Graphics Support

Lucid3D.System Design Specification

# Overview

## Customer Scenarios

The key scenarios to deliver in the graphics portion of the system layer are:

1. Streamlined basic graphics stack creation if they don’t have their own
2. Streamlined resource creation for common cases (buffers, textures, shaders, etc…)
3. Bundled convenience data for various resources (stride, offset, primitive type, etc…)
4. Lightweight low-level shader management system
   1. Determining input layout from a vertex shader
   2. Managing constant buffers and shader variables
5. Simple quad rendering, including full screen
6. Simple primitive rendering (lines, boxes, etc…)

## Goals

1. Lightweight to use. Should be fairly non-intrusive to the existing code base if being used for support functionality.
2. Intuitive and easy to find functionality in the component.
3. Very high performance. In many cases, these pieces of functionality will be the foundation of games, and must allow that game to reach its maximum potential.
4. Work well with existing code, and work well with other Lucid3D code. Consistency. This code is designed to be complimentary to Direct3D, not replace it.
5. Help clients reduce the amount of code they need to write for common scenarios.
6. Help manage the graphics stack and related resources in a consistent and convenient manner.

## Non-Goals

1. Be a replacement for all the graphics code in a product.
2. Wrap or façade Direct3D completely.

# Features

To achieve the goals and scenarios for this component, the work will be divided into the following features. It is intentional that these line up almost exactly with the customer scenarios:

1. Basic graphics stack creation and/or integration with existing stack.
2. Basic resource creation and management
3. Low-level shader management system
4. Simple shape rendering, including quads

# Detailed Design

## Basic Graphics Stack Creation

For many developers who don’t have specific graphics device or configuration needs, and who don’t have an existing graphics stack, a basic standard Direct3D graphics set up will do. Since this is fairly straightforward, yet tedious and verbose to create properly and manage, it provides a lot of value to have Lucid3D help out with this step.

In order for this to be successful, a couple of small requirements must be met: The function, or set of functions, must be significantly easier to use than manually setting up the stack. There needs to be a balance between how much flexibility is offered versus how hassle free it is to get up and running. Also, as the rest of Lucid3D never assumes that you created the stack this way, we must expose all the underlying objects if asked for them, so that we can cooperate properly with the other subsystems.

The basic steps involved in creating a Direct3D device, immediate device context, swap chain, default render target (back buffer), and default depth and stencil buffer are:

1. Set up swap chain and back buffer flags
2. Set up device creation flags and feature levels
3. Create the device, context, and swap chain
4. Obtain the back buffer from the swap chain and create a render target view around it
5. Create a depth/stencil texture and resource view

All of the created resource should be returned to the client if they desire, as they are free to do what they want with these. The render target and depth/stencil view can easily be obtained from the context later, so it’s not a problem if the caller doesn’t want them right. However, the swap chain cannot easily be obtained later, so we must require that clients take that object from us on creation.

As input, we require the window handle or object (in WinRT) that the swap chain should be bound to. We don’t own the window and should allow the client to pass in any window they like. Additionally, we should receive some desired dimensions and a flag for whether or not we are to create a full screen swap chain or not. Prototype objects and methods follow:

struct GraphicsDeviceSettings

{

byte numFeatureLevels;

D3D\_FEATURE\_LEVEL\* featureLevels;

HWND hwnd;

ushort backBufferWidth;

ushort backBufferHeight;

bool fullscreen;

};

\_\_interface IGraphicsDevice

{

ID3D11Device\* GetDevice();

IDXGISwapChain\* GetSwapChain();

ID3D11DeviceContext\* GetImmediateContext();

ID3D11RenderTargetView\* GetRenderTargetView();

ID3D11DepthStencilView\* GetDepthStencilView();

D3D\_FEATURE\_LEVEL GetFeatureLevel() const;

void Clear( XMFLOAT4 color, float depth, byte stencil );

void ClearRenderTarget( XMFLOAT4 color );

void ClearDepthStencil( float depth, byte stencil );

HRESULT Present( bool waitForVSync );

};

typedef std::shared\_ptr<IGraphicsDevice> GraphicsDevicePtr;

HRESULT CreateGraphicsDevice(

\_\_in const GraphicsDeviceSettings& settings,

\_\_deref\_out GraphicsDevicePtr\* ppGraphicsDevice );

Note that this method creates a Lucid3D object, exposing the IGraphicsDevice interface. From this interface, you get access to all of the components of the basic graphics stack, such as the swap chain and default views. When the last reference to this type goes away and the Lucid type is deleted, it will release its references to the Direct3D types.

Because Lucid3D is the one creating these resources, it will associate custom names with the resources it creates. This allows easier leak debugging if any of the objects don’t get cleaned up on debug builds.

An equally important scenario is to construct a Lucid3D graphics wrapper around an existing graphics stack. This allows clients to create their own devices and swap chains and then pass them to Lucid3D to get wrapped in the same IGraphicsDevice interface. This gives them all the same functionality that the interface will expose, without restricting them to use the creation path provided by Lucid.

The entry points for this are:

HRESULT CreateGraphicsDevice(

\_\_in ID3D11Device\* pDevice,

\_\_in IDXGISwapChain\* pSwapChain,

\_\_deref\_out GraphicsDevicePtr\* ppGraphicsDevice );

The function obtains the immediate context and related views directly from the device, which helps reduce the parameters the client is required to pass in.

## Basic Resource Creation and Management

Creating graphics resources isn’t particularly tricky, but it’s a tedious task and there are subtle performance pitfalls around setting up access flags. Lucid can provide value by streamlining common resource creation and management patterns, including handling the optimal allocation and configuration of these resources. There are three main types of resources which Lucid can assist in creating and managing: buffers, shaders, and textures.

Lucid3D exposes an IDeviceContext interface, which wraps a device context. This object can be constructed off any existing context, and provides several useful resource binding methods which can be used with the resources described below. Here is the API prototype for the device context portion:

\_\_interface IDeviceContext

{

ID3D11DeviceContext\* GetDeviceContext();

void SetVertexShader( \_\_in VertexShaderPtr& vertexShader );

void SetPixelShader( \_\_in PixelShaderPtr& pixelShader );

void SetVertexBuffer( \_\_in BufferPtr& vertexBuffer );

void SetVertexBuffers( \_\_in BufferPtr& vertexBuffer, \_\_in BufferPtr& instanceBuffer);

void SetIndexBuffer( \_\_in BufferPtr& indexBuffer );

void SetTexture2D( \_\_in Texture2DPtr& texture );

void DrawIndexed( uint numIndices, uint baseVertex, uint indexOffset );

};

HRESULT CreateDeviceContext(

\_\_in GraphicsDevicePtr graphicsDevice,

\_\_in ID3D11DeviceContext\* pContext,

\_\_deref\_out DeviceContextPtr\* ppDeviceContext );

### Buffers

The most common usage of data buffers is to create static geometry descriptions in the form of vertex and index buffers, and then use these each frame to draw the geometry. Since, in this common case, the data never needs to be updated, it makes sense to optimize these buffers for GPU access and disallow CPU access. Another common scenario is to create buffers that are designed to be updated frequently, such as for dynamic data or instance data. These have very different allocation and configuration requirements than the first case. Lucid can help handle all this for the client, providing the controls to pick which in an intuitive manner. Prototype APIs follow:

namespace BufferType

{

enum Enum

{

Unknown,

Vertex,

Index,

};

}

namespace ResourceAccess

{

enum Enum

{

None = 0x00,

Read = 0x01,

Write = 0x02,

};

};

\_\_interface IGraphicsResource

{

ID3D11Resource\* GetResource();

ResourceAccess::Enum GetGPUAccess() const;

ResourceAccess::Enum GetCPUAccess() const;

D3D11\_USAGE GetUsage() const;

};

\_\_interface IBuffer

{

ID3D11Buffer\* GetBuffer();

BufferType::Enum GetType() const;

uint GetStride() const;

uint GetNumElements() const;

};

\_\_interface IGraphicsDevice

{

// <rest of graphics device interface omitted>

// Creates a vertex buffer, optionally filling it with data

HRESULT CreateVertexBuffer(

\_\_in uint bytesPerVertex,

\_\_in uint numVertices,

\_\_in ResourceAccess::Enum gpuAccess,

\_\_in ResourceAccess::Enum cpuAccess,

\_\_in\_opt void\* pInitialData,

\_\_deref\_out UniformBufferPtr\* ppBuffer );

// Creates an index buffer, optionally filling it with data

HRESULT CreateIndexBuffer(

\_\_in uint numIndices,

\_\_in ResourceAccess::Enum gpuAccess,

\_\_in ResourceAccess::Enum cpuAccess,

\_\_in\_opt void\* pInitialData,

\_\_deref\_out UniformBufferPtr\* ppBuffer );

};

These creation methods off of the Lucid GraphicsDevice allow for convenient, optimized creation of the buffer resources. There are, however, cases where the client already has an existing buffer that they wish to use with the rest of the Lucid system. The following two methods in the Graphics namespace allow for that:

HRESULT CreateVertexBuffer(

\_\_in ID3D11Buffer\* pBuffer,

\_\_deref\_out UniformBufferPtr\* ppVertexBuffer );

HRESULT CreateIndexBuffer(

\_\_in ID3D11Buffer\* pBuffer,

\_\_deref\_out UniformBufferPtr\* ppIndexBuffer );

### Shader Resources

In addition to vertex and index buffers, the next most common type of resources are probably shaders and associated constant buffers. These can also be somewhat tricky or tedious to create efficiently, especially the input layouts for the shaders and the variable management for the constant buffers. This is probably the one area where Lucid3D truly excels at providing a powerful, flexible way to manage these aspects. The Lucid3D shader system will be described in more detail later in the shader management section, but the resource creation is prototyped as follows:

\_\_interface IShader

{

uint GetNumConstantBuffers() const;

ConstantBufferPtr GetConstantBufferByIndex(uint index);

ShaderVariablePtr FindVariableByName(PCSTR const name);

};

\_\_interface IVertexShader : public IShader

{

ID3D11VertexShader\* GetShader();

ID3D11InputLayout\* GetInputLayout();

};

\_\_interface IPixelShader : public IShader

{

ID3D11PixelShader\* GetShader();

};

\_\_interface IConstantBuffer

{

ID3D11Buffer\* GetBuffer();

uint GetNumVariables() const;

ShaderVariablePtr GetVariableByIndex(uint index);

ShaderVariablePtr GetVariableByName(PCSTR const name);

// Resolve flushes cached variable state to the device context

void Resolve( \_\_in ID3D11DeviceContext\* pContext );

};

\_\_interface IShaderVariable

{

PCSTR const GetName() const;

void SetFloat(float value);

void SetVector2(XMFLOAT2 value);

void SetVector3(XMFLOAT3 value);

void SetVector4(XMFLOAT4 value);

void SetMatrix(XMFLOAT4X4 value);

};

\_\_interface IGraphicsDevice

{

// <rest of graphics device interface omitted>

// Creates a runtime vertex shader from the given bytecode

HRESULT CreateVertexShader(

\_\_in const void\* pShaderByteCode,

\_\_in size\_t byteCodeLength,

\_\_deref\_out VertexShaderPtr\* ppShader );

// Creates a runtime vertex shader from the given bytecode, using custom input layout

HRESULT CreateVertexShader(

\_\_in const void\* pShaderByteCode,

\_\_in size\_t byteCodeLength,

\_\_in\_ecount(numElements) D3D11\_INPUT\_ELEMENT\_DESC\* pInputElements,

\_\_in uint numElements,

\_\_deref\_out VertexShaderPtr\* ppShader );

// Creates a runtime pixel shader from the given bytecode

HRESULT CreatePixelShader(

\_\_in const void\* pShaderByteCode,

\_\_in size\_t byteCodeLength,

\_\_deref\_out PixelShaderPtr\* ppShader );

};

The methods take in the compiled byte code for the shader, and use reflection interfaces to parse out and build up support structures, such as the input layout and constant buffers. The Lucid device context object will resolve constant buffers during binding of the shader. However, if the client makes changes to the constant buffer after binding the shader, it should call Resolve again manually to ensure the GPU has the correct data.

### Textures

Textures are another extremely common resource for which Lucid helps streamline creation and management of. The API is simple and straightforward:

\_\_interface ITexture2D : IGraphicsResource

{

ID3D11Texture2D\* GetTexture();

ID3D11ShaderResourceView\* GetResourceView();

uint GetWidth() const;

uint GetHeight() const;

DXGI\_FORMAT GetFormat() const;

};

\_\_interface IGraphicsDevice

{

// <rest of graphics device interface omitted>

// Creates a new 2D texture

HRESULT CreateTexture2D(

\_\_in uint width,

\_\_in uint height,

\_\_in DXGI\_FORMAT surfaceFormat,

\_\_in bool generateMipChain,

\_\_in ResourceAccess::Enum gpuAccess,

\_\_in ResourceAccess::Enum cpuAccess,

\_\_in\_opt void\* pInitialData,

\_\_deref\_out Texture2DPtr\* ppTexture );

};

**TODO: Add support for non-2D textures.**

## Shader Management System

One of the most flexible and useful features of Lucid3D.System’s graphics layer is the shader management system. The shader system provides facilities to parse out and auto-generate input layouts for vertex shaders, parse and construct an object model for the variables accessible to the shader through constant buffer and shader variable interfaces, and allows for efficient variable manipulation before bulk resolving the updates to the GPU. The actual resources and functions to access the shader system are provided above in the resource creation and management section. Here, we will show a sample use of the system:

The most common scenario is a game using a vertex and pixel shader. They know the vertex shader has a variable called g\_WorldViewProjection, which is a 4x4 matrix. The code to build the shaders, input layout and constant buffers, and then set the matrix and bind all these resources to the GPU pipeline follows:

CHECKHR( CompileShader("BasicEffect.hlsl", "vsMain", "vs\_4\_0", &spByteCode) );

CHECKHR( graphicsDevice->CreateVertexShader(spByteCode->GetBufferPointer(), spByteCode->GetBufferSize(), &spVS) );

CHECKHR( CompileShader("BasicEffect.hlsl", "psMain", "ps\_4\_0", &spByteCode) );

CHECKHR( graphicsDevice->CreatePixelShader(spByteCode->GetBufferPointer(), spByteCode->GetBufferSize(), &spPS) );

auto spVar = spVS->FindVariableByName("g\_WorldViewProjection");

if (spVar)

spVar->SetMatrix(worldViewProj);

// Bind everything that the GPU needs, including the input layout & constant buffers

spContext->SetVertexShader(spVS);

spContext->SetPixelShader(spPS);

## Simple Shape and Quad Drawing

It is quite common for games to require simple shape drawing. This could be for debug rendering of shapes, or for things like quad rendering or fullscreen effects. In any case, it’s something that’s nontrivial to do and having this facility available is useful. The simple shape rendering code must be efficient, as it could be used extensively for things like debug renders or sprite drawing.

### Quads

For screen space quad rendering, the client will need to provide the inputs as normalized viewport coordinates. There are a few reasons for not taking resolution-dependent coordinates. The first is that it would add considerable overhead for Lucid to track resolution and backbuffer size changes the application might be making, and cumbersome to take in the resolution as parameters to the quad drawing system. The second, and more important reason however, is that most UI or other screen space rendering systems will already be in normalized coordinates anyways for layout purposes. Having to convert to screen space, just to have Lucid3D convert back to normalized coordinates is inefficient.

To support quad rendering, Lucid3D’s system layer will contain a built in vertex shader which is set up for instancing, and a vertex and index buffer to represent the quad. The quad rendering function, available on the device context interface, will take in an array of quad coordinates and will set the appropriate state on the context and display the quads using whatever active pixel shader and textures are set. The pixel shader input is just a float2 texture coordinate.

\_\_interface IDeviceContext

{

// the rest of the interface is omitted from this snippet

HRESULT InitializeQuadRendering();

HRESULT DrawQuads( uint numQuads, \_\_in RectangleF\* quads );

}

### Simple Line Drawing for Simple Shapes

TBD