Tutorial Pratico coi Geometry Shaders

Approcciare la prima volta con i geometry shaders non è facile, soprattutto perché non se ne capisce subito l'utilità. Vediamo di capirci qualcosa in piu' con un completo esempio pratico.
Il progetto prevede, a partire da un singolo triangolo, di creare un intero cubo tramite Geometry Shader.
I requisiti sono pochi: fondamenti del device, shaders, matrici e tanta, TANTA pazienza. Partiamo subito con il codice allora.

```
#include <windows.h>
#include <d3d10.h>
#include <d3dx10.h>

#define BEGINSCENE      //Servono solo per ordinare un pò le idee.
#define ENDSCENE

#pragma comment(lib,"d3d10.lib")
#pragma comment(lib,"d3dx10d.lib")
```

Sono i soliti include che conosciamo benissimo e i pragma per effettuare il linking alle librerie. (d3dx10d.lib è la libreria di debug, per la retail basta togliere la d).

I 2 define sono solo a tenere ordinato il codice. Ho l'abitudine, per via di DirectX9, di racchiudere la zona di rendering tra queste 2 parti. Capirete in seguito.

```
struct VertexType
{
	D3DXVECTOR3 position;
	D3DXVECTOR2 text;
};
```

Qui definiamo il nostro formato del vertice: contiene solo posizione e coordinate di texture.

```
void InitWindow(HINSTANCE hInstance);
void InitD3D10();
void SetMatrix();
void Chiudi();
void Render();
```

Queste sono le nostre funzioni: InitWindow crea la finestra di Windows, InitD3D10 inizializza il device Direct3D10…insomma si spiegano da sole.

```
const char g_szClassName[] = "myWindowClass";
ID3D10Device                *device; //Device
IDXGISwapChain              *swapChain; //Swapchain
ID3D10RenderTargetView  *RenderTarget; //RenderTarget
ID3D10DepthStencilView  *depthstencil; //Depth Stencil
ID3D10ShaderResourceView    *texture;   //Texture
ID3D10Effect                *shader;    //Shader
ID3D10Buffer                *vertexBuffer; //Vertexbuffer
HWND                        hwnd;       //Finestra
VertexType                  vertices[3];//Vertici
```

I nostri oggetti globali: 1 device, 1 swapchain, 1 rendertarget, una superfice di depth stencil, una texture, uno shader, un vertexbuffer, 3 vertici e una finestra di windows. Niente di spieciale fin qui.
Per chi non lo sapesse ancora, il DepthStencil è una superfice che tiene traccia delle informazioni sulla

profondità. Senza questa non sarebbe possibile avere il 3D. Se non lo sapevate vi consiglio inoltre di non continuare a leggere perché probabilmente non capireste.

Definiamo ora le nostre funzioni InitWindow e InitD3D10

```
void InitWindow(HINSTANCE hInstance)
{
	WNDCLASSEX wc;

	wc.cbSize        = sizeof(WNDCLASSEX);
	wc.style         = 0;
	wc.lpfnWndProc   = WndProc;
	wc.cbClsExtra    = 0;
	wc.cbWndExtra    = 0;
	wc.hInstance     = hInstance;
	wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
	wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
	wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
	wc.lpszMenuName  = NULL;
	wc.lpszClassName = g_szClassName;
	wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

	RegisterClassEx(&wc);

	hwnd = CreateWindowEx(
		WS_EX_CLIENTEDGE,
		g_szClassName,
		"DirectX10",
		WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT, CW_USEDEFAULT, 800, 600,
		NULL, NULL, hInstance, NULL);

	if(hwnd == NULL)
	{
		MessageBox(NULL, "Creazione della Finestra Fallita!", "Errore!",
			MB_ICONEXCLAMATION | MB_OK);
	}

	ShowWindow(hwnd,SW_SHOWDEFAULT);
	UpdateWindow(hwnd);

}
```

Le cose sono sempre le stesse: WNDCLASSEX, registriamo la classe, creiamo la finestra e stiamo a posto.

```
void InitD3D10()
{


	{

		DXGI_SWAP_CHAIN_DESC sd;
		memset(&sd,0,sizeof(sd));

		sd.BufferCount = 1;
```

```
		sd.BufferDesc.Height = 600;
		sd.BufferDesc.Width = 800;
		sd.BufferDesc.RefreshRate.Numerator = 60;
		sd.BufferDesc.RefreshRate.Denominator = 1;
		sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
		sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
		sd.OutputWindow = hwnd;
		sd.Windowed = true;
		sd.SampleDesc.Count = 1;
		sd.SampleDesc.Quality = 0;

		D3D10CreateDeviceAndSwapChain(NULL,D3D10_DRIVER_TYPE_HARDWARE,NULL,
			D3D10_CREATE_DEVICE_DEBUG,D3D10_SDK_VERSION,
			&sd,&swapChain,&device);

	}

	ID3D10Texture2D *backbuffer;


	swapChain->GetBuffer(0,__uuidof(ID3D10Texture2D),(LPVOID*)&backbuffer);
	device->CreateRenderTargetView(backbuffer,NULL,&RenderTarget);
	backbuffer->Release();


	{
		D3D10_VIEWPORT vp;
		vp.Width = 800;
		vp.Height = 600;
		vp.MinDepth = 0.0f;
		vp.MaxDepth = 1.0f;
		vp.TopLeftX = 0;
		vp.TopLeftY = 0;

		device->RSSetViewports(1,&vp);
	}

	ID3D10Texture2D *stenciltext;
	{
		D3D10_TEXTURE2D_DESC d;
		d.CPUAccessFlags = 0;
		d.Usage = D3D10_USAGE_DEFAULT;
		d.Width = 800;
		d.Height = 600;
		d.MipLevels = 1;
		d.Format = DXGI_FORMAT_D32_FLOAT;
		d.ArraySize = 1;
		d.SampleDesc.Count = 1;
		d.SampleDesc.Quality = 0;
		d.BindFlags = D3D10_BIND_DEPTH_STENCIL;
		d.MiscFlags = 0;

		device->CreateTexture2D(&d,0,&stenciltext);
	}

	{
		D3D10_DEPTH_STENCIL_VIEW_DESC p;
		p.Format = DXGI_FORMAT_D32_FLOAT;
```

```
		p.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;
		p.Texture2DArray.MipSlice = 0;

		device->CreateDepthStencilView(stenciltext,&p,&depthstencil);

		device->OMSetRenderTargets(1,&RenderTarget,depthstencil);
		stenciltext->Release();
	}


}
```

Anche qui le solite cose: device, render target, depth stencil, settiamo il tutto ed ecco fatto.
Veniamo ora al WinMain

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                           LPSTR lpCmdLine, int nCmdShow)
{
	InitWindow(hInstance);
	InitD3D10();

	ID3D10Blob *Errori;
	D3DX10CreateEffectFromFile("Gs.fx",NULL,NULL,"fx_4_0",D3D10_SHADER_DEBUG|D3D1
0_SHADER_OPTIMIZATION_LEVEL3,0,device,NULL,NULL,&shader,&Errori,NULL);

	if (Errori)
	{
		MessageBox(NULL,(char *)Errori->GetBufferPointer(),"Errore",MB_OK);
		Errori->Release();
		CloseWindow((HWND)GetModuleHandle(NULL));
	}

	shader->Optimize();

	SetMatrix();
	D3DX10CreateShaderResourceViewFromFile(device,"text.jpg",NULL,NULL,&texture,N
ULL);


	//Set up the vertex buffer

	//Vertex setting
	vertices[0].position = D3DXVECTOR3(0.5f,-0.5f,-0.5f);
	vertices[0].text = D3DXVECTOR2(1,1);
	vertices[1].position = D3DXVECTOR3(-0.5f,-0.5f,0.5f);
	vertices[1].text = D3DXVECTOR2(0,1);
	vertices[2].position = D3DXVECTOR3(0.5,0.5f,0);
	vertices[2].text = D3DXVECTOR2(1,0);

	{
		D3D10_BUFFER_DESC bufferDesc;

		bufferDesc.Usage = D3D10_USAGE_DEFAULT;
		bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
		bufferDesc.CPUAccessFlags = 0;
		bufferDesc.MiscFlags = 0;
		bufferDesc.ByteWidth = 3 * sizeof(VertexType);
```

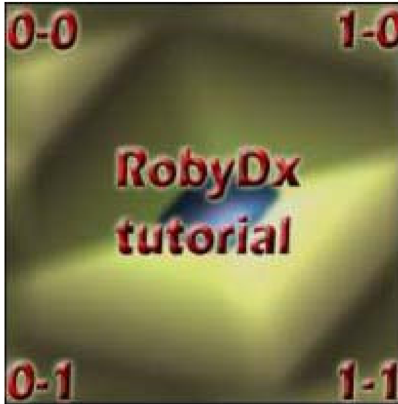
```
		D3D10_SUBRESOURCE_DATA Init;

		Init.pSysMem = vertices;

		device->CreateBuffer(&bufferDesc,&Init,&vertexBuffer);
	}

	//Input layout for shader.

	ID3D10InputLayout *VertexDesc;
	{
		D3D10_INPUT_ELEMENT_DESC VertexLayout[] =
		{

	{"POSITION",0,DXGI_FORMAT_R32G32B32_FLOAT,0,0,D3D10_INPUT_PER_VERTEX_DATA,0},

	{"TEXCOORD",0,DXGI_FORMAT_R32G32_FLOAT,0,12,D3D10_INPUT_PER_VERTEX_DATA,0},
		};

		D3D10_PASS_DESC dc;

		shader->GetTechniqueByIndex(0)->GetPassByIndex(0)->GetDesc(&dc);
		device-
>CreateInputLayout(VertexLayout,2,dc.pIAInputSignature,dc.IAInputSignatureSize,&Ver
texDesc);
		device->IASetInputLayout(VertexDesc);


	}

	MSG Msg;

	ZeroMemory(&Msg,sizeof(MSG));

	while( WM_QUIT != Msg.message )
	{
		if( PeekMessage( &Msg, NULL, 0, 0, PM_REMOVE ) )
		{
			TranslateMessage( &Msg );
			DispatchMessage( &Msg );
		}
		else
		{
			Render();
		}
	}


	return Msg.wParam;
}
```

Qui dobbiamo dire qualcosa in piu'.
Dopo aver richiamato le 2 funzioni prima definite, creiamo uno shader da un file .fx (gs.gx, per l'appunto) utilizzando i flag di DEBUG e WARNING3. In pratica il compilatore HLSL ci segnalerà il massimo numero di warning possibili. Inoltre controlliamo che se il Blob di errore è diverso da 0 ci sono stati errori di compilazione: dunque li controlliamo e li mandiamo a video con un Msgbox
Dopo di chè c'è la chiamata a Set Matrix, che vediamo qui sotto

```
void SetMatrix()
{
	D3DXMATRIXA16 World;
	D3DXMATRIXA16 View;
	D3DXMATRIXA16 Proj;

	D3DXVECTOR3 Eye( 0.0f, 0.0f, -2.0f );
	D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
	D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );

	D3DXMatrixIdentity(&World);
	D3DXMatrixPerspectiveFovLH(&Proj,(float)D3DX_PI * 0.5f,4/3,0.1f,100.0f);
	D3DXMatrixLookAtLH(&View,&Eye,&At,&Up);

	shader->GetVariableByIndex(0)->AsMatrix()->SetMatrix((float*)&View);
	shader->GetVariableByIndex(1)->AsMatrix()->SetMatrix((float*)&World);
	shader->GetVariableByIndex(2)->AsMatrix()->SetMatrix((float*)&Proj);

}
```

Niente di particolare: le 3 matrici fondamentali (la World è messa a Identity (è una matrice che non fa niente), e settiamo tutto nello shader (il codice di quest'ultimo lo vedremo dopo).

Tornando al WinMain, c'è il caricamento della texture da file e la definizione dei vertici.
Il D3DXVECTOR3 è la posizione, D3DXVECTOR2 le coordinate di texture. Supponendo di usare quest'immagine



Mappiamo i vertici adeguatamente.
Se non riuscite bene a focalizzare il triangolo, prendete carta e penna, disegnate un piano cartesiano e fate i 3 punti nello spazio, poi uniteli.

Creati i vertici, facciamo un vertexbuffer, il vertex declaration per lo shader (che contiene coordinate di Texture e Posizione) e poi si inizia il ciclo dei messaggi.

Vediamo ora la funzione Render

```
{
	float color[4] = { 0, 0, 1, 0};
	device->ClearRenderTargetView(RenderTarget,color);
```

```
	device-
>ClearDepthStencilView(depthstencil,D3D10_CLEAR_STENCIL|D3D10_CLEAR_DEPTH,1.0f,0);

	BEGINSCENE
	shader->GetVariableByIndex(3)->AsShaderResource()->SetResource(texture);
	shader->GetTechniqueByIndex(0)->GetPassByIndex(0)->Apply(0);
	device->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);

	UINT stride = sizeof(VertexType);
	UINT offset = 0;
	device->IASetVertexBuffers(0,1,&vertexBuffer,&stride,&offset);

	device->Draw(3,0);
	ENDSCENE

		swapChain->Present(0,0);
}
```

C'è il clear del render target e del depth stencil, poi si setta la texture, applica per confermare, e si disegna.
Notate che il BEGINSCENE ed ENDSCENE sono define che non servono a niente. Hanno solo lo scopo di tenere un ordine mentale, ciò è dov'è che si renderizza per davvero. Ecco in questo caso stanno messi male, perché il vero rendering avviene nel Draw.

Ricordo do per scontato che voi sappiate fare queste cose. Vediamo ora lo shader.

```
float4x4	ViewMatrix;
float4x4	WorldMatrix;
float4x4	ProjMatrix;

texture2D	Text;

RasterizerState Rast
{
	CullMode = None;
};

SamplerState		SamplText
{
	Filter = MIN_MAG_POINT_MIP_LINEAR;
	AddressU = Wrap;
	AddressV = Wrap;
};

struct VS_INPUT
{
	float4		Pos	:	POSITION;
	float2		Tex	:	TEXCOORD;
};

struct GS_INPUT
{
	float4		Pos	:	POSITION;
	float2		Tex	:	TEXCOORD;
};
```

```
struct PS_INPUT
{
	float4	Pos	:	SV_POSITION;
	float2	Tex	:	TEXCOORD;
};

GS_INPUT vs_main(VS_INPUT In)
{
	GS_INPUT Out = (GS_INPUT)0;
	Out.Pos = mul(mul(mul(In.Pos,WorldMatrix),ViewMatrix),ProjMatrix);
	Out.Tex = In.Tex;

	return Out;
}

float4 ps_main(PS_INPUT In)	:	SV_TARGET
{
	return Text.Sample(SamplText,In.Tex);
}

[maxvertexcount(32)]
void gs_main(triangle GS_INPUT In[3], inout TriangleStream<PS_INPUT> OutStream)
{
	PS_INPUT NewVertex[5]; //Vertici


//Vertici necessari
	NewVertex[0].Pos = mul(mul(mul(float4(-
0.5,0.5,0,1),WorldMatrix),ViewMatrix),ProjMatrix);
	NewVertex[0].Tex = float2(0,0);

	NewVertex[1].Pos =
mul(mul(mul(float4(0.5,0.5,0.5,1),WorldMatrix),ViewMatrix),ProjMatrix);
	NewVertex[2].Pos = mul(mul(mul(float4(0.5,-
0.5,0.5,1),WorldMatrix),ViewMatrix),ProjMatrix);
	NewVertex[3].Pos = mul(mul(mul(float4(-
0.5,0.5,0.5,1),WorldMatrix),ViewMatrix),ProjMatrix);
	NewVertex[4].Pos = mul(mul(mul(float4(-0.5,-
0.5,0.5,1),WorldMatrix),ViewMatrix),ProjMatrix);

	 for (int i = 0; i < 3; i++)
	{
		OutStream.Append((PS_INPUT)In[i]);
	}

	OutStream.Append(NewVertex[0]);
	OutStream.RestartStrip();

//Secondo quadrato
	In[2].Tex = float2(0,0);

	NewVertex[1].Tex = float2(1,0);
	In[0].Tex = float2(0,1);
	NewVertex[2].Tex = float2(1,1);

	OutStream.Append((PS_INPUT)In[2]);
	OutStream.Append(NewVertex[1]);
	OutStream.Append((PS_INPUT)In[0]);
```

```
	OutStream.Append(NewVertex[2]);

	OutStream.RestartStrip();

//Terzo quadrato

	In[1].Tex = float2(1,1);
	NewVertex[0].Tex = float2(1,0);
	NewVertex[3].Tex = float2(0,0);
	NewVertex[4].Tex = float2(0,1);

	OutStream.Append((PS_INPUT)In[1]);
	OutStream.Append(NewVertex[0]);
	OutStream.Append(NewVertex[4]);
	OutStream.Append(NewVertex[3]);

	OutStream.RestartStrip();

//Quarto quadrato

	OutStream.Append(NewVertex[2]);
	OutStream.Append(NewVertex[1]);
	OutStream.Append(NewVertex[4]);
	OutStream.Append(NewVertex[3]);

	OutStream.RestartStrip();

	//Tralasciati i quadrati superiore e inferiore
}

technique10 Gs
{
	Pass P0
	{
		SetVertexShader(CompileShader(vs_4_0,vs_main()));
		SetGeometryShader(CompileShader(gs_4_0,gs_main()));
		SetPixelShader(CompileShader(ps_4_0,ps_main()));

		SetRasterizerState(Rast);
	}
}
```

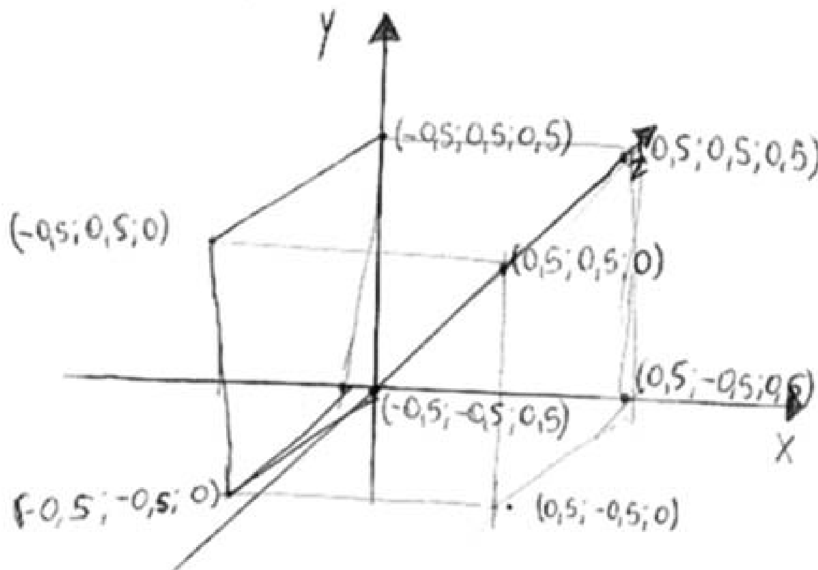
Pixel e Vertex shader lasciamoli stare, sono le solite cose di posizione e texture. Notiamo però la creazione di un RasterState, settato nel Pass, che serve a farci vedere l'oggetto sia davanti che da dientro (un concetto che ancora devo capire anche io, sta di fatto che se non lo mettiamo non va bene)
Nel geometry shader:
il primo parametro tra [ ] indica il numero massimo di vertici che usciranno dal geometry: se sono in piu non li renderizzerà. Utile per porre dei limiti

Nella definizione della funzione abbiamo in input un array di 3 GS_INPUT (restituiti dal vertex shader). La parola triangle significa, appunto, che stiamo avendo un triangolo, ma è possibile scrivere anche point per dire che sono 3 punti non collegati tra loro.
Il secondo parametro è ciò che uscirà dal geometry. inout è obbligatorio. Il <PS_INPUT> indica che cacceremo fuori un array di PS_INPUT che andranno nel Pixel Shader

Avendo 3 vertici, manca ne mancano 5 per formare il cubo: dichiaramo un altro PS_INPUT e lo creiamo con l'ultima coordinata mancante e, seguendo sempre le coordinate texture, mappiamo anche quest'ultimo vertice. A questo punto, tramite l'istruzione Append, inseriamo i 3 vertici piu' l'ultimo. Per indicare che questo triangolo è finito, diamo un RestartStrip().

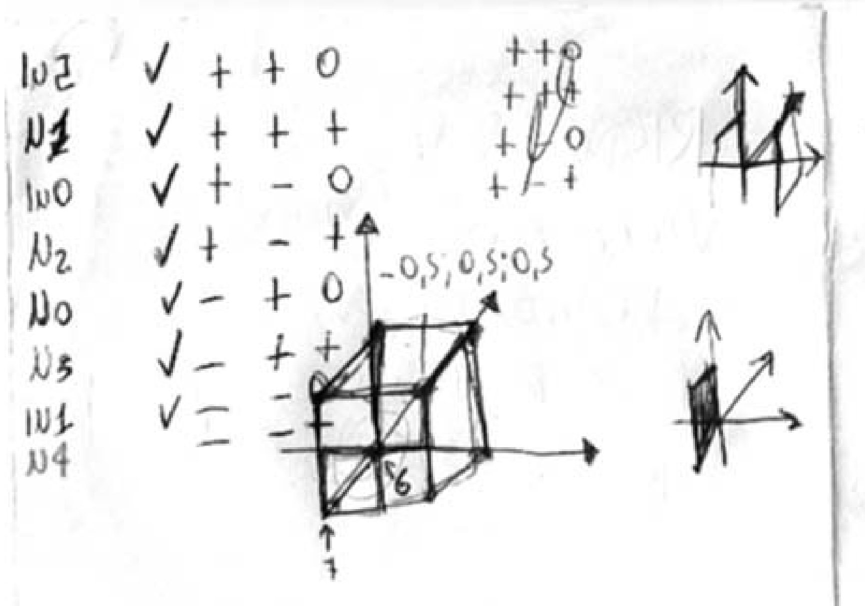
Adesso non ci resta che creare gli altri quadrati per finire il cubo. E ora ALT.
Sono sicuro al 99% che vi starete chiedendo: ma le coordinate di texture e del vertice come le fai a creare?
Bene, quando usavo D3D9 sinceramente copiavo le coordinate dei vertici, perché tanto utilizzavo le mesh. Ora invece, con i Geometry shader, sembra proprio che ci ritroveremo spesso a lavorare con triangoli e a doverne costruire vari manualmente. Cerchiamo quindi di capire come facciamo i quadrati restanti.

Per prima cosa si prende carta e penna e si disegnano i 3 assi cartesiani X,Y,Z.
Prendiamo quindi una qualunque unità di misura che chiameremo **t**.
Essendo il cubo una figura dai lati uguali, tutti i lati, per nostra fortuna, misureranno t. Nel caso dell'esempio che state qui leggendo, t = 0,5f, penso l'abbiate capito.
Vediamo come ho costruito il primo quadrato (il triangolo + nuovo vertice)
Il quadrato è una figura bidimensionale, quindi niente profondità. Ciò implica che tutti i vertici avranno 0 come valore Z. Partendo quindi dal punto O(0,0,0), andiamo a destra di 0,5: abbiamo il primo lato. (0,5;0;0). Saliamo su ed ecco il secondo lato (0,5;0,5;0). Poi a sinistra, e poi giu'. Spostandoci costantemente di 0,5 abbiamo creato il nostro primo quadrato (che nell'esempio corrisponde proprio al triangolo + vertice aggiunto).
4 vertici: siamo già a metà dell'opera: per finire il quadrato non ci resta che duplicare i vertici inserendoci anche la componente Z (che sarà sempre 0,5) per avere altri 4 vertici. Uniamo adeguatamente i lati ed ecco il nostro cubo, una cosa del genere



Notate che anche se c'è un punto (-0.5;-0.5;0.5) che coincide con l'origine degli assi, ciò non è vero: è che salendo su Z casualmente tocca l'origine degli assi. Spero che abbiate capito ciò che significa :D

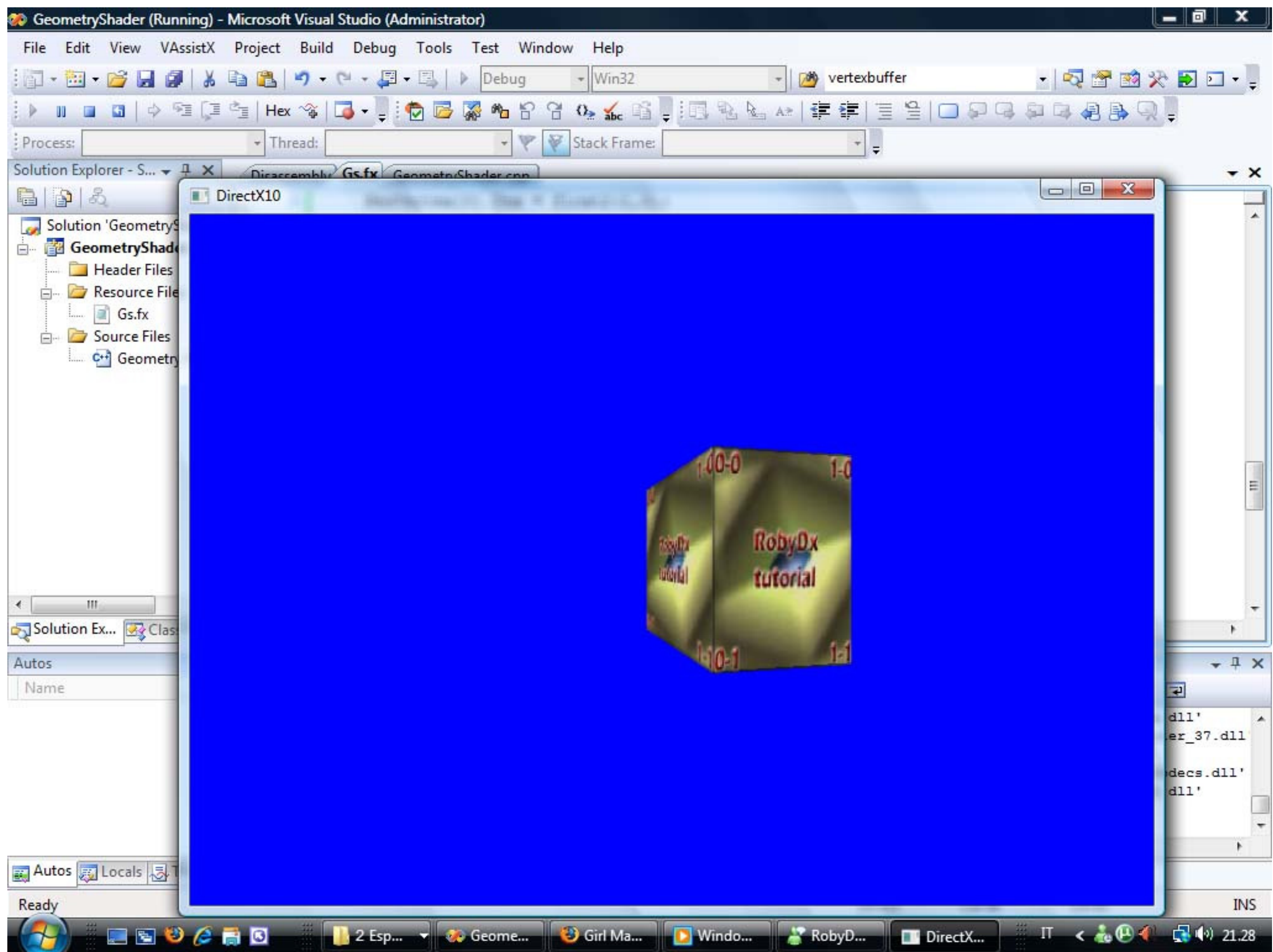
Nel creare il cubo, sempre che non copiate il codice già da me scritto, vi sarà molto utile tenere traccia della struttura che contiene i vertici e delle sue coordinate. Essendo sempre 0.5, io ho preferito tenere traccia dei segni delle coordinate.



Gli altri disegni non prendeteli in considerazioni, erano “prove”.
Per esempio, volendo creare il quadrato laterale, osservo dal disegno che mi serve il vertice con 2 positivi e uno 0, quello con 3 positivi..e così via.
Dal mio schema subito trovo che il primo vertice è nella struttura In[2].
Risparmierete molto tempo così.

Non è facile da capire, ci ho sbattuto la testa alcuni giorni prima di portare a termine tutto, usando piu’ volte approcci sbagliati (creare TUTTI I VERTICI (4 * 8 = 32) invece di usarne solo 8), texture che sbagliavano e chissà quali altri problemi. Resto comunque a disposizione per qualsiasi tipo di chiarimento.

Ohhh se avete fatto tutto bene, premete F5 e godetevi il risultato (notate la finestra di Msn con RobyDx PERENNEMENTE aperta)



Se disattivare il Geometry Shader (adeguatamente, non basta mettere NULL nella funzione apposita ma anche modificare il VertexShader affinchè restituisca un PS_INPUT anziché un GS_INPUT), vedrete che apparirà soltanto un triangolo.

A cosa serve il geometry shader? A regolare dinamicamente, per esempio, il numero di poligoni di un oggetto, per risparmiare memoria. A cosa serve usare 1000 triangoli se l'oggetto è lontanissimo e l'occhio ne percepisce massimo 2-3?

Quest'operazione era possibile anche prima con le ProgressiveMesh (ID3DXProgressiveMesh9) ma l'operazione era gestita da CPU e non GPU.

Vincenzo Chianese