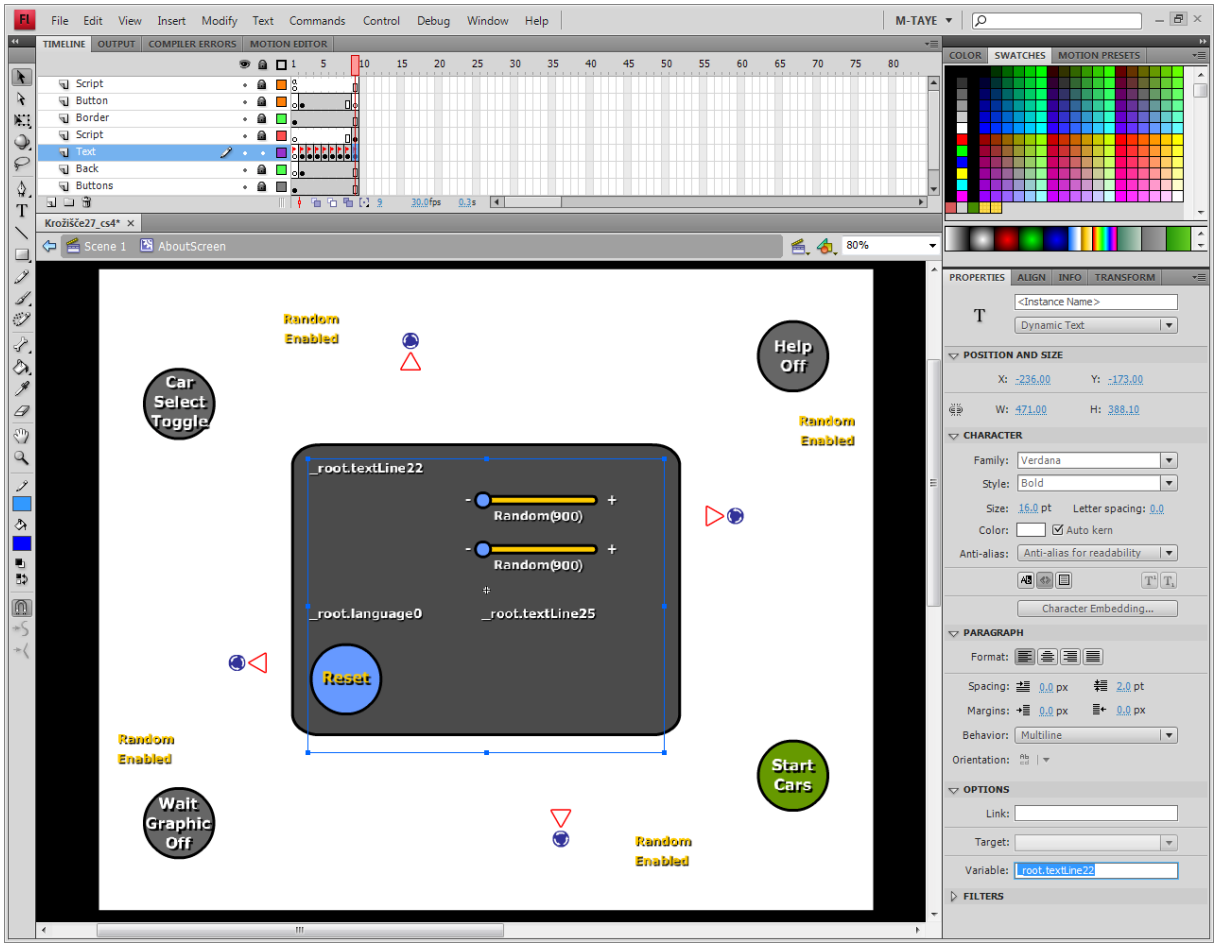
Vir: Lasten

3.6 *Uporabniški vmesnik*

O interakciji z avtomobili ter mačko sem že govoril, zato bom tu opisal preostale elemente, ki sestavljajo uporabniški vmesnik.

3.6.1 *Meni in gumbi*

Sprva sploh nisem imel namena narediti menija, temveč samo gumbe, ki sem si jih označil na skici z začetka projekta. Za meni sem se odločil komaj kasneje, ko sem se zavedal, da bi bilo zanimivo, če bi aplikacija vsebovala nekakšno pomoč, kjer bi bile na kratko razložene funkcije gumbov. Naloge sem se lotil tako, da sem ustvaril MC, ki bi vseboval vse gumbe ter meni. Njegov prvi *frame* je bil rezerviran samo za gumbe, ki bi naj bili prikazani nenehno. To so tisti, ki jih ob zagonu aplikacije vidimo takoj. Naslednji *frame*-i pa so bili rezervirani za različna stanja menija, ki se je prikazal ob kliku na gumb *Pomoč*. Ob ponovnem kliku istega gumba se je meni ugasnil.



Slika 64: MC *AboutScreen* vsebuje poleg vseh gumbov tudi meni ter prometne znake

Vir: Lasten

S premikom miškinega kazalca na sam meni se da dostopati do nastavitev. Tukaj se nahajata gumb za ponastavitev števil seštevanja prometa ter gumb za spremembo jezika. Prav tako se tu nahajata dva drsnika – prvi uravnava prehodnost avtomobilov skozi krožišče, medtem ko drugi omejuje število dovoljenih pešcev na ekranu.

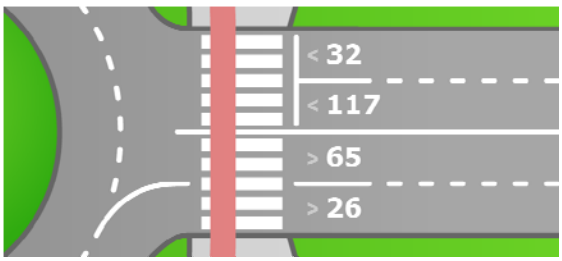
Zagotoviti sem moral, da je v Flashevem seznamu globine ta MC vedno nad drugimi objekti, zato sem tako avtomobilom kot pešcem ter mački vnesel kodo, ki je ob njihovem nastanku zamenjala lastno globino z globino tega MC-ja. Vsi elementi, ki so bili v tem MC-ju, so nato prav tako bili pred drugimi.

3.6.2 Menjava jezika

Najprej sem imel namen tekst napisati kar v okno, vendar sem se odločil, da ga bom raje shranil v spremenljivke. To sem storil zaradi ideje o možnosti zamenjave jezika. Vse besedilo sem napisal vnaprej tako v slovenščini kot angleščini, vsako v svojih spremenljivkah. Eno izmed teh besedil je bilo izbrano privzeto in to se je avtomatsko prekopiralo v delovne spremenljivke, katerih vsebina se je nato prikazala v oknu. Za prikaz teksta iz spremenljivk sem uporabil dinamično tekstovno polje (*Dynamic Text*). Zamenjavo jezika sem dosegel tako, da sem naredil gumb, ki je ob kliku preprosto spremenil, kateri izmed tekstov je bil izbran, in ga prekopiral v delovne spremenljivke. S tem sem dosegel priročno menjavo jezika.

3.6.3 Seštevanje prometa

Povsem slučajno sem se spomnil, da bi bilo zanimivo prikazati tudi, koliko avtomobilov je peljalo skozi posamezne pasove krožišča. Tako sem na vsakem kraku krožišča ustvaril dinamična tekstovna polja, ki so za vsak avtomobil to količino beležila.



Slika 65: Seštevanja prometa na kraku

Vir: Lasten

3.6.4 Izbira avtomobilove smeri z gumbi

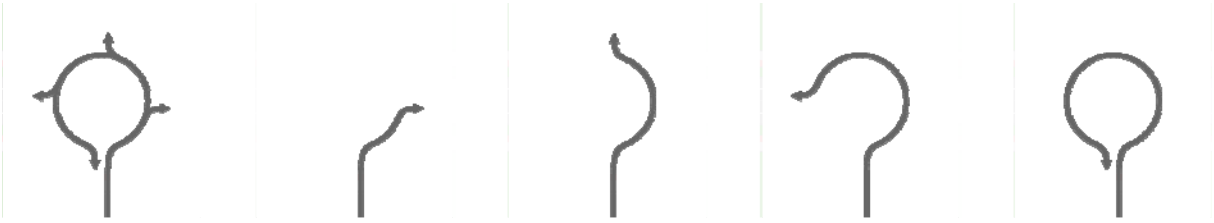
Preden se je dalo avtomobilu spremeniti smer s pomočjo miške in tipkovnice, so to storili prav za to namenjeni gumbi. Vsak izmed njih je deloval za svoj krak in imel za izbiro na razpolago več možnosti – *prvi*, *drugi*, *tretji* in *četrti izhod* ter *naključno*, skozi katere se da prehajati s klikom. Prav zaradi te spremenljive funkcionalnosti, kjer mora gumb vsakič

narediti nekaj drugega, sem se odločil, da bom za njegovo drobovje namesto *Button*-a uporabil MC. S tem in z nekaj kode v AS-u sem dosegel, da ima gumb več možnih stanj.

Na začetku sem naredil tako, da je klik gumba spremenil le privzeto izbiro smeri, ki so jo dobili avtomobili, ki so nastali na istem kraku. To je pomenilo, da se je smer spremenila komaj tistim avtomobilom, ki so nastali po spremembi, ne pa tudi tistim, ki so takrat že obstajali in so bili v koloni oziroma so že vozili v krožišču. S tem nisem bil zadovoljen, saj sem prišel do spoznanja, da bi bilo bolj smiselno, če bi se smer spremenila tudi tistim avtomobilom, ki so čakali v koloni, a še niso zapeljali v krožišče. Da pa bi to lahko storil, bi moral te avtomobile najprej najti.

Najprej sem MC-ju avtomobila dodal spremenljivko, ki je beležila, ali je še v koloni. Nato sem napisal algoritem za spremembo smeri, ki je preletel vse avtomobile, ki so bili na ekranu. Med njimi je preveril, ali so nastali na enakem kraku, na katerem sem se odločil spremeniti smer, in če so, je nato preveril samo še, ali so se tedaj nahajali v koloni. V kolikor je bil tudi zadnji odgovor pritrdilen, je ta avtomobil spadal med tiste, katerim sem želel spremeniti smer. Po zaslugi te funkcije sem dosegel, da se je smer spremenila tudi avtomobilom v koloni.

Proti koncu se mi je zdelo, da morda ni dovolj jasno razumljivo, kako ti gumbi delujejo, zato sem ustvaril še dodaten grafični prikaz izbrane poti, ki je le-to prikazal jasno in nazorno.



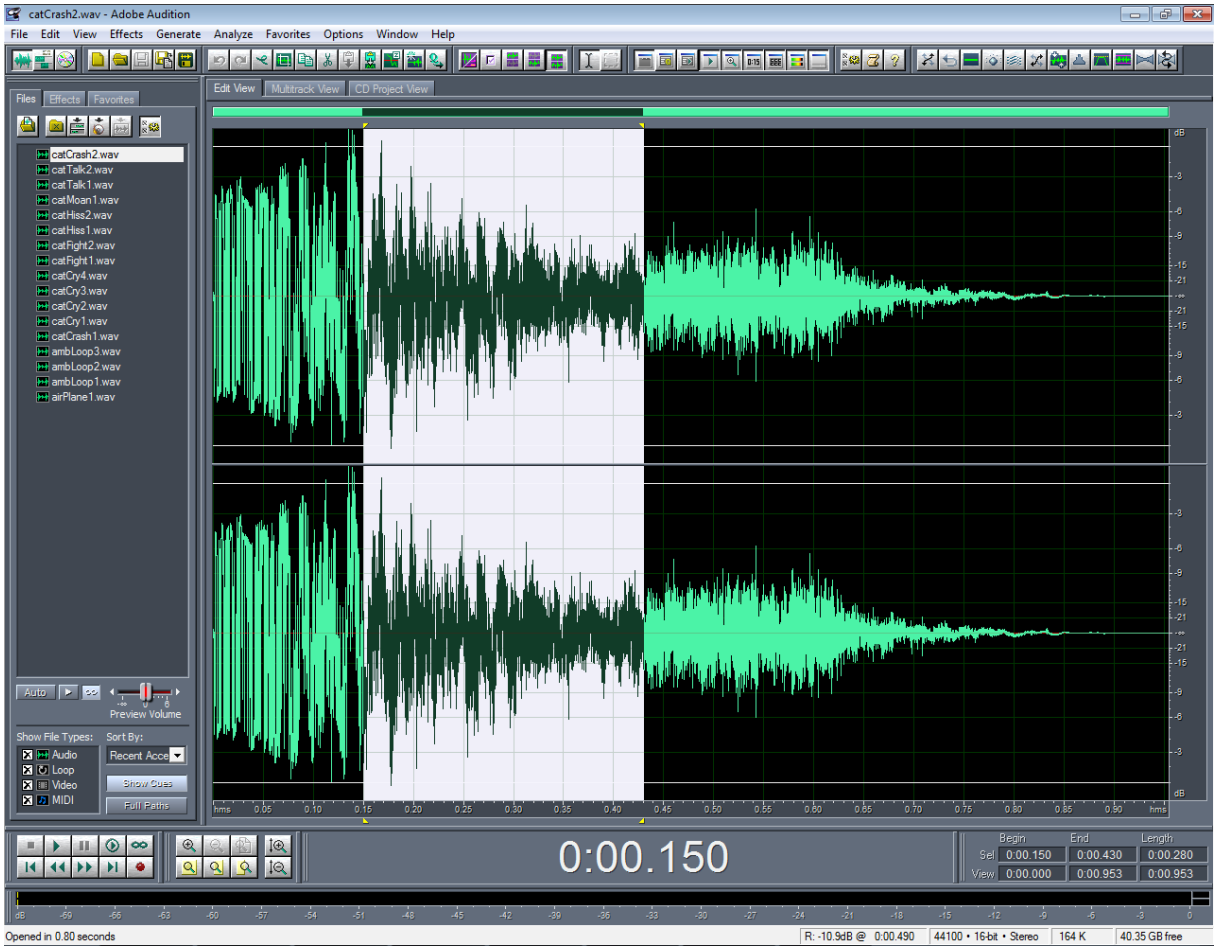
Slika 66: Grafični prikaz izbrane poti – *naključno*, *prvi*, *drugi*, *tretji* in *četrti izhod*

Vir: Lasten

3.7 Zvok

Krožišče je bilo brez zvoka nekako žalostno, saj mu je manjkalo tistega znanega prijetnega vzdušja, ki ga ne moremo dobiti samo z vizualnimi dražljaji. Zaradi tega sem bil prav radoveden, kako mi bo uspelo različne zvoke preplesti v enotno vzdušje. Ampak preden sem se tega sploh lahko lotil, sem moral zvoke najti.

Vedel sem, da na spletu obstaja kup spletnih strani[5], ki ponujajo zvočne učinke ter glasbo po licenci Creative Commons[6], kjer lahko avtor izdelka s pomočjo vnaprej pripravljenih licenc jasno določi, kateri del pravic da v prosto uporabo. Teh licenc je več, zato sem pazil, da sem pri iskanju zvočnih datotek uporabil le tiste, katerih licenca se je skladala z naravo mojega projekta[7] in dejstvom, da sem nekatere zvoke imel namen obdelati.



Slika 67: Adobe Audition 1.5

Vir: Lasten

[5] Te so napisane pod točko 5. Literatura in viri.

[6] Več o Creative Commons najdete na www.creativecommons.org, www.creativecommons.si, dne 15. 12. 2010

[7] Gre za neprofitni in nekomercialni izdelek.

Zvoke sem iskal na naslednjih spletnih straneh:

- http://www.flashkit.com/soundfx/, dne 15. 12. 2010
- http://www.freesound.org/, dne 15. 12. 2010
- http://www.partnersinrhyme.com/pir/PIRsfx.shtml, dne 15. 12. 2010
- http://www.soundsnap.com/, dne 15. 12. 2010
- http://amazingsounds.iespana.es/en/, dne 15. 12. 2010
- http://www.pachd.com/sounds.html, dne 15. 12. 2010
- http://www.mediacollege.com/downloads/sound-effects/, dne 15. 12. 2010

Kar se ustreznosti zvokov tiče, sem za nekatere takoj vedel, da jih bom uporabil, medtem ko sem moral druge najprej vstaviti v izdelek, da sem lahko ocenil, ali so pravšnji. Zvokov nisem mogel uporabiti kar takšnih, kot sem jih dobil, saj sem jih moral prilagoditi za uporabo v Flashu. Nekatere sem na primer zmešal skupaj, drugi so bili predolgi, zato sem jih skrajšal, in podobno. Za te potrebe sem uporabil program Adobe Audition.

3.7.1 Zvok križišča

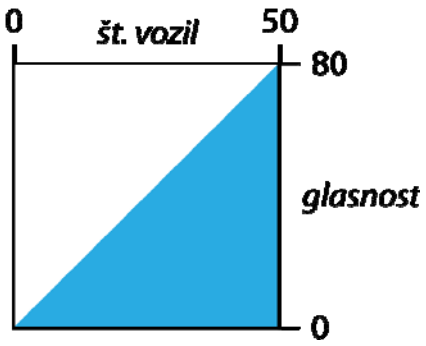
Najprej sem imel nenavadno zamisel, da bi zvok prometa dosegel s tem, da bi vsak avto sam predvajal svoj zvok vožnje. Ne glede na trud se zamisel ni obnesla, saj je bil zvok preveč monoton. Tako sem prišel do zamisli, da bi zvočno vzdušje križišča ločil na dve fazi, in sicer na fazo mirovanja ter fazo prometa.

Mirovanje

S to fazo imamo opravka takrat, kadar na cestišču ni nobenega avtomobila. Za ta del sem izbral zvok mirne soseske z blagim šumom prometa v ozadju. Ta zvok se v križišču sliši vedno in glasnost se mu ne spreminja.

Promet

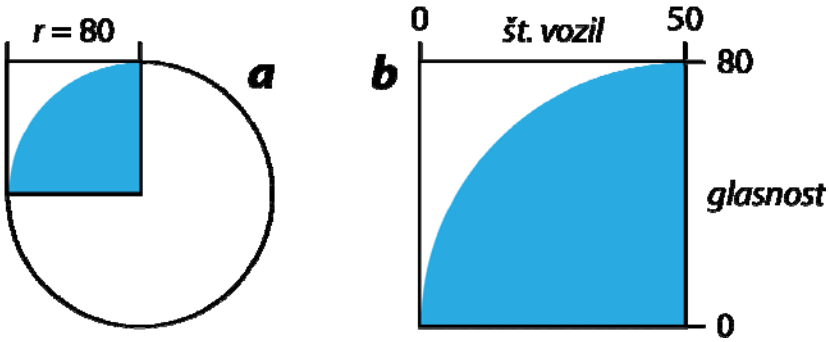
Ta faza se pojavi, ko po križišču vozi kakršno koli število vozil – od enega avtomobila do petdeset, kjer je meja njihovega nastajanja. Tu sem uporabil zvok hrupa prometa, ki pa je bil dokaj glasen, zato ga za razliko od zvoka v fazi mirovanja nisem mogel uporabiti kar tako, saj se je slišal prehod. Zamisliti sem si moral nek dinamičen način glasnosti glede na količino avtomobilov na ekranu. Kaj hitro sem prišel do naslednjega grafa (*Slika 68*).



Slika 68: Graf glasnosti glede na število vozil

Vir: Lasten

S tem načinom sem bil najprej zelo zadovoljen, toda sčasoma me je začel motiti. Zakaj, ko je bila na ekranu polovica avtomobilov, je bil promet že zelo gost. Glasnost je bila tedaj komaj na polovici končne vrednosti. Če sem od tod naprej v krožišče spustil še ostalo polovico avtomobilov, se je glasnost povečala še za enkrat toliko. To mi ni odgovarjalo, saj se v realnem svetu glasnost ne poveča sorazmerno s številom avtomobilov, torej v ravni črti, temveč prej v obliki krivulje.



Slika 69: Graf glasnosti glede na število vozil v obliki krivulje

Vir: Lasten

Najbolj primeren kandidat za poosebitev te krivulje se mi je zdela četrtina kroga (*slika 69a*). Po grafu sodeč vidimo, da glasnost ob malem številu avtomobilov narašča hitro (*slika 69b*), ko pa se začnemo bližati zgornji meji vozil, se ublaži. S tem je zvok prometa hitro postal glasen, na kar se njegova glasnost več ni pretirano večala. Do formule, ki je skrbela za računanje te glasnosti (*slika 70*), sem prišel s kombinacijo trigonometrije in lastne pameti.

$$\sin\left(\operatorname{acos}\left(\left(80-\left(\frac{\text{št.vozil}*2}{100}\right)*\left(\frac{\text{št.vozil}*2}{80}\right)\right)\right)\right)*80$$

Slika 70: Formula za glasnost

Vir: Lasten

Tu lahko omenim tudi, da sem letalu prav tako dodal zvok. Šlo je za zvok preleta, ki je še poudaril prometno vzdušje okolja.

3.7.2 Zvok mačke

Če se je kdaj zares opazilo, da je manjkal zvok, je bilo to takrat, kadar je avtomobil povozil mačko in slišala se je le tišina. Zvok trka z avtom je bil torej prvi, na katerega sem se spomnil. Ob spremljanju obnašanja mačke pa sem se spomnil še preostalih (*tabela 6*).

Za predvajanje teh zvokov sem potreboval nekaj zmogljivejšega kot samo monotono ponavljanje enega in istega. S tem v mislih sem mački ustvaril dodaten MC, ki je vseboval vse omenjene zvoke. Tako mi je preostalo le, da sem v logiko mačke vnesel dejanja, ki so sprožila predvajanje izbranega zvoka. Pri zvokih trka v avto sem si dal duška in ju oblikoval tako, da se nesreča sliši hudo ter komično hkrati. Ostalih zvokov vsebinsko nisem spreminjal.

Ime zvoka	Razlaga zvoka
soundCatCrash1	Udarec v avto, ki mu sledita mačji krik ter avtomobilska hupa.
soundCatCrash2	Udarec v avto, ki mu sledita mačji krik ter blagi zvok trganja mesa.
cathiss1	Različna zvoka pihanja mačke
cathiss2	
cattalk1	Različna zvoka mijavkanja mačke
cattalk2	
catcry1	Številni zvoki jokanja mačke
catcry2	
catcry3	
catcry4	

Tabela 6: Seznam vseh zvokov za mačko

Vir: Lasten

Sploh se ne zavedamo, kaj nam manjka, dokler manjkajočega ne občutimo – nekako tako lahko na kratko opišem svojo izkušnjo z zvokom.

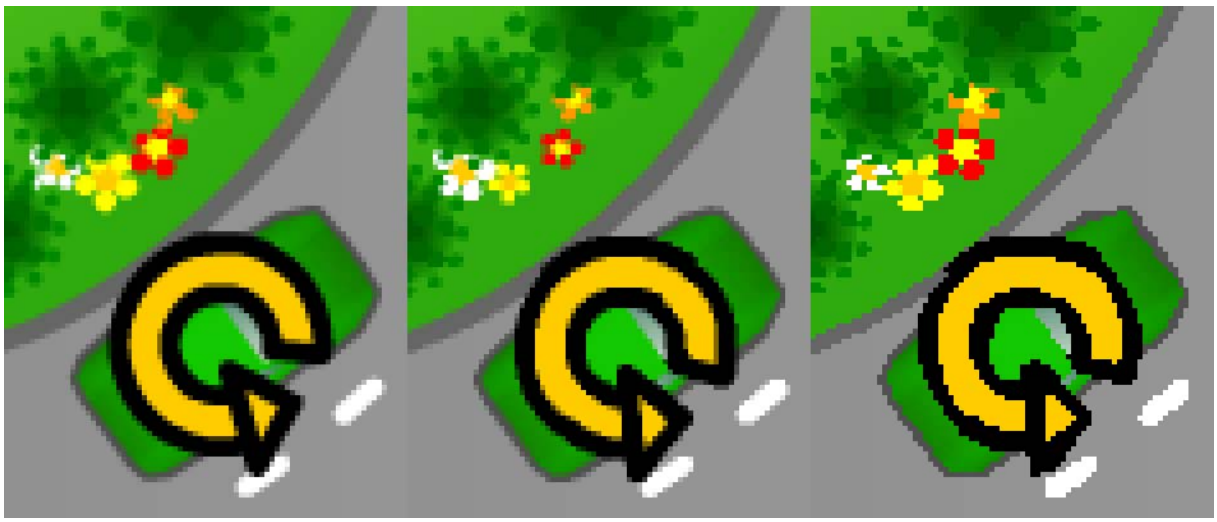
3.8 Optimizacija

Ko se projekt bliža koncu, pride čas optimizacije delovanja aplikacije. Pri tem gre za prečiščevanje delov aplikacije z namenom, da ta začne teči hitreje. Čeprav je ta faza v glavnem namenjena pridobivanju hitrosti, sem precejšnji del namenil tudi iskanju in odstranjevanju napak.

3.8.1 Počasna grafika

Motilo me je, da krožišče na počasnejših računalnikih ni delovalo gladko. S pomočjo preprostega preizkusa, kjer sem vso grafiko pretvoril v črne kocke, sem ugotovil, da za to ni bila kriva obširnost kode v AS, temveč podrobna vektorska grafika. Za izris črnih kock namreč ne potrebujemo toliko procesorske moči kot za zapletene oblike z barvnimi prelivi, zato je bila ob spremembi pohitritev več kot očitna. Seveda nisem želel, da bi po ekranu vozile črne kocke, zato sem zmanjšal količino podrobnosti ter zbrisal določene prelive.

S tem sem krožišče malenkost pohitril, a vseeno ne dovolj, da na slabših računalnikih ne bi bilo težav. Za gladkost sem bil pripravljen poseči tudi po sledeči drastični metodi – v Flashu sem privzeto kakovost izrisa[8] s *high* zmanjšal na *medium*. To je obremenitev procesorja zmanjšalo kar precej, toda še vedno nisem bil tam, kjer sem želel. Po dolgotrajnem sprehajanju več nisem vedel, kaj bi optimiziral, in ker pri kakovosti izrisa na manj od *medium* nisem želel iti, sem se enostavno moral sprijazniti s končnim rezultatom.



Slika 71: Povečan prikaz razlike v kakovosti izrisa – *high*, *medium* in *low*

Vir: Lasten

[8] Možnosti, ki so na razpolago, so *high*, *medium* in *low*.

3.8.2 Lov za hrošči

Vedno, ko sem imel občutek, da je krožišče delovalo brez težav, se je pojavil nov hrošč[9], ki je imel potencial, da me razjezi. Lov za hrošči je upravičeno najmanj priljubljen del programiranja, še posebej pri projektih, kot je tale, v katerih se dogajanje ne odvija vedno enako.

Najhujši hrošč je nastopil, ko je avtomobil v krožišču kar naenkrat izginil in se v istem trenutku pojavil na drugi strani krožišča. Takrat še nisem vedel, čemu bi se to lahko zgodilo, a sem po dolgotrajnem in frustrirajočem opazovanju le ugotovil. Avtomobil je pri izhodu iz krožišča počakal na pešca in če si mu takrat z miško spremenil smer, se je teleportiral[10] na tisti krak, kamor je smer določala. Temu sem naredil konec tako, da sem avtomobilu po izhodu iz srednjega dela krožišča preprečil, da bi mu sprememba smeri spremenila njegovo pot. To je bil edini hrošč, ki me je zares presenetil, saj sem bil prepričan, da promet deluje brezhibno.

[9] Izraz *hrošč* ali *bug* izvira iz programiranja in predstavlja napako v kodi.

[10] Teleport je nekaj, kar omogoča instantno potovanje skozi čas in prostor. Lahko je bodisi stroj bodisi portal.

4 ZAKLJUČEK

Ob začetku projekta nisem imel popolnoma realiziranega načrta. Zapisane sem imel le cilje, ki so me opomnili, katere elemente sem želel izdelati. Zaradi tega sem napredoval počasneje, kot bi želel, saj sem si delovanje posameznih delov krožišča izmišljeval sproti. Čeprav bi z bolje nastavljenim in podrobnejšim načrtom projekt dokončal v krajšem času, to tukaj ni prišlo v poštev, saj menim, da za tak načrt tedaj nisem imel dovolj izkušenj. K temu je veliko pripomoglo tudi prepričanje, da sem se spuščal v nekaj neznanega. Dodaten razlog je bila tudi precejšnja količina elementov, ki bi naj bili vodeni z AS-om. Programirati sem znal le za vzorec, zato sem skušal slediti najbolj očitni smeri, tudi če je to pomenilo, da je bilo treba napredovati po polžje.

S tem, ko sem odstranil čas kot ključen dejavnik pri razvoju izdelka, sem imel lepo možnost eksperimentirati, kar sem največ počel v kombinaciji pisala in lista papirja, veliko pa tudi s spreminjanjem lastnosti posameznih delov aplikacije. Ko sem pri snovanju logike prišel do potencialne rešitve, sem se jo odpravil prenesti v kodo. Ker AS zajema veliko funkcij, bi bilo nerealno pričakovati, da sem poznal vse že od samega začetka. Zato je velik razlog za neboleč napredek pri pisanju kode bila možnost najti pomoč. Tukaj sem imel srečo, saj Flash vsebuje izvrstno bazo pomoči[11], kjer je razloženo, kako se sleherna funkcija vede. Pridodani so tudi bolj ali manj jasni primeri, ki lahko pomenijo razliko med uspehom in brezuspešnim razbijanjem glave. Ni pa to bil edini vir pomoči pri ustvarjanju tega projekta, saj sem poleg tega izvedel veliko tudi z brskanjem po spletu. Omenjena pomoč mi je pomagala le pri razumevanju posameznih funkcij AS-a, ne pa tudi pri ustvarjanju logike, kjer sem bil prepuščen samemu sebi.

Zaradi precejšnje vsebnosti AS-a delo ni potekalo linearno, temveč v nekakšni sinusni krivulji poteka dela skozi čas in prostor. Težko je opisati občutek, ki prevladuje v programiranju, toda dejstvo je, da sem zaradi nevidnih povezav med objekti in AS-om moral pri ustvarjanju posameznih elementov krožišča opraviti precej skakanja iz enega dela projekta v drugega. Zgodilo se je celo, da je bilo treba zaradi majhne spremembe kode na enem mestu spremeniti velik del kode na drugem, da ne omenim napak, ki so se mi dogajale zaradi površnosti pri pisanju kode. Flash ti jih sicer javi, toda nekaterih ne opiše pravilno, zato sem se včasih znašel v položaju, ko sem skušal popraviti popolnoma delujoči del kode, medtem ko se je napaka

[11] Najdemo jo na spletni strani http://help.adobe.com/en_US/FlashPlatform/reference/ActionScript/2/help.html, dne 15. 12. 2010

skrivala povsem drugje. Prav tako je znala nastati zmeda pri iskanju hroščev, za katerih odkritje je bilo včasih treba preveriti vsak najmanjši kotiček kode.

V kolikor bi se ponovno lotil tega projekta, bi marsikaj naredil drugače, morda celo vse. Trenutno logiko, ki vodi posamezne elemente krožišča, bi gotovo popolnoma spremenil, saj so nekateri njeni vidiki preveč omejujoči. Lep primer je vnaprej pripravljena vožnja avtomobila po tirnicah, ki ne omogoča, da bi se avto ustavil, kadar se mu zahoče. Zato bi naslednjič cestišče ustvaril tako, da bi posedovalo namige o strukturi zavojev. Te namige bi avtomobili koristili za matematični izračun poti vožnje, kar bi pomenilo, da njihova vožnja več ne bi bila narejena vnaprej. Pešci so v bistvu že zdaj narejeni v takem slogu, le ustaviti bi se še morali znati. Krožišče bi s podobnimi spremembami postalo še bolj dinamično, saj bi lahko vnesel tudi način, kjer bi nekateri vozniki vozili manj po pravilih in bolj po svojih (ne)zmožnostih – tako kot v realnosti. To bi vožnjo avtomobilov naredilo znatno bolj nepredvidljivo in s tem zanimivejšo, da ne pomislim na težave, ki bi jih ob prečkanju cestišča v tej zmešnjavi imela uboga mačka. Da bi vse omenjeno delovalo s trenutno količino elementov, bi najverjetneje moral presedlati z AS-a 2.0 na AS 3.0, saj je bolje prilagojen upravljanju večjih količin objektov, poleg tega pa je njegovo izvajanje kode precej hitrejše.

Ne glede na omenjeno sem z nastalim izdelkom izjemno zadovoljen. Hkrati sem tudi presenečen, da mi je uspelo narediti vse, kar sem si zamislil. Sicer je res, da to krožišče nima neke praktične vrednosti, saj gre tu le za bežno zanimivost, a pomembnejše je dejstvo, da sem skozi izdelavo tega izdelka pridobil veliko novih izkušenj in znanj ter hkrati utrdil obstoječa. Napredoval sem predvsem pri snovanju smiselne logike in njenem pretvarjanju v kodo. S tem sem dobil dodatno motivacijo za morebitno učenje drugih programskih jezikov v prihodnosti. Največ, kar sem v obsegu tega projekta dosegel, je povečanje lastnega znanja. Upam, da bodo postopki, ki sem jih omenil v tej nalogi, v prihodnosti pomagali še marsikomu.

Menim, da današnja hitrost življenja definira način učenja, ki je potreben, da se lahko uspešno kosamo z neustavljivim napredkom v tehnologiji. Ne samo da se moramo učiti sami zase, znanje moramo usvajati čim hitreje ter čim bolje, hkrati pa po svojih sposobnostih. Tukaj nastopi splet, kjer lahko s klikom na miškino uho dostopamo do dnevno posodobljenih učnih gradiv. Ne gre samo za priročnost, temveč tudi za možnost, da dobimo prav tisto informacijo, ki jo iščemo, ne pa tudi tiste, ki je ne potrebujemo. Tehnologijo imamo, čas je, da se je začnemo posluževati stoodstotno. Navsezadnje je tehnologija stvar, ki nam res lajša življenje.

5 LITERATURA IN VIRI

5.1 Literatura

1. Russel, C.: *Adobe Flash CS4 Professional*, Adobe Press, Kalifornija, ZDA, 2009.
2. Rich, S.: *Learning Flash CS4 professional*, O'Reilly, Kalifornija, ZDA, 2009.

5.2 Spletni viri

1. http://en.wikipedia.org/wiki/Adobe_Flash/, dne 15. 12. 2010
2. http://www.adobe.com/products/flash/, dne 15. 12. 2010
3. http://livedocs.adobe.com/flash/9.0/main/flash_as2_learning.pdf, dne 15. 12. 2010
4. http://help.adobe.com/en_US/FlashPlatform/reference/ActionScript/2/help.html, dne 15. 12. 2010
5. http://www.mzp.gov.si/fileadmin/mzp.gov.si/pageuploads/DPR/Predlogi_predpisov_DPR/10_01_20-Zakon_o_pravilih_cestnega_prometa.pdf, dne 15. 12. 2010
6. http://www.flashkit.com/soundfx/, dne 15. 12. 2010
7. http://www.freesound.org/, dne 15. 12. 2010
8. http://www.partnersinrhyme.com/pir/PIRsfx.shtml, dne 15. 12. 2010
9. http://www.soundsnap.com/, dne 15. 12. 2010
10. http://amazingsounds.iespana.es/en/, dne 15 .12. 2010
11. http://www.pachd.com/sounds.html, dne 15 .12. 2010
12. http://www.mediacollege.com/downloads/sound-effects/, dne 15. 12. 2010

6 SLOVAR TUJIH BESED

Actions – akcije

ActionScript – skriptni jezik programa Adobe Flash

Align – poravnava

Bitmap – bitna slika

Button – grafični simbol Button

Classic Tween – klasična animacija gibanja

Collision Detection – preverjanje trka

Color – barva

Compiler Errors – napake prevajalnika

Drawing Object – risalni objekt

Dynamic Text – dinamični tekst

Fill – polnilo

Frame – okvir

Frame by Frame – okvir za okvirjem

Frames per Second – število okvirjev na sekundo

Graphic – grafični simbol Graphic

Graphic Symbol – grafični simbol

Height – višina

Hertz – herc

High – visoka kakovost

Info – informacije

Layer – plast

Library – knjižnica

Low – nizka kakovost

Medium – srednja kakovost

Morphing – učinek pri animaciji, kjer se slika skozi čas pretvori v drugo v obliki preobrazbe

Motion Guide – pot animacije

Motion Tween – animacija gibanja

MovieClip – grafični simbol MovieClip

Output – izpis

Pixel – slikovna pika

Position – pozicija

Properties – lastnosti

Rotation – rotacija

Shape Tween – animacija oblike

Swatches – odtenki

Stroke – obroba

Timeline – časovna linija

Tools – orodja

Width – širina

7 PRILOGE

Poleg tega dokumenta prilagam DVD z aplikacijo Interaktivni prikaz pravilne vožnje v krožišču v Adobe Flash CS4. Vse datoteke, ki se nahajajo na DVD-ju, so prikazane v tabeli 7.

Ime datoteke	Vrsta datoteke	Razlaga
Diplomska naloga.pdf	Dokument PDF	Diplomska naloga v elektronski obliki
Krožišče.exe	Program	Izdelek v obliki programa (za Windows)
Krožišče.fla	Flash dokument	Odprta izvorna datoteka Flash
Krožišče.html	Dokument HTML	Povezava do izdelka v SWF formatu
Krožišče.swf	Shockwave datoteka	Izdelek v SWF formatu (za splet)

Tabela 7: Datoteke na priloženem DVD-ju

Vir: Lasten