Université de REIMS
Champagne-Ardenne
UFR de Sciences
Exactes et Naturelles

Travail d'Etude et de Recherche

DirectX 10

Boris Brugevin
Master I STIC - Informatique

Responsable : P. Mignot

Remerciements

Nous remercions l'université de Reims de nous permettre de réaliser des travaux d'étude et de recherche. Nous remercions tout particulièrement Mr Pascal Mignot pour son encadrement et sa disponibilité pendant toute la durée du TER.

Table des matières

1 Le Geometry Shader et le Stream Output (Boris Brugevin)

J'ai souhaité effectuer un TER avec Mr Pascal Mignot car je me passionne pour le développement 3D temps réel. Ayant déjà eu des cours d'OpenGL et de DirectX 9, j'ai voulu en apprendre davantage. Ce stage m'a donc permis d'entrevoir les nouvelles fonctionnalités de la bibliothèque graphique de Microsoft : Direct3D 10.

1.1 Présentation des fonctionnalités utilisées

Mon travail d'étude et de recherche s'est tout particulièrement porté sur la partie Geometry Shader et Stream Output du nouveau pipeline de DirectX 10. Ce shader fait son apparition entre le Vertex Shader qui effectue le plus souvent des transformations de sommets et le Rasterizer qui s'occupe du clipping, de la projection (division par w) et du calcul de la profondeur du pixel.

Le geometry shader constitue la grosse nouveauté de Direct3D 10, ce nouvel étage du pipeline permet de s'affranchir de la limite "1 sommet entrant/1 sommet sortant" qui existe depuis l'apparition des vertex shader. Le geometry shader travaille sur des primitives complètes (point, segment, triangle). Il a non seulement accès à tous les sommets de la primitive, mais également à ceux des primitives adjacentes le cas échéant.

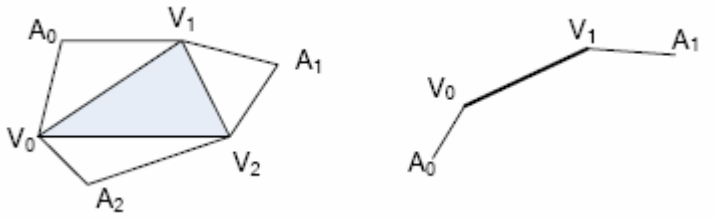


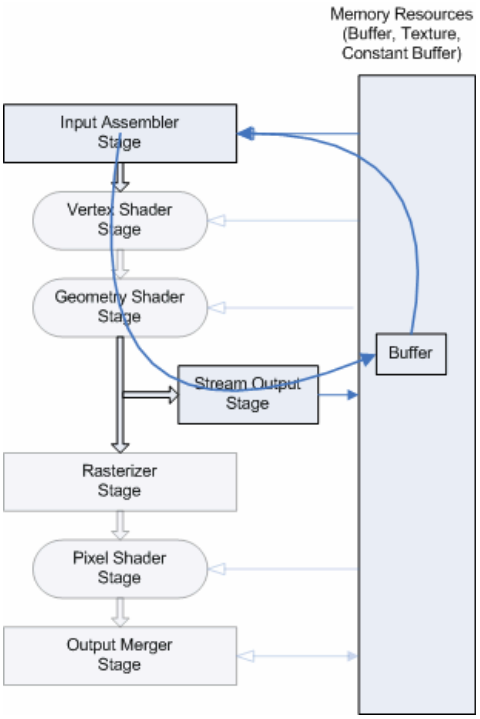
Figure : triangle avec adjacence et ligne avec adjacence

Les geometry shaders sont capables de générer plusieurs sommets en sortie formant ainsi de nouvelles primitives, ou tout simplement de supprimer la primitive courante. La limite des geometry shaders est qu'ils ne peuvent générer plus de 1024 valeurs 32-bit en sortie. Cette limite est d'ailleurs le résultat d'un compromis entre une implémentation matérielle efficace, et les souhaits des développeurs. En effet, d'un point de vue de programmeur les primitives doivent être traitées dans l'ordre dans lequel elles sont envoyées au GPU. En pratique ce n'est pas le cas mais ce doit être transparent pour le programmeur : le matériel doit se comporter exactement comme si l'ordre était préservé. De la même façon les primitives générées par le geometry shader doivent conserver cet ordre ce qui est problématique au niveau de l'implémentation qui repose sur des unités parallèles. Pour résoudre ce problème, le résultat des geometry shaders est stocké dans des buffers, qui sont ensuite traités dans l'ordre d'arrivée.

Les possibilités offertes par ces nouveaux shaders sont multiples : comme il est capable de générer plus de primitives qu'il n'en a reçu, un geometry shader peut effectuer de l'amplification de géométrie. De la même façon il peut générer moins de primitives qu'il n'en reçoit ce qui peut permettre d'implémenter un système de niveaux de détails (LoD) sur le GPU.

L'amplification massive de géométrie même si elle reste possible, ne faisait pas partie des objectifs du geometry shader. Pour cette tâche il était initialement prévu un autre étage : le *Tesselator* qui était visible dans les premières versions du pipeline.

Le rôle du Stream Output est de permettre de stocker les données des primitives en mémoire, on appelle cette technique render-to-vertex-array. Cette technique n'est pas nouvelle mais jusqu'ici elle nécessitait un passage complet dans le pipeline 3D, ce n'est plus le cas désormais. Notons que l'utilisation du Stream Output n'interrompt pas le traitement des primitives : elles peuvent être à la fois stockées en mémoire et continuer leur traitement en direction du rasterizer.



Une fois stockées en mémoire les données des primitives peuvent être envoyées dans le pipeline pour y être traitées une nouvelle fois autorisant ainsi des algorithmes multipasses sans intervention du CPU, ou être lues par le CPU.

Les écritures effectuées à ce stade du pipeline sont purement séquentielles, le Stream Output n'autorise pas les écritures aléatoires dans le buffer de sortie.

1.2 Génération de surfaces implicites sur le GPU

La première application que j'ai dû réaliser a été la génération d'une surface implicite sur le GPU. Pour cela, il faut savoir ce qu'est une surface implicite et connaître des algorithmes simples et efficaces que l'on peut implémenter dans un Geometry Shader.

a) Les surfaces implicites

Une surface implicite est définie à partir d'une fonction de potentiel. C'est une fonction f de $R^3 \rightarrow R$ qui à tout point $P(x_p, y_p, z_p)$ de l'espace R^3 associe une valeur de potentiel C_p. Une surface implicite est alors définie par l'ensemble des points de R^3 pour lesquels la fonction f associe la même valeur de potentiel C_0.

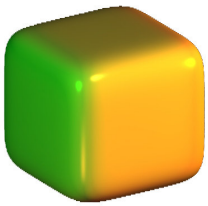
La définition de la surface est implicite car on ne peut pas directement calculer les points de la surface.

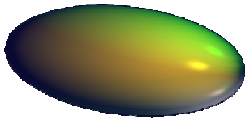
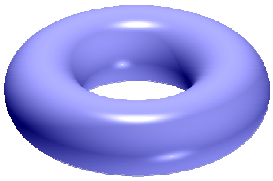
Par contre la fonction de potentiel définit complètement le volume :

- *$S = \{ P \in R^3 / f(P) = C_0 \}$*
- *Si $f(P) > C_0$ le point Pest à l'extérieur du volume*
- *Si $f(P) < C_0$ le point Pest à l'intérieur du volume*
- *$V = \{ P \in R^3 / f(P) \leq C_0 \}$ défini un solide*

En général, $C_0 = 0$.

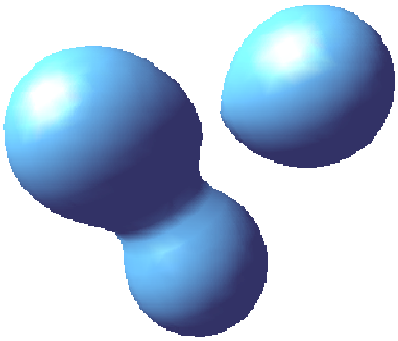
Quelques exemples de surfaces implicites (algébriques) :

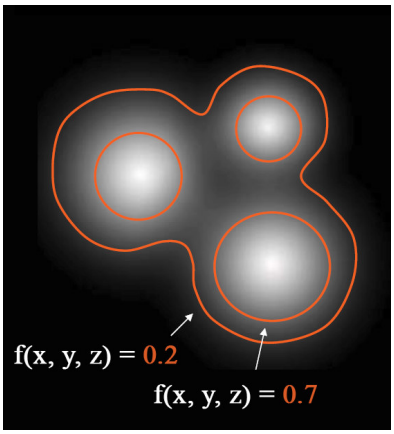
Nom	Equation	Image
Cube	$x^6+y^6+z^6=1$	

Coeur	$(x^2+9/4*y^2+z^2-1)^3 - x^2z^3-9/80*y^2z^3=0$	
Dattel	$3*x^2+3*y^2+z^2=1$	
Torus	$(x^2+y^2+z^2+R^2-r^2)^2 = R^2 (x^2+y^2)$	

Il existe plusieurs familles de surfaces implicites. Je me suis tourné vers le rendu des metaballs car la formule permet d'obtenir des formes très variées avec une seule équation. La formule du gradient sert à calculer la normale au point concerné.

Rendu de Metaballs :





Equation implicite :

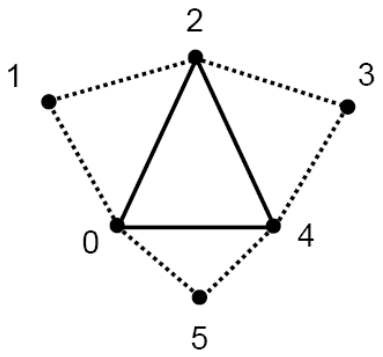
$$\sum_{i=1}^{N} \frac{r_i^2}{\left\|\mathbf{x}-\mathbf{p}_i\right\|^2} = 1$$

Equation du gradient :

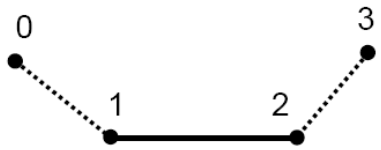
$$\mathbf{grad}(f) = -\sum_{i=1}^{N} \frac{2 \cdot r_i^2}{\left\|\mathbf{x}-\mathbf{p}_i\right\|^4} \cdot (\mathbf{x}-\mathbf{p}_i)$$

b) Méthode du Marching Thetrahedras par nVidia

Nvidia a mis au point un rendu de metaballs dans son 10ème SDK utilisant DirectX 10 et HLSL (*2006-GDC-DX10-Practical-Metaballs*). La méthode qu'ils utilisent est le Marching Tetrahedras car le Geometry Shader ne peut prendre en entrée que six sommets au maximum (triangle avec adjacence), ils n'ont donc pas tenté d'implémenter l'algorithme du Marching Cube qui nécessite huit sommets par cube. Pour cette méthode nous avons juste besoin de quatre sommets par tétraèdre ils ont donc utilisé le format ligne avec adjacence qui correspond parfaitement. Il existe plusieurs agencements de tétraèdres (5 ou 6) pour créer un cube.



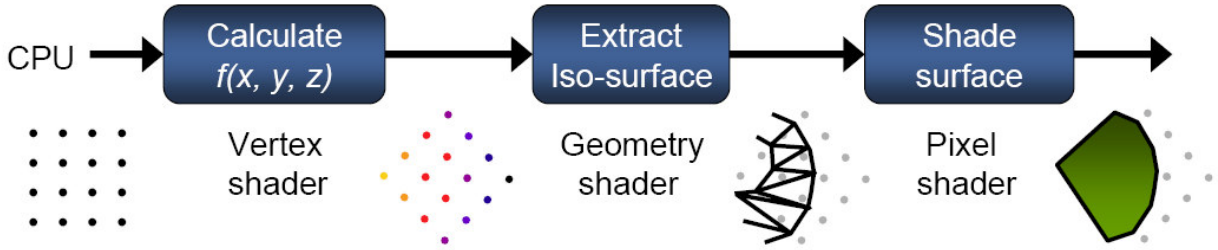
Triangles with adjacency



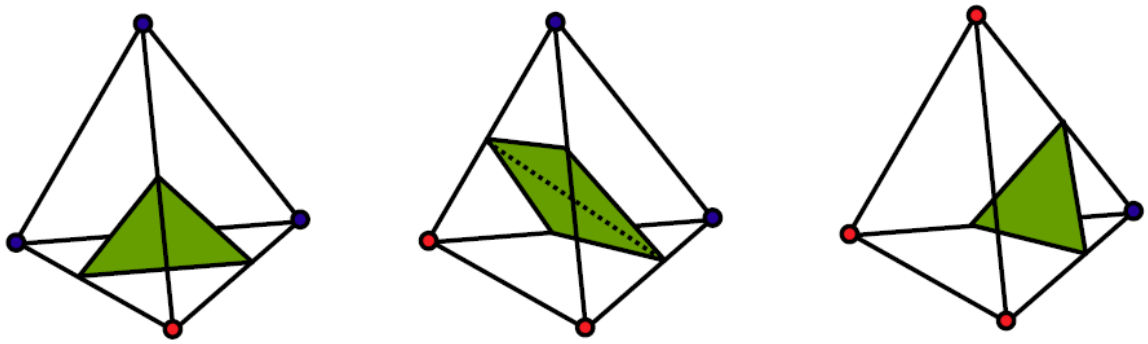
Lines with adjacency

Afin de réaliser cet effet, nous devons donc découper la scène en une grille de sommets qui vont représenter un cube. Un index buffer va permettre de reconstituer chacun des tétraèdres pour mettre en pratique l'algorithme.

Dans un premier temps, on utilise le Vertex Shader pour transformer chacun des sommets et on calcul leur potentiel avec la fonction f(x, y, z). Puis le Geometry Shader va générer les triangles par tétraèdre afin de rendre la surface.



La solution du Marching Tetrahedras est simple et sans ambiguïté et au maximum on a 2 triangles en sortie. Il y a 3 cas possibles.



c) Explication succincte des Shaders utilisés par nVidia

L'exemple de nVidia s'effectue en une seule passe.

- *Structures d'entrée/sortie du VS et GS*

```
// Grid vertex
struct SampleData
{
      float4 Pos        : SV_POSITION;    // Sample position
      float3 N          : NORMAL;         // Scalar field gradient
      float Field       : TEXCOORD0;      // Scalar field value
      uint IsInside     : TEXCOORD1;      // "Inside"flag
};

// Surface vertex
struct SurfaceVertex
{
      float4 Pos  : SV_POSITION;    // Surface vertex position
      float3 N    : NORMAL;         // Surface normal
};
```

- *Vertex Shader*

Dans le vertex program, on effectue la transformation du sommet, on lui donne sa valeur de potentiel et grâce au gradient on peut calculer sa normale.

```
// Metaball function
// Returns metaballfunction value in .w
// and its gradient in .xyz
float4 Metaball(float3 Pos, float3 Center, float RadiusSq)
{
      float4 o;
      float3 Dist = Pos -Center;
      float InvDistSq = 1 / dot(Dist, Dist);
      o.xyz = -2 * RadiusSq* InvDistSq* InvDistSq* Dist;
      o.w = RadiusSq* InvDistSq;
      return o;
}


#define MAX_METABALLS 32

SampleData VS_SampleField(    float3 Pos : POSITION,
                              uniform float4x4 WorldViewProj,
                              uniform float3x3 WorldViewProjIT,
                              uniform uint NumMetaballs,
                              uniform float4 Metaballs[MAX_METABALLS])
{
      SampleData o;
      float4 Field = 0;
      for(uint i = 0; i<NumMetaballs; i++)
            Field += Metaball(Pos, Metaballs[i].xyz, Metaballs[i].w);
      o.Pos = mul(float4(Pos, 1), WorldViewProj);
      o.N = mul(Field.xyz, WorldViewProjIT);
      o.Field = Field.w;
      o.IsInside = Field.w> 1 ? 1 : 0;
      return o;
}
```

- *Geometry Shader*

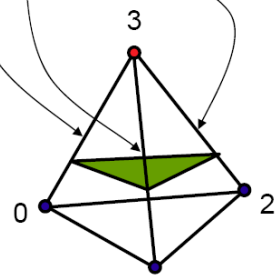
Le geometry program va générer les triangles par tetraèdre en fonction d'une table « EdgeTable ».

```
// Estimate where isosurfaceintersects grid edge
SurfaceVertex CalcIntersection(SampleData v0, SampleData v1)
{
      SurfaceVertex o;
      float t = (1.0 -v0.Field) / (v1.Field -v0.Field);
```

```
	o.Pos = lerp(v0.Pos, v1.Pos, t);
	o.N = lerp(v0.N, v1.N, t);
	return o;
}


[MaxVertexCount(4)]
void GS_TesselateTetrahedra(  lineadj SampleDataIn[4],
                              inout TriangleStream<SurfaceVertex> Stream)
{
	// construct index for this tetrahedron
	uint index = (In[0].IsInside << 3) | (In[1].IsInside << 2) |
	             (In[2].IsInside << 1) | In[3].IsInside;

	const struct{ uint4 e0; uint4 e1; } EdgeTable[] = {
	{ 0, 0, 0, 0, 0, 0, 0, 1 }, // all vertices out
	{ 3, 0, 3, 1, 3, 2, 0, 0 }, // 0001
	{ 2, 1, 2, 0, 2, 3, 0, 0 }, // 0010
	{ 2, 0, 3, 0, 2, 1, 3, 1 }, // 0011 -2 triangles
	{ 1, 2, 1, 3, 1, 0, 0, 0 }, // 0100
	{ 1, 0, 1, 2, 3, 0, 3, 2 }, // 0101 -2 triangles
	{ 1, 0, 2, 0, 1, 3, 2, 3 }, // 0110 -2 triangles
	{ 3, 0, 1, 0, 2, 0, 0, 0 }, // 0111
	{ 0, 2, 0, 1, 0, 3, 0, 0 }, // 1000
	{ 0, 1, 3, 1, 0, 2, 3, 2 }, // 1001 -2 triangles
	{ 0, 1, 0, 3, 2, 1, 2, 3 }, // 1010 -2 triangles
	{ 3, 1, 2, 1, 0, 1, 0, 0 }, // 1011
	{ 0, 2, 1, 2, 0, 3, 1, 3 }, // 1100 -2 triangles
	{ 1, 2, 3, 2, 0, 2, 0, 0 }, // 1101
	{ 0, 3, 2, 3, 1, 3, 0, 0 }}; // 1110


	// don't bother if all vertices out or all vertices in
	if (index > 0 && index < 15)
	{
		uint4 e0 = EdgeTable[index].e0;
		uint4 e1 = EdgeTable[index].e1;

		// Emit a triangle
		Stream.Append(CalcIntersection(In[e0.x], In[e0.y]));
		Stream.Append(CalcIntersection(In[e0.z], In[e0.w]));
		Stream.Append(CalcIntersection(In[e1.x], In[e1.y]));

		// Emit additional triangle, if necessary
		if (e1.z != 0)
			Stream.Append(CalcIntersection(In[e1.z], In[e1.w]));
	}
}
```

Exemple de génération de triangles :

```
const struct { uint4 e0; uint4 e1; } EdgeTable[] = {
        // …
        { 3, 0, 3, 1, 3, 2, 0, 0 },        // index = 1
        // …
};
```

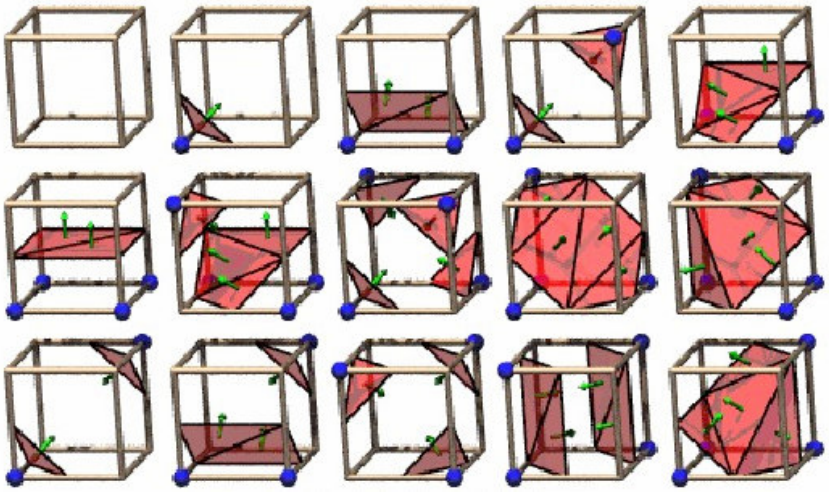


Index = 0001,
i.e. vertex 3 is “inside”

d) Méthode du Marching Cubes

C’est la méthode la plus connue pour représenter des surfaces implicites en synthèse d’images. L’algorithme a été inventé par Bill LORENSEN et Harvey CLINE. Il s’agit d’une méthode surfacique permettant d’extraire une surface équipotentielle (isosurface) d’un maillage structuré et uniforme 3D. Le principe est de calculer, les différentes configurations que peut prendre l’isosurface dans un élément de volume, selon la répartition des intersections de l’isosurface sur les arêtes de cet élément de volume. Dans notre cas le volume est un cube, la scène est donc découpée en une grille de voxels et nous allons générer la surface à partir de ceux-ci.

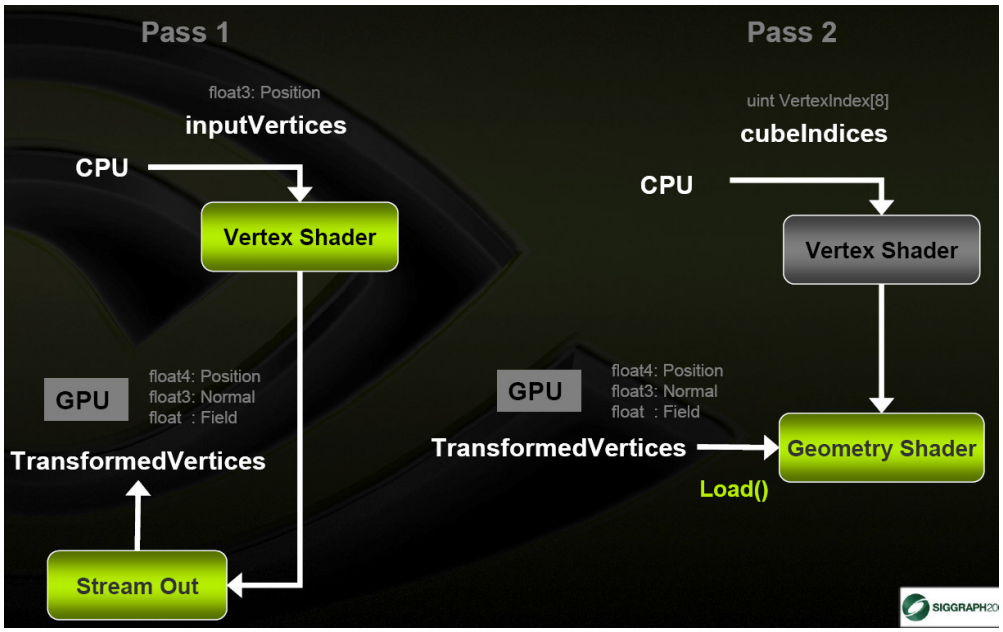
Pour chaque cube, on va déterminer pour chaque sommet s’il est à l’intérieur ou à l’extérieur de la surface. Cela représente 256 (2^8) possibilités, mais elles peuvent se réduire à 15 en tenant compte des symétries.



Afin de pouvoir avoir accès aux huit sommets qui composent le cube, nous n'allons pas nous servir de l'index buffer mais d'une structure de vertex ayant huit entiers et passer le tableau de sommets au shader par une ressource.

J'ai donc implémenté cette méthode que nVidia n'a pas testée. Il n'avait pas pensé à utiliser un buffer autre que le tableau de sommets (vertex buffer) pour envoyer les sommets de la mesh au shader dans leur première tentative de génération de surfaces implicites. Grâce à un tbuffer nous pouvons avoir accès à n'importe quel sommet de la primitive (*2006-SIGGRAPH-DX10-Effects*).

Dans un premier temps, on utilise le Vertex Shader pour transformer chacun des sommets et on calcul leur potentiel avec la fonction f(x, y, z), on fera par la suite une sortie sur le Stream Output. On effectuera ensuite une seconde passe en envoyant les indices des sommets qui composent chaque cube dans le vertex buffer. Puis le Geometry Shader va générer les triangles par cube afin de rendre la surface.



e) Explication succincte de mes Shaders

1ère Passe :

- *Structures d'entrée/sortie du VS et GS*

```
struct VS_INPUT
{
    float3 Position     : POSITION; // vertex position
```

```
};

struct VS_OUTPUT
{
    float4 Position     : SV_POSITION;    // vertex position transform
    float3 Normal       : NORMAL;         // normal
    float Field         : TEXCOORD;       // isolevel
};
```

- *Vertex Shader et Geometry Shader*

En entrée, nous n'avons besoin que de la position du sommet dans l'espace. Grâce à celle-ci nous pouvons calculer sa nouvelle position après transformation, sa normale et enfin son potentiel. Enfin, le geometry shader envoie ce nouveau buffer dans le stream output.

```
float4 Metaball(float3 Pos, float3 Center, float RadiusSq)
{
      float4 o;

      float3 d = Pos - Center;
      float DistSq = dot(d, d);
      float InvDistSq = 1 / (DistSq + 0.001);

      o.xyz = -2 * RadiusSq * InvDistSq * InvDistSq * d;
      o.w = RadiusSq * InvDistSq;

      return o;
}

VS_OUTPUT VSmain( VS_INPUT input )
{
    VS_OUTPUT Output = (VS_OUTPUT)0;

      Output.Position = mul(float4(input.Position, 1.0), mWorldViewProj);

      float4 field = 0.0;
      for (int i=0;i<NumMetaballs;i++)
            field += Metaball(input.Position.xyz, Metaballs[i].xyz,
                        Metaballs[i].w);

      Output.Normal = -normalize(mul(field.xyz, mViewInv));
      Output.Field = field.w;

    return Output;
}

[maxvertexcount(1)]
void GSmain( point VS_OUTPUT input[1], inout PointStream<VS_OUTPUT> Stream )
{
      Stream.Append( input[0] );
}
```

2ème Passe :

- *Structures d'entrée/sortie du VS et GS et Tables*

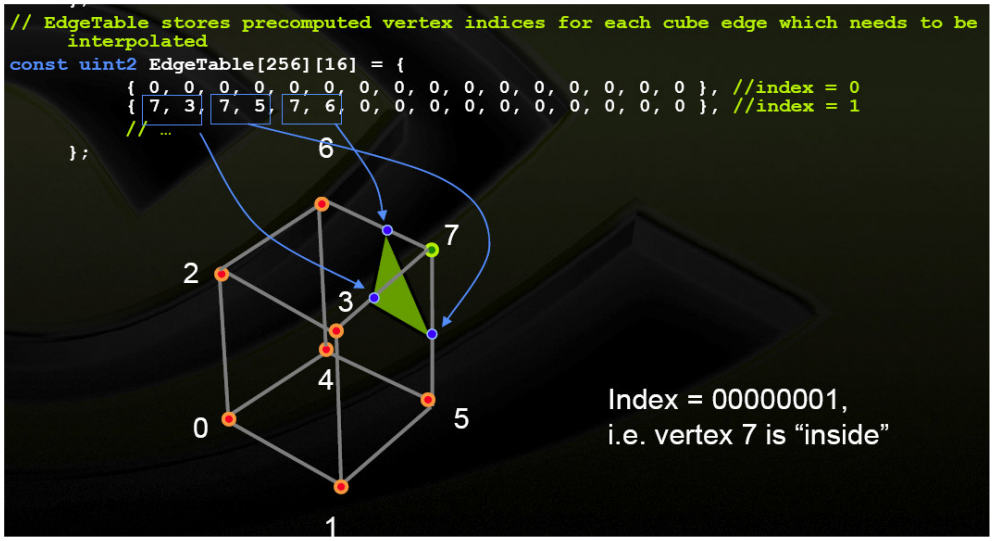
```
struct VS_INPUT
{
      float4 Index1           : POSITION; // index 1
      float4 Index2           : COLOR;    // index 2
};

struct VS_OUPUT
{
      float VertexIndex[8]    : INDEX;    // cube index
};

struct GS_OUTPUT
{
      float4 Position         : SV_POSITION;    // vertex position
      float4 Color            : COLOR0;         // colour
};

// 256 cas pour la génération des triangles
static const int edgeTable[256]=
      {0x0  , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
      0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
      0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
      0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
      … };

// pour la génération des triangles
static const int triTable[256][16] =
      {{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
      {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
      {0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
      {1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
      … };
```

- *Geometry Shader*

```
SurfaceVertex CalcIntersection(float4 v0, float4 nf0, float4 v1, float4 nf1)
{
      SurfaceVertex p;

      float t = (isolevel - nf0.w) / (nf1.w - nf0.w);
      p.Position = lerp(v0, v1, t);
      p.Normal = lerp(nf0.xyz, nf1.xyz, t);

      return p;
}
```

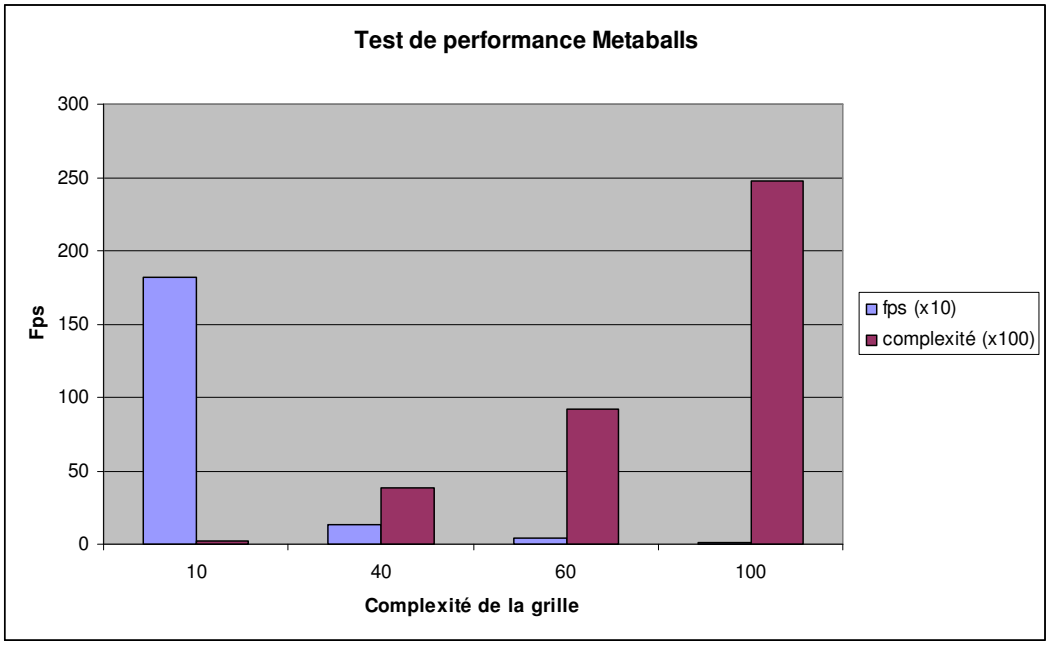
```
[maxvertexcount(15)]
void GSmain( point VS_OUPUT input[1], inout TriangleStream<GS_OUTPUT> Stream )
{
      GS_OUTPUT output = (GS_OUTPUT)0;

      uint cubeindex = 0;
      for(int i=7;i>-1;i--)
            cubeindex |= (uint(vb.Load( input[0].VertexIndex[i]*2+1).w <
                        isolevel) << i);

      SurfaceVertex vertlist[12];

      if(edgeTable[cubeindex] != 0x0)
      {
            if (edgeTable[cubeindex] & 1)
              vertlist[0] = CalcIntersection(
vb.Load( input[0].VertexIndex[0]*2 ), vb.Load( input[0].VertexIndex[0]*2+1 ),
vb.Load( input[0].VertexIndex[1]*2 ), vb.Load( input[0].VertexIndex[1]*2+1));

            if (edgeTable[cubeindex] & 2)
                  …

      for(int i=0;triTable[cubeindex][i]!=-1;i+=3)
      {
            output.Color = float4(vertlist[triTable[cubeindex][i+2]].Normal, 1.0);
            output.Position = vertlist[triTable[cubeindex][i+2]].Position;
            Stream.Append( output );

            output.Color = float4(vertlist[triTable[cubeindex][i+1]].Normal, 1.0);
            output.Position = vertlist[triTable[cubeindex][i+1]].Position;
            Stream.Append( output );

            output.Color = float4(vertlist[triTable[cubeindex][i]].Normal, 1.0);
            output.Position = vertlist[triTable[cubeindex][i]].Position;
            Stream.Append( output );
            Stream.RestartStrip();
      }
}
}
```

Exemple de génération de triangles :



f) Tests de performance

Scène : 4 metaballs tournant sur 3 axes (x, y, z)

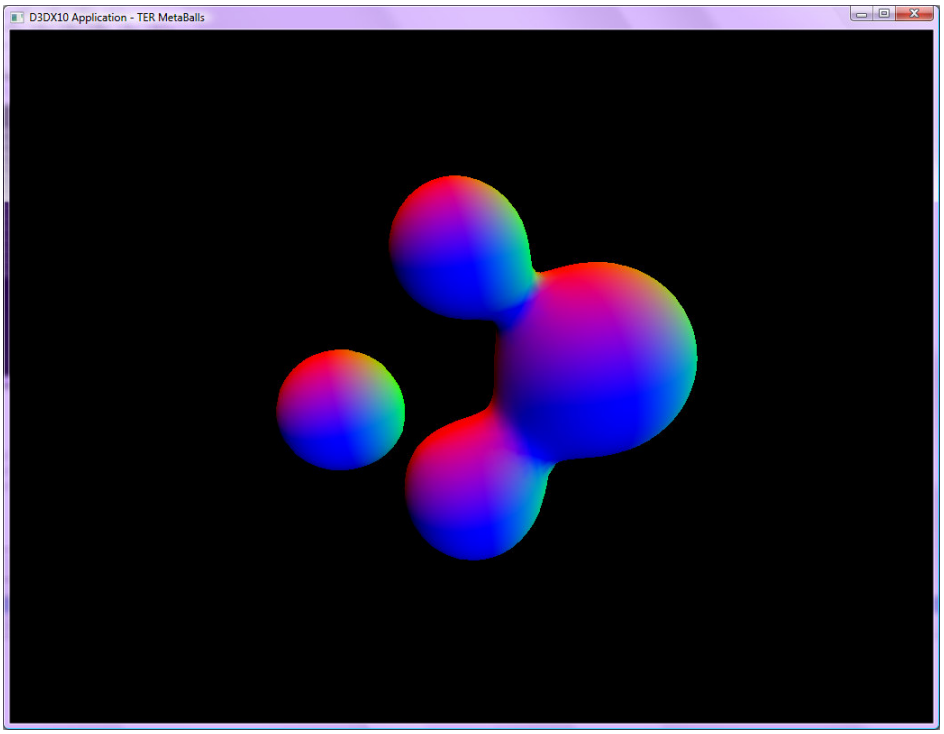


Ce test de performance met en corrélation la rapidité de rendu de la scène 3D comportant quatre metaballs, avec la complexité de la grille et le nombre de triangles émis par le geometry shader.

On remarque clairement qu’avec l’augmentation de la complexité, le nombre de primitives générées par le geometry shader augmente. Le nombre d’images par

seconde baisse en conséquence.

Rendu de l'application



Annexes

Précisions et correction de la documentation

g) Les cbuffer (Boris Brugevin)

Il existe une version de cbuffer « Immediate Buffer » qui peut être initialisé dans le shader en dur et non modifiable. J'ai constaté ceci en me plongeant dans le code du shader compilé. Pour créer ce genre de buffer il faut le déclarer en « static const ».

h) Création d'un geometry shader avec stream output (Boris Brugevin)

La fonction pour créer un geometry shader avec stream output est : « CreateGeometryShaderWithStreamOutput ». La documentation indique pour l'avant dernier paramètre : la taille de la structure de la déclaration de sortie (D3D10_SO_DECLARATION_ENTRY), mais ceci est une erreur, il faut indiquer la taille de la structure de sortie d'un élément après le stream output.

i) Désactivation du Rasterizer (Boris Brugevin)

Le passage du pixel shader à « NULL » (par exemple lors d'une sortie sur le stream output) ne désactive pas le Rasterizer. Ceci implique que l'écriture dans le Z-Buffer se fait tout de même si on ne désactive pas le depth buffer.

j) Multi stream output (Boris Brugevin)

Précision de la documentation, le multi stream output est limité à 4 sorties comprenant de 1 à 4 variables de 32 bits. Sinon il ne faut utiliser qu'un seul stream de sortie.

k) Stream Output (Boris Brugevin)

Le stream output ne peut pas sortir des buffers de 3 variables. Il ne peut que produire des buffers de 1, 2 ou 4 variables de 32 bits. Ceci est sûrement dû à un problème d'alignement.

I) Création d'une vue sur un buffer (Boris Brugevin)

Il est normalement possible de mettre le 2ème paramètre de la fonction « CreateShaderResourceView » à NULL pour donner l'accessibilité complète à la ressource au shader. Mais dans le cas d'un « Buffer » ceci n'est pas possible, il faut le spécifier explicitement.

Bibliographie

[1] Yury Uralsky. *"Practical Metaballsand Implicit Surfaces".* Game Developpers Conference. March 2006.

[2] Simon Green. *"Next Generation Games with Direct3D 10".* Game Developpers Conference. 2006.

[3] Sarah Tariq. *"DirectX10 Effects".* Siggraph. 2006.

[4] Jos Stam. *"Evaluation of Loop Subdivision Surfaces".* 1998.

[5] Arul Asirvatham and Hugues Hoppe. *"Terrain Rendering Using GPU-Based Geometry Clipmaps".* GPU Gems 2. 2005.

[6] Kevin Myers. *"Using D3D10 NowUsing Now".* Game Developpers Conference. 2007.

[7] Nick Thibieroz. *"DirectX 10 for Techies".* Game Developpers Conference. July 2006.

[8] Guennadi Rigue. *"DirectX10: porting,performance and "gotchas"".* Game Developpers Conference. March 2007.