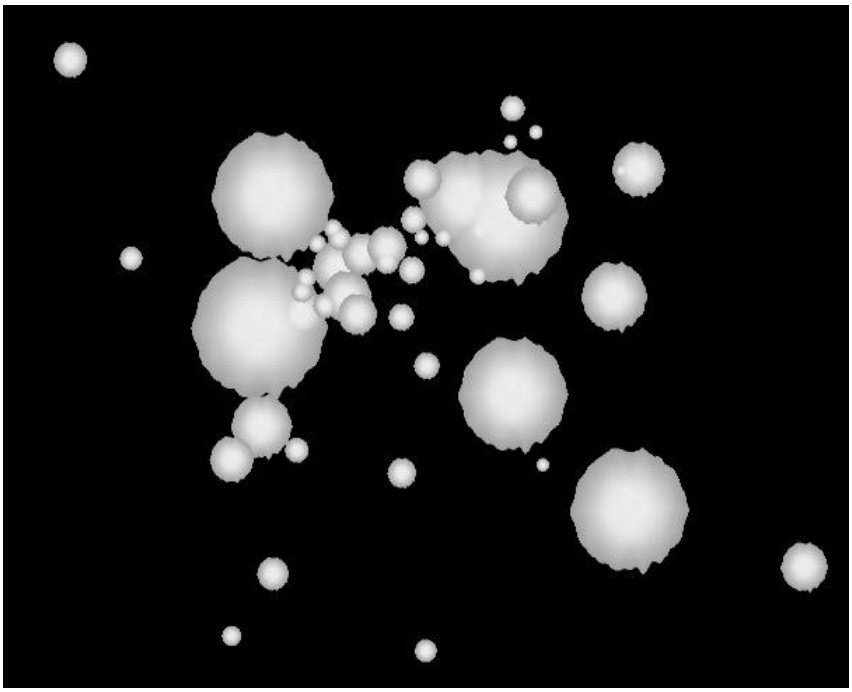
Galaktyka

Ten tutorial nie dotyczy bezpośrednio grafiki komputerowej, ale pokazuje kilka dodatkowych technik związanych z programowaniem shaderów, które nie pojawiły się do tej pory.



Będziemy symulować system cząstek, oddziaływujących grawitacyjnie ze sobą nawzajem, wykonując wszystkie obliczenia w shaderach.

Cząstek będzie 64. Renderowane będą jako bilbordy, wszystkie w jednym przebiegu. Siatka użyta w tym przebiegu składa się z 64 kwadratów jednostkowych, przy czym każdy ma w swoich wierzchołkach przypisane identyczne współrzędne tekstury, odpowiadające środkowi jednego z tekseli tekstury o wymiarach 8x8. Wykorzystamy tutaj funkcjonalność, pozwalającą odczytać dane z tekstury w shaderze wierzchołków „vertex texture fetch". Funkcjonalnośc ta współdziała tylko z określonymi formatami tekstur, w tym A32B32G32R32F.

Położenia i prędkości cząstek są pamiętane w teksturach 8x8, o formacie, w którym dane pojedynczego teksela są czwórką zmiennych rzeczywistych pojedynczej precyzji (32-bitowy float). W sumie wykorzystywane będą dwie tekstury przechowujące położenie cząstek i dwie tekstury przechowujące prędkości cząstek – jedna z każdej pary reprezentuje wartości z poprzedniego kroku symulacji, druga z aktualnego.

Poza przebiegiem renderującym bilbordy pojawią się jeszcze dwa inne przebiegi. Będą one skonstruowane według tego samego schematu. Renderowany będzie kwadrat wypełniający cały obszar widoku, z jedną z tekstur położenia i jedną z tekstur prędkości nałożonymi na jego powierzchnię. Wynik renderowania kwadratu zapisywany będzie jednocześnie do dwóch pozostałych tekstur położenia i prędkości (wykorzystamy funkcjonalność „multiple render targets"). Dzięki odpowiedniemu doborowi współrzędnych pozycji i współrzędnych tekstury wierzchołków kwadratu, uzyskany efekt będzie taki, że dla każdej pary tekseli tekstur, do których renderujemy, zostanie wywołany dokładnie jeden raz shader pikseli. Co więcej ten

shader pikseli otrzyma na wejściu parę współrzędnych, które pozwolą odczytać odpowiednie dane ze źródłowych tekstur.

Jeden przebieg renderowania będzie służył policzeniu wartości położenia i prędkości w kolejnym kroku symulacji na podstawie tych z poprzedniego kroku symulacji. Wykorzystamy tutaj zwykłą jawną metodę Eulera numerycznego rozwiązywania równań różniczkowych zwyczajnych (obliczamy przyspieszenie, prędkość zwiększa się o przyspieszenie pomnożone przez krok czasu, położenie zwiększa się o prędkość pomnożoną przez krok czasu). Drugi przebieg posłuży do przekopiowania policzonych danych tak, żeby obie tekstury położenia i prędkości zawierały te same informacje. Ten krok jest niezbędny, ponieważ w Rendermonkey nie można posługiwać się wskaźnikami na tekstury i podmieniać ich po wykonaniu przebiegu. Przebieg kopiujący posłuży jednocześnie do zainicjowania tekstur, jeśli tylko zostanie stwierdzone, że czas symulacji nie przekracza 0.5 sekundy.

Przyspieszenie cząstki będziemy liczyć zwyczajnie wykonując w piksel shaderze w pętli 64 razy obliczenia siły grawitacji pomiędzy tą cząstkę i każdą z 64 cząstek. Nie będziemy wykonywali obliczeń jeśli odległość między dwoma cząstkami będzie zbyt mała – to zapobiegnie sprawdzaniu cząstki z samą sobą oraz zapobiegnie osiąganiu zbyt dużych przyspieszeń dla bliskich cząstek, co mogłoby zdestabilizować obliczenia numeryczne.

1. Utwórz nowy projekt w Rendermonkey i dodaj do jego gałęzi następujące obiekty

    a) dwie tekstury dwuwymiarowe: jedną o nazwie random z pliku rgbarand.bmp (plik znajduje się w materiałach dołączonych do tego tutorialu), drugą o nazwie flare z pliku flaretexture.dds (również w dołączonych materiałach)

    b) cztery renderowalne tekstury, bez automatycznego generowania mipmap, o wymiarach 8x8 w formacie A32B32G32R32F o nazwach: particlePosition, particlePosition_back, particleVelocity, particleVelocity_back

    c) macierz matViewProjection o semantyce ViewProjection (jedna z predefiniowanych macierzy)

    d) wektor vViewSide o semantyce ViewSideVector i vViewUp o semantyce ViewUpVector

    e) parametr time o semantyce fTime0_X

    f) parametr timestep i ustal wartość na 0.5

    g) model o nazwie quad z pliku ScreenAlignedQuad.3ds i model o nazwie particles z pliku particles8x8.x (oba są w dołączonych materiałach)

    h) strumień o nazwie quad_stream zawierający tylko pozycję (float3) oraz strumień o nazwie particles_stream zawierający pozycję (float3) i współrzędne tekstury (float2)

2. Utwórz nowy zupełnie pusty efekt DirectX i dodaj do niego trzy przebiegi o nazwach kolejno: Copy_Position, Display, Update_Position

3. W przebiegu Copy_Position:

   a) Ustaw dwa render targety. Oznaczony indeksem 0 skojarz z teksturą particlePosition, a oznaczony indeksem 1 skojarz z teksturą particleVelocity.

   b) Dodaj odnośnik do modelu quad i strumienia quad_stream

   c) Dodaj trzy obiekty tekstur: posTex skojarzony z particlePosition_back, velTex skojarzony z particleVelocity_back, randTex skojarzony z random (filtrowania nie trzeba zmieniać, ale jeżeli już to MinFilter i MagFilter powinny być dla wszystkich ustawione na POINT, a MipFilter na NONE)

   d) Dodaj Render State i ustaw w nim D3DRS_ZENABLE na D3DZB_FALSE (wyłączenie bufora głębokości) oraz D3DRS_CULLMODE na D3DCULL_NONE (brak odrzucania trójkątów zwróconych tyłem)

   e) Treść shadera wierzchołków w tym przebiegu będzie odpowiadała za rozciągnięcie kwadratu na cały ekran i dobranie takich współrzędnych tekstury w jego rogach, żeby piksel shader otrzymywał współrzędne odpowiadające dokładnie środkowi pewnego teksela (można zauważyć przesunięcie o pół teksela)

```
struct VS_OUTPUT
{
   float4 Position : POSITION0;
   float2 TexCoord : TEXCOORD0;
};
VS_OUTPUT vs_main( float4 Position : POSITION0 )
{
   VS_OUTPUT Output;
   Output.TexCoord = 0.5 * (float2(Position.x,- Position.y) +
1.0) + float2(0.0625,0.0625);
   Output.Position = float4(Position.xy, 0.0, 1.0);
   return( Output );
}
```

   f) W treści piksel shadera umieść funkcję obliczającą masę danej cząstki (masy cząstek będą dość zróżnicowane):

```
float mass(float2 index)
{
   return 1.0 + exp(index.x*index.y*4.0)*1.0;
}
```

   Zadeklaruj tekstury i parametry, które będą wykorzystywane:

```
sampler posTex;
sampler velTex;
sampler randTex;
float time;
```

Zadeklaruj typ danych zwracanych przez ps_main (tym razem będą to dwa kolory, po jednym dla każdego render targetu):

```
struct PS_OUTPUT
{
   float4 color[2] : COLOR0;
};
```

Wypełnij funkcję ps_main tak, aby dla czasu symulacji poniżej 0.5 sekundy, liczona była pewna pseudolosowa pozycja początkowa i prędkość cząstki, a dla czasów powyżej kopiowane były dane bezpośrednio odczytane z drugiego zestawu tekstur położenia i prędkości:

```
PS_OUTPUT ps_main(float2 index : TEXCOORD0)
{
   PS_OUTPUT Output;

   if (time <0.5)
   {
      float3 rand_vel = tex2D(randTex, float2(index.x,
index.y*1.3765)).xyz*2.0-1.0;
      float3 rand_pos = tex2D(randTex, float2(0.6357*index.x,
3.13*index.y)).xyz*2.0-1.0;
      Output.color[0] = float4(rand_pos*80.0,1.0);
      Output.color[1] = float4(rand_vel*5.0/mass(index),0.0);
   }
   else
   {
      Output.color[0] = tex2D(posTex, index);
      Output.color[1] = tex2D(velTex, index);
   }
   return Output;
}
```

4. W przebiegu Update_Position wypełnij wszystko identycznie jak w przebiegu Copy_Position z wyjątkiem zamiany rolami tekstur particlePosition z particlePosition_back i particleVelocity z particleVelocity_back (render targets, texture objects). Następnie:

   a) Zmień ustawienie Vertex Shader Target na vs_3_0 (na ekranie edycji treści shadera) oraz Pixel Shader Target na ps_3_0 (wersja 3.0 będzie potrzebna, żeby pozbyć się limitu 64 instrukcji shadera, która występuje w wersji 2.0)

   b) W shaderze pikseli dodaj deklarację zmiennej globalnej

   float timestep;

   , po czym w funkcji ps_main() usuń wnętrze z wyjątkiem pierwszej i ostatniej instrukcji. We wnętrzu ps_main() pobierz wartości położenia i prędkości cząstki z poprzedniego kroku

   ```
   float3 pos = tex2D(posTex, index).xyz;
   float3 vel = tex2D(velTex, index).xyz;
   ```

policz przyspieszenie grawitacyjne od wszystkich 64 cząstek, które są od danej odległe o więcej niż dziesięć jednostek w układzie świata:

```
float3 gravity = float3(0.0,0.0,0.0);
for (int i = 0; i < 8; ++i)
for (int j = 0; j < 8; ++j)
{
 float2 index1 = float2(0.0625+j*0.125,0.0625+i*0.125);
 float3 pos1 = tex2D(posTex, index1).xyz;
 float3 r = pos1-pos;
 float  dist = length(r);
 if (dist > 10.0)
 {
  r = normalize(r);
  float m = mass(index1);
  gravity += m * r / (dist*dist);
 }
}
```

,wykonaj krok całkowania dla jawnej metody Eulera

```
pos += vel * timestep;
vel += gravity * timestep;
```

i zapisz wynik

```
Output.color[0] = float4(pos,1.0);
Output.color[1] = float4(vel,0.0);
```

5. W przebiegu Display

   a) Dodaj odnośnik do modelu particles i strumienia particles_stream

   b) Dodaj Render State i ustaw w nim test alfa (D3DRS_ALPHATESTENABLE na TRUE, D3DRS_ALPHAREF na 0xa0, D3DRS_ALPHAFUNC na D3DCMP_GREATEREQUAL)

   c) Dodaj obiekt tekstury flareTex odnoszący się do flare. Ustaw filtrowanie MagFilter, MinFilter i MipFilter na Linear (dwukrotny klik na obiekt tekstury)

   d) Dodaj Vertex Texture Object (tekstura, która może być czytana w shaderze wierzchołków – ma czerwoną ikonę) o nazwie posTex, odnoszący się do particlePosition

   e) Zmień Vertex Shader Target na vs_3_0 i Pixel Shader Target na ps_3_0 (w oknie edycji shaderów wierzchołków i pikseli). Tylko w wersji 3.0 możliwy jest odczyt tekstury w shaderze wierzchołków.

   f) W shaderze wierzchołków dodaj deklaracje zmiennych globalnych

```
sampler posTex;
float4 vViewUp;
float4 vViewSide;
float size;
float4x4 matViewProjection;
```

struktur

```
struct VS_INPUT
{
   float3 Position : POSITION0;
   float2 TexCoord : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 Position : POSITION0;
   float2 TexCoord : TEXCOORD0;
};
```

oraz funkcji mass() identycznej jak w pozostałych dwóch przebiegach (tylko tam w shaderach pikseli):

```
float mass(float2 index)
{
   return 1.0 + exp(index.x*index.y*4.0)*1.0;
}
```

We wnętrze funkcji vs_main()

```
VS_OUTPUT vs_main( VS_INPUT Input )
{
   VS_OUTPUT Output;
//...
   return( Output );

}
```

wstaw kolejno, pobieranie pozycji cząstki z tekstury:

```
   float4 pos = tex2Dlod(posTex,
float4(Input.TexCoord.xy,0.0,1.0));
```

liczenie współrzędnych tekstury bilbordu:

```
   Output.TexCoord = Input.Position.xy;
```

liczenie współrzędnych wierzchołka bilbordu na podstawie kierunków kamery, pozycji cząstki i jej masy:

```
  float2 dp  = mass(Input.TexCoord.xy) * 5.0 *
(Input.Position.xy*float2(2.0,-2.0)+1.0);
   pos += dp.x * vViewSide;
   pos += dp.y * vViewUp;
   pos.w = 1.0;
```

rzutowanie perspektywiczne do widoku z kamery:

```
Output.Position = mul( pos, matViewProjection );
```

g) Shader pikseli w przebiegu Display jest bardzo prosty i realizuje tylko teksturowanie bez oświetlenia:

```
sampler flareTex;
float4 ps_main(float2 texcoord : TEXCOORD0) : COLOR0
{
   float l = tex2D(flareTex, texcoord).x;
   return( l,l,l,l );
}
```

6. W przebiegu Update_Position wejdź w ustawienia Render State i zmień D3DRS_ALPHATESTENABLE na FALSE (test alfa zostaje włączony po poprzednim przebiegu)

7. Teraz warto w menu Edit wybrać Preferences i na zakładce General ustawić wartość Cycle time for pre-defined ‘time’ wariable (sec) na sensowną liczbę (np. 10 sekund lub mniej w trakcie testowania – co taki okres animacja będzie wywoływana od nowa dla zainicjowanych poprawnie danych, więc tyle maksymalnie trzeba czekać na jej start po każdej zmianie projektu). Symulacja odbywa się dla ustalonej wartości kroku timestep, więc w zależności od czasu renderowania klatki animacji można go próbować zmieniać w pewnym zakresie (po zwiększeniu animacja przebiega szybciej, ale obliczenia są stabilne i poprawne tylko dla małych wartości). W szczególności prędkość animacji zależy od wielkości okna widoku.