// (c) Martyn Prigmore 2013 and beyond. All rights reserved.

//

// Declares classes and helper methods we need to use DirectX for games programming.

//

// Notes

// 1) In header files I use fully qualified identifiers.

//

// Design simplifications for a teaching tool

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// This code is intended as a teaching tool for first year students of games

// programming. It provides a very simple starting point for a Windows Store

// DirectX game. The design makes some simplyfying assumptions that are not

// realistic for a proper game, but are reasonable in a teaching tool.

//Include DirectXTK libraries

#include <CommonStates.h>

#include <SimpleMath.h>

#include <SpriteBatch.h>

#include <SpriteFont.h>

#include <vector>

#include <string>

#include <stdlib.h>

#include "GameController.h"

///<summary>Simple class for implementing a sprite based (i.e. 2D) game</summary>

///<remarks>

///The class is defined as a plain old C++ class, but to work in a Windows Store

///app it must use the C++ Component Extensions (C++/CX).

///To work with DirectX component object model (COM) objects it must also use the

///Windows Runtime C++ Template Library (WRL). These are advanced topics, but

///don't impact on our use of this class to develop a sprite game.

///Further information at:

///http://msdn.microsoft.com/en-us/library/windows/apps/hh438466(v=vs.110).aspx

///http://msdn.microsoft.com/en-us/library/windows/apps/hh699871(v=vs.120).aspx

///</remarks>

class SpriteGame

{

public:

SpriteGame();

virtual ~SpriteGame();

///<summary>Creates resources that are independent of the Direct3D device</summary>

///<remarks>

///For example, could set up Direct2D, DirectWrite and Windows Imaging Component

///(WIC) in this method.

///</remarks>

virtual void CreateDeviceIndependentResources();

///<summary>Creates the Direct3D device and related resources</summary>

virtual void CreateDeviceResources();

///<summary>Creates resources that depend on the size of the window<summary>

///<remarks>

///These resources need to the re-initialized when the window changes size.

///</remarks>

virtual void CreateWindowSizeDependentResources();

///<summary>Initializes the Direct3D objects and prepares them for use</summary>

virtual void Initialize(Windows::UI::Core::CoreWindow^ ourWindow);

///<summary>Updates the state of game objects.</summary>

virtual void Update(int tickTotal, int tickDelta, float secondsTotal, float secondsDelta);

///<summary>Renders (i.e. draws) the game graphics for a single animation frame</summary>

virtual void Render();

///<summary> Presents the rendered animation frame to the monitor.</summary>

virtual void Present();

protected:

///<summary>Helper method: convert measurements in DIPs to measurements in physical pixels</summary>

///<remarks>

///Direct2D uses Device Independent Pixels (DIPs) as its basic unit. One DIP

///is defined as 1/96 of a logical inch.

///Each monitor will have a Dots Per (Logical) Inch setting: it's DPI. With

///a DPI of 96, one DIP is equal to one physical pixel. However, with a higher

///DPI this breaks down. E.g. at a DPI of 120 one DIP is equal to 1.25 physical

///pixels.

///Direct2D uses the DPI to automatically convert DIPs to physical pixels when

///it draws things to the screen. The CoreWindow object also uses DIPs to

///specify its bounds and pointer position.

///However in Direct3D the swap chain and render target need to be specified in

///physical pixels hence this method!

///</remarks>

float ConvertDipsToPixels(float dips);

//The CoreWindow object for this app

Platform::Agile<Windows::UI::Core::CoreWindow> coreWindow;

//The DirectX objects

//Note: These objects are automatically freed when their last reference goes

// out of scope when the app terminates; see http://msdn.microsoft.com/en-us/library/windows/apps/hh780565.aspx.

Microsoft::WRL::ComPtr<ID3D11Device1> d3dDevice;

Microsoft::WRL::ComPtr<ID3D11DeviceContext1> d3dDeviceContext;

Microsoft::WRL::ComPtr<IDXGISwapChain1> swapChain;

Microsoft::WRL::ComPtr<ID3D11RenderTargetView> backBufferTarget;

//Window and graphics hardware properties, cached for convenient access

D3D\_FEATURE\_LEVEL d3dFeatureLevel;

Windows::Foundation::Size backBufferTargetSize;

Windows::Foundation::Rect windowBounds;

Windows::Graphics::Display::DisplayOrientations displayOrientation;

// DirectX Tool Kit objects

//Note: These should be protected since only the SpriteGame object will use them

std::unique\_ptr<DirectX::SpriteBatch> spriteBatch; //The object that gets our sprites ready for rendering by DirectX

std::unique\_ptr<DirectX::CommonStates> commonStates; //Standard settings for adjusting the SpriteBatch object to work with our images

std::unique\_ptr<DirectX::SpriteFont> spriteFont;

std::unique\_ptr<DirectX::SpriteFont> eleFont;

//Game World

//Note: Again, protected member variables as no other object will need to get

//or set their values.

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> rockTexture;

DirectX::SimpleMath::Vector2 rockPosition;

DirectX::SimpleMath::Color rockColor;

std::unique\_ptr<RECT> rockSourceRect;

//For the cabinet sprite

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> cabinetTexture;

DirectX::SimpleMath::Vector2 cabinetPosition;

DirectX::SimpleMath::Color cabinetColor;

std::unique\_ptr<RECT> cabinetSourceRect;

//For the grass animation sprite

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> grassTexture;

DirectX::SimpleMath::Vector2 grassPosition;

DirectX::SimpleMath::Color grassColor;

std::unique\_ptr<RECT> grassSourceRect;

int grassFrame;

int grassLastFrameChange;

const int grassSheetLength;

int grassIncrement;

// For the moving square

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> squareTexture;

DirectX::SimpleMath::Vector2 squarePosition;

DirectX::SimpleMath::Color squareColor;

std::unique\_ptr<RECT> squareSourceRect;

float squareVelocity;

// For the set of barrels

const int barrelFrameHeight;

const int barrelFrameWidth;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> barrelTexture;

DirectX::SimpleMath::Vector2 barrelPosition;

DirectX::SimpleMath::Color barrelColor;

std::unique\_ptr<RECT> barrelSourceRect;

// For the set of flipped barrels

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> flippedTexture;

DirectX::SimpleMath::Vector2 flippedPosition;

DirectX::SimpleMath::Color flippedColor;

std::unique\_ptr<RECT> flippedSourceRect;

// For the partial image of cabinets

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> partialTexture;

DirectX::SimpleMath::Vector2 partialPosition;

DirectX::SimpleMath::Color partialColor;

std::unique\_ptr<RECT> partialSourceRect;

// For the cupboard

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> cupboardTexture;

DirectX::SimpleMath::Vector2 cupboardPosition;

DirectX::SimpleMath::Color cupboardColor;

std::unique\_ptr<RECT> cupboardSourceRect;

// For the trophy

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> trophyTexture;

DirectX::SimpleMath::Vector2 trophyPosition;

DirectX::SimpleMath::Color trophyColor;

std::unique\_ptr<RECT> trophySourceRect;

// For the glass doors

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> glassTexture;

DirectX::SimpleMath::Vector2 glassPosition;

DirectX::SimpleMath::Color glassColor;

std::unique\_ptr<RECT> glassSourceRect;

// For the reversed animated grass sprite

const int reverseSheetLength;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> reverseTexture;

DirectX::SimpleMath::Vector2 reversePosition;

DirectX::SimpleMath::Color reverseColor;

std::unique\_ptr<RECT> reverseSourceRect;

int reverseFrame;

int reverseLastFrameChange;

int reverseSubtraction;

// For the disc

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> discTexture;

DirectX::SimpleMath::Vector2 discPosition;

DirectX::SimpleMath::Color discColor;

std::unique\_ptr<RECT> discSourceRect;

DirectX::SimpleMath::Vector2 discVelocity;

DirectX::SimpleMath::Vector2 leftPosition;

DirectX::SimpleMath::Vector2 rightPosition;

DirectX::SimpleMath::Vector2 bottomPosition;

DirectX::SimpleMath::Vector2 topPosition;

// For the plasma rock

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> plasmaTexture;

DirectX::SimpleMath::Vector2 plasmaPosition;

DirectX::SimpleMath::Color plasmaColor;

std::unique\_ptr<RECT> plasmaSourceRect;

// For the background

DirectX::SimpleMath::Color clearToThisColor;

DirectX::SimpleMath::Color addToThisColor;

DirectX::SimpleMath::Color addGreen;

DirectX::SimpleMath::Color addBlue;

// For the worm

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> wormTexture;

DirectX::SimpleMath::Vector2 wormPosition;

DirectX::SimpleMath::Color wormColor;

std::unique\_ptr<RECT> wormSourceRect;

int wormColumn;

int wormRow;

int wormLastFrameChange;

int wormFrameRate;

const int wormFrameWidth;

const int wormFrameHeight;

const int wormSheetWidth;

// For the gnome and troll

const int creatureFrameWidth;

const int creatureFrameHeight;

const int creatureSheetWidth;

DirectX::SimpleMath::Vector2 creaturePosition;

DirectX::SimpleMath::Color creatureColor;

std::unique\_ptr<RECT> creatureSourceRect;

int creatureFrame;

int creatureLastFrameChange; //In ticks

int creatureFrameRate; //In ticks

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> creatureGnomeTexture;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> creautureTrollTexture;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> creatureCurrentTexture;

// For the analog controlled disc

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> movingDiscTexture;

DirectX::SimpleMath::Vector2 movingDiscPosition;

const float movingDiscSpeed;

DirectX::SimpleMath::Color movingDiscColor;

Windows::Foundation::Rect movingDiscboundingBox;

DirectX::SimpleMath::Color changeColor;

// For the troll!

const int trollFrameWidth;

const int trollFrameHeight;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> trollTexture;

DirectX::SimpleMath::Vector2 trollPosition;

DirectX::SimpleMath::Color trollColor;

std::unique\_ptr<RECT> trollSourceRect;

// For the game pad

GameController gamepad;

// For the number of tick

std::wstring trackTicks;

std::wstring trackTrigger;

//For the random time intervals

int interval;

};

///<summary>Helper function: Allows DirectX APIs to work with exceptions</summary>

inline void ThrowIfFailed(HRESULT hr)

{

if (FAILED(hr))

{

// Set a breakpoint on this line to catch Win32 API errors.

throw Platform::Exception::CreateException(hr);

}

}