Katedra počítačovej grafiky a spracovania obrazu
Fakulta matematiky fyziky a informatiky
Univerzita Komenského

Diplomová práca

Martin Štefček

BRATISLAVA 2006

Katedra počítačovej grafiky a spracovania obrazu
Fakulta matematiky fyziky a informatiky
Univerzita Komenského

Vysoko úrovňové programovanie súčasných GPU

Autor: Martin Štefček
Konzultant: RNDr. Peter Borovský BRATISLAVA,

Apríl 2006

Čestne vyhlasujem, že som túto diplomovú prácu vypracoval samostatne, len s použitím uvedenej literatúry.

. . . . . .

V Bratislave 9.apríla 2006 *Martin Štefček*

Abstrakt

Cielom diplomovej práce je predstaviť možnosti grafického hardveru.

V prvej kapitole sa budeme zaoberať optimalizáciou známej a rozšírenej geometrickej reprezentácií a síce NURBS plochami.

V druhej kapitole sa budeme zaoberať využitím MRT (multiple render targets). Aby sme dobre predviedli silu paralelizmu, ktorým súčasné GPU disponujú tak sa zameriame na celulárne automaty. Ukážeme si jeho možnosti a obmedzenia pri tvorbe rozšírených celulárnych automatov a ich rozšírenia o ďalšie vlastnosti. Predstavíme si jednoduchý programovací jazyk na programovanie takýchto celulárnych automatov a ich rozšírení. Predvedieme si využitie na tvorbu statických textúr, ako aj animovaných.

Doba prudkého rozmachu hardverovej akcelerácie zobrazovania grafiky priniesla mnoho rozmanitých grafických kariet a pokročila do štádia, kedy výrobcovia ponúkajú komerčné karty s programovateľným procesorom, na rozdiel od starších kariet, ktoré podporovali len prepínanie stavov (fixed pipeline) a neposkytovali programátorovi možnosť písať vlastné programy a tým musel nutne túto činnosť vykonávať na CPU. Prechod k programovateľným kartám prináša nové možnosti i nové postupy vyžadujúce písanie programov v iných jazykoch, než sú tie, ktoré boli na túto činnosť používané v prípade programov bežiacich na CPU.

Poďakovanie

Chcel by som sa poďakovať všetkým, ktorých som počas môjho štúdia spoznal a tiež tým, ktorí mi boli podporou nie len pri tejto práci.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

NURBS plochy

1.1 Úvod

V tejto kapitole si povieme čo sú to NURBS[1] plochy[2] a krivky[3]. Ukážeme si ako ich implementovať na GPU. Kapitola je zameraná na optimalizáciu na VS 1.4. Okrem jednoduchého zobrazenia sa budeme snažiť vypočítať čo najviac vecí pre plochu a posunúť čo najviac do PS. Budeme sa snažiť spočítať aj tangenciálny priestor pre jednotlivé vrcholy. Podmienkou optimalizácie bude rovnosť texturových koordinátov a parametrických súradníc, z dôvodu výpočtu tangenciálnych priestorov.

Budeme klásť dôraz na minimálnu pamäťovú náročnosť. Systémy ako napr. CAD môžu pri modelovaní používať až neúmerné množstvo NURBS plôch. A teda predpočítať si každú do pamäte by bolo nemožne, preto sa táto kapitola zaoberá možnosťou kreslenia takýchto plôch v reálnom čase.

1.2 Implementácia NURBS krivky

1.2.1 Čo je to NURBS krivka

Keďže krivka je jednoduchšia ako plocha a vo svojej podstate sú si podobné, tak si najprv ukážeme krivku. Krivka má jednoduchší vzorec

[1] z angličtiny Non-Uniform Rational Beta Spline
[2] http://mathworld.wolfram.com/NURBSSurface.html
[3] http://mathworld.wolfram.com/NURBSCurve.html

$$C(t) = \frac{\sum_{i=0}^{n} N_{i,p}(t)\ w_i\ P_i}{\sum_{i=0}^{n} N_{i,p}(t)\ w_i}. \tag{1.1}$$

V implementácií budeme uvažovať iba krivky 3. stupňa. Krivky nižšieho stupňa sú jednoduchšie a vyššieho stupňa sa už nedajú až tak efektívne implementovať[4]. Krivky 3. stupňa sú považované za dostatočne hladké.

1.2.2 Jednoduchá implementácia

```
struct OST
{
    float4 HPosition    : POSITION;
};

struct IN
{
    float4 Position     : POSITION;
};

uniform float3 Points[4];
uniform float Knots[7];

float KnotP(float t, int i, int j)
{
    float dif = Knots[j]-Knots[i];
    if (dif>0.0)
        return (t-Knots[i])/dif;
    else
        return 0.0;
}
```

[4] bude to zrejmé z implementácie

```
float KnotN(float t, int i, int j)
{
    float dif = Knots[j]-Knots[i];
    if (dif<0.0)
        return (t-Knots[i])/dif;
    else
        return 0.0;
}

float N0(float t)
{
    return KnotN(t,4,1)*KnotN(t,4,2)*KnotN(t,4,3);
}

float N1(float t)
{
    return KnotP(t,1,4)*KnotN(t,4,2)*KnotN(t,4,3)+
            KnotN(t,5,2)*KnotP(t,2,4)*KnotN(t,4,3)+
            KnotN(t,5,2)*KnotN(t,5,3)*KnotP(t,3,4);
}

float N2(float t)
{
    return KnotP(t,2,5)*KnotP(t,2,4)*KnotN(t,4,3)+
            KnotP(t,2,5)*KnotN(t,5,3)*KnotP(t,3,4)+
            KnotN(t,6,3)*KnotP(t,3,5)*KnotP(t,3,4);
}

float N3(float t)
{
    return KnotP(t,3,6)*KnotP(t,3,5)*KnotP(t,3,4);
}
```

```
float3 Nurbs(float t)
{
    return Points[0]*N0(t)+Points[1]*N1(t)+Points[2]*N2(t)+Points[3]*N3(t);
}

OST main(IN input, uniform float4x4 ProjectionAndModelview)
{
    OST ost;
    ost.HPosition = mul(ProjectionAndModelview, float4(Nurbs(input.Position.x),1)
    return ost;
}
```

Pre jednoduchosť si ukážeme implementáciu NUBS(t.j. krivka ktorá nemá váhy vo vrcholoch) krivky. Skompilovaný program(ktorý už je čiastočne optimalizovaný) má 130 inštrukcií a 4 registre. Avšak shader model 1.4 podporuje maximálne 128 inštrukcií. Krivka nám poslúžila iba ako príklad, ako programovať vo VS. Sústrediť sa budeme na plochu.
Bázické funkcie majú vzorce:

$$N_{i,0}(t) = \begin{cases} 1 & \text{Ak} \quad t_i \leq t \leq t_{i+1} \quad \text{a zároveň} \quad t_i < t_{i+1} \\ 0 & \text{V opačnom prípade} \end{cases} \tag{1.2}$$

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t). \tag{1.3}$$

Bez ujmy na všeobecnosti nech $t \in < t_j, t_{j+1} >$ potom pre j–3, j–2, j–1 a j je $N_{i,p}(t) \neq 0$, Keďže iba $N_{j,0}(t) = 1$, pre $(i \neq j) \Rightarrow (N_{i,0} = 0)$. Teraz budeme potrebovať tieto už

rozpísané rovnice :

$$N_{j-3,3}(t) = \frac{t_{j+1}-t}{t_{j+1}-t_{j-2}} \frac{t_{j+1}-t}{t_{j+1}-t_{j-1}} \frac{t_{j+1}-t}{t_{j+1}-t_j} \tag{1.4}$$

$$\begin{aligned} N_{j-2,3}(t) = & \frac{t-t_{j-2}}{t_{j+1}-t_{j-2}} \frac{t_{j+1}-t}{t_{j+1}-t_{j-1}} \frac{t_{j+1}-t}{t_{j+1}-t_j} + \\ & \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}} \frac{t-t_{j-1}}{t_{j+1}-t_{j-1}} \frac{t_{j+1}-t}{t_{j+1}-t_j} + \\ & \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}} \frac{t_{j+2}-t}{t_{j+2}-t_j} \frac{t-t_j}{t_{j+1}-t_j} \end{aligned} \tag{1.5}$$

$$\begin{aligned} N_{j-1,3}(t) = & \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}} \frac{t-t_{j-1}}{t_{j+1}-t_{j-1}} \frac{t_{j+1}-t}{t_{j+1}-t_j} + \\ & \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}} \frac{t_{j+2}-t}{t_{j+2}-t_j} \frac{t-t_j}{t_{j+1}-t_j} + \\ & \frac{t_{j+3}-t}{t_{j+3}-t_j} \frac{t-t_j}{t_{j+2}-t_j} \frac{t-t_j}{t_{j+1}-t_j} \end{aligned} \tag{1.6}$$

$$N_{j,3}(t) = \frac{t-t_j}{t_{j+3}-t_j} \frac{t-t_j}{t_{j+2}-t_j} \frac{t-t_j}{t_{j+1}-t_j} \tag{1.7}$$

A teda vzorec pre krivku je

$$C(t) = N_{j-3,3}(t)P_{j-3} + N_{j-2,3}(t)P_{j-2} + N_{j-1,3}(t)P_{j-1} + N_{j,3}(t)P_j \tag{1.8}$$

Points Krivka je v každom bode kubická t.j. nám stačia 4 vrcholy

Knots Časť uzlového vektora, k ostatným častiam sa výpočet nedostane (zobrazovanie prebieha po segmentoch)

float N0(float t) $N_{j-3,3}(t)$

float N1(float t) $N_{j-2,3}(t)$

float N2(float t) $N_{j-1,3}(t)$

float N3(float t) $N_{j,3}(t)$

1.3 Čo sú to NURBS plochy?

Najskôr si stručne povieme čo sú to NURBS plochy a aké je ich využitie. NURBS plochy sú definované podľa vzorca

$$S(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)\, N_{j,q}(v)\, w_{i,j}\, P_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)\, N_{j,q}(v)\, w_{i,j}.} \tag{1.9}$$

Kde $N_{i,p}$ a $N_{j,s}$ sú bázické funkcie. $P_{i,j}$ sú riadiace body a $w_{i,j}$ sú váhy k ním prislúchajúce.

1.4 Jednoduchá implementácia

Princíp implementácie je podobný ako u krivky.

```
struct OST
{
    float4 HPosition    : POSITION;
};

struct IN
{
    float4 Position     : POSITION;
};

uniform float3 Points0[4];
uniform float3 Points1[4];
uniform float3 Points2[4];
uniform float3 Points3[4];
uniform float KnotsS[7];
uniform float KnotsT[7];

float KnotSP(float u, int i, int j)
{
```

```
   float dif = KnotsS[j]-KnotsS[i];
   if (dif>0.0)
      return (u-KnotsS[i])/dif;
   else
      return 0.0;
}

float KnotSN(float u, int i, int j)
{
   float dif = KnotsS[j]-KnotsS[i];
   if (dif<0.0)
      return (u-KnotsS[i])/dif;
   else
      return 0.0;
}

float KnotTP(float v, int i, int j)
{
   float dif = KnotsT[j]-KnotsT[i];
   if (dif>0.0)
      return (v-KnotsT[i])/dif;
   else
      return 0.0;
}

float KnotTN(float v, int i, int j)
{
   float dif = KnotsT[j]-KnotsT[i];
   if (dif<0.0)
      return (v-KnotsT[i])/dif;
   else
      return 0.0;
}
```

```
float SN0(float t)
{
   return KnotSN(t,4,1)*KnotSN(t,4,2)*KnotSN(t,4,3);
}

float SN1(float t)
{
   return KnotSP(t,1,4)*KnotSN(t,4,2)*KnotSN(t,4,3)+
          KnotSN(t,5,2)*KnotSP(t,2,4)*KnotSN(t,4,3)+
          KnotSN(t,5,2)*KnotSN(t,5,3)*KnotSP(t,3,4);
}

float SN2(float t)
{
   return KnotSP(t,2,5)*KnotSP(t,2,4)*KnotSN(t,4,3)+
          KnotSP(t,2,5)*KnotSN(t,5,3)*KnotSP(t,3,4)+
          KnotSN(t,6,3)*KnotSP(t,3,5)*KnotSP(t,3,4);
}

float SN3(float t)
{
   return KnotSP(t,3,6)*KnotSP(t,3,5)*KnotSP(t,3,4);
}

float TN0(float t)
{
   return KnotTN(t,4,1)*KnotTN(t,4,2)*KnotTN(t,4,3);
}

float TN1(float t)
{
   return KnotTP(t,1,4)*KnotTN(t,4,2)*KnotTN(t,4,3)+
```

```
         KnotTN(t,5,2)*KnotTP(t,2,4)*KnotTN(t,4,3)+
         KnotTN(t,5,2)*KnotTN(t,5,3)*KnotTP(t,3,4);
}

float TN2(float t)
{
   return KnotTP(t,2,5)*KnotTP(t,2,4)*KnotTN(t,4,3)+
         KnotTP(t,2,5)*KnotTN(t,5,3)*KnotTP(t,3,4)+
         KnotTN(t,6,3)*KnotTP(t,3,5)*KnotTP(t,3,4);
}

float TN3(float t)
{
   return KnotTP(t,3,6)*KnotTP(t,3,5)*KnotTP(t,3,4);
}

float3 Nurbs(float u, float v)
{
   return  Points0[0]*SN0(u)*TN0(v)+Points0[1]*SN1(u)*TN0(v)+
           Points0[2]*SN2(u)*TN0(v)+Points0[3]*SN3(u)*TN0(v)+
           Points1[0]*SN0(u)*TN1(v)+Points1[1]*SN1(u)*TN1(v)+
           Points1[2]*SN2(u)*TN1(v)+Points1[3]*SN3(u)*TN1(v)+
           Points2[0]*SN0(u)*TN2(v)+Points2[1]*SN1(u)*TN2(v)+
           Points2[2]*SN2(u)*TN2(v)+Points2[3]*SN3(u)*TN2(v)+
           Points3[0]*SN0(u)*TN3(v)+Points3[1]*SN1(u)*TN3(v)+
           Points3[2]*SN2(u)*TN3(v)+Points3[3]*SN3(u)*TN3(v);
}

OST main(IN input, uniform float4x4 ProjectionAndModelview)
{
    OST ost;
    ost.HPosition = mul(ProjectionAndModelview,
                        float4(Nurbs(input.Position.x, input.Position.y),1));
```

```
    return ost;
}
```

Prvotná implementácia je jednoduchá avšak používa až 279 inštrukcií. Čo je viac než dva krát povolený limit pre profil na ktorý sa snažíme implementovať. A to ešte máme len vizualizáciu. Program pracuje analogicky ako pre krivky. V jednoduchosti je možno krása ale nie rýchlosť a potrebná veľkosť.

1.5 Matematické pozadie

Pre tangenciálny priestor bude potrebovať parciálne derivácie v smeroch. Parciálnym derivovaním funkcie(1.9) dostávame nasledovné vzorce:

$$\frac{\partial S(u,v)}{\partial u} = \frac{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}\frac{\partial N_{i,p}(u)}{\partial u}N_{j,q}(v)w_{i,j}P_{i,j}\right)\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)}{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)^2} - \frac{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}P_{i,j}\right)\left(\sum_{i=0}^{m}\sum_{j=0}^{n}\frac{\partial N_{i,p}(u)}{\partial u}N_{j,q}(v)w_{i,j}\right)}{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)^2} \tag{1.10}$$

$$\frac{\partial S(u,v)}{\partial v} = \frac{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)\frac{\partial N_{j,q}(v)}{\partial v}w_{i,j}P_{i,j}\right)\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)}{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)^2} - \frac{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}P_{i,j}\right)\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)\frac{\partial N_{j,p}(q)}{\partial v}w_{i,j}\right)}{\left(\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,p}(u)N_{j,q}(v)w_{i,j}\right)^2} \tag{1.11}$$

Zo vzorcov pre derivácie (1.10 a 1.11) je vidieť že na výpočet budeme potrebovať derivácie $N_{i,p}(u)$ a $N_{j,q}(v)$. Zo vzorcov 5 (1.4, 1.5, 1.6 a 1.7) po úpravách :

$$\frac{\partial N_{j-3,3}(t)}{\partial t} = -\frac{t_{j+1}-t}{t_{j+1}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t_{j+1}-t}{t_{j+1}-t_{j-2}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t_{j+1}-t}{t_{j+1}-t_{j-2}}\frac{t_{j+1}-t}{t_{j+1}-t_{j-1}} \tag{1.12}$$

$$\begin{aligned}\frac{\partial N_{j-2,3}(t)}{\partial t} = {} & \frac{t_{j+1}-t}{t_{j+1}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t-t_{j-2}}{t_{j+1}-t_{j-2}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t-t_{j-2}}{t_{j+1}-t_{j-2}}\frac{t_{j+1}-t}{t_{j+1}-t_{j-1}} - \\ & \frac{t-t_{j-1}}{t_{j+1}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} + \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}}\frac{t-t_{j-1}}{t_{j+1}-t_{j-1}} - \\ & \frac{t_{j+2}-t}{t_{j+2}-t_j}\frac{t-t_j}{t_{j+1}-t_j} - \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t_{j+2}-t}{t_{j+2}-t_{j-1}}\frac{t_{j+2}-t}{t_{j+2}-t_j}\end{aligned} \tag{1.13}$$

$$\begin{aligned}\frac{\partial N_{j-1,3}(t)}{\partial t} = {} & \frac{t-t_{j-1}}{t_{j+1}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} + \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}}\frac{t_{j+1}-t}{t_{j+1}-t_j} - \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}}\frac{t-t_{j-1}}{t_{j+1}-t_{j-1}} + \\ & \frac{t_{j+2}-t}{t_{j+2}-t_j}\frac{t-t_j}{t_{j+1}-t_j} - \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t-t_{j-1}}{t_{j+2}-t_{j-1}}\frac{t_{j+2}-t}{t_{j+2}-t_j} - \\ & \frac{t-t_j}{t_{j+2}-t_j}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t_{j+3}-t}{t_{j+3}-t_j}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t_{j+3}-t}{t_{j+3}-t_j}\frac{t-t_j}{t_{j+2}-t_j}\end{aligned} \tag{1.14}$$

$$\frac{\partial N_{j,3}(t)}{\partial t} = \frac{t-t_j}{t_{j+2}-t_j}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t-t_j}{t_{j+3}-t_j}\frac{t-t_j}{t_{j+1}-t_j} + \frac{t-t_j}{t_{j+3}-t_j}\frac{t-t_j}{t_{j+2}-t_j} \tag{1.15}$$

1.6 Optimalizácia na vertex shader 1.4

V tejto časti sa ukazuje priestorová náročnosť plôch vo VS. Už máme všetky potrebné vzorce k dispozícii, môžeme začať optimalizovať pôvodný zdrojový kód. Najprv sa budeme snažiť uzátvorkovať výrazy čo najlepšie. Tu si treba dať pozor, vhodné uzátvorkovanie výrazov $N_{j-3,p}$, $N_{j-2,p}$, $N_{j-1,p}$ a $N_{j,p}$ nie je vhodné pre výpočet derivácií a opačne. Ďalej treba všetko čo sa ráta pre viac ako jeden vertex dať preč z GPU, z dôvodu rýchlosti a veľkosti výsledného VS. Problem prerozdelenia výpočtu CPU a GPU je v tom že ak sa niečo čo je rovnaké pre celý segment počíta na GPU tak sa to ráta zbytočne veľa krát pretože GPU si to nevie uložiť. Preto sa to z hľadiska rýchlosti a samozrejme aj veľkosti VS oplatí predpočítať na CPU a do VS to už iba poslať ako predpočítanú hodnotu. Vzorce

1.4, 1.5, 1.6 a 1.7 si prepíšeme tak aby indexy začínali od 0.

$$N_{0,3}(t) = \frac{t_4-t}{t_4-t_1}\frac{t_4-t}{t_4-t_2}\frac{t_4-t}{t_4-t_3} \tag{1.16}$$

$$\begin{aligned} N_{1,3}(t) = {} & \frac{t-t_1}{t_4-t_1}\frac{t_4-t}{t_4-t_2}\frac{t_4-t}{t_4-t_3} + \\ & \frac{t_5-t}{t_5-t_2}\frac{t-t_2}{t_4-t_2}\frac{t_4-t}{t_4-t_3} + \\ & \frac{t_5-t}{t_5-t_2}\frac{t_5-t}{t_5-t_3}\frac{t-t_3}{t_4-t_3} \end{aligned} \tag{1.17}$$

$$\begin{aligned} N_{2,3}(t) = {} & \frac{t-t_2}{t_5-t_2}\frac{t-t_2}{t_4-t_2}\frac{t_4-t}{t_4-t_3} + \\ & \frac{t-t_2}{t_5-t_2}\frac{t_5-t}{t_5-t_3}\frac{t-t_3}{t_4-t_3} + \\ & \frac{t_6-t}{t_6-t_3}\frac{t-t_3}{t_5-t_3}\frac{t-t_3}{t_4-t_3} \end{aligned} \tag{1.18}$$

$$N_{3,3}(t) = \frac{t-t_3}{t_6-t_3}\frac{t-t_3}{t_5-t_3}\frac{t-t_3}{t_4-t_3} \tag{1.19}$$

$$\frac{\partial N_{0,3}(t)}{\partial t} = -\frac{t_4-t}{t_4-t_2}\frac{t_4-t}{t_4-t_3} - \frac{t_4-t}{t_4-t_1}\frac{t_4-t}{t_4-t_3} - \frac{t_4-t}{t_4-t_1}\frac{t_4-t}{t_4-t_2} \tag{1.20}$$

$$\begin{aligned} \frac{\partial N_{1,3}(t)}{\partial t} = {} & \frac{t_4-t}{t_4-t_2}\frac{t_4-t}{t_4-t_3} - \frac{t-t_1}{t_4-t_1}\frac{t_4-t}{t_4-t_3} - \frac{t-t_1}{t_4-t_1}\frac{t_4-t}{t_4-t_2} - \\ & \frac{t-t_2}{t_4-t_2}\frac{t_4-t}{t_4-t_3} + \frac{t_5-t}{t_5-t_2}\frac{t_4-t}{t_4-t_3} - \frac{t_5-t}{t_5-t_2}\frac{t-t_2}{t_4-t_2} - \\ & \frac{t_5-t}{t_5-t_3}\frac{t-t_3}{t_4-t_3} - \frac{t_5-t}{t_5-t_2}\frac{t-t_3}{t_4-t_3} + \frac{t_5-t}{t_5-t_2}\frac{t_5-t}{t_5-t_3} \end{aligned} \tag{1.21}$$

$$\begin{aligned} \frac{\partial N_{2,3}(t)}{\partial t} = {} & \frac{t-t_2}{t_4-t_2}\frac{t_4-t}{t_4-t_3} + \frac{t-t_2}{t_5-t_2}\frac{t_4-t}{t_4-t_3} - \frac{t-t_2}{t_5-t_2}\frac{t-t_2}{t_4-t_2} + \\ & \frac{t_5-t}{t_5-t_3}\frac{t-t_3}{t_4-t_3} - \frac{t-t_2}{t_5-t_2}\frac{t-t_3}{t_4-t_3} + \frac{t-t_2}{t_5-t_2}\frac{t_5-t}{t_5-t_3} - \\ & \frac{t-t_3}{t_5-t_3}\frac{t-t_3}{t_4-t_3} + \frac{t_6-t}{t_6-t_3}\frac{t-t_3}{t_4-t_3} + \frac{t_6-t}{t_6-t_3}\frac{t-t_3}{t_5-t_3} \end{aligned} \tag{1.22}$$

$$\frac{\partial N_{3,3}(t)}{\partial t} = \frac{t-t_3}{t_5-t_3}\frac{t-t_3}{t_4-t_3} + \frac{t-t_3}{t_6-t_3}\frac{t-t_3}{t_4-t_3} + \frac{t-t_3}{t_6-t_3}\frac{t-t_3}{t_5-t_3} \tag{1.23}$$

V zdrojovom kóde (CPU) si predpočítame potrebné hodnoty ktoré sa nemenia pre jeden segment plochy t.j. keď $u \in < u_i, u_{i+1} > \wedge v \in < v_j, v_{j+1} >$.

$$\begin{aligned} NN1[0] &= (u_3 - u_0) * (u_3 - u_1) * (u_3 - u_2) \\ NN1[1] &= (u_5 - u_2) * (u_3 - u_2) * (u_4 - u_2) \\ NN1[2] &= (v_3 - v_0) * (v_3 - v_1) * (v_3 - v_2) \\ NN1[3] &= (v_5 - v_2) * (v_3 - v_2) * (v_4 - v_2) \end{aligned}$$

$$\begin{aligned} NN2[0] &= (u_3 - u_1) * (u_3 - u_2) * (u_4 - u_1) \\ NN2[1] &= (u_4 - u_2) * (u_3 - u_2) * (u_4 - u_1) \\ NN2[2] &= (v_3 - v_1) * (v_3 - v_2) * (v_4 - v_1) \\ NN2[3] &= (v_4 - v_2) * (v_3 - v_2) * (v_4 - v_1) \end{aligned}$$

Tieto hodnoty sú najvýhodnejšie pre náš výpočet. Skúsený GPU programátor vie že GPU potrebuje rovnako veľa inštrukcií na prácu zo skalárom ako aj so štvor rozmerným vektorom. Pretože GPU je RISC a robí operácie paralelne. Preto je jedno či operáciu na 1D vektore alebo 4D vektore. Preto sme sa rozhodli pre 2 štvor rozmerné vektory. Tieto hodnoty sa nachádzajú v menovateľoch zlomkov pre výpočet plochy(1.9). Ďalej budeme potrebovať ešte samotné hodnoty $u_0 \cdots u_5$ a $v_0 \cdots v_5$. Hodnoty $u_1 \cdots u_4$ a $v_1 \cdots v_4$ si uložíme do poľa o 4 prvkoch kde každý prvok bude obsahovať dvoj rozmerný vektor (u_i, v_i) teda:

$$\begin{aligned} KnotsST[0] &= (u_1, v_1) \\ KnotsST[1] &= (u_2, v_2) \\ KnotsST[2] &= (u_3, v_3) \\ KnotsST[3] &= (u_4, v_4) \end{aligned}$$

A u_0, v_0, u_5 a v_5 si uložíme do jedného 4 rozmerného vektora. Prečo sme si práve takto rozhodli uložiť vstupné parametre si ukážeme neskôr. Ako rýchlo sa dajú spočítať pomocou týchto hodnôt vzorce 1.16, 1.17, 1.18 a 1.19. Teraz si postupne vysvetlíme krok za

krokom ako tento optimalizovaný VS pracuje.

```
uv05 = uv.xyxy-Knots05;
uv1 = uv-KnotsST[0];
uv2 = uv-KnotsST[1];
uv3 = uv-KnotsST[2];
uv4 = uv-KnotsST[3];
```

uv je vstupný parameter pre plochu(1.9), je to dvojrozmerný vektor. Z matematického hľadiska dostávame nasledovné:

$$\begin{aligned} uv05 &= (u-u_0, v-v_0, u-u_5, v-v_5) \\ uv1 &= (u-u_1, v-v_1) \\ uv2 &= (u-u_2, v-v_2) \\ uv3 &= (u-u_3, v-v_3) \\ uv4 &= (u-u_4, v-v_4) \end{aligned}$$

Teraz si dodefinujeme 4 pomocné vektory, aby sa nám lepšie(menej inštrukcií) pracovalo. Znovu použiteľnosť je jedna z možností optimalizácie na pamäť ale aj na rýchlosť.

```
s4 = float4(uv1.x,uv2.x,uv3.x,uv4.x);
t4 = float4(uv1.y,uv2.y,uv3.y,uv4.y);
ss4 = float4(uv1.x,uv4.x,uv1.y,uv4.y);
tt4 = float4(uv2.x,uv3.x,uv2.y,uv3.y);
```

$$\begin{aligned} s4 &= (u-u_1, u-u_2, u-u_3, u-u_4) \\ t4 &= (v-v_1, v-v_2, v-v_3, v-v_4) \end{aligned}$$

$$\begin{aligned} ss4 &= (u-u_1, u-u_4, v-v_1, v-v_4) \\ tt4 &= (u-u_2, u-u_3, v-v_2, v-v_3) \end{aligned}$$

Z pohľadu programátora ktorý neprogramuje na GPU, sa tieto definície môžu zdať divné, pretože ide v podstate o to isté(s4,t4 obsahuje rovnaké hodnoty ako ss4,tt4). Avšak ako už bolo spomenuté programovať GPU a CPU je veľký rozdiel vďaka architektúre GPU.

```
L1 = s4.xzyw;
L1 /= NN2.xxyy;
L2 = t4.xzyw;
L2 /= NN2.zzww;
```

$$
\begin{aligned}
L1 &= \left(\frac{u-u_1}{(u_3-u_1)(u_3-u_2)(u_4-u_1)}, \frac{u-u_3}{(u_3-u_1)(u_3-u_2)(u_4-u_1)},\right. \\
&\quad \left.\frac{u-u_2}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \frac{u-u_4}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}\right) \\
L2 &= \left(\frac{v-v_1}{(v_3-v_1)(v_3-v_2)(v_4-v_1)}, \frac{v-v_3}{(v_3-v_1)(v_3-v_2)(v_4-v_1)},\right. \\
&\quad \left.\frac{v-v_2}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}, \frac{v-v_4}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}\right)
\end{aligned}
$$

O delenie 0 nemusíme mať strach, pretože namiesto chyby je výsledok rovný 0, a to je presne to čo potrebujeme, pretože deliť 0 sa bude iba v prípade že $u_i = u_j$ resp. $v_i = v_j$ a zároveň $i \neq j$. A teda do takéhoto intervalu sa parameter u resp. v dostať nemôže.

```
K1.x = dot(float4(1),L1);
K1.y = dot(L1.xz,s4.zw);
K1.z = dot(float4(1),L2);
K1.w = dot(L2.xz,t4.zw);
```

$$
\begin{aligned}
K1.x &= \frac{u-u_1}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{u-u_3}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} \\
&\quad \frac{u-u_2}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} + \frac{u-u_4}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} \\
K1.y &= \frac{(u-u_1)(u-u_3)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} \\
K1.z &= \frac{v-v_1}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{v-v_3}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \\
&\quad \frac{v-v_2}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} + \frac{v-v_4}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} \\
K1.w &= \frac{(v-v_1)(v-v_3)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}
\end{aligned}
$$

Na tejto časti je vidieť ďalší rozdiel medzi programovaním GPU a CPU. Menej inštrukcií nám zoberie skalárny súčin, s vektorom (1,1,1,1), oproti sčítaniu zložiek vektora.

```
K2.xyzw = tt4.yxwz/NN1;
```

$$K2 = \left(\frac{u-u_3}{(u_3-u_0)(u_3-u_1)(u_3-u_2)}, \frac{u-u_2}{(u_5-u_2)(u_3-u_2)(u_4-u_2)}, \frac{v-v_3}{(v_3-v_0)(v_3-v_1)(v_3-v_2)}, \frac{v-v_2}{(v_5-v_2)(v_3-v_2)(v_4-v_2)}\right)$$

```
M=tt4.yxwz*K2;
```

$$M = \left(\frac{(u-u_3)(u-u_3)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)}, \frac{(u-u_2)(u-u_2)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)}, \frac{(v-v_3)(v-v_3)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)}, \frac{(v-v_2)(v-v_2)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)}\right)$$

1.6.1 Bod plochy

```
SS = M.xyzw*tt4.yxwz;
TT = M.xyzw*uv05.xzyw+K1.yyww*ss4.yxwz;
```

$$SS = \left(\frac{(u-u_3)(u-u_3)(u-u_3)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)}, \frac{(u-u_2)(u-u_2)(u-u_2)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)}, \frac{(v-v_3)(v-v_3)(v-v_3)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)}, \frac{(v-v_2)(v-v_2)(v-v_2)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)}\right)$$

$$TT = \left(\begin{array}{l}\frac{(u-u_3)(u-u_3)(u-u_0)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)} + \frac{(u-u_1)(u-u_3)(u-u_4)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_2)(u-u_4)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\ \frac{(u-u_2)(u-u_2)(u-u_5)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)} + \frac{(u-u_1)(u-u_3)(u-u_1)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_2)(u-u_4)(u-u_1)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\ \frac{(v-v_3)(v-v_3)(v-v_0)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)} + \frac{(v-v_1)(v-v_3)(v-v_4)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_2)(v-v_4)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}, \\ \frac{(v-v_2)(v-v_2)(v-v_5)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)} + \frac{(v-v_1)(v-v_3)(v-v_1)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_2)(v-v_4)(v-v_1)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}\end{array}\right)$$

Tieto vzorce už sú podobné potrebným vzorcom(1.16, 1.17, 1.18 a 1.19). Presnejšie

```
S = float4(SS.x,TT.xy,SS.y);
T = float4(SS.z,TT.zw,SS.w);
```

Teda :

$$\begin{aligned} S.x &= -N_{0,3}(u) \\ S.y &= N_{1,3}(u) \\ S.z &= -N_{2,3}(u) \\ S.w &= N_{3,3}(u) \end{aligned}$$

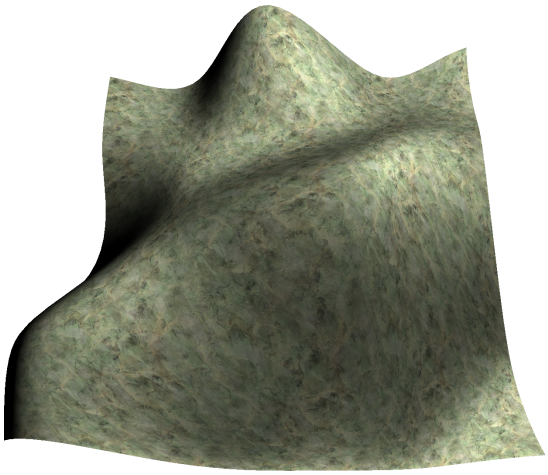
$$\begin{aligned} T.x &= -N_{0,3}(v) \\ T.y &= N_{1,3}(v) \\ T.z &= -N_{2,3}(v) \\ T.w &= N_{3,3}(v) \end{aligned}$$

Najprv sa pozrime ako vyzerá výsledný bod :

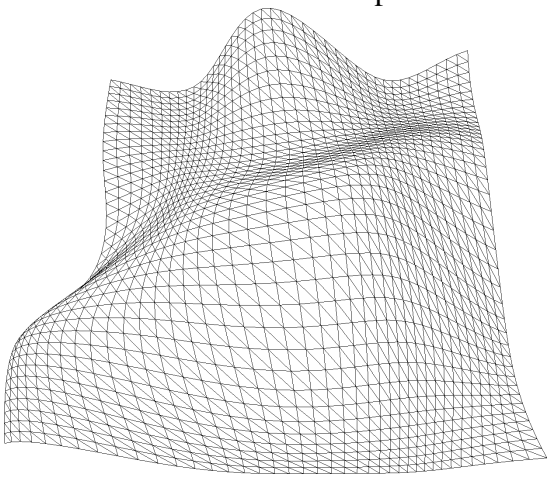
```
float W=dot(T,mul(Weight,S));
V = float3(dot(T,mul(PointsX,S)),dot(T,mul(PointsY,S)),dot(T,mul(PointsZ,S)))/W
```

$$\begin{aligned} W &= T * Weight * S \\ V &= \frac{(T * PointsX * S, T * PointsY * S, T * PointsZ * S)}{W} \end{aligned}$$

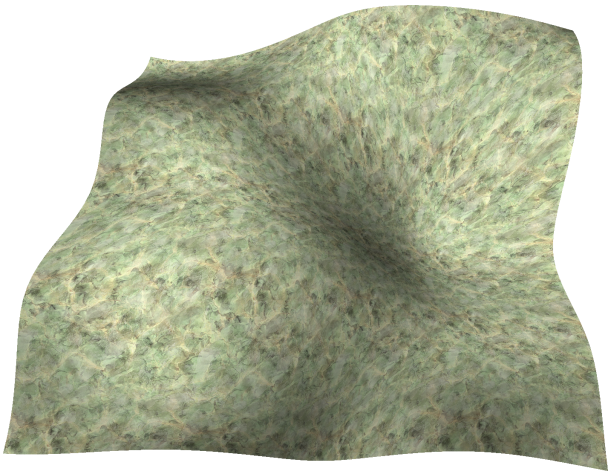
Pýtate sa čo so znamienkami? Prečo nie je v čitateľovi započítaná váha? Oba problémy sa riešia totiž ešte na CPU, takto sme ušetrili pár inštrukcií navyše. *Weight* je matica ktorá obsahuje váhy bodov aktuálneho segmentu 4x4. *PointsX* resp. *PointsY*, resp. *PoinstZ* sú matice x-ových, resp. y-ových, resp. z-ových súradníc.



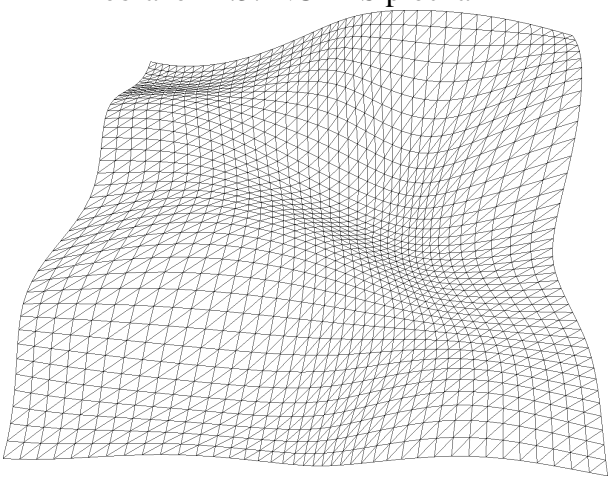
obrázok 1.1: NURBS plocha



obrázok 1.2: Sieťový model



obrázok 1.3: NURBS plocha



obrázok 1.4: Sieťový model

1.6.2 Tangenciálny priestor

Teraz nám ešte zostáva vypočítať na prvý pohľad komplikované vzorce (1.10 a 1.11). Avšak vďaka dobrej implementácií máme skoro všetko už predpočítané. Stačí nám teda urobiť nasledovné:

```
float4 Q = 2*uv05*K2.xzyw+K1.ywyw;
NS1 = 3*M;
NT1 = M+ss4.yxwz*K1.xxzz+Q.xzyw;
NormalS = float4(NS1.x,NT1.xy,NS1.y);
NormalT = float4(NS1.z,NT1.zw,NS1.w);
float WSd = dot(T,mul(Weight,NormalS));
float WTd = dot(NormalT,mul(Weight,S));
Tangent = (float3(dot(T,mul(PointsX,NormalS)),
                  dot(T,mul(PointsY,NormalS)),
                  dot(T,mul(PointsZ,NormalS)))-V*WSd)/W;
Bitangent = (float3(dot(NormalT,mul(PointsX,S)),
                    dot(NormalT,mul(PointsY,S)),
                    dot(NormalT,mul(PointsZ,S)))-V*WTd)/W;
```

Teraz si matematicky rozpíšeme čo dostávame:

$$
\begin{aligned}
Q \;=\; \Bigg(& \frac{2(u-u_3)(u-u_0)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)} + \frac{(u-u_1)(u-u_3)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\
& \frac{2(v-v_3)(v-v_0)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)} + \frac{(v-v_1)(v-v_3)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}, \\
& \frac{2(u-u_2)(u-u_5)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)} + \frac{(u-u_1)(u-u_3)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\
& \frac{2(v-v_2)(v-v_5)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)} + \frac{(v-v_1)(v-v_3)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} \Bigg)
\end{aligned}
$$

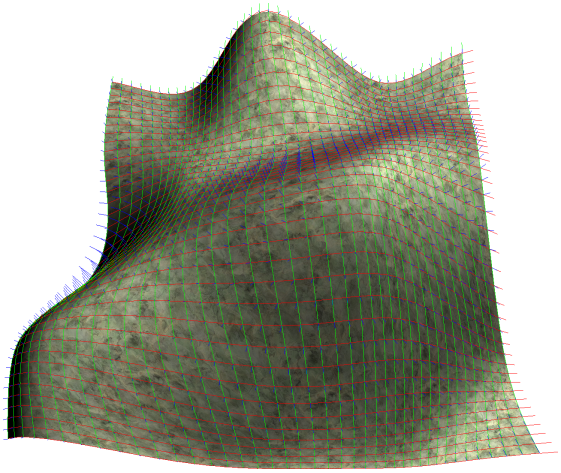
$$
\begin{aligned}
NS1 \;=\; \Bigg(& \frac{3(u-u_3)(u-u_3)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)}, \frac{3(u-u_2)(u-u_2)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)}, \\
& \frac{3(v-v_3)(v-v_3)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)}, \frac{3(v-v_2)(v-v_2)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)} \Bigg)
\end{aligned}
$$

$$
\begin{aligned}
NT1 = \Bigg(& \frac{(u-u_3)(u-u_3)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)} + \frac{2(u-u_3)(u-u_0)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)} + \frac{(u-u_1)(u-u_3)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} \\
& + \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_1)(u-u_4)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_3)(u-u_4)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} \\
& \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_4)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\
& \frac{(u-u_2)(u-u_2)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)} + \frac{2(u-u_2)(u-u_5)}{(u_5-u_2)(u_3-u_2)(u_4-u_2)} + \frac{(u-u_1)(u-u_3)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} \\
& + \frac{(u-u_2)(u-u_4)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_1)(u-u_1)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_3)(u-u_1)}{(u_3-u_1)(u_3-u_2)(u_4-u_1)} \\
& \frac{(u-u_2)(u-u_1)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)} + \frac{(u-u_4)(u-u_1)}{(u_4-u_2)(u_3-u_2)(u_4-u_1)}, \\
& \frac{(v-v_3)(v-v_3)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)} + \frac{2(v-v_3)(v-v_0)}{(v_3-v_0)(v_3-v_1)(v_3-v_2)} + \frac{(v-v_1)(v-v_3)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} \\
& + \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_1)(v-v_4)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_3)(v-v_4)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \\
& \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_4)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)}, \\
& \frac{(v-v_2)(v-v_2)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)} + \frac{2(v-v_2)(v-v_5)}{(v_5-v_2)(v_3-v_2)(v_4-v_2)} + \frac{(v-v_1)(v-v_3)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} \\
& + \frac{(v-v_2)(v-v_4)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_1)(v-v_1)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_3)(v-v_1)}{(v_3-v_1)(v_3-v_2)(v_4-v_1)} + \\
& \frac{(v-v_2)(v-v_1)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} + \frac{(v-v_4)(v-v_1)}{(v_4-v_2)(v_3-v_2)(v_4-v_1)} \Bigg)
\end{aligned}
$$

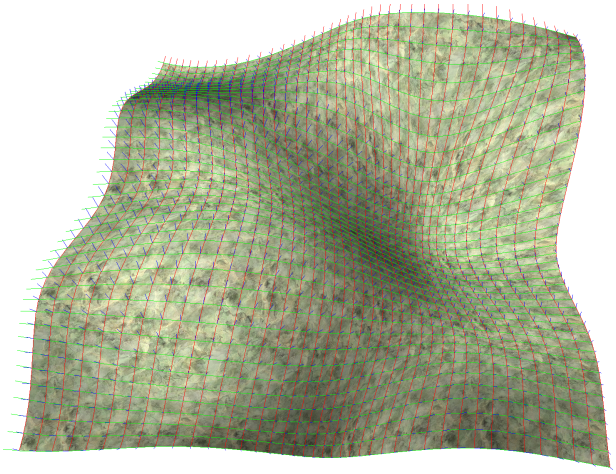
$$
\begin{aligned}
\frac{\partial N_{0,3}(u)}{\partial u} &= -NS1.x \\
\frac{\partial N_{1,3}(u)}{\partial u} &= NT1.x \\
\frac{\partial N_{2,3}(u)}{\partial u} &= -NT1.y \\
\frac{\partial N_{3,3}(u)}{\partial u} &= NS1.y
\end{aligned}
$$

$$\begin{aligned}
\frac{\partial N_{0,3}(v)}{\partial v} &= -NS1.z \\
\frac{\partial N_{1,3}(v)}{\partial v} &= NT1.z \\
\frac{\partial N_{2,3}(v)}{\partial v} &= -NT1.w \\
\frac{\partial N_{3,3}(v)}{\partial v} &= NS1.w
\end{aligned}$$

Pri rátaní samotnej tangenty a bitangenty nastáva rovnaký problem ako pri rátaní bodu(17), avšak ten už bol vyriešený na CPU. Normálu dostávame jednoduchým vektorovým súčinom tangenty a bitangenty.



obrázok 1.5: Tangenciálny priestor



obrázok 1.6: Tangenciálny priestor

1.6.3 Optimalizovaný VS

```
struct OST
{
   float4 HPosition   : POSITION;
   float4 Color       : COLOR0;
   float2 TexCoord    : TEXCOORD0;
   float3 Distance    : TEXCOORD1;
```

```
   float3 LightDir    : TEXCOORD2;
   float3 Normal      : TEXCOORD3;
};

uniform float2 KnotsST[4];
uniform float4 Knots05;
uniform float4x4 PointsX;
uniform float4x4 PointsY;
uniform float4x4 PointsZ;
uniform float4x4 Weight;
uniform float3 LightPos;
uniform float AttenPower;
uniform float AttenPower2;
uniform float4 NN1,NN2;

struct IN
{
   float4 Position      : POSITION;
   float3 Color         : COLOR0;
   float2 TexCoord      : TEXCOORD0;
};

void Nurbs(float2 uv, out float3 V, out float3 Tangent,
           out float3 Bitangent, out float3 Normal)
{
   float4 NormalS,NormalT;
   float4 uv05;
   float2 uv1,uv2,uv3,uv4;
   float4 K1,K2,L1,L2,M;
   float4 S,T;
   float4 s4,t4;
   float4 ss4,tt4;
   float4 ab1,ab2;
```

```
float4 SS,TT;
float4 NS1,NT1;

uv05 = uv.xyxy-Knots05;
uv1 = uv-KnotsST[0];
uv2 = uv-KnotsST[1];
uv3 = uv-KnotsST[2];
uv4 = uv-KnotsST[3];

s4 = float4(uv1.x,uv2.x,uv3.x,uv4.x);
t4 = float4(uv1.y,uv2.y,uv3.y,uv4.y);
ss4 = float4(uv1.x,uv4.x,uv1.y,uv4.y);
tt4 = float4(uv2.x,uv3.x,uv2.y,uv3.y);

L1 = s4.xzyw;
L1 /= NN2.xxyy;
L2 = t4.xzyw;
L2 /= NN2.zzww;

K1.x = dot(float4(1),L1);
K1.y = dot(L1.xz,s4.zw);
K1.z = dot(float4(1),L2);
K1.w = dot(L2.xz,t4.zw);


K2.xyzw = tt4.yxwz/NN1;

M=tt4.yxwz*K2;

SS = M.xyzw*tt4.yxwz;
TT = M.xyzw*uv05.xzyw+K1.yyww*ss4.yxwz;

float4 Q;
```

```
   Q = 2*uv05*K2.xzyw+K1.ywyw;

   NS1 = 3*M;
   NT1 = M+ss4.yxwz*K1.xxzz+Q.xzyw;

   S = float4(SS.x,TT.xy,SS.y);
   T = float4(SS.z,TT.zw,SS.w);
   NormalS = float4(NS1.x,NT1.xy,NS1.y);
   NormalT = float4(NS1.z,NT1.zw,NS1.w);

   float W=dot(T,mul(Weight,S));
   V = float3(dot(T,mul(PointsX,S)),
              dot(T,mul(PointsY,S)),
              dot(T,mul(PointsZ,S)))/W;
   float WSd = dot(T,mul(Weight,NormalS));
   float WTd = dot(NormalT,mul(Weight,S));
   Tangent = (float3(dot(T,mul(PointsX,NormalS)),
                     dot(T,mul(PointsY,NormalS)),
                     dot(T,mul(PointsZ,NormalS))) - V*WSd)/W;
   Bitangent = (float3(dot(NormalT,mul(PointsX,S)),
                       dot(NormalT,mul(PointsY,S)),
                       dot(NormalT,mul(PointsZ,S))) - V*WTd)/W;
   Normal = cross(Tangent, Bitangent);
}

OST main(IN input,
         uniform float4x4 ProjectionAndModelview)
{
   OST Output;
   Output.Color.rgb = input.Color; // MOT result.color.xyz, vertex.color;

   float3 V,T,B,N,L;
```

```
    Nurbs(input.Position.xy,V,T,B,N);

    Output.TexCoord = input.TexCoord;
    Output.Normal = N;
    L = LightPos-V;
    Output.LightDir = L;
    Output.HPosition = mul(ProjectionAndModelview, float4(V,1));

    N = normalize(N);
    T = normalize(T);
    L = float3(dot(T,L),dot(cross(N,T),L),dot(N,L));

    float a = L.z*L.z;
    Output.Color.w = 1/(dot(L,L)*0.0004+1);
    return Output;
}
```

Tento optimalizovaný VS využíva všetky možné dočasne registre v profile vp20, a 127 inštrukcií, a teda máme miesto ešte na jednu inštrukciu. Tá môže byť využitá napr. na posunutie ďalšieho parametra pre vrcholy (napr. farba). Aj keď sme pôvodne optimalizovali na profil vp20 (teda vs 1.4) tak si uvedieme výsledky aj pre ostatné profily.

Metóda	FPS	
	Presnosť	
	10	30
VS - VP20	275	201
VS - VP30	275	201
VS - ARBVP1	540	320
VS - VP40	540	320
CPU(optimalizovaný algoritmus)	150	19

tabuľka 1.1: Porovnanie rýchlosti

Pre porovnanie uvedieme aj výsledky pre CPU. Na CPU algoritmus uvedený v tabuľke

1.1 sme kládli rovnaké požiadavky ako na GPU algoritmus a sice pamäťové nároky $O(1)$. Z tabuľky je vidieť že GPU algoritmus naberá na sile pri väčšej presnosti(jemnejšie rozdelenie parametrického priestoru). Pri menšej presnosti sa častejšie aktualizujú uniformné parametre VS ako napr. geometria ďalšieho segmentu.

1.7 Budúca práca

Nech by sme sa ako koľvek snažili vždy sa niektoré vrcholy budú rátať viac ako raz. Toto sa však bude dať obísť z veľkej časti. Až tak že v rámci segmentu sa žiadny vrchol nebude počítať viac ako raz. Toto ale bude možné až na grafických kartách ďalšej generácie s podporou *SM 4.0*. Ktorá okrem *unified shader* prináša aj novú programovaciu jednotku *geometry shader*. Ktorý je schopný generovať trojuholníky na grafickej karte. Takže by sa na grafickú kartu poslali iba predpočítané parametre a na grafickej karte by sa spravil cyklus kreslenia segmentu ktorý je teraz na CPU. Grafické karty tohto druhu budú komerčne dostupné zhruba na konci roka 2006.

Kapitola 2

Celulárne automaty

2.1 Úvod

Celulárne automaty sú systémy, ktoré fungujú na mriežke. Element systému je bunka. Systém sa pre každú bunku správa rovnako. To ako sa daná bunka zmení, záleží na nej a okolitých bunkách. Naše rozšírenia týchto klasických automatov poskytujú nové možnosti. Keďže sa zameriavame na GPU, kde najmenšia možná jednotka výstupu je fragment, tak v našom programe je element jeden fragment. Na popis správania sa bunky máme jazyk pomenovaný Texture Language (1) , ktorý v tejto práci rozšírime a implementujeme na GPU.

2.1.1 Implementačný pohľad

Program napísaný v našom jazyku je prevedený na GLSL (2) shader a ten je následne skompilovaný na GPU v ovládačoch grafickej karty. Dôvod prečo sme zvolili GLSL je že GLSL je platformovo nezávislí a kompilátor na GLSL je priamo v ovládačoch grafickej karty. Takže za kompiláciu nezodpovedá tretia strana. Program na výpočet používa MRT (multiple render targets). Dnešné GPU (GeForce 6800) dovoľujú zapisovať naraz až do 4 textúr naraz. V našom programe dovoľujeme zapisovať iba 2 textúr (v druhých 2 je uložená predchádzajúca iterácia). Každá textúra je RGBA(128 bitová - 4x32 bitové reálne čísla). Vďaka vysokej presnosti(fp32) môžeme generovať aj HDR(high dynamic range) mapy. MRT bufre sa správajú tak ako obyčajné textúry, teda sa im dajú nastaviť vlastnosti tak ako ostatným textúram napr. filtrovanie atd.. Avšak na dnešnom GPU všetko nefunguje ako

by malo. A niektoré vlastnosti sú nastaviteľné iba za istých podmienok (napr. ak chceme nastaviť fp32 tak nemôže byť naša textúra ”power of two”).Niektoré vlastnosti sa nedajú nastaviť vôbec (napr. ”clamping”, vždy sa nastaví vlastnosť GL_CLAMP_TO_EDGE). Pred príchodom MRT sa na tieto účely. používalo renderovanie do textúry. V porovnaní s MRT (Gray-Scottov reakčný difúzny systém [1]) by však tento prístup bol 15 krát pomalší na tom istom hardvery. A podporoval by iba výstup do jednej textúry.

2.2 Popis jazyka

2.2.1 Premenné a funkcie

V tomto odseku si popíšeme všetky možné funkcie a premenné ktoré má užívateľ k dispozícií v Texture Language.

+	−	∗	/	>	<	<=	>=	=

tabuľka 2.1: Operátory

Sin(x)	$\sin(x)$	Cos(x)	$\cos(x)$
Sqrt(x)	$\sqrt{x}$	Sqr(x)	x^2
Abs(x)]	$\lvert x\rvert$	Ln(x)	$\ln(x)$
Floor(x)	$\lfloor x \rfloor$	Round(x)	Zaokrúhli číslo x
Frac(x)	Desatinná časť x		

tabuľka 2.2: Unárne funkcie

Exp(x, y)	x^y	Atan(x, y)	$\arctan\left(\frac{x}{y}\right)$
Div(x, y)	$\left\lfloor \frac{x}{y} \right\rfloor$	Mod(x, y)	$x - y\left\lfloor \frac{x}{y} \right\rfloor$
Max(x, y)	$\max(x, y)$	Min(x, y)	$\min(x, y)$

tabuľka 2.3: Binárne funkcie

Random	Náhodné číslo od 0 do 1
center[vrstva]	Hodnota v bunke
upper[vrstva]	Hodnota v hornej susednej bunke
lower[vrstva]	Hodnota v dolnej susednej bunke
left[vrstva]	Hodnota v ľavej susednej bunke
right[vrstva]	Hodnota v pravej susednej bunke
ul[vrstva]	Hodnota v ľavej hornej susednej bunke
ur[vrstva]	Hodnota v pravej hornej susednej bunke
ll[vrstva]	Hodnota v ľavej dolnej susednej bunke
lr[vrstva]	Hodnota v pravej dolnej susednej bunke
cell(x, y)[vrstva]	Hodnota v bunke x, y v danej vrstve
texN(x, y)[vrstva]	Vrstva môže byť 0-3 (RGBA). Namiesto N sa dosadí číslo textúry z ktorej chceme ťahať hodnotu. K dispozícií máme tex0 až tex12
time	Doba trvania behu programu. Doba sa meria v počte iterácii a nie podľa času.
x	x-ová pozícia bunky
y	y-ová pozícia bunky
maxX	šírka priestoru nad ktorým program pracuje
maxY	výška priestoru nad ktorým program pracuje

tabuľka 2.4: **Ostatné.** *Pre všetky ťahania hodnôt (cell, texN, center, atd.) sa používa GL_REPEAT.*

2.2.2 Programovanie v Texture Language

Texture Language nepodporuje cykly, vlastné premenné, programové skoky a pod. Podporuje iba podmienku a priradenie výsledku. V každom kroku je možné pristúpiť k ľubovoľnej bunke v ľubovoľnej vrstve z predchádzajúceho kroku. Čo nám dodáva výpočtovú silu oproti klasickým celulárnym automatom, ktoré môžu pristupovať iba k 8 susedom a bunke samotnej.
Každý program v Texture Language popisuje správanie jednotlivých vrstiev. Na odlíšenie časti programu pre konkrétnu vrstvu sa použije jej číselný identifikátor v hranatých zátvorkách:

```
[0]
:1;
```

tabuľka 2.5: Automat na generovanie červenej textúry(pri štandardnom nastavení)

Ďalej sa nachádza časť pre výpočet hodnoty bunky pre danú vrstvu. Môže byť priamo zapísaná konkrétna hodnota viď. vyššie, alebo môže byť podmienka. Použitím podmieňovacieho príkazu sa určí, ktorá hodnota sa má do bunky zapísať

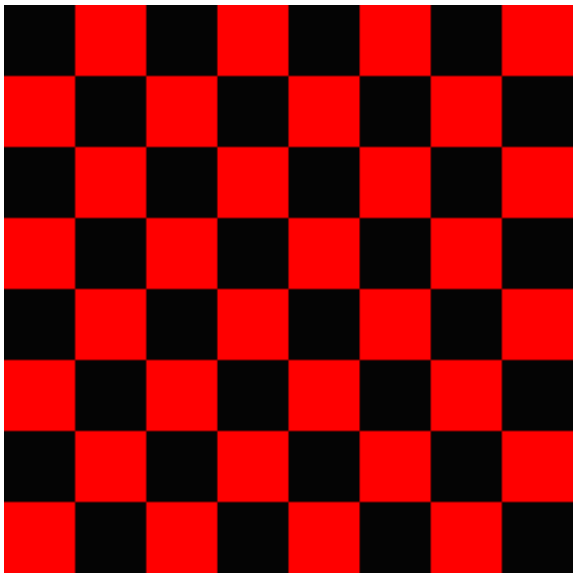
```
[0]
mod(div(x,div(maxX,8))+div(y,div(maxY,8),2)=0:1;
:0;
```

tabuľka 2.6: Generovanie šachovnice 8x8 (obr. 2.1)

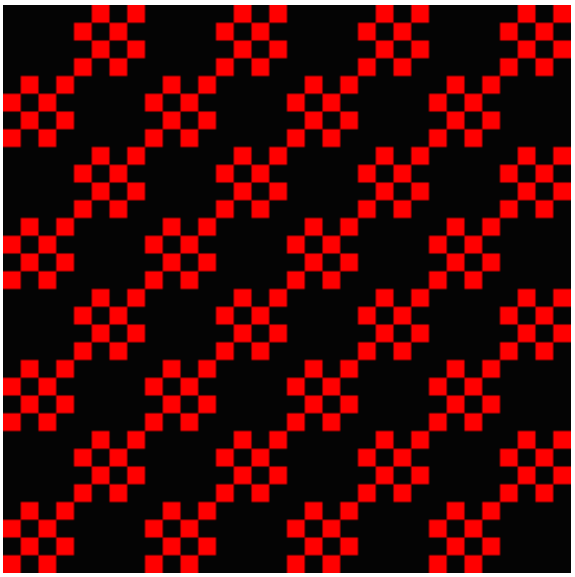
V tomto príklade(2.6) sa rozhodne podľa pozície bunky. V podmienke môže byť porovnanie ľubovoľných výrazov. Za nimi nasleduje výstup pre kladnú vetvu podmienky. A potom musí nasledovať časť pre zápornú časť podmienky. Interpretovaný program končí výpočet pre danú vrstvu ak je priradená nejaká hodnota výsledku. Ako sme už spomenuli jednotlivé podmienky sa môžu do seba vnárať. Jednoduché vnorenie si ukážeme na príklade.

```
[0]
mod(div(x, div(maxX, 8))+div(y, div(maxY, 8), 2)=0:
  mod(div(x, div(maxX, 32))+div(y, div(maxY, 32), 2)=0:1;
  :0;
:0;
```

tabuľka 2.7: Generovanie šachovnice kde biele políčka sú šachovnice 4x4 (obr. 2.2)



obrázok 2.1: Šachovnica (tab. 2.6)



obrázok 2.2: Šachovnica (tab. 2.7)

2.3 Na čo si treba dať pozor pri programovaní v Texture Language

2.3.1 Presnosť

Aj keď sa zdá že fp32 by mala byť postačujúca, treba si dať pozor na prácu s vysokými číslami. Problem presnosti :

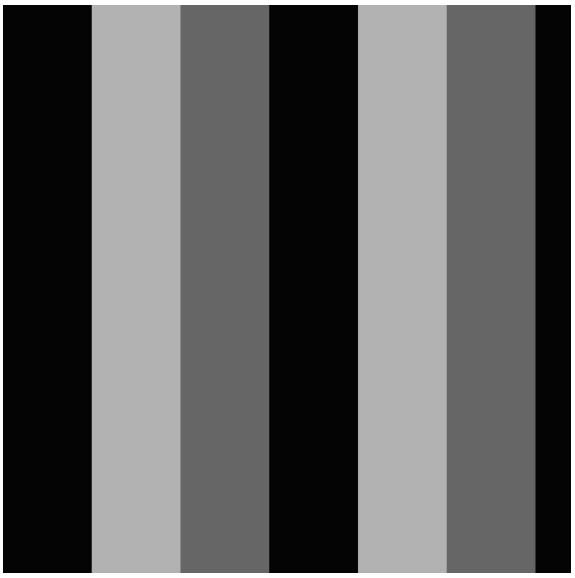
```
[0]
:sin(x/40+time/40);
[1]
:sin(x/40+time/40);
[2]
:sin(x/40+time/40);
```

tabuľka 2.8: Generovanie sínusoidy (obr. 2.3)

```
[0]
:sin(x/40+time/40+20000000);
[1]
:sin(x/40+time/40+20000000);
[2]
:sin(x/40+time/40+20000000);
```

tabuľka 2.9: Generovanie sínusoidy s veľkými číslami (obr. 2.4)

obrázok 2.3: Generovanie sínusoidy (tab. 2.8)



obrázok 2.4: Generovanie sínusoidy s veľkými číslami (tab. 2.9)

2.3.2 Náhodné čísla

Ak by ste sa pozreli do špecifikácie GLSL, našli by ste funkcie pre náhodné čísla na GPU. A to noise1, noise2, noise3 a noise4. Avšak v čase písania tejto práce žiadne komerčné GPU ešte nemá implementované tieto funkcie, a preto sa aj náš Texture Language musel zaobísť bez týchto funkcií. Náhodne čísla nie sú teraz počítané, ale sú ťahané z textúry kde je náhodný šum. To ktorá hodnota sa vytiahne z textúry záleží na čase (iterácií) a pozícií bunky. Takto vieme zaručiť rovnaké náhodne čísla pri každom spustení. Avšak nevýhodou je že v rámci jednej iterácie pre jednu bunku môžme mať len jednu náhodnú hodnotu.

2.4 Využitie niektorých rozšírení

V tejto časti si ukážeme akú silu pridali niektoré rozšírenia.

2.4.1 Time

Využitie tohto rozšírenia bolo možné vidieť na predchádzajúcom príklade zo sínusom (tab. 2.8).

2.4.2 Pozícia bunky

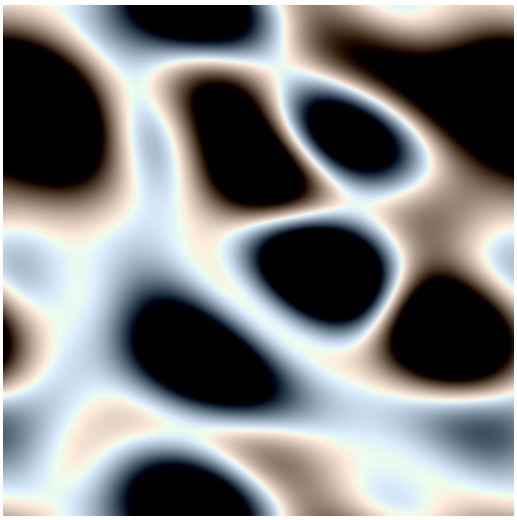
Pozícia bunky je daná v celých číslach. Bunka s pozíciou 0,0 sa nachádza v rohu vľavo dole. Pozícia je využívaná pri väčšom množstve procedurálnych textúr. Jednoduché využitie je tiež ukázane v TLA 4. Ako príklad si ukážeme jednoduché generovanie tzv. plazmy.

2.4.3 tex*N*

Funkcia texN slúži na vybratie hodnoty z konštantného bufra, kde ako v tomto príklade môže byť uložená textúra.

```
[0]
:1-abs((center[5]+(0.42-center[5])*center[4]/0.42))/5;
[1]
:1-abs((center[5]+(0.42-center[5])*center[4]/0.42)-1.0/3.0)/5;
[2]
:1-abs((center[5]+(0.42-center[5])*center[4]/0.42)-2.0/3.0)/5;
[4]
:sin(x/80+time/40)+sin(y/40+3*time/80)+sin(x/40+y/80-time/80)
  +sin(x/80+y/80);
[5]
:sin(x/80+time/60)+sin(y/40+time/40)+cos(x/80-y/80+time/60)
  +sin(x/80+y/40-time/60);
```

tabuľka 2.10: Plazma (obr. 2.5)



obrázok 2.5: plazma (tab. 2.10)

Vo vrstve 4 a 5 sa generuje klasická procedurálna textúra známa tiež ako plazma. Pre lepší efekt tieto dve plazmy interferujú podľa vzorca (((center[5] + (0.42 - center[5]) * center[4] / 0.42) a následne je aplikované akési paletovanie. Na lepšiu ukážku využitia rozšírenia Time je potrebné zhliadnuť videa na priloženom DVD.

2.4.4 cell

Možnosť ťahať z ľubovoľnej bunky.

```
[0]
:tex0(maxX/2+cos(center[3])*center[4],maxY/2+
  sin(center[3])*center[4])[0];
[1]
:tex0(maxX/2+cos(center[3])*center[4],maxY/2+
  sin(center[3])*center[4])[1];
[2]
:tex0(maxX/2+cos(center[3])*center[4],maxY/2+
  sin(center[3])*center[4])[2];
[3]
:atan(x-maxX/2,y-maxY/2)+mod(time/40,3.141592*2);
[4]
:sqrt(sqr(x-maxX/2)+sqr(y-maxY/2));
```

tabuľka 2.11: Rotácia textúry (obr. 2.6)

Keby sme v 3. vrstve kde sa počíta uhol otočenia, dali namiesto mod(time/40,3.141592*2) dali iba time/40. Tak by sme po chvíli prišli k chybe (tab. 2.4).

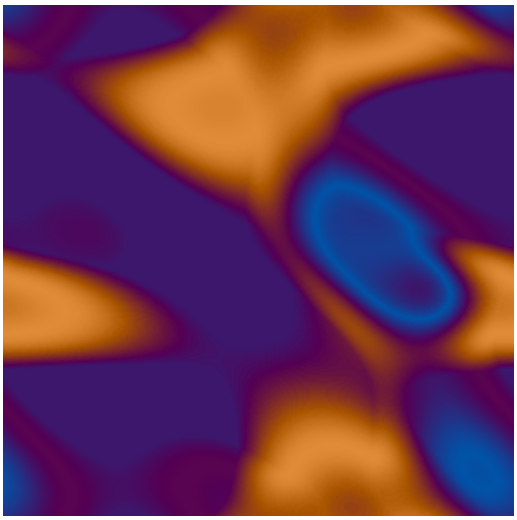
obrázok 2.6: Rotácia textúry (tab. 2.11)

2.5 GUI na programovanie v *Texture Language*

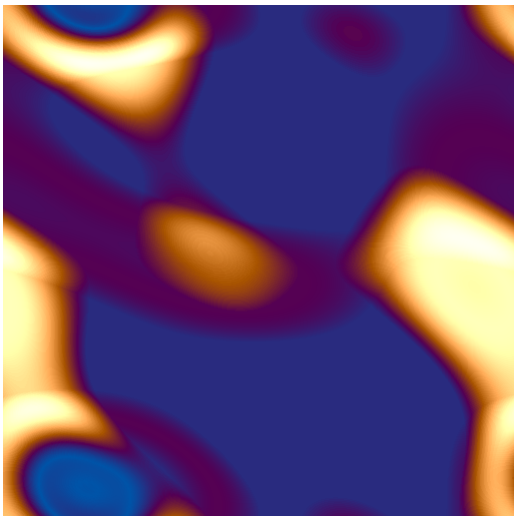
Hlavné okno (obr. 2.9) slúži na pridávanie textúr a nahrávanie TLA súborov. Po spustení máme k dispozícií jednu textúru a to čiernu. Textúry ktoré máme k dispozícií je možné vidieť v Texture Window. Ak chceme nejakej vrstve nastaviť textúru, tak si ju najprv treba

```
[0]
:cell(center[4]*maxX/16,center[5]*maxY/16)[6];
[1]
:abs(cell(center[4]*maxX/16,center[5]*maxY/16)[6]-0.33);
[2]
:abs(cell(center[4]*maxX/16,center[5]*maxY/16)[6]-0.66);
[4]
:sin(x/80+time/40)+sin(y/40+3*time/80)+
  sin(x/40+y/80-time/80)+sin(x/80+y/80);
[5]
:sin(x/80+time/60)+sin(y/40+time/40)+cos(x/80-y/80+time/60)+
  sin(x/80+y/40-time/60);
[6]
:abs((center[5]+(0.42-center[5])*center[4]/0.42))/15;
```

tabuľka 2.12: Plazma (obr. 2.7, 2.8)

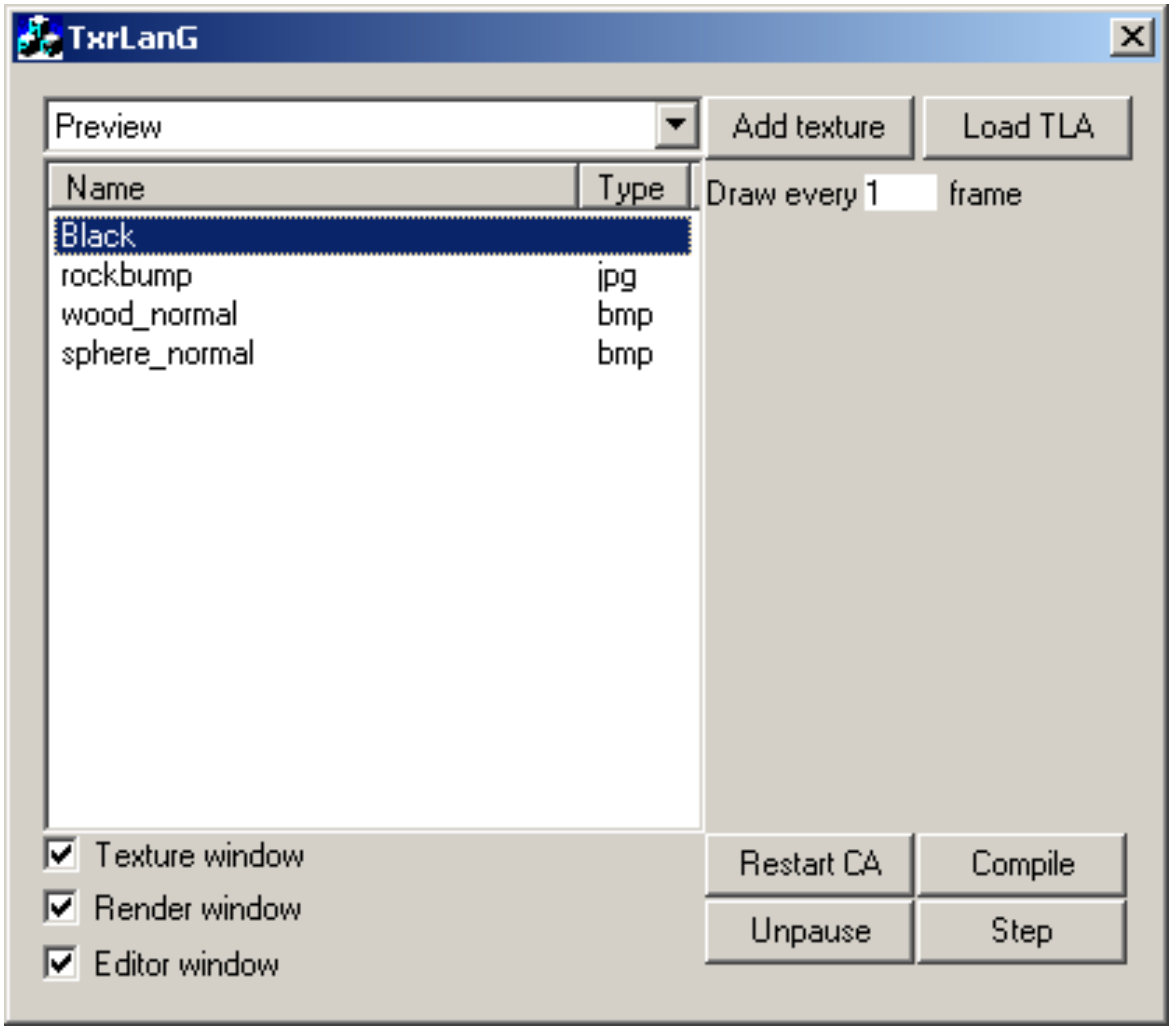


obrázok 2.7: cell (tab. 2.12)



obrázok 2.8: cell (tab. 2.12)

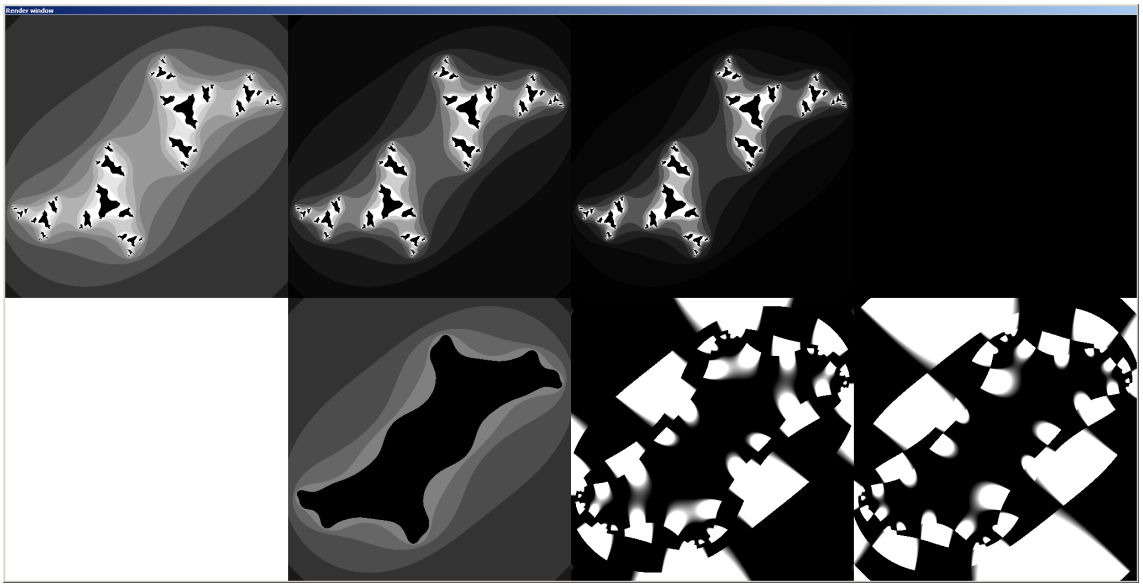
nastaviť v combo boxe.



obrázok 2.9: Hlavné okno

2.5.1 Zobrazovacie okno

V tomto okne vidíme priebeh nášho programu. Užívateľ má možnosť v kontextovom menu zvoliť typ vizualizácie. Štandardne je zapnuté aby sa vrstvy 0,1,2,3 zobrazovali ako RGBA. Ďalšia možnosť je zapnúť aby sa vrstvy 4,5,6,7 zobrazovali ako RGBA. A posledná možnosť je vhodná na akési ladenie programu. Pri tejto metóde sa každá vrstva zobrazí zvlášť vo vizualizačnom okne. Takto užívateľ môže vidieť čo sa deje vo všetkých vrstvách v každom kroku. Je vhodné si pri ladení priebeh času zastaviť a iterovať po jednom kroku. Ako také ladenie môže vyzerať je na obrázku 2.10.



obrázok 2.10: Zobrazovacie okno

2.6 Ďalšie zaujímavé ukážky

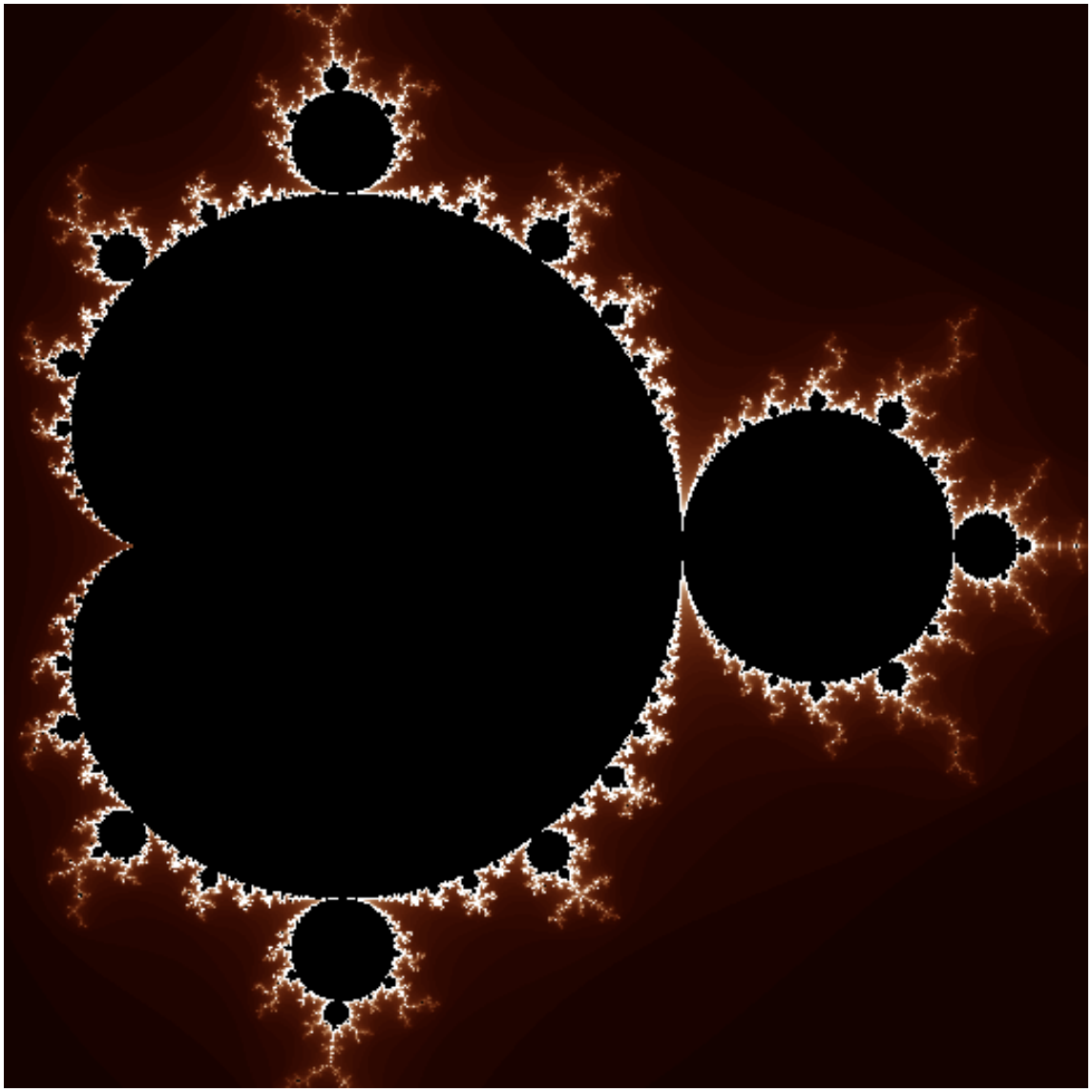
2.6.1 Fraktály

Fraktály sú vhodný kandidát na tvorbu celulárnym automatom.

Mandelbrotova množina je počítaná podľa vzorca $z_{n+1}z_n^2 + c$

```
[0] // červená zložka
center[3]<0.001:0;
:center[3];
[1] // zelená zložka
center[3]<0.001:0;
:center[3]*center[3];
[2] // modrá zložka
center[3]<0.001:0;
:center[3]*center[3]*center[3];
[3] // zistí sa či už daný bod nediverguje
center[3]<0.001:sqrt(sqr(center[4])+sqr(center[5]))>2:time/50;:0;
:center[3];
[4] // ak bod ešte nedivergoval tak iterujeme ďalej x-ovú súradnicu
center[3]<0.5:sqr(center[4])-sqr(center[5])+center[6];
:center[4];
[5] // to isté ako v [4] ale pre y-ovú súradnicu
center[3]<0.5:2*center[4]*center[5]+center[7];
:center[5];
[6] // x-ová pozícia bunky
:(x-maxX/2)/(maxX/2)+0.5;
[7] // y-ová pozícia bunky
:(y-maxY/2)/(maxY/2);
```

tabuľka 2.13: Mandelbrotova množina (obr. 2.11)

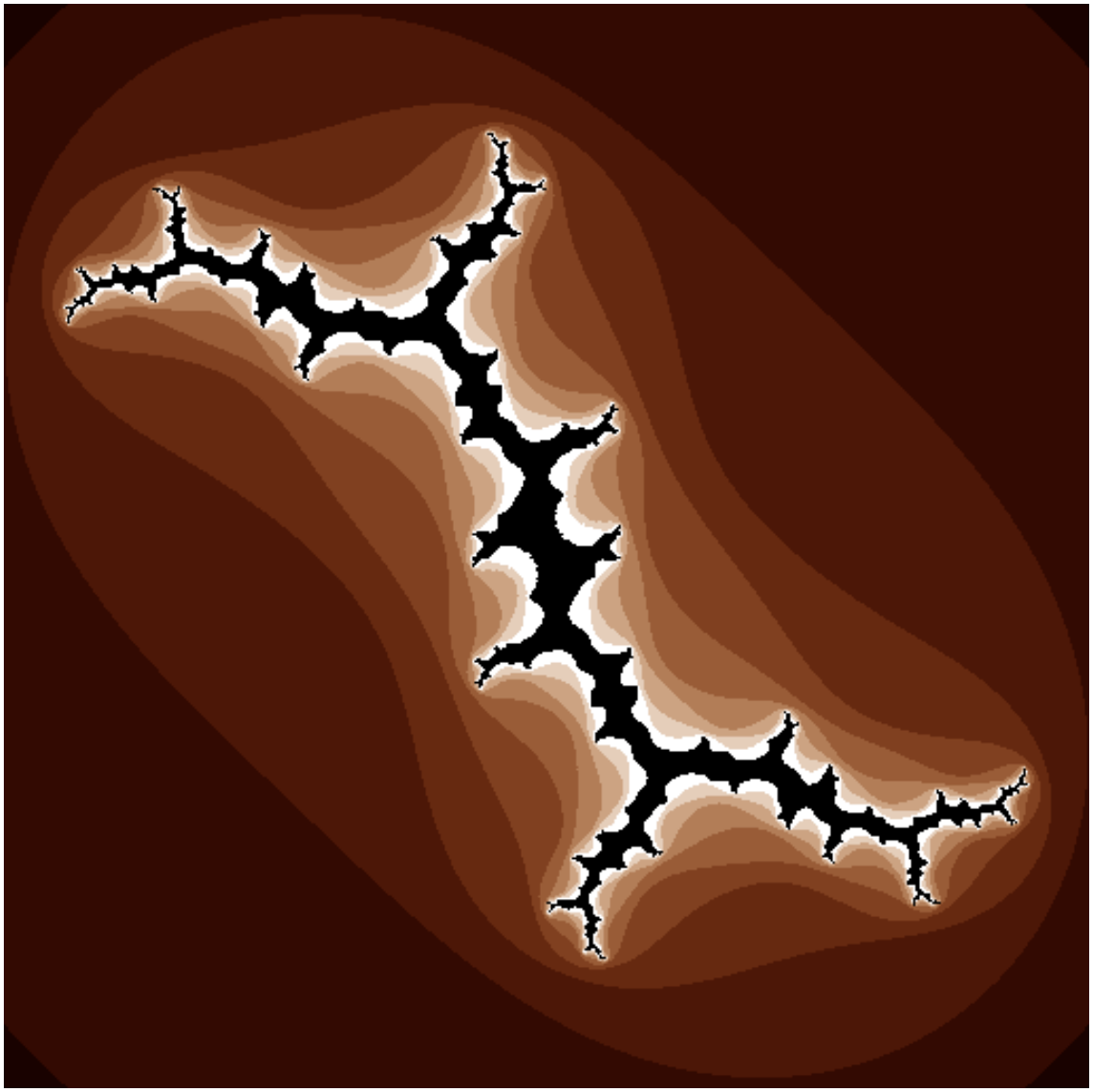


obrázok 2.11: Mandelbrotova množina (tab. 2.13)

```
[0]
mod(time,12)>0:center[0];
center[5]<0.001:0;
:center[5];
[1]
mod(time,12)>0:center[1];
center[5]<0.001:0;
:center[5]*center[5];
[2]
mod(time,12)>0:center[2];
center[5]<0.001:0;
:center[5]*center[5]*center[5];
 [4]
mod(time,12)=0:time/120;
:center[4];
[5]
mod(time,12)=0:0;
center[5]<0.001:sqrt(sqr(center[6])+sqr(center[7]))>2:mod(time,12)/10;
:0;
:center[5];

[6]
mod(time,12)=0:((x-maxX/2)/(maxX/2))*1.5;
center[5]<0.001:sqr(center[6])-sqr(center[7])+sin(center[4]);
:center[6];
[7]
mod(time,12)=0:((y-maxY/2)/(maxY/2))*1.5;
center[5]<0.001:2*center[6]*center[7]+cos(center[4]);
:center[7];
```

tabuľka 2.14: Animovaná Juliová množina (obr. 2.12)



obrázok 2.12: Animovaná Juliová množina (tab. 2.14)

Pri tomto programe sa oplatí nastaviť v prostredí aby zobrazovalo až každú 12 iteráciu. Pretože iba raz za každých 12 iterácií sa obnovia vrstvy 0,1,2 a začne sa výpočet fraktálu znova ale s iným parametrom c v rovnici $z_{n+1} \longleftarrow z_n * z_n + c$.

2.6.2 Raytracing

```
[0]
:tex0(center[3],center[4])[0]/
  (1+1*sqr(center[4]/maxY+3*sin(time/40)));
[1]
:tex0(center[3],center[4])[1]
  /(1+1*sqr(center[4]/maxY+3*sin(time/40)));
[2]
:tex0(center[3],center[4])[2]
  /(1+1*sqr(center[4]/maxY+3*sin(time/40)));
[3]
:atan(center[5],center[6])*maxX;
[4]
:maxY*center[7]/sqrt(sqr(center[5])+sqr(center[6]));

[5]
:(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  cos(time/120)+((-1)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))
*sin(time/120)
[6]
:(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  sin(time/120)*sin(time/60)+(((y/maxY)-0.5)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/60)-((-1)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/120)*sin(time/60);
[7]
:(-1)*(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  sin(time/120)*cos(time/60)+(((y/maxY)-0.5)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*sin(time/60)+((-1)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/120)*cos(time/60);
```

tabuľka 2.15: Tunel s textúrou (obr. 2.13)

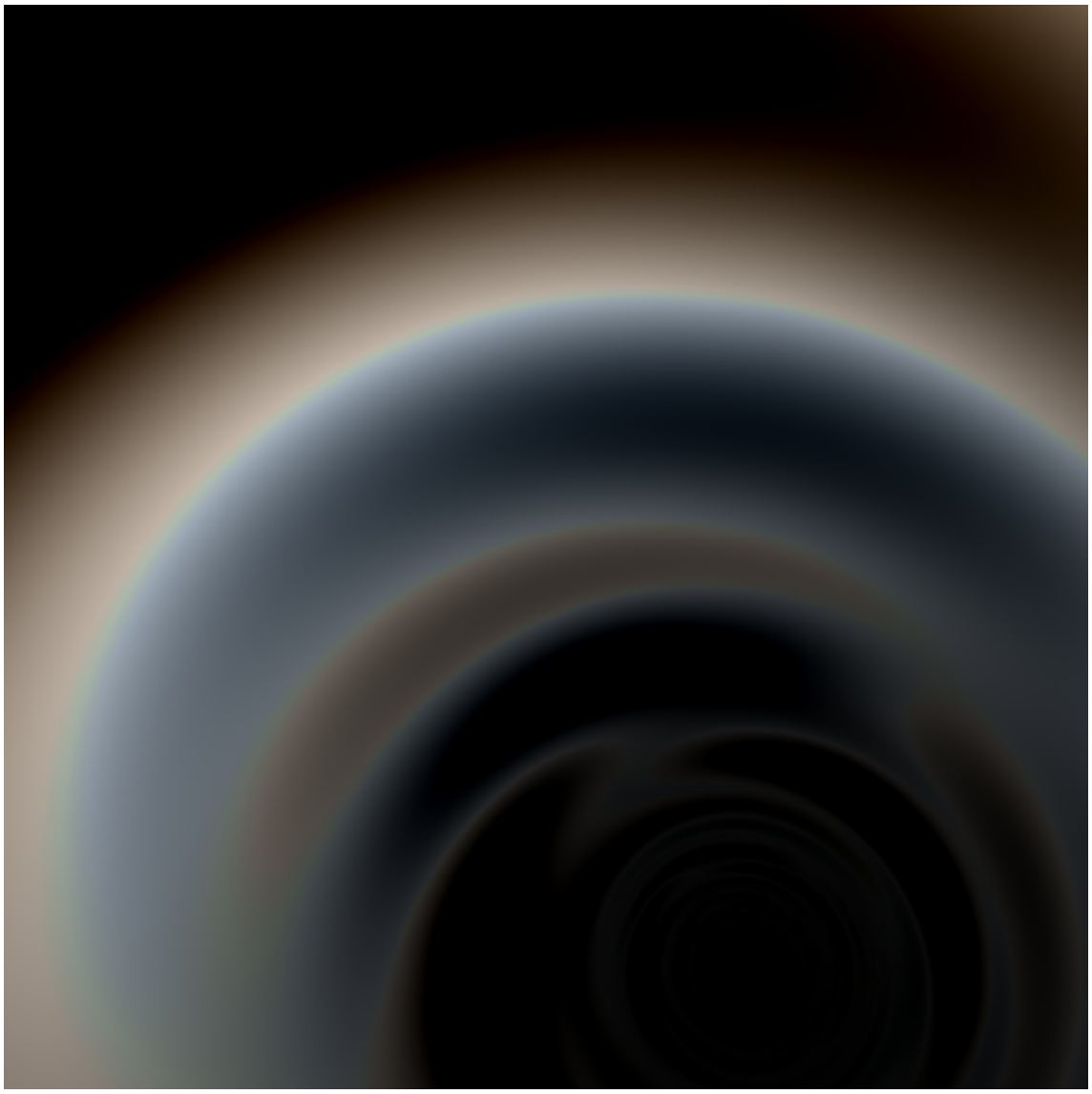
obrázok 2.13: Tunel s textúrou (tab. 2.15)

```
[0]
:(1-abs((center[4]+(0.42-center[4])*center[3]/0.42))/5)*2/
  (1+sqr(center[7]/sqrt((sqr(center[5])+sqr(center[6])))));
[1]
:(1-abs((center[4]+(0.42-center[4])*center[3]/0.42)-1.0/3.0)/5)*2/
  (1+sqr(center[7]/sqrt((sqr(center[5])+sqr(center[6])))));
[2]
:(1-abs((center[4]+(0.42-center[4])*center[3]/0.42)-2.0/3.0)/5)*2/
  (1+sqr(center[7]/sqrt((sqr(center[5])+sqr(center[6])))));


[3]
:sin(atan(center[5],center[6])+time/40)+
  sin((center[7]/sqrt(sqr(center[5])+
  sqr(center[6])))*2+3*time/80)+
  sin(atan(center[5],center[6])*2+
  (center[7]/sqrt(sqr(center[5])+sqr(center[6])))-time/80)+
  sin(atan(center[5],center[6])+
  (center[7]/sqrt(sqr(center[5])+sqr(center[6]))));
[4]
:sin(atan(center[5],center[6])+time/60)+
  sin((center[7]/sqrt(sqr(center[5])+sqr(center[6])))*2+time/40)+
  cos(atan(center[5],center[6])-(center[7]/
  sqrt(sqr(center[5])+sqr(center[6])))+time/60)+
  sin(atan(center[5],center[6])+(center[7]/
  sqrt(sqr(center[5])+sqr(center[6])))*2-time/60);

[5]
:(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  cos(time/120)+((-1)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*sin(time/120)
[6]
:(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  sin(time/120)*sin(time/60)+(((y/maxY)-0.5)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/60)-((-1)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/120)*sin(time/60);
[7]
:(-1)*(((x/maxX)-0.5)/sqrt(sqr((x/maxX)-0.5)+sqr((y/maxY)-0.5)+1))*
  sin(time/120)*cos(time/60)+(((y/maxY)-0.5)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*sin(time/60)+((-1)/sqrt(sqr((x/maxX)-0.5)+
  sqr((y/maxY)-0.5)+1))*cos(time/120)*cos(time/60);
```

tabuľka 2.16: Tunel s animovanou plazma textúrov (obr. 2.14)



obrázok 2.14: Tunel s animovanou plazma textúrov (tab. 2.16)

Kapitola 3

Reliéfne textúrové mapovanie pre zakrivené plochy

3.1 Úvod

Na úvod si vysvetlíme niektoré osvetľovacie modely, povieme si ich použitie a nevýhody. Potom si v stručnosti povieme čo je to *reliéfne textúrové mapovanie*. A ukážeme si problémy ktoré tato kapitola ma za úlohu riešiť. A to reliéfne textúrové mapovanie pre krivé plochy.

3.2 *Bump mapping*

Bump mapping je všeobecne známy osvetľovací model. Ide o dojem hrbolatosti povrchu pomocou zmeny normály, vďaka normálovej mape. A teda sa nepočíta s normálou polygónu. Niekedy je táto metóda označovaná *normal mapping*(teda mapovanie normály). Vyplýva to z toho že na polygón sa namapuje textúra ktorá ma uložené v texeloch normály. Keďže textúra (pokiaľ nie je v móde fp16 alebo fp32) ma hodnoty $0 \ldots 255$. Táto normála sa použije pri osvetľovacom modely, či už je to phong, alebo niečo iné.

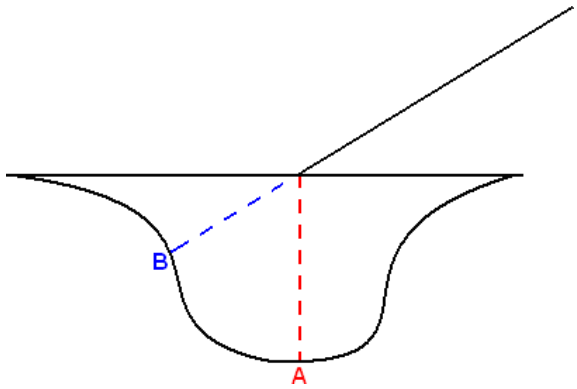
```
uniform sampler2D t2Diffuse;
uniform sampler2D t2Normal;
uniform float3 LightPos;
uniform float3 EyePos;

void main(
        in float4 HPosition    : POSITION,
        in float3 LightVec     : TEXCOORD0,
        in float3 Tangent      : TEXCOORD1,
        in float3 Binormal     : TEXCOORD2,
        in float3 Normal       : TEXCOORD3,
        in float2 TexCoord     : TEXCOORD4,
        in float3 EyeVec       : TEXCOORD5,
        in float4 WorldPos     : TEXCOORD6,

        out float4 color0 : COLOR0
    )
{
   LightVec = normalize(LightVec);
   EyeVec   = normalize(EyeVec);
   color0 = tex2D(t2Diffuse,TexCoord.xy);

   float3 NormalBump = tex2D(t2Normal, TexCoord.xy).xyz;
   NormalBump = normalize(NormalBump*2-1);
   float NdotL = dot(NormalBump.xyz,LightVec);
   float Lighting = NdotL;
   if (NdotL > 0)
   {
      float3 RefVec = 2*NdotL*NormalBump.xyz-LightVec;
      float EdotR = pow(saturate(dot(EyeVec,RefVec)),8);
      Lighting += EdotR;
   }
   color0 *= Lighting;
}
```

tabuľka 3.1: Bump mapping



obrázok 3.1: Chyba mapovania normály

3.3 *Parallax mapping*

Táto metóda je podobná mapovaniu normály. Do programu pridáme iba 2 riadky:

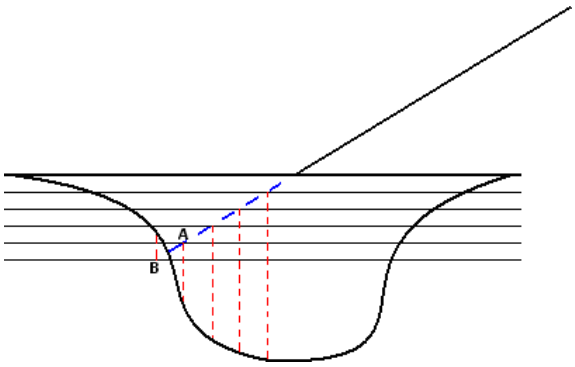
```
float HeightScaleBias = (1-Height)*Scale-Bias;
float2 NewCoord = HeightScaleBias*EyeVec.xy+TexCoord;
```

Parametre *scale* a *bias* sú rôzne pre rôzne materiály. Získaním nových textúrových koordinátov získame dojem že daná textúra je 3D. Na dnešných grafických kartách sa oplatí namiesto metódy mapovania normál použiť. Pretože náročnosť nie je oveľa vyššia oproti tomu ako nám stúpne vizuálna stránka.

3.4 Reliéfne textúrové mapovanie

Predchádzajúce modely stále neposkytli dostačujúci vizuálny efekt. Pri klasickom mapovaní normály by sa zobral z textúry bod A(z normálovej mapy) a normála v bode A. Ale reálne by mal lúč doraziť až do bodu B. Preto treba zistiť polohu bodu B. Pri dopade do roviny sa zoberie vektor dopadu. A bude sa sledovať kde lúč z kamery do tohto bodu pretne výškovú textúru. Problém v tejto metóde je že tento povrch nie je opísaný žiadnou funkciou. Vyrábať funkciu pre každí možní povrch je nad rámec tejto práce. Náš povrch je ale dobre popísaný už spomínanou textúrou. Preto namiesto hľadania presného prieniku sa použije lineárne[1] sledovanie lúča. Keď lúč dopadne na povrch, vieme že je nad výškovou

[1] ide sa po krokoch rovnakej veľkosti až pokiaľ nenastane prienik

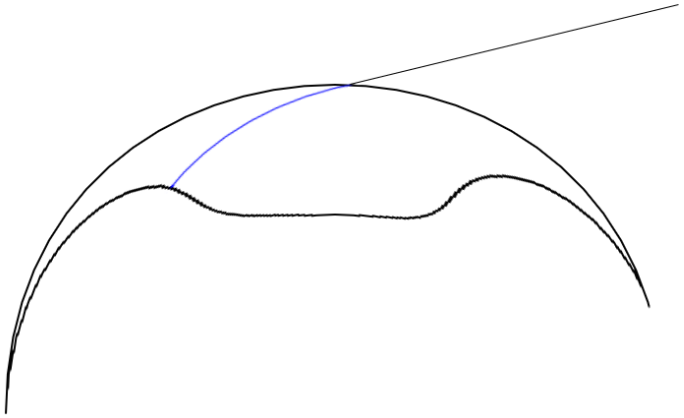


obrázok 3.2: Detekcia výšky

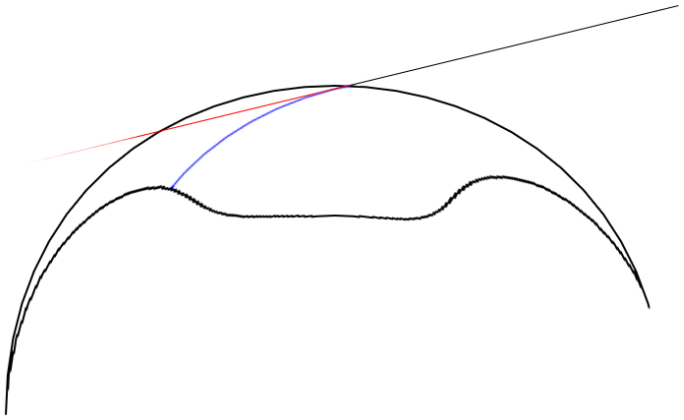
textúrov. Teraz vstupuje do priestoru výškovej textúry. Ak sa nám pri tomto postupnom hľadaní podarí náisť bod pod povrchom. Tak je jasné, že lúč musel niekde preťať túto textúru a teda nami hľadaný bod je medzi aktuálnym bodom testovania a predchádzajúcim. Na upresnenie jeho pozície použijeme binárne vyhľadávanie. Binárne vyhľadávanie môže prísť k zlému výsledku. Keďže výška na danom úseku môže niekoľko krát preťať priamku sledovaného lúča. Avšak nás to netrápi pretože rovnako sme mohli minúť správny prienik aj pri postupnom hľadaní (viz obr. 3.2, testy sa vykonávajú iba na rezoch, takže ak je medzi nimi *tenký kopček*, tak sa nezistí). Možnosť predídenia takýmto chybám neexistuje, môžme zmenšiť krok pri lineárnom hľadaní (to samozrejme vedie k nižšej rýchlosti). Avšak ako vidieť na obrázku 3.3, keď povrch nie je rovný ale zakrivený, tak lúč sa zakrivý rovnako. Ale je jasné že lúč sa kriviť nemá. Na obrázku 3.4 je vidieť ako sa algoritmus správa(modrá farba) a ako by sa mal správať(červená farba). A o tom je aj táto kapitola. Ako sa má sledovať lúč pri zakrivenom povrchu.

3.5 Reliéfne textúrové mapovanie pre zakrivené plochy

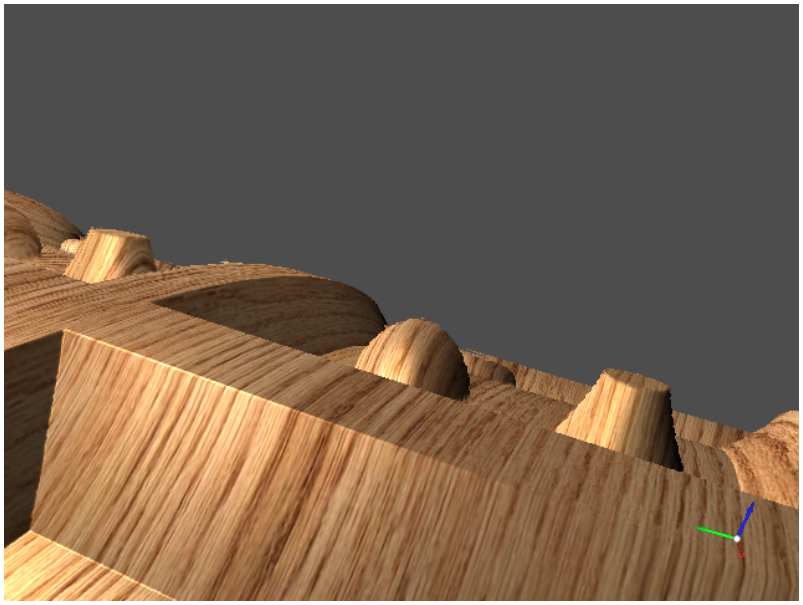
Riešenie nie je komplikované, ale prináša nové požiadavky na hardver. Ako vidieť z obrázku 3.4 počet krokov už nebude vopred známy. Preto budeme potrebovať všeobecný cyklus(while) a nie len s pevným počtom opakovaní(for). Rozdiel oproti predchádzajúcemu prístupu je v tom, že zlom lúča musí byť prepočítaný pri každom kroku. Keďže lúč nemusí byť kolmý na rovinu(trojuholník), jeho projekciou do roviny bude úsečka. To znamená, že aj jednotlivé body, v ktorých sa zisťuje prienik s výškovou mapou, budú mať svoje



obrázok 3.3: Chyba pri zakrivenom povrchu

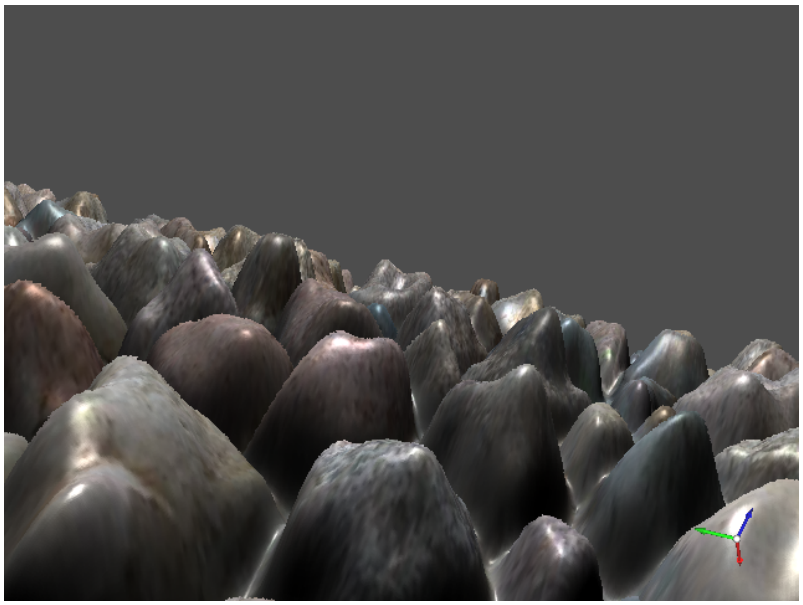


obrázok 3.4: Správne sledovanie lúča pri zakrivenom povrchu



obrázok 3.5: Vysledné reliéfne mapovanie pre zakrivené povrchy

projekcie na tejto úsečke(nie nutne v tom istom bode). Je samozrejme, že k rôznym bodom úsečky prislúchajú rôzne tangenciálne priestory. A teda by sa v týchto bodoch mal inak lámať lúč. Problémom zostáva iba získať tento tangenciálny priestor. Toto sa už ale dá získať jednoduchou matematikou pomocou barycentrických súradníc. Na prevod textúrovych súradníc do barycentrických postačuje matica o veľkosti 2x2. V každom kroku sa musí lúč zalomiť podľa príslušného tangenciálneho priestoru. Takto sa získa korektnejšie zakrivenie. Binárne vyhľadávanie zostáva nezmenené, z rovnakého dôvodu. Riešenie viditeľnosti ak nahodou lúč nepretína v žiadnom bode výskovú mapu je cez alfa maskovanie. Výsledný efekt je možné vidieť na obrázkoch.



obrázok 3.6: Vysledné reliéfne mapovanie pre zakrivené povrchy

Zoznam použitej literatúry

[1] RNDr. Peter Borovský: *Procedural Textures*, diploma thesis, Facutly of Mathematics and Physics, Comenius Univesrity, Bratislava, Slovakia (2000)

[2] www.opengl.org

[3] Eric Lengyel, *Mathematics for 3D Game Programming and Computer Graphics , Second Edition*, Charles River Media, 2003, ISBN: 1584502770

[4] http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/RTM.pdf