NVIDIA™

DirectX 8 Graphics Overview

Dave Horne
Technical Developer Relations
NVIDIA Corporation

Initialisation Changes

- Merged DirectDraw and Direct3D
  - Simplified API
  - Improved data allocation and performance
- Promoting overall robustness
  - App code simplification
  - Fewer test paths

Simplified API

- Don't need to use QueryInterfaces, and no GUIDs in DirectX 8
- One line global D3D object creation
  - `m_pD3D = Direct3DCreate8(D3D_SDK_VERSION);`
  - `D3D_SDK_VERSION` ensures compilation with correct headers
- No more callbacks!
  - Query Adapters(physical pieces of hardware), devices and modes directly from D3D object
- `GetDeviceCaps()` replaces `GetCaps()`
  - called from D3D object
  - No need to create devices to determine caps

Device Types

- HAL device
  - Supports hardware accelerated rasterization and both hardware and software vertex processing
- Reference device
  - Supports all features of API correctly (if not quickly!) in software only. Intended for feature testing, demos, etc, not for retail apps
- Software device
  - DX8 supports ‘pluggable’ software devices (DDK)
  - MS will not be providing a SW device
- HAL or REF device variant : Pure device
  - Hardware only vertex processing
  - Highly restricted querying of device state
  - Performance advantage over non-pure devices

NVIDIA.

Device Creation I

- Set ‘Presentation Parameters’ structure
  - Creation of back buffer(s)
    - Width, height and bit depth for full-screen
  - Windowed/Full-screen
  - Specifying buffer ‘Swap’ type
    - Flip or Copy(Blit)
  - Choose auto depth/stencil creation
  - Multi-sample type
    - Anti-aliasing, etc
  - Full-screen refresh rate
  - Can set 0 for some parameters to get default
    - Set `D3DPRESENTFLAG_LOCKABLE_BACKBUFFER` if you need access to the back buffer
    - Setting this flag <u>will</u> reduce performance

NVIDIA

Device Creation II

- Presentation parameters structure very similar for windowed and full-screen device creation
  - Also used in device `Reset()` from lost device
- Behaviour flags in `CreateDevice()`
  - `D3DCREATE_*_VERTEXPROCESSING`
    - Software, Hardware or Mixed
  - `D3DCREATE_FPU_PRESERVE`
  - `D3DCREATE_MULTITHREADED`
  - `D3DCREATE_PUREDEVICE`
    - Limited renderstate shadowing
    - Hardware vertex processing only

Device Creation III

```
D3DPRESENT_PARAMETERS d3dpp;
memset(&d3dpp, 0, sizeof(d3dpp));

d3dpp.BackBufferWidth                = 1024; // must be 0 in windowed mode
d3dpp.BackBufferHeight               = 768;  // must be 0 in windowed mode
d3dpp.BackBufferFormat               = D3DFMT_A8R8G8B8;
d3dpp.BackBufferCount                = 1;
d3dpp.Windowed                       = FALSE;
d3dpp.SwapEffect                     = D3DSWAPEFFECT_FLIP;
d3dpp.EnableAutoDepthStencil         = TRUE;
d3dpp.AutoDepthStencilFormat         = D3DFMT_D32;
d3dpp.MultiSampleType                = D3DMULTISAMPLE_NONE;
d3dpp.hDeviceWindow                  = g_hwnd;

/* Following elements must be zero for Windowed mode */
d3dpp.FullScreen_RefreshRateInHz      = D3DPRESENT_RATE_DEFAULT;
d3dpp.FullScreen_PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;

pEnum->CreateDevice(
      D3DADAPTER_DEFAULT,          // iAdapter
      D3DDEVTYPE_HAL,              // device type
      hWindow,                     // focus
      D3DCREATE_FPU_PRESERVE,      // dwBehaviorFlags
      &d3dpp,                      // presentation parameters
      &pDevice);                   // [out] pDevice
```

NVIDIA.

Lost Devices I

- Supercedes DX7 Lost Surfaces
- Device can be in an operational or a lost state
  - Lost state caused by loss of focus, Alt-Tab, power management events, etc.
- Once in lost state only error returns are from
  - `Present()`
  - `TestCooperativeLevel()`
- All other methods succeed
  - Even if they don't do anything

Lost Devices II

- App needs to detect and respond to Lost device
  - Rendering surfaces are gone
  - All unmanaged video/agp mem resources are invalid
    - includes VBs and Index buffers as well as textures
  - All HW states, lights and shaders are invalid
- Rebuild the device using the `Reset()` command
  - App MUST free all unmanaged video/agp resources
  - Use same presentation params as `CreateDevice()`
- App must recreate any unmanaged resources, as well as renderstates, lights and shaders

NVIDIA

Lost Devices III

```
// Test the cooperative level to see if it's okay to render
// Call once per render loop
if(FAILED(hr = m_pD3DDevice->TestCooperativeLevel()))
{
    // If the device was lost, do not render until we get it back
    if(D3DERR_DEVICELOST == hr) return S_OK;

    // Check if the device is ready to be reset
    if(D3DERR_DEVICENOTRESET == hr)
    {
        // Call app specific non-d3d managed resource destruction
        DestroyAppManagedResources();

        // Resize the device
        if(FAILED(hr = m_pD3DDevice->Reset(&m_d3dpp))) return hr;

        // Call app specific non-d3d managed resource creation
        CreateAppManagedResources();

        // Restore renderstates, lights, shaders, etc.
        RestoreOtherStuff();
    }

    return hr;
}
```

Resources

- Unified resource manager
  - Geometry
    - Vertex buffers
    - Index buffers
  - Textures
    - 2D Textures
    - Volume textures
    - Cube maps (CubeTexture)

Resource Creation I

- **Explicit types**
  - `CreateVolumeTexture()` **distinct from** `CreateCubeTexture(), CreateTexture()`
  - **No** `CreateSurface(),GetAttachedSurface()`
- **Pool**
  - `D3DPOOL_DEFAULT`
    - **Placed in memory type based on device and usage**
    - **Always lost on device loss**
  - `D3DPOOL_SYSTEMMEM`
    - **Placed in sys mem.**
    - **Never lost**
  - `D3DPOOL_MANAGED`
    - **Managed by D3D or driver.**
    - **Never lost**
    - **Can't use with** `D3DUSAGE_DYNAMIC` **flag**

Resource Creation II

- Usage
  - `D3DUSAGE_DEPTHSTENCIL, D3DUSAGE_RENDERTARGET`
    - Indicates using as depth/stencil or render target
  - `D3DUSAGE_SOFTWAREPROCESSING`
  - `D3DUSAGE_DONOTCLIP`
  - `D3DUSAGE_DYNAMIC, D3DUSAGE_WRITEONLY`
    - Allows driver to place resource in optimal location
- VB only ‘hints’ to allow the driver to put VB in CPU accessible memory if no native HW support
  - `D3DUSAGE_POINTS`
    - Point sprites
  - `D3DUSAGE_RTPATCHES`
    - 'RT' stands for rectangular and triangular high order surfaces
  - `D3DUSAGE_NPATCHES`

NVIDIA.

Resource Creation III

- Format
  - Now an enumerated type
  - Runtime predefines several types of each class
    - Color, luminance, DXTn, vertex, index, etc
  - IHVs may still add vendor specific formats via the FOURCC codes
- Always allocate all `D3DPOOL_DEFAULT` resources before `D3DPOOL_MANAGED`
  - Otherwise D3D resource manager will get an inaccurate picture of available memory

DX8 Vertex Buffer Changes I

- Vertex buffers are now the standard usage for `DrawPrimitive()` and `DrawIndexedPrimitive()`
- 'User Pointer' calls for apps that can't use VBs for some reason
  - `DrawPrimitiveUP()`
  - `DrawIndexedPrimitiveUP()`
  - Don't use! Will almost certainly result in a redundant copy of vertex data by driver
- VBs can consist of *arbitrary data* in multiple *streams*
  - Use *Vertex Shaders* to interpret data

NVIDIA

DX8 Vertex Buffer Changes II

- `CreateVertexBuffer()` changes
  - Size now defined as number of *bytes*
    - DX7 defined it as number of *vertices*
  - Usage flags
  - FVF code can be *zero*
    - Called a *Non-FVF* VB
    - Represents VB stream for vertex shader
    - Format implied by `CreateVertexShader()` declaration
  - Must declare which memory pool required
- No more maximum VB size
  - DX7 max size is 0xffff

Index Buffers I

- New resource type for DX8
- Removes redundant driver copy of indices in the same way VBs stop driver copy of vertices
- `CreateIndexBuffer()` size, usage and pool parameters the same as VB create
  - Format parameter allows creation of either 16 or 32 bit indices
    - `D3DFMT_INDEX16, D3DFMT_INDEX32`
    - 32 bit indices new to DX8

Index Buffers II

- Semantics the same as for VBs
  - Create
  - `Lock()`
  - Fill
  - `Unlock()`
  - Use
- Indices now explicitly set via `SetIndices()`
  - In DX7 they were part of the DP/DIP calls
  - 'User Pointer' calls still require explicit index parameters
    - Similar to DX7
    - Currently set IB is ignored
    - Reminder: Don't use User Pointers!

Locking Resources I

- Grants CPU access to resource
- Texture locks
  - `LockRect()` for 2D textures and cube maps
  - `LockBox()` for volume textures
  - Typically only one lock per resource
- Geometry locks
  - `Lock()` for both index and vertex buffers
  - Multiple locks per buffer allowed
- Must relinquish lock with `UnLock()` before resource can be used by device
  - Any device operations with locked resource will be serialised until resource unlocked

NVIDIA

Locking Resources II

- General lock flags
  - `D3DLOCK_READONLY`
    - Won't write to resource so no recompression on unlock
    - Can't use with VBs/IBs created with `WRITEONLY` flag
  - `D3DLOCK_NOSYSLOCK`
    - No system-wide critical section taken
- Texture specific lock flags
  - `D3DLOCK_NO_DIRTY_UPDATE`
    - Don't update dirty region
- Geometry specific lock flags
  - `D3DLOCK_DISCARD`
  - `D3DLOCK_NOOVERWRITE`

Locking Textures I

- `LockRect()` can lock whole surface or sub-rect of 2D texture or face of cube map
  - Lock with `RECT` or NULL for entire surface
  - Returns `D3DLOCKED_RECT`
    - Consists of *data pointer* and *pitch* (in bytes)
    - Simpler than DX7 surface lock return `DDSURFACEDESC2`
  - Must specify mip level required for lock
  - Cube textures must also specify face

Locking Textures II

- Use `LockBox()` for to lock volume or sub-volume of *3D texture*
  - Lock with `D3DBOX` structure or NULL for entire volume
  - Must also specify mip level
    - All 3 dimensions of each level are divided by 2 (rounding down) down to minimum 1x1x1
  - Returns `D3DLOCKED_BOX`
    - Consists of data pointer, *row pitch* and *slice pitch*
- Compressed DXT formats
  - Can only be locked on 4x4 boundaries
  - Minimum actual size is 4x4 on a side
    - Textures or mip levels less than this are padded

Locking Geometry

- DX8 `Lock()` on VBs and IBs allows sub-ranges to be specified
  - DX7 could only lock whole VB
  - Multiple locks on single buffer permitted
  - Allows efficient usage of large VBs containing multiple models, as you can render from one sub range whilst locking and modifying another
- `D3DLOCK_DISCARD` and `D3DLOCK_NOOVERWRITE` flags are valid only on buffers created with `D3DUSAGE_DYNAMIC`
  - Therefore can't be managed buffers

Programmable Vertex Shaders

- New feature for DX8
- Can be used in place of D3D 'fixed function' T&L pipeline
- Allows extremely versatile geometry handling by using vertex programs to process vertex data
- Operates at the vertex level only
  - No knowledge of other vertices (no primitives)
  - Cannot create or destroy vertices
- Vertex output must be in homogenous clip-space with colour, texture coordinate, fog, and sprite point size as needed

NVIDIA

Streams

- New for DX8
- Allows multiple VBs to be set as data sources for vertex processing
  - Stream elements are combined at draw time
- Single set of indices for all streams in DX8
- GeForce/GeForce2 can support up to 8 vertex streams, fed to the fixed-function T&L pipeline only
- DX8 level hardware must be able to support up to 16 vertex streams, fed to the fixed-function OR vertex shading pipeline

NVIDIA.

Creating Vertex Shaders

- Use `CreateVertexShader()`
  - Pass in a declaration defining input to shader
    - Define format of vertex streams and bindings to vertex shader input registers
    - Bind primitive tessellator to input registers
    - Loading of data into constant memory
  - Pass in pointer to vertex shader program token data
    - E.g. from D3DXAssembleShader
    - Or use NULL to get fixed function processing
    - Usage flag
      - `D3DUSAGE_SOFTWAREPROCESSING` if you're using software vertex processing
  - Returns handle for use in `SetVertexShader()`

NVIDIA

Setting Streams

- Call `SetStreamSource()`
  - Stream index
    - 0 to (maximum number of streams - 1)
  - Pointer to VB
  - Stride of vertex
    - For FVF VBs must be the size of vertex computed from FVF code
    - For non-FVF VBs must match the size implied in the `CreateVertexShader()` declaration
- Number and format of streams set must match the current vertex shader declaration

Using Vertex Shaders

- Ensure that all VB streams have been set
- Set an IB with `SetIndices()` if using indexed data
- Use `SetVertexShader()` to select shader
  - Pass in a handle from `CreateVertexShader()`
  - Or pass a valid FVF code to use the fixed function pipeline
- Use `DrawPrimitive()` or `DrawIndexedPrimitive()` to render primitives

Fixed Function Pipeline

- Can use the existing 'fixed function' T&L pipeline with standard flexible vertex format codes
- Use `SetVertexShader()` with an FVF code instead of a vertex shader handle
  - Only stream 0 is valid
- Example pseudocode

```
gVB = CreateVertexBuffer(); // with FVF code
// Lock, fill, unlock gVB
SetVertexShader(); // with same FVF code
SetStreamSource(0, gVB, sizeof(Vertex));
DrawPrimitive();
```

Fixed Function - Multiple Streams

- Create and set a vertex shader with declarator describing streams, and NULL program
- Create VBs for each of the separate streams, and set them as stream sources
- Pseudocode for a 2 stream XYZ and colour multiple stream fixed function shader

```
hVS = CreateVertexShader(pDec, NULL,...);
gVB_Pos = CreateVertexBuffer(FVFXYZ);
gVB_Col = CreateVertexBuffer(FVFCOLOR);
SetVertexShader(hVS);
SetStreamSource(0, gVB_Pos, sizeof(XYZ));
SetStreamSource(1, gVB_Col, sizeof(COLOR));
DrawPrimitive();
```

NVIDIA.

Scene Presentation

- New device function `Present()`
  - Subsumes DX7 `Flip()` and `Blit()`
  - Consistent for full-screen and windowed
  - Simplifies renderloop
  - No more windowed app lag due to excessive blit buffering
- Swap chain now a property of device
  - Number of buffers specified in device creation
  - Always at least one chain
    - Additional ones can be created if needed
  - Front buffer now part of device

Front Buffer

- No more concept of ‘Primary Surface’
  - Front buffer not directly accessible
    - Cannot lock or render to
  - Mouse cursor API available
    - Hardware if available
  - Can get to front buffer via GetFrontBuffer()
    - Guaranteed to be slow
    - Only really useful for debugging

Depth/Stencil Surfaces

- Z buffer now set into the device
  - AutoDepthStencil
    - Set in presentation params – created, destroyed and resized automatically by the device
  - Explicit DepthStencil creation
    - Created by app, and attached to the device
      - Use `SetRenderTarget()`
    - Useful when switching render targets halfway through a scene
    - No auto resize on mode or window size changes
    - No auto destruction

`Reset()` (again)

- As well as responding to lost devices, `Reset()` can be used to
  - Resize the buffers
  - Change buffer format
  - Swap between full-screen and windowed
  - Switch stereo on and off
- Calling `Reset()` causes all unmanaged video/agp memory resources, and all state information, to be lost.
  - So similar (if not identical) code path to recovery from device loss

NVIDIA

Summary

- DX8 initialisation is far easier than DX7
- Device creation and loss management improved
- New Index Buffer resource gives VB style management for indices
- Vertex streams allows vertex components in separate VBs
- Vertex shaders will give unprecedented programmability in the transform pipeline
- Scene presentation simplified

NVIDIA