Seminár k záverečnej práci

Patrik Sakáč

3Ib, 2017 - 2018

Abstrakt. Práca sa zaoberá rastom stromov, kríkov v počítačových hrách pomocou L-systémov. Naším riešením je vytvoriť obrázok pomocou L-systémov, ktorý bude uchovávať dôležité informácie o raste daného druhu. Tento obrázok následne použiť a implementovať rast.

Kľúčové slová: L-systém, vrchol, mesh, shader

1 Úvod

V hrách je rast stromov, kríkov a iných rastlín bežným problémom, s ktorým sa vývojári stretávajú. Pri veľkom počte inštancií sa upúšťa od tejto funkcionality, pretože si vyžaduje veľké nároky na operačnú pamäť, procesor či grafickú kartu. Inou možnosťou je znižovanie nárokov na konečný grafický vzhľad.

Existuje veľa rôznych riešení, ktoré boli použité pri implementáciách v jednotlivých hrách avšak najznámejšími sú dve z nich, kde prvým je nahradenie celých objektov a druhým animácia jedného objektu pomocou kostí alebo pomocou vrcholov[1]. Spomenuté riešenia majú svoje výhody aj nevýhody. Výhodou je jednoduchá implementácia pre vývojárov. Nevýhodou je najmä zložitá výroba pre grafikov. Riešenie pomocou animovaných objektov, ktoré animátor vytvoril pomocou kostí je optimálnejšie ako pomocou vrcholov, ale je omnoho zložitejšie vytvoriť takúto animáciu. Animácie majú tú nevýhodu, že rovnaká animácia v inej fáze prehrávania sa považuje za novú a teda musí byť dostupná v operačnej pamäti. Pri väčšom počte animácii ľahko dosiahneme veľké nároky na operačnú pamäť. Pri riešení nahradzovaním objektov je pri každom nahradení objektu výrazná zmena pre oko hráča. Preto chceme implementovať riešenie, ktoré bude optimálne a implementácia či výroba nebude časovo náročná. Použijeme pri tom fraktály a to konkrétne L-systémy, ktoré sú určené na modelovanie rastlín a živých organizmov.

L-systémy predstavujú gramatiku, ktorá generuje reťazce. Tieto reťazce v sebe nesú informácie o vykreslení daného obrazca. Poznáme viacero druhov L-systémov napríklad parametrické, stochastické, bezkontextové a iné. Pri vykresľovaní sa používa tzv. *korytnačia grafika*. Každý symbol v reťazci má svoj význam a presne určuje, čo sa má vykonávať pri vykresľovaní.

Naším cieľom je zoznámiť sa s týmito L-systémami a niektoré z nich, konkrétne parametrické, implementovať. Pomocou týchto L-systémov následne

[1] Vrchol je trojica desatinných čísel, ktoré určujú pozíciu, kde sa nachádza.

vykreslíme textúru[2] ktorá bude predstavovať tzv. masku[3]. Ďalej vhodne použijeme túto masku pri raste rôznych druhov stromov, kríkov a budeme analyzovať scénu pri veľkom počte inštancii.

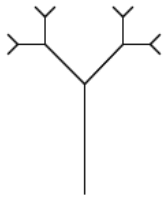
1.1 Ilustračný príklad

Základný typ L-systému je tzv. *D0L-systém*. *D* znamená deterministický a *0* znamená bezkontextový. Formálne je to trojica $G = (\sum,S,P)$, kde $\sum$ je abeceda symbolov, *S* je štartovací symbol z abecedy, *P* je množina prepisovacích pravidiel.

Príklad:
$\sum=\{0,1,[,]\}$
S=0
P= {(1→11), (0→1[0]0)}

1. iterácia: 1[0]0
2. iterácia: 11[1[0]0]1[0]0
3. iterácia: 1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0

0 – kresli čiaru (skonči v liste)
1 – kresli čiaru
[- vlož do zásobníka a otoč o 45° vľavo
] – vyber zo zásobníka a otoč o 45° vpravo



Obrázok 1. Vykreslená tretia iterácia.

1.2 Prehľad súčasného stavu

Implementácia celkového rastu stromov, kríkov je úspešne dokončená. Taktiež sú implementované všetky súčasti potrebné k celkovému rastu.

---

[2] Ľubovoľný obrázok.
[3] Obrázok obsahujúci rôzne informácie v každom kanáli. Nezáleží ako ten obrázok bude vyzerať.

2 Riešenie

Riešenie sa skladá z viacerých častí. Postupne sme vykonávali nasledujúce kroky.

Prvým krokom je vygenerovať reťazec zo zadaných pravidiel. Pravidlá môžu byť bezparametrické alebo parametrické.

Druhým krokom je vykreslenie vygenerovaného reťazca. Vykreslenie rozdelíme na dve menšie podúlohy. Prvou podúlohou je príprava dát a druhou je zobrazenie. Príprava dát spočíva vo vytvorení objektových údajov reprezentujúcich konárov (čiar) a listov, spolu s ich vlastnosťami ako sú pozícia, rotácia, veľkosť. Zobrazenie je už len vykreslenie pripravených údajov. Zobrazené údaje následne uložíme do súboru s príponou ".tga". Tento súbor budeme nazývať maska.

Tretím krokom je príprava samotného 3D objektu. Táto príprava spočíva vo vytvorení modelu, ktorý sa bude vedieť animovať a zmení svoj výzor na iný model. To znamená, že animácia začne v jednom modeli a skončí v inom, pričom platí rovnosť počtu vrcholov v oboch modeloch. Takáto animácia sa nazýva "mesh blending".

Štvrtým krokom je implementácia shader-a[4], ktorý podporuje "mesh blending" a bude rozumieť informáciám v maske.

Posledným piatym krokom je spojenie predchádzajúcich bodov do jedného fungujúceho celku a analýza scény takto vytvorených objektov.

2.1 Generovanie reťazcov

Vyššie sa vysvetľuje postup pri vytváraní výsledného reťazca na ilustračnom príklade. Začíname zo štartovacieho symbolu alebo reťazca, ktorý prechádzame a postupne pri každej iterácii nahradíme každý znak, ktorý má prepisovacie pravidlo v množine pravidiel. Ak sú to parametrické pravidlá tak dosadíme aktuálne hodnoty parametrov do premenných. Následne vyhodnotíme výrazy v parametroch a máme nové aktuálne hodnoty parametrov. Toto celé opakujeme toľkokrát, koľko je zadaných iterácii.

[4] Súbor s pravidlami a funkciami pre grafickú kartu, ktorým definujeme pokročilejšie funkcie pri zobrazovaní objektov.

```
CreateString(axiom, iteration)
  result ← axiom
  for i ← 0 to iteration
    iterationResult ← ""
    for j ← 0 to result.length
      ch ← result[j]
      if (ch >= 65 && ch <= 90) || ch == '+' || ch == '-'
        stringParam ← ""

        if j+1 < result.length && result[j+1] == '('
          stringParam ← SearchParameters(result, j)
          param ← ParsingParameters(stringParam)
          rule ← FindRule()
          ReplaceParameterValues(rule, param)
          EvaluateExpresions(rule)
          iterationResult ← iterationResult + rule
          continue

        iterationResult ← iterationResult + result[j]
      result ← iterationResult
```

2.2 Vykreslenie reťazca

Túto časť si rozdelíme na prípravu dát a samotné zobrazenie. Rozdelíme si to z dôvodu, že keď budeme chcieť zmeniť výslednú vizuálnu podobu tak nám stačí nanovo implementovať len druhú časť. Príprava dát je univerzálna a nezáleží ako bude následne zobrazená.

2.2.1 Príprava dát

Algoritmus dostane na vstupe reťazec, ktorý reprezentuje nejaký obrázok. Tento reťazec je vytvorený pomocou L-Systémov. Reťazec môže byť parametrický alebo bezparametrický. Pri reprezentácii znakov v reťazci sa používa tzv. korytnačia grafika. Korytnačka začína zo svojej štartovacej pozície a so štartovacou rotáciou. Postupným čítaním reťazca po jednotlivých znakoch korytnačka vie akú operáciu má vykonať. Základné znaky ako '+','-' znamenajú otočenie doprava alebo doľava o zadaný uhol. Znak '[' znamená vložiť aktuálnu pozíciu a rotáciu korytnačky do zásobníka. Znak ']' znamená nastaviť korytnačke pozíciu a rotáciu zo zásobníka. Malé písmena abecedy znamenajú posun vpred korytnačky bez kreslenia čiary a veľké písmena abecedy reprezentujú kreslenie čiary. Výnimkou je 'X' a to znamená, že korytnačka spraví svoj odtlačok.

```
PrepareData(s, pos, length, rot, angle)
  for i ← 0 to s.length
    if s[i] == '+'
      Rotate(angle)
    else if s[i] == '-'
      Rotate(-angle)
    else if s[i] == '['
      PushToStack(pos, rot)
    else if s[i] == ']'
      PopFromStack(pos, rot)
    else if s[i] >= 'a' && s[i] <= 'u'
      Step(length)
    else if s[i] >= 'A' && s[i] <= 'Z' && s[i] != 'X'
      line.startPos ← pos
      Step(length)
      line.endPos ← pos
      AddLine(line)
    else if s[i] == 'X'
      AddPos(pos)
```

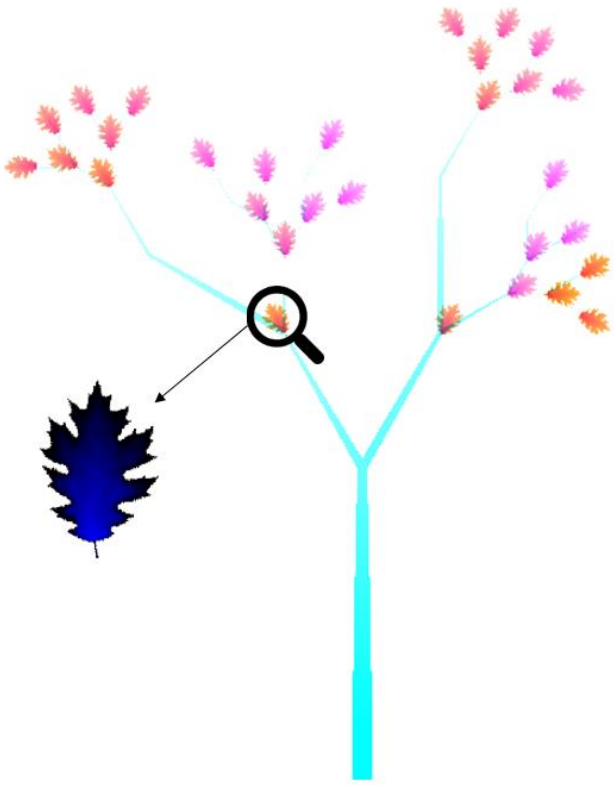
Po tomto algoritme vieme povedať koľko bude vykresľovaných čiar, o každej čiare kde začína a kde končí. Rovnako vieme presné pozície listov.

2.2.2 Zobrazenie pripravených dát

Keďže používame prostredie Unity [7], tak čiaru bude reprezentovať komponent "*LineRenderer*". Tento komponent obsahuje pole bodov, medzi ktorými vytvorí súvislú čiaru. Vieme nastaviť hrúbku čiary a to aj s rozdielnymi hodnotami na začiatku a konci. List bude reprezentovať jednoduchý objekt "*Quad*", ktorý je súčasťou prostredia Unity. Na tento objekt si nastavíme obrázok s nejakým listom. Použijeme Obrázok 2, ktorý obsahuje v modrom kanáli postupnú interpoláciu, ktorú použijeme neskôr pri raste listov.

Obrázok 2. Použitý obrázok listu (pečiatka korytnačky).

Obrázok 2 s konkrétnym listom predstavuje akúsi pečiatku korytnačky. List je zobrazený na pozícii a v smere aktuálneho postavenia a natočenia korytnačky. Vzniknutý obrázok si následne uložíme do obrázka vo formáte ".tga". Obrázok bude obsahovať v jednotlivých kanáloch odlišné informácie. Červený obsahuje informáciu o raste konára, zelený informáciu o raste listu a modrý predstavuje kedy začína daný list rásť. Výslednú masku môžeme vidieť na Obrázku 3 spolu s použitým listom.

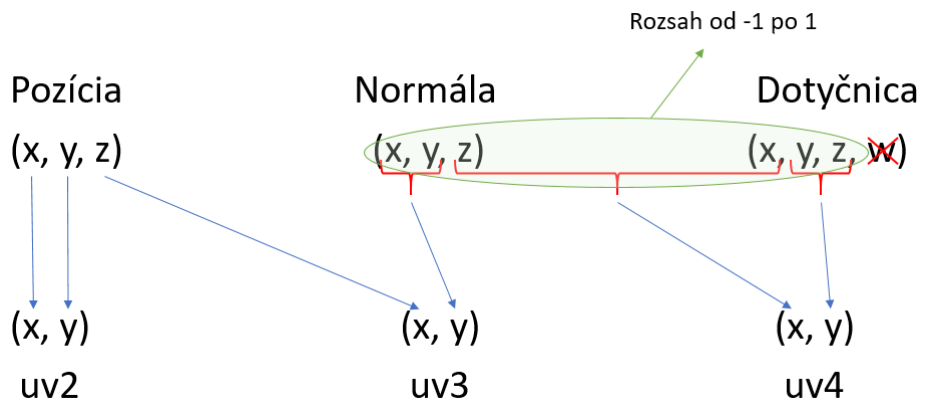


Obrázok 3. Výsledná maska.

2.3 Mesh blending

Animácia pomocou prelínania jedného meshu na iný sa zvykne nazývať "*mesh blending*" alebo aj "*morph target*" či "*blend shapes*". Takáto animácia sa najčastejšie používa vo filmoch a hrách na animácie tvárí osôb. Pri veľmi detailných objektoch, ktoré sú tvorené miliónmi vertexov je to zvyčajná metóda. Dá sa tak veľmi jednoducho simulovať rozprávanie ľudí. Stačí mať tvár vytvorenú v dôležitých pozíciách. My použijeme takúto animáciu na rast stromov. Strom v jednej fáze sa postupne zmení na strom v nasledujúcej fáze rastu. Unity však nepodporuje takýto typ animácie a preto si musíme vytvoriť náhradu. Náhrada spočíva vo vytvorení objektu, ktorý obsahuje informácie potrebné na takúto animáciu. My sme si zvolili objekt, ktorý uchováva v sebe n-1 meshov, ak n je počet meshov, čo sa má animovať. Každý i-ty mesh si uchováva potrebné informácie o nasledujúcom (i+1) meshi. Potrebné informácie sú pozície, normály a dotyčnice vertexov. Problémom je, že kam si tieto potrebné údaje zapíšeme. Existujú dve možnosti kam ich môžeme zapísať. Prvou možnosť je zapísať potrebné údaje do obrázka. Druhou možnosťou je komprimovať údaje a použiť voľné premenné v meshi. Za voľné premenné sa považujú také, ktoré nevyužívame pri žiadnej aktivite vrátane samotného renderovania. Keďže sme v 3D priestore tak pozícia, normála a dotyčnica obsahujú každá po tri premenné. Voľné premenné máme "*uv2*","*uv3*","*uv4*". Tieto obsahujú po dve premenné. Potrebujeme teda zapísať deväť premenných do šiestich. Využijeme fakt, že normála a dotyčnica majú rozsah od -1 po 1. Takže hodnoty normály a dotyčnice skomprimujeme po dvoch do jednej premennej ako v pseudokóde nižšie.

```
Pack(x, y)
  x ← Clamp01((x + 1) / 2)
  y ← Clamp01((y + 1) / 2)
  return x + Round(y * 65535)
```



Obrázok 4. Prehľad využitých premenných.

2.4 Shader

Shader v Unity slúži na presné definovanie nastavení a vlastností pri samotnom renderovaní. Pomocou shaderu vieme jednoducho zmeniť farbu alebo aj tvar vykresľovaného objektu. Zmenu tvaru objektu využijeme pri raste hlavného stromu a zmenu farby pri raste konárov a listov. Náš shader sa skladá z troch podstatných celkov. "*Mesh blending*", rast konárov a listov podľa masky a pohyb stromu vo vetre.

2.4.1 Mesh blending - shader

Táto časť úzko súvisí s uložením údajov o nasledujúcom meshi (viď 2.3). Nejaké premenné máme skomprimované, tak ich potrebujeme extrahovať. Môžeme použiť funkciu frac(x) priamo od NVIDIA, ktorá nám vráti hodnotu za desatinnou čiarkou.

```
Unpack(input)
  x ← frac(input)
  y ← (input - x) / 65535
  return Vector2(x, y) * 2 - 1
```

Ak vieme ako sa dostať k jednotlivým hodnotám, tak môžeme urobiť lineárnu interpoláciu. Tú použijeme medzi dvoma pozíciami vertexov, ich normálami aj dotyčnicou. Takto vyzerá výsledný pseudokód.

```
Blend(blend, v)
  float3 pos ← float3(v.uv2.xy, v.uv3.x);

  float2 unpack0 ← Unpack(v.uv3.y);
  float2 unpack1 ← Unpack(v.uv4.x);
  float2 unpack2 ← Unpack(v.uv4.y);
  float3 norm ← float3(unpack0.xy, unpack1.x);
  float3 tan ← float3(unpack1.y, unpack2.xy);

  v.vertex.xyz ← lerp(v.vertex.xyz, pos, blend);
  v.normal.xyz ← lerp(v.normal, norm, blend);
  v.tangent.xyz ← lerp(v.tangent.xyz, tan, blend);
```

2.4.2 Rast konárov a listov

Pri písaní shaderu môže byť veľmi neefektívne použitie klasického podmienkového príkazu *if*. Dôvodom je to, že sa nevie dopredu predpokladať, ktorá vetva sa vykoná tak sa vykonávajú všetky kombinácie vetiev. Chceme sa vyhnúť tomuto problému tak nebudeme používať žiadne vetvenie pomocou if-ov.

Vzniká nový problém ako rozhodnúť kde má byť aká farba. Vždy sa pozeráme aké informácie sú v maske a podľa týchto údajov a úrovne rastu musíme vedieť rozhodnúť, či sa na danom mieste nachádza drevená, listová časť stromu alebo žiadna (priesvitná). Pomocou jednoduchých matematických operácií ako sú násobenie, delenie, sčítanie a odčítanie vieme určiť čo sa má kde nachádzať. Používame hlavne princíp, že ak sa informácia na danom bode nemá nachádzať, tak sa násobí s nulou, a ak sa má nachádzať tak násobíme jednotkou.

Pri vykresľovaní jedného bodu sa pozrieme podľa *uv* mapovania na masku. Túto farbu použijeme jednotlivo po zložkách. Červený kanál obsahuje číslo, ktoré ak je väčšie ako úroveň rastu, tak znamená, že sa tu má nachádzať drevená časť stromu. Modrý kanál obsahuje číslo, ktoré ak je väčšie ako úroveň rastu, tak znamená, že tento list už je v nejakej fáze rastu. Posledný zelený kanál obsahuje informáciu, ktorá ak je väčšia ako úroveň rastu, tak znamená, že na tomto mieste má byť aj farba listu. To však iba vtedy, keď aj v modrom kanáli bola väčšia hodnota. Ak nastane situácia, že na danom mieste má byť aj drevená časť aj listová časť, tak prioritu má listová časť.

Ak zapojíme pri mixovaní farieb aj časovú hodnotu, tak môžeme dosiahnuť aj zmenu stromu podľa ročných období.

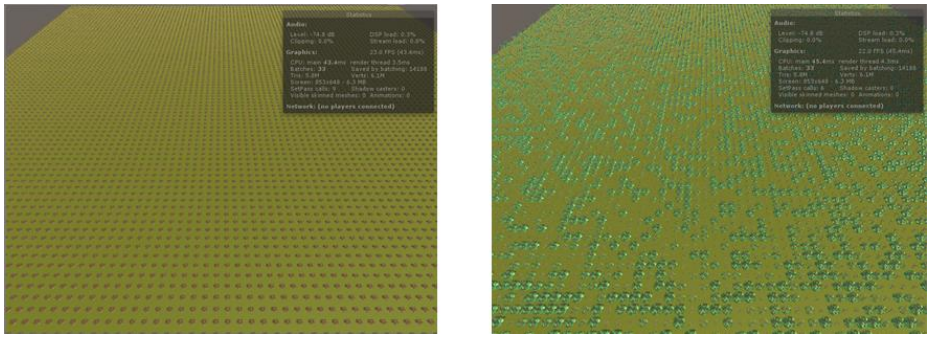
2.4.3 Vietor

Pohyb stromov vo vetre sa už bežne implementuje pomocou shadera. V našej implemetnácií sme takýto vietor pridali tiež do shadera. Intenzita pohybu vertexov vetrom je uložená vo farbe vertexu. Každý vertex obsahuje nejakú farbu a strom je vyrobený tak, aby na koncoch vetiev bola silnejšia intenzita. Naopak pri kmene stromu je intenzita slabšia.

Obrázok 5. Finálny rast stromu.

2.5 Analýza scény

Pri analyzovaní scény sme si vytvorili 10 000 inštancií stromu. Nastavili sme kameru, aby neoptimalizovala objekty, ktoré nie sú viditeľné. Najprv sme použili štandardný shader od vývojárov Unity. Takéto vykreslenie scény je statické a pri zapnutom inštancovaní na grafickej karte máme najoptimálnejšie vykreslenie aké Unity podporuje. Inštancovanie na grafickej karte znamená, že všetky objekty s rovnakým materiálom sú vykreslené v jednom "*draw call*" grafickej karty. Ak počet vertexov je väčší ako dovoľuje grafický ovládač tak je to rozdelené do viacerých *draw calls*. V našom prípade to je konkrétne 33 "*draw calls*" alebo inak nazývané "*batches*". Čas potrebný na celkové renderovanie je 43.4 ms a použité boli 9 "*Pass*" volania. "*Pass*" volaní je počet na koľkokrát sa daný objekt vykresľuje. Ak sme zmenili shader zo štandardného na náš vytvorený, tak počet "*draw calls*" sa nezmenil. Tiež podporujeme inštancovanie na grafickej karte. Počet volaní sa zmenilo na 6 a celkový čas renderovania sa zvýšil na 45.4 ms. Môžeme usúdiť, že sa vykresľovanie spomalilo o veľmi malý čas, čiže sme dosiahli veľmi efektívny výsledok. Každý strom môže byť v inej fáze rastu, pretože používame inštancované premenné a predsa sa ich grafická karta vníma ako rovnaké objekty. Stromy sú animované a rýchlosť vykresľovania je takmer ako pri statických objektoch.



Obrázok 6. Analýza scény.

3 Záver

Implementovali sme rast stromu, ktorý je animovaný na grafickej karte. Použili sme vývojové prostredie Unity. V Unity sa dajú veľmi efektívne vykresľovať statické objekty. Pri implementácii sme použili generovanie L-systémov a známu animáciu mesh blending, ktorá nie je súčasťou vývojového prostredia Unity. Rast sme vyskúšali na veľkom počte inštancií a zistili sme, že rast je takmer tak efektívny ako statické objekty. K rastu sme pridali aj animáciu vetra.

Literatúra

1. Pedrosa D.S.: GALSYS - Procedural Creation of Trees - through Combination of Genetic Algorithms and Lindenmayer Systems, 2014, ISBN 978-3639643763
2. Quigley E., et al.: Real-time Interactive Tree Animation, IEEE Transactions on Visualization and Computer Graphics Vol 22, 2017
3. Fornander P.: Game Mechanics Integrated with a Lindenmayer System, thesis, School of Computing, Blekinge Institute of Technology, Sweden, 2013
4. Hornby G.S., Pollack J.B.: Evolving L-systems to generate virtual creatures, Computer & Graphics (25) 6, 2001, pp. 1041-1048
5. Trube B.: Fractals - A Programmer's Approach, Amazon Digital Services, 2013

Prusinkiwicz P., Hanan J.: Lindenmayer systems, fractals, and plants, Springer Science & Business Media, Springer Verlag, 1989, ISBN 978-0387970929

6. Prusinkiewicz P., Hanan J.: Lindenmayer Systems, Fractals, and Plants, 1988
7. Unity, [online] Dostupné na: https://unity3d.com/