SIGGRAPH2006

EGL, OpenGL ES 1.x and OpenGL ES 2.0

Lars M. Bishop
Handheld Developer Technologies, NVIDIA

GL ES Guiding Principles

- Create a compact but powerful 3D rendering standard for embedded platforms
  - Leverage the best and most appropriate parts of OpenGL, without taking on legacy/baggage
  - Extend the result to add functionality required/desired by the embedded and mobile multimedia space
  - Be nimble and up-to-date (or ahead) w.r.t. commercial 3D hardware and software

GL ES Version / Platform Specification Methodology

- Each profile in each version of GL ES is specified as:
  - A subset of a version of full OpenGL
  - Additonal extensions and/or “core additions” that add entrypoints or functionality suited to embedded platforms

GL ES: Parallel Tracks

- Two tracks:
  - GL ES 1.x: Fixed-function pipeline implementations
  - GL ES 2.x: Programmable pipeline implementations
- The two tracks move in parallel, but are not required to be backwards compatible

GL ES 1.x

- Focused on fixed-function hardware
  - Although some vendors have added programmable shading extensions
- Designed for both HW and SW implementations
  - Later versions of 1.x are more strongly focused on HW-only implementations

GL ES 1.0

- Based on OpenGL 1.3
- Suited for both HW and SW implementations
- Includes a profile that is particularly well-suited to SW-only implementations

GL 1.3 Features not in GL ES 1.0

- “Workstation” features:
  - Selection
  - Feedback
  - Evaluators
- In fact, these features are not in any version of GL ES 1.x

GL 1.3 Features not in GL ES 1.0

- Left out due to the code complexity they would add to *all* GL ES 1.0 implementations
  - Attribute stacks
  - Display lists
- Not in 1.0, but added later (we'll discuss them then)
  - Vertex Buffer Objects (added as extension in some 1.0 HW)
  - Most dynamic render state queries
  - User clipping planes

GL 1.3 Features not in GL ES 1.0

- Features that were less popular in modern GL or that map poorly to modern HW
  - Color index mode
  - Quad and quad strip primitives
  - General polygon primitives
  - Immediate-mode rendering (glBegin/glEnd)
  - Polygon mode (the mode is always **`GL_FILL`**)
  - Line and polygon stippling (can be eumlated)

Supported “Real” Datatypes

- **`GLfloat`** (common profile only)
- **`GLfixed`** (common and lite profiles)
  - All supported GL functions that use **`GLfloat`** are replicated with versions (x) that accept **`GLfixed`**
  - This is a “core addition” to GL ES 1.x
- **`GLdouble`** is not supported
  - All GL functions that use **`GLdoubles`** are replaced with fixed (x) and float (f) versions

GLfixed?

- 16.16 (some call it s15.16) format
  - 16 bits of integral, 16 bits of fractional precision
  - Range is [-32768, 32768)
  - Precision is ~1.5×10^{-5}
- Think of it as:

```
GLfixed fixedValue = (int)(floatValue * 65536.0f)
GLfixed fixedValue = (intValue << 16)
```

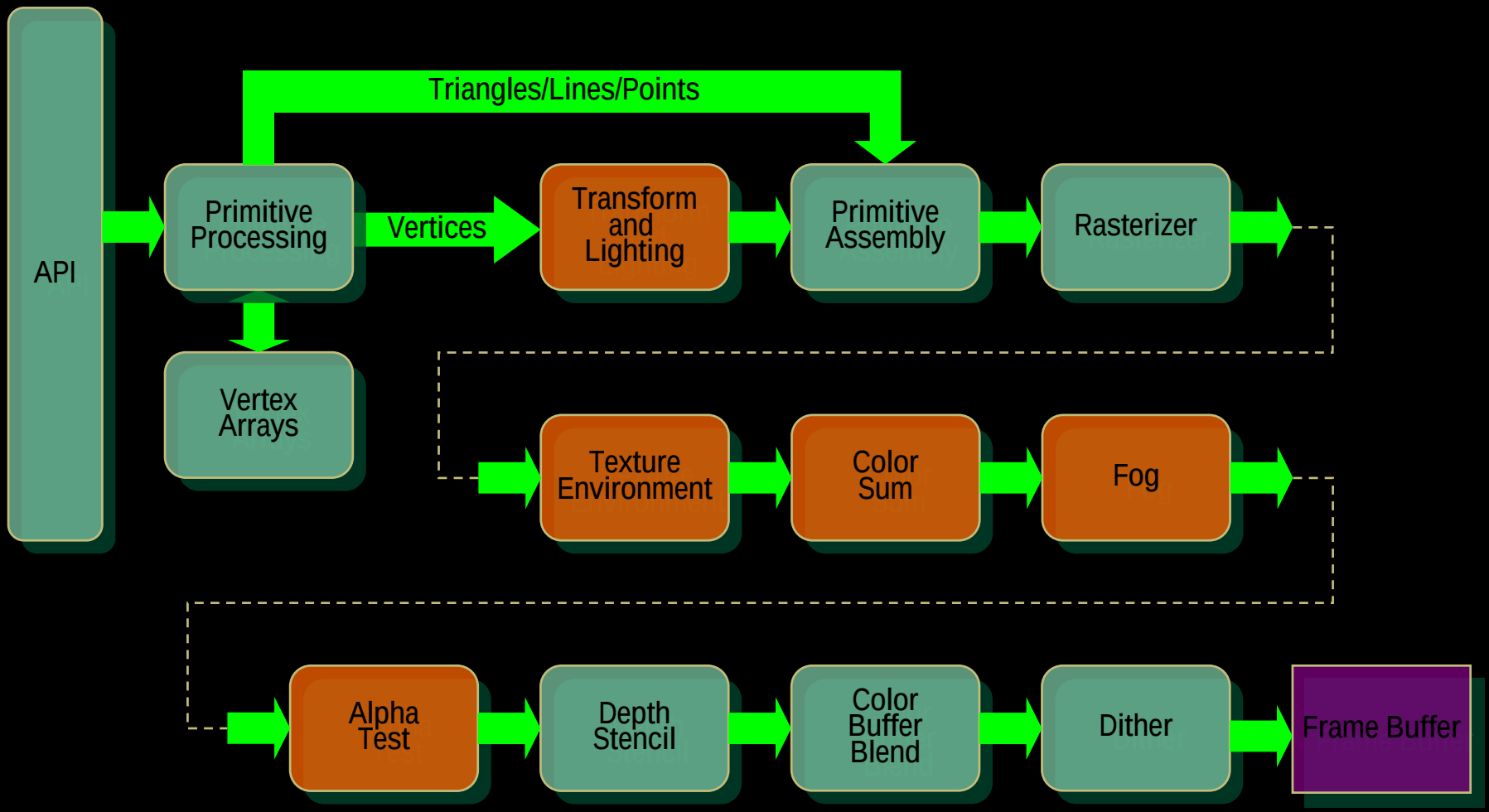
Why GLfixed?

- Many/most current handheld CPUs have no floating-point HW
- Floating-point must then be emulated in SW
- Fixed-point numbers can give some of the same features as floating-point, but standard operations are fast on integer CPUs

Profiles

- Common
  - What we'll consider GL ES 1.0 for today
- Common "Lite"
  - GL ES 1.0 with only the fixed-point functions/data
  - No support for floating-point
  - Targeted mainly at SW-only implementations

GL ES 1.0 Pipeline



Walking Through the Pipeline

- Geometry specification
- Primitive rendering
- Transforms
- Vertex processing state (e.g. lighting)
- Texture-related state
- Fragment processing state

Geometry specification

- **`glBegin` / `glEnd`** are not supported
- Geometry is specified using vertex arrays
  - **`glVertexPointer` (2D, 3D, 4D)**
  - **`glNormalPointer` (3D)**
  - **`glTexCoordPointer` (2D, 3D, 4D)**
  - **`glColorPointer` (4D)**
- Interleaved arrays are supported
  - But not via **`glInterleavedArrays`**

Per-object Vertex Components

- Three “immediate mode” functions were kept
  - **`glColor`**
  - **`glNormal`**
  - **`glMultiTexCoord`**
- These specify constant, *per-object* values
- The constant values are ignored if the pointer for that component is enabled

Supported Vertex Component Formats

- All components:
  - **`GLfloat`** (common profile only)
- All components except color:
  - **`GLfixed`**
  - **`GLshort`**
  - **`GLbyte`**
- Only valid for color
  - **`GLubyte`**

Primitive rendering

- Indexed / non-indexed primitives supported:
  - **`glDrawArrays`**
  - **`glDrawElements`**
- The most popular primitives are supported:
  - **`GL_LINES`**, **`GL_LINE_LOOP`**, **`GL_LINE_STRIP`**
  - **`GL_TRIANGLES`**, **`GL_TRIANGLE_FAN`**, **`GL_TRIANGLE_STRIP`**
- Index arrays must be **`GLubyte`** or **`GLushort`**

Transforms

- Most matrices supported (as stacks)
  - **`MODELVIEW`**
  - **`PROJECTION`**
  - **`TEXTURE_N`**
  - *NOT* **`COLOR`**
- All matrix load/concat operations supported
  - except the “**`TransposeMatrix`**” operations
- Matrices are in fixed-point or float (common)

Vertex processing state

- **`glTexGen`** not supported
- Lighting is supported (up to 8 lights)
  - But no support for separate secondary color
- Vertex normal rescaling and renormalization are supported

Vertex Lighting

- **`glMaterial`** must be **`FRONT_AND_BACK`**
  - Cannot have different front/back materials
- **`COLOR_MATERIAL`** mode is supported
  - But *only* in **`AMBIENT_AND_DIFFUSE`** mode
- **`glLightModel`** only supports
  - **`GL_AMBIENT`**
  - **`GL_LIGHT_MODEL_TWO_SIDE`**

Texturing

- Only 2D, rectangular textures are supported in GL ES 1.0
  - 1D and 3D textures are out
  - Cube map textures are out in GL ES 1.0 and 1.1
- Other unsupported texture features include
  - Border and clamp-to-border addressing modes
  - Texture proxies, prioritization, and residency

Texture Image Formats

- **RGB**, **RGBA**, **LUMINANCE**, **ALPHA** and **LUMINANCE_ALPHA** formats are supported
- **UNSIGNED_BYTE** is supported for the above types
- Also supports the following 16-bpp RGB(A) formats
  - **UNSIGNED_SHORT_5_6_5**
  - **UNSIGNED_SHORT_4_4_4_4**
  - **UNSIGNED_SHORT_5_5_5_1**

Paletted Textures

- Supported via **`glCompressedTex(Sub)Image2D`**
  - Required extension
- Supports 4- or 8-bit palette indices (16/256 entries)
- Supports the following palette entry formats
  - RGB 565, RGBA 4444, RGBA 5551
  - RGB 888, RGBA 8888
- Palette must be specified with each texture
  - Allows non-paletted implementations to implement easily

Mipmapping

- Fully supported, including all filtering modes
- Auto mipmap generation is *not* supported
- Explicit texture LOD control not supported

Texture Environments

- Multiple texture stages supported, but not required
- Most texture environment modes supported
  - **GL_REPLACE**, **GL_MODULATE**
  - **GL_ADD**
  - **GL_BLEND**, **GL_DECAL**
- But not **GL_COMBINE**, thus no
  - **COMBINE_RGB**, **COMBINE_ALPHA**
  - **SOURCE_{012}_RGB**, **SOURCE_{012}_ALPHA**

Fragment processing state

- Most of the GL fragment processing state is supported in GL ES 1.0
- However, the pixel blending only supports the additive equation, so **`glBlendEquation`** is not needed/supported
  - But all of the various blending *functions* are available (src color, dest color, src alpha, etc)

Fragment Processing State

- While depth and stencil ops are available, an implementation is not required to support *either* of them
- Ditto multisample antialiasing
- Scissoring is supported

Whole-Framebuffer Ops

- Setting the “draw buffer” is not supported, as multiple drawing buffers are not supported
- Accumulation buffers are not supported
- Color masking is supported
- All standard clear operations are supported

Unsupported Raster/Pixel Ops

- Most pixel, bitmap, rectangle operations are *not* in GL ES
  - No **`glDrawPixels`**, **`glCopyPixels`**, **`glPixelZoom`**, etc
- The entire imaging subset is unavailable as well

Supported Raster/Pixel Ops

- **`glReadPixels`** is supported, but format conversions are limited
  - Still a good function to avoid…
- **`glPixelStorei`** is limited to changing the:
  - Packing alignment for **`glReadPixels`**
  - Unpacking alignment for **`glTex[Sub]Image2D`**

Synchronization

- **`glFlush`** and **`glFinish`** are both supported
- As with GL apps, use of these should be considered carefully
- Misuse can hurt performance

Hints

- A few of the GL hints are supported
- These are of particular interest to SW implementations, where the image-quality vs. performance tradeoffs may be worthwhile
  - **`PERSPECTIVE_CORRECTION_HINT`**
  - **`POINT_SMOOTH_HINT`**
  - **`LINE_SMOOTH_HINT`**
  - **`FOG_HINT`**

GL ES 1.1

- Based on OpenGL 1.5
- Focuses more on HW implementations
- Adds paths to better feed HW vertex processing
- Adds more powerful texturing environment

Geometry Specification

- Adds vertex and index buffer objects
- Based on full GL VBOs
- Important on handheld devices, which often have slow/narrow system busses
- GL ES version has no support for memory mapping
  - No **`glMapBuffer`** / **`glUnmapBuffer`**

Rendering Primitives

- Adds required support for point sprites
- Also requires point size arrays
- Put together, these make it more like that applications can avoid having to use tri-based “screen quad” particle systems

Transformations

- Adds required matrix “get” functions
  - Makes the matrix stack tops available to applications
  - Can avoid the need for applications to mirror the GL ES matrix stacks or implement the stacks themselves

Matrix Palette Skinning (Optional)

- Adds support for an array of matrices per object
  - Not a stack of these arrays – just a single array
- Adds ability to set per-vertex arrays for:
  - The set of matrix indices for each vertex
  - The set of matrix weights for each vertex

Skinning Extension Minima

- Requires at least a 9-matrix palette
- Requires at least 3 matrices per vertex
- These minima were a bit too low to be a useful base case
  - Implementations can support more, but applications couldn’t depend on that in practice
- Thus, we’ll see an update in the GL ES 1.1 extension pack

Vertex Processing State

- Adds required support for at least one user-supplied clipping plane
  - Implementations can support/expose more
  - A single plane can be useful for portal-based rendering engines, thus the low minimum

Texturing

- Adds support for automatic mipmap generation for incomplete textures
  - Useful for rendered textures
- Automatic mipmap generation is enabled per texture via the texture parameter **`GENERATE_MIPMAP`**

Texturing Environment

- Requires support for at least two texture stages
- Adds many more texture combine environments
  - Most importantly, adds DOT3!
  - Adds *all* of the GL 1.5 modes except for the complex texture crossbar mode (see extensions)

Render State Retrieval

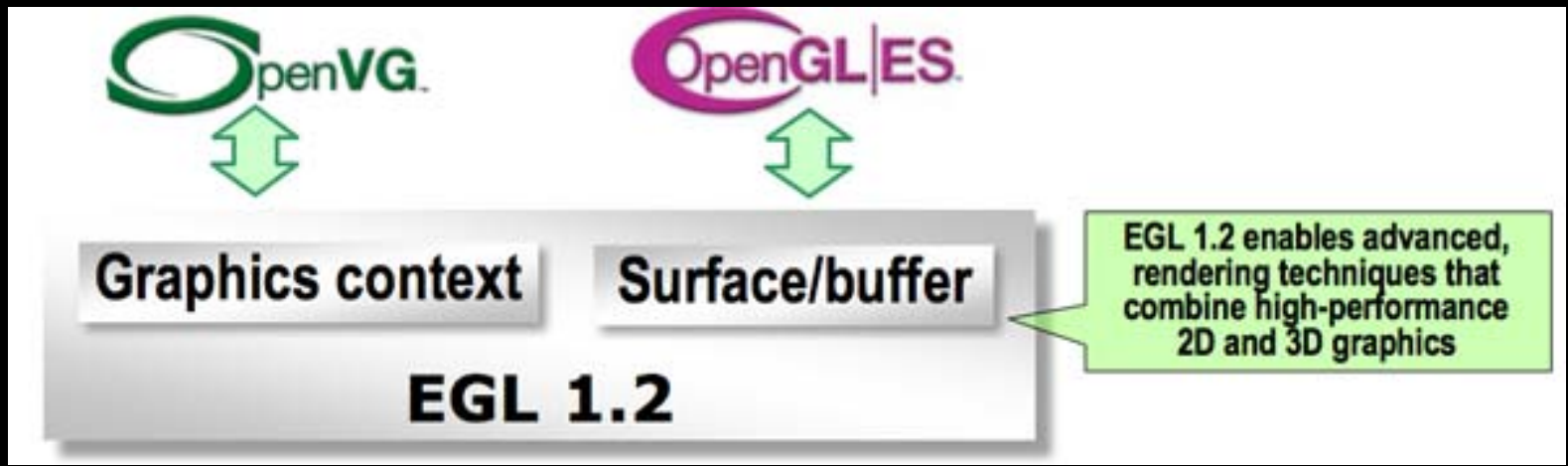
- The ability to query many of the dynamic render state values is added by the GL ES 1.1 spec
  - Mainly added for “layered” applications that need to implement their own pushing and popping of rendering state when working together with another rendering system that assumes that it is the only modifier of rendering state
- See the spec for a list of supported state

EGL

- EGL is GL ES's native platform interface
- It is designed to replace the per-platform systems used for GL
  - e.g. WGL on MS Windows
- It does *not* implement or replace the native platform's windowing system or native graphics system

EGL

- EGL also allows for resource sharing and synchronization of rendering between multiple Khronos graphics APIs



What EGL Manages

- Display devices
- Rendering contexts
- Rendering surfaces
- We will discuss EGL1.1 for most of these slides. It is focused on GL ES
- EGL1.2 generalizes many of the concepts to support other Khronos APIs

GL ES Contexts

- Although contexts are used by all GL ES rendering, GL ES cannot create them
- EGL is used to create contexts
- A context includes all client and server-side rendering state
- Applications may (and often do) have more than one context

Contexts and Sharing

- Contexts can share “large” state items:
  - Textures
  - VBOs
- Each context can be passed a “share” context, with which it will share these objects
- But be careful regarding modifications to these objects between contexts
  - See the docs for details on this

Configurations

- Configurations define the overall format of a set of rendering buffers, including:
  - Compatibility with Khronos rendering APIs
  - Color format
  - Depth/Stencil buffer (existence/bits)
  - Multisample settings
  - Compatibility with pixmap/offscreen/texture rendering, native options

Configuration Discovery

- EGL allows for applications to query all available configurations with a set of desired qualities using **`eglChooseConfig`**
  - Returns an array of matching configs sorted by a set of match metrics defined by EGL
  - Applications can then narrow the list themselves
- EGL also allows the application to get the list of all available configs via **`eglGetConfigs`**

Using Configurations

- Configurations are used during the creation of both rendering contexts and rendering surfaces
- Configurations are associated with a display object, and do not change for the life of that display object

Surfaces

- Surfaces are EGL’s rendering buffer objects
- They can represent
  - On-screen surfaces
  - Off-screen GL ES surfaces
  - Off-screen native surfaces
- They do not independently represent ancillary buffers like depth buffers
  - Ancillary buffers are part of the surface

Window Surfaces

- Window surfaces are onscreen surfaces that allow visible rendering to the current display
- They are associated with a native platform window handle
- Window surfaces can resize, depending on the platform
- Remember that EGL is not a windowing system in and of itself

PBuffer Surfaces

- PBuffers are offscreen surfaces that can be used by GL ES and EGL
- They can be a different format than the onscreen surfaces
  - This requires an additional GL ES context
- Their most common use is rendered textures

EGL1.0 Render-to-texture

- In EGL1.0 and GL ES 1.1, render-to-texture is done as follows:
  1) Make a PBuffer surface current
  2) Render to the PBuffer normally
  3) Copy the PBuffer contents to a texture via

     `glCopyTex[Sub]Image2D`
- This requires a copy from the PBuffer to the texture (although often a very fast copy)

EGL1.1 Render-to-texture

- EGL1.1 adds a new interface for direct render-to-texture: **`eglBindTexImage`**
- This allows a PBuffer to be bound directly as a texture, with no copies (usually)
- But it requires a PBuffer per rendered texture
  - Including any depth/stencil/ancillary buffers
- Support for render-to-texture is *not* required!
  - EGL doc notes how to create a supported PBuffer

Direct Render-to-texture

- As we will see, both the GL ES 1.x and 2.x tracks are moving to make direct render-to-texture required in the latest versions
- This removes the need for the previously mentioned render-to-texture methods in terms of pure GL ES support moving forward
- However, other Khronos APIs (OpenVG) still use PBuffers

Pixmap Surfaces

- Pixmap surfaces are offscreen surfaces that are “wrapped around” platform-native graphics rendering surfaces
- They allow the mixing of platform-native rendering and GL ES rendering
- By their nature, the exact way that these are used is platform-dependent

Copying into Pixmaps

- GL ES rendering buffers can be copied into pixmaps using **`eglCopyBuffers`**
- Allow for the mixing of GL ES rendering and native rendering
- Performance and exact feature support is implementation-dependent

Surfaces and Contexts

- Surfaces are associated with a rendering context dynamically via **`eglMakeCurrent`**
- A surface that is attached to a context must be “compatible” with the context
  - Have the same color/depth format as the context
  - Have been created from the same display object
- PBuffers often require a new context because they are a different pixel format

“Swapping” Buffers

- eglSwapBuffers causes the backbuffer of a windowed surface to be shown onscreen
- The call has no effect on PBuffers and Pixmaps
- Note that EGL does not guarantee swapping or copying behavior. The backbuffer contents are undefined after a swap

Synchronization

- EGL provides the interfaces to synchronizing GL ES rendering and native rendering
  - **`eglWaitGL`** waits for GL ES rendering to complete
    - Similar to **`glFinish`**
  - **`eglWaitNative`** waits for a specified native rendering interface to complete
    - **`EGL_CORE_NATIVE_ENGINE`** is always defined
    - Other native APIs can be defined by platform extensions

Extension Queries

- EGL also contains the interface for querying EGL and GL ES extension functions
  - **`eglGetProcAddress()`**
- GL ES is still used to query the **`GL_EXTENSIONS`** string for GL ES extensions
  - **`glGetString(GL_EXTENSIONS)`**

EGL1.2

- EGL 1.2 extends many of the behaviors of EGL1.1 to other APIs, such as OpenVG
  - Adds the concept of “current rendering API” via **`eglBindAPI`**, which currently accepts:
    - **`EGL_OPENGL_ES_API`**
    - **`EGL_OPENVG_API`**
  - Configurations include API compatibility bits (GL ES, VG)
  - The concept of configuration compatibility allows more GL ES configurations to be compatible with VG configurations
    - E.g. VG does not care about stencil buffers

GL ES 1.1 Extension Pack and Proposed GL ES 1.2 Reqs

- GL ES 1.1 included a set of numerous rendering feature extensions that were designed to be supported as a set
- Implementations that supported all of the items in the GL ES 1.1 Extension Pack were *likely* to be conformant to GL ES 1.2 as well
- We'll discuss them as one for now

Extended Matrix Palettes

- Makes matrix palette skinning more useful by increasing minima
  - Requires at least 32 matrices entries in a palette
  - Requires at least 4 matrices per vertex
- In general, this allows more skinned geometry to be rendered in a single draw call

Cube Map Texture Coordinate Generation

- Two modes of texgen are enabled
- **`REFLECTION_MAP`**
  - Uses generated reflection vectors to reference the cube map
- **`NORMAL_MAP`**
  - Uses the eye-space normals directly to reference the cube map

Cube Map Images

- Adds the ability to specify 6-faced cube map texture images for cube mapping:
  - **`glBindTexture(GL_TEXTURE_CUBE_MAP,…`**
  - **`glTexImage2D(GL_TEXTURE_CUBE_MAP_…`**
  - **`glCompressedTexImage2D(GL_TEXTURE_CUBE_MAP_…`**
- There are several restrictions on cube map images
  - Each face of a cube map must be square
  - All faces of a cube map must be the same size

Texture Addressing

- Adds mirrored texture wrapping
- Automatically makes any texture a “repeating”
  - i.e. without sharp border transitions
- In theory, at least – careful authoring is still required to avoid obvious texture repeated and “bookmatching” artifacts

Texture Environment

- Adds support for the texture crossbar
- This allows a texture from *any* unit to be used as a source to any texturing stage
- Still not as flexible as pixel shading, which has lead some 1.x vendors to expose proprietary pixel shading extensions

Pixel Blending Equation

- 1.1 supports only additive alpha blending
  - $C = C_S S + C_D D$ (**GL_FUNC_ADD**)
- The extension pack adds support for two others
  - $C = C_S S - C_D D$ (**GL_FUNC_SUBTRACT**)
  - $C = C_D S - C_S D$ (**GL_FUNC_REVERSE_SUBTRACT**)
- Can also support independent RGB and Alpha blending equations/functions

Stencil Action Additions

- Adds required support for two new stencil actions:
  - **`DEC_WRAP`**
  - **`INC_WRAP`**
- These are particularly useful for even/odd-based techniques with low-range stencil buffers

Framebuffer Objects

- A render-to-texture extension that is a subset of GL's **`EXT_framebuffer_object`**
- Allows for direct render-to-texture, including rendering to cube map faces
  - (Generally) avoids the overhead inherent in the **`glCopyTexImage2D`** rendered texture method
  - Allows for fewer GL ES contexts and better buffer sharing than the **`eglBindTexImage`** method

FBOs and PBuffers

- With the intention to make the GL ES 1.1 extension pack required for 1.2, FBOs become a required feature
- This greatly reduces the need for PBuffer support in the 1.x track moving forward
- Applications that can assume this version will not need to write “fallback” code for “copy-to-texture” or EGL’s “bind texture”

GL ES 2.0

- Designed to feed programmable-pipeline 3D hardware
- Based on OpenGL 2.0, but is shaders-only
- Really two parts
  - The APIs (discussed here)
  - The shading language (described elsewhere)

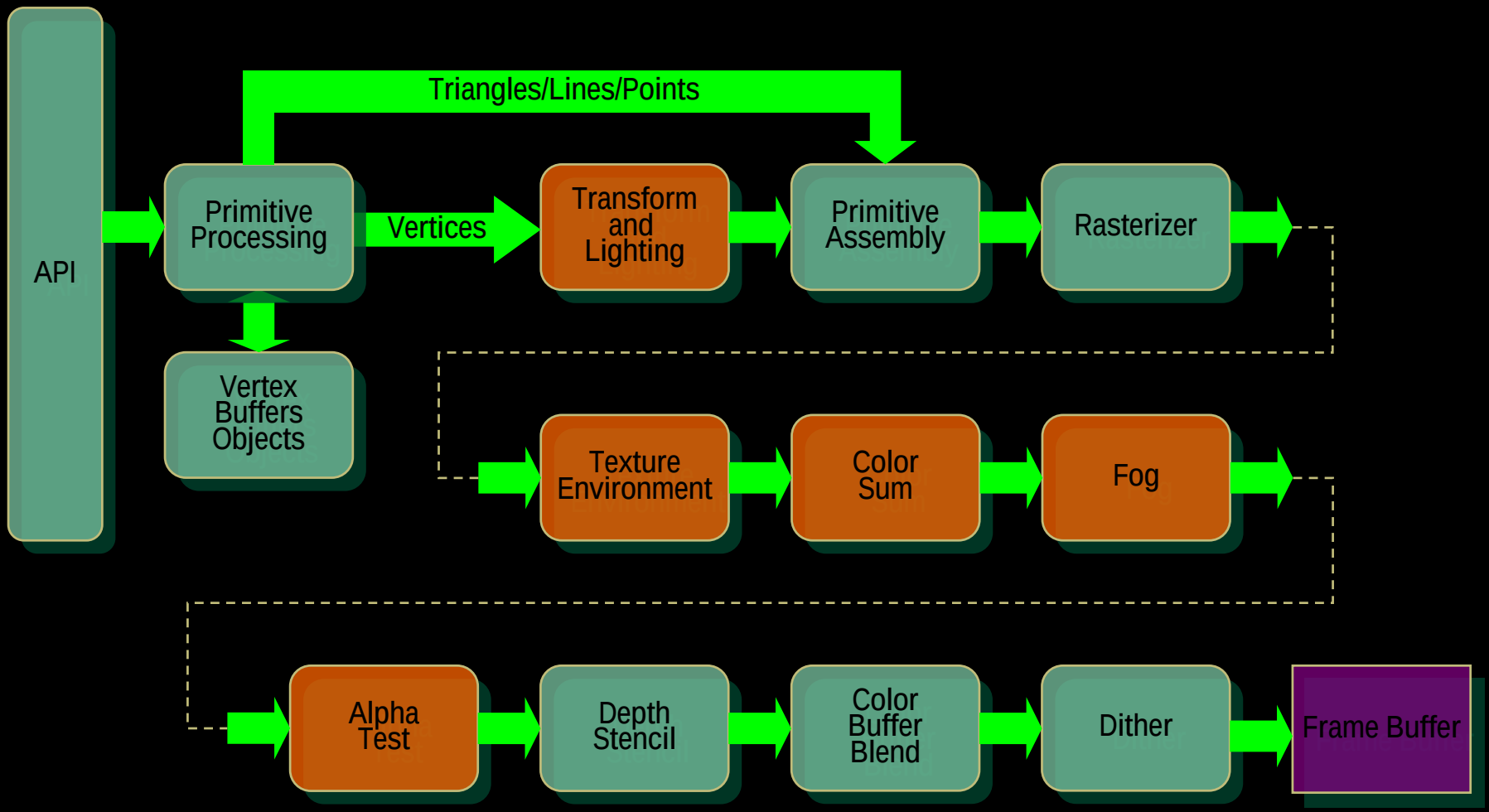
GL ES 2.0 and GL ES 1.x

- GL ES 2.0 is not backwards compatible with GL ES 1.x!
- This is a part of the more general GL ES goal of not carrying around outdated API entrypoints for the sake of backwards compatibility

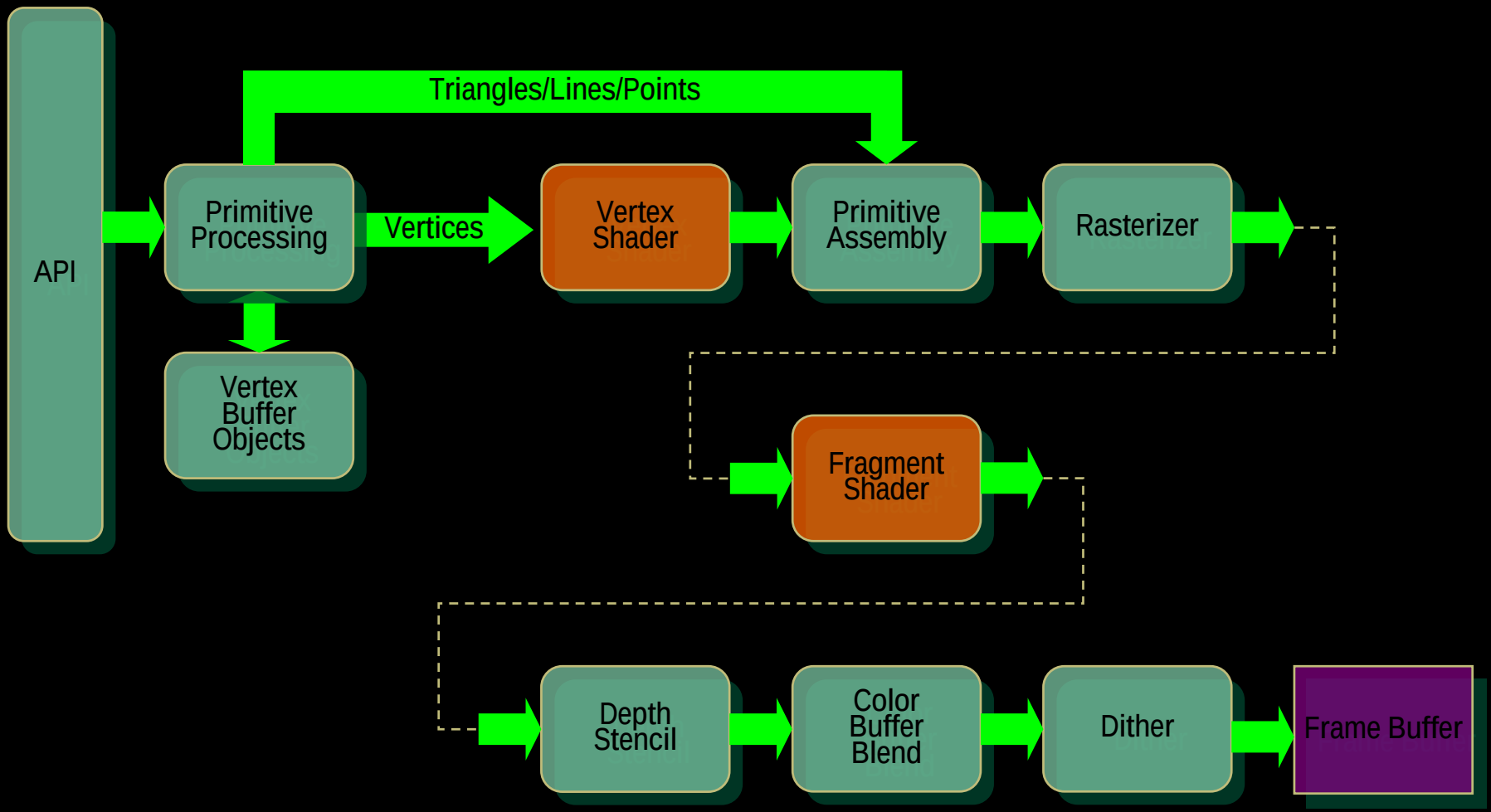
GL ES 2.0: The Big Changes

- The GL 2.0 fixed-function vertex transform pipeline is *not* supported
  - GL ES 2.0 only supports vertex shaders
- The GL 2.0 fixed-function texture environment pipeline is *not* supported
  - GL ES 2.0 only supports fragment shaders

Reminder: GL ES 1.x Pipeline



GL ES 2.0 Pipeline



GL ES 2.0 APIs

- What does this mean for the GL ES 2.0 APIs?
- It means that a *large* number of GL 2.0 entrypoints simply disappear from GL ES 2.0
- This leads to a simple, lean, uncomplicated and yet flexible API spec

Supported Types

- Unlike GL ES 1.x, 2.0 does not support fixed-point and floating-point versions of command parameters
- To simplify the APIs, only floating-point are supported for most “Real” parameters
- Fixed-point data is still supported for vertex attributes

Geometry Specification

- No fixed function pipeline means that all geometry is specified generally
- Vertex components are specified via integer indices to identify them
  - Without specifying how they are used in the shader
  - Thus, no “vertex”, “normal”, “texcoord”, etc
- These are called vertex “attributes”

Attributes

- Vertex attributes can each be specified in the API as 1D, 2D, 3D or 4D
- Attributes appears in the vertex shader as a global value, but the value differs per vertex
- Implementations must support at least 8 attribute vectors
  - Note that each attribute “costs” the same
  - So pack 1-3D values in sets of 4 into attributes

Attribute Specification Functions

- **`glVertexAttribPointer`** sets arrays of per-vertex attributes
  - Supports all types; fixed, float, byte, short, etc
- **`glVertexAttribf[v]`** sets per-primitive constant attributes
  - Float only, as performance/memory is not an issue for per-primitive values
  - Not the only/best way to do per-primitive constants

VBOs

- Support for VBOs is required
- MapBuffer and UnmapBuffer are not required
  - But support for them is exposed in an optional extension

Transformations

- None of the GL transform and matrix stack functions are supported
- GL ES 2.0 has no notion of matrix stacks
- All transforms are passed to the shaders as shader constants (uniforms)
- The application is responsible for managing and concatenating transform matrices

Lighting

- None of the GL lighting functions are supported through the APIs
- Per-vertex lighting can be computed in the vertex shader
- Per-pixel lighting can be computed via a collaboration between the vertex and pixel shaders

Other Vertex Features

- User clipping planes are not supported
  - If special clipping is required, it must be done in the fragment shader (using fragment kill) based on values computed in the vertex shader
- TexGen is not supported
  - Easier and much more general to compute in the vertex shader

Passing Data to Vertex Shaders

- “Global” data such as transforms or lighting vectors are passed to the vertex shader as Uniforms
- Uniforms are specified as
  - Scalar floats or ints
  - 2-4D float or int vectors
  - 2x2, 3x3, or 4x4 float matrices (and matrix arrays)
- At least 384 total components must be supported within a vertex shader in GL ES 2.0
  - Vector or matrix uniforms take up multiple components

Mapping Uniforms and Attributes

- Uniforms and attributes are named with strings in shader code
- But both are specified/mapped in the API using numbered indices
- GL ES allows these mappings to be queried
- This makes shader code more general avoiding over-coupling of app code and shader code

Mapping Functions

- **`glGetUniformLocation`** maps the name of a uniform in a shader to its index
- **`glGetAttribLocation`** does the same for per-vertex attributes
- **`glGetActiveAttrib/Uniform`** allows for all active attributes and uniforms in a shader to be queried (type, name, index, etc)

Vertex Shaders and Textures

- Vertex shaders can also read textures!
- But this functionality is more limited than in the fragment shader
- See the GL ES Shading Language manual for details

Vertex Shader Output

- Vertex shaders write their output to values called “varying”
- They are read/write values in the vertex shader
- They are passed as inputs to the fragment shader
- They do not appear in the API, as they go directly from the vertex shader to the fragment shader

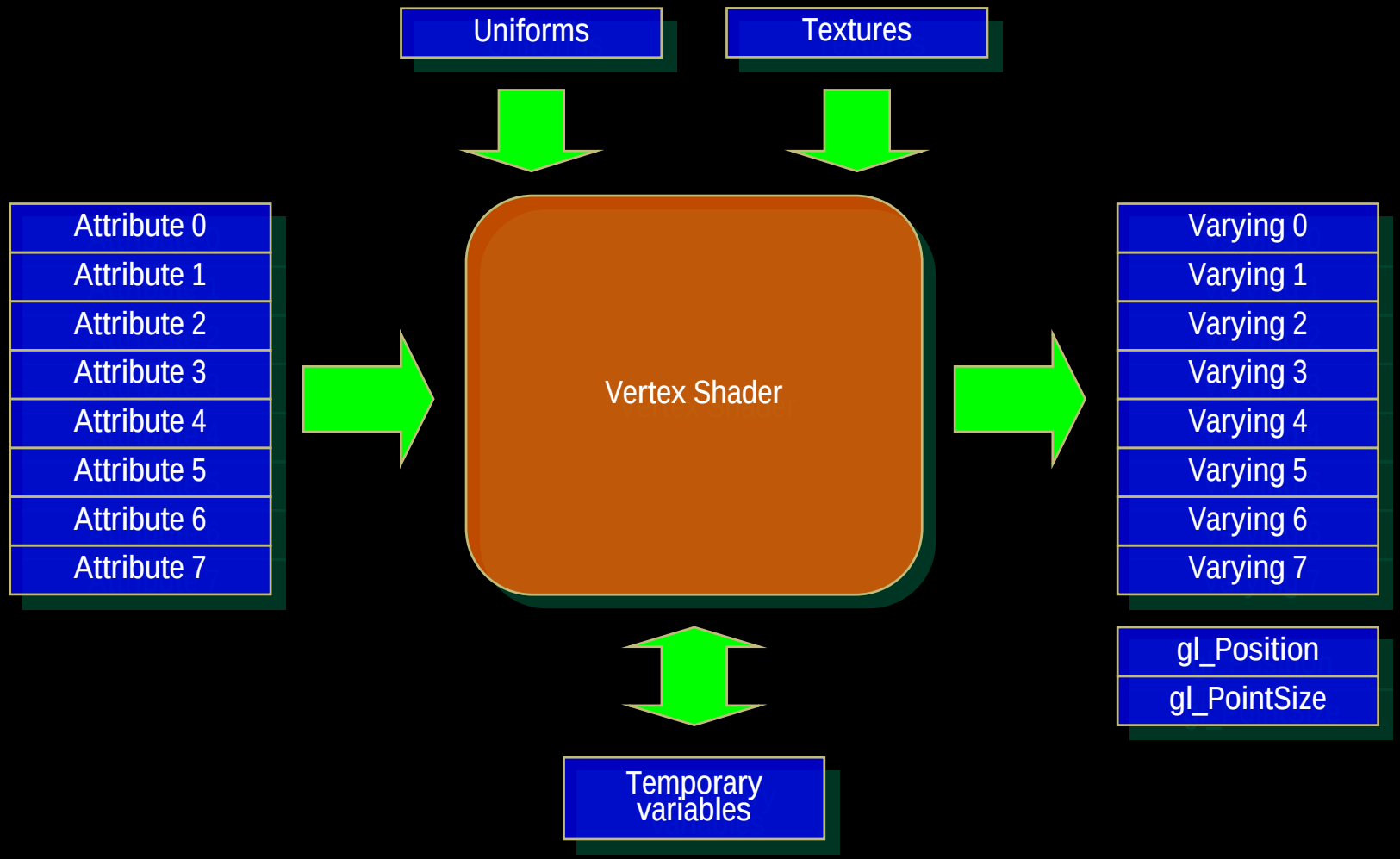
Varying

- Varyings are floats, and can be
  - Scalars
  - 1-4D vectors
  - 2x2, 3x3, or 4x4 matrices
- Implementations must support at least 32 floating-point varying components (8 4-vecs)
- These are interpolated and provided as per-fragment values to the fragment shader

Vertex Shader Special Vars

- There are two other special output variables within a shader:
  - **`gl_Position`** must be written with the homogeneous position of the vertex
  - **`gl_PointSize`** may be written with the size of a point when rendering point sprites

Overall Vertex Shader Block

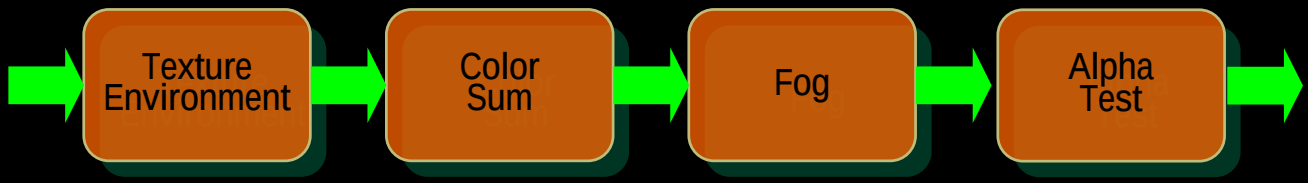


Viewport Transformations

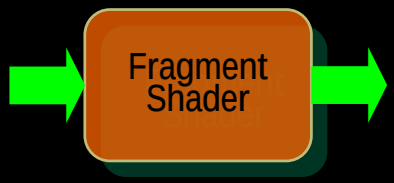
- glViewport and glDepthRange are supported
- These happen post-vertex shader and are still fixed-function operations

The Fragment Pipeline

- The GL 2.0 pipeline stages:



- Are all replaced in GL ES 2.0 by:



Inputs to the Fragment Shader

- The varyings output from the vertex shader
- Several built-in read-only variables:
  - **`gl_FragCoord`**: window-relative fragment position
  - **`gl_FrontFacing`**: true if the fragment is front-facing, false if it is back-facing
- Uniforms are also supported for fragment shaders
  - Implementation may only support 64 components

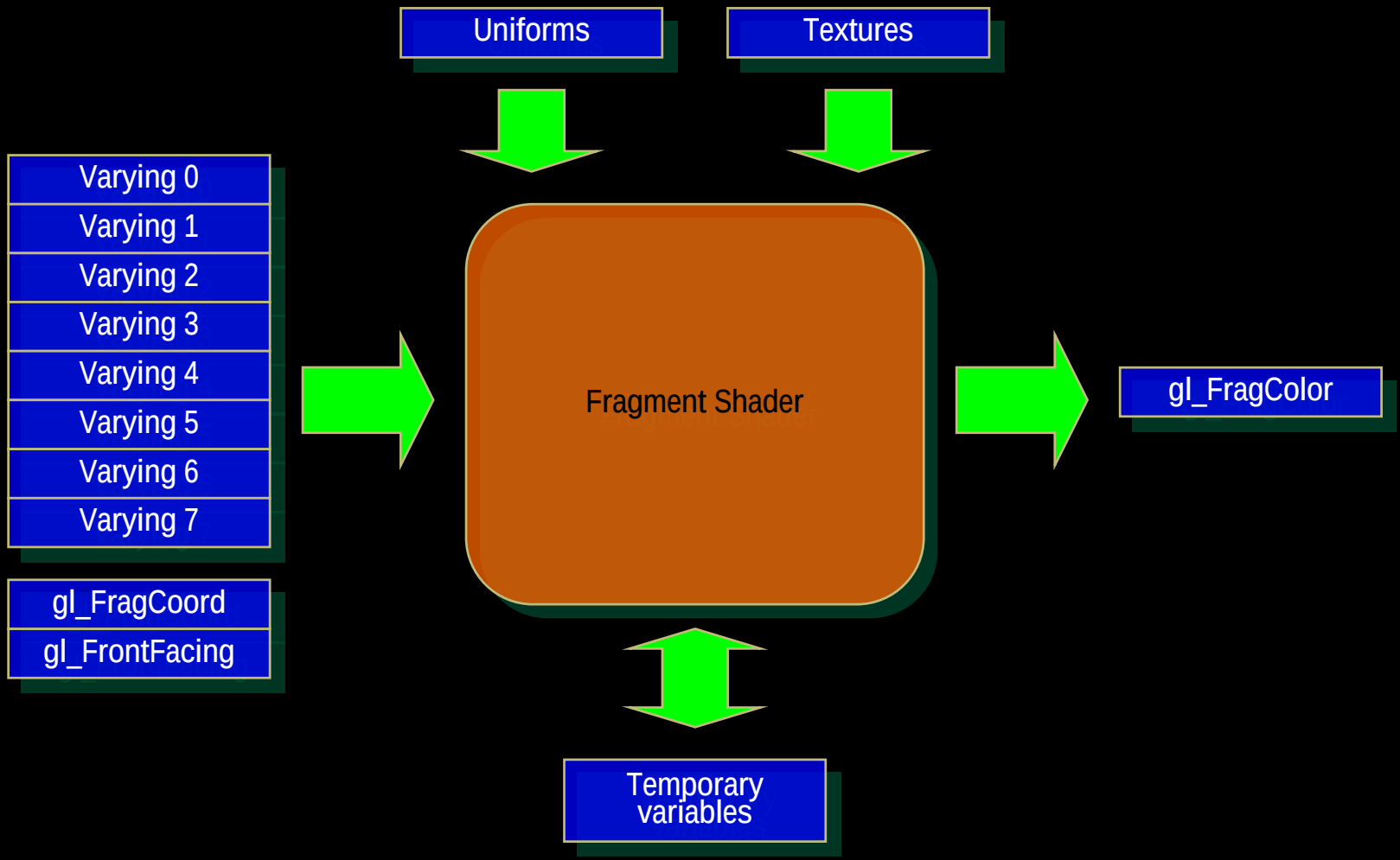
Textures

- Of course, textures are also supported in fragment shaders
- Including:
  - Cube maps
  - Non-power-of-2 textures
    - But mipmapping and texture repeat are not required
  - 3D textures are optional (see extensions)
  - Dependent texturing (in shader)

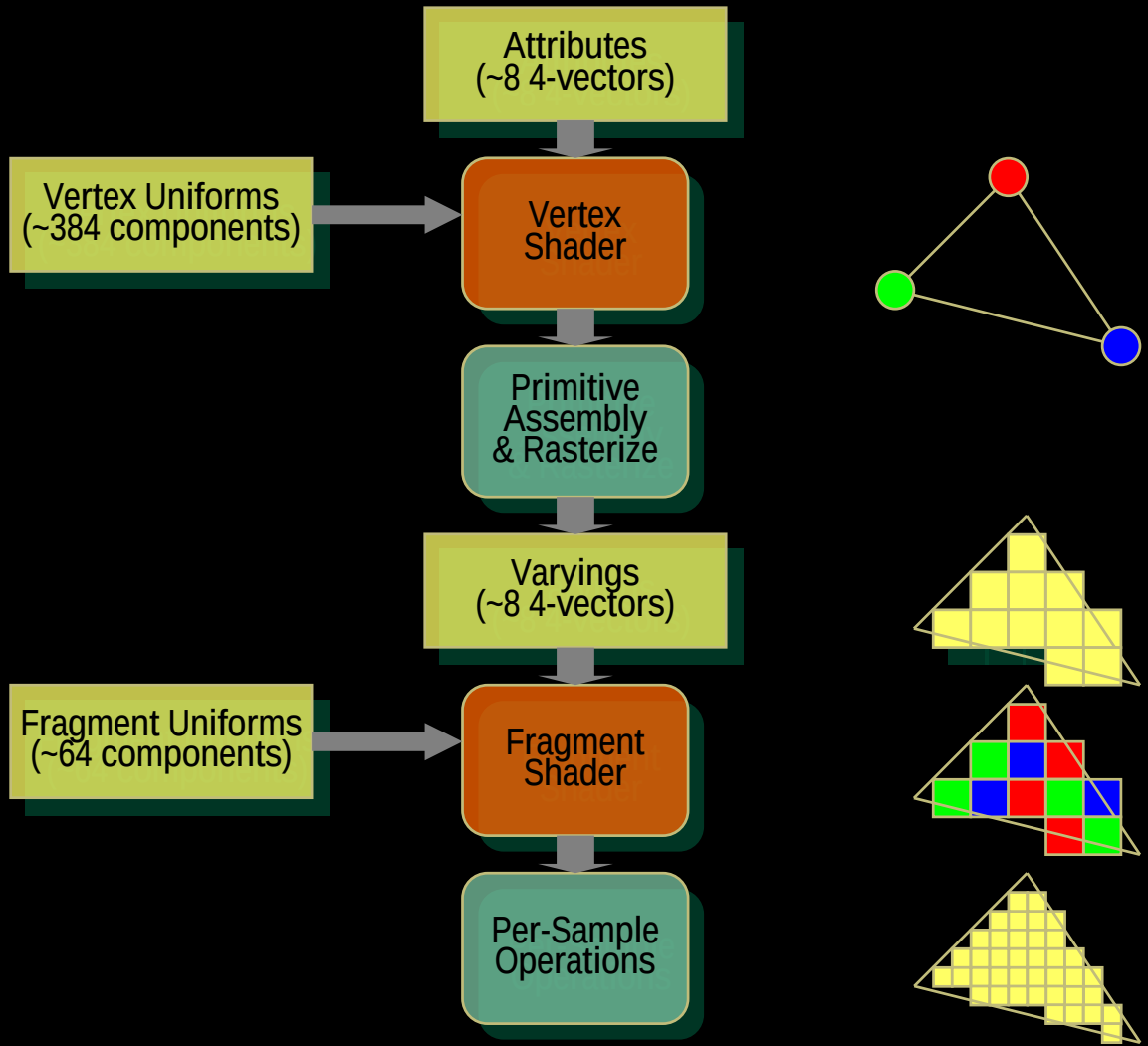
Fragement Shader Output

- Fragment shaders must output the final fragment color into the output register
  - `gl_FragColor`

Overall Fragment Shader Block



Final Programming Model



Loading and Using Shaders

- GL 2.0 requires that the shader compiler be in the implementation, and only loads shader source code
  - This could be too slow or heavyweight for GL ES
- GL ES 2.0 allows implementations to support either (or both) of the following two methods:
  - GL 2.0 model: load source and compile
  - Load platform-specific precompiled binary shaders

Replaced Fragment APIs

- Obviously, fragment shaders make parts of the per-fragment pipeline redundant
- These fixed-function APIs from GL 2.0 are replaced by fragment shaders
  - Texture environments
  - Color summing
  - Alpha testing

Supported Fragment APIs

- Depth testing is supported
  - The spec *requires* a config with 16 bits of depth
- Stencil testing is supported
  - The spec *requires* a config with 8 bits of stencil
- Pixel blending is supported
  - Supported modes similar to GL ES 1.2

Whole Framebuffer APIs

- Roughly equivalent to GL ES 1.2
  - Support for color masking
  - Support for **`glReadPixels`**
  - No support for **`glCopyPixels`**
  - No support for multiple draw buffers
  - No support for accumulation buffers

GL ES 2.0 Required Extensions

- **`OES_read_format`**
  - Platform-specific format support for **`glReadPixels`**
- **`OES_stencil8`**
  - Requires at least one configuration with a depth buffer and 8 bits of stencil
- **`OES_framebuffer_object`**
  - Made required (like the slated GL ES 1.2 spec)

GL ES 2.0 Optional Extensions

- GL ES 2.0 includes a lot of standard but optional extensions
- The ones listed here are a subset

FBO Render Mipmap

- **`OES_fbo_render_mipmap`**
- Adds mipmap-level rendering options to framebuffer objects
- If supported, adds the ability to render into any mipmap level of a texture

Half Float Vertex Components

- **`OES_vertex_half_float`**
- Adds support for vertex attributes that are 16-bit floating-point
- Format
  - 1 sign bit
  - 5 exponent bits
  - 10 mantissa bits

Floating-point Textures

- **`OES_texture_half_float`**
  `OES_texture_float`
  - Support 16- and 32-bit floating-point textures with only NEAREST texture filtering (w/ mipmapping)
- **`OES_texture_half_float_linear`**
  `OES_texture_float_linear`
  - As above, but add linear texture filtering
- 16-bit formats are the same as for attributes

32-bit Element Indices

- **`OES_element_index_uint`** adds support for 32-bit index array elements
- Allows for very large objects to be drawn in a single call

Other Texture Extensions

- 3D Textures
  - **`OES_texture_3D`**
- Non Power-of-two Textures
  - **`OES_texture_npot`**
  - Adds repeat wrapping and mipmapping to non-power-of-two textures
  - If 3D textures are also supported, this indicates that 3D non-power-of-two textures are supported

ETC Compressed Textures

- **`OES_compressed_ETC1_RGB8_texture`**
- Adds support for Ericsson's texture compression formats

High-precision Fragments

- OES fragment precision high
- Adds support for “high precision” qualifiers in fragment shader code
- Supported on floating-point and integer types in fragment shaders