#include "Direct3D.h"

**using** **namespace** std;

**struct** Vertex // variation or different style for the FVF – this allows for textures.

{

Vertex() {}

Vertex(**float** \_x, **float** \_y, **float** \_z, D3DCOLOR c)

{

x = \_x; y = \_y; z = \_z; color = c;

}

Vertex(**float** \_x, **float** \_y, **float** \_z, D3DCOLOR c, **float** \_tu, **float** \_tv)

{

x = \_x; y = \_y; x = \_z; tu = \_tu; tv = \_tv;

}

**static** **const** DWORD FORMAT = D3DFVF\_XYZ | D3DFVF\_DIFFUSE | D3DFVF\_TEX1;

**float** x, y, z, tu, tv; //position

D3DCOLOR color; //color

**static** **const** DWORD FVF;

};

// Credits to ZophusX and Ber'Zophus - 2005-09 -

// \*NEW\*

// This is an Untransformed vertex with a Diffuse colour and Texture.

**struct** Vertex\_UDTx {

// The Untransformed and diffuse are the same and still in this order.

**float** x, y, z;

DWORD colour;

// But we also have two extra floats, generally called tu and tv. They represent the

// orientation of the texture. Hence, we can control where the upper-left and lower-right is.

// A value of 0.0 is furthest left, or up. 1.0 is furthest right, or down. Hence, from 0.0 to

// 1.0 is a complete drawing of the texture. You can even specify 2.0, which will draw the

// texture twice.

**float** tu, tv;

// The format is a lot like Vertex\_UD, only we add TEX1, actually means "1 set of texture

// components."

**static** **const** DWORD FORMAT = D3DFVF\_XYZ | D3DFVF\_DIFFUSE | D3DFVF\_TEX1;

**static** **const** **int** STRIDE\_SIZE = 24; // 5 floats + 1 DWORD = 24 bytes.

};

**const** DWORD Vertex::FVF = D3DFVF\_XYZ | D3DFVF\_DIFFUSE;

IDirect3DVertexBuffer9\* VB; // VertexBuffer interface

IDirect3DIndexBuffer9\* IB; // **Index buffer interface**

IDirect3DTexture9\* texture = NULL; // Texture interface

D3DXMATRIX World; // World matrix (product of rX, rY, rX)

**class** \_winmain : **public** Direct3D // the MAIN runtime class for the application -

{

**public**:

\_winmain(HINSTANCE hInstance); // constructor AND main function

**bool** Init() override; // will require override by direct3D child class

**void** Update(**float** dt) override; // as we will override them from base class

**void** Render(**float** dt) override; // ditto

};

\_winmain::\_winmain(HINSTANCE hInstance) : Direct3D(hInstance) // call constructor of base class

{

// nothing is required to be done here.

}

// translation and rotation variables (including some runtime buffer variables)

**float** rotY;

**float** rotX;

**float** rotZ;

**float** transX;

**float** transY;

**float** transZ;

**float** targX;

**float** targY;

**float** targZ;

**bool** bFirstLaunch = **true**;

**bool** autoWorld = **false**;

**bool** xRot = **false**;

**bool** yRot = **false**;

**bool** zRot = **false**;

**bool** xTrans = **false**;

**bool** yTrans = **false**;

**bool** zTrans = **false**;

**int** debugWindowBuffer = 0;

IDirect3DSurface9 \*background;

D3DXMATRIX Scale, Rx, Ry, Rz, Translation; // our world matrices.

// the current parameter has no use - it may serve as a clock in latter versions.

// This function will UPDATE all currently rendered objects/sprites on the screen.

**void** \_winmain::Update(**float** dt)

{

//ViewSetup(); // could be used in further development to add lighting and a camera setup.

xRot = **false**; // |

yRot = **false**; // |

yTrans = **false**; // By default, we are making no changes unless we decide otherwise (by hotkey downpress)

xTrans = **false**; // |

zTrans = **false**; // |

dt+= 0.01f; // if we want to, in the future, auto update every item per cycle. (not currently..)

**if**(autoWorld == **true**) // activated by hotkey - we will update each rotation matrix by 0.01 radians per cycle - smooth and effective.

{

rotY += 0.01f; // Y increment

rotX += 0.01f; // X increment

rotZ += 0.01f; // Z increment

}

**if**(GetAsyncKeyState(VK\_SPACE)) // this will destroy the spash screen and begin with the world view stage when the spacebar is pressed

bFirstLaunch = **false**;

// ----- HOTKEYS -----

// \* Depending on which hotkey is currently DOWN, we will change the translation of the sprite

// or the rotation.

// \* The DUAL-AND validation statements (line 141 to 157) are to ensure the sprite is within

// the bounds of the screen at the world view (average projection size/view)

**if**(GetAsyncKeyState(VK\_UP)) //

{

rotZ -= 0.05f;

xRot = **true**;

}

**if**(GetAsyncKeyState(VK\_DOWN))

{

rotX += 0.05f;

xRot = **true**;

}

**if**(GetAsyncKeyState(VK\_LEFT))

{

rotY += 0.05f;

yRot = **true**;

}

**if**(GetAsyncKeyState(VK\_RIGHT))

{

rotY -= 0.05f;

yRot = **true**;

}

**if**((GetAsyncKeyState(0x45) && (transY <= 10))) // line 141

{

transY += 0.1f;

yTrans = **true**;

}

**if**((GetAsyncKeyState(0x44) && (transY >= -10)))

{

transY -= 0.1f;

yTrans = **true**;

}

**if**((GetAsyncKeyState(0x53)) && (transX <= 19))

{

transX += 0.1f;

xTrans = **true**;

}

**if**((GetAsyncKeyState(0x46)) && (transX >= -19)) // line 157

{

transX -= 0.1f;

xTrans = **true**;

}

**if**(GetAsyncKeyState(VK\_LBUTTON))

{

transZ += 0.1f;

zTrans = **true**;

}

**if**(GetAsyncKeyState(VK\_RBUTTON))

{

transZ -= 0.1f;

zTrans = **true**;

}

**if**(GetAsyncKeyState(0x4F)) // change the current render state of our sprite, this allows for wireframe and full-fill.

{

DWORD fillMode;

m\_pDevice->GetRenderState(D3DRS\_FILLMODE, &fillMode);

**if**(fillMode == D3DFILL\_SOLID) fillMode = D3DFILL\_WIREFRAME;

**else** fillMode = D3DFILL\_SOLID;

m\_pDevice->SetRenderState(D3DRS\_FILLMODE, fillMode);

Sleep(200);

}

**if**(GetAsyncKeyState(0x0D))

{

**if**(autoWorld == **false**) autoWorld = **true**;

**else** autoWorld = **false**;

Sleep(200);

}

D3DXMATRIX viewMatrix; // the view matrix

D3DXVECTOR3 position = D3DXVECTOR3(0.0f, 0.0f, 8.0f); //the position of our camera

D3DXVECTOR3 target = D3DXVECTOR3(targX, targY, targZ); //the lookat target of our camera

// LH = LEFT HANDED. Builds a left-handed, look-at matrix.

// we need to tell direct3d which way is up, down, left and right, as it can't assume this, unfortunately.

D3DXMatrixLookAtLH(&viewMatrix,

&position, // cam (or eye)

&target, // target

&D3DXVECTOR3(0.f, 1.0f, 0.0f)); // up direction

// the VIEW matrix is the transformation matrix declared and set-up above.

m\_pDevice->SetTransform(D3DTS\_VIEW, &viewMatrix);

// These functions apply the Mathematical matrix operations to our rotation vcariables.

D3DXMatrixRotationY(&Ry, rotY);

D3DXMatrixRotationX(&Rx, rotX);

D3DXMatrixRotationZ(&Rz, rotZ);

D3DXMatrixScaling(&Scale, 1.0f, 2.0f, 1.0f);

//D3DXMatrixTranslation(&Translation, rotX, rotY, rotZ);

D3DXMatrixTranslation(&Translation, transX, transY, transZ);

World = Scale \* Rx \* Ry \* Rz \* Translation;

}

**void** \_winmain::Render(**float** dt)

{

BeginScene(); // standard prolog

**if**(bFirstLaunch != **true**)

{

m\_pDevice->SetTransform(D3DTS\_WORLD, &World);

m\_pDevice->SetStreamSource(0, VB, 0, **sizeof**(Vertex));

m\_pDevice->SetIndices(IB);

m\_pDevice->SetFVF(Vertex::FORMAT);

**for** (**int** i = 0; i < 6; i++)

m\_pDevice->DrawPrimitive(D3DPT\_TRIANGLESTRIP, i \* 4, 2);

// display to the screen what's happening depending on the current transform state (left, right, x, y...)

**if**(xRot == **true**)

statistics(1);

**if**(yRot == **true**)

statistics(2);

**if**(xTrans == **true**)

statistics(3);

**if**(yTrans == **true**)

statistics (4);

**if**(zTrans == **true**)

statistics(5);

} **else**

{

drawInfo(); // the spacebar hasn't been pressed, draw our splash info

drawMenu(); // and menu

}

EndScene(); // standard epilog

}

**bool** \_winmain::Init()

{

**if**(Direct3D::Init())

{

Direct3D::InitializeD3D(); // start Direct3D ready for usage.

// BerZ'ophus, 2013

Vertex\_UDTx vertices[] = { // define our CUBE.

// Front Face (1-2-3-4)

{ -1.0f, 1.0f, -1.0f, 0xffffffff, 0.0f, 0.0f },

{ 1.0f, 1.0f, -1.0f, 0xffffffff, 1.0f, 0.0f },

{ -1.0f, -1.0f, -1.0f, 0xffffffff, 0.0f, 1.0f },

{ 1.0f, -1.0f, -1.0f, 0xffffffff, 1.0f, 1.0f },

// Right Face (2-6-4-8)

{ 1.0f, 1.0f, -1.0f, 0xff7f7fff, 0.0f, 0.0f },

{ 1.0f, 1.0f, 1.0f, 0xff7f7fff, 1.0f, 0.0f },

{ 1.0f, -1.0f, -1.0f, 0xff7f7fff, 0.0f, 1.0f },

{ 1.0f, -1.0f, 1.0f, 0xff7f7fff, 1.0f, 1.0f },

// Top Face (5-6-1-2)

{ -1.0f, 1.0f, 1.0f, 0xff7fff7f, 0.0f, 0.0f },

{ 1.0f, 1.0f, 1.0f, 0xff7fff7f, 1.0f, 0.0f },

{ -1.0f, 1.0f, -1.0f, 0xff7fff7f, 0.0f, 1.0f },

{ 1.0f, 1.0f, -1.0f, 0xff7fff7f, 1.0f, 1.0f },

// Back Face (6-5-8-7)

{ 1.0f, 1.0f, 1.0f, 0xffff7f7f, 0.0f, 0.0f },

{ -1.0f, 1.0f, 1.0f, 0xffff7f7f, 1.0f, 0.0f },

{ 1.0f, -1.0f, 1.0f, 0xffff7f7f, 0.0f, 1.0f },

{ -1.0f, -1.0f, 1.0f, 0xffff7f7f, 1.0f, 1.0f },

// Left Face (5-1-7-3)

{ -1.0f, 1.0f, 1.0f, 0xffffff7f, 0.0f, 0.0f },

{ -1.0f, 1.0f, -1.0f, 0xffffff7f, 1.0f, 0.0f },

{ -1.0f, -1.0f, 1.0f, 0xffffff7f, 0.0f, 1.0f },

{ -1.0f, -1.0f, -1.0f, 0xffffff7f, 1.0f, 1.0f },

// Bottom Face (3-4-7-8)

{ -1.0f, -1.0f, -1.0f, 0xff3f9f3f, 0.0f, 0.0f },

{ 1.0f, -1.0f, -1.0f, 0xff3f3f3f, 1.0f, 0.0f },

{ -1.0f, -1.0f, 1.0f, 0xff3f3f3f, 0.0f, 1.0f },

{ 1.0f, -1.0f, 1.0f, 0xff3f3f3f, 1.0f, 1.0f }

};

D3DXMATRIX view; // view matrix

D3DXMATRIX proj; // projection matrix

//view setup

D3DXVECTOR3 position = D3DXVECTOR3(0.0f, 0.0f, -1.0f);

D3DXVECTOR3 target = D3DXVECTOR3(0.0f, 0.0f, 1.0f);

D3DXVECTOR3 up = D3DXVECTOR3(0.0f, 1.0f, 0.0f);

D3DXMatrixLookAtLH(&view, &position, &target, &up);

m\_pDevice->SetTransform(D3DTS\_VIEW, &view);

//projection (camera) and D3D default setup

D3DXMatrixPerspectiveFovLH(&proj, D3DX\_PI / 4, **static\_cast**<**float**>(m\_nClientWidth)/m\_nClientHeight,

1.0f, 1000.0f);

m\_pDevice->SetTransform(D3DTS\_PROJECTION, &proj);

m\_pDevice->SetRenderState(D3DRS\_LIGHTING, **false**);

m\_pDevice->SetRenderState(D3DRS\_FILLMODE, D3DFILL\_WIREFRAME);

m\_pDevice->SetRenderState(D3DRS\_SHADEMODE, D3DSHADE\_GOURAUD);

m\_pDevice->CreateVertexBuffer(**sizeof**(vertices), 0, Vertex::FORMAT, D3DPOOL\_MANAGED,

&VB, NULL);

m\_pDevice->CreateIndexBuffer(36 \* **sizeof**(WORD),

D3DUSAGE\_WRITEONLY,

D3DFMT\_INDEX16,

D3DPOOL\_MANAGED,

&IB,

NULL);

VOID\* pVerts;

VB->Lock(0, **sizeof**(vertices), (**void**\*\*)&pVerts, 0);

memcpy(pVerts, vertices, **sizeof**(vertices));

VB->Unlock();

//VOID\* pIndices;

//IB->Lock(0, sizeof(indices), (void\*\*)&pIndices, 0);

//memcpy(pIndices, indices, sizeof(indices));

//IB->Unlock();

HRESULT hr = D3DXCreateTextureFromFile(m\_pDevice, TEXT("tardisTextureMap.png"), &texture);

**if**(hr != D3D\_OK)

MessageBoxA(NULL, "Failed..", "BG", MB\_ICONERROR);

m\_pDevice->SetTexture(0, texture);

**return** **true**;

}

**else** **return** **false**;

}

**int** WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, **int** nCmdShow)

{

\_winmain main(hInstance);

**if**(!main.Init()) **return** 1;

**else** **return** main.Run();

}