SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1851

PROGRAMI ZA SJENČANJE GEOMETRIJE

Josip Maričević

Zagreb, lipanj 2010.

Sadržaj

Uvod 1

1. Programiranje programa za sjenčanje 3
   1.1. Jezici za programiranje 3
   1.2. GLUT 3
   1.3. GLEW 3
   1.4. Korištenje programa za sjenčanje u neovisnom programu 4
   1.5. GLSL 6
2. Primjeri programa za sjenčanje 8
   2.1. Tehnika prikaza Cel-shading 8
   2.2. Transformacije geometrije 10
      2.2.1. Izvijanje 11
      2.2.2. Skaliranje 13
   2.3. Prozirnost 14
      2.3.1. Jednostavna prozirnost 14
      2.3.2. Efekt stakla 16
   2.4. Osvjetljenje po slikovnom elementu 19
3. Generiranje staničnih tekstura 22
   3.1. Načini generiranja staničnih tekstura 23
      3.1.1. Generiranje staničnih tekstura ponašajnim metodama 23
      3.1.2. Generiranje staničnih tekstura pomoću točki značajki 24
      3.1.3. Stohastičke metode 28
      3.1.4. Ostale manje poznate stohastičke tehnike 30
4. Programi za sjenčanje geometrije 31
   4.1. Ograničenja i mogućnosti programa za sjenčanje geometrije 33
   4.2. Primjena programa za sjenčanje geometrije 36

4.2.1. Razina detalja prikaza ... 36
4.2.2. Odbacivanje po projekcijskom volumenu ... 36
4.2.3. Preslikavanje s kocke u jednom prolazu ... 37
5. Implementacija generiranja stanične geometrije i tekstura u stvarnom vremenu ... 38
5.1. Generiranje bodlji ... 38
5.1.1. Program za sjenčanje vrhova ... 40
5.1.2. Program za sjenčanje geometrije ... 41
5.1.3. Program za sjenčanje fragmenata ... 43
5.1.4. Žičani prikaz ... 44
5.1.5. Performanse ... 46
5.2. Generiranje dlake ... 48
5.2.1. Bezierove krivulje ... 48
5.2.2. Program za sjenčanje vrhova ... 51
5.2.3. Program za sjenčanje geometrije ... 51
5.2.4. Program za sjenčanje fragmenata ... 54
5.2.5. Rezultati ... 55
5.2.6. Performanse ... 55
5.3. Glavni program ... 56
Zaključak ... 59
Literatura ... 60
Sažetak ... 62
Summary ... 63
Privitak ... 65

Uvod

U računalnoj grafici, programi za sjenčanje su skup instrukcija koji se koristi prvenstveno za izračunavanje efekata prilikom iscrtavanja slike na grafičkom sklopovlju. Pomoću njih se programiraju cjevovodi za ostvarivanje prikaza, što predstavlja velik pomak u odnosu na cjevovode s fiksnim funkcijama koji su dozvoljavali samo jednostavne geometrijske transformacije i sjenčanje slikovnih elemenata.

Ime su dobili jer su se inicijalno koristili samo za sjenčanje slikovnih elemenata, no taj izraz se zadržao i sad se koristi i za druge faze koje se odvijaju u grafičkim cjevovodima.

Kako se programirljivo grafičko sklopovlje razvijalo, svim većim programskim bibliotekama, kao što su npr. *OpenGL* i *DirectX*, su se počele dodavati mogućnosti pisanja programa za sjenčanje.

U osnovi, programi za sjenčanje su manji programi koji opisuju neka svojstva slikovnih elemenata (npr. boja, z-dubina, alpha vrijednost) ili vrha (npr. pozicija, koordinate teksture).

U *Direct3D*-u i *OpenGL*-u postoje tri osnovna tipa programa za sjenčanje:

- **Programi za sjenčanje vrhova** (engl. *vertex shader*) – programi za sjenčanje koji se izvode za svaki vrh proslijeđen grafičkom procesoru. Njihova svrha je da transformiraju trodimenzionalnu poziciju svakog vrha u dvodimenzionalne koordinate za prikaz na ekranu. Programi za sjenčanje vrhova mogu manipulirati svojstvima dobivenog vrha, no ne mogu stvarati nove vrhove. Rezultati programa za sjenčanje vrhova se tada prosljeđuju programima za sjenčanje geometrije ako postoje, a ako ne tada idu direktno u sustav za rasterizaciju.
- **Programi za sjenčanje geometrije** (engl. *geometry shader*) – za razliku od programa za sjenčanje vrhova mogu dodavati ili uklanjati vrhove iz mreže (engl. *mesh*). Koriste se da proceduralno definiraju geometriju, ili da dodaju

detalje sceni koji bi bili prezahtjevni da se obrađuju na procesoru. Izlaz iz ovih programa se prosljeđuje na ulaz sustava za rasterizaciju.

- **Programi za sjenčanje** slikovnih elemenata (engl. *pixel shader*) – dobivaju ulaz od sustava za rasterizaciju, te potom primjenjuju željene efekte poput preslikavanja izbočina i toniranja boja. U *OpenGL*-u se koristi izraz program za sjenčanje fragmenata (engl. *fragment shader*), što je tehnički točnije jer ne postoji jedan na jedan veza između poziva programa za sjenčanje slikovnih elemenata i prikazanih slikovnih elemenata.

1. Programiranje programa za sjenčanje

1.1. Jezici za programiranje

U *OpenGL*-u od verzije 1.5 nadalje postoji jezik sličan C-u, *OpenGL Shading Language* ili *GLSL*. Za potrebe ovog seminara, svi programi za sjenčanje su isprogramirani u *GLSL*-u.

U Microsoftovom *Direct3D*-u programi za sjenčanje se programiraju pomoću jezika koji se zove *High Level Shading Language* ili *HLSL*.

Mogućnosti gore navedenih jezika su skoro identične, razlike postoje samo u semantici i načinu programiranja, tako da postoje alati koji služe za automatsku konverziju programa iz jednog jezika u drugi.

Osim navedenih jezika, postoji još i *Cg* ili *C for Graphics,* koji je Nvidia razvila u suradnji s Microsoftom, te dijeli dosta sličnosti s *HLSL*-om.

1.2. GLUT

GLUT biblioteka je skup funkcija za *OpenGL* programe, koje većinom obavljaju ulazno izlazne operacije na razini operacijskog sustava, te time olakšavaju taj posao programeru. To uključuje definiciju prozora, kontrolu i upravljanje prozorom i očitavanje ulaza sa tipkovnice i miša. Također su dostupne funkcije za prikaz nekih primitiva poput kocke, kugle i čajnika (*Utah teapot*).

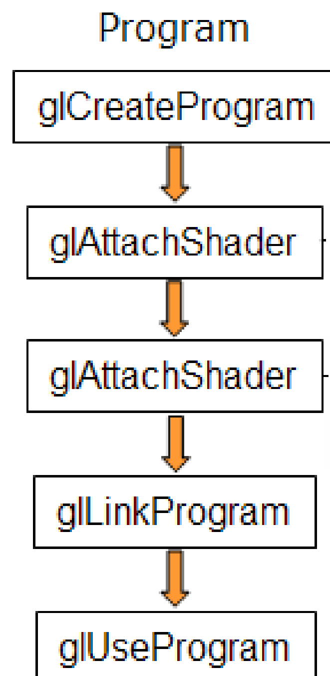
1.3. GLEW

GLEW je C/C++ višeplatformska biblioteka koja olakšava pozivanje i učitavanje *OpenGL* ekstenzija. Omogućava nam efikasno provjeravanje koje *OpenGL* ekstenzije su dostupne za vrijeme izvođenja programa. Sve dostupne ekstenzije su izložene u jednoj .h datoteci koja se generira pomoću službene liste ekstenzija. Sam naziv *GLEW* je nastao kao kratica od engleskog naziva *OpenGL Extension Wrangler Library.*

1.4. Korištenje programa za sjenčanje u neovisnom programu

Postoje dva osnovna načina za prevođenje i povezivanje programa za sjenčanje u glavnom programu pomoću *OpenGL*-a. Prvi je korištenje izvornih *OpenGL* 2.0 funkcija, ili u slučaju da nemamo podršku za *OpenGL* 2.0 korištenje ARB ekstenzija. Naziv ARB je nastao kao skraćenica od Architecture Review Board, što je neovisni konzorcij koji predlaže i odobrava promjene *OpenGL* specifikacija, nova izdanja i ispituje usklađenost sa standardom. Neka dodatna funkcionalnost postaje dio ARB ekstenzija (tzv. standardnih ekstenzija) tek ako nekoliko proizvođača implementira istu funkcionalnost i ako Architecture Review Board nakon detaljnog pregleda odobri navedene implementacije.

Sljedeća slika (Slika 1) prikazuje minimalne potrebne osnovne korake da prevedemo i povežemo program za sjenčanje s našim glavnim programom.



Slika 1. Potrebni koraci za prevođenje i korištenje programa za sjenčanje.

Pošto Windowsi XP nemaju izvornu podršku za *OpenGL* 2.0, u prvom dijelu ovog rada su korištene ARB ekstenzije za prevođenje i povezivanje programa za sjenčanje.

Prvi korak je kreiranje objekta koji će služiti kao spremnik za program za sjenčanje. U ovom slučaju umjesto `GLuint glCreateProgram(void)` koristimo ARB ekvivalent `GLhandleARB glCreateProgramObjectARB(void)`. Broj programa koji možemo ovako napraviti nije ograničen, a za vrijeme ostvarivanja prikaza možemo se prebacivati između programa, pa se čak i vratiti na fiksnu funkcionalnost tijekom prikaza jednog okvira. Na primjer, možemo iscrtati čajnik primjenjujući na njemu neki od programa za sjenčanje dok se istovremeno pozadina prikazuje koristeći fiksnu funkcionalnost *OpenGL*-a.

Idući korak uključuje dodavanje programa za sjenčanje. U ovom trenutku oni ne trebaju biti prevedeni, pa čak nije ni nužno da postoje, potreban nam je samo spremnik koji smo napravili u prvom koraku. ARB funkcija koju koristimo za to je `void glAttachObjectARB(GLhandleARB program, GLhandleARB shader)`.

Prevođenje programa za sjenčanje se radi pomoću funkcija `void glShaderSourceARB(GLhangleARB shader, GLuint number_strings, const GLcharARB** strings, Glint * length)` i `void glCompileShader(GLhandleARB shader)`.

Zadnji korak je povezivanje i stavljanje programa u uporabu, što radimo sa funkcijama `void glLinkProgramARB(GLhandleARB program)` i `void glUseProgramObjectARB(GLhandleARB prog)`. Ako za parametar *prog* proslijedimo 0, aktivira se fiksna funkcionalnost cjevovoda.

Naravno, prije svega navedenog potrebno je provjeriti da li uopće postoji podrška za GLSL.

Primjer provjere dostupnosti ARB ekstenzija za rad sa programima za sjenčanje:

```
glewInit();
if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
      printf("Graficka kartica podrzava GLSL\n");
```

```
else
{
      printf("Nema podrske za GLSL\n");
      exit(1);
}
```

1.5. GLSL

GLSL sadrži sve operatore iz programskog jezika C uz iznimku pokazivača. Operatori za rad s bitovima su dodani u verziji 1.30.

Slično C-u, podržava petlje i različite oblike grananja uključujući `if`, `else`, `for`, `do-while`, `break` i `continue`. Korisnički definirane funkcije su također podržane i većina često korištenih funkcija je ugrađena. Ovo dozvoljava proizvođačima grafičkih kartica da optimiziraju te funkcije na sklopovskoj razini. Primjer takvih funkcija su `exp()` i `abs()`. Također su ugrađene i neke funkcije koje su specifične za rad s računalnom grafikom poput `smoothstep()` i `texture2D()`.

Varijable se dijele na tri osnovna tipa:

- **Atributne** (engl. *attribute)* - varijable koje se zadaju na razini vrha. Ova vrsta varijabli se odnosi isključivo samo na procesor vrhova.

- **Uniformne** (engl. *uniform)* - varijable zadane na razini primitive. Ove se varijable prenose procesoru vrhova, procesoru fragmenata ili procesoru geometrije.

- **Promjenjive** (engl. *varying)* - varijable koje procesor vrhova prenosi procesoru fragmenata ili geometrije.

Programi za sjenčanje napravljeni pomoću *GLSL*-a nisu samostalni, već zahtijevaju aplikaciju koja koristi *OpenGL* programsko sučelje. Implementacija *OpenGL* sučelja je dostupna na mnogim različitim platformama, a postoje i sučelja za različite programske jezike.

Sami programi za sjenčanje su jednostavno niz tekstualnih naredbi koji se prosljeđuju upravljačkom programu uređaja da ih prevede u izvršne programe. Skup funkcija koje se koriste za prevođenje i prosljeđivanje parametara *GLSL* programima je specificiran u tri ekstenzije *OpenGL*-a, a postali su sastavni dio *OpenGL*-a od verzije 2.0.

2. Primjeri programa za sjenčanje

U sljedećih nekoliko poglavlja opisani su programi za sjenčanje koji su isprogramirani za demonstraciju mogućnosti i prednosti programa za sjenčanje nad klasičnim ostvarivanjem prikaza pomoću cjevovoda s fiksnim funkcijama. U opisu svakog programa za sjenčanje bit će navedeni samo dijelovi koda koji su bitni za njih, dok se dio koda koji se bavi općenitim ostvarivanjem prikaza i manipuliranjem scene ne smatra temom ovog rada, ali se može vidjeti na priloženom CD-u.

Kao pomoć prilikom dizajniranja i programiranja samih programa za sjenčanje, korišten je *AMD RenderMonkey 1.81*, dok je glavni program za demonstraciju korištenja programa za sjenčanje unutar samostalnog projekta napravljen u C-u (pomoću *Visual Studia 2005*), te koristi *GLUT* i *GLEW* biblioteke.

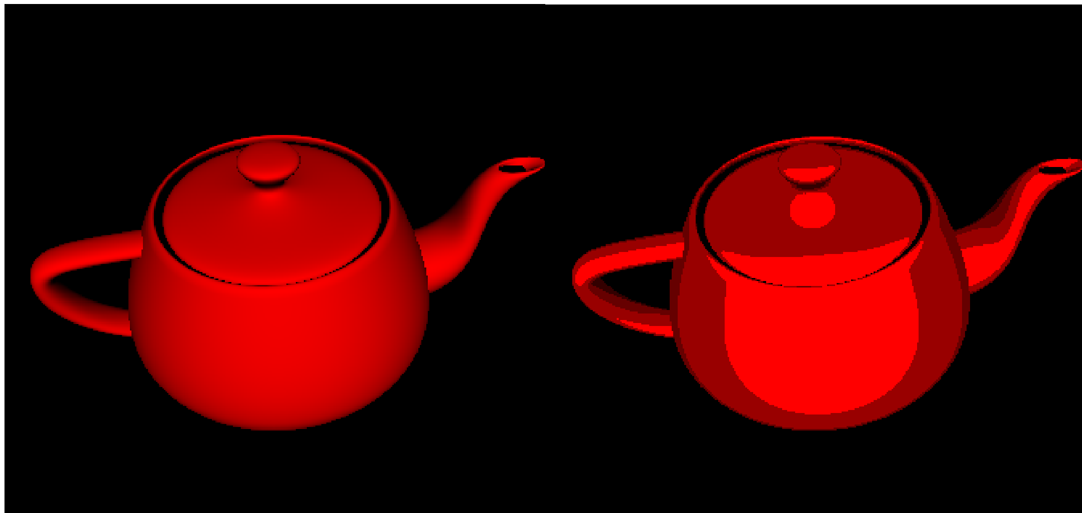
Također, bitno je napomenuti da za *GLSL* ne postoji efektivan način ispravljanja grešaka (postoje neki zaobilazni načini), tako da pronalaženje grešaka koje nisu greške kod prevođenja postaje vremenski veoma zahtjevan posao.

2.1. Tehnika prikaza Cel-shading

Jedan od najpoznatijih nefotorealističnih načina prikaza, dizajniran da se dobije dojam crtanja rukom. Najčešće se koristi u animiranim filmovima ili stripovima, ali ima i drugih primjena poput npr. povećanja čitljivosti tehničkih ilustracija. Glavna razlika između konvencionalnog ostvarivanja prikaza i cel-shadinga je u načinu računanja osvjetljenja.

Prvo se izračuna normalno osvjetljenje za svaki slikovni element, te se potom tako izračunata vrijednost pretvara u neku od malog broja diskretnih vrijednosti. Svjetlija i tamnija područja tada izgledaju kao blokovi boje umjesto ravnomjernih prijelaza.

Nakon toga se po volji mogu dodavati efekti poput iscrtavanja obruba. Ako se ne iscrtavaju rubovi, dovoljan je samo jedan prolaz.



Slika 2. Čajnik bez uporabe programa za sjenčanje (lijevo) i Čajnik s programom za sjenčanje koji vrši diskretiziranje u četiri nijanse (desno).

Program za sjenčanje vrhova:

```
uniform vec3 lightDir;
varying float intensity;

void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;

   vec3 normal = gl_NormalMatrix*gl_Normal;
   intensity=max(0.0, dot(lightDir, normal));

}
```

Program za sjenčanje fragmenata:

```
uniform vec4 color;
varying float intensity;

void main(void)
{
   if(intensity<=0.2)
      gl_FragColor=color*0.2;

   if(intensity>0.2 && intensity<=0.5)
```

```
        gl_FragColor=color*0.4;

    if(intensity>0.5 && intensity<=0.8)
        gl_FragColor=color*0.6;

    if(intensity>0.8)
        gl_FragColor=color;

}
```

Kao što vidimo, program za sjenčanje vrhova ne utječe na samu poziciju vrha (varijabla *gl_Position*), nego je izračunava standardno kako bi se izračunavala bez posebno napisanog programa. Bitna stvar je računanje intenziteta svjetla u tom vrhu koji se prosljeđuje programu za sjenčanje fragmenata.

Program za sjenčanje fragmenata potom kontinuirani intenzitet jednostavno dijeli u 4 diskretne vrijednosti. Jedan od načina na koji bi se ovo moglo postići je i da napravimo teksturu koja bi tada po volji mogla prema vrijednosti intenziteta određivati boju bloka. Primjer takve teksture imamo na slici 3.

Slika 3. Primjer teksture (uvećane) koja može poslužiti za diskretiziranje. Prava veličina teksture je 32x1.

Kad bi imali takvu teksturu, onda bi se krajnja vrijednost boje slikovnog elementa (*gl_FragColor*) određivala na sljedeći način:

```
        vTexCoord.x = intensity;
        vTexCoord.y = 0.0;
gl_FragColor = texture2D(CelShaderTexture, vTexCoord);
```

2.2. Transformacije geometrije

Jedan od korisnih efekata koji se mogu postići pomoću programa za sjenčanje vrhova, bez značajnog utjecaja na performanse je mijenjanje geometrije scene na

različite načine. Od najjednostavnijeg skaliranja, preko različitih izobličenja scene, pa do kompliciranijih efekata tipa pretvaranja iz jednog oblika u drugi.

Također, kao argument o kojemu ovisi bilo koja od funkcija izobličenja može biti bilo što, što se može proslijediti iz glavnog programa, npr. vrijeme, pozicija miša, pozicija na sceni, slučajni broj i sl. Kombinirajući to s ostvarivanjem prikaza u teksturu moguće je postići mnogo različitih efekata (npr. efekt tkanine koja leluja na vjetru, efekt LED oglasne ploče, efekt pokvarenog televizora) koji bi inače zahtijevali puno više procesorske snage.

2.2.1. Izvijanje

Primjer efekta izvijanja pomoću programa za sjenčanje s jednim prolazom:

Program za sjenčanje vrhova:

```
varying vec2  texCoord;
varying float shading;

void main(void)
{
vec4 v1 = gl_ModelViewProjectionMatrix * rotMat1 *
vec4(gl_Vertex.xyz,1.0);

vec4 v2 = gl_ModelViewProjectionMatrix * rotMat2 *
vec4(gl_Vertex.xyz,1.0);

      gl_Position = v1 * (1.0-mat2Influence) + v2 *
mat2Influence;

// Samo prenesi koordinate teksture
      texCoord = gl_MultiTexCoord0.xy;

       shading = -dot(LightDir,gl_Normal.xyz);
       shading = max(shading,0.4);
 }
```

Program za sjenčanje fragmenata:

```
uniform sampler2D BaseTex;
```

```
varying vec2  texCoord;
varying float shading;

void main(void)
{
   vec4 baseColor = texture2D(BaseTex,texCoord);
   gl_FragColor = baseColor * shading;
}
```

Matrice *rotMat1* i *rotMat2* su rotacijske matrice koje program za sjenčanje dobiva iz glavnog programa, a u glavnom programu se postavljaju ovisno o vremenu. U ovom slučaju te matrice su:

$$\text{rotMat1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(angle) & -\sin(angle) & 0 \\ 0 & \sin(angle) & \cos(angle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

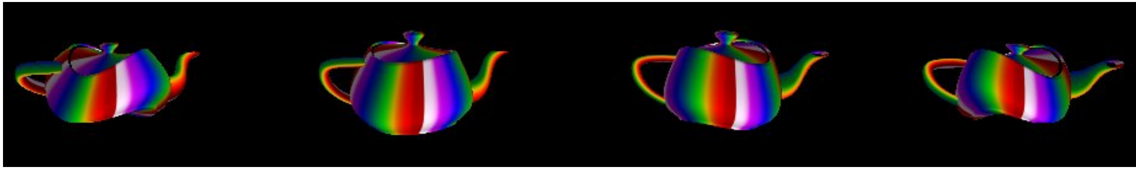
$$\text{rotMat2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(angle) & \sin(angle) & 0 \\ 0 & -\sin(angle) & \cos(angle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dok je *angle* varijabla koja se mijenja u ovisnosti o vremenu. Varijabla *mat2influence* je varijabla koja za svaki vrh u ovisnosti o njegovoj *x* koordinati računa koliki je utjecaj pojedine matrice rotacije, i također se prosljeđuje iz glavnog programa. Što je *x* koordinata vrha bliža 0, matrica *rotMat1* ima veći utjecaj, dok udaljavanjem od 0 *rotMat2* dobiva veći utjecaj.

U samom programu za sjenčanje vrhova prvo računamo *v1* koji predstavlja poziciju vrha koju bi dobili kad bi ga rotirali samo pomoću prve matrice, a potom *v2* koji za matricu rotacije uzima drugu matricu. Na kraju završnu poziciju određujemo tako da računamo težinski prosjek od *v1* i *v2*, pri čemu *mat2influence* predstavlja težinu vrhova.

Pošto je ovaj efekt baziran na transformaciji geometrije, sam program za sjenčanje fragmenata je trivijalan, primjenjuje intenzitet osvjetljenja izračunat u programu za sjenčanje vrhova na potrebnu točku teksture.

Efekt koji dobivamo ovim programom možemo vidjeti na slici (**Slika 4**).



Slika 4. Slike čajnika u različitim vremenskim trenucima prilikom izvođenja.

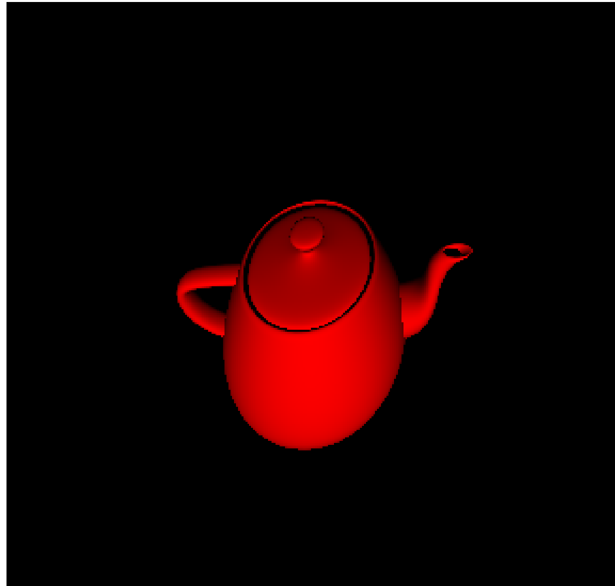
2.2.2. Skaliranje

Efekt skaliranja također nije kompliciran za implementaciju pomoću programa za sjenčanje. Pošto se u osnovi skaliranje svodi na množenje matrice projekcije vektorom u kojem elementi označavaju faktor skaliranja po svakoj osi, to je jedina dodatna linija koju trebamo dodati u odnosu na ostale programe. Program za sjenčanje fragmenata se ne mijenja, pa ga ne treba dodatno ni objašnjavati.

Program za sjenčanje vrhova:

```
uniform vec4 scale;
uniform vec3 lightDir;
varying float intensity;

void main(void)
{
   vec4 a=gl_Vertex;

   a*=scale;
   gl_Position = gl_ModelViewProjectionMatrix*a;

   vec3 normal = gl_NormalMatrix*gl_Normal;
   intensity=max(0.0, dot(lightDir, normal));
}
```

Varijabla *scale* je 4-dimenzionalni vektor koju program za sjenčanja dobiva iz glavnog programa, te u ovom slučaju mijenja vrijednost u ovisnosti o položaju miša. Prvi element označava faktor skaliranja po x-osi, drugi po y, a treći po z.



Slika 5. Čajnik skaliran sa scale = $[0.5\ 0.5\ 1.0\ 1.0]^T$.

2.3. Prozirnost

2.3.1. Jednostavna prozirnost

Ponekad nam je potreban jednostavan efekt prozirnosti, u kojem možemo samo označiti koji dijelovi objekta su potpuno prozirni, a koji nisu uopće. Tako možemo dobiti izgled objekata kojima nedostaju dijelovi, npr. korodirani objekti, ili objekti sa rupama od projektila.

Jedan od načina određivanja koje slikovne elemente treba ukloniti je pomoću alfa vrijednosti teksture za određeni slikovni element no broj mogućih načina je vrlo velik – od uklanjanja samo slikovnih elemenata određenih boja do potpuno slučajnog odabira.

Program za sjenčanje vrhova:

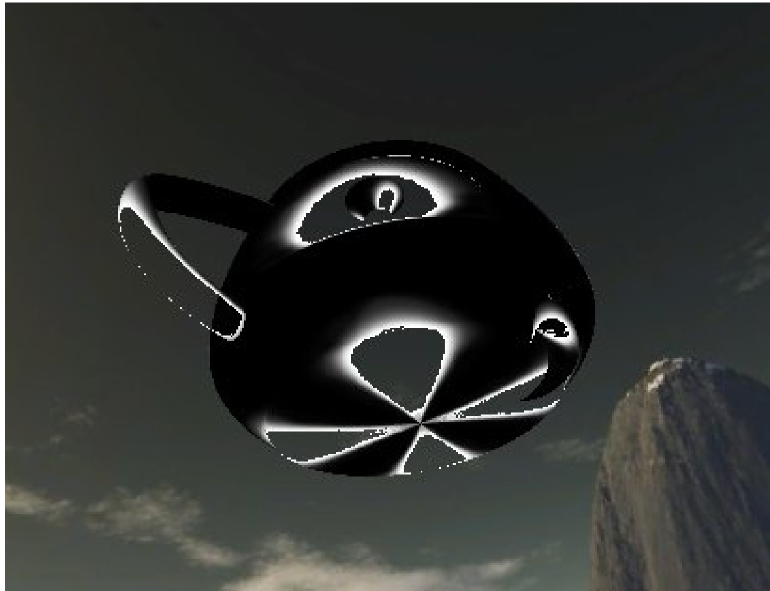
```
void main(void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Program za sjenčanje fragmenata:

```
uniform sampler2D tekstura;

#define epsilon 0.0001

void main (void)
{
  vec4 vrijednost = texture2D(tekstura, vec2(gl_TexCoord[0]));

  if(vrijednost[0]>1.0-epsilon) discard;

  gl_FragColor = vrijednost;
}
```

Kao što vidimo iz koda, sav posao ovdje odrađuje program za sjenčanje fragmenata. U ovisnosti o zadanoj osjetljivosti *epsilon* i komponenti crvene boje teksture trenutnog slikovnog elemenata pomoću posebne ključne riječi `discard` jednostavno odbacuje slikovne elemente koje smo odredili kao prozirne.

Sam okoliš je prikazan pomoću preslikavanja s kocke (engl. *cube mapping*) da bi se bolje uočio način rada programa. Rezultate možemo vidjeti na slikama 6 i 7.



Slika 6. Pogled na čajnik s donje strane.

Slika 7. Pogled na čajnik s desne strane.

2.3.2. Efekt stakla

No ako želimo vjerno prikazati materijal koji je proziran, onda to ne možemo izvesti na jednostavan način, jer ne postoje materijali koji su savršeno prozirni, već

svi imaju određeni koeficijent refleksije. Sam efekt fotorealističnog prikaza prozirnog stakla je tema za sebe, tako da ćemo se ovdje usredotočiti samo na model korišten za implementiranje ovog programa za sjenčanje.

U programu za sjenčanje vrhova, osim normalnog računanja pozicije vrha, računamo i vektor pogleda *vViewVec* koji će nam biti potreban u programu za sjenčanje fragmenata.

Potom u programu za sjenčanje fragmenata računamo varijablu *v* kao skalarni produkt normaliziranog vektora pogleda i vektora normale, koji nam u biti određuje prozirnost trenutačnog fragmenta. Što je *v* veći, to znači da nam je vektor pogleda okomitiji na taj fragment, što implicira i bolju prozirnost i manju refleksiju. No osim prozirnosti, varijabla *v* nam također služi za dobivanje efekta disperzije svjetlosti te nju koristimo da odredimo koordinate elementa teksture duge za trenutačni fragment.

Vektor refleksije vektora pogleda u odnosu na normalu ne moramo računati ručno, nego ga jednostavno možemo dobiti pomoću ugrađene funkcije `reflect()` koja je dio *GLSL*-a. Tada pozivamo funkciju `textureCube()` koja kao argumente uzima teksturu za uzorkovanje i trodimenzionalni vektor koji odgovara smjeru odbijanja vektora pogleda na određeni vrh. Iz tog vektora, funkcija određuje koju stranu kocke treba uzorkovati, te zatim projicira vektor na tu plohu kako bi se dobile dvodimenzionalne koordinate elementa teksture.

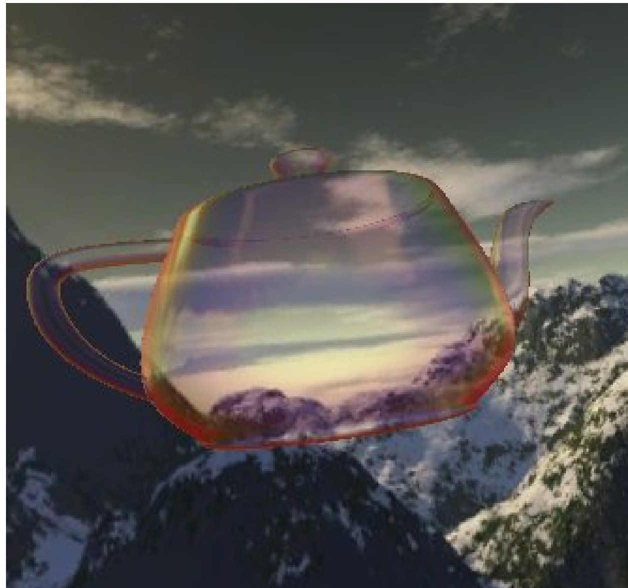
Krajnja boju fragmenta dobivamo kao linearnu interpolaciju između dva dobivena elementa teksture uz pomoć funkcije `mix()`.

Program za sjenčanje vrhova:

```
uniform vec4 view_position;
varying vec3 vNormal;
varying vec3 vViewVec;

void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * glVertex,

   vNormal = gl_Normal;
   vViewVec = view_position.xyz - glVertex.xyz;
}
```

Program za sjenčanje fragmenata:

```
uniform sampler2D Duga;
uniform samplerCube Okolis;
uniform float k_duga;

varying vec3 vNormal;
varying vec3 vViewVec;

void main(void)
{
   float v =  dot(normalize(vViewVec), vNormal);

   vec3 duga = texture2D(Duga, vec2(v,0.0)).xyz;

   vec3 reflVec =  reflect(-vViewVec, vNormal);
   vec3 refl = textureCube(Okolis, reflVec).xyz;

   gl_FragColor = vec4 (mix(refl, duga, k_duga * v), 1.0 - v);
}
```



Slika 8. Čajnik sa efektom stakla.

2.4. Osvjetljenje po slikovnom elementu

Osvjetljenje po slikovnom elementu (engl. *per-pixel lighting)* je često korištena metoda za računanje osvjetljenja u računalnoj grafici. Dobivene slike su mnogo realističnije nego one dobivene pomoću računanja osvjetljenja vrhova (Gouradovo sjenčanje), no istovremeno i mnogo zahtjevnije što se tiče procesorske snage. Za potrebe aplikacija koje se izvode u realnom vremenu, implementacija osvjetljenja po slikovnom elementu se tipično izvodi pomoću programa za sjenčanje.

Ova tehnika se često koristi u kombinaciji s preslikavanjem izbočina, zrcalnim osvjetljenjem i volumenima sjena.

Program za sjenčanje vrhova:

```
uniform vec4 lightDir;
varying vec2  vTexCoord;
varying vec3  vNormal;
varying vec3  vLightVec;
varying vec3  vViewVec;

void main(void)
{
   vLightVec   = -lightDir.xyz;
   vNormal = normalize( gl_NormalMatrix * gl_Normal);
   vViewVec = -normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
   vTexCoord = vec2(gl_MultiTexCoord0);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Program za sjenčanje fragmenata:

```
uniform sampler2D Texture0;
uniform float n_specular;
uniform float Ks;
uniform float Ka;
uniform float Kd;
uniform vec4 diffuse;
uniform vec4 specular;
uniform vec4 ambient;
```

```
varying vec2  vTexCoord;
varying vec3  vNormal;
varying vec3  vLightVec;
varying vec3  vViewVec;

void main(void)
{

vec3 vReflect = normalize(2.0 * dot(vNormal, vLightVec) *
vNormal - vLightVec);

vec4 AmbientColor = ambient * Ka;

vec4 DiffuseColor = diffuse * Kd * max( 0.0, dot( vNormal,
vLightVec ));

vec4 SpecularColor = specular * Ks * pow( max( 0.0,
dot(vReflect, vViewVec)), n_specular );

vec4 FinalColor = (AmbientColor + DiffuseColor) * texture2D(
Texture0, vTexCoord) + SpecularColor;

      gl_FragColor = FinalColor;
}
```

U programu za sjenčanje vrhova, nakon pozicije koja se računa konvencionalno, računamo vektor prema svjetlu u prostoru pogleda koji ćemo proslijediti programu za sjenčanje fragmenata. Potom nakon toga poziciju vrha, normalu i vektor pogleda u prostoru pogleda.

Program za sjenčanje fragmenata prvo računa vektor refleksije te potom ambijentalno osvjetljenje, difuzno osvjetljenje i zrcalno osvjetljenje. Krajnja boja fragmenta se računa tako da dodamo ambijentalno i difuzno osvjetljenje, primijenimo ih na teksturu objekta te na kraju samo zbrojimo sa zrcalnom komponentom.

Slika 9. Čajnik sa teksturom i osvjetljenjem po slikovnom elementu.

3. Generiranje staničnih tekstura

Proceduralno generiranje je vrlo širok pojam koji se koristi u računalnim znanostima, a označava sadržaj koji je generiran algoritamski, a ne ručno. Najčešće se to odnosi na generiranje sadržaja prilikom izvođenja programa, a ne prije distribucije programa. Iako je područje primjene proceduralnog generiranja vrlo široko, najčešće se koristi i spominje u kontekstu računalne grafike. Pojam proceduralno se odnosi na proces koji računa određenu funkciju. Fraktali su jedan od najpoznatijih i najraširenijih primjera proceduralnog generiranja.

U računalnoj grafici, obično se taj postupak koristi za generiranje tekstura i mreža. Razlozi stvaranja sadržaja za vrijeme izvođenja su raznoliki, od uštede prostora, kreiranja velikog broja različitih modela i tekstura koje bi bilo nepraktično i vremenski zahtjevno raditi ručno, pa sve do generiranja tekstura koje izravno odgovaraju površini modela.

Iako se proceduralno generirane teksture dijele na više vrsta, ovdje ćemo pažnju prvenstveno usmjeriti prema generiranju staničnih tekstura. Stanične teksture se razlikuju od većine drugih tehnika generiranja proceduralnih tekstura u tome da se ne zasnivaju na funkcijama šuma, nego ih često koriste kao dodatak osnovnoj tehnici. Osnova algoritama za generiranje staničnih tekstura je bazirana na točkama značajki koje su raspršene u trodimenzionalnom prostoru. Te točke se onda koriste da podijele prostor u manje, slučajno posložene regije nazvane stanice. Stanice najčešće izgledaju kao gušterove ljuske ili različiti kamenčići. Iako je svaka stanica kao regija diskretna, osnovna funkcija za generiranje je kontinuirana u bilo kojoj točki prostora.

Problem generiranja staničnih tekstura je već dugi niz godina prisutan u računalnoj grafici. Reljefno teksturiranje i slične metode često postignu izgled detaljne geometrije površine bez da je zapravo stvore. No kada se gledište dovoljno približi, geometrijska struktura površine teksture postane očigledna, te se gubi željeni efekt. Ono što zapravo pokušavamo napraviti ovom metodom su površine pokrivene geometrijskim elementima. Te elemente modeliramo kao male trodimenzionalne

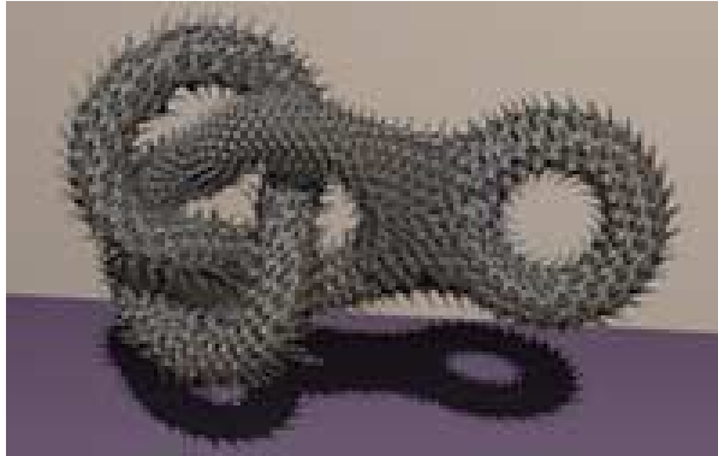
stanice ograničene da leže na površini modela.

3.1. Načini generiranja staničnih tekstura

3.1.1. Generiranje staničnih tekstura ponašajnim metodama

Fleischer u svom radu (Fleischer et al, 1995) generira stanične teksture kombinacijom različitih tehnika. Te tehnike su čestični sustavi, razvojne metode i reakcija-difuzija. Čestičnim sustavima su zadana specifična svojstva koja se odnose na ponašanje stanica. Ograničavajuće metode se potom koriste da se stanice generiraju unutar dozvoljene površine na površini modela. Stanice su potom generirane ponašajnim modelom koji je razvio Reynolds u svom radu (Reynolds, 1987). Na ponašanje stanica potom utječe proces nazvan 'reakcija-difuzija', koji potiče iz Turingovog rada iz 1952 (Turing, 1952). U Turingovom radu se raspravlja o staničnim svojstvima kao što su proizvodnja pigmenta koja su stalna kroz razvoj embrija, i ovise o određenim kemikalijama koje on naziva morfogeni. Određeni uzorci pigmentacije na životinjama su rezultat difuzije navedenih morfogena kroz tkivo, i kemijske reakcije koja se dogodila tijekom tog procesa.

Fleischerov rad tako generira stanice za koje korisnik može definirati početno stanje, potrebne varijable, izvanstanični utjecaj i raspon programa koji određuju njihovo ponašanje. Rezultati ove metode su vrlo dobri u postizanju organskog, prirodnog izgleda tekstura. Nažalost, unatoč impresivnim rezultatima, zbog velikog broja metoda računanja, te generiranja velike količine geometrije, a i zbog određenih ograničenja prilikom izrade programa za sjenčanje geometrije, još uvijek nije moguće implementirati ovu metodu u potpunosti pomoću programa za sjenčanje .



Slika 10. Primjer staničnih tekstura dobivenih Fleischerovim metodama.

3.1.2. Generiranje staničnih tekstura pomoću točki značajki

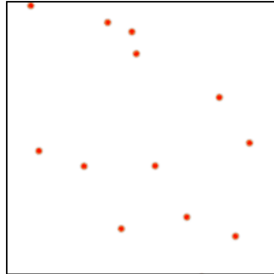
Worleyev šum predstavlja osnovnu, ali moćnu tehniku za generiranje staničnih tekstura. Ova metoda se bazira na točkama značajki (engl. *feature points*). Točke značajki su definirane kao slučajno raspršeni elementi u euklidskom 3-prostoru R^3. Za svaku točku unutar R^3, udaljenost x se postavlja kao udaljenost do najbliže točke značajki. Ovu metodu ćemo nazvati $F_{1.}$ Time vrijednost x definira jednadžbu polja za svaku točku prostora. Lokacije vrhova su tamo gdje je x jednako udaljen od dvije točke značajki, što rezultira Voronoievim dijagramom. Potom osnovna funkcija izračunava pozicije svih vrhova, te interpolira između vrhova i područja najbližih točkama značajki vraćajući skalarne vrijednosti. Te vrijednosti se tada mogu koristiti za određivanje boje ili informacije o reljefu.

Ovaj proces se može proširiti na nekoliko načina. Umjesto da izabiremo F_1 slučaj, možemo izabrati F_n slučaj (izabiremo n-tu najbližu točku značajki). Također možemo dodavati ili oduzimati različite funkcije kako bi dobili još različitih slučajeva. Još zanimljivije rezultate možemo dobiti kombiniranjem fraktala s osnovnom funkcijom.

Implementacija

Prva metoda implementacije bi bila generirati matricu željenih dimenzija te potom postaviti točke na slučajne koordinate. Nakon toga moramo za svaku točku u prostoru obaviti pretragu i naći točku koja je najbliže za zadanu funkciju udaljenosti.

Na početku moramo postaviti polje teksture, gdje su W i H širina, odnosno visina teksture, te potom postaviti točke značajki na slučajne koordinate unutar tog polja.



Slika 11. Inicijalno postavljanje točki značajki.

Kod funkcije za generiranje teksture:

```
void generate(double grid[W][H])
{
      double test, f1, f2, f3, f4, f5, f6;  // meduspremnici
      double value_buffer[W][H];

      inicijaliziraj tocke znacajki;
      inicijaliziraj value_buffer da sadrži fn
vrijednosti za svaki slikovni element;


      za(svaki slikovni element u mreži)
      {
            f1 = beskonacno;
            za(svaku tocku znacajki)
            {
test = udaljenost(trenutni slikovni element, trenutna tocka
znacajki);
                  ako(test<f1)
                  {
                        f2-f6 pridružimo vrijednosti f1-f5;
                        f1 = test;
                  }
                  inace ako(test<f2)
ponavljamo proces provjeravanja fn za svaki od fn
meduspremnika;
```

```
                }
                // f1-f6 sad sadrže vrijednosti za trenutni element
                // samo trebamo izabrati koju vrijednost ili
                // kombinaciju vrijednosti korististit
                // npr samo f1, f2-f1, sqrt(f3) itd.
                value_buffer = fn;
        }

        grid[W][H] = vrijednosti iz value_buffer;
}
```

Pseudokod za inicijalizaciju:

```
void init()
{
        za(svaku tocku znacajki)
        {
                dodijeli slucajnu x koordinatu;
                dodijeli slucajnu y koordinatu;
        }
}
```

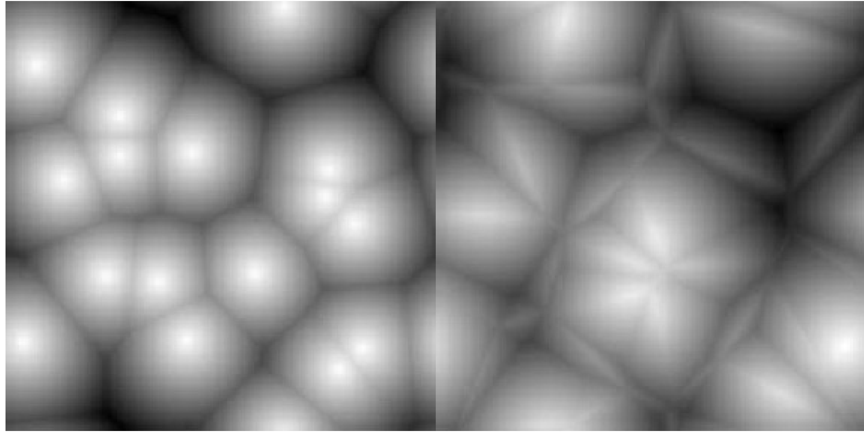
Funkciju udaljenosti također možemo proizvoljno definirati. Najčešće se koristi euklidska udaljenost

$d_{fp} = \sqrt{(Xfp - Xpix)^2 + (Yfp - Ypix)^2}$

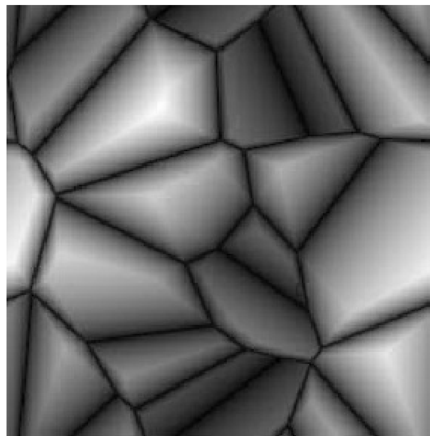
d_{fp} – rezultantna udaljenost

Xfp, Yfp – koordinate točke značajki

Xpix, Ypix – koordinate slikovnog elementa za koji računamo udaljenost



Slika 12. Primjer tekstura dobiven pomoću metode f1 (lijevo) i f2 (desno).

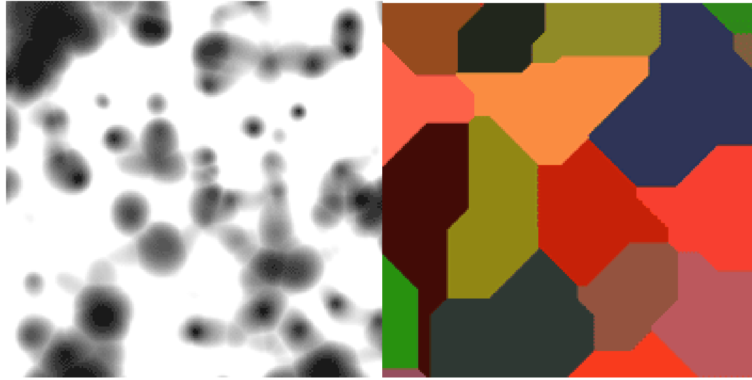


Slika 13. Tekstura dobivena oduzimanjem f2-f1.

Ali i ostale funkcije udaljenosti mogu imati zanimljiv efekt. Na slici imamo primjer teksture dobivene korištenjem 'city-block' funkcije udaljenosti.

Slika 14. F1 tekstura dobivena korištenjem 'city-block' udaljenosti.

Na kraju na dobivenu teksturu možemo primijeniti operacije po volji da bismo dobili neke zanimljive efekte, kao što vidimo na slici (**Slika 15**).

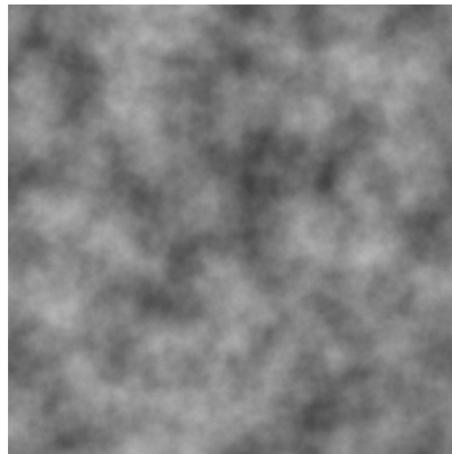


Slika 15. Tekstura dobivena multiplikacijom s fraktalnom funkcijom (lijevo) i izabiranjem boje prema pripadajućoj najbližoj točki značajke (desno).

3.1.3. Stohastičke metode

Perlinov šum

Perlinov šum je vrlo popularna metoda generiranja tekstura, u čijoj je osnovi Perlinova funkcija šuma u rešeci. Ova metoda se sastoji od stvaranja rešetke s pseudoslučajnim gradijentom za svaku točku unutar nje, a interpolacijom dobivamo ostale vrijednosti.



Slika 16. Primjer teksture dobivene pomoću Perlinovog šuma.

Najčešći postupak dobivanja ove vrste šuma pomoću računalnih programa je kombiniranje nekoliko glatkih funkcija šuma s različitim frekvencijama i amplitudama. Također, uobičajeno je da se za svaku sljedeću funkciju nakon prve uzima dvostruka frekvencija i upola manja amplituda od prethodne, iako se Perlinov šum drukčijih karakteristika može dobiti i korištenjem različitih postupaka za određivanje amplituda i

frekvencija.

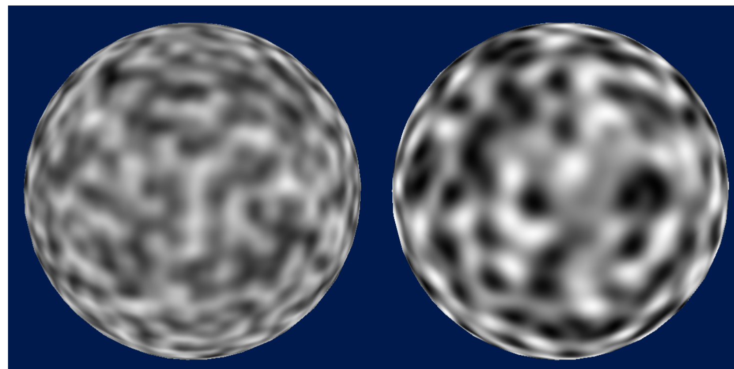
Simpleks šum

Simpleks šum je metoda za konstruiranje n-dimenzionalne funkcije šuma poput Perlinovog šuma, ali sa mnogo manjim procesorskim zahtjevima.

Prednosti simpleks šuma nad Perlinovim šumom:

- Simpleks šum ima manju kompleksnost računanja i koristi manje operacija množenja
- Složenost izračunavanja, koja je za simpleks šum $O(n^2)$ prilikom prelaska na više dimenzije (4D, 5D i više) raste puno sporije nego kod klasičnog Perlinovog šuma, koji ima složenost $O(2^n)$
- Simpleks šum je izotropan
- Simpleks šum ima kontinuirani gradijent koji se može jednostavno izračunati
- Simpleks šum je jednostavan za implementaciju na grafičkom sklopovlju

Glavna razlika u postupcima je da dok klasični Perlinov šum interpolira između vrijednosti iz krajnjih točaka okolne hipermreže (npr. sjever, jug, istok i zapad u dvodimenzionalnom prostoru), simpleks šum dijeli prostor u simplekse (npr. trokut u dvodimenzionalnom prostoru, tetraedar u trodimenzionalnom itd.) između kojih interpolira. Ovo smanjuje broj podataka s kojima treba računati, jer dok hiperkocka u N dimenzija ima 2^N vrhova, simpleks u N dimenzija ima samo N+1 vrh.

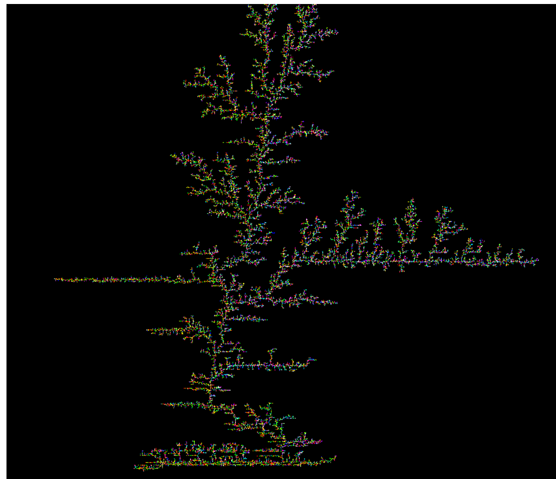


Slika 17. Usporedba Perlinovog šuma (lijevo) i simpleks šuma (desno).

3.1.4. Ostale manje poznate stohastičke tehnike

Difuzijsko limitirana agregacija

U difuzijsko limitiranoj agregaciji počinjemo s mrežom točaka koje su u jednom od dva stanja, nazovimo ih pozitivno i negativno. Potom uvedemo novi element koji se pojavi na slučajnom mjestu unutar mreže. Ako bilo koji od okolnih elemenata je označen kao pozitivan, onda je i novi element postavljen u pozitivno stanje. Ovim procesom dobivamo rezultate koji nalikuju na prirodan rast koralja.



Slika 18. Primjer slike dobivene difuzijsko limitiranom agregacijom.

Auto-regresija

Ovdje također počinjemo s mrežom točaka. Krećući od jednog kraja, postavljamo red točaka sa slučajnim vrijednostima intenziteta. Za točku u idućem redu, razmatramo susjedne ćelije iz prethodnog reda u kombinaciji sa slučajnim elementom da odredimo novi intenzitet. Nastavljamo s ovim postupkom sve dok se ne popuni cijela mreža.

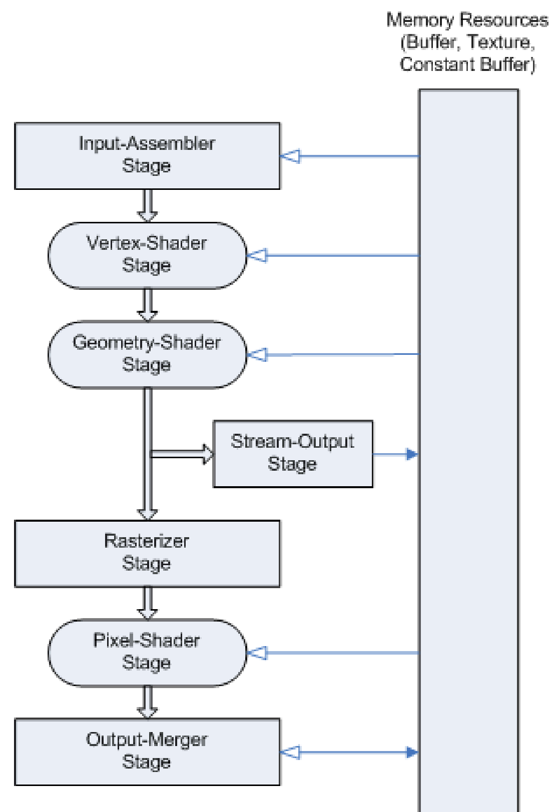
Stanični automati

Počinjemo s mrežom u kojoj su sve točke označene ili kao žive ili kao mrtve. Sustav pravila se potom primjenjuje na svaku točku, što rezultira novim rasporedom. Ta pravila se mogu primijeniti proizvoljni broj puta, da bi dobili željenu teksturu. Najpoznatiji sustav pravila za automat ovog tipa je Conwayeva igra života.

4. Programi za sjenčanje geometrije

Model programa za sjenčanje geometrije je prvi put predstavljen kao dio Unificiranog modela programa za sjenčanje (engl. *Unified Shader Model*) u *DirectX*-u 10. *Nvidia GeForce 8800* grafički procesori su bili prvi koji su imali sklopovsku podršku za izvođenje ovih programa. U *OpenGL* standard su uključeni od verzije 3.2, a iz prijašnjih verzija im se može pristupati pomoći *EXT_geometry_shader4* ekstenzije. Za razliku od ostalih programa za sjenčanje, programi za sjenčanje geometrije su opcionalni.

Izvršavaju se poslije programa za sjenčanje vrhova, a prije programa za sjenčanje fragmenata. Zbog toga, ako postoji potreba da se radi na vrhovima koji nisu već projekcijski transformirani, onda se i izračunavanje transformacija mora prebaciti u njih, i ukloniti iz programa za sjenčanje vrhova.



Slika 19. Pojednostavljeni prikaz cjevovoda u unificiranom modelu programa za sjenčanje.

Osnovna namjena programa za sjenčanje geometrije je stvaranje novih grafičkih primitiva ili uklanjanje već postojećih. Standardne primitive koje može primati su točka, linija i trokut, uz mogućnost i prosljeđivanja informacija o neposrednom susjedstvu.

Unificirani model programa za sjenčanje, poznat i pod nazivom model 4.0, koristi dosljedan skup instrukcija za sve tipove programa. Također, svi tipovi programa imaju gotovo iste mogućnosti: čitanje iz tekstura, međuspremnika podataka i obavljanje istog skupa aritmetičkih instrukcija. Međutim, skup instrukcija nije identičan između različitih tipova. Na primjer, samo programi za sjenčanje fragmenata mogu čitati teksture s implicitnim koordinatama, samo programi za sjenčanje geometrije mogu emitirati dodatne primitive itd.

Raniji modeli 1.x su koristili potpuno različite instrukcije za programe za sjenčanje vrhova i fragmenata, dok se sa kasnijim verzijama (2.x i 3.0) razlika sve više smanjivala, približavajući se tako sadašnjem unificiranom modelu.

Kada grafičko sklopovlje podržava unificirani model, onda je obično dizajnirano tako da procesorske jedinice za izvršavanje programa za sjenčanje mogu izvoditi bilo koji tip programa. Takva arhitektura se naziva unificirana arhitektura (engl. *Unified Shader Architecture*), te ima veliku prednost u brzini izvođenja nad sklopovljem koje ne implementira unificiranu arhitekturu. Prednost dolazi od mogućnosti dinamičkog organiziranja i balansiranja opterećenja između procesorskih jedinica. Na primjer, ako imamo nekoliko prolaza koji koriste samo programe za sjenčanje vrhova i fragmenata, tada možemo svim procesorskim jedinicama dodijeliti određene zadaće, iskorištavajući tako sve dostupne resurse, dok bi u slučaju zasebnih arhitektura dio procesorskih jedinica za izvođenje programa za sjenčanje geometrije ostao besposlen. Dobar primjer su također scene koje su iznimno zahtjevne za jednu vrstu programa, dok su ostali trivijalni, pa se u skladu tome raspoređuje broj jedinica koje rade na kojim vrstama programa.

Unatoč tome, sklopovlje ne mora nužno implementirati unificiranu arhitekturu da bi moglo podržavati unificirani model programa za sjenčanje i obrnuto.

4.1. Ograničenja i mogućnosti programa za sjenčanje geometrije

Prvo i najznačajnije ograničenje programa za sjenčanje geometrije je relativno nizak broj primitiva koje se mogu generirati. U prvim generacijama grafičkih kartica koje su to podržavale, taj broj je, ovisno o tipu izlaznih primitiva, bio vrlo nizak, ponegdje čak i ispod 64. No napretkom sklopovlja korištenog u grafičkim karticama, danas oni iznose nekoliko tisuća. Na primjer, grafička kartica *GeForce 9400M* na kojoj je izrađen ovaj rad, ima mogućnost generiranja do maksimalno 1024 izlazna trokuta za svaki ulazni trokut. Ta vrijednost za druge primitive može rasti ili padati u ovisnosti o tipu primitiva. Ako je izlazna primitiva tipa traka trokuta (engl. *triangle strip*), njihov maksimalan broj opada na 128, dok za točku maksimalan broj raste na 4096.

Uzrok tome je ograničena količina memorije u međuspremniku koja se može proslijediti dalje, te zbog toga broj maksimalnih primitiva ovisi o tome koliko memorije koji tip primitive zauzima.

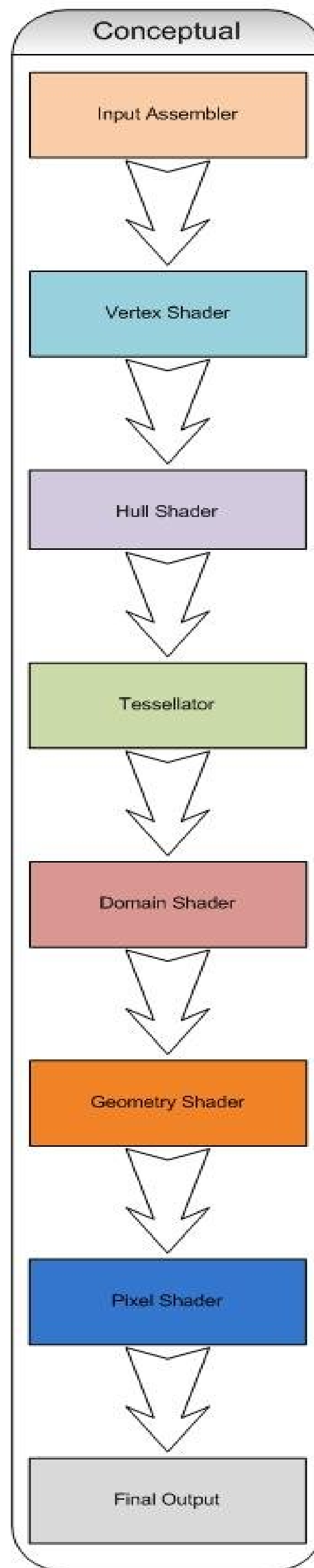
Sljedeći problem je nedostupnost informacija o ostatku modela te o netom generiranoj geometriji. Naime, predviđeni način prosljeđivanja informacija o susjedima za bilo koji tip primitive je ograničen samo na neposredne susjede, a i tada mi moramo u programu generirati i proslijediti tu informaciju. Tako nešto često zahtjeva ili mijenjanje načina učitavanja i prikazivanja modela u memoriji, ili dinamičko generiranje informacija o susjednim vrhovima, što može usporiti program, a i stavlja više opterećenja na procesor. Također, do informacija o novonastaloj geometriji ne možemo doći do idućeg prolaza.

Zbog ovih gore navedenih razloga, implementacija metode koju predlaže Fleischer u svom radu (Fleischer et al, 1995) bi bila prekompleksna i neefikasna. Prvenstveno zbog potrebe da stanica ima informacije o svim svojim susjednim stanicama i njihovom trenutačnom stupnju i načinu razvoja, a potom i zbog kompleksnosti novonastale geometrije.

Iako su se unatoč ovim problemima programi za sjenčanje geometrije uz neke pomoćne i zaobilazne metode koristili za vrlo širok raspon problema, uvođenje novih

vrsta programa za sjenčanje i teselaciju u *DirectX*-u 11 eliminira potrebu korištenja programa za sjenčanje geometrije kao programa čija je primarna namjena generiranje nove geometrije. Ovime se problem generiranja nove geometrije prebacuje na programe za sjenčanje ljuske (engl. *hull shaders*), programe za sjenčanje područja (engl. *domain shaders*) i programe za teselaciju (engl. *tesselator*).

U tom slučaju najčešći tip uporabe programa za sjenčanje geometrije neće biti generiranje, već uklanjanje geometrije za npr. mijenjanje razine detalja prikaza. Generiranje geometrije u programima za sjenčanje geometrije će biti ograničeno na specifične primjene za koje nisu potrebne dodatne informacije, poput generiranja točkastih sličica (engl. *point sprite*).



Slika 20. Konceptualni prikaz Direct3D 11 cjevovoda.

4.2. Primjena programa za sjenčanje geometrije

Zbog generičkog skupa instrukcija koji je dio unificiranog modela, način mogućih primjena programa za sjenčanje geometrije je vrlo raznolik i proteže se u rangu od osnovnih problema poput razine detalja prikaza do egzotičnih primjena koje nemaju nužno veze sa samom grafikom poput modeliranja ponašanja nekih likova unutar igre. Dobar primjer je demo koji je napravio ATI za *HD 4800* seriju kartica, naziva *Froblins*, u kojem ogromnim brojem “*froblina”* koji kopaju kamen, nose ga i spremaju, jedu kada nađu gljive i na kraju kad se umore odlaze na spavanje, upravljaju programi za sjenčanje geometrije. No mi ćemo ovdje ukratko opisati samo najbitnije primjene programa za sjenčanje geometrije.

4.2.1. Razina detalja prikaza

U računalnoj grafici, ovaj pojam podrazumijeva mijenjanje kompleksnosti prikaza trodimenzionalnih objekata u ovisnosti o udaljenosti od točke gledišta, važnosti objekta, brzini ili o nekom drugom parametru. Koristi se da poboljša performanse prikaza smanjujući opterećenje grafičkog sklopovlja, ponajprije onog zaduženog za transformaciju vrhova. Smanjena kvaliteta modela se često ne primjećuje zbog puno manjeg utjecaja objekta na izgled scene kad je udaljen ili se brzo kreće.

Iako se u većini slučajeva primjenjuje samo na geometriju, ova tehnika se može generalizirati. Nedavno su se počele implementirati i tehnike za upravljanje programima za sjenčanje fragmenata kontrolirajući kompleksnost prikaza.

4.2.2. Odbacivanje po projekcijskom volumenu

Odbacivanje po projekcijskom volumenu (engl. *frustum-culling*) koristi činjenicu da su svi poligoni izvan projekcionog volumena nevidljivi, tj. ono što kamera trenutno ne gleda, nije potrebno iscrtavati, te se ti poligoni odbace u programu za sjenčanje geometrije, stvarajući tako manje posla za programe za sjenčanje fragmenata.

Često se zna dogoditi da neki objekt leži na granici volumena. Takvi objekti se podijele na dijelove duž te granice, te se tada odbacuju samo oni dijelovi koji su van granica.

4.2.3. Preslikavanje s kocke u jednom prolazu

Osnovna ideja prilikom preslikavanja s kocke u jednom prolazu je da programu za sjenčanje geometrije proslijedimo šest matrica projekcije, i za svaku primitivu modela koji prikazujemo, generiramo šest drugih primitiva koje su projicirane svaka na svoju stranu kocke.

Ovim načinom izbjegli smo potrebu šesterostrukog prolaza kroz scenu, koji je dosad bio uobičajeni način preslikavanja s kocke, a samim time i smanjili broj poziva API funkcija šest puta.

Postoje optimizacije gore navedenog algoritma, koje ga još dodatno ubrzavaju koristeći instanciranje geometrije, no postupak u osnovi ostaje isti.

5. Implementacija generiranja stanične geometrije i tekstura u stvarnom vremenu

Kao demonstraciju navedenih tehnika i mogućnosti primjene programa za sjenčanje geometrije, za potrebe ovog rada implementirana su dva programa. Kao osnovno polazište su izabrane metode koje predlaže Fleischer u svom radu (Fleischer et al, 1995), uz modifikacije koje su uzimale u obzir ograničenja programa za sjenčanje geometrije, te potrebu da se program izvodi u realnom vremenu.

Za razliku od predloženih metoda, koje samo za generiranje potrebne geometrije trebaju od od nekoliko sekundi pa do nekoliko sati, ovisno o tipu stanica, metode razvijene za potrebu ovog rada moraju generirati i izračunati projekciju cijele scene unutar 40-100ms. Zbog toga je kompleksnost nove stanične geometrije koja se generira za zadani model smanjena na razinu koja omogućava izvođenje u realnom vremenu na današnjem grafičkom sklopovlju.

5.1. Generiranje bodlji

Originalna ideja je vrlo jednostavna, izabrati slučajne nakupine trokuta na površini modela kojima susjedi nemaju već generirane bodlje, te za svaku od njih izvršiti program koji simulira rast bodlji sa određenim parametrima.

Prva bitnija promjena koja je napravljena s obzirom na originalnu zamisao je da se ne izabiru pojedine primitive za koje se program izvršava, već se izvršava za sve primitive. Razlog tomu je da dohvaćanje informacije o susjedima i već generiranoj geometriji u tom prolazu kroz scenu zahtjeva značajnije promjene u samom učitavanju i prikazu modela u memoriji, te implementaciju programa za sjenčanje geometrije koja podržava povratnu vezu transformacija (*engl. transform feedback*) što nadilazi okvire ovog rada.

Druga značajna promjena je u generiranju teksture za novu geometriju. Dok se u originalnoj metodi boja pojedinih vrhova generira ovisno o parametrima i proteklom vremenu simulacije ponašanja stanica, ovdje se uzima modifikacija Worleyeve

metode generiranja staničnih tekstura. Dakle, na generiranu Worleyevu teksturu, dodajemo ili oduzimamo dodatni intenzitet koji ovisi o udaljenosti ostalih vrhova početnog poligona od vrha bodlje. Da bi se naglasila razlika, izabiremo boju koja je različita od boje korištene za prvobitno generiranu teksturu.

Slika 21. Originalni model iscrtan bez dodatnih programa za sjenčanje geometrije. Kugla je teksturirana običnom Worleyevom teksturom.

Slika 22. Isti model sa uključenim programima za sjenčanje geometrije.

5.1.1. Program za sjenčanje vrhova

```
varying vec2 vTexCoord;
varying vec3 vNormal;
varying vec3 vViewVec;

void main(void)
{
      vNormal = gl_Normal;
      vTexCoord = vec2(gl_MultiTexCoord0);
      gl_Position = gl_Vertex;
}
```

Kao što vidimo, program za sjenčanje vrhova je trivijalan. Pošto su nam u programu za sjenčanje geometrije potrebne koordinate nad kojima nije izvršena transformacija, samo prosljeđujemo potrebne varijable, a to su vektor normale, koordinate teksture i koordinate vrha koji se obrađuje.

5.1.2. Program za sjenčanje geometrije

```
#version 130
#extension GL_EXT_geometry_shader4: enable

vec3 V0, V01, V02;

uniform float height;

in vec3 vNormal[];
in vec3 vViewVec[];
in vec2 vTexCoord[];

out vec2 ovTexCoord;
out vec3 ovViewVec;
out vec3 ovNormal;
out vec4 ovVertColor;

void main()
{
  int i, j;

  V01 = (gl_PositionIn[1] - gl_PositionIn[0]).xyz;
  V02 = (gl_PositionIn[2] - gl_PositionIn[0]).xyz;
  V0 = gl_PositionIn[0].xyz;

  vec4
centr=(gl_PositionIn[0]+gl_PositionIn[1]+gl_PositionIn[2])/3;

  centr.x+=1;
  vec2
centrTexCoord=(vTexCoord[0]+vTexCoord[1]+vTexCoord[2])/3;

  for(i=0;i<gl_VerticesIn;i++)
    for(j=0;j<gl_VerticesIn;j++)
if(i!=j)
      {
  ovVertColor = vec4(0.0, 0.0, fract(length(gl_PositionIn[i]-
centr))*0.1, 1.0);

        ovTexCoord = vTexCoord[i];
```

```
        ovNormal = normalize(gl_NormalMatrix * vNormal[i]);
  ovViewVec = -normalize(vec3(
gl_ModelViewMatrix * gl_PositionIn[i]));

        gl_Position =
gl_ModelViewProjectionMatrix*gl_PositionIn[i];
        EmitVertex();

  ovVertColor = vec4(0.0, 0.0, fract(length(gl_PositionIn[j]-
centr))*0.1, 1.0);

        ovTexCoord = vTexCoord[j];
        ovNormal = normalize(gl_NormalMatrix * vNormal[j]);
        ovViewVec = -normalize(vec3(
gl_ModelViewMatrix * gl_PositionIn[j]));

        gl_Position =
gl_ModelViewProjectionMatrix*gl_PositionIn[j];
        EmitVertex();

        ovTexCoord = centrTexCoord;
        vec3 centrNormal = normalize(cross(V01, V02));
        float area = abs(length(cross(V01, V02)))/2;

        ovVertColor = vec4(0.0, 0.0, 0.1, 1.0);
        centr.w = height;
        ovNormal = normalize(gl_NormalMatrix * centrNormal);
        ovViewVec = -normalize(vec3(gl_ModelViewMatrix *
centr));
        gl_Position = gl_ModelViewProjectionMatrix * centr;
        EmitVertex();

      EndPrimitive();
      }
}
```

Na početku svakog programa za sjenčanje geometrije moramo definirati koju verziju *GLSL* standarda koristimo i koje su nam ekstenzije potrebne, inače se program neće moći uspješno prevesti. Ovdje je to verzija *GLSL* jezika 1.30, a

potrebna ekstenzija je *GL_EXT_geometry_shader4*. Idući korak je deklariranje atributnih, promjenjivih i uniformnih varijabli.

Može se primjetiti da za razliku od *GLSL* verzije 1.20, koja je korištena za programe za sjenčanje fragmenata i vrhova, verzija 1.30 zahtjeva da za sve promijenjive varijable točno odredimo jesu li izlazne ili ulazne pomoću ključe riječi `in` odnosno `out`. Također, sve ulazne varijable moramo deklarirati kao polje, jer program ne zna unaprijed koji tip primitiva će dobivati za vrijeme izvršenja i koliko vrhova će sadržavati pojedina primitiva. Kao ulazne primitive deklariramo sve varijable koje će nam proslijediti program za sjenčanje vrhova, a to su normala vrha, koordinate vrha i koordinate teksture. Iste te varijable, samo pod drugim imenom deklariramo kao izlazne, jer će nam biti potrebne prilikom izračunavanja osvijetljenja fragmenta. Uz njih, imamo još i dodatnu izlaznu varijablu, `ovVertColor` koja služi za već spomenuto modificiranje Worleyeve teksture, te nam je potrebna u programu za sjenčanje fragmenata. Atributne varijable `V0,` `V01` i `V02` su redom: vektor iz ishodišta do prvog vrha trokuta, vektor iz prvog prema drugom vrhu trokuta i vektor iz prvog prema trećem vrhu trokuta.

Na početku glavnog dijela programa izračunamo težište trokuta, jer je to početna točka koju ćemo koristiti za izvorište rasta bodlje. Broj vrhova ulazne primitive možemo saznati iz globalne varijable `gl_VerticesIn` koja je definirana na razini *GLSL* standarda, te potom prolazimo svaki neuređeni par različitih vrhova primitive, i računamo koordinate novih poligona koji "rastu" iz trenutnog para vrhova u željenom smjeru. Pošto projekcija nije obavljena u programu za sjenčanje vrhova, nakon izračuna željenih koordinata moramo ih pomnožiti sa `gl_ModelViewProjectionMatrix`.

Nakon izračuna svih potrebnih varijabli za novi vrh, pomoću funkcije `EmitVertex()` ih prosljeđujemo dalje, a kao oznaku da smo završili jednu primitivu pozivamo `EndPrimitive()`.

5.1.3. Program za sjenčanje fragmenata

```
uniform sampler2D Texture0;
varying vec2 ovTexCoord;
varying vec3 ovNormal;
```

```
varying vec3 ovViewVec;
varying vec4 ovVertColor;

void main(void)
{
  vec3 ovLightVec = vec3(1.0, -1.0, 2.0);

  vec3 vReflect = normalize(2.0 * dot(ovNormal, ovLightVec) *
ovNormal - ovLightVec);

  vec4 LightColor = vec4(0.8, 0.8, 0.8, 1.0);
  vec4 AmbientColor = LightColor * 0.4;
  vec4 DiffuseColor = LightColor * 0.6 * max(0.0, dot(ovNormal,
ovLightVec));

  vec4 SpecularColor = LightColor * 0.3 * pow(max(0.0,
dot(vReflect, ovViewVec)), 0.8);

  vec4 FinalColor = (AmbientColor + DiffuseColor) *
(texture2D(Texture0, ovTexCoord)+ovVertColor) + SpecularColor;

  gl_FragColor = FinalColor;
}
```

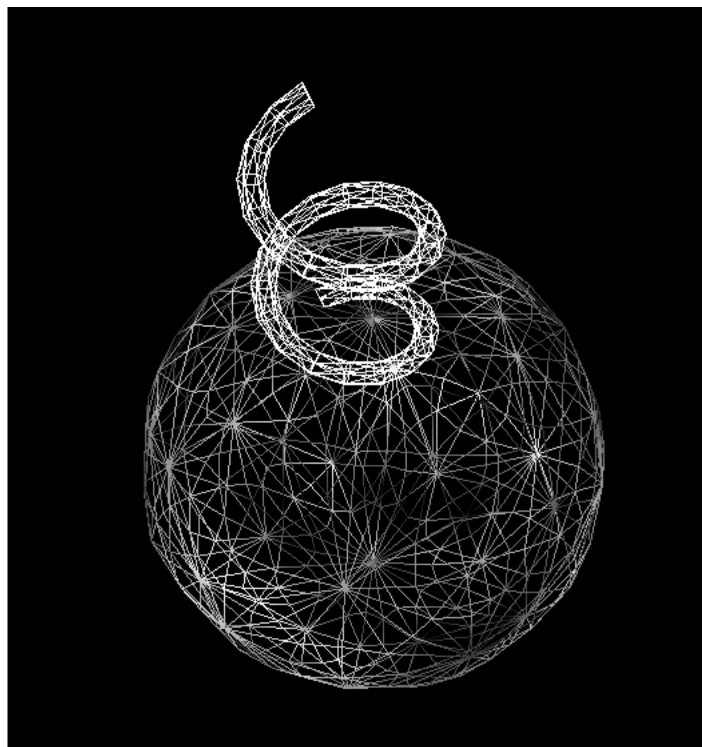
Program za sjenčanje fragmenata implementira postupak osvijetljenja po slikovnom elementu. Modifikacije u odnosu na već objašnjeni program iz prvog dijela rada su minimalne, tako da postupak nije potrebno dodatno objašnjavati. Osnovna promjena je da se boja teksture prije primjene ambijentalnog i difuznog osvijetljenja modificira pomoću varijable `ovVertColor` koja je generirana u programu za sjenčanje geometrije.

Koeficijenti ambijentalnog k_a, difuznog k_d i zrcalnog osvijetljenja k_s su redom 0.4, 0.6 i 0.3.

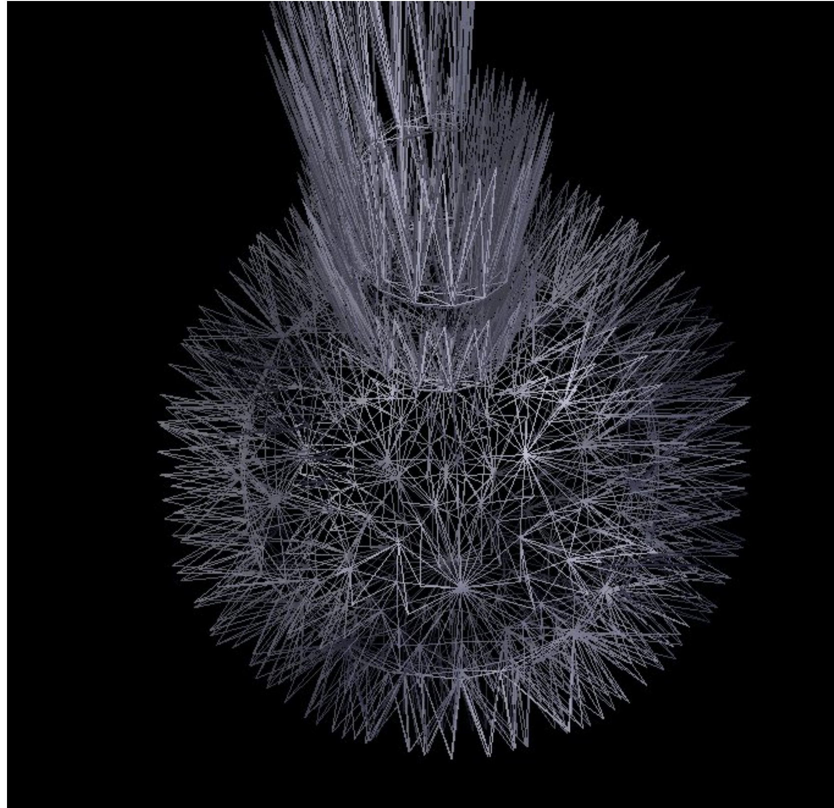
5.1.4. Žičani prikaz

U program je također implementirana mogućnost žičanog prikaza, da bi se

mogla detaljnije primjetiti sama razlika u generiranoj geometriji. Na slici **Slika 23** vidimo prikaz samog modela, sa isključenim programima za sjenčanje i bez dodatne geometrije, dok na slici Slika **24** vidimo svu novogeneriranu geometriju.



Slika 23. Žičani prikaz modela bez programa za sjenčanje.



Slika 24. Žičani prikaz modela s uključenim programom za sjenčanje geometrije.

5.1.5. Performanse

Testno računalo se sastojalo od *Intel Core 2 Duo 2.00GHz* procesora, *1GB RAM*-a i *GeForce 9400M* grafičke kartice, što spada u nižu klasu računala koja mogu izvršavati ovaj program i za posljedicu ima sporije rezultate. Na testnom računalu program za generiranje bodlji postigao je sljedeće rezultate:

Tablica 5.1 Performanse programa za generiranje bodlji.

Model	Originalni broj poligona	Br. poligona nakon programa za sjenčanje	Rezolucija prikaza	Okvira po sekundi	Okvira po sekundi bez programa za sjenčanje
Sphere.ms3d	1116	4464	1024x768	57	85
Teddy_low.ms3d	2814	11256	1024x768	26	85

Teddy.ms3d	19152	76608	1024x768	13	85

Oba modela se bez programa za sjenčanje izvode maksimalno ograničenom brzinom, dok primjećujemo da broj iscrtanih okvira po sekundi pada proporcionalno broju generiranih poligona.

Možemo primjetiti da se usporenje ne događa samo zbog novonastalih poligona, već zbog izvođenja programa za sjenčanje geometrije. To se uočava na modelu *Teddy.ms3d* koji originalno ima 19152 poligona, i sam model se iscrtava maksimalnom brzinom, dok se *Teddy_low.ms3d* koji nakon generiranja bodlji još uvijek ima skoro dvostruko manji broj poligona izvršava puno sporije.

Slika 25. Teddy_low.ms3d s programima za sjenčanje.

5.2. Generiranje dlake

Originalni Fleischerov prijedlog postupka pomoću kojeg se generira dlaka se razlikuje od ostalih postupaka samo u programima koji se koriste za simulaciju rasta stanica. Razlika je ponovno u tome da se „rast“ novih stanica ne simulira izvodeći programe koji oponašaju ponašanje stanice, već da se cijela stanica generira u jednom prolazu.

Pošto je u ovom slučaju stanica oblika linije, kao zamjenu sam izabrao Bezierove krivulje. Bezierove krivulje su odabrane jer iako nisu računski prezahtjevne, još uvijek omogućavaju prilično fleksibilno modeliranje krivulje, što nam pomaže u postizanju prirodnog izgleda zakrivljene dlake.

5.2.1. Bezierove krivulje

Bezierove krivulje su parametarske krivulje koje se često koriste u računalnoj grafici i ostalim srodnim poljima računalnih znanosti. Generalizacija Bezierovih krivulja na više dimenzije su Bezierove površine.

U vektorskoj grafici, Bezierove krivulje se koriste za modeliranje glatkih krivulja kojima možemo proizvoljno mijenjati veličinu. Često se koriste i u vremenskoj domeni, naročito za animacije i dizajniranje sučelja. Na primjer, Bezierova krivulja se može koristiti da odredi brzinu objekta u različitim trenucima u vremenu, što kao posljedicu ima rezultat prirodnijeg gibanja pri micanju objekta s jednog mjesta na drugo, nego da samo postavimo brzinu na fiksni iznos.

Popularizirao ih je 1962. francuski inžinjer Pierre Bezier, koji ih je koristio za dizajniranje automobila. Razvijene su 1959. kada je Paul de Casteljau pronašao Casteljauov algoritam koji je rekurzivnom metodom izračunavao vrijednosti polinoma u Bernsteinovom obliku. Iako je algoritam sporiji nego izravni pristup, numerički je stabilniji.

5.2.1.1 Opći slučaj

Ako imamo točke P_0, P_1, ..., P_n onda se Bezierova krivulja n-tog stupnja može generalizirati na sljedeći način:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i, t \in [0,1]$$

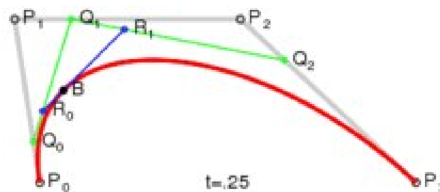
Što se može napisati kao:

$$B(t) = (1-t)^n P_0 + \binom{n}{1}(1-t)^{n-1} P_1 + ... + \binom{n}{n-1}(1-t)t^{n-1} P_{n-1} + t^n P_n, t \in [0,1]$$

5.2.1.2 Kubična Bezierova krivulja

Za potrebe ovog programa, izabrana je kubična Bezierova krivulja. Četiri kontrolne točke P_0, P_1, P_2 i P_3 u ravnini ili trodimenzionalnom prostoru definiraju kubičnu Bezierovu krivulju.

Krivulja počinje iz točke P_0 ide prema točki P_1 i dospijeva do P_3 iz smjera P_2. Obično, ne prolazi kroz točke P_1 i P_2; ove točke su jedino za određivanje smjera. Udaljenost između P_0 i P_1 određuje koliko dugo se krivulja kreće prema točki P_2 prije nego skrene prema točki P_3.



Slika 26. Konstrukcija kubične Bezierove krivulje.

Parametarski oblik krivulje je:

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3$$

5.2.1.3 Implementacija u GLSL-u

Funkcije za izračunavanje krivulje implementirane su kao dio programa za sjenčanja geometrije, jer za svaku ulaznu primitivu moramo generirati jednu ili više Bezierovih krivulja. Postupak je podijeljen u tri logičke cjeline.

Prva je implementirana u funkciji `evaluateBezierPosition` koja kao parametre prima četverodimenzionalni vektor koordinata kontrolnih točaka `v` i vrijeme `t`, a vraća koordinate izračunate točke. Sama implementacija funkcije je sljedeća:

```
vec3 evaluateBezierPosition(vec3 v[4], float t)
{
	float omt = 1.0 - t;
	float b0 = omt*omt*omt;
	float b1 = 3.0*t*omt*omt;
	float b2 = 3.0*t*t*omt;
	float b3 = t*t*t;
	return b0*v[0] + b1*v[1] + b2*v[2] + b3*v[3];
}
```

Druga funkcija koju smo implementirali je funkcija za računanje tangente u određenoj točci Bezierove krivulje. Naziv funkcije je `evaluateBezierTangent`, prima iste parametre kao `evaluateBezierPosition`, a vraća vektor tangente. Računanje vektora smjera tangente je jednostavno računanje prve derivacije:

```
vec3 evaluateBezierTangent(vec3 v[4], float t)
{
	float omt = 1.0 - t;
	float b0 = -3.0*omt*omt;
	float b1 = 3.0*omt*omt - 6.0*t*omt;
	float b2 = 6.0*t*omt - 3.0*t*t;
	float b3 = 3.0*t*t;
	return b0*v[0] + b1*v[1] + b2*v[2] + b3*v[3];
}
```

Zadnja funkcija je emit_bezier koja dijeli krivulju na željeni broj segmenata, u ovom slučaju 8, i potom šalje segmente funkciji za transformaciju koja ih proslijeđuje programu za sjenčanje fragmenata. Parametri koje prima su kontrolne točke.

```
void emit_bezier(vec3 cv[4])
{
	int segments = 8;
	int i;
	float u;
	vec3 p;
	vec3 t;

	for(i=0;i<segments;i++)
	{
		u = i / (segments-1);
		p = evaluateBezierPosition(cv, u);
		t = normalize(evaluateBezierTangent(cv, u));
		emitVertexTransform(p, t, u);
	}
	EndPrimitive();
}
```

5.2.2. Program za sjenčanje vrhova

Pošto su nam i ovdje potrebne netransformirane koordinate vrhova, program za sjenčanje vrhova je trivijalan i ima istu funkciju kao u programu za generiranje bodlji, te se ovdje neće posebno opisivati ili navoditi. Osnovna namjena mu je samo proslijeđivanje potrebnih parametara iz programa za sjenčanje vrhova programu za sjenčanje geometrije.

5.2.3. Program za sjenčanje geometrije

Dijelovi programa koji se bave samim izračunavanjem Bezierovih krivulja su već navedeni, stoga ćemo ovdje samo navesti i objasniti ostatak programa.

```
#version 130
#extension GL_EXT_geometry_shader4: enable

uniform float hair_len;
uniform float dir_x;
uniform float dir_y;
uniform float dir_z;

in vec3 vNormal[];
```

```
in vec2 vTexCoord[];

out vec2 ovTexCoord;
out vec3 ovViewVec;
out vec3 ovNormal;
out vec4 ovVertColor;

void emitVertexTransform(vec3 position, vec3 normal, float u)
{
  gl_Position=gl_ModelViewProjectionMatrix*vec4(position, 1.0);
  ovNormal=normalize(gl_NormalMatrix * normal);
  ovViewVec=-
normalize(vec3(gl_ModelViewMatrix*vec4(position,1.0)));
  ovTexCoord = (vTexCoord[0] + vTexCoord[1] + vTexCoord[2]) /
3;

  EmitVertex();
}
// ostale funkcije za računanje Bezierovih krivulja su
izostavljene
// . . .

void main()
{
  vec4 rand;
  rand.x =
fract(sin(dot(vTexCoord[0].xy,
vec2(12.9898,78.233)))*43758.5453);

  rand.y =
fract(sin(dot(vTexCoord[1].xy,
vec2(12.9898,78.233)))*43758.5453);

  rand.z =
fract(sin(dot(vTexCoord[2].xy,
vec2(12.9898,78.233)))*43758.5453);

  rand.q =
fract(sin(dot(gl_PositionIn[0].xy,vec2(12.9898,78.233)))*43758.
5453);
```

```
  float a = rand.x;
  float b = rand.y;
  if(a+b>1.0)
  {
      a = 1.0 - a;
      b = 1.0 - b;
  }
  float c = 1 - a - b;

  vec3 pos =
gl_PositionIn[0].xyz*a+gl_PositionIn[1].xyz*b
+gl_PositionIn[2].xyz*c;

  vec3 normal = vNormal[0]*a + vNormal[1]*b + vNormal[2]*c;
  vec3 tangent = vec3(dir_x, dir_y, dir_z);  // smjer rasta

  float length = hair_len + rand.z*hair_len;

  vec3 cv[4];
  cv[0] = pos;
  cv[1] = pos + normal*length*0.5;
  cv[2] = pos + normal*length + tangent*length*0.5;
  cv[3] = pos + normal*length + tangent*length;

  ovVertColor = vec4(length/2, length/2, length/2, 1.0);
  emit_bezier(cv);
}
```

Na početku main funkcije prvo generiramo slučajni četverodimenzionalni vektor koji će nam poslije poslužiti za za određivanje duljine i kontrolnih točaka krivulje koje ćemo generirati za dobivenu primitivu. Kako u GLSL jeziku ne postoji ugrađena funkcija koja vraća slučajne brojeve, pseudoslučajne brojeve moramo generirati ručno, vodeći pri tom računa o performansama programa.

Taj generirani vektor nam služi i za određivanje početne pozicije rasta dlake unutar trokuta. Iz njega pridružimo prve dvije slučajne vrijednosti varijablama `a` i `b`, te potom izračunamo varijablu `c` tako da je zbroj `a, b` i `c` jednak 1. One će određivati koliku težinu će imati koji od vrhova pri određivanju početne točke rasta. U prvoj verziji programa koristilo se težište trokuta, no tada su rezultati izgledali pretjerano

pravilni, što u prirodi nije slučaj.

Normalu iz početne točke rasta također računamo kao težinsku sredinu normala iz ostala tri vrha. Vektor smjera rasta se određuje preko uniformnih varijabli koje se proslijeđuju iz glavnog programa za vrijeme izvođenja, te ih je moguće po potrebi mijenjati. Minimalna dužina dlake se također zadaje preko uniformne varijable `hair_len`, i njoj se dodaje još neki slučajni dio, tako da sve dlake ne budu jednako dugačke.

Kontrolne točke se računaju na sljedeći način:

- prva točka se postavlja na već prije izračunatu pozicija početka rasta dlake
- druga je pomaknuta u smjeru vektora normale za polovični iznos dužine dlake u odnosu na prvu točku
- treća je pomaknuta prvo u smjeru vektora normale za iznos cijele dužine dlake, te potom u smjeru unaprijed određenog rasta dlake za polovični iznos dužine dlake u odnosu na prvu točku
- četvrta i zadnja je pomaknuta u smjeru vektora normale za cijeli iznos dužine dlake, te potom u smjeru rasta također za cijeli iznos dužine dlake u odnosu na početnu točku

Nakon toga, jednostavno pozovemo funkciju koja će izračunati broj potrebnih točaka u krivulji, te ih potom transformirati u prostor pogleda, te zajedno sa ostalim potrebnim varijablama proslijediti programu za sjenčanje fragmenata.

Dodatno, na samu boju dlake još utječe i dužina dlake, neovisno o tome da li je sam model teksturiran ili ne. Tako su dlake koje su duže automatski i svjetlije, dok su one kraće tamnije. Taj dodatni intenzitet se u programu za sjenčanje fragmenata primjenjuje na boju teksture u točci rasta dlake.

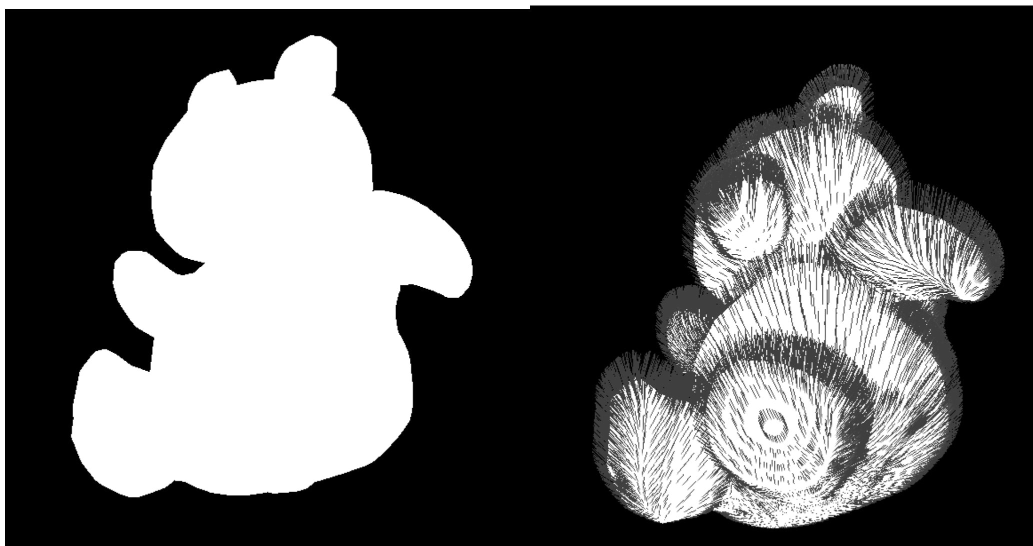
5.2.4. Program za sjenčanje fragmenata

Kao i u slučaju programa za sjenčanje vrhova, program za sjenčanje fragmenata je vrlo sličan programu koji se već koristio pri generiranju bodlji, te ga zbog uštede prostora ovdje nećemo navoditi i posebno opisivati. Dovoljno je samo

reći da također računa standardno osvijetljenje po pikselu uračunavajući dodatni intenzitet koji je proslijeđen iz programa za sjenčanje geometrije. Za sve ostale detalje se može pogledati priloženi izvorni kod svih programa koji su razvijeni za svrhu ovog rada.

5.2.5. Rezultati

Na slici Slika **27** možemo vidjeti usporedbu rezultata sa samo pokrenutim programom koji učitava i iscrtava model, i identičnog programa sa uključenim programima za sjenčanje.



Slika 27. Prolaz bez programa za sjenčanje (lijevo) i sa programima za sjenčanje (desno).

5.2.6. Performanse

Na testnom računalu program za generiranje dlake postigao je sljedeće rezultate:

Tablica 5.2 Rezultati programa za generiranje dlake

Model	Originalni broj poligona	Originalni broj vrhova	Broj vrhova nakon programa	Rezolucija prikaza	Okvira po sekundi	Okvira po sekundi bez programa

			za sjenčanje			za sjenčanje
Sphere.ms3d	1116	617	9545	1024x768	85	85
Teddy.ms3d	19152	9578	162794	1024x768	17	85

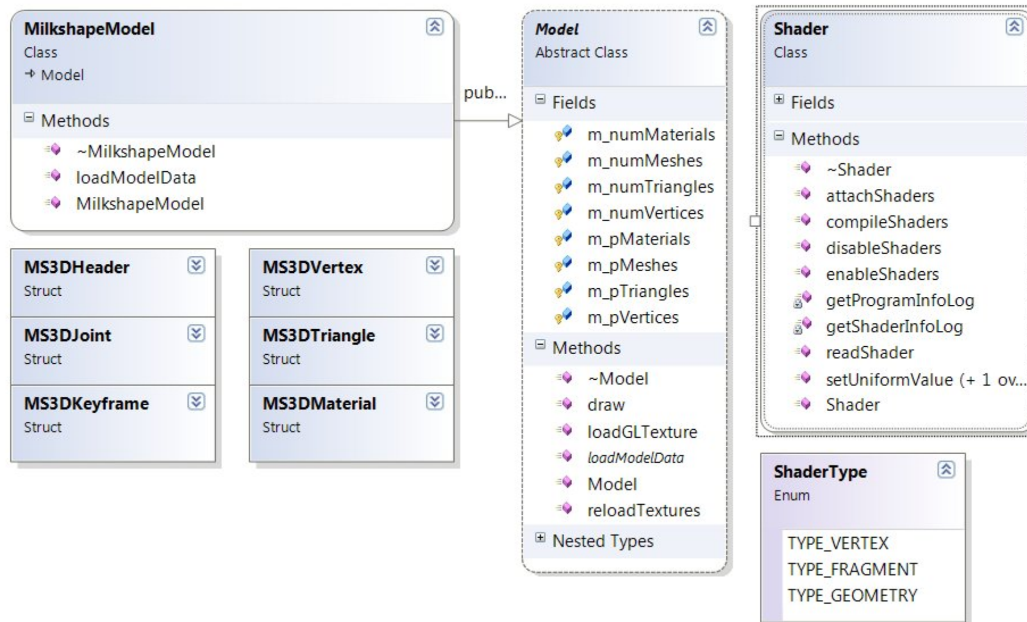
Bitno je primjetiti da na modelu sa velikim brojem poligona brzina izvođenja pada na 17 okvira po sekundi što se već primjeti u smanjenoj glatkoći izvođenja i odzivu programa na kontrole. Tako da nije ni čudno što je orginalna metoda zahtjevala i po nekoliko sati samo za generiranje modela, dok ovdje i generiramo i napravimo projekciju scene svakih 60ms.

Na modelu sa manjim brojem poligona se primjeti i da sa generiranom dodatnom geometrijom broj okvira po sekundi ostaje na maksimalnom ograničenom iznosu, ali porastom broja poligona, osmerostruko brže raste i broj novonastalih vrhova, pa tako već i manje promjene broja poligona mogu dovesti do značajnijih razlika u brzini izvođenja.

5.3. Glavni program

Zbog određenih zahtjeva ovog dijela rada u odnosu na dio koji se bavio samo programima za sjenčanje vrhova i fragmenata, za ovaj dio je napravljen zaseban i puno kompleksniji glavni program. Također, implementirana je mogućnost učitavanja *Milkshape3D* modela. Kako ovaj program sadrži mnogo veću količinu koda, nastala je potreba da ga se napravi u duhu objektno orijentiranog programiranja zbog lakšeg snalaženja.

Druga bitnija razlika je da je ovaj program implementiran koristeći samo *Win32 API* i nativne *OpenGL* 3.2 funkcije, dakle uklonili smo potrebu za zastarjelim *GLUT* i *GLEW* bibliotekama, te *ARB* ekstenzijama.



Slika 28. Dijagram razreda projekta.

Razred *Model* implementiran je kao apstraktni razred, što nam olakšava posao ako budemo imali potrebu dodavati učitavanje različitih tipova modela. Ako želimo napraviti nekoliko prolazaka kroz scenu sa različitim programima za sjenčanje, samo trebamo napraviti željeni broj instanci razreda *Shader* i svakoj doznačiti željene programe. Isto tako, razred *Shader* se lako može nadograditi da prihvaća nove tipove programa za sjenčanje, poput programa za sjenčanje ljuske i površine koji su dio novog modela 5.0.

Jedine poteškoće su bile rezultat toga da je *OpenGL* sučelje zamrznuto na verziju 1.1 na svim verzijama Windowsa, i standardne .h i .lib datoteke koje dolaze sa Microsoftovim prevoditeljima nisu ažurirane od 1995. Rješenje je korištenje novih `glext.h`, `wglext.h` i `gl3.h` zaglavnih datoteka koje se mogu naći na službenoj stranici *OpenGL*-a, a definiraju sve nove simbole i funkcije. Potom da bi pristupili novim *OpenGL* funkcijama, moramo deklarirati pokazivače na te funkcije na sljedeći način:

```
#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/wglext.h>

extern PFNGLACTIVETEXTUREPROC glActiveTexture;
```

```
PFNGLACTIVETEXTUREPROC glActiveTexture;
```

Te potom dohvatiti pokazivače na funkcije pomoću `wglGetProcAddress` funkcije:

```
glActiveTexture =
(PFNGLACTIVETEXTUREPROC) wglGetProcAddress("glActiveTexture");
```

Zaključak

Iako su se prvi grafički čipovi koji su omogućavali izvedbu programa za sjenčanje pojavili tek prije nekoliko godina, danas je njihovo korištenje u igrama i ostalim oblicima zabavnih programa i simulacija nezaobilazno. Pošto je samo sklopovlje na kojemu se izvode ti programi prilagođeno za paralelni rad sa vektorima i matricama, primjena tih programa se proširila na područja poput kriptografije, obrade signala, raspoznavanja uzoraka i svih ostalih u kojima je moguć vrlo visok stupanj paralelne obrade zadataka.

Jezici za programiranje programa za sjenčanje također su pratili brzi razvoj sklopovlja, te više nismo prisiljeni raditi u asembleru s ograničenim brojem instrukcija, već se možemo usredotočiti na kreiranje puno kompleksnijih programa u višim programskim jezicima.

Novi unificirani model programa za sjenčanje koristi konzistentni skup instrukcija za sve tipove programa za sjenčanje, tako da nam to osim prednosti u performansama za rad sa grafičkim efektima omogućava i lakše pisanje programa općenite namjene.

Ukratko, trenutačno najbolje rješenje s obzirom na cijenu za probleme koji zahtijevaju obradu velike količine podataka u što kraćem vremenu su programi za sjenčanje koji se izvode na programirljivom grafičkom sklopovlju. Jedina bolja opcija što se tiče performansi je izgradnja sklopovlja specifičnog za taj problem ili grozdova računala, no njihova cijena je toliko veća da u većini slučajeva ne opravdava dodatni trošak.

Literatura

[1] MIKE BAILEY, **Using GPU shaders for visualization,** IEEE Computer Graphics and Applications, Rujan 2009., pp. 96-100

[2] KURT W. FLEISCHER, DAVID H. LAIDLAW, BENA L. CURRIN, ALAN H. BARR, **Cellular Texture Generation,** California Institute of Technology, Pasadena, 1995

[3] JOHN KESSENICH, DAVE BALDWIN, RANDI ROST, **The OpenGL® Shading Language,** Revision 1.20.8, 2006.

[4] **OpenGL Shading Language @ Lighthouse3D – GLSL Tutorial,** **http://www.lighthouse3d.com/opengl/glsl/,** prosinac 2009.

[5] NATASHA TATARCHUK**, Beginner Shader Programming with RenderMonkey™,** GDC 2003.

[6] SEBASTIEN ST-LAURENT**, Shaders for Game Programmers and Artists,** First edition, 2004.

[7] The Art of Texturing Using The OpenGL Shading Language**,** **http://www.ozone3d.net/tutorials/glsl_texturing.php,** prosinac 2009.

[8] JERZY KARCZMARCZUK, **Functional Approach to Texture Generation**, University of Caen, France, 2002.

[9] KURT W. FLEISCHER, The **Biological Simulator Behind "Cellular Texture Generation"**, Pixar Animation Studios, 1997.

[10] GREG TURK, **Texture Synthesis on Surfaces**, GVU Center, College of Computing, Georgia Institute of Technology, 2001.

[11] STEPHANE GOBRON, DENIS FINCK, HENRI POINCARE, **Generating Surface Textures based on Cellular Networks,** University, LORIA, Saint-Die-des-Vosges Institute of Technology, 2006.

[12] ANDREW DAVIS, **A Simulation of Scaly Skin Textures**, University of Teesside 1998.

[13] JURAJ KONECNÝ, **Catmull–Clark Subdivision Surfaces on GPU,** Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia, 2007.

[14] SURYAKANT PATIDAR, SHIBEN BHATTACHARJEE, JAG MOHAN SINGH, P. J. NARAYANAN, **Exploiting the Shader Model 4.0 Architecture**, Center for Visual Information Technology, IIIT Hyderabad, 2007.

[15] NeHe Productions: OpenGL Lesson #31, **http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=31**, svibanj 2010.

[16] Bézier curve, **http://en.wikipedia.org/wiki/B%C3%A9zier_curve**, svibanj 2010.

[17] ALAN M. TURING, The chemical basis of morphogenesis, Philosophical Transactions of the Royal Society of London. B 327, pp 37–72 (1952)

[18] CRAIG W. REYNOLDS, **Flocks, Herds, and Schools: A Distributed Behavioral Model**, (SIGGRAPH '87 Conference Proceedings), 1987, pp 25-34

Sažetak

(Programi za sjenčanje geometrije)

U radu se detaljno proučavaju mogućnosti programa za sjenčanje fragmenata, vrhova i geometrije. Također se razmatraju I demonstriraju različiti načini implementacije i povezivanja programa za sjenčanje. Implementirano je nekoliko programa za sjenčanje vrhova i fragmenata za usporedbu i analizu.

Posebno su razrađeni programi za sjenčanje geometrije. Njihova uporaba je prikazana na problematici staničnih tekstura i geometrije za vrijeme izvođenja programa. Ukratko su opisane neke poznatije već postojeće metode generiranja staničnih tekstura, te modifikacije tih metoda koje su implementirane u sklopu ovog rada. Analiziran je utjecaj različitih parametara na performanse programa.

Ključne riječi: programi za sjenčanje fragmenata, programi za sjenčanje vrhova, programi za sjenčanje geometrije, stanične teksture, generiranje staničnih tekstura, iscrtavanje u stvarnom vremenu

Summary

(Geometry shaders)

This paper examines capabilities of pixel, vertex and geometry shaders. It also studies and demonstrates different methods of shader linking and implementation. Several shader programs are created for comparison and analysis.

Geometry shaders are examined in greater detail. Their application is shown on real time cellular texture generation examples. Some of the more known cellular texture generation methods are described together with modified methods which were implemented as a part of this paper. Performance is analyzed regarding various parameters.

Keywords: shaders, pixel shaders, vertex shaders, geometry shaders, cellular texture, cellular texture generation, real time rendering

Privitak

Zahtjevi i upute za pokretanje programa za sjenčanje vrhova i fragmenata iz prvog dijela rada

Svi programi za sjenčanje fragmenata i vrhova su napisani i testirani u programu *AMD RenderMonkey 1.81*, te su zahtjevi za njihovo pokretanje jednaki zahtjevima tog programa. Svi projekti koji sadržavaju programe navedene u prvom dijelu rada se nalaze u direktoriju **`Shaders`**.

Program za demonstraciju korištenja programa za sjenčanje unutar samostalnog projekta je napravljen pomoću razvojnog sučelja *Visual Studio 2005*, te koristi *GLUT* i *GLEW* biblioteke. U direktoriju programa su sve potrebne biblioteke, te se prilikom kopiranja programa treba voditi računa da se kopira cijeli direktorij, ako navedene biblioteke nisu već instalirane na računalu. Ovaj projekt se nalazi u direktoriju **`glutProjekt`**.

U direktoriju **`Library`** nalaze se verzije *GLUT* i *GLEW* biblioteka koje su korištene, zajedno s pripadajućim izvornim uputama za instalaciju.

Grafička kartica treba podržavati minimalno *Shader Model 2.0.*

Zahtjevi i upute za pokretanje programa za generiranje geometrije iz drugog dijela rada

Programi za generiranje geometrije uključujući i glavni program zahtjevaju instalirane upravljačke programe koji podržavaju minimalno *OpenGL 3.2* i ekstenziju *GL_EXT_geometry_shader4*. Dodatne biblioteke u ovom slučaju nisu potrebne.

Svi korišteni modeli se također nalaze unutar odgovarajućih direktorija svakog projekta, I u slučaju kopiranja izvršnih datoteka treba također paziti da se kopiraju i odgovarajući modeli.

Grafička kartica treba podržavati minimalno *Shader Model 4.0* i ekstenziju *GL_EXT_geometry_shader4.* U trenutku pisanja ovog teksta, to su sve NVidia kartice od serije 8800 pa nadalje.

Upute za kontroliranje programa

U oba programa:

q - odmicanje objekta od kamere

z - približavanje objekta kameri

w - pomicanje objekta prema gore

a - pomicanje objekta prema lijevo

s - pomicanje objekta prema dolje

d - pomicanje objekta prema desno

strelica gore - rotacija po x osi u negativnom smjeru

strelica dole - rotacija po x osi u pozitivnom smjeru

strelica lijevo - rotacija po y osi u negativnom smjeru

strelica desno - rotacija po y osi u pozitivnom smjeru

PgDn - rotacija po z osi u negativnom smjeru

PgUp - rotacija po z osi u pozitivnom smjeru

C - uključuje/isključuje žičani prikaz

I - povećava duljinu dlake/bodlje

K - smanjuje duljinu dlake/bodlje

razmaknica (space) - zaustavlja i postavlja rotaciju objekta na početne vrijednosti

U programu za generiranje bodlji

P - uključuje/isključuje programe za sjenčanje geometrije

U programu za generiranje dlake:

P - uključuje/isključuje drugi prolaz (prolaz s programima za sjenčanje)

G - uključuje/isključuje prvi prolaz (prolaz bez programa za sjenčanje)

1 – postavlja x vrijednost vektora smjera rasta dlake na -1

2 – postavlja x vrijednost vektora smjera rasta dlake na +1

3 – postavlja y vrijednost vektora smjera rasta dlake na -1

4 – postavlja y vrijednost vektora smjera rasta dlake na +1

5 – postavlja z vrijednost vektora smjera rasta dlake na -1

6 – postavlja z vrijednost vektora smjera rasta dlake na +1