UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Găsirea drumului optim și algoritmi de coliziune în contextul animației 3D

student

Bogdan-Marian Mihalachi

Sesiunea: *iulie, 2018*

Coordonator științific

Conf. Dr. Anca Vitcu

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Găsirea drumului optim și algoritmi de coliziune în contextul animației 3D

Bogdan-Marian Mihalachi

Sesiunea: *iulie, 2018*

Coordonator științific

Conf. Dr. Anca Vitcu

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …….. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Găsirea drumului optim și algoritmi de coliziune în contextul animației 3D*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent *Bogdan-Marian Mihalachi*

_____

(semnătura în original)

Cuprins

I. Introducere

Alegerea acestei teme de licență a fost în primul rând stimulată de dorința de a face ceva nou, de a intra într-un mediu de lucru mai puțin folosit în facultate în care să pot învăța lucruri noi, dar și să-mi îmbunătățesc tehnicile de programare dobândite până în acest moment. Având o oarecare pasiune pentru jocuri, mi-am îndreptat tema licenței către zona modelării și animațiilor 3D. Ideea temei a fost practic construită pe parcurs. Inițial știam că va trebui să-mi aleg un decor, un personaj, câteva obiecte care să fie modelate și animate în mediul 3D. Din mai multe idei puse pe foaie și după câteva zile de gândire, am rămas asupra ideii de a modela un căluț, un peisaj și câteva obiecte de decor. Pe măsură ce am modelat și gândindu-mă la animarea căluțului în realizarea unei sărituri, am descoperit posibilitatea creării unor obstacole (garduri, stâlpi, copaci, pietre, etc.) și a unor algoritmi care să detecteze proximitatea între căluț și piedici și să-l determine pe acesta să le ocolească. Tot de ce mai era nevoie era un traseu pe care căluțul să-l parcurgă, traseu pe care să fie plasate obstacolele create și necesar a fi evitate. Așa a luat naștere ideea modelării peisajului sub forma unui graf ale cărui muchii să fie drumurile posibile pe care căluțul le poate urma, iar nodurile intersecțiile dintre acestea, alegerea drumului optim fiind realizată de un algoritm din zona grafurilor. Puse cap la cap, a rezultat modelul și animațiile ce vor fi prezentate în secțiunea descrierii aplicației, având la bază algortimi de detecție a coliziunii și de găsire a drumului optim într-un graf orientat.

Algoritmii folosiți sunt destul de uzuali și de bază în ariile lor. Ceea ce conferă noutate aplicației este modalitatea în care acești algoritmi sunt combinați și simulați în cadrul unei scene 3D. Rezultatele acestora pot fi vizualizate concret dispunând de obiectele 3D care practic vor deveni datele de intrare pentru algoritmi.

Metodologia folosită în acest proiect urmărește îndeaproape îndeplinirea obiectivelor care să ne conducă la soluție. Ca surse de studiu în realizarea cercetării au fost folosite cărți ca suport de documentație, atât pentru partea de modelare și animație 3D, cât și pentru partea de algoritmică și tutoriale online, toate acestea fiind citate în bibliografie. Cercetarea întreprinsă a fost una descriptivă prin descrierea metodelor folosite în construcția soluției,

analitică prin analiza și evaluarea rezultatelor oferite de algoritmi și comparativă întrucât situații diferite, paralele sau identice dar din timpi diferiți au fost puse față în față și comparate în vederea testării corectitudinii informațiilor obținute. Mediile de lucru folosite pentru implementarea lucrării sunt Autodesk Maya 2017 și Eclipse Java Oxygen configurat cu interpretorul de Python PyDev, cele două fiind la rândul lor conectate pentru a putea fi posibilă transmiterea și rularea scripturilor Python din Eclipse în Maya.

Problema care trebuie rezolvată este următoarea: dându-se un traseu generat aleator format din puncte de oprire și drumuri care unesc aceste puncte și un căluț, se cere să se animeze căluțul astfel încât să parcurgă drumul care conține cele mai multe obstacole de evitat; în momentul în care acesta se află în fața unui obstacol, va trebui să decidă dacă poate sări peste impediment, iar în caz afirmativ să se formeze animația care să-i permită să facă acest lucru, altfel să ocolească obiectul pe lângă una din lateralele acestuia. Soluția oferită începe prin modelarea 3D a tuturor obiectelor necesare și punerea acestora împreună pentru a forma o scenă 3D. Traseul va avea în spate o structură de graf ce va fi desenat pe un plan 3D – harta suport pentru toate obiectele ce interacționează. Se formează apoi toate animațiile necesare pentru a aduce căluțul la viață – mers, alergat și sărit, folosind funcționalitățile puse la dispoziție de Maya. Căluțul se află inițial într-un punct sursă și dorește să ajungă într-un punct destinație, alegând drumul care îl solicită la cel mai mare efort – îi pune în cale cele mai multe obstacole de evitat. Acest lucru este realizat cu ajutorul algoritmului de programare dinamică din zona grafurilor care determină drumul de cost maxim într-un graf orientat și aciclic. Costul maxim în cazul nostru este reprezentat de numărul maxim de obstacole de evitat. Odată formată și animația de parcurgere a traseului, aceasta este activată și se verifică la fiecare pas dacă există obstacole în fața căluțului folosind algoritmii de detecție a proximității, iar în caz pozitiv, se aplică o nouă animație care mișcă corpul căluțului pe deasupra sau pe lângă impediment.

Structura lucrării:

1. Animațiile 3D – detalii despre modelarea căluțului și realizarea animațiilor de mers, galopat și sărit; de asemenea, cuprinde un scurt studiu asupra mersului la cai

2. Detecția coliziunii – descriere a funcției care detectează apropierea („coliziunea la distanță”) între două obiecte și analiza condițiilor ce trebuie satisfăcute pentru a putea fi realizată evitarea obstacolelor cu succes
3. Reprezentarea și rolul grafului orientat și aciclic în Maya
    a. Găsirea drumului de cost maxim într-un graf orientat și aciclic – sortarea topologică a grafului, rezolvarea algoritmică a problemei folosind un algoritm de programare dinamică
    b. Generarea de grafuri orientate și aciclice – generare aleatoare astfel încât graful să fie valid (să nu aibă muchii duplicate sau muchii a căror capăt inițial să fie nodul destinație al grafului sau muchii a căror capăt final să fie nodul sursă al grafului)
    c. Desenarea grafului în Maya – crearea unei reprezentări vizuale a grafului în mediul 3D (graful va fi plasat pe un teren, nodurile acestuia vor fi reprezentate de puncte de localizare, muchiile reprezentate de linii poligonale, iar costul unei muchii reprezentat prin obstacolele plasate pe aceasta)
    d. Parcurgerea drumului – drumul optim conform condiției stabilite, determinat de algoritm, este traversat de către căluț de la nodul de start la nodul final

Problemele rezolvate în fiecare capitol se pun cap la cap pentru formarea animației finale. Animațiile ce constituie deplasarea căluțului sunt importate împreună cu acesta pe hartă și contribuie la naturalețea mișcărilor. La prima parcurgere a drumului se formează animația finală prin apelul la funcțiile de coliziune care ajută căluțul să aprecieze posibilitatea de a sări peste sau posibilitatea ocolirii acestora și să realizeze efectiv acțiunea.

II. Contribuții

Principalele contribuții în realizarea lucrării:

- realizarea modelelor de la zero pe bază de schițe și adăugarea elementelor necesare pentru a putea fi animate (joint-uri, controale, mânere)
- studiul mișcărilor unui cal pentru stabilirea pozițiilor picioarelor și a valorilor înregistrate în cadrul frame-urilor ce formează animația
- adaptarea algoritmilor de coliziune 2D între dreptunghiuri la algoritmi de coliziune 3D între paralelipipede dreptunghice
- găsirea unui set de condiții potrivite pentru testarea posibilității evitării unui obiect sărind peste acesta
- tratarea mai multor situații în ceea ce privește direcția și axa pe care se apropie călușul de obstacole
- realizarea funcției de generare aleatoare a unui traseu valid
- găsirea și transpunerea unei reprezentări a unui graf orientat și aciclic în mediul 3D
- unirea tututor părților în formarea animației finale de parcurgere a traseului optim cu evitarea obstacolelor

III. Structura aplicației

1. Animațiile 3D

Descrierea problemei

Obiectul animației noastre îl constituie un căluț care va trebui să parcurgă un traseu încărcat cu obstacole (garduri, stâlpi, bănci, copaci, etc.) pe care va trebui să le evite (în caz de este posibil, să sară peste ele, altfel să le ocolească) pentru a ajunge la obiectivul dorit. Se cere să se construiască animațiile care să permită căluțului deplasarea pe hartă și ocolirea impedimentelor de pe drum.

Descrierea soluției

Ca mediu de lucru pentru realizarea animațiilor 3D s-a folosit Autodesk Maya 2017 care ne-a pus la dispoziție librăria maya.cmds scrisă în limbajul Python. Aceasta ne-a oferit acces la comenzile MEL (Maya Embedded Language) pe care le-am folosit în scripturi Python pentru a modela obiectele din scena de lucru și pentru a prelua, modifica și lucra cu atributele acestora.

Până a ajunge la etapa în care să poată fi animat, căluțul a trecut printr-o serie de faze:

- modelarea – folosind tool-urile puse la dispoziție de Maya și imagini reper cu proiecții paralele ale căluțului (laterală, verticală, frontală), acesta a fost modelat pornind de la un singur cub prin extrudare, lucru la nivel de vârf, muchie și față
- texturarea – desfășurarea modelului în plan 2D folosind Maya UV Editor și potrivirea acestuia pe imaginea ce conține textura
- rigging-ul – crearea unui sistem organizat format dintr-o ierarhie de joint-uri (după cum se poate observa în Figura 1 de pe pagina următoare) ce formează scheletul și un set de controale, aplicat obiectului pentru a putea fi ușor și eficient animat

Având organizarea în ierarhie și fiecare joint acționând asupra unei secțiuni din model prin intermediul controalelor atașate, devine posibil lucrul la partea de animație. Pentru căluț au fost realizate trei animații: mers, alergat și sărit.

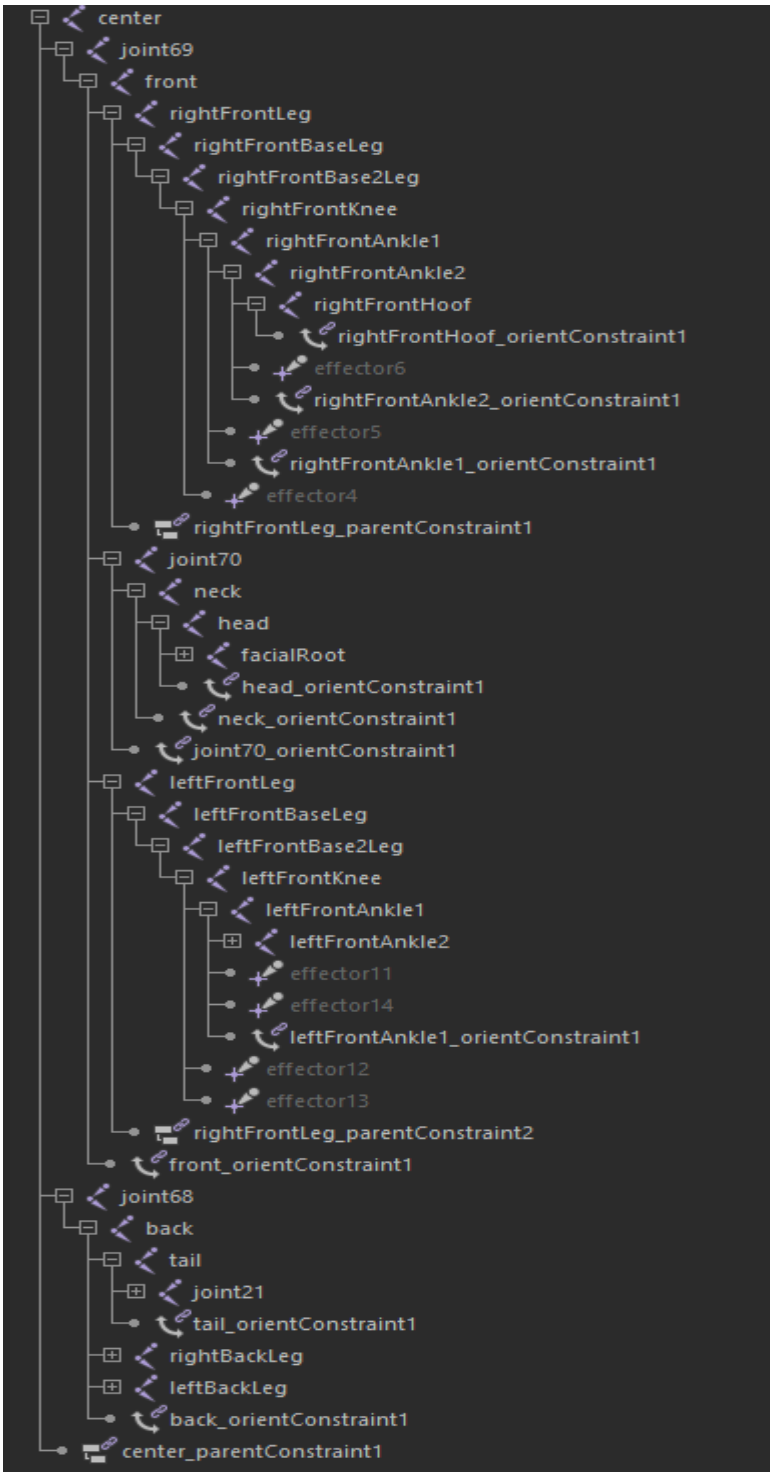

Figura 1: *Fragment din ierarhia de joint-uri corespunzătoare căluțului*

Scurt studiu asupra mersului calului

Corpurile animalelor sunt extrem de flexibile, motiv pentru care mișcările devin complexe. Corpul se întinde și se comprimă în mod constant, mișcarea transferându-se de la un picior la altul și într-o anumită ordine pentru a oferi stabilitate și echilibru.

Având căluțul ca referință, pattern-ul de mișcare pentru partea din spate este format din cinci poziții: cele două poziții de contact pentru fiecare picior în parte, atunci când acesta atinge cu vârful copitei pământul, poziția când își lasă greutatea pe piciorul posterior iar spatele este jos, poziția când spatele este sus și își ridică celălalt picior pregătindu-l pentru contact și poziția de legătură între ultimele două precizate. Pattern-ul pentru partea din față este aproximativ la fel cu cel pentru partea din spate, precizând faptul că picioarele se deplasează mai puțin în sus și în jos. Punându-le împreună, partea din spate a calului se află jos atunci când partea din față este sus și invers, iar pozițiile de contact și de legătură se regăsesc între acestea, după cum poate fi observat în Figura 2 de mai jos.

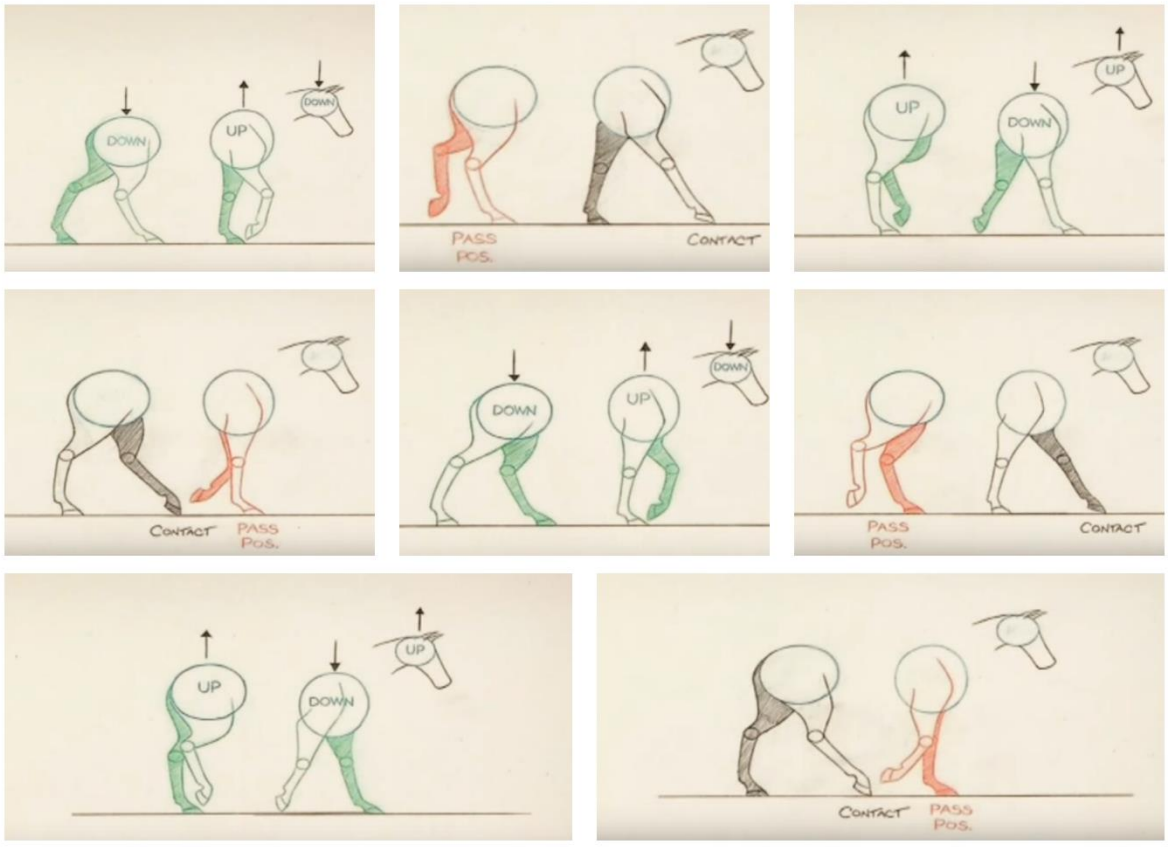

Figura 2: *Frame-urile ce constituie mersul calului;* [5]

În ceea ce privește joint-urile, pentru picioarele anterioare, genunchiul se deplasează în față pe măsură ce copita merge în spate și invers. Puterea în mișcare provine de la picioarele din spate. Căluțul își lasă greutatea pe picioarele posterioare și apoi se împinge în acestea, ceea ce contribuie la menținerea echilibrului.

Studii asemănătoare există și pentru alergat și sărit. Frame-urile stabilite pentru animația de alergat pot fi observate în Figura 3 de mai jos.

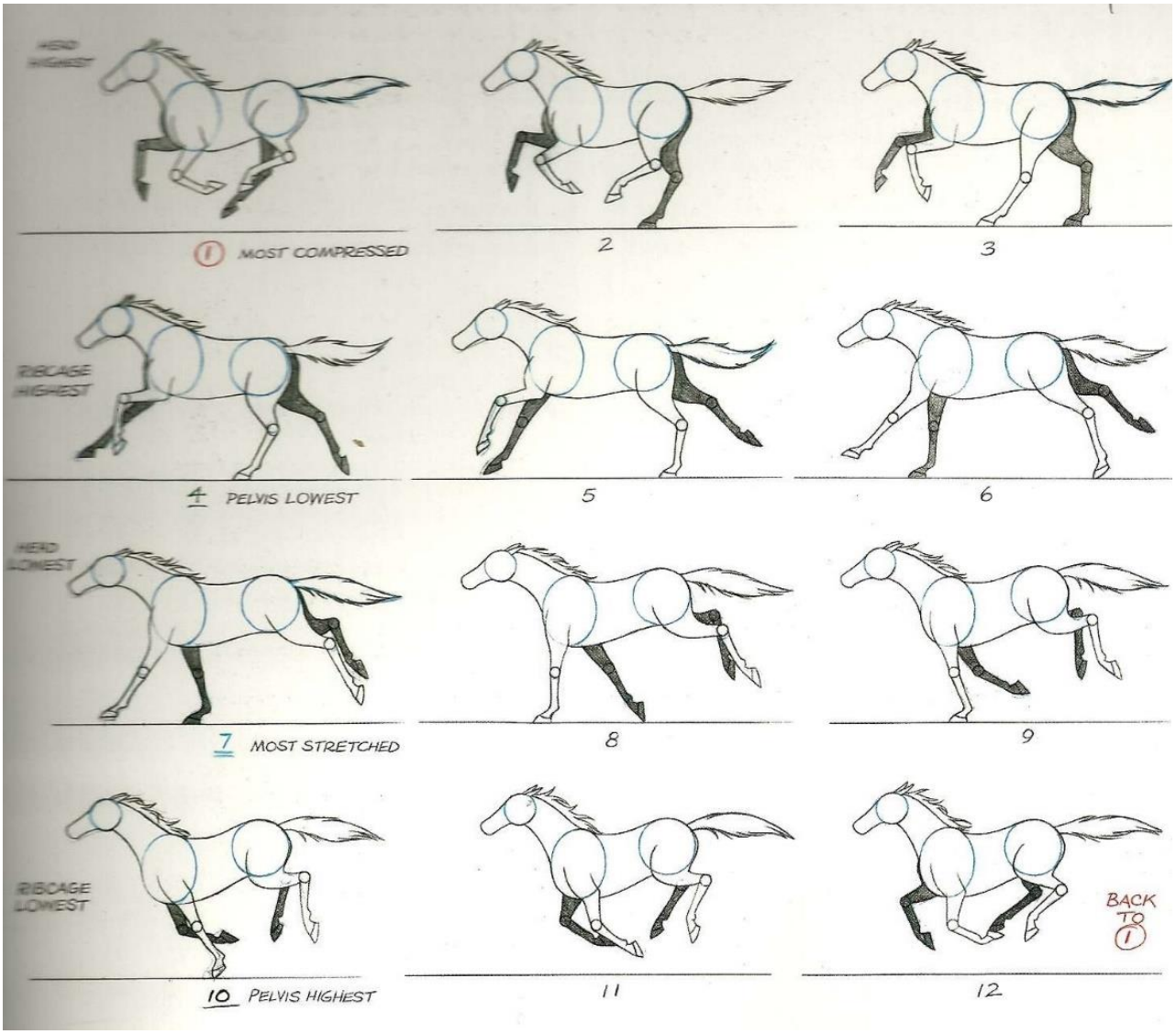

Figura 3: *Pozițiile picioarelor în cadrul frame-urilor ce constituie animația de alergare;* [26]

Descrierea soluției

Un frame reprezintă un moment de timp în care atributele unui obiect (translațiile, rotațiile și scalările în jurul axelor X, Y și Z) au anumite valori setate. Între două frame-uri succesive, valorile atributelor fac trecerea de la valorile setate în primul frame la valorile setate în al doilea frame prin intermediul unei funcții liniare, spline, platou, etc.

Pentru realizarea animațiilor căluțului, au fost înregistrate valori pentru translațiile și rotațiile picioarelor și genunchilor în cazul mersului și al alergatului și picioarelor, genunchilor și întregului corp pentru sărituri. Apoi, acestea au fost setate în cadrul frame-urilor succesive prin intermediul comenzii *cmds.setKeyframe*, putând reda astfel animația prin parcurgerea frame-urilor. Un exemplu de cod figurează în Tabelul 1 de mai jos.

```
# FRAME 5 HORSE JUMP
translationsRotationsRightFrontLeg = [-0.275, 1.798, 0, 0, 0, -88.944]
translationsRotationsRightFrontKnee = [1.806, 0, -0.407, 0, 0, 0]
translationsRotationsLeftFrontLeg = [2.744, 0.002, 0, 0, 0, -0.578]
translationsRotationsLeftFrontKnee = [-2.198, -1.676, 0, 0, 0, 0]
translationsRotationsRightBackLeg = [-0.878, 0.935, 0, 0, 0, -86.225]
translationsRotationsRightBackKnee = [1.806, 0, -0.407, 0, 0, 0]
translationsRotationsLeftBackLeg = [3.662, 0.543, 0, 0, 0, 19.69]
translationsRotationsLeftBackKnee = [-5.528, 0, 0, 0, 0, 0]
translationsRotationsBody = [0, 0, 0, -0.084, 0.137, 4.253]
.....
cmds.setKeyframe(control + '.translateX', value=TR[0], time=frame,
inTangentType="linear", outTangentType="linear")
cmds.setKeyframe(control + '.translateY', value=TR[1], time=frame,
inTangentType="linear", outTangentType="linear")
cmds.setKeyframe(control + '.translateZ', value=TR[2], time=frame,
inTangentType="linear", outTangentType="linear")
```

Tabel 1: *Înregistrarea valorilor în vectori de forma [object.translateX, object.translateY, object.translateZ, object.rotateX, object.rotateY, object.rotateZ] pentru picioarele, genunchii și corpul căluțului, pentru al cincilea frame din cadrul animației de săritură; setarea valorilor translațiilor pentru obiectul control în cadrul frame-ului frame folosind comanda setKeyframe cu tangente liniare*

2. Algoritmi de coliziune

Descrierea problemei

Se dau două paralelipipede dreptunghice care nu au aplicate rotații asupra lor și se cere să se creeze o funcție care să verifice dacă acestea se află în coliziune sau nu. Folosind acest rezultat, să se extindă funcția pentru testare a „coliziunii la distanță", criteriu ce intră în setul de condiții pe care trebuie să le îndeplinească primul paralelipiped pentru a putea sări peste al doilea. Să se formeze apoi animațiile prin care un obiect sare sau ocolește un alt obiect.

Descrierea soluției

```
def rangeIntersect(min1, max1, min2, max2):
    return max(min1, max1) >= min(min2, max2) and min(min1, max1) <=
max(min2, max2)
```

Tabel 2: *Funcție care verifică dacă două intervale se suprapun sau nu*

Pentru detecția coliziunii între două paralelipipede este nevoie de o funcție care să testeze suprapunerea a două intervale în planul 2D. O astfel de funcție este cea din Tabelul 2 care primește ca parametri capetele celor două intervale și returnează *True* sau *False* după cum unul dintre capetele unui interval se găsește în interiorul celuilalt interval sau nu. Întrucât intervalele pot fi date în orice ordine, testele se fac folosind minimul și maximul dintre capete.

Având această funcție, două paralelipipede dreptunghice se află în coliziune atunci când intervalele care formează lățimea, înălțimea și respectiv adâncimea acestora se suprapun. Având de exemplu colțul din dreapta sus și din față al paralelipipedului dreptunghic, intervalele vor avea ca prim capăt chiar coordonata (translația) *x, y* și respectiv *z* a colțului, iar drept capăt secund aceeași valoare la care se scade lățimea, înălțimea și respectiv adâncimea obiectului. Pentru a extinde funcția la testarea „coliziunii la distanță" (presupunem că avem două obiecte, dintre care al doilea static; spunem că cele două obiecte se află în coliziune la distanță dacă translatând primul obiect cu o anumită valoarea pe direcția și axa corespunzătoare, atunci cele două obiecte se află în coliziune), tot ce trebuie

făcut este translația primului interval cu o valoare stabilită pe direcția și axa din care se apropie obiectul (în cazul nostru, obiectul se apropie fie pe axa X de la „-” la „+”, fie pe axa X de la „+” la „-”, fie pe axa Z de la „-” la „+”, fie pe axa Z de la „+” la „-”), iar apoi testarea suprapunerii noului interval cu cel de-al doilea interval. În Tabelul 3 pot fi observate condițiile necesare pentru a putea spune că două obiecte se află în coliziune la distanță, unde *rightTopFront1* și *rightTopFront2* reprezintă vectori ce conțin coordonatele vârfurilor din dreapta sus și din față ale obiectelor, *direction* reprezintă direcția din care se apropie primul obiect, *xDistance* distanța pe axa X care, parcursă de primul obiect, l-ar introduce în coliziune cu cel de-al doilea obiect, iar *width*, *height* și *depth* dimensiunile obiectelor. În Figura 4 se regăsește o reprezentare grafică a formulelor de mai sus.

```
return rangeIntersect(rightTopFront1[0] + direction * xDistance,
                      rightTopFront1[0]  +  direction  *  xDistance  -
direction     * width1,
                      rightTopFront2[0],
                      rightTopFront2[0] - direction * width2) and \
       rangeIntersect(rightTopFront1[1],
                      rightTopFront1[1] - height1,
                      rightTopFront2[1],
                      rightTopFront2[1] - height2) and \
       rangeIntersect(rightTopFront1[2] + direction * zDistance,
                      rightTopFront1[2]  +  direction  *  zDistance  -
direction * depth1,
                      rightTopFront2[2],
                      rightTopFront2[2] - direction * depth2)
```

Tabel 3: *Testarea coliziunii la distanță între două obiecte atunci când primul se deplasează către al doilea pe axa X*

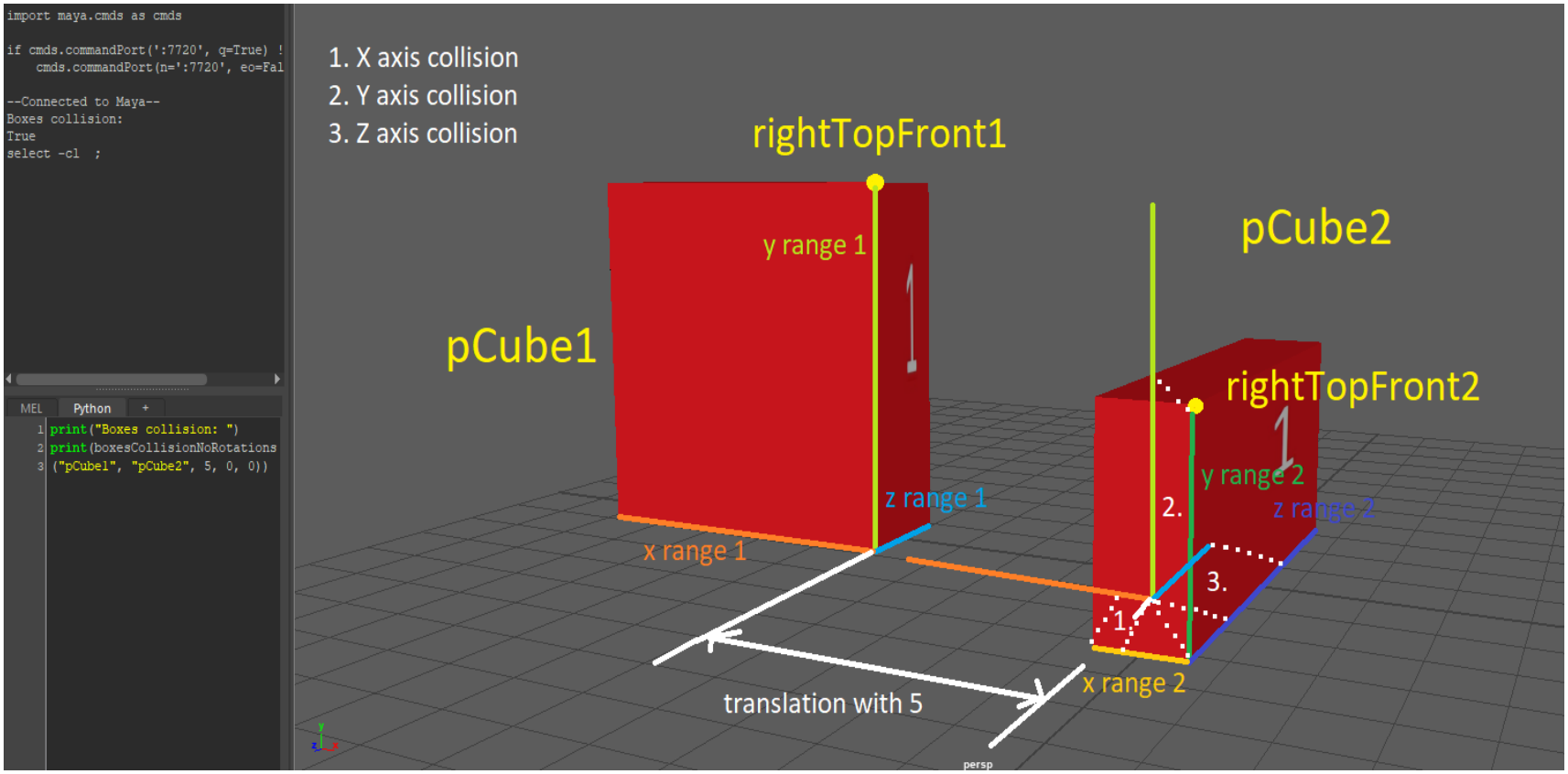

Figura 4: *Reprezentare grafică a coliziunii la distanță între două paralelipipede dreptunghice*

Condițiile ce trebuie satisfăcute pentru ca primul paralelipiped dreptunghic să poată sări peste cel de-al doilea:

- să aibă loc coliziunea primului paralelipiped, deplasat cu *distanceSafeToJump[1]* pe axa corespunzătoare, față de cel de-al doilea paralelipiped, dar să nu aibă loc în cazul deplasării cu *distanceSafeToJump[0]*; *distanceSafeToJump* este o variabilă globală declarată în program ce conține practic limitele între care trebuie să se găsească primul paralelipiped la distanță față de al doilea pentru a putea realiza o săritură fără impact

```
return boxesCollisionNoRotations(box1, box2, distanceSafeToJumpOver[1] if axis == "X" else 0, distanceSafeToJumpOver[1] if axis == "Z" else 0, direction) and \
       not boxesCollisionNoRotations(box1, box2, distanceSafeToJumpOver[0] if axis == "X" else 0, distanceSafeToJumpOver[0] if axis == "Z" else 0, direction) and \
```

- diferența dintre înălțimea primului obiect și înălțimea celui de-al doilea obiect să fie mai mare decât o limită dată; în program această limită este stocată în variabila globală *maxHeightSafeToJumpOver*

```
rightTopFront1[1] - rightTopFront2[1] >= maxHeightSafeToJumpOver and \
```

- lățimea celui de-al doilea paralelipiped să fie mai mică decât o limită precizată; în program, această limită este stocată în variabila globală *maxWidthSafeToJumpOver*

```
width2 <= maxWidthSafeToJumpOver
```

În cazul în care una dintre condiții nu este îndeplinită și obiectele noastre peste care nu se poate sări având adâncimi destul de mici și fiind situate la o distanță apropiată de obiectul mișcător, primul paralelipiped va ocoli pe cel de-al doilea pe una dintre cele două căi posibile (sus sau jos, stânga sau dreapta în funcție de axa, X sau Z, pe care se găsește).

Animația de săritură a primului paralelipiped dreptunghic peste cel de-al doilea se realizează în cinci frame-uri ce formează un arc de cerc pe deasupra celui de-al doilea paralelipiped, din poziția primului paralelipiped până, simetric față de al doilea paralelipiped, de partea cealaltă a acestuia. Săritura se realizează de-a lungul axei (X sau Z) pe care obiectul parcurge traseul.

În Figura 5 de mai jos pot fi observate cele cinci frame-uri și transformările pe care le suferă primul obiect pe masură ce le parcurge. În cadrul frame-ului central, înălțimea maximă la care se ridică obiectul este dată de formula:

```
center1[1] + height2 + maxHeightSafeToJumpOver + 1
```

, unde *center1[1]* reprezintă coordonata *y* a centrului primului obiect, *height2* reprezintă înălțimea celui de-al doilea obiect, iar *maxHeightSafeToJumpOver* înălțimea maximă peste care poate sări obiectul. Această formulă asigură faptul că primul obiect nu va intra în coliziune cu cel de-al doilea pe parcursul săriturii. În cadrul frame-urilor secundare (2 și 4), înălțimea la care se ridică obiectul este valoarea formulei de mai sus înjumătățită. De asemenea, pe parcursul frame-urilor, obiectul care sare va trece prin următoarele rotații:

- 0 → 45 → 0 → -45 → 0 în jurul axei Oz dacă primul paralelipiped dreptunghic se apropie de al doilea pe axa Ox, pe direcția negativ → pozitiv
- 0 → -45 → 0 → 45 → 0 în jurul axei Oz dacă primul paralelipiped se apropie de al doilea pe axa Ox, pe direcția pozitiv → negativ
- 0 → -45 → 0 → 45 → 0 în jurul axei Ox dacă primul paralelipiped se apropie de al doilea pe axa Oz, pe direcția negativ → pozitiv
- 0 → 45 → 0 → -45 → 0 în jurul axei Ox dacă primul paralelipiped se apropie de al doilea pe axa Oz, pe direcția pozitiv → negativ

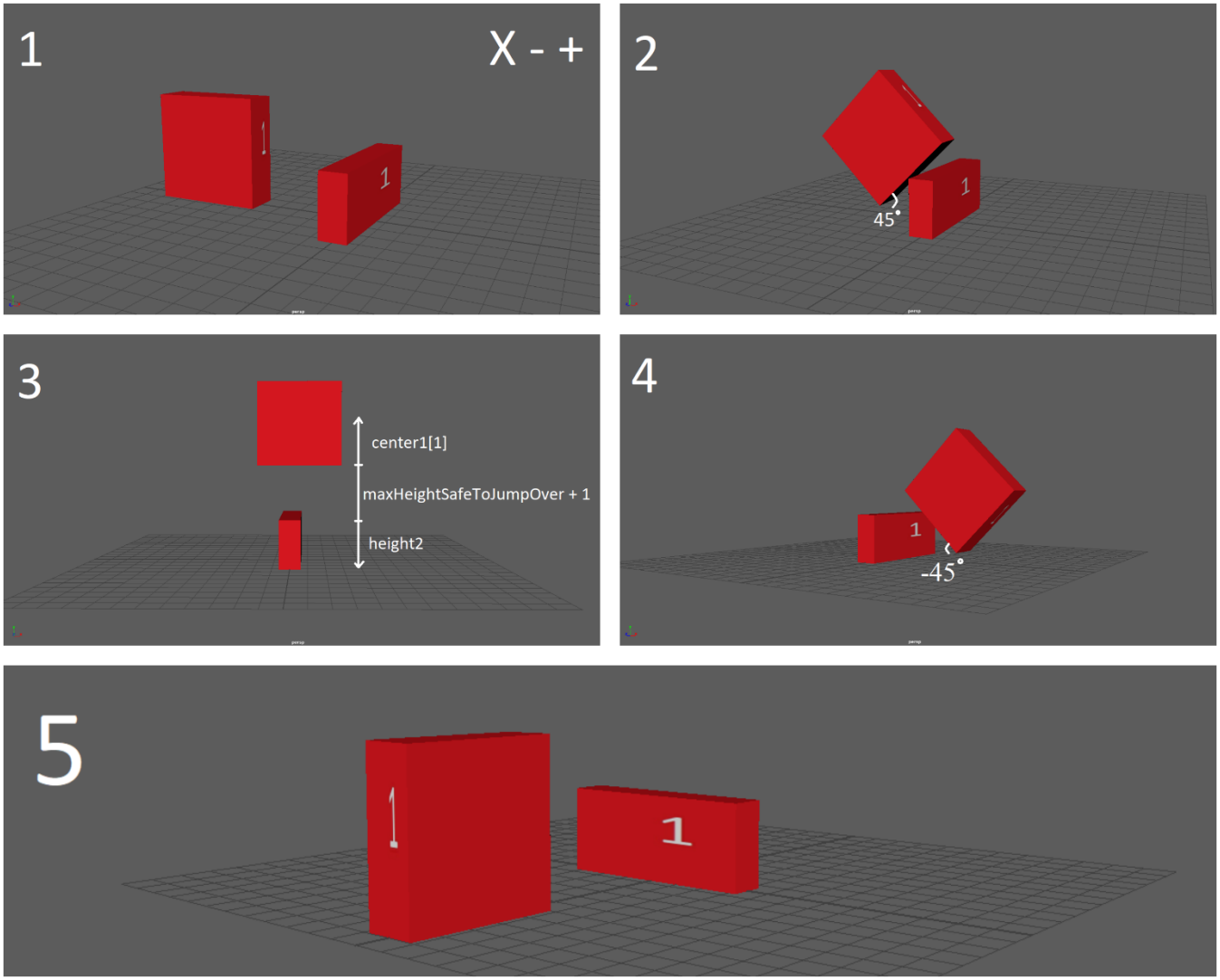

Figura 5: *Animație de săritură între două paralelipipede dreptunghice*

Animația prin care primul paralelipiped dreptunghic îl ocolește pe cel de-al doilea se realizează în cinci frame-uri ce formează un arc de cerc în jurul celui de-al doilea paralelipiped, din poziția primului până, simetric față de al doilea, de partea cealaltă a acestuia.

În Figura 6 de mai jos pot fi observate cele cinci frame-uri și transformările pe care le suferă primul obiect pentru a-l ocoli fără coliziune pe cel de-al doilea. În cadrul frame-ului central, distanța la care se abate primul obiect pentru a realiza ocolirea este egală cu lungimea în cazul axei X și respectiv adâncimea în cazul axei Z a obiectului țintă a ocolirii plus 1. De asemenea, asupra obiectului sunt aplicate următoarele rotații cu 0 → 45/-45 → 0 → -45/45

→ 0 în jurul axei Oy, semnele fiind stabilite în funcție de porțiunea pe unde se face ocolirea (sus sau jos/stânga sau dreapta).

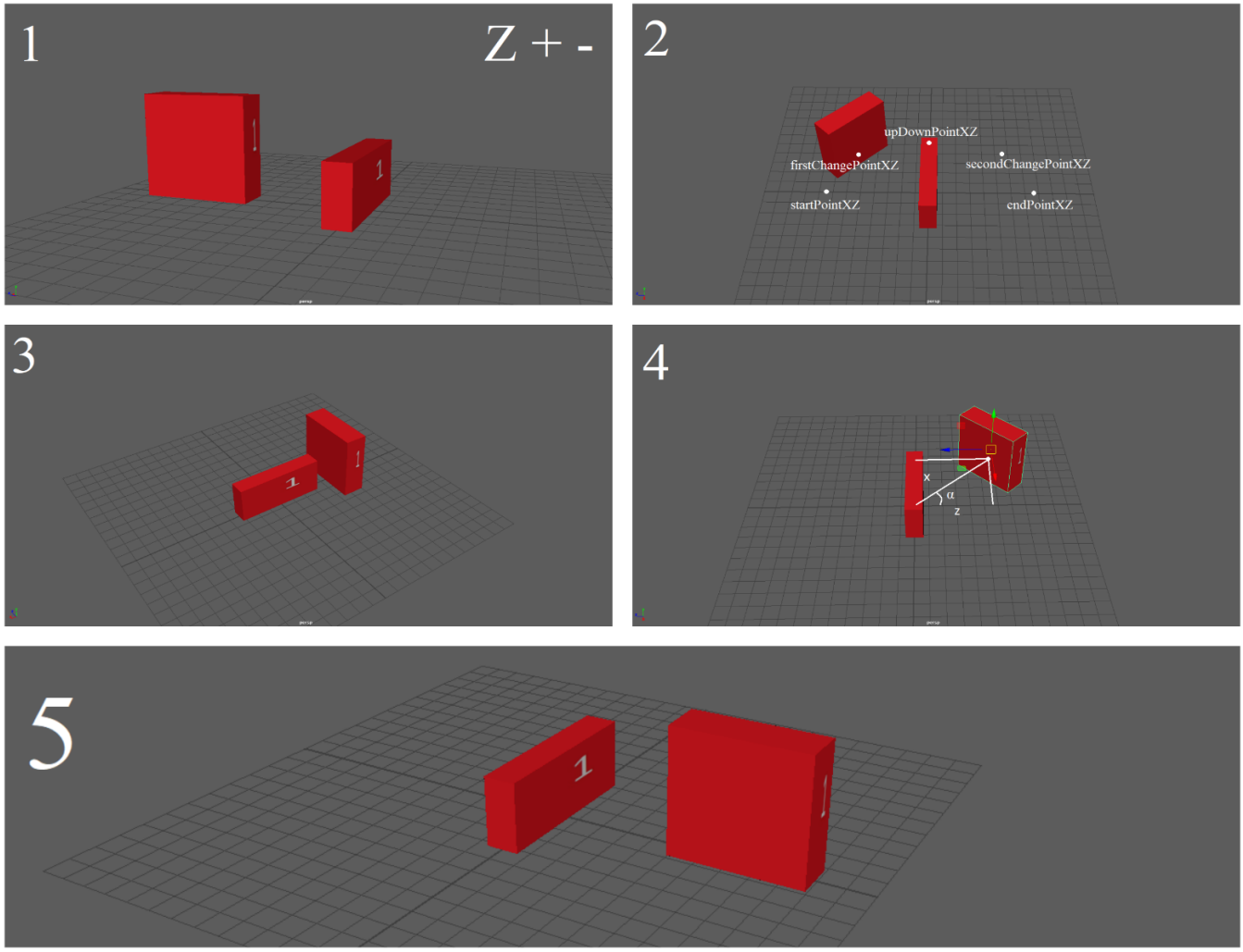

Figura 6: *Animație de ocolire a unui paralelipiped de către alt paralelipiped*

Pe parcursul frame-urilor, obiectul mișcător se găsește în ordine în următoarele puncte și având următoarele coordonate (acest exemplu se încadrează în cazul în care ocolirea are loc pe axa Z pe direcția pozitiv → negativ; similar, se deduc formulele și pentru celelalte trei cazuri):

- *startPointXZ* de coordonate *x, z* ale centrului obiectului mișcător
- *upDownPointXZ* de coordonate *x*-ul colțului de sus al obiectului static mai apropiat de obiectul mișcător și *z*-ul centrului obiectului static

- *endPointXZ,* simetricul punctului *startPointXZ* față de *upDownPointXZ*
- *firstChangePointXZ* de coordonate *x*sin α + center2X* și *z*cos α + center2Z,* unde *x* reprezintă distanța dintre centrul obiectului static și punctul maxim de ocolire, *z* reprezintă distanța între centrele celor două obiecte, *center2X* și *center2Z* coordonata *x* a centrului obiectului static și respectiv coordonata *z,* iar *α* unghiul format de dreapta ce unește centrul obiectului static cu punctul ce urmează a fi determinat, cu aza Oz
- *secondChangePointXZ,* simetricul punctului *firstChangePointXZ* față de *upDownPointXZ*

Aceste puncte se găsesc pe ecuația unei elipse cu centrul în centrul obiectului static și având drept raze distanța dintre centrele celor două obiecte și distanța de ocolire (*width2/depth2 + 1*) .

Trecerea de la coliziune între paralelipipede dreptunghice la coliziune între obiecte propriu-zise nu ridică multe dificultăți. Se va lucra cu bounding box-urile obiectelor – paralelipipede dreptunghice ce încorporează în mod optim obiectele și care pot fi obținute în program prin intermediul comenzii *cmds.xform(box, q=True, bb=True)* ce returnează un vector de forma *[xmin, ymin, zmin, xmax, ymax, zmax,]* din care se pot calcula dimensiunile – lățime, înălțime și adâncime. Centrele obiectelor au coordonatele egale cu translațiile lor la care se adună/scade eventual o anumită valoare ce reprezintă o deplasare în cazul în care obiectele importate nu au fost modelate în centrul scenei.

O altă modificare intervine în funcția care creează animația de săritură. Presupunând că avem un animăluț care sare peste diverse obstacole, în cadrul animației de săritură din contextul coliziunii, nu va mai fi nevoie a se aplica rotații corpului în jurul axelor Ox sau Oz întrucât acestea au fost încorporate în animația de săritură specifică animăluțului din contextul animației 3D. Singurele keyframe-uri ce vor mai trebui stabilite vor fi pentru translația corpului pe axa Oy.

3. Aplicații ale algoritmilor din teoria grafurilor în animația 3D

Găsirea drumului de cost maxim într-un graf orientat și aciclic (DAG – Directed Acyclic Graph)

Descrierea problemei

Fie un grid – graf orientat ce poate fi parcurs doar pe direcția est și sud, ale cărui muchii sunt etichetate cu valori ce reprezintă costul muchiei (ex.: număr de obiective turistice, număr de obstacole, etc.) și având un nod sursă, cel mai din nord-vest și un nod destinație, cel mai din sud-est. Se cere să se găsească un drum de cost maxim de la nodul sursă la nodul destinație, folosind un algoritm de programare dinamică.

Descrierea soluției

Problema rezolvată în lucrare este o generalizare a Manhattan Tourist Problem – găsirea drumului de cost maxim într-un DAG (Directed Acyclic Graph), care nu mai respectă neapărat structura de grid. Un DAG reprezintă un graf orientat ce nu are cicluri orientate, astfel încât acesta poate fi ordonat și parcurs într-o manieră liniară.

Ancorându-ne în decorul nostru, graful va reprezenta mapa, terenul pe care căluțul va merge, galopa și sări. Nodurile grafului vor fi puncte de oprire ale căluțului, iar muchiile acestuia drumurile pe care le poate alege căluțul ce se constituie într-o structură labirintică. Costul unei muchii este dat de numărul de obstacole de pe aceasta. Căluțul dorește să ajungă dintr-un nod sursă într-un nod destinație, parcurgând drumul ce conține cele mai multe obstacole pentru a-și spori rezistența și forța.

Această generalizare a problemei Manhattan poate fi rezolvată de asemenea printr-un algoritm de programare dinamică. Presupunând că avem un nod *v* cu gradul interior 3, având predecesorii $\{u_1, u_2, u_3\}$, costul maxim al drumului către v este:

$$c(v) = \max \begin{cases} c(u1) + costul\ muchiei\ de\ la\ u1\ la\ v \\ c(u2) + costul\ muchiei\ de\ la\ u2\ la\ v \\ c(u3) + costul\ muchiei\ de\ la\ u3\ la\ v \end{cases}$$

În general:

$$c(v) = \max_{u\ \epsilon\ Predecesori(v)} (c(u) + costul\ muchiei\ de\ la\ u\ la\ v) \quad (*)$$

Întrucât fiecare muchie participă într-o singură recurență, complexitatea timp a algoritmului este O(m), unde *m* reprezintă numărul de muchii din graf.

O problemă apare la ordinea în care trebuie parcurse nodurile în timpul calculului costului c. Ordinea este importantă întrucât, ajunși la analiza nodului *v*, valorile c(*u*) pentru toți predecesorii săi trebuie să fi fost deja calculate. De aceea, se calculează mai întâi o sortare topologică a grafului – o funcție injectivă ce asociază fiecărui nod un număr de ordine astfel încât pentru orice muchie, capătul din stânga să aibă ordinul mai mic decât capătul din dreapta. **Orice DAG admite o sortare topologică**. **Un digraf nu are circuite dacă și numai dacă admite o sortare topologică**.

```
class Graph:
    def __init__(self, filename="", n="", m="", source="", sink="",
edges=""):
……
……
def inDegree(self, vertex):
      return len([edge[0] for edge in self.edges if edge[1] == vertex])
def outDegree(self, vertex):
      return len([edge[1] for edge in self.edges if edge[0] == vertex])
# returns the vertices which enter the vertex given as parameter
# and the associated weights on the corresponding edges
def inNodesWithWeights(self, vertex):
```

```
      return [(edge[0], edge[2]) for edge in self.edges if edge[1] ==
vertex]
# returns the vertices that leave the vertex given as parameter
# and the associated weights on the corresponding edges
def outNodesWithWeights(self, vertex):
      return [(edge[1], edge[2]) for edge in self.edges if edge[0] ==
vertex]
# returns the topological order of the vertices or false if the graph
isn't acyclic
def TopologicalSorting(self):
……
……# return the longest path from source to sink using dynamic
programming
def Manhattan(self):
……
……
```

Tabel 4: *Structura clasei Graph folosită în program pentru a defini un graf*

Dispunem de clasa *Graph* având structura ca în Tabelul 4 de mai sus. Un graf conține următorii membri: *filename* – numele fișierului în care se găsesc *n* – numărul de noduri ale grafului (acestea sunt numerotate de la 1 la *n*), *m* – numărul de muchii ale grafului, *source* – nodul sursă al grafului, *sink* – nodul destinație al grafului și *edges* – muchiile grafului, perechi de forma *(nod sursă, nod destinație, cost)*. Procedura *TopologicalSorting* este cea care realizează sortarea topologică a grafului și returnează indicii nodurilor împreună cu numerele lor de ordine sau *False* în cazul în care graful nu admite o sortare topologică. Procedura *Manhattan* este cea care returnează drumul de cost maxim de la sursă la destinație format din perechi de noduri ce îi constituie muchiile, precum și costul maxim (în cazul nostru, numărul de obiecte evitate) sau *False* în cazul în care graful conține cicluri (nu admite o sortare topologică).

Considerăm acum un DAG exemplu (Figura 7 de mai jos) pe care vom aplica sortarea topologică (Figura 8 de mai jos), în urma căreia va rezulta graful sortat topologic (Figura 9 de mai jos) peste care vom aplica algoritmul de programare dinamică pentru aflarea drumului de cost maxim (Figura 10 de mai jos).

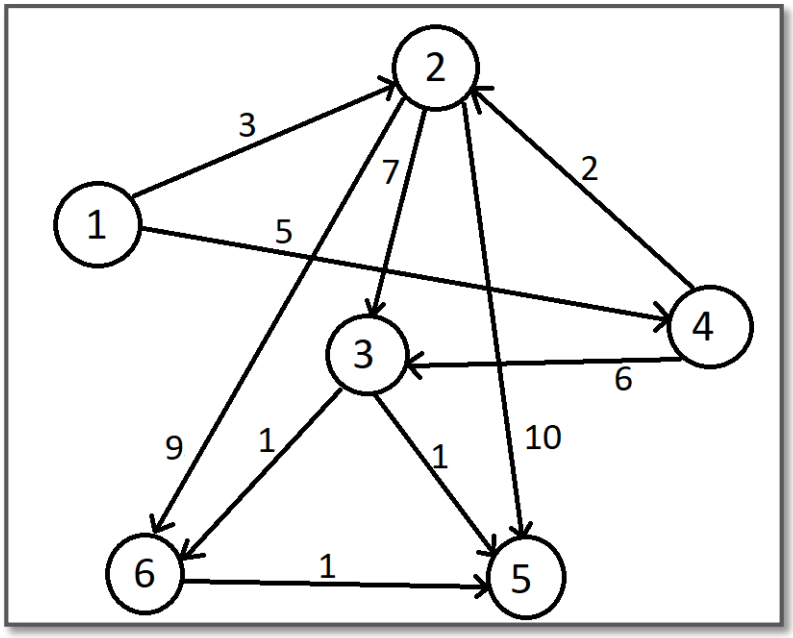

Figura 7: *Graf orientat și aciclic, nesortat topologic*

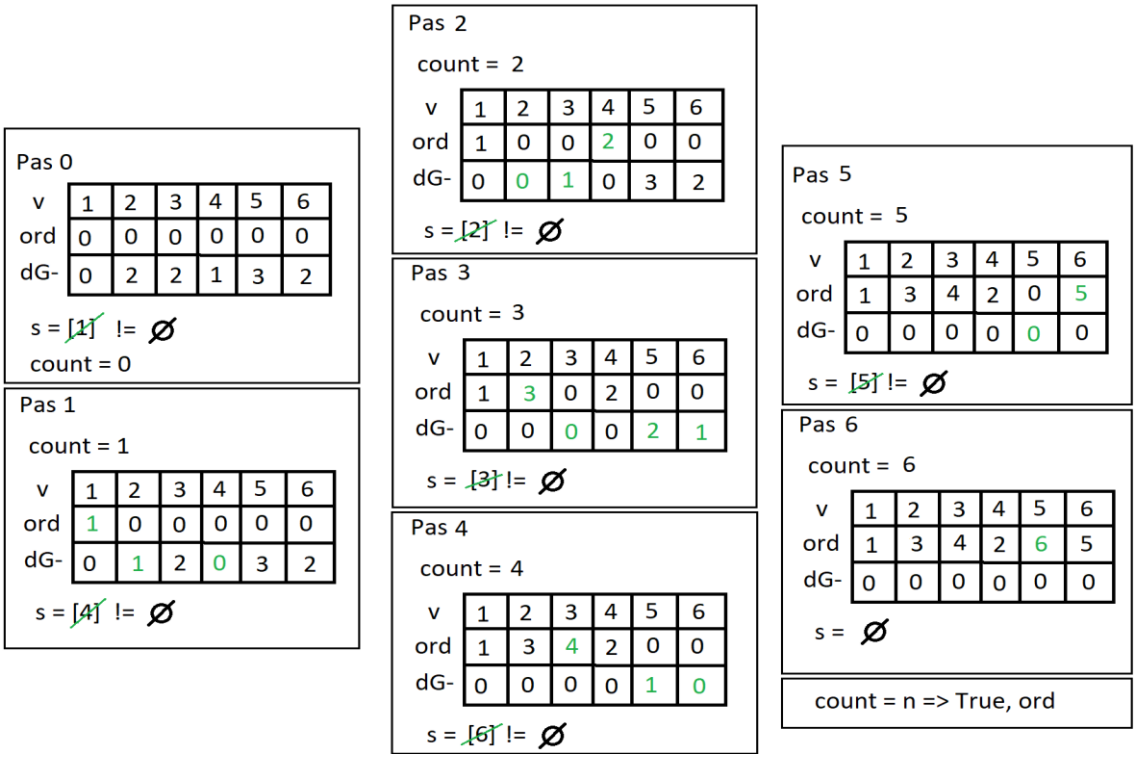

Figura 8: *Pașii algoritmului de sortare topologică pentru graful din Figura 7; v – nodurile, ord – numerele de ordine asociate nodurilor, dG- – gradele interioare ale nodurilor, count – numărul de ordine curent, s – stiva nodurilor curente de procesat*

Practic nodurile sunt alese în ordinea numărului de predecesori. Cât timp stiva nodurilor curente nu este vidă, se extrage nodul din vârful stivei, i se atribuie numărul de ordine curent, iar apoi se decrementează gradele interioare pentru toți succesorii săi. Dacă pe parcursul algoritmului stiva devine vidă – în urma eliminării ultimului nod procesat, nu mai există noduri cu gradul interior 0, acest lucru semnalează existența unui ciclu. În Tabelul 5 de mai jos se găsește secvența de cod care procesează nodurile. Algoritmul se termină cu succes atunci când toate nodurile au atribuite numere de ordine.

```
while len(s) != 0:
      v = s.pop(0)
      count += 1
      ord[v] = count
      for w in Successors(v):
           dG-[w] -= 1
           if dG-[w] == 0:
               s.append(w)
```

Tabel 5: *Sortare topologică*

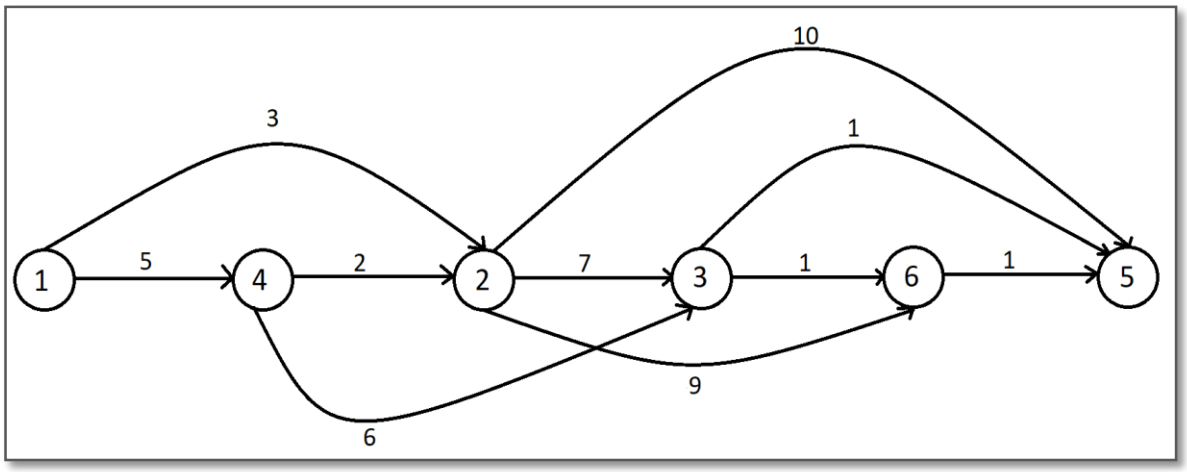

Figura 9: *Graful din Figura 7, sortat topologic*

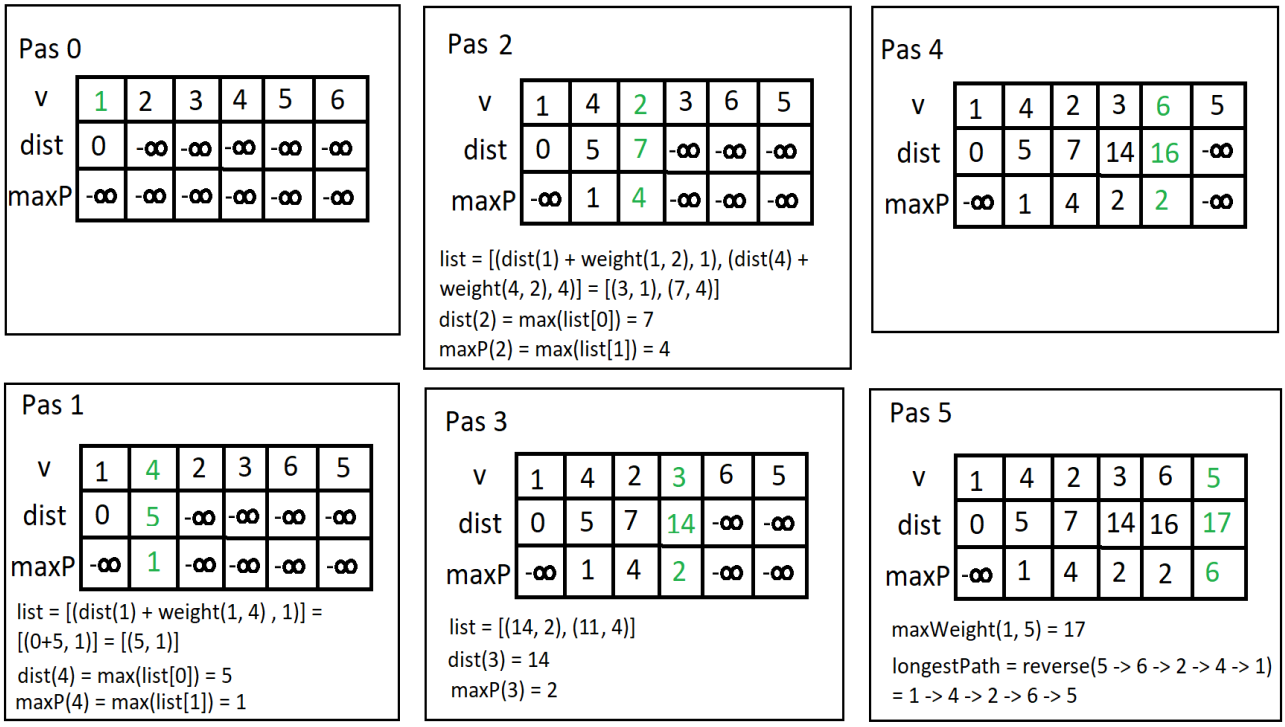

Figura 10: *Găsirea costului maxim și a drumului aferent de la sursă la destinație pentru graful din Figura 9*

Având nodurile ordonate de la sursă la destinație, la fiecare pas se determină costul maxim de la nodul sursă la nodul curent pe baza formulei (*), ceea ce este ilustrat în secvența de cod din Tabelul 6 de mai jos.

```
for v in [v[0] for v in orderTuples if v[1] > order[self.source]]:
      myList = [(distance[u[0]] + u[1], u[0]) for u in
PredecessorsPlusInWeights(v)]
      if len(myList) == 0:
          distance[v] = -float('inf')
          maxPredecessors[v] = -float('inf')
      else:
          distance[v] = maxListOfTuplesOrderByFirstElement(myList)[0]
          maxPredecessors[v] =
maxListOfTuplesOrderByFirstElement(myList)[1]
```

Tabel 6: *Algoritmul de programare dinamică ce completează vectorul de costuri; maxListOfTuplesOrderByFirstElement – returnează elementul maxim al unei liste formată din tuple de câte două elemente, sortarea făcându-se crescător după primul element al tuplelor (int)*

Pe parcursul algoritmului a fost reținut și predecesorul care aduce costul maxim pe muchia aferentă pentru fiecare nod. Acest lucru a fost folosit pentru reconstituirea drumului ce va fi traversat, parcurgând predecesorii care au adus costul maxim în ordine inversă.

Generarea și reprezentarea de DAG-uri în Maya

Descrierea problemei

Se cere să se genereze în manieră aleatoare un graf orientat și aciclic valid.

Descrierea soluției

Funcția de generare de DAG-uri primește ca parametri:

- *filename* – numele fișierului în care vor fi stocate datele despre graf; fișierul va fi creat în directorul curent
- *nRange* – intervalul în care poate varia numărul de noduri
- *mRange* – intervalul în care poate varia numărul de muchii, doar că inclusiv cele invalide (muchii duplicate), care vor fi eliminate printr-o funcție de normalizare; practic, numărul real de muchii se va încadra într-un interval puțin mai mic decât acest parametru
- *weightsRange* – intervalul în care poate varia numărul de obstacole ce se găsesc pe o muchie – secțiune de drum; limita inferioară a acestui interval poate fi cel puțin 0

Se alege un număr întreg aleator între limitele date de *nRange* ce va reprezenta numărul de noduri ale grafului și un număr întreg aleator între limitele date de *mRange* ce va reprezenta o limită superioară pentru numărul de muchii. Nodurile vor fi numerotate de la 1 la numărul lor. Se alege o etichetă pentru nodul sursă și o etichetă diferită pentru nodul destinație. Se generează apoi muchiile, alegând un număr de obstacole aleator, o etichetă pentru nodul inițial diferită de eticheta nodului destinație, o etichetă pentru nodul final diferită de eticheta nodului inițial și de eticheta nodului sursă și verificând dacă nu cumva prin adăugarea muchiei apare un ciclu în graf. Dacă se formează un ciclu, se alege aleator o

altă muchie. O problemă apare întrucât pot exista muchii duplicate. Pentru aceasta am folosit funcția *normalizeList* care primește lista de muchii și practic le unește pe cele duplicate, formând o singură muchie având costul egal cu suma costurilor muchiilor duplicate. Numărul real de muchii va fi egal cu dimensiunea listei, iar apoi, având toate informațiile necesare, graful va fi scris în fișierul dat ca parametru.

Ca o concluzie până aici, călușul va parcurge drumul cu cele mai multe obstacole, determinat pe baza generalizării algoritmului Manhattan, pe o hartă ce reprezintă un graf orientat și aciclic generat aleator și sortat topologic.

Reprezentarea grafului în Maya

Descrierea problemei

Se cere să se reprezinte în mediul 3D graful generat împreună cu terenul, muchiile și nodurile acestuia și obstacolele de pe traseu.

Descrierea soluției

```
planeDimension = 100 * (graph.n - 1)
plane = cmds.polyPlane(n="ground", sx=100, sy=100, sh=1, sw=1,
h=planeDimension, w=planeDimension)
```

Folosind comanda *cmds.polyPlane*, se creează suprafața pătratică peste care va fi reprezentat graful. Aceasta va fi denumită „ground”, va avea înălțimea și lățimea egală cu 100 * (numărul de noduri - 1), va avea 100 subdiviziuni pe direcțiile X și Y și i se va aplica o textură inițială de iarbă.

Vor fi desenate apoi nodurile grafului, reprezentate de space locators – puncte de localizare, folosind comanda *cmds.spaceLocator*. Acestea vor fi plasate pe axa X de la poziția 0 către infinit, fiind poziționate la distanța de 100 de unități între ei. Nodurile se vor afla în ordinea dată de sortarea topologică.

Muchiile vor fi reprezentate de curbe, vor fi desenate cu comanda *cmds.curve* și se împart în două categorii:

- linii drepte – trasate între două noduri succesive; acestea sunt reținute prin perechi de forma *((etichetă nod inițial, etichetă nod final, număr de obstacole), [(x-ul capătului inițial, 0, 0), (x-ul capătului final, 0, 0)])*
- linii poligonale dreptunghiulare – trasate între noduri aflate la cel puțin 200 de unități distanță (noduri care nu mai sunt consecutive din punct de vedere al sortării topologice); acestea sunt reținute prin perechi de forma *((etichetă nod inițial, etichetă nod final, număr de obstacole), [(x-ul capătului inițial, 0, 0), (x-ul capătului inițial, 0, h), (x-ul capătului final, 0, h), (x-ul capătului final, 0, 0)])*, unde *h* reprezintă înălțimea la care se alungește muchia în sus sau în jos; această înălțime este aleasă aleator din mulțimea $\{((planeHeight \div 2) \div graph.n) \times i \mid i \in [-graph.n,\ graph.n] \cap (Z - \{0\})\}$

Căluțul este importat în decor și plasat în locator-ul sursă al grafului având orientarea inițială pe direcția primei muchii pe care trebuie să o parcurgă. Momentan animațiile acestuia de mers sau galopat se derulează pe loc. În secțiunea următoare este descris modul în care a fost construită animația de parcurgere a drumului optim.

Prin intermediul funcției *placeObstaclesOnMap*, pe fiecare muchie se plasează obstacolele asociate acesteia. Obiectele sunt importate și plasate random astfel încât să nu fie prea aproape de colțurile liniei drepte/poligonale și să aibă o oarecare distanță între ele. În variabila globală *listOfObjectNames* se vor stoca numele tuturor obstacolelor importate, iar în variabila globală *avoidedObjects* câte un indicator pentru fiecare obiect care ne spune dacă acesta a fost ocolit deja sau nu. Acestea vor fi folosite în momentul în care se va combina animația de parcurgere a drumului cu animațiile de săritură și ocolire din zona coliziunii.

Un exemplu de DAG reprezentat în Maya poate fi observat în Figura 11 de mai jos.

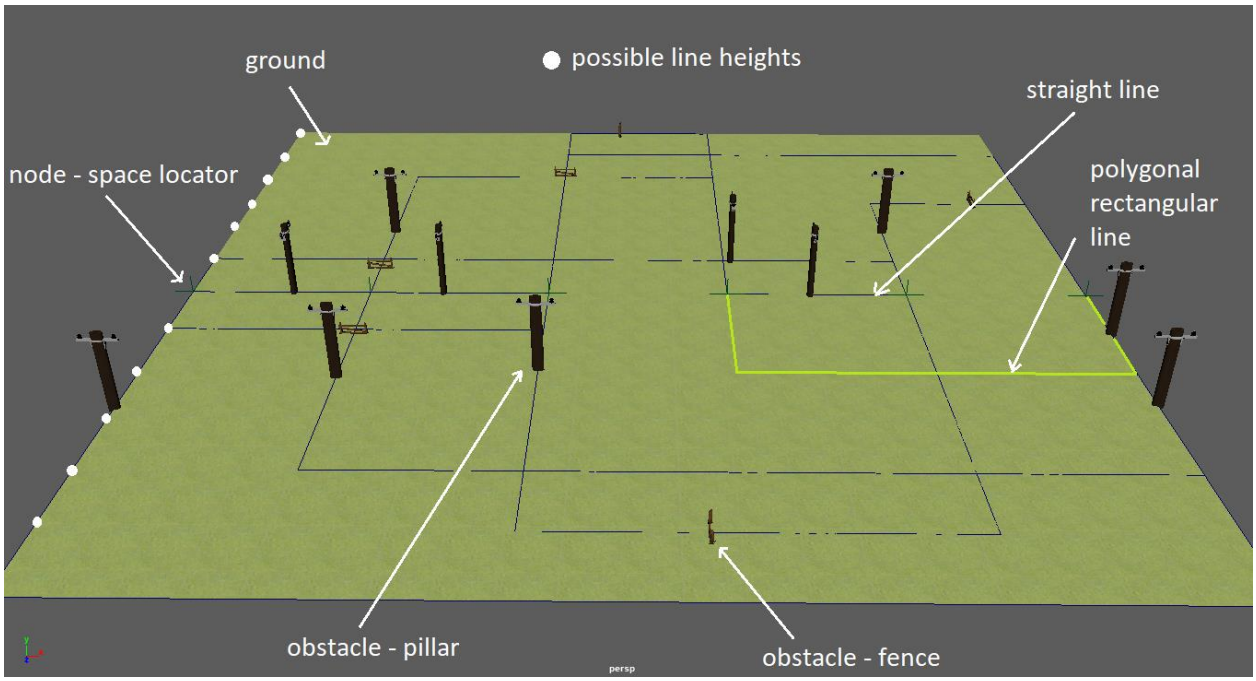

Figura 11: *DAG reprezentat în Maya împreună cu toate elementele sale*

Formarea animației de parcurgere a grafului și legătura cu algoritmii de coliziune

Descrierea problemei

Pe baza rezultatelor oferite de generalizarea algoritmului Manhattan pentru grafuri orientate și aciclice și algoritmilor de coliziune, se cere să se formeze animația 3D prin care călușul parcurge drumul cu cele mai multe obstacole și le evită pe măsură ce i se ivesc pe traseu.

Descrierea soluției

Dispunem de căluț ale cărui picioare sunt animate conturându-i mersul și alergatul. Inițial acesta merge, iar după un număr de frame-uri prestabilit acesta începe să alerge. Doar că momentan toate acestea se realizează pe loc. Ce mai trebuie realizat este animația mișcării corpului care deplasează călușul de-a lungul muchiilor drumului de cost maxim și animația care ridică corpul în sus, combinată cu animația de săritură a călușului, atunci când acesta are în față un obstacol peste care poate sări, sau care deplasează corpul de-a lungul unei

elipse ce ocolește un obstacol prea înalt sau poziționat incorect pentru a se putea sări peste el.

În cazul deplasării căluțului de-a lungul drumului cu cele mai multe obstacole, numărul de frame-uri necesar pentru parcurgerea unei secțiuni de muchie este egal cu dimensiunea secțiunii împărțită la 3. Pentru o linie dreaptă, corpul căluțului este translatat pe axa X fie pe direcția negativ → pozitiv, fie invers. Pentru o linie poligonală dreptunghiulară, corpul căluțului trece prin trei translații: una pe aza Z, una pe axa X și încă una pe axa Z pe direcțiile corespunzătoare astfel încât să ajungă din nodul curent în nodul următor de pe drumul de cost maxim. Cu puțin înainte să ajungă la o intersecție între două secțiuni de muchie sau când trebuie să facă cale întoarsă într-un nod, intervine funcția *aroundTheCorner* care îi setează rotațiile corespunzătoare în jurul axei Y astfel încât să facă trecerea într-o manieră arcuită și naturală. Acest lucru se poate observa în Figura 12 de mai jos.

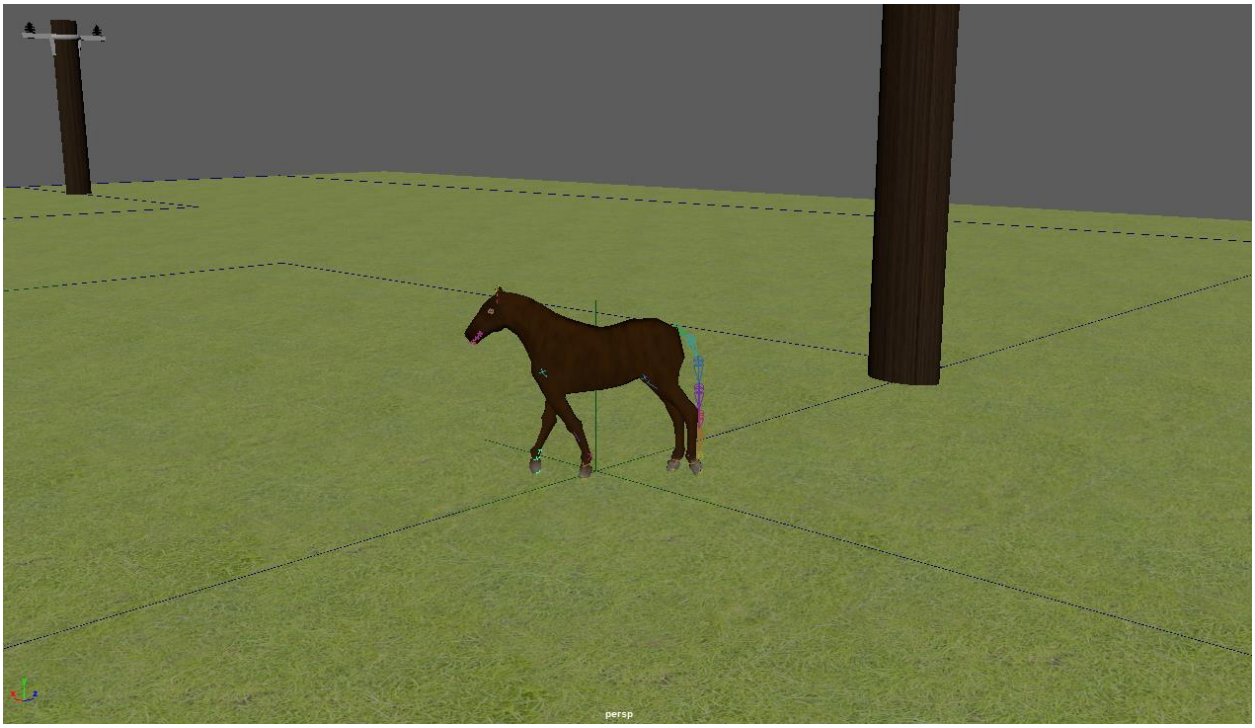

Figura 12: *Trecerea printr-o intersecție de muchii*

După finalizarea animației de parcurgere a traseului optim conform condiției impuse, urmează ultimul pas: utilizarea algoritmilor de coliziune pentru crearea animațiilor de săritură sau ocolire a obiectelor ce reprezintă obstacole în drumul urmat de căluț.

Se parcurg frame-urile în ordine, iar la fiecare dintre ele se iterează prin lista tuturor obiectelor plasate pe hartă. În funcție de orientarea căluțului (rotația în jurul axei Y dată de formula de mai jos),

```
cmds.xform(myObjectControlName, q=1, rotation=1, worldSpace=1)[1]
```

se determină axa și direcția folosite pentru testarea coliziunii la distanță între căluț și obiectele din fața sa. Dacă obiectul din față nu a fost evitat deja și sunt îndeplinite condițiile care să permită căluțului să-l evite printr-o săritură, se inserează în linia temporară o animație de săritură specifică căluțului și o animație de săritură din zona algoritmilor de coliziune care practic ridică căluțul la

```
if avoidedObjects[objectName] == 0 and
collision.checkIfCanJump(myObjectName, myObjectControlName, objectName,
axis, direction):
      horse_movements.horseJump(1, currentFrame - 5)
      collision.objectJump(myObjectName,           myObjectControlName,
objectName, axis, direction, currentFrame, currentFrame + 10)
      avoidedObjects[objectName] = 1
```

înălțimea prestabilită deasupra obstacolului. Dacă obiectul din față nu a fost evitat deja, există o condiție care nu este îndeplinită și care, astfel, nu permite căluțului să sară peste obstacol, dar totuși se află în coliziune la distanță cu acesta, atunci se activează animația de ocolire din zona algoritmilor de coliziune.

```
# Z from negative to positive
if avoidedObjects[objectName] == 0 and
not     collision.checkIfCanJump(myObjectName,     myObjectControlName,
objectName, axis, direction) and \
collision.boxesCollisionNoRotations(myObjectName,  myObjectControlName,
objectName, 0, collision.distanceSafeToJump[1], direction):
```

```
      collision.objectDodge(myObjectName,          myObjectControlName,
objectName, axis, direction, currentFrame, currentFrame + 15)
      avoidedObjects[objectName] = 1
```

O reprezentare se regăsește în Figurile 13 și 14 de mai jos.

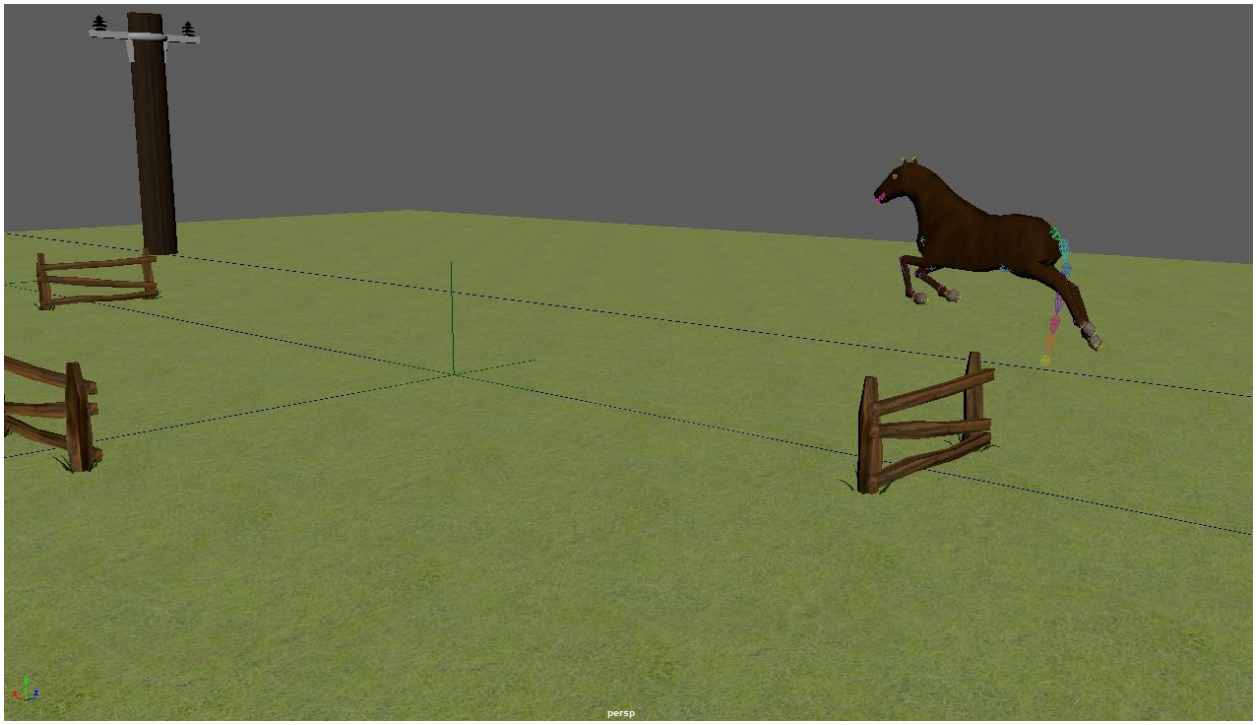

Figura 13: *Saltul peste un obstacol întâlnit în timpul parcurgerii*

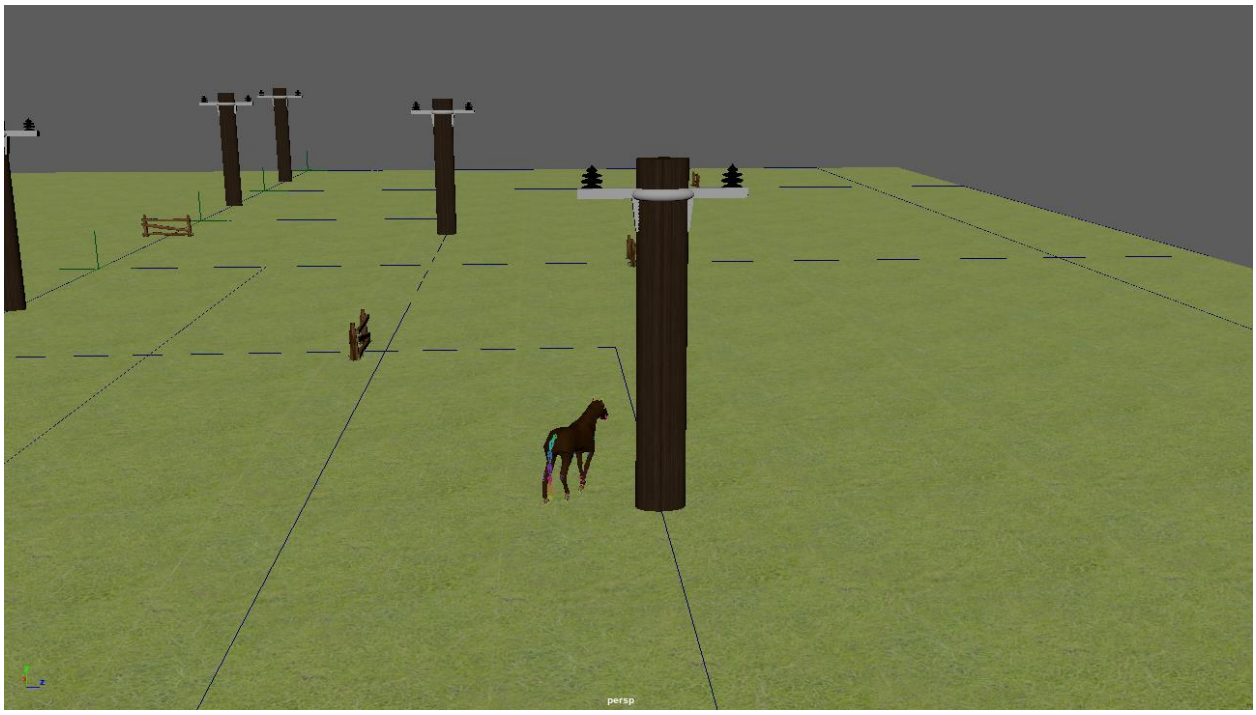

Figura 14: *Ocolirea unui obstacol întâlnit în timpul parcurgerii*

IV. Concluziile lucrării

Îmbinând tehnici de modelare și animație 3D cu metode de recunoaștere a coliziunii și a proximității între obiecte și cu algoritmică de grafuri, aplicația își conturează scopuri bine definite și lasă loc totodată de viitoare îmbunătățiri. În continuare voi prezenta câteva dintre direcțiile în care aplicația își găsește întrebuințare, la ce ar putea folosi.

Aplicația oferă în primul rând un suport pentru partea de învățare a algoritmilor. Abordarea temei într-o manieră care să includă reprezentare vizuală stimulează învățarea, multe noțiuni fiind fixate cu usurință atunci când rezultatele sunt reprezentate grafic.

Un alt domeniu în care lucrarea își găsește utilitate este industria jocurilor, plecând de la posibilitatea folosirii modelelor și animațiilor create în scene independente, până la importarea întregii animații de parcurgere a traseului cu evitarea obstacolelor ca un video introductiv, de exemplu pentru jucătorul căruia i se va da după controlul ca să completeze și el o rută similară. Aplicația ar putea fi extinsă pe o platformă de creat jocuri astfel încât călușul să fie controlabil, iar evitarea obstacolelor misiunea utilizatorului. Ar putea fi introdus și un factor competitiv, extinderea jocului la curse de cai fie între player și AI, fie între mai mulți playeri în rețea.

Partea de grafică și de animație poate fi de asemenea folosită în crearea de desene animate. Mișcările călușului sunt documentate și realizate astfel încât să reflecte mersul și alergatul real. Partea de programare pune bazele înțelegerii formulelor de grafică și cum se tratează o problemă în mediul de dezvoltare 3D.

În opinia mea, rezultatele obținute în lucrare sunt destul de precise, doar că se întind pe o gamă scurtă de opțiuni din cauza constrângerilor impuse de lucrul într-un mediu 3D. Mereu există loc de îmbunătățiri și voi prezenta în continuare potențiale direcții viitoare de dezvoltare a temei abordate, ce ar fi putut fi făcut în plus sau modificat pentru a ridica aplicația la un nivel mai înalt.

În ceea ce privește modelele și animațiile, toate acestea pot fi lucrate în continuare mai în detaliu. O problemă ce poate fi remediată este realizare cozii și a coamei căluțului într-o manieră care să împiedice alungirea exagerată a firelor de păr. Alte îmbunătățiri ce pot fi aduse modelului includ utilizarea tool-ului Maya ce crează un sistem de mușchi, detalierea texturii aplicate peste căluț, modelarea interiorului gurii (limbă și dinți). Partea de animație poate fi extinsă la părți ale corpului și la nivelul capului unde se pot realiza controale care să permită închiderea și deschiderea ochilor și a gurii. De asemenea, se poate extinde gama de obstacole cuprinzând obiecte de dimensiuni mai diverse pe care se pot întreprinde mai multe teste în ceea ce privește decizia căluțului asupra modalității de ocolire a acestora. Un exemplu ar fi un copac lateral drumului, dar care are o creangă mai joasă deasupra acestuia, astfel încât căluțul să urmeze un algoritm de coliziune care să formeze animația de aplecare pe sub creangă.

În ceea ce privește coliziunea, extinderea cazurilor de apropiere de obstacole ar aduce un plus, testatea proximității inclusiv pe diagonale sub diverse unghiuri, precum și tratarea situației în care obiectele implicate în coliziune au aplicate diverse rotații asupra acestora. O abordare mai eficientă și modernă ar fi utilizarea unor algoritmi de învățare automată sau rețele neuronale astfel încât căluțul să învețe singur din greșeli cum sa evite obstacolele.

Pe partea de grafuri și teren, o serie de lucruri ce ar putea fi schimbate pentru a conferi o calitate mai mare aplicației sunt prezentate în continuare.

Condiția de optimalitate a drumului ar putea ține cont de mai multe criterii. Pe lângă numărul de obstacole, ar putea exista și un număr de beneficii (de exemplu mere sau mici izvoare cu apă – pentru care ar fi necesare animații specifice prin care căluțul să mănânce merele sau să bea apă) plasate pe traseu și un indice care să indice gradul de abrupție sau denivelare a drumului. Pe baza unei formule care să combine aceste trei atribute ale drumului, s-ar obține un indice care să reprezinte noul cost și să reflecte câștigul obținut de căluț prin alegerea acelui drum. Căluțul ar parcurge astfel drumul cu cele mai puține obstacole, cele mai multe beneficii și cu un grad acceptabil de denivelare.

La desenarea grafului în Maya, ar putea fi luate în calcul mai multe modalități de trasare a muchiilor cum ar fi de exemplu sub formă de arce eliptice. Sau în cazul liniilor poligonale dreptunghice, acestea ar putea fi arcuite la capete.

Ca ultimă ajustare, ne putem gândi la realizarea de mici interfețe (plug-in-uri) în care să se poată selecta valori pentru parametrii funcțiilor și care să dispună de butoane prin care să se regleze funcționalitatea aplicației.

V. Bibliografie

Cărți

1. de Byl Penny (2017) – Holistic Game Development with Unity, CRC Press;
2. Jones C. Neil & Pevzner A. Pavel (2004) – An Introduction to Bioinformatics Algorithms, MIT Press;
3. McGugan Will (2007) – Beginning Game Development with Python and Pygame, Apress;
4. Mechtley Adam & Trowbridge Ryan (2012) – Maya Python For Games and Film, Elsevier;
5. Millington Ian (2006) – Artificial Intelligence for Games, Elsevier;
6. Palamar Todd (2016) – Mastering Autodesk Maya 2016, Autodesk Official Press & Sybex;
7. Park Edgar John (2005) – Understanding 3D Animation using Maya, Springer;
8. Watkins Adam (2011) – Creating Games with Unity and Maya, Focal Press.

Link-uri

9. http://www.3dbunk.com/index.php?p=t&t=set_eclipse
10. https://knowledge.autodesk.com/support/maya/getting-started/caas/simplecontent/content/maya-documentation.html
11. https://stackoverflow.com/questions/15268511/maya-python-place-image-on-a-plane
12. https://stackoverflow.com/questions/21344945/applying-a-file-as-a-texture-in-maya-python
13. https://www.youtube.com/watch?v=INQx-Lzs8mU
14. https://www.youtube.com/watch?v=BEw-2WBoNEA
15. https://www.youtube.com/watch?v=NZHzgXFKfuY

16. https://www.youtube.com/watch?v=NGh-Vh_NYO0
17. https://www.youtube.com/watch?v=B0aGFBo_Ki0
18. https://www.youtube.com/watch?v=CPCD2Qn7OqI
19. https://www.youtube.com/watch?v=vFvwyu_ZKfU
20. https://www.youtube.com/watch?v=vE7P5iFIbX8
21. https://www.youtube.com/watch?v=4ZAzUnZf7hk
22. https://www.youtube.com/watch?v=I2XUHDm4gEU
23. https://www.youtube.com/watch?v=tkVzxVus3Xw
24. https://www.youtube.com/watch?v=11pJ5GTdNYg
25. https://www.youtube.com/watch?v=a2iDVqnGOIY
26. https://www.google.com/search?q=horse+run+frames&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiUncj9j_LbAhUQMuwKHbulB8oQ_AUICigB&biw=1536&bih=750#imgrc=K7P4fQZY0vMGZM: