EG 2004

Tutorial 5: Programming Graphics Hardware

Controlling the GPU from the CPU: The 3D API

Cyril Zeller

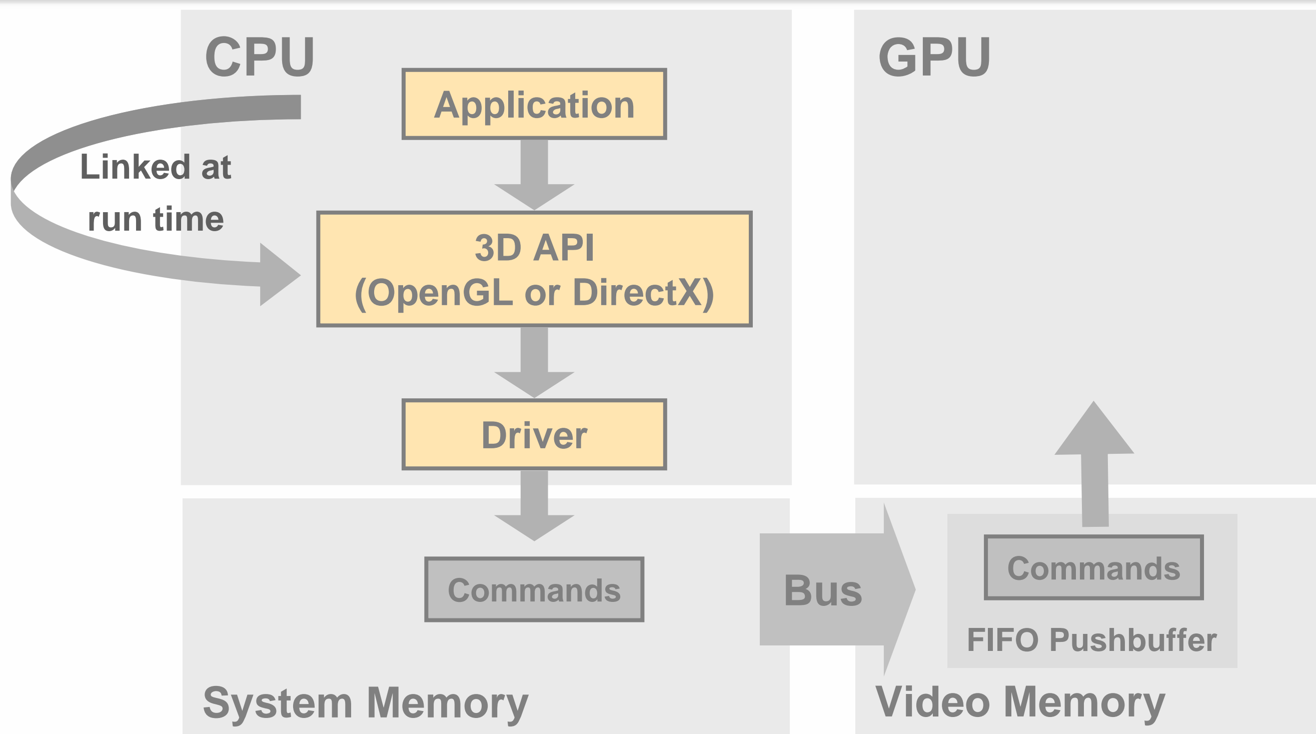
NVIDIA®

Overview

- Software architecture
- Double buffering
- Basic skeleton of a graphics application
  - Initialization
  - Rendering loop
- Specific topics:
  - DirectX's effect framework
  - Accessing resources from the CPU
  - Occlusion query
  - Scene management

NVIDIA.

CPU Side of a Graphics Application



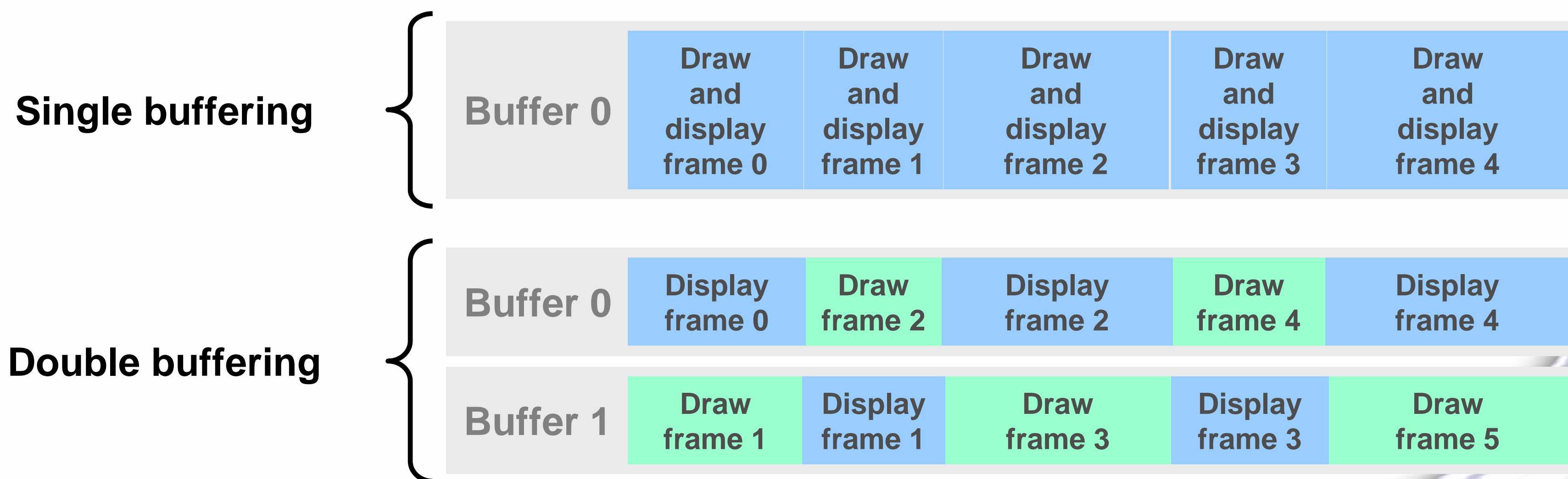
- **The application is programmed to the 3D API and links with the driver at run time**
- **With DirectX, there is an intermediate software layer between the application and the driver called the DirectX runtime**

OpenGL vs. DirectX

	OpenGL	DirectX
Ownership:	OpenGL Architectural Review Board	Microsoft Corporation
Platforms:	Windows, Linux, Unix, MacOS	Windows
Native language:	C	C++
Most popular fields:	Computer aided design, scientific visualization, movie industry, academic world...	Game industry
API specification:	OpenGL specification	DirectX documentation
API evolution:	Mostly through extensions to the core API	Mostly through complete revisions of the core API every year or so
API functionalities:	Mostly equivalent	

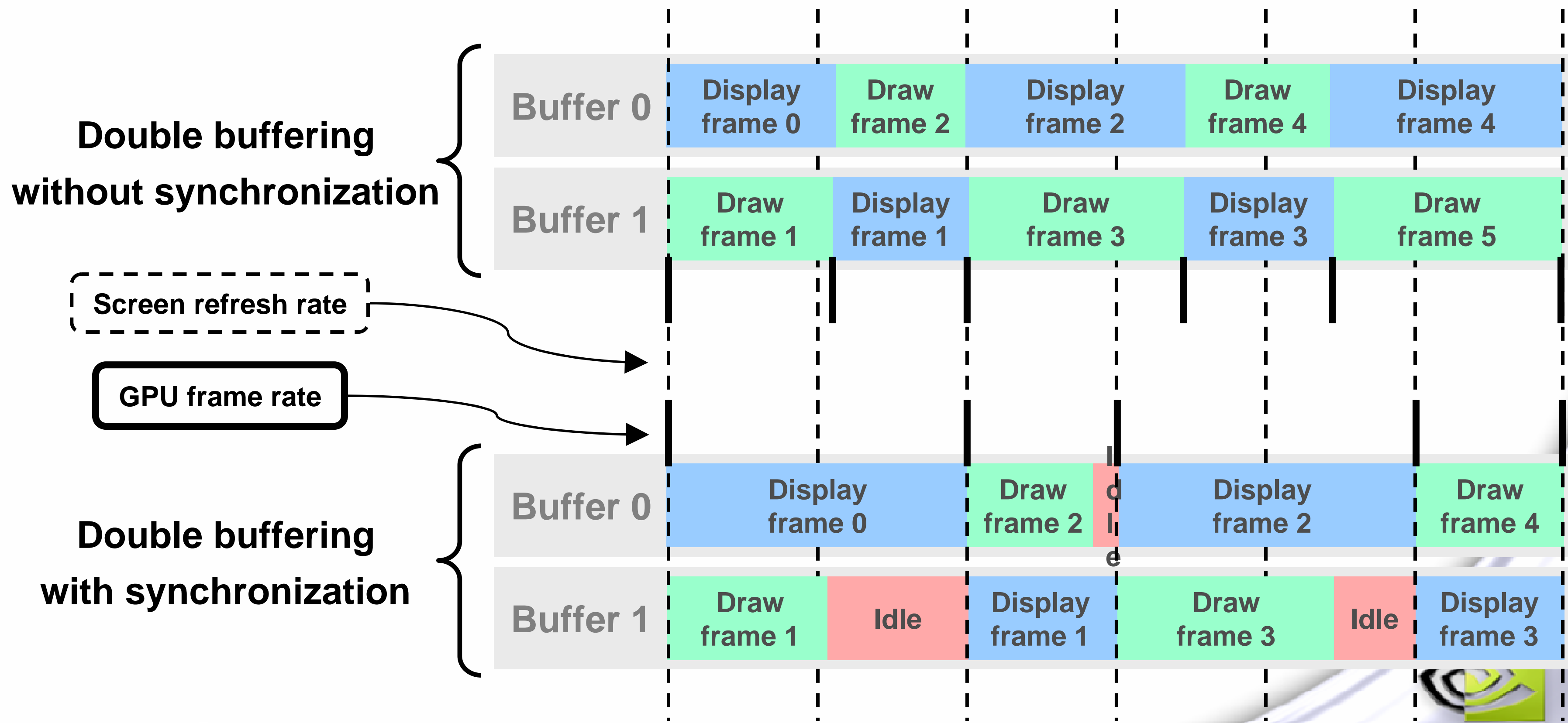
Double Buffering

- **Problem:** Drawing the scene to the buffer that is shown on screen (front buffer) produces tearing
- **Solution:** Use an additional off-screen buffer (back buffer) to draw the scene and swap buffers when done



Double Buffering with VSync

- **Problem:** Buffer swapping may cause tearing too
- **Solution:** Synchronize swapping with refresh rate



Basic Skeleton

Initialization

- Initialize API
- Check hardware capabilities
- Create resources

Rendering loop

- For every frame:
    - Draw to back buffer
    - Swap back buffer and front buffer

Initialization: Initialize API

- Create a window
- Create a device (DirectX) or rendering context (OpenGL) that defines:
  - Back buffer (dimension, pixel format, multisampling type, ...)
  - Front and back buffer swapping method
  - Window or full-screen mode
  - Etc.

Initialization: Initialize API in DirectX

```
// Create a window
HWND hWnd = CreateWindow(...);

// Create a Direct3D object
LPDIRECT3D9 pD3D = Direct3DCreate9(D3D_SDK_VERSION);

// Specify desired device parameters
D3DPRESENT_PARAMETERS d3dpp;
d3dpp.BackBufferFormat = D3DFMT_A8R8G8B8;
d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
d3dpp.Windowed = FALSE;
...

// Create a device
LPDIRECT3DDEVICE9 pDevice;
pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                   D3DCREATE_HARDWARE_VERTEXPROCESSING, &d3dpp, &pDevice);
```

Initialization: Initialize API in OpenGL (Windows)

```
// Create a window
HWND hWnd = CreateWindow(...);

// Specify desired device parameters
PIXELFORMATDESCRIPTOR pfd;
pfd.nSize        = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion     = 1;
pfd.dwFlags      = PFD_DRAW_TO_WINDOW | PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = 32;
pfd.cDepthBits = 32;
...

// Create a rendering context
HDC deviceContext = GetDC(hWnd);
unsigned int pixelFormat = ChoosePixelFormat(deviceContext, &pfd);
SetPixelFormat(deviceContext, pixelFormat, &pfd);
HGLRC renderingContext = wglCreateContext(deviceContext);
```

Initialization: Initialize API in Practice

- DirectX:
  - Start with one of the sample that comes with the SDK in "`C:\Program Files\Microsoft DirectX 9.0 SDK (Summer 2004)\Samples\C++\Direct3D`"
- OpenGL:
  - Use GLUT (OpenGL Utility Toolkit):

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
glutInitWindowSize(512, 512);
glutCreateWindow("Simple application");
...
```

  - Simple and cross-platform!

Initialization: Check Hardware Capabilities

- DirectX:
  - Check caps bits:

```
D3DCAPS9 d3dcaps;
pDevice->GetDeviceCaps(&d3dcaps);
...
```

- OpenGL:
  - Check available extensions:

```
const char* extensions = (const char*)glGetString(GL_EXTENSIONS);
...
```

OpenGL Extension Mechanism

- Why use extensions?
  - To expose new hardware features as soon as possible and let early adopters try them before incorporating them into multi-vendor extensions and then into the core API
- Life of an extension:
  - GL_NVX_foo – eXperimental
  - GL_NV_foo – vendor specific (NV = NVIDIA)
  - GL_EXT_foo – multi-vendor
  - GL_ARB_foo
  - Core OpenGL

Initialization: Create Resources

- Render targets
- Vertex and pixel shaders
- Textures
- Index and vertex buffers

Initialization: Render Targets

- A render target is a piece of the video memory that can serve as color or depth buffer
- Render to texture: Render targets are typically used to compute intermediate images subsequently read as textures to contribute to the final image in the back buffer

Initialization: Render Targets in DirectX

```
// Create a color buffer for the render target
IDirect3DTexture9* pTexture;
pDevice->CreateTexture(..., D3DUSAGE_RENDERTARGET, ..., &pTexture, ...);

// Optionally create a z-buffer for the render target
IDirect3DSurface9* pDepthStencilBuffer;
pDevice->CreateDepthStencilSurface(..., &pDepthStencilBuffer, ...);
```

Initialization: Render Targets in OpenGL

- Pixel buffers or pbuffers extensions:
  WGL_ARB_pbuffer (Windows),
  GLX_SGIX_pbuffer (Linux)

```
// Create render target
HPBUFFERARB pbuffer = wglCreatePbufferARB(...)
HDC pbufferDeviceContext = wglGetPbufferDCARB(pbuffer);
```

Initialization: Shaders in DirectX

- Compiler is decoupled from runtime

```
// Compile
char* pSource = ...;
ID3DXBuffer* pCode;
ID3DXConstantTable* pConstantTable;
D3DXCompileShader(pSource, ..., &pCode, ..., &pConstantTable);
```

```
// Create vertex shader or...
IDirect3DVertexShader9* pShader;
pDevice->CreateVertexShader(pCode->GetBufferPointer(), &pShader);

// ... pixel shader
IDirect3DPixelShader9* pShader;
pDevice->CreatePixelShader(pCode->GetBufferPointer(), &pShader);
```

Initialization: Shaders in OpenGL

- Extensions: GL_ARB_shader_objects, GL_ARB_vertex_shader, GL_ARB_fragment_shader

```
// Create program object
GLhandleARB program = glCreateProgramObjectARB();

// Create and compile a shader and add it to the program
GLhandleARB shader = glCreateShaderObjectARB(GL_VERTEX/PIXEL_SHADER_ARB);
GLcharARB* source = ...;
glShaderSourceARB(shader, ..., &source, ...); // Specify source code
glCompileShaderARB(shader); // Compile
glAttachObjectARB(program, shader); // Add

// Optionally add more shaders to the program and link the program
glLinkProgramARB(program);
```

Initialization: Textures in DirectX

```
// Allocate
IDirect3DTexture9* pTexture;
pDevice->CreateTexture(..., &pTexture, ...);

// Load data into texture
D3DLOCKED_RECT rect;
pTexture->LockRect(..., &rect, ...);
// Write at address (void*)rect.pBits
pTexture->UnlockRect(...);
```

Initialization: Textures in OpenGL

```
// Generate ID
GLuint textureID;
glGenTextures(1, &textureID);

// Bind texture
glBindTexture(GL_TEXTURE_2D, textureID);

// Load data into texture
unsigned char* data = ...;
glTexImage2D(... , data);
```

Initialization: Index and Vertex Buffers in DirectX

```
// Allocate vertex buffer or...
IDirect3DVertexBuffer9* pBuffer;
pDevice->CreateVertexBuffer(..., &pBuffer, ...);
// ...index buffer
IDirect3DIndexBuffer9* pBuffer;
pDevice->CreateIndexBuffer(..., &pBuffer, ...);

// Load data into buffer
void* pData = ...;
pBuffer->Lock(..., &pData, ...);
// Write at address pData
pBuffer->Unlock(...);
```

```
D3DVERTEXELEMENT9 vertexElements[] = {
{ 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
{ 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   0 },
... };
pDevice->CreateVertexDeclaration(vertexElements, &pVertexDeclaration);
```

Initialization:
Index and Vertex Buffers in OpenGL

- **Extensions: GL_ARB_vertex_buffer_object**

```
// Generate ID
GLuint bufferID;
glGenBuffersARB(1, &bufferID);

// Load data into...
void* pData = ...;
// ... index buffer or...
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, bufferID);
glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB, ..., pData, ...);
// ... vertex buffer
glBindBufferARB(GL_ARRAY_BUFFER_ARB, bufferID);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, ..., pData, ...);
```

Rendering Loop: Draw to Back Buffer

- DirectX only: `pDevice->BeginScene();`
- Clear frame buffer
- For each mesh of the scene:
  - For each rendering pass:
    - Set index and vertex buffers
    - Set vertex and pixel shaders and their parameters
    - Set texture sampling parameters
    - Set render states
    - Set render target
    - Draw
- Add postprocessing effects
- DirectX only: `pDevice->EndScene();`

Rendering Loop: Clear Frame Buffer

- DirectX:

```
pDevice->Clear(..., D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, ...);
```

- OpenGL:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```


NVIDIA.

Rendering Loop: Index and Vertex Buffers in DirectX

```
// Set vertex declaration
pDevice->SetVertexDeclaration(pVertexDeclaration);

// Set vertex buffer
pDevice->SetStreamSource(0, pVertexBuffer0, ...);
pDevice->SetStreamSource(1, pVertexBuffer1, ...);
...

// Set index buffer
pDevice->SetIndices(pIndexBuffer);
```


NVIDIA.

Rendering Loop:
Index and Vertex Buffers in OpenGL

```
// Bind vertex and index buffers
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertexBufferID);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, indexBufferID);

// Specify vertex data sizes and offsets into the vertex buffer
glVertexPointer(3, GL_FLOAT, vertexStride, vertexOffset);
glNormalPointer(GL_FLOAT, normalStride, normalOffset);
...
```

Rendering Loop: Shaders in DirectX

```
// Set vertex shader parameters
D3DXHANDLE hParam = pConstantTable->GetConstantByName(0, "WorldViewProjection");
pConstantTable->SetMatrix(pDevice, hParam, pWorldViewProjection);
... // SetFloat(), SetVector(), SetTexture(), SetFloatArray(), etc.

// Set vertex shader
pDevice->SetVerterShader(pShader);

// Set pixel shader parameters
D3DXHANDLE hParam = pConstantTable->GetConstantByName(0, "TextureSampler");
D3DXCONSTANT_DESC description;
pConstantTable->GetConstantDesc(hParam, &description, ...);
pDevice->SetTexture(description.TextureIndex, texture);
... // SetFloat(), SetVector(), , SetMatrix(), SetFloatArray(), etc.

// Set pixel shader
pDevice->SetPixelShader(pShader);
```

Rendering Loop: Shader Built-In Parameters in OpenGL

```
// Set projection and modelview matrices
// (referred to as gl_ModelViewProjectionMatrix in the GLSL program)
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(...); glRotatef(...); glScalef(...);
glLoadMatrixf(...);

// Set other built-in uniform parameters...
```

Rendering Loop: Shaders in OpenGL

```
// Set vertex shader parameters
GLint cameraPos = glGetUniformLocationARB(vertexProgram, "CameraPosition");
glUniform4fARB(cameraPos, x, y, z);
... // glUniform[1,2,3,4][i,f][v]ARB(), glUniformMatrix[2,3,4]fvARB(), etc.

// Set vertex program
glUseProgramObjectARB(vertexProgram);

// Set fragment shader parameters
GLint texSampler = glGetUniformLocationARB(fragmentProgram, "TextureSampler");
glUniform1iARB(texSampler, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureID);
... // glUniform[1,2,3,4][i,f][v]ARB(), glUniformMatrix[2,3,4]fvARB(), etc.

// Set fragment program
glUseProgramObjectARB(fragmentProgram);
```

Rendering Loop: Texture Sampling Parameters in DirectX

```
// Set sampling parameters for sampler N
pDevice->SetSamplerState(N, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
pDevice->SetSamplerState(N, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
pDevice->SetSamplerState(N, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
pDevice->SetSamplerState(N, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
```

Rendering Loop: Texture Sampling Parameters in OpenGL

```
// Bind texture
glBindTexture(GL_TEXTURE_2D, textureID);

// Set sampling parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Rendering Loop: Render States in DirectX

```
// Enable alpha blending and set the blending mode
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

// Enable z-buffering
pDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
pDevice->SetRenderState(D3DRS_ZWRITEENABLE, TRUE);

// Set culling state
pDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);

...
```

Rendering Loop: Render States in OpenGL

```
// Enable alpha blending and set the blending mode
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

// Enable z-buffering
glEnable(GL_DEPTH_TEST);
glDepthMask(GL_TRUE);

// Set culling state
glEnable(GL_CULL_FACE);


...
```

Rendering Loop: Render Target in DirectX

```
// Set render target
pDevice->SetRenderTarget(0, pTexture);
pDevice->SetDepthStencilSurface(pDepthStencilBuffer);

// Clear render target
pDevice->Clear(..., D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, ...);
```


NVIDIA.

Rendering Loop: Render Target in OpenGL

```
// Set a rendering context that is compatible with the pbuffer
HGLRC renderingContext = ...;
                    // = wglCreateContext(pbufferDeviceContext);
wglMakeCurrent(pbufferDeviceContext, renderingContext);

// Clear render target
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Rendering Loop: Draw

- **DirectX:**

```
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, ...);
```

- **OpenGL: GL_EXT_draw_range_elements**

```
glDrawRangeElements(GL_TRIANGLES, ...);
```

  - **Immediate mode:**

```
glBegin(GL_TRIANGLES);
   // First triangle
   glColor3f(0.0f, 1.0f, 0.0f);
   glVertex3f(0.0f, 130.0f, 0.0f);
   glColor3f(0.0f, 0.0f, 1.0f);
   glVertex3f(0.0f, 0.0f, 0.0f);
   glColor3f(1.0f, 0.0f, 0.0f);
   glVertex3f(130.0f, 0.0f, 0.0f);
   // Remaining triangles
   ...
glEnd();
```

Rendering Loop: Postprocess

- 2D-postprocessing is usually performed by rendering a full-screen quad (or smaller quads) and fetching from a texture the image computed so far
- Typical postprocessing: Fog, tone mapping, depth of field, motion blur, glow effects, etc.
- Other applications:
  - Video processing
  - Deferred shading: Lighting itself is handled as a 2D-postprocess:
    - For each mesh:
      - Render positions, normals, etc. to multiple render targets
    - For each light:
      - Render light volume and compute light contribution to the scene

Rendering Loop: Swap Buffers

- **DirectX:**

```
pDevice->Present(...);
```

- **OpenGL:**

```
glutSwapBuffers();
```

DirectX's Effect Framework: File Format

```
// Shaders and global variables
void MyVertexShader0(uniform float4x4 worldViewProjection, ...) { ... }
void MyPixelShader0(uniform sampler2D textureSampler...) { ... }
float4x4 WorldViewProjection; sampler2D Texture; sampler2D TextureSampler = ...;
...

// Techniques
technique Technique0 {
  pass Pass0 {
    AlphaBlendEnable = True; SrcBlend = SrcAlpha; DestBlend = InvSrcAlpha;
    ZEnable = True; ZWriteEnable = True;
    CullMode = CW;
    VertexShader = compile vs_3_0 MyVertexShader0(WorldViewProjection, ...);
    PixelShader = compile ps_3_0 MyPixelShader0(TextureSampler, ...);
  }
  pass Pass1 { ... }
  ...
}
technique Technique1 { ... }
...
```

DirectX's Effect Framework: Initialization

- FX files can be precompiled:
  - Either using the `fxc` executable
  - Or by the application itself using `D3DXCreateEffectCompiler()`
- Then, for each FX file an effect is created:

```
ID3DXEffect* pEffect;
void* pCode = ...; // Can be either source code or object code
D3DXCreateEffect(pDevice, pCode, ..., &pEffect, ...);
```

DirectX's Effect Framework: Rendering Loop

```
// Set uniform parameters
pEffect->SetMatrix("WorldViewProjection", pWorldViewProjection);
pEffect->SetTexture("Texture", pTexture);
...

// Apply one technique contained in the effect
pEffect->SetTechnique("Technique1");
unsigned int cPasses;
pEffect->Begin(&cPasses, 0);
for (unsigned int iPass = 0; iPass < cPasses; iPass++) {
   pEffect->BeginPass(iPass);
   DrawMesh();
   pEffect->EndPass();
}
pEffect->End();
```

Accessing Resources from the CPU

- Usually, resources are accessed by the CPU only when created during initialization
- Sometimes, resources need to be accessed during the rendering loop requiring CPU/GPU synchronization because of concurrent access
- To minimize synchronization cost, inform the driver of your intention through the API usage flags:
  - At creation time:
    - DirectX: D3DUSAGE_DYNAMIC, D3DUSAGE_WRITEONLY
    - OpenGL: STATIC_DRAW_ARB, STATIC_READ_ARB, STATIC_COPY_ARB, DYNAMIC_DRAW_ARB, DYNAMIC_READ_ARB, etc.
  - At access time:
    - DirectX: D3DLOCK_DISCARD, D3DLOCK_READONLY, D3DLOCK_NOOVERWRITE
    - OpenGL: READ_ONLY_ARB, WRITE_ONLY_ARB, READ_WRITE_ARB

Accessing Resources: DirectX

```
// Appending data to the end of a buffer
void* pData = ...;
pBuffer->Lock(..., &pData, D3DLOCK_NOOVERWRITE);
// Write at address pData
pBuffer->Unlock(...);
```

```
// Updating the entire content of a texture
D3DLOCKED_RECT rect;
pTexture->LockRect(..., &rect, D3DLOCK_DISCARD);
// Write at address (void*)rect.pBits
pTexture->UnlockRect(...);
```

Accessing Resources: OpenGL

- Extensions: GL_ARB_vertex_buffer_object, GL_EXT_pixel_buffer_object

```
// Updating the entire content of a buffer
glBindBufferARB(GL_ARRAY_BUFFER_ARB, bufferID);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, ..., 0, GL_DYNAMIC_DRAW_ARB);
void* pData = glMapBufferARB(GL_ARRAY_BUFFER_ARB, GL_WRITE_ONLY);
// Write at address pData
glUnmapBufferARB(GL_ARRAY_BUFFER_ARB);
```

```
// Updating the entire content of a texture
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_EXT, bufferID);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_EXT, ..., 0, GL_DYNAMIC_DRAW_ARB);
void* pData = glMapBufferARB(GL_PIXEL_UNPACK_BUFFER_EXT, GL_WRITE_ONLY);
// Write at address pData
glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_EXT);
glTexSubImage2D(GL_TEXTURE_2D, ..., 0);
```

Reading Back Buffer or Render Target

- Must copy to some resource first:
  - DirectX: To a texture

```
IDirect3DSurface9* pSourceSurface = ...;
                                    // = GetBackBuffer() or GetSurfaceLevel()
IDirect3DSurface9* pDestSurface = ...;
                                  // = GetSurfaceLevel()
pDevice->StretchRect(pSourceSurface, 0, pDestSurface, 0, D3DTEXF_LINEAR);
```

  - OpenGL: To a pixel buffer object

```
glBindBuffer(GL_PIXEL_PACK_BUFFER_EXT, destBufferID);
glReadPixels(..., 0);
```

- And then read this resource as previously mentioned

Occlusion Query

- An occlusion query asynchronously returns the number of fragments that pass z-test
- Typical use: In multi-pass rendering, skip subsequent passes if the first one rendered too few pixels
- Usage:
  - Create the query
  - Issue a begin event to start counting
  - Draw something
  - Issue an end event to stop counting
  - Get the result

Occlusion Query: DirectX

```
// Create query
IDirect3DQuery9* pQuery;
pDevice->CreateQuery(D3DQUERYTYPE_OCCLUSION, &pQuery);
...

// Count
pQuery->Issue(D3DISSUE_BEGIN);
Draw(...);
pQuery->Issue(D3DISSUE_END);
...

// Get result
int fragmentsDrawn;
while (pQuery->GetData(&fragmentsDrawn, ...) == S_FALSE);
```

Occlusion Query: OpenGL

- Extension: GL_ARB_occlusion_query

```
// Generate ID
GLuint queryID;
glGenQueriesARB(1, &queryID);
...

// Count
glBeginQueryARB(GL_SAMPLES_PASSED_ARB, queryID);
Draw(...);
glEndQueryARB(GL_SAMPLES_PASSED_ARB);
...

// Get result
int fragmentsDrawn;
glGetQueryObjectuivARB(queryID, GL_QUERY_RESULT_ARB, &fragmentsDrawn);
```

Scene Management

- For each frame of the rendering loop the application should:
  - Cull invisible triangles
    - Per group of triangles, not per triangle!
    - Scene graph makes it efficient
  - Sort the triangle batches
    - To minimize state changes between draw calls
    - To maximize the effectiveness of the z–buffer algorithm

Getting Started!

- Grab the latest driver from the website of your card's manufacturer
- DirectX:
  - Install the latest DirectX 9.0c SDK that comes with a full documentation
  - Start with one of the sample that comes with the SDK in "`C:\Program Files\Microsoft DirectX 9.0 SDK (Summer 2004)\Samples\C++\Direct3D`"
- OpenGL:
  - Download GLUT from www.opengl.org
  - Start with one of the sample provided at www.opengl.org
  - Full specification is at www.opengl.org

Questions

- **Support e-mail:**
  - devrelfeedback@nvidia.com [Technical Questions]
  - sdkfeedback@nvidia.com [Tools Questions]