

# OpenGL (and Friends) in the Future A Notional View

Dave Shreiner  
ARM, Inc.



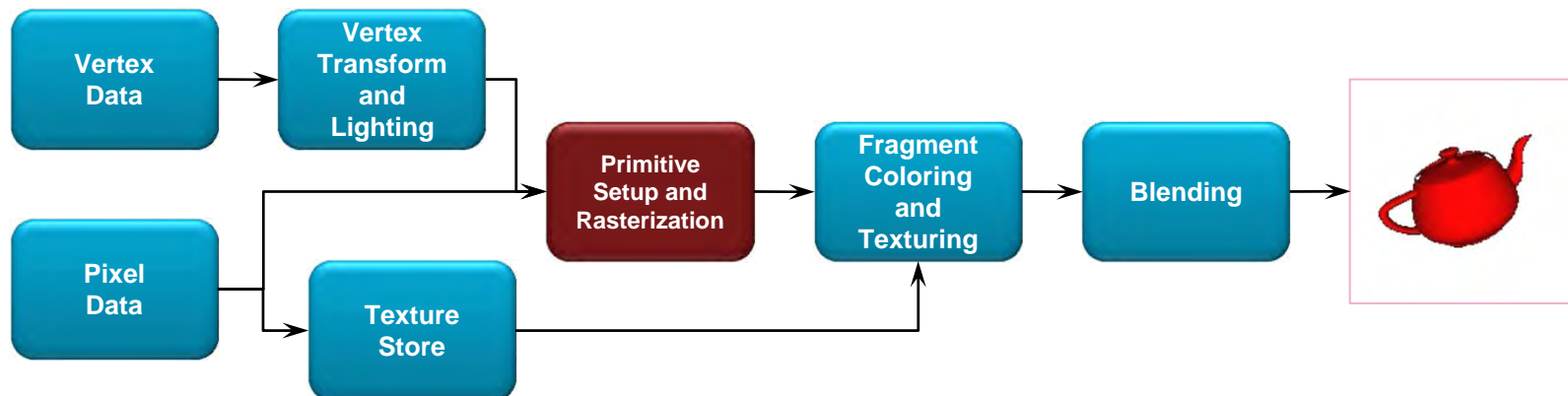
# First: A Retrospective

The Evolution of the OpenGL Pipeline



# In the Beginning ...

- OpenGL 1.0 was released on July 1<sup>st</sup>, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



- The pipeline evolved, but remained fixed-function through OpenGL versions 1.1 through 2.0 (released Sept. 7<sup>th</sup>, 2004)

# Fixed-Function Application Interface

---

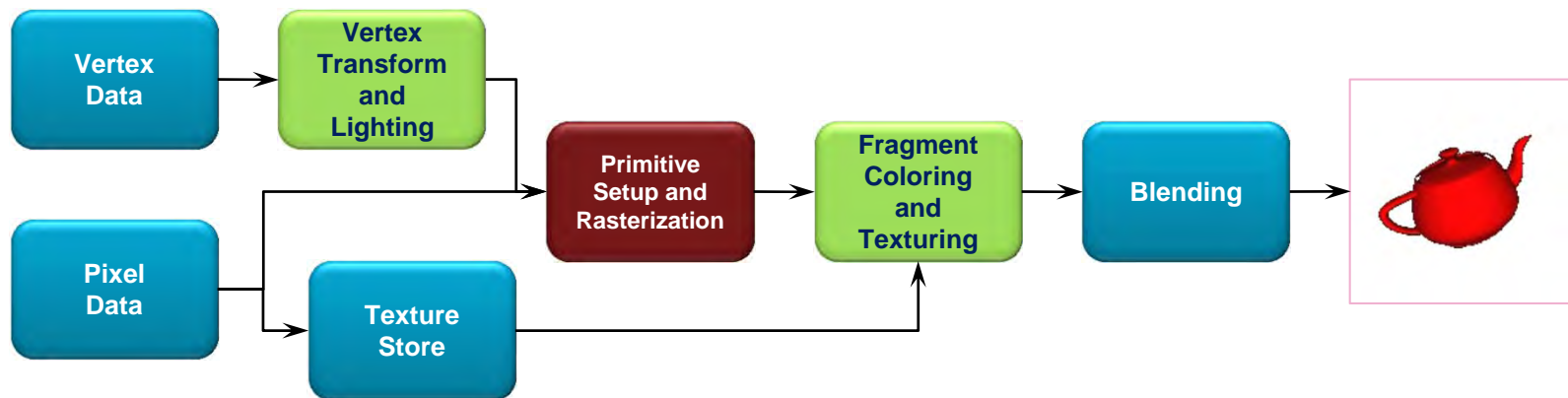
```
glColor3f( ... );
glBegin( GL_TRIANGLES );
    glVertex3f( ... );
    glVertex3f( ... );
    glVertex3f( ... );
glEnd();

GLfloat data[] = { ... };
glColor3f( ... );
glVertexPointer( 3, GL_FLOAT,
                0, data );
glDrawArrays( GL_TRIANGLES, 0, 3 );
```

- Everything the API was capable of is accessed through function calls
- Lots of fine-grained memory writes
  - when this API was developed, most computer systems directly mapped device registers and poked values into them
  - API made sense for systems of that time

# The Start of the Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



- The pipeline remained the same until OpenGL 3.1 (released March 24<sup>th</sup>, 2009)

# An Evolutionary Change

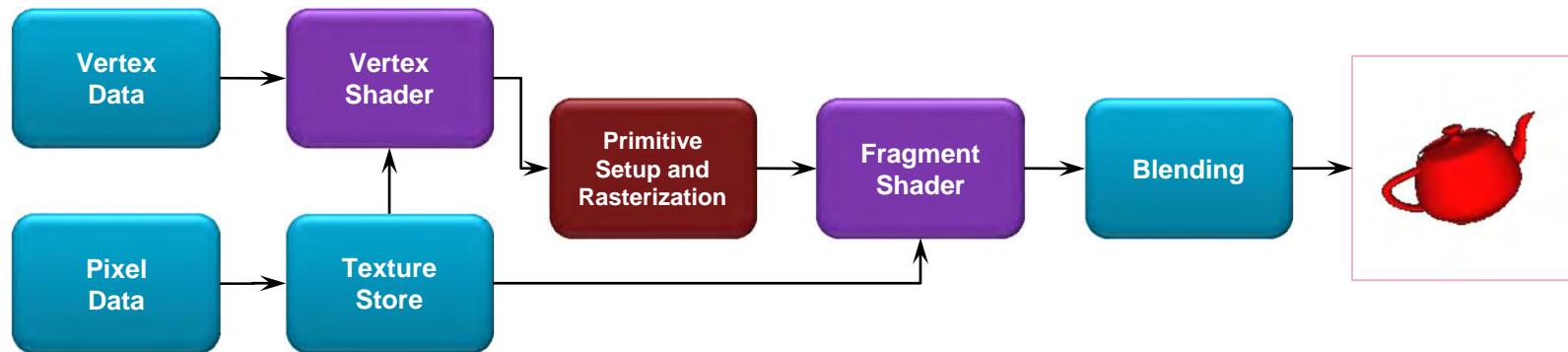
---

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- Introduced a change in how OpenGL contexts are used
  - an OpenGL *context* is the driver data structure that stores OpenGL state information (e.g., textures, shaders, etc.)
  - two types of contexts became available

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

# The Exclusively Programmable Pipeline

- OpenGL 3.1 removed<sup>(\*)</sup> the fixed-function pipeline
  - programs were required to use only shaders

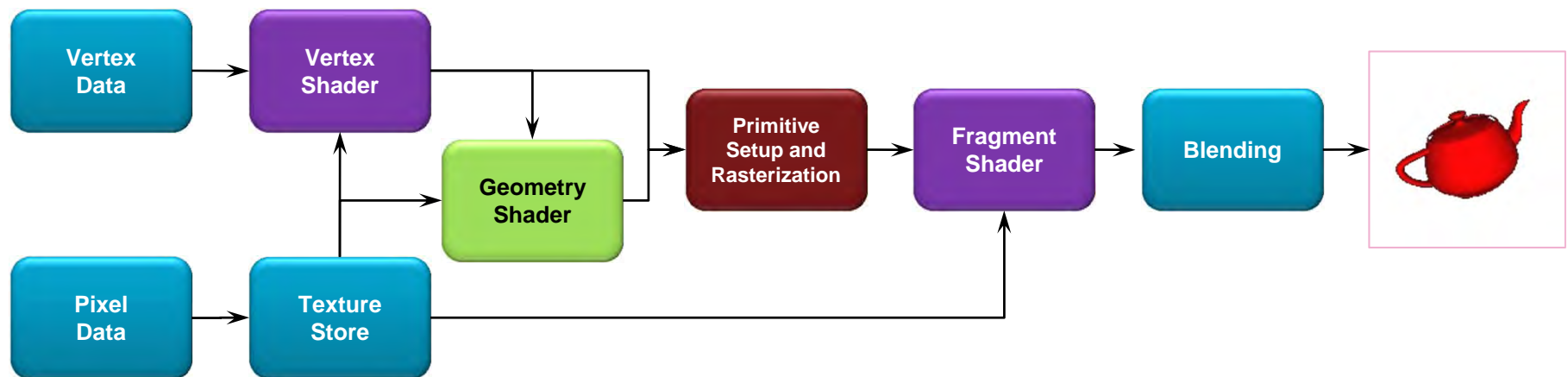


- Additionally, almost all data is *GPU-resident*
  - all vertex data sent using buffer objects

<sup>(\*)</sup> OpenGL 3.1 included an extension – [GL\\_ARB\\_compatibility](#) – which re-enabled all removed functionality

# More Programmability

- OpenGL 3.2 (released August 3<sup>rd</sup>, 2009) added an additional shading stage – *geometry shaders*



# More Evolution – Context Profiles

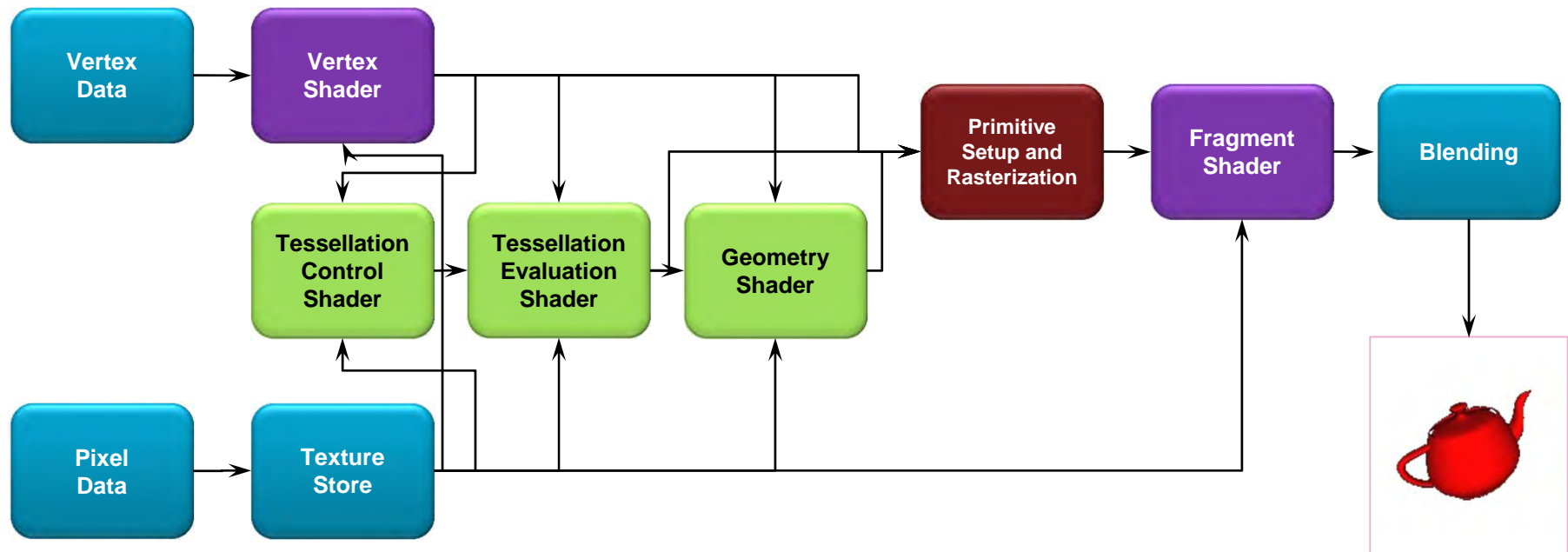
---

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
    - it's like [GL\\_ARB\\_compatibility](#), only not insane 😊
  - currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	<b>Not supported</b>

# The Latest Pipeline

- OpenGL 4.0 (released March 11<sup>th</sup>, 2010) added additional shading stages – *tessellation-control and tessellation-evaluation shaders*



- OpenGL 4.1 (released July 26<sup>th</sup>, 2010) and 4.2 (released, August 10<sup>th</sup>, 2011) added features, but no new shading stages

# Programmable Shader Interface

---

```
GLchar *vertPgm = "in vec4 vPosition; ...";
GLchar *fragPgm = "...";

GLuint vertShdr =
    glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vertShdr, 1,
    NULL, vertPgm );
glCompileShader( vertShdr );

GLuint fragShdr =
    glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fragShdr, 1,
    NULL, fragPgm );
glCompileShader( fragShdr );

GLuint program = glCreateProgram();
glAttachShader( program, vertShdr );
glAttachShader( program, fragShdr );
glLinkProgram();

GLuint vPos = glGetAttribLocation(
    program, "vPosition" );

GLfloat data[] = { ... };

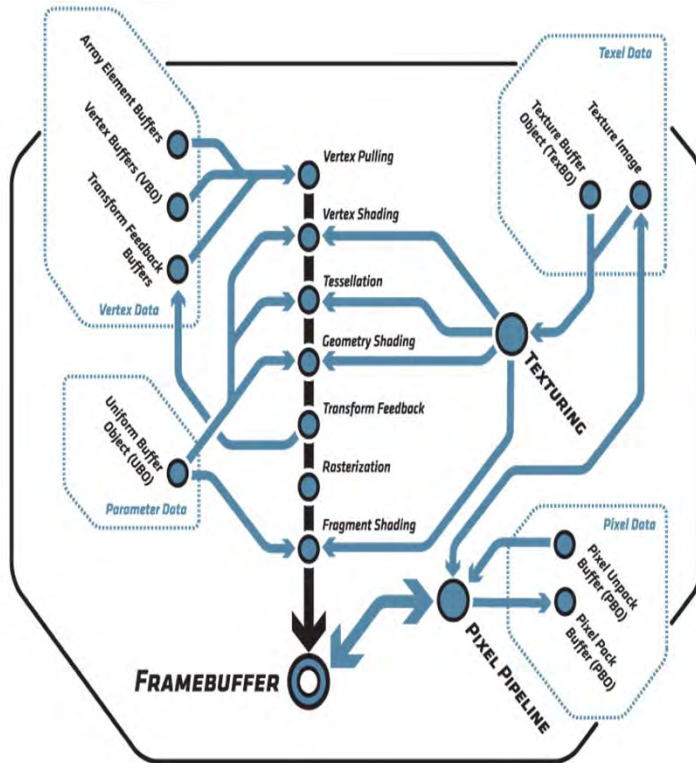
GLuint VAO;
glGenVertexArrays( 1, &VAO );
glBindVertexArray( VAO );

GLuint VBO;
glGenBuffer( 1, &VBO );
glBindBuffer( GL_VERTEX_BUFFER, VBO );
glBufferData( GL_VERTEX_BUFFER,
    sizeof(data), data, GL_STATIC_DRAW );

glVertexAttribPointer( vPos, 3, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray( vPos );

glDrawArrays( GL_TRIANGLES, 0, 3 );
```

# Accelerating OpenGL Innovation



Bringing state-of-the-art functionality to cross-platform graphics



OpenGL 4.1

OpenGL 3.3/4.0

OpenGL 3.2

OpenGL 3.1

OpenGL 3.0

OpenGL 2.0

OpenGL 2.1



2004

2005

2006

2007

2008

2009

2010

2011

DirectX 9.0c

DirectX 10.0

DirectX 10.1

DirectX 11



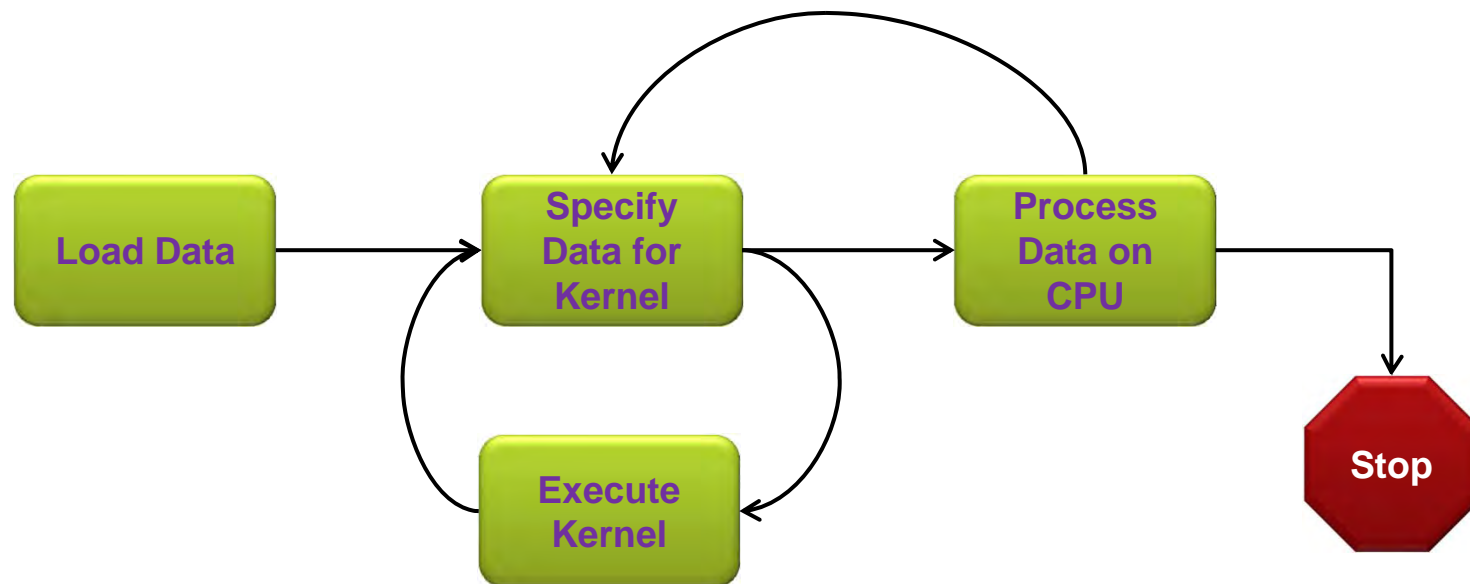
# Trend Check ...

---

- Almost all innovation of the latest versions has been in adding new shader stages or shader capabilities
  - less graphics focused– more compute focused
  - there are still a few stages that are “fixed-function” (e.g., blending), but those may become programmable soon as well
- Clear trend towards:
  1. initialize a chunk of data
  2. process it using a shader
  3. if ( !done) go to 1
- That’s (generally) good news for driver developers
  - most changes confined to shader compiler
- Heading that direction (could) have a logical conclusion ...

# OpenCL

- OpenCL (Compute Language) provides a common framework for heterogeneous computing
  - write one “kernel” (OpenCL vernacular for “shader”), and OpenCL will make it available for each supported compute device in a system



# A View Towards the Future

(from this point, it's likely anything I say will be false in a short time ... or maybe not 😊)



# Moving Data Downwards

---

- System buses are still the bottlenecks in almost all systems
- APIs are trying to limit programmer data interaction
  - move from small-grained API interaction to large data-block mechanisms
- Khronos APIs are trending (if not there already) to handing data to the GPU in chunks
  - OpenGL's buffer objects
    - VBOs, PBOs, TexBOs, UBOs, ...
  - explicit loading/retrieval operations (through API calls)
    - actually, very useful for knowing when data's changed
      - ask anyone who's worked on client-side vertex arrays

# Feature Convergence

---

- OpenGL's acquiring more OpenCL-like features:
  - Random-access reading and writing to images (i.e., buffers)
  - Atomic operations on shader variables
  - Asynchronous thread execution
- OpenCL comes with some graphics features as well:
  - Filtered image sampling
  - Writing to images
- What's still different?
  - Mostly hardware accelerated features:
    - rasterizer
    - blending and depth-buffering hardware
  - But it's possible to implement these in a kernel
    - it's just not as optimal as having hardware

# Impact on Device Drivers

---

- OpenGL and OpenCL are separate APIs
  - likely implemented in separate DSOs
- Data sharing is permitted between the APIs
  - KHR extension providing OpenCL access to OpenGL buffers
  - requires data synchronization
    - both APIs support fence-like facilities for synchronization

# Thanks!

## Questions?

(and maybe even some answers 😊)

