OpenGL pipeline Evolution
and
OpenGL Shading Language (GLSL)

Part 1/3

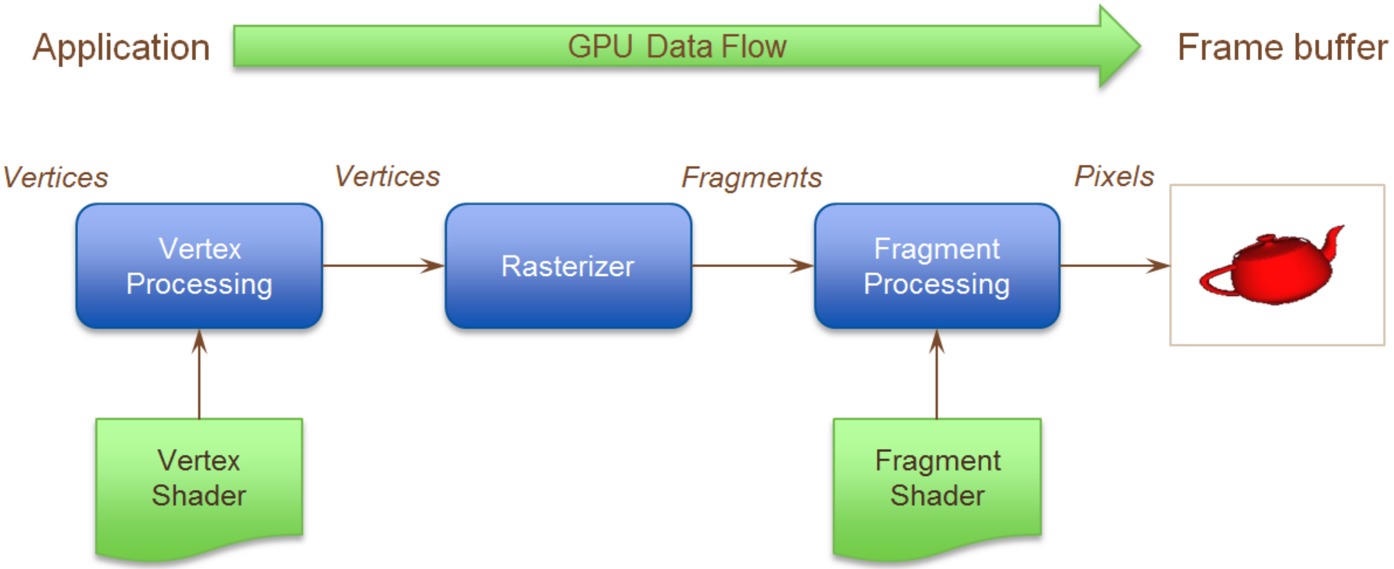
Prateek Shrivastava
CS12S008
shrvstv@cse.iitm.ac.in

INTRODUCTION

OpenGL Shading Language (GLSL)

- **"mini-programs"** written in GLSL are often referred to as **shader programs**, or simply **shaders**.
- GLSL programs don't stand on their own, **they must be a part of a larger program.**
- Shaders can be used for algorithms related to
  - Lighting.
  - Shading (coloring).
  - Tessellation.
  - Generalized computation.
  - Performing animation.

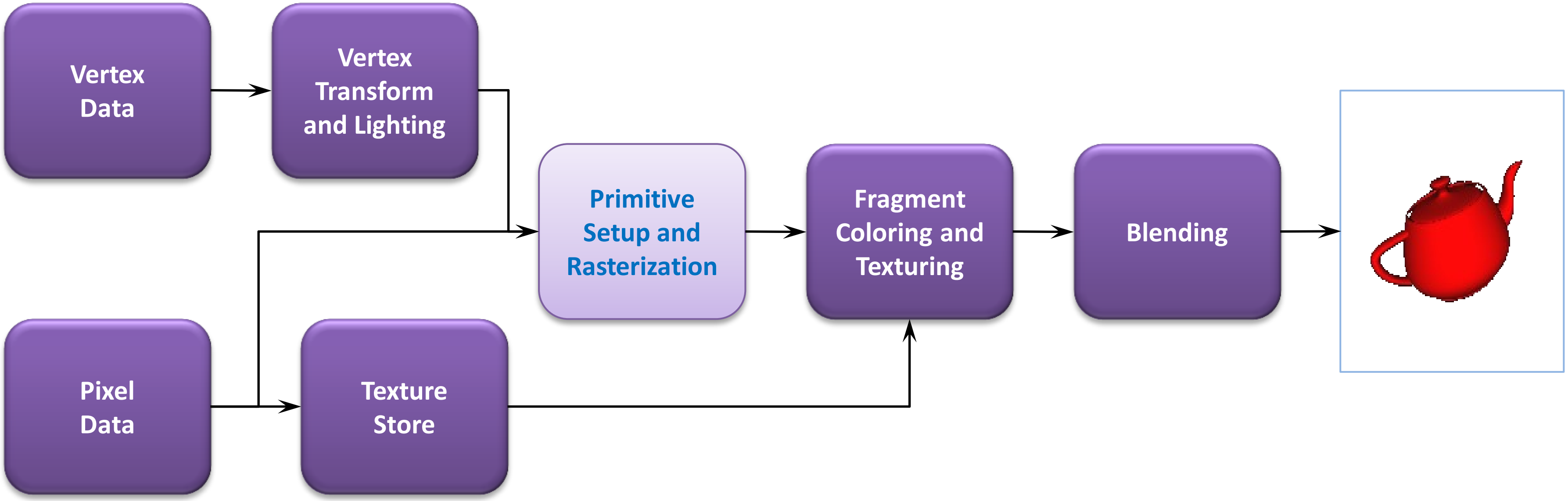
SIMPLIFIED PIPELINE MODEL



- Application will provide ***vertices***, which are collections of data that are composed to form geometric objects, to the OpenGL pipeline. **The *vertex processing* stage** uses a vertex shader to process each vertex, doing any computations necessary to determine where in the frame buffer each piece of geometry should go.
- After all the vertices for a piece of geometry are processed, the ***rasterizer*** determines which pixels in the frame buffer are affected by the geometry, and for each pixel, the ***fragment processing* stage** is employed, where the *fragment shader* runs to determine the final color of the pixel.

Evolution of OpenGL

OpenGL 1.0 pipeline

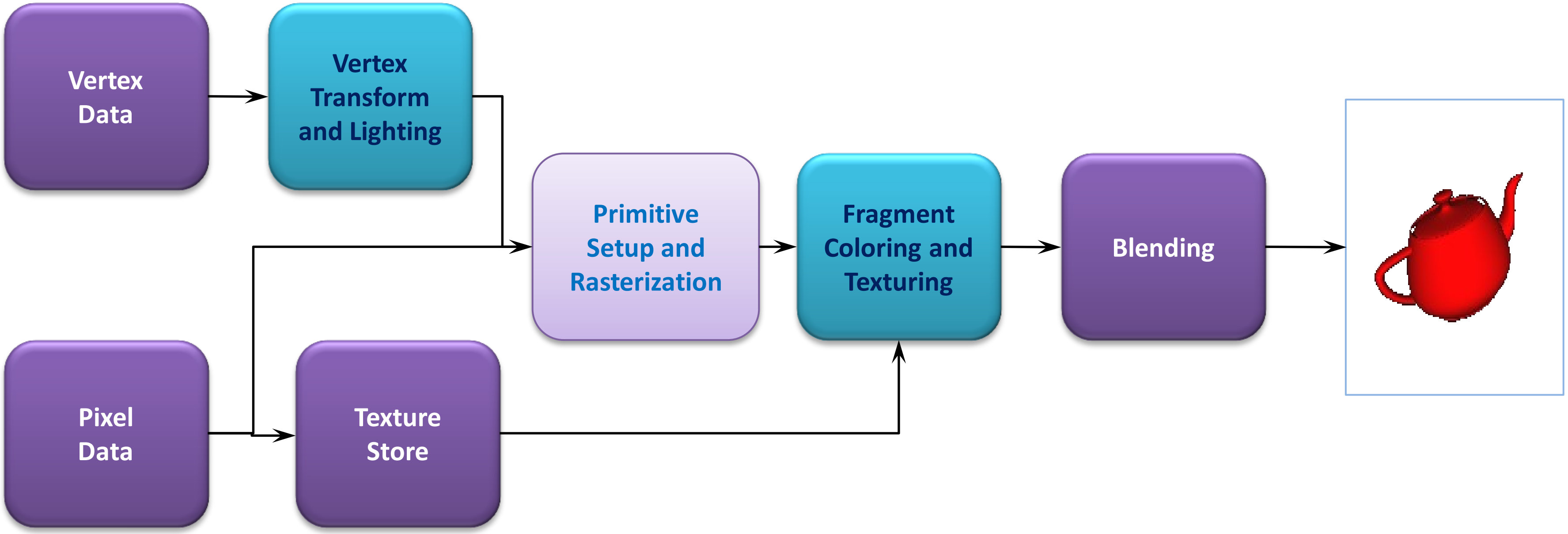

Vertex Data
Vertex Transform and Lighting
Primitive Setup and Rasterization
Fragment Coloring and Texturing
Blending
Pixel Data
Texture Store

OpenGL 1.0 was released on July 1st, 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 (Sept. 2004)

OpenGL 2.0 pipeline


Vertex Data
Vertex Transform and Lighting
Primitive Setup and Rasterization
Fragment Coloring and Texturing
Blending
Pixel Data
Texture Store

OpenGL 2.0 (officially) added programmable shaders

- ***vertex shading*** enabled the application full control over manipulation of the 3D geometry provided by the application.
- ***fragment shading*** provided the application capabilities for *shading* pixels (pixel's color).

However, the fixed-function pipeline was still available

Depreciation Model

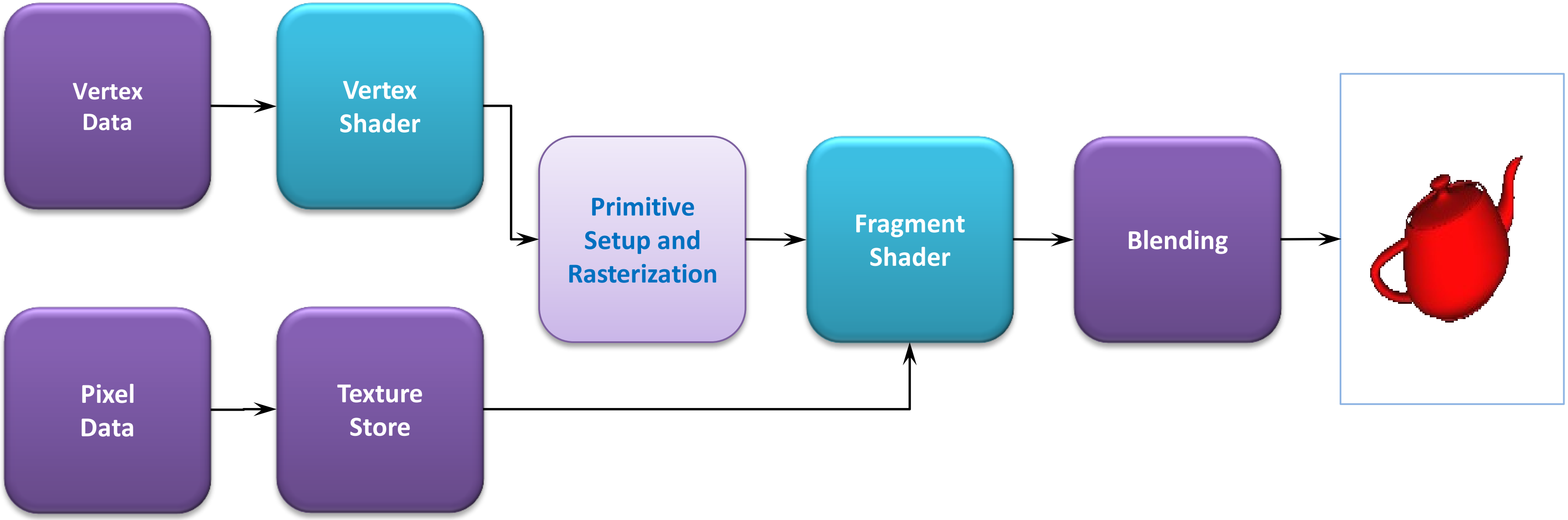
- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- OpenGL uses an opaque data structure called a *context*, which OpenGL uses to store shaders and other data.

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)

What we can’t do ?

- Any use of the **fixed function** vertex or fragment operations; **shaders are mandatory.**
- Use of **glBegin/glEnd** and **Display lists** to define primitives; **vertex arrays and vertex buffers for geometry.**
- Use of quad or polygon primitives; **only triangles.**
- Use of most of the built-in attribute and uniform variables in GLSL; **pass them manually to shaders.**

OpenGL 3.1 pipeline

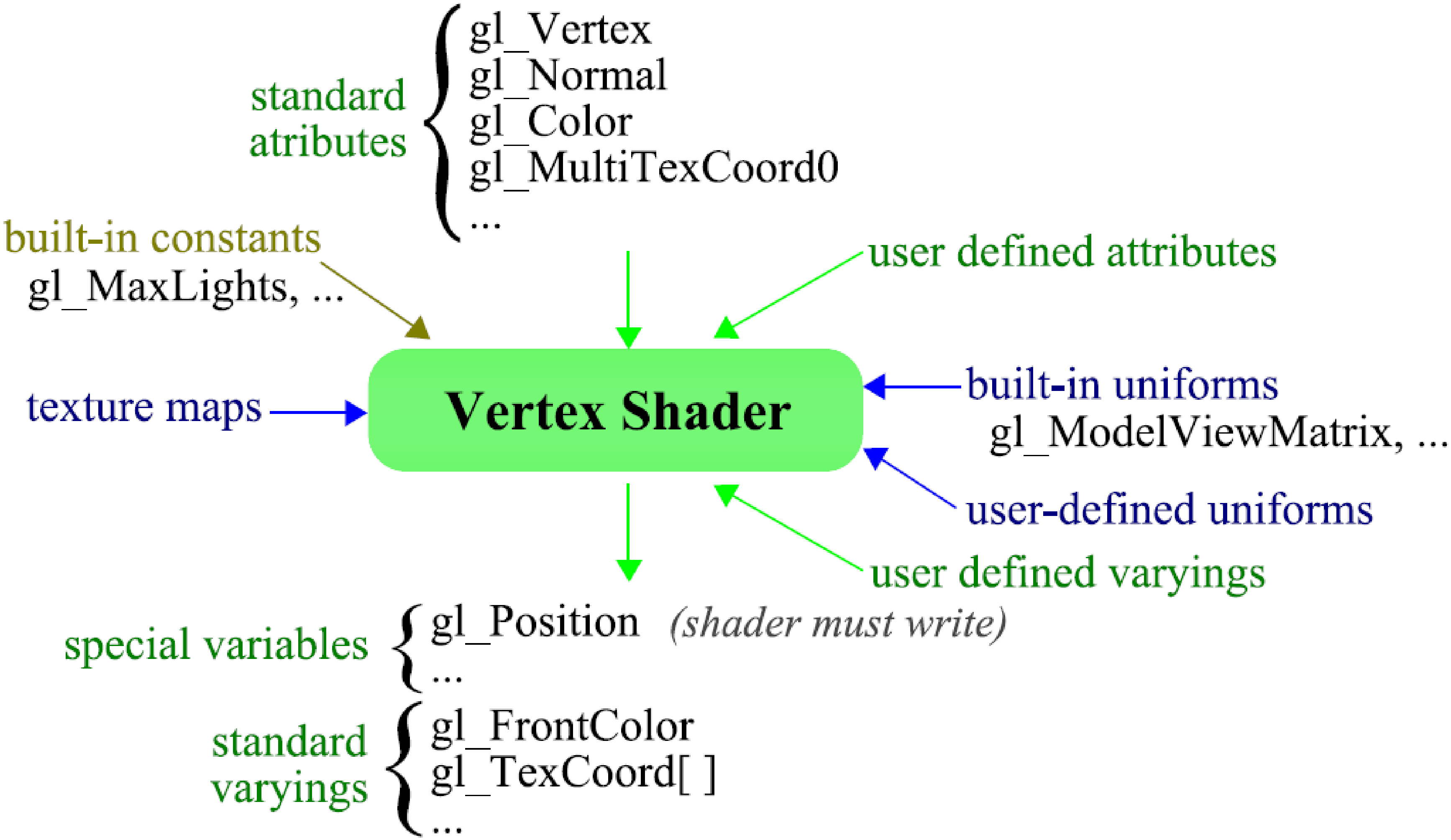


- **OpenGL 3.1 removed the fixed-function pipeline programs were required to use only shaders.**
- Almost all data is ***GPU-resident***
- All vertex data sent using **buffer objects**.

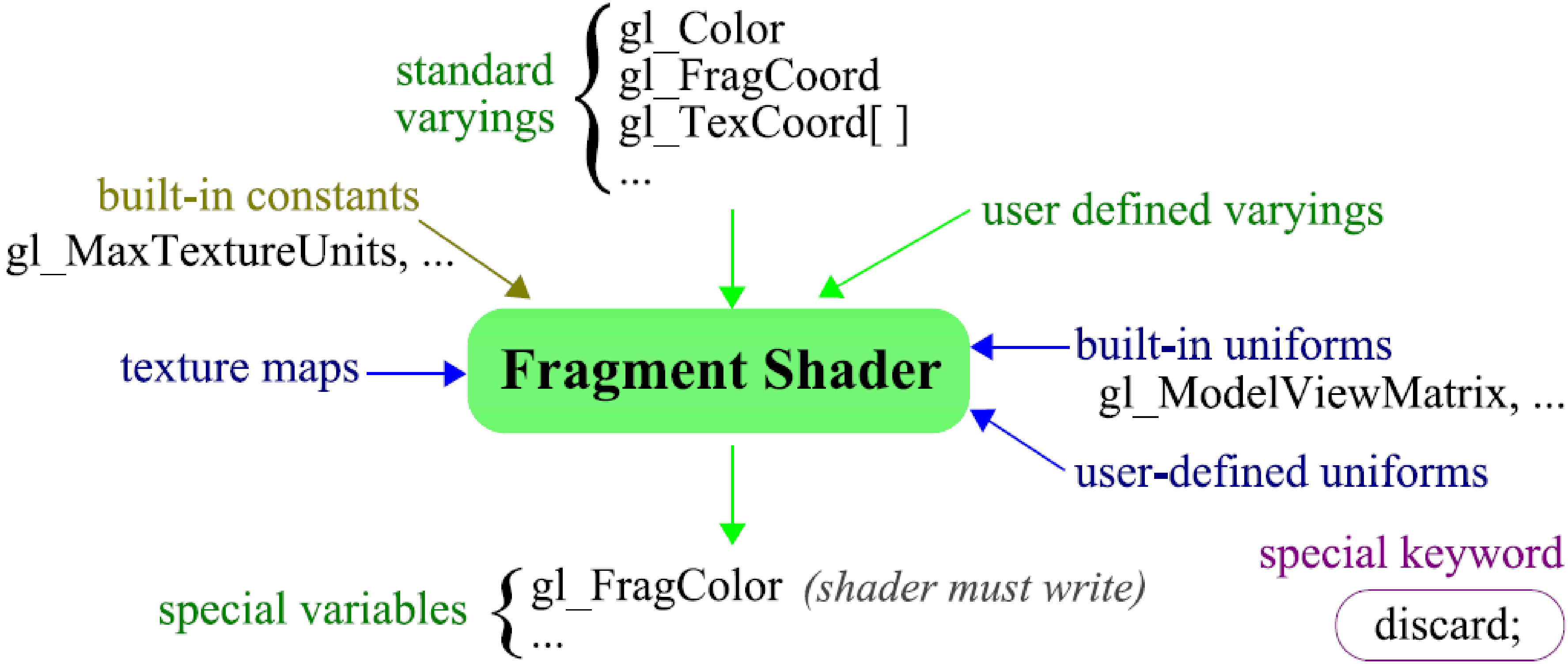
Vertex Shaders

- The main application of vertex shaders is to change the vertices of the primitives you already have defined and to setup variables such as lightening that depend of the vertices.
- The vertex shader is a one vertex in, one vertex out process, and it can't create more vertices.(Geometry and Tessellation shaders do that)

Vertex Shader

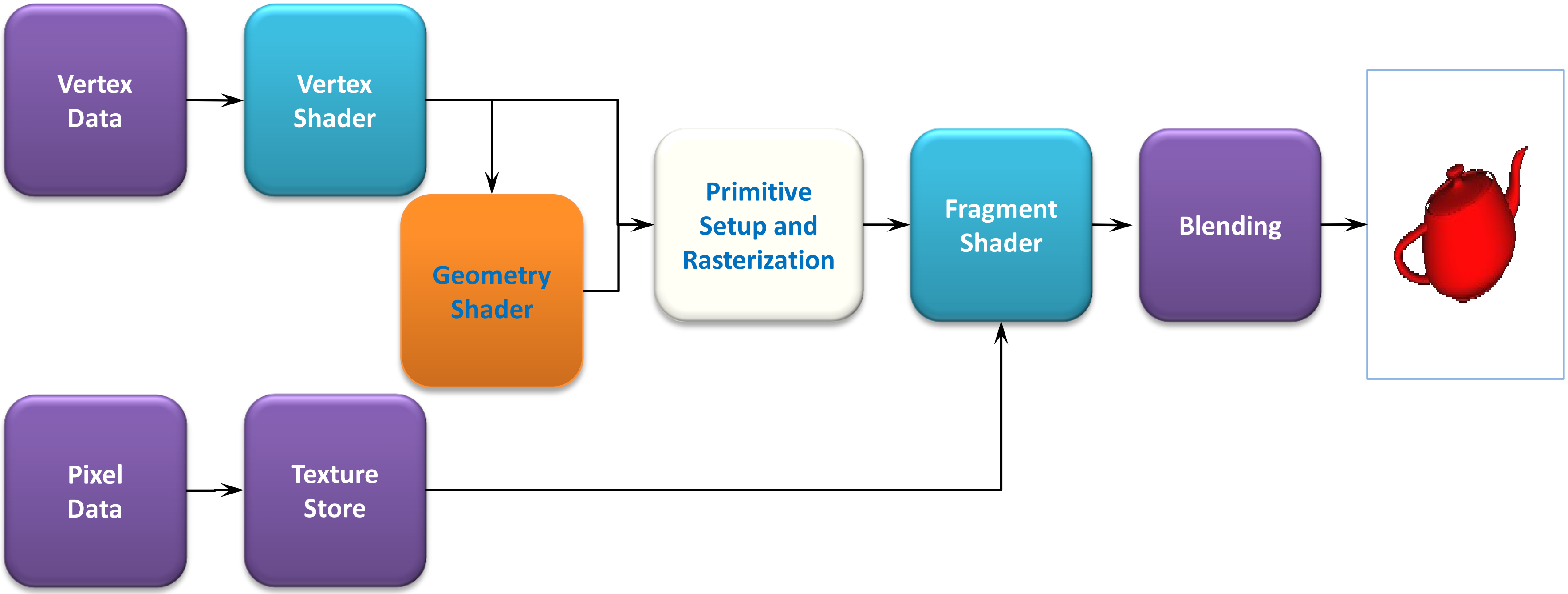


Fragment Shader



- **Discarding Pixels**
- **Anisotropic Shading**
- **Data Driven Shading**

OpenGL 3.2 pipeline



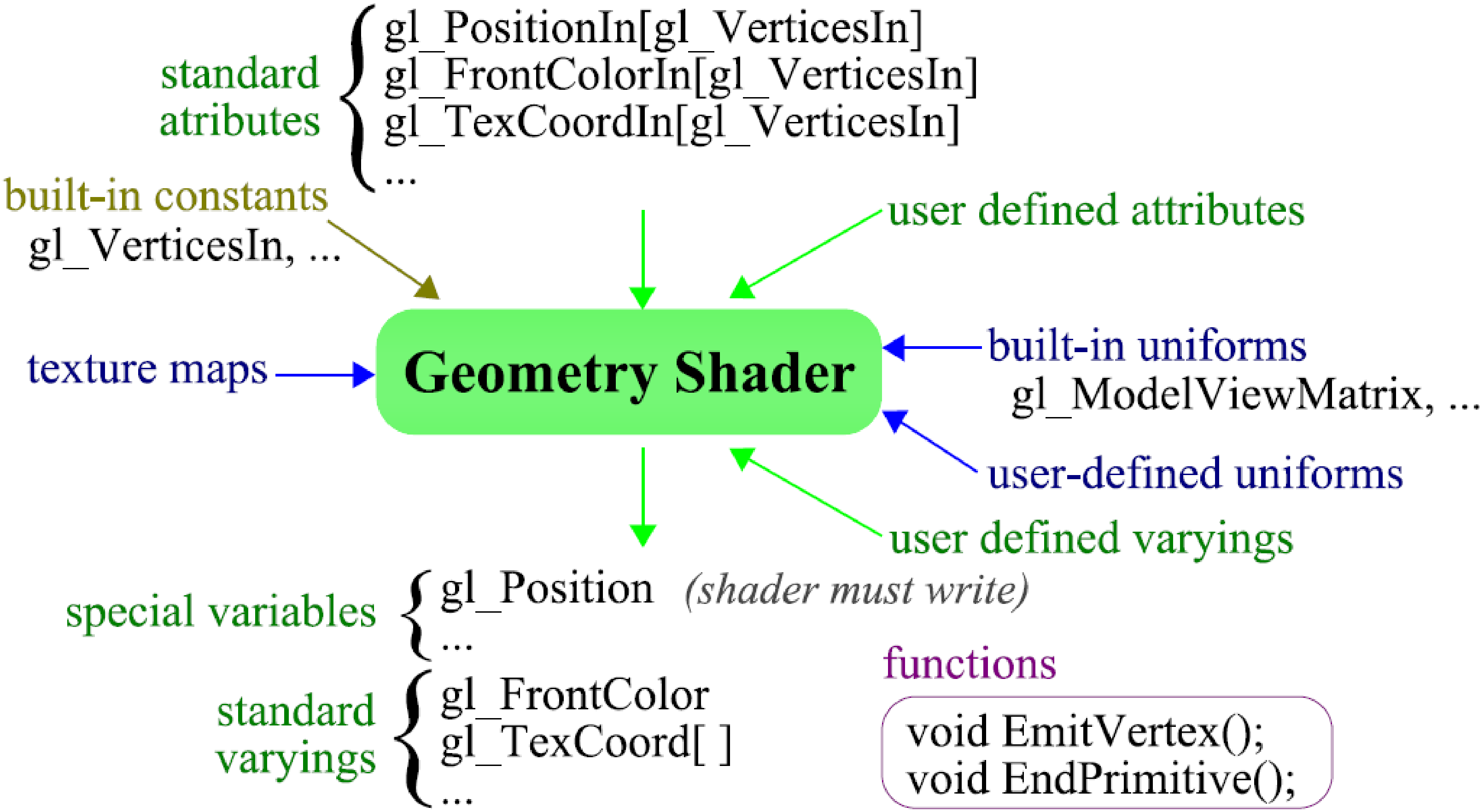
OpenGL version 3.2 added a new shader stage called *geometry shading* **which allows the modification (and generation) of geometry** within the OpenGL pipeline.

Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
    - it’s like `GL_ARB_compatibility`
  - 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	Not supported

Geometry Shader



standard atributes

gl_PositionIn[gl_VerticesIn]
gl_FrontColorIn[gl_VerticesIn]
gl_TexCoordIn[gl_VerticesIn]
...

built-in constants
gl_VerticesIn, ...

user defined attributes

texture maps

Geometry Shader

built-in uniforms
gl_ModelViewMatrix, ...

user-defined uniforms

user defined varyings

special variables

gl_Position *(shader must write)*
...

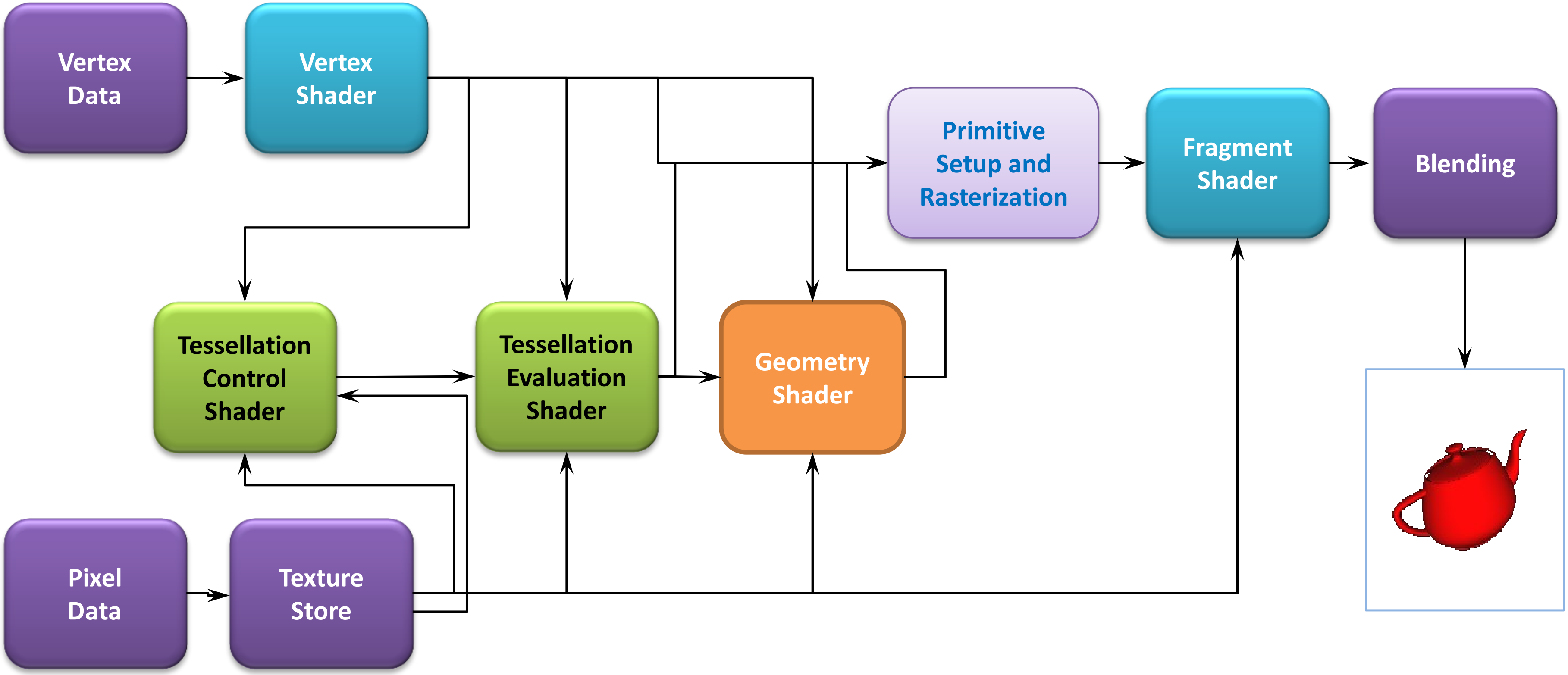
standard varyings

gl_FrontColor
gl_TexCoord[ ]
...

functions

```
void EmitVertex();
void EndPrimitive();
```

OpenGL 4.3 pipeline


Vertex Data
Vertex Shader
Primitive Setup and Rasterization
Fragment Shader
Blending
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Pixel Data
Texture Store

Released at SIGGRAPH 2012.

OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:

1. Create **shader programs**
2. Create **buffer objects** and load data into them
3. “**Connect**” data locations with shader variables
4. Render

REFERENCES

- SIGGRAPH 2012 Course : Introduction to Modern OpenGL-Ed Angel University of New Mexico, Dave Shreiner ARM.
- Graphics Shaders :Theory and Practice 2nd edition, Mike Bailey,Steve Cunningham CRC Press.