EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Computer Graphics

Computer Graphics
(Graphische Datenverarbeitung)

- OpenGL 1-

Hendrik Lensch

WS 2021/2022

Corona

- Regular random lookup of the 3G certificates

- Contact tracing: We need to know who is in the class room
  - New ILIAS group for every lecture slot
  - Register via ILIAS or this QR code (only if you are present in this room)

Overview

- Last lecture:
  - Rasterization
  - Clipping

- Today:
  - OpenGL
    - Vertex shader
    - Geometry shader
    - Fragment shader
    - Textures

- Next lecture:
  - Advanced OpenGL features

Ray Tracing vs. Rasterization

- Ray tracing
  - For every pixel
    - Locate first object visible in a certain direction
  - Requires spatial index structure to be fast

- Rasterization
  - For every object
    - Locate all covered pixels
    - Interpolate values for every fragment in a primitive
  - Uses 2D image coherence but not necessarily an index structure

History

- Graphics in the ‘80ies
  - Designated memory in RAM
  - Set individual pixels directly via memory access
    - peek & poke, getpixel & putpixel, …
  - Everything done on CPU, except for driving the display
  - Dump „frame buffer“
- Today
  - Separate graphics card connected via high-speed link (e.g. PCIe)
    - Autonomous, high performance GPU (much more powerful than CPU
    - Up to 2496 SIMD processors, 264 GB/s memory access
    - Up to 6 GB of local RAM plus virtual memory
  - Performs all low-level tasks & a lot of high-level tasks
    - Clipping, rasterization, hidden surface removal, …
    - Procedural shading, texturing, animation, simulation, …
    - Video rendering, de- and encoding, deinterlacing, ...
    - Full programmability at several pipeline stages

Introduction to OpenGL

- Brief history of graphics APIs
  - Initially every company had its own 3D-graphics API
  - Many early standardization efforts
    - CORE, GKS/GKS-3D, PHIGS/PHIGS-PLUS, ...
  - 1984: SGI´s proprietary Graphics Library (GL / IrisGL)
    - 3D rendering, menus, input, events, text rendering, ...
    - „Naturally grown“
  - OpenGL (1992, Mark Segal & Kurt Akeley):
    - Explicit design of a general vendor independent standard
      - Close to hardware but hardware-independent
      - Efficient
      - Orthogonal
      - Extensible
    - Common interface from mobile phone to supercomputer
  - Direct3D (1996, Microsoft - part of DirectX)
  - Flash (2000, Adobe)
    - 2D, since 2011 3D extension

Introduction to OpenGL

- What is OpenGL?
  - Software interface for graphics hardware (API)
    - Thin hardwre abstraction layer – almost direct access to HW
    - AKA an “instruction set” for the GPU
  - Controlled by the Architecture Review Board (ARB, now Khronos WG)
    - SGI, Microsoft, IBM, Intel, Apple, Sun, and many more
  - Only covers 2D/3D rendering
    - Other APIs: MS Direct3D (older: IrisGL, PHIGS, Starbase, …)
    - Related GUI APIs - X Window, MS Windows GDI, Apple, ...
  - Original <3.0 API focused on **immediate-mode** operation
    - Triangles as base primitives – directly submitted by application
    - More efficient batch processing with vertex arrays (and display lists)
  - Current >=3.0 Core API: **retained mode** only
    - Buffers, Renderbuffers, Textures
  - Network-transparent protocol
    - GLX-Protocol – X Window extension (only in X11 environment!)
    - Direct (hardware access) versus indirect (protocol) rendering

Introduction to OpenGL

- What is OpenGL (cont´d)?
  - Low-level API
    - Difficult to program OpenGL efficiently
      - Assembly language for graphics
    - Few good high level scene graph APIs
      - OpenSG, OpenScenegraph, Performer, Java3D, Optimizer/Cosmo3D, OpenInventor, Direct3D-RM, NVSG, ...
  - Extensions
    - Explicit request for extensions (at compile and run time)
    - Allows HW vendors to add new features independent of ARB
      - No central control (by MS)
      - Could accelerate innovation
  - OpenGL APIs < 3.0
    - ~ OpenGL 1.2 + mandatory set of extensions
      → All functionality of OpenGL 1.1 is kept
  - OpenGL APIs >= 3.0
    - Independend specification → deprecated functionality is removed

OpenGL Version History

- OpenGL 1.0
  - Initial release (1992)
- OpenGL 1.1
  - Vertex arrays and texture objects
  - Newest OpenGL that is usable directly on windows
    - For newer Versions, function pointers have to be retrieved manually
- OpenGL 1.2
  - Extension support, most extensions are designed for this version
- OpenGL 1.3 - 1.5
  - More flexibility, minor new features
- OpenGL 2.x
  - Vertex and pixel shaders
  - Offscreen rendertargets

OpenGL Version History

- OpenGL 3.0 – 3.1
  - Introduction of Profiles:
    - “Compatibility” for backward compatibility
    - “Core” for a slim API that contains everything important
- OpenGL 3.2 – 3.3
  - Geometry shaders
  - OpenCL interoperability
- OpenGL 4.0 – 4.2
  - Hardware tessellation
- OpenGL 4.3
  - Compute shaders

The remaining slides show OpenGL 4.2!

Latest: Vulcan

Creating an OpenGL Context

- Not part of the OpenGL specification
- Platform dependent interfaces
  - Windows: WGL
  - Linux: GLX
  - More details on http://www.opengl.org
    - Windows: “http://www.opengl.org/wiki/Context_creation”
    - Linux: “Tutorial: OpenGL 3.0 Context Creation (GLX)”
- Platform independent wrappers available
  - SDL
  - GLUT, freeGLUT
  - GLFW
  - …

The following examples will assume that an OpenGL context is already established!

Reference Card

OpenGL® is the only cross-platform graphics API that enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually-compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality.

Specifications are available at www.opengl.org/registry

- *See FunctionName* refers to functions on this reference card.
- [n.n.n] and [Table n.n] refer to sections and tables in the OpenGL 4.5 core specification.
- [n.n.n] refers to sections in the OpenGL Shading Language 4.50 specification.

Command Execution [2.3]

OpenGL Errors [2.3.1]

enum **GetError**(void);

Graphics Reset Recovery [2.3.2]

enum **GetGraphicsResetStatus**(void);
Returns: NO_ERROR, GUILTY_CONTEXT_RESET, {INNOCENT, UNKNOWN}_CONTEXT_RESET

GetIntegerv(RESET_NOTIFICATION_STRATEGY);
Returns: NO_RESET_NOTIFICATION, LOSE_CONTEXT_ON_RESET

Flush and Finish [2.3.3]

void **Flush**(void); void **Finish**(void);

Floating-Point Numbers [2.3.4]

16-Bit	1-bit sign, 5-bit exponent, 10-bit mantissa
Unsigned 11-Bit	no sign bit, 5-bit exponent, 6-bit mantissa
Unsigned 10-Bit	no sign bit, 5-bit exponent, 5-bit mantissa

Command Letters [Tables 2.1, 2.2]

Where a letter denotes a type in a function name, T within the prototype is the same type.

b -	byte (8 bits)	ub -	ubyte (8 bits)
s -	short (16 bits)	us -	ushort (16 bits)
i -	int (32 bits)	ui -	uint (32 bits)
i64 -	int64 (64 bits)	ui64 -	uint64 (64 bits)
f -	float (32 bits)	d -	double (64 bits)

OpenGL Command Syntax [2.2]

GL commands are formed from a return type, a name, and optionally up to 4 characters (or character pairs) from the Command Letters table (to the left), as shown by the prototype:

return-type **Name**{1234}{b s i i64 f d ub us ui ui64}{v} (*[args ,] T arg1 , . . . , T argN [, args]*);

The arguments enclosed in brackets ([args ,] and [, args]) may or may not be present.

The argument type T and the number N of arguments may be indicated by the command name suffixes. *N* is 1, 2, 3, or 4 if present. If "v" is present, an array of *N* items is passed by a pointer. For brevity, the OpenGL documentation and this reference may omit the standard prefixes.

The actual names are of the forms: glFunctionName(), GL_CONSTANT, GLtype

Synchronization

Sync Objects and Fences [4.1]

void **DeleteSync**(sync *sync*);

sync **FenceSync**(enum *condition*, bitfield *flags*);
condition: SYNC_GPU_COMMANDS_COMPLETE
flags: must be 0

Waiting for Sync Objects [4.1.1]

enum **ClientWaitSync**(sync *sync*, bitfield *flags*, uint64 *timeout_ns*);
flags: SYNC_FLUSH_COMMANDS_BIT, or zero

void **WaitSync**(sync *sync*, bitfield *flags*, uint64 *timeout*);
timeout: TIMEOUT_IGNORED

Sync Object Queries [4.1.3]

void **GetSynciv**(sync *sync*, enum *pname*, sizei *bufSize*, sizei *length*, int *values*);
pname: OBJECT_TYPE, SYNC_{STATUS, CONDITION, FLAGS}

boolean **IsSync**(sync *sync*);

Asynchronous Queries [4.2, 4.2.1]

void **GenQueries**(sizei *n*, uint *ids*);

void **CreateQueries**(enum *target*, sizei *n*, uint *ids*);
target: See BeginQuery, plus TIMESTAMP

void **DeleteQueries**(sizei *n*, const uint *ids*);

void **BeginQuery**(enum *target*, uint *id*);
target: ANY_SAMPLES_PASSED[_CONSERVATIVE], PRIMITIVES_GENERATED, SAMPLES_PASSED, TIME_ELAPSED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN

void **BeginQueryIndexed**(enum *target*, uint *index*, uint *id*);
target: See BeginQuery

void **EndQuery**(enum *target*);

void **EndQueryIndexed**(enum *target*, uint *index*);

boolean **IsQuery**(uint *id*);

void **GetQueryiv**(enum *target*, enum *pname*, int *params*);
target: See BeginQuery, plus TIMESTAMP
pname: CURRENT_QUERY, QUERY_COUNTER_BITS

void **GetQueryIndexediv**(enum *target*, uint *index*, enum *pname*, int *params*);
target: See BeginQuery, plus TIMESTAMP
pname: CURRENT_QUERY, QUERY_COUNTER_BITS

void **GetQueryObjectiv**(uint *id*, enum *pname*, int *params*);

void **GetQueryObjectuiv**(uint *id*, enum *pname*, uint *params*);

void **GetQueryObjecti64v**(uint *id*, enum *pname*, int64 *params*);

void **GetQueryObjectui64v**(uint *id*, enum *pname*, uint64 *params*);
pname: QUERY_TARGET, QUERY_RESULT[_NO_WAIT, _AVAILABLE]

Timer Queries [4.3]

Timer queries track the amount of time needed to fully complete a set of GL commands.

void **QueryCounter**(uint *id*, TIMESTAMP);

void **GetIntegerv**(TIMESTAMP, int *data*);

void **GetInteger64v**(TIMESTAMP, int64 *data*);

Buffer Objects [6]

void **GenBuffers**(sizei *n*, uint *buffers*);

void **CreateBuffers**(sizei *n*, uint *buffers*);

void **DeleteBuffers**(sizei *n*, const uint *buffers*);

Create and Bind Buffer Objects [6.1]

void **BindBuffer**(enum *target*, uint *buffer*);
target: [Table 6.1] {ARRAY, UNIFORM}_BUFFER, {ATOMIC_COUNTER, QUERY}_BUFFER, COPY_{READ, WRITE}_BUFFER, {DISPATCH, DRAW}_INDIRECT_BUFFER, {ELEMENT_ARRAY, TEXTURE}_BUFFER, PIXEL_[UN]PACK_BUFFER, SHADER_STORAGE_BUFFER, TRANSFORM_FEEDBACK_BUFFER

void **BindBufferRange**(enum *target*, uint *index*, uint *buffer*, intptr *offset*, sizeiptr *size*);
target: ATOMIC_COUNTER_BUFFER, {SHADER_STORAGE, UNIFORM}_BUFFER, TRANSFORM_FEEDBACK_BUFFER

void **BindBufferBase**(enum *target*, uint *index*, uint *buffer*);
target: See BindBufferRange

void **BindBuffersRange**(enum *target*, uint *first*, sizei *count*, const uint *buffers*, const intptr *offsets*, const sizeiptr *size*);
target: See BindBufferRange

void **BindBuffersBase**(enum *target*, uint *first*, sizei *count*, const uint *buffers*);
target: See BindBufferRange

Create/Modify Buffer Object Data [6.2]

void **BufferStorage**(enum *target*, sizeiptr *size*, const void *data*, bitfield *flags*);
target: See BindBuffer
flags: Bitwise OR of MAP_{READ, WRITE}_BIT, {DYNAMIC, CLIENT}_STORAGE_BIT, MAP_{COHERENT, PERSISTENT}_BIT

void **NamedBufferStorage**(uint *buffer*, sizeiptr *size*, const void *data*, bitfield *flags*);
flags: See BufferStorage

void **BufferData**(enum *target*, sizeiptr *size*, const void *data*, enum *usage*);
target: See BindBuffer
usage: DYNAMIC_{DRAW, READ, COPY}, {STATIC, STREAM}_{DRAW, READ, COPY}

void **NamedBufferData**(uint *buffer*, sizeiptr *size*, const void *data*, enum *usage*);

void **BufferSubData**(enum *target*, intptr *offset*, sizeiptr *size*, const void *data*);
target: See BindBuffer

void **NamedBufferSubData**(uint *buffer*, intptr *offset*, sizeiptr *size*, const void *data*);

void **ClearBufferSubData**(enum *target*, enum *internalFormat*, intptr *offset*, sizeiptr *size*, enum *format*, enum *type*, const void *data*);
target: See BindBuffer
internalformat: See TexBuffer on pg. 3 of this card
format: RED, GREEN, BLUE, RG, RGB, RGBA, BGR, BGRA, {RED, GREEN, BLUE, RG, RGB}_INTEGER, {RGBA, BGR, BGRA} _INTEGER, STENCIL_INDEX, DEPTH_{COMPONENT, STENCIL}

void **ClearNamedBufferSubData**(uint *buffer*, enum *internalFormat*, intptr *offset*, sizeiptr *size*, enum *format*, enum *type*, const void *data*);
internalformat, format, type: See ClearBufferSubData

void **ClearBufferData**(enum *target*, enum *internalformat*, enum *format*, enum *type*, const void *data*);
target, internalformat, format: See ClearBufferSubData

void **ClearNamedBufferData**(uint *buffer*, enum *internalformat*, enum *format*, enum *type*, const void *data*);
internalformat, format, type: See ClearBufferData

Map/Unmap Buffer Data [6.3]

void ***MapBufferRange**(enum *target*, intptr *offset*, sizeiptr *length*, bitfield *access*);
target: See BindBuffer
access: The Bitwise OR of MAP_*X*_BIT, where *X* may be READ, WRITE, PERSISTENT, COHERENT, INVALIDATE_{BUFFER, RANGE}, FLUSH_EXPLICIT, UNSYNCHRONIZED

void ***MapNamedBufferRange**(uint *buffer*, intptr *offset*, sizeiptr *length*, bitfield *access*);
target: See BindBuffer
access: See MapBufferRange

void ***MapBuffer**(enum *target*, enum *access*);
access: See MapBufferRange

void ***MapNamedBuffer**(uint *buffer*, enum *access*);
access: See MapBufferRange

void **FlushMappedBufferRange**(intptr *offset*, sizeiptr *length*);

void **FlushMappedNamedBufferRange**(uint *buffer*, intptr *offset*, sizeiptr *length*);

boolean **UnmapBuffer**(enum *target*);
target: See BindBuffer

boolean **UnmapNamedBuffer**(uint *buffer*);

Invalidate Buffer Data [6.5]

void **InvalidateBufferSubData**(uint *buffer*, intptr *offset*, sizeiptr *length*);

void **InvalidateBufferData**(uint *buffer*);

Buffer Object Queries [6, 6.7]

boolean **IsBuffer**(uint *buffer*);

void **GetBufferSubData**(enum *target*, intptr *offset*, sizeiptr *size*, void *data*);
target: See BindBuffer

void **GetNamedBufferSubData**(uint *buffer*, intptr *offset*, sizeiptr *size*, void *data*);

void **GetBufferParameteri[64]v**(enum *target*, enum *pname*, int[64]*data*);
target: See BindBuffer
pname: [Table 6.2] BUFFER_SIZE, BUFFER_USAGE, BUFFER_{ACCESS[_FLAGS]}, BUFFER_MAPPED, BUFFER_MAP_{OFFSET, LENGTH}, BUFFER_{IMMUTABLE_STORAGE, ACCESS_FLAGS}

void **GetNamedBufferParameteri[64]v**(uint *buffer*, enum *pname*, int[64]*data*);

void **GetBufferPointerv**(enum *target*, enum *pname*, const void **params*);
target: See BindBuffer
pname: BUFFER_MAP_POINTER

void **GetNamedBufferPointerv**(uint *buffer*, enum *pname*, const void **params*);
pname: BUFFER_MAP_POINTER

Copy Between Buffers [6.6]

void **CopyBufferSubData**(enum *readTarget*, enum *writeTarget*, intptr *readOffset*, intptr *writeOffset*, sizeiptr *size*);
readTarget and writeTarget: See BindBuffer

void **CopyNamedBufferSubData**(uint *readBuffer*, uint *writeBuffer*, intptr *readOffset*, intptr *writeOffset*, sizeiptr *size*);

Shaders and Programs

Shader Objects [7.1-2]

uint **CreateShader**(enum *type*);
type: {COMPUTE, FRAGMENT}_SHADER, {GEOMETRY, VERTEX}_SHADER, TESS_{EVALUATION, CONTROL}_SHADER

void **ShaderSource**(uint *shader*, sizei *count*, const char * const * *string*, const int *length*);

void **CompileShader**(uint *shader*);

void **ReleaseShaderCompiler**(void);

void **DeleteShader**(uint *shader*);

boolean **IsShader**(uint *shader*);

void **ShaderBinary**(sizei *count*, const uint *shaders*, enum *binaryformat*, const void *binary*, sizei *length*);

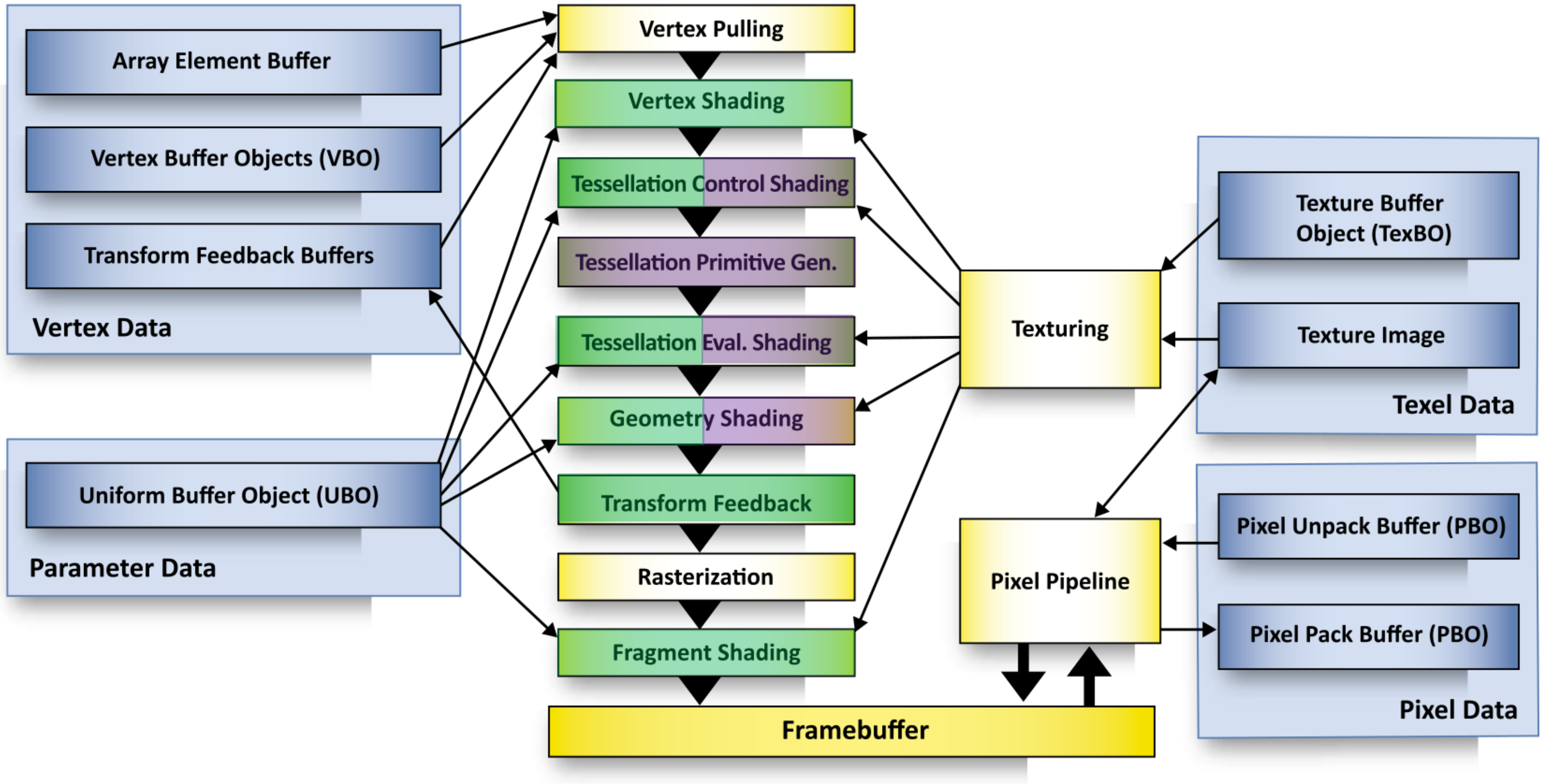
(Continued on next page) ▶

Objects, Targets and States

- The OpenGL API is not object oriented!
  - Only names of objects (Gluint) are exposed
- Objects can be bound to targets
  - To become active
  - To change their properties (their state)
- Properties of objects can be changed
  - By using the object’s name
  - By binding the object to a target and calling functions related to the target
- There are global properties
  - Target bindings
  - Viewport size
  - Clear color
  - …

The current global state of OpenGL and the state of the bound objects determine what happens at a draw call!

Overview



- Programmable pipeline stages
- Optional pipeline stages

Creating Shader Programs

- Shader: Defines the behaviour of one programmable pipeline stage
- Shader Program: Defines the behaviour of ALL programmable pipeline stages
- Building a shader program from files containing the shader source (tessellation and geometry are optional):

```
Gluint createShaderProgram(const char* vsFilename, const char* tcFilename,
        const char* teFilename, const char* gsFilename, const char* fsFilename)
{
    // Create the shader program
    sp = glCreateProgram();

    // Create vertex shader
    vs = glCreateShader(GL_VERTEX_SHADER);
    ifstream vsFile(vsFilename);
    string buffer((std::istreambuf_iterator<char>(vsFile)),
            std::istreambuf_iterator<char>());
    vsFile.close();
    code = buffer.c_str();
    bufferLength = buffer.length();
    glShaderSource(vs, 1, &code, &bufferLength);
    glCompileShader(vs);
    glAttachShader(sp, vs);
    glDeleteShader(vs); // Only flags for deletion if shader is attached!
```

Creating Shader Programs

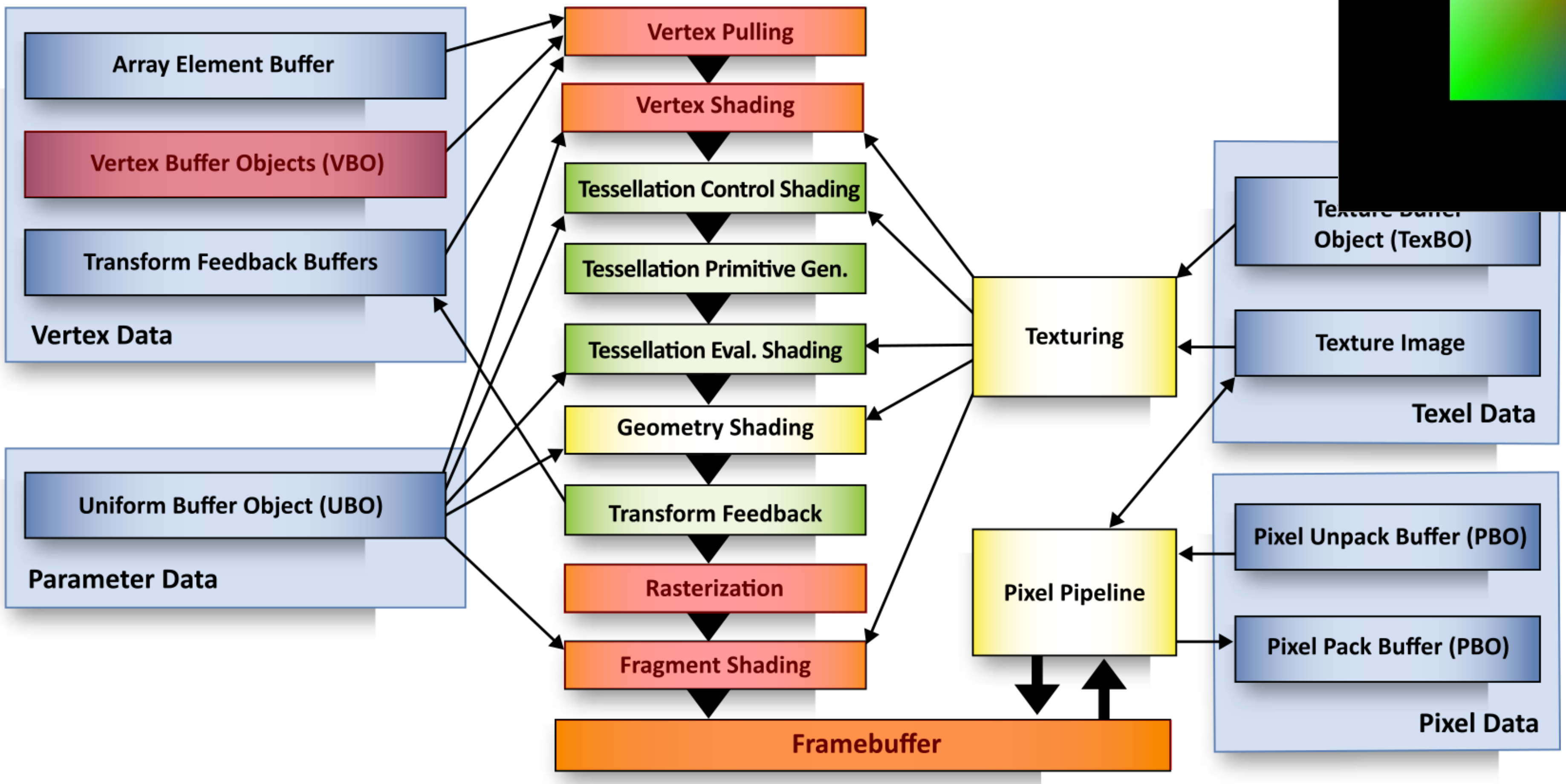
```
    // Create tessellation evaluation shader
    if (tcFilename)
    {
        tc = glCreateShader(GL_TESS_CONTROL_SHADER);
        ifstream tcFile(tcFilename);
        string buffer((std::istreambuf_iterator<char>(tcFile)),
                std::istreambuf_iterator<char>());
        tcFile.close();
        code = buffer.c_str();
        bufferLength = buffer.length();
        glShaderSource(tc, 1, &code, &bufferLength);
        glCompileShader(tc);
        glAttachShader(sp, tc);
        glDeleteShader(tc); // Only flags for deletion if shader is attached
    }

    // ======================================
    // Build te, gs and fs the same way here!
    // ======================================

    // Link shader program
    glLinkProgram(sp);

    // Return the name of the linked shader program
    return sp;
}
```

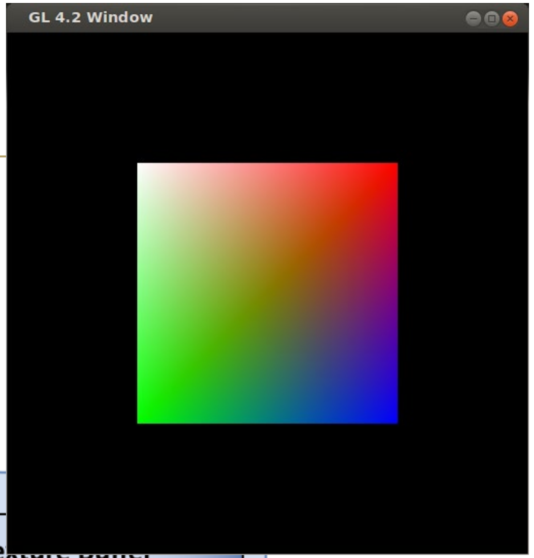
Examples

- The following slides show examples from minimal to advanced:
  - Colored quad (vertex + fragment shaders)
  - Point sprites (geometry shaders)
  - Textured quad (textures)
  - Animated sprites (Transform Feedback)
  - Tessellated quad (tesselation)
  - Picture in picture (rendering to textures)
- For each example, the new techniques used are explained
- Examples build upon each other!

Colored Quad

Colored Quad





Colored Quad - Shaders

- Massively parallel in hardware
  - One thread per vertex, primitive or fragment
- Additional qualifiers for variables
  - `in`: From the previous pipeline stage
  - `out`: To the next pipeline stage
  - `uniform`: Defined in the host program, read only inside the shader, the same for all threads / data items in one shader program
  - Local: All variables defined without additional qualifiers
- In and out variables of successive shaders must match!
- Builtin variables
  - Needed as interface to non-programmable pipeline stages
  - E.g. predefined `out vec4 gl_Position` vertex shader variable which is connected to the rasterizer (if tesselation and geometry shading is disabled)

Colored Quad - Shaders

- Vector support
  - Shading languages support vec[2-4] with some overloaded operators, casts and builtin functions.
- Vertex shaders
  - Executed once per vertex
  - See only the data of one vertex plus uniform variables
- Fragment shaders
  - Executed once per rendered (sub)pixel
  - All vertex data from previous programmable pipeline stages gets interpolated for fragment processing based on barycentric coordinates

Colored Quad – Vertex Shader

```
#version 420 core
in vec2 inPosition;
in vec3 inColor;

out VertexData     // naming structure for transfer of output data to next shader phase
{
    vec3 Color;
} outData;         // name of output variable (structure)

void main(void)
{
    gl_Position = vec4(inPosition, 0.0, 1.0);
    outData.Color = inColor;
}
```

- Input / Output variables can be grouped
- Creates a homogenous 3d position from the 2d input position
- Forwards the color to the next pipeline stage
  - Here: fragment shader

Colored Quad – Fragment Shader

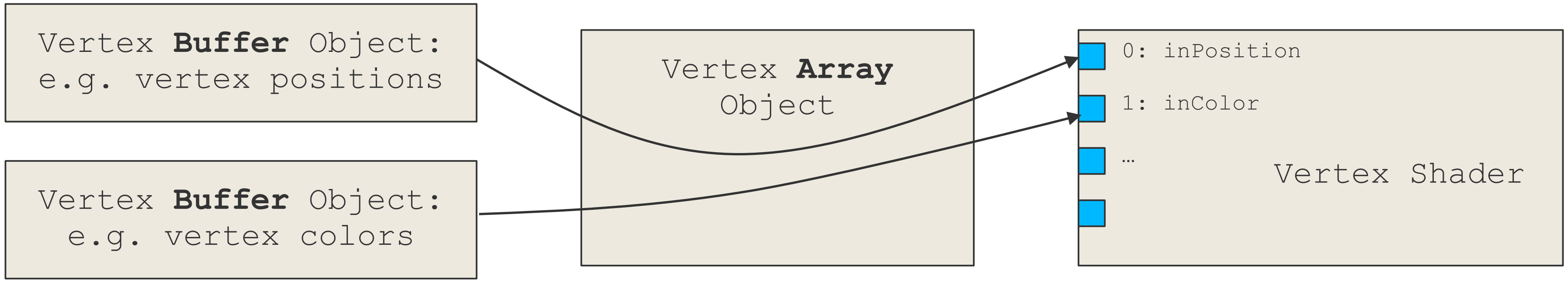
EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
#version 420 core

in VertexData           // incoming from vertex shader
{
    vec3 Color;
} inData;

out vec4 outColor;      // could be flexible

void main(void)
{
     outColor = vec4(inData.Color, 1.0);      // just hand over the incoming color
}
```

- The gl_Position variable from the vertex shader was removed during rasterization
- **Rasterization interpolates all other variables**

→ Color from vertex shader can be used directly as fragment color to produce smooth shading

Colored Quad – Vertex Pulling



- Vertex Buffer Object: Data buffer in GPU memory
- Vertex Array Object
  - Links data in Array Buffers to input locations (0:, 1:, …) of the vertex shader
  - Stores how OpenGL reads from the Array Buffers
    - Which data type?
    - Vector dimensions?
    - Offset
    - Stride

Colored Quad – Rasterization

- Rasterization interprets the vertex coordinates homogeneously
  - (x’, y’, z’) = (x, y, z) / w
- A Viewport defines which area of the target framebuffer will be drawn
  - Usually, the viewport is the whole framebuffer
- Mapping vertices to the (quadratic) viewport:

  x’ = -1 left border

  x’ = 1 right border

  y’ = -1 bottom border

  y’ = 1 top border
- z’ is used for depth comparison and written to the depth buffer.

Colored Quad – Host Code

```
// Create shader
GLuint sp = createShaderProgram("Shaders/colored2d.vs", NULL, NULL, NULL,
        "Shaders/colored2d.fs");

// Create mesh
float mesh[] =
{   -0.5, 0.5, 0.5, 0.5, -0.5, -0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5};
float colors[] =
{   1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f};

// Vertex Buffer Object for positions
GLuint vb;     // vertex buffer object name
glGenBuffers(1, &vb);     // generate new object name
glBindBuffer(GL_ARRAY_BUFFER, vb);  // make sure to use the object
                                    // for all following operations
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 2 * 6, mesh, GL_STATIC_DRAW);
                              // assign/copy data to the bufferobject

// Vertex Buffer Object for colors
GLuint cb;     // vertex buffer object name
glGenBuffers(1, &cb);
glBindBuffer(GL_ARRAY_BUFFER, cb);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 3 * 6, colors,
        GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, 0);    // just for safety reasons bind to null
```

Colored Quad – Host Code

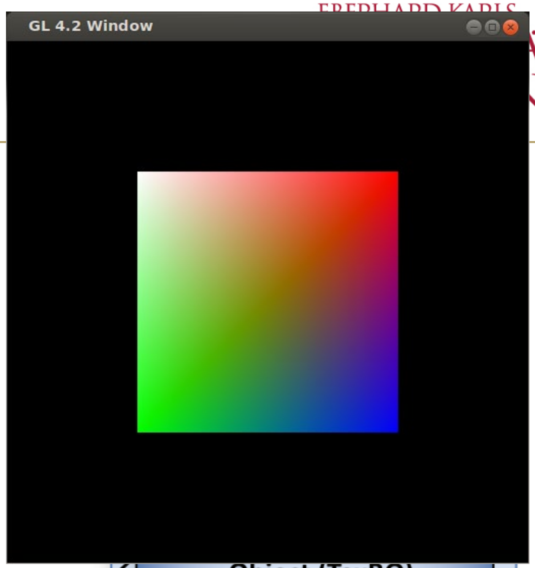
```
// create a Vertex Array Object
GLuint va;
glGenVertexArrays(1, &va);
glBindVertexArray(va);

// Link position and color to vertex shader input variables
glBindBuffer(GL_ARRAY_BUFFER, vb);

// the variables that should be passed to the shader need to be enabled
glEnableVertexAttribArray(glGetAttribLocation(sp, "inPosition"));
glVertexAttribPointer(glGetAttribLocation(sp, "inPosition"), 2, GL_FLOAT,
        GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, cb);
glEnableVertexAttribArray(glGetAttribLocation(sp, "inColor"));
glVertexAttribPointer(glGetAttribLocation(sp, "inColor"), 3, GL_FLOAT,
        GL_FALSE, 0, 0);

// unbind VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
// unbind VAO
glBindVertexArray(0);
```

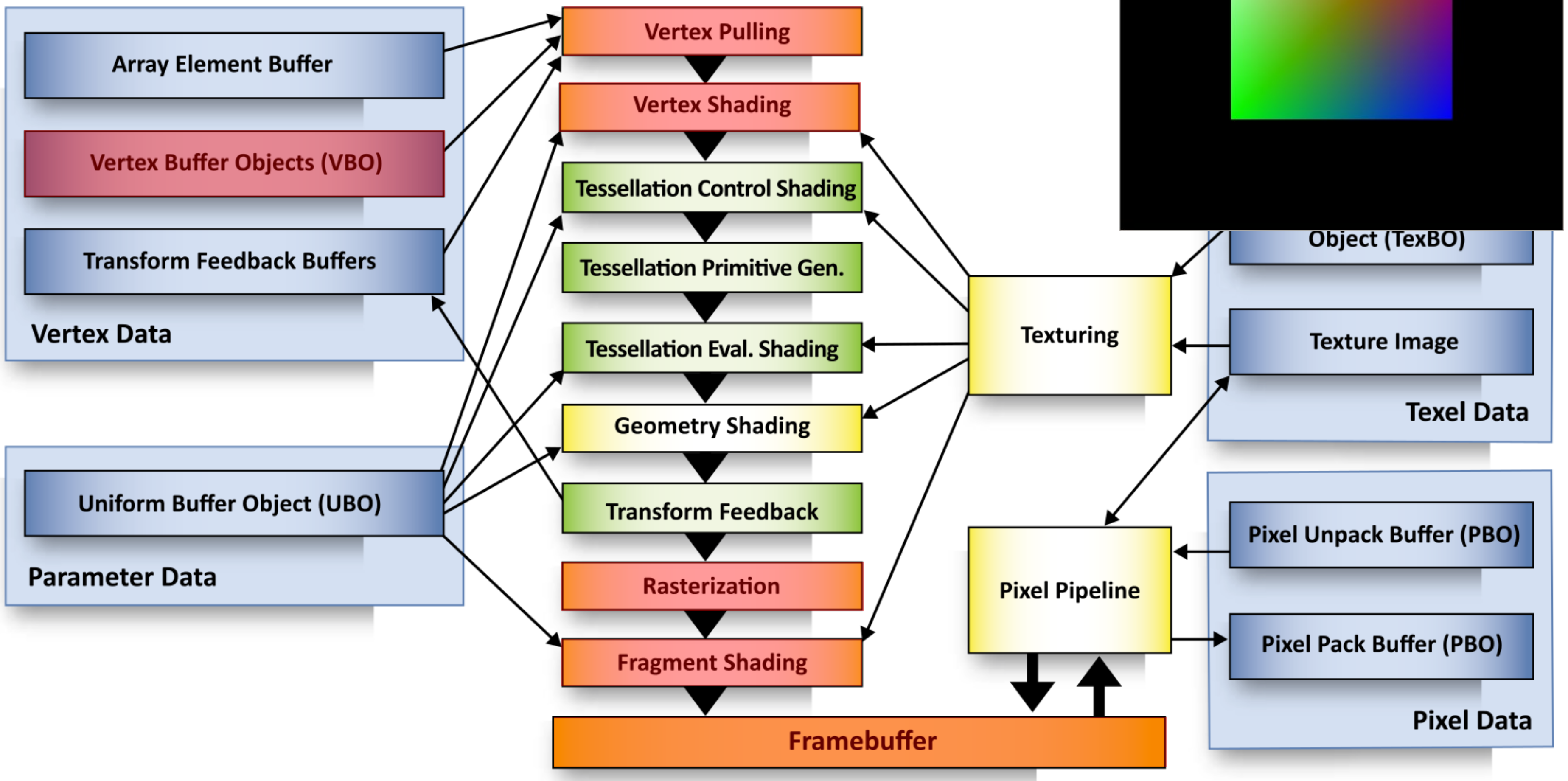
same name as in shader

Colored Quad – Host Code

```
// Draw
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(sp);
glBindVertexArray(va);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
glUseProgram(0);
glXSwapBuffers(display, win);

.
.
.

// Cleanup
glDeleteProgram(sp);
glDeleteVertexArrays(1, &va);
glDeleteBuffers(1, &vb);
glDeleteBuffers(1, &cb);
```

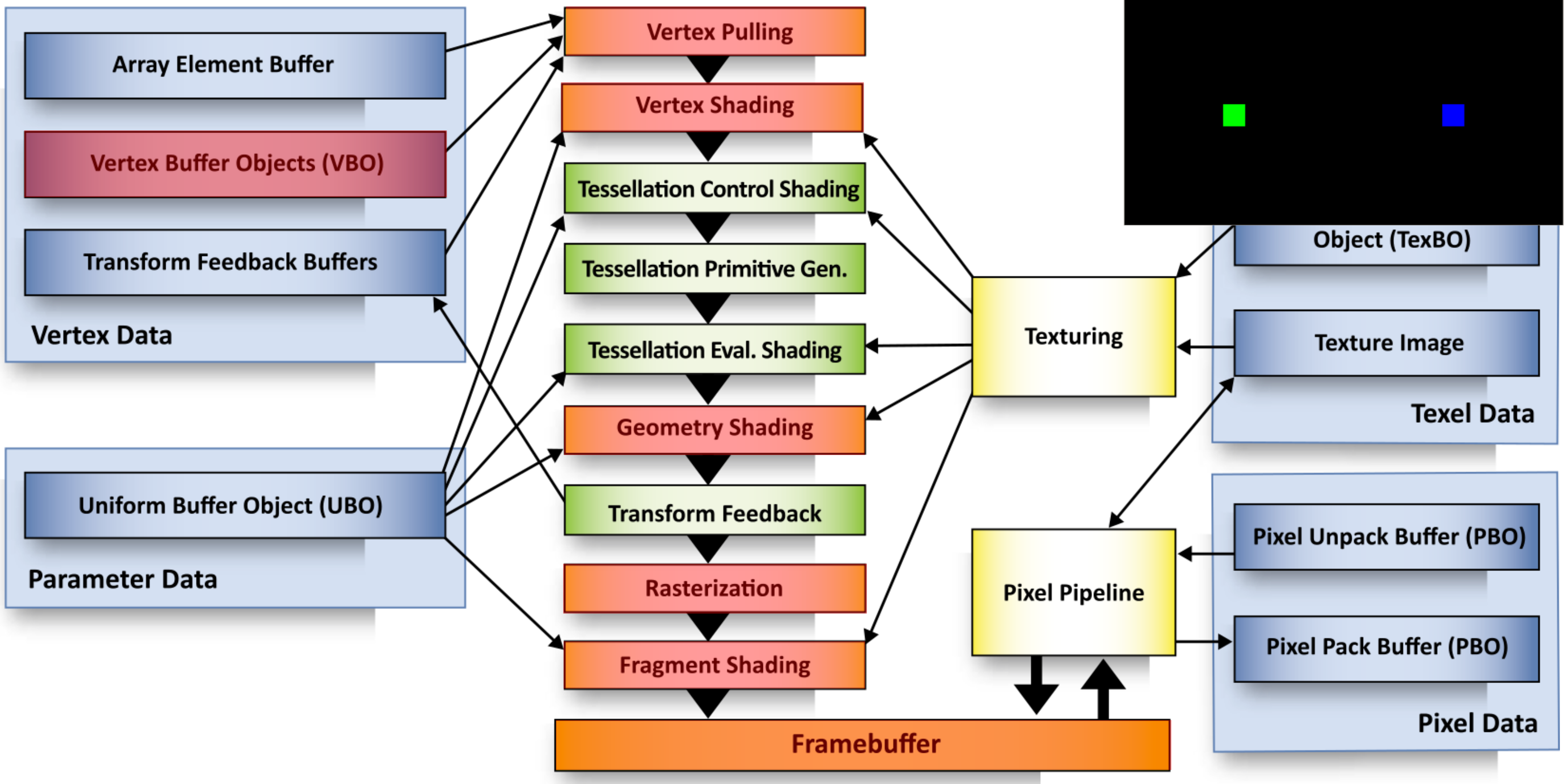
Colored Quad





Point Sprites

Point Sprites



Point Sprites – Shaders

- Geometry Shaders
  - Executed once per primitive (point, line, triangle)
  - Can access all vertex data of the whole primitive
  - May emit a flexible amount of primitives (bound by an implementation dependent maximum)

Point Sprites – Geometry Shader

```
#version 420 core
// multiple points in, 2 tringles out
layout(points) in;
layout(triangle_strip, max_vertices=6) out;

in VertexData
{
    vec3 Color;
} inData[];      // array as we want to access the color of all vertices

out VertexData
{
    vec3 Color;
} outData;

// built-in variable:  gl_Position

// some constants
vec4 ul = vec4(-0.05, 0.05, 0.0, 0.0);
vec4 ur = vec4(0.05, 0.05, 0.0, 0.0);
vec4 ll = vec4(-0.05, -0.05, 0.0, 0.0);
vec4 lr = vec4(0.05, -0.05, 0.0, 0.0);
```

Point Sprites – Geometry Shader

```
void main(void)
{
    // Output a quad consisting of two triangles acound each input point

    gl_Position = gl_in[0].gl_Position + ul;
    outData.Color = inData[0].Color;
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + ur;
    outData.Color = inData[0].Color;
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + ll;
    outData.Color = inData[0].Color;
    EmitVertex();

    EndPrimitive();

    gl_Position = gl_in[0].gl_Position + ur;
    outData.Color = inData[0].Color;
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + lr;
    outData.Color = inData[0].Color;
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + ll;
    outData.Color = inData[0].Color;
    EmitVertex();

    EndPrimitive();
}
```

Point Sprites– Host Code

```
// Create shader
GLuint sp = createShaderProgram("Shaders/colored2d.vs", NULL, NULL,
         "Shaders/colored2d.gs", "Shaders/colored2d.fs");
glUseProgram(sp);

// Create mesh
float mesh[] =
{    -0.5, 0.5, 0.5, 0.5, -0.5, -0.5, 0.5, -0.5};
float colors[] =
{    1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f};

GLuint vb;
glGenBuffers(1, &vb);
glBindBuffer(GL_ARRAY_BUFFER, vb);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 2 * 4, mesh, GL_STATIC_DRAW);

GLuint cb;
glGenBuffers(1, &cb);
glBindBuffer(GL_ARRAY_BUFFER, cb);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 3 * 4, colors,
         GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, 0);
```

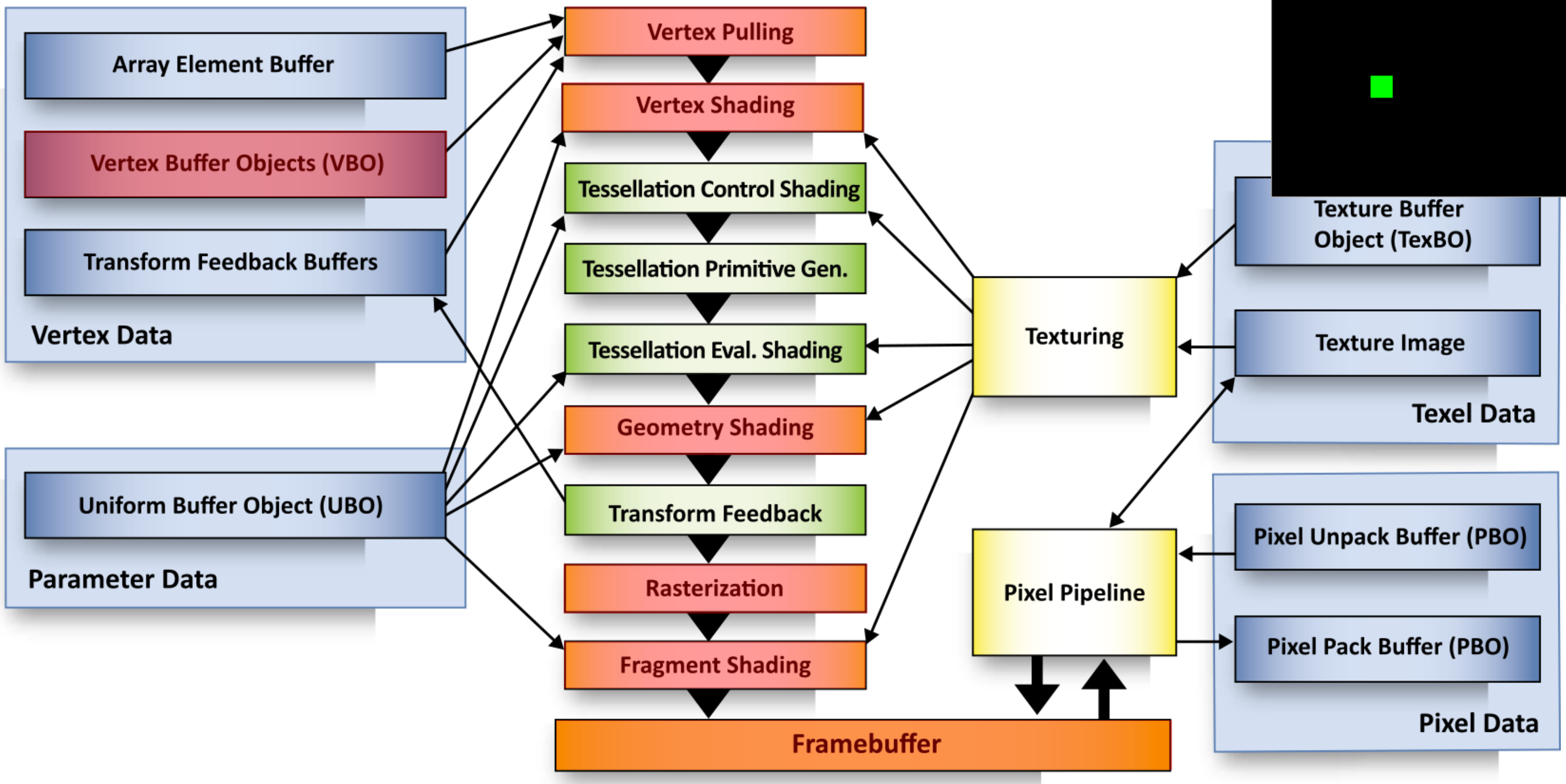
Point Sprites– Host Code

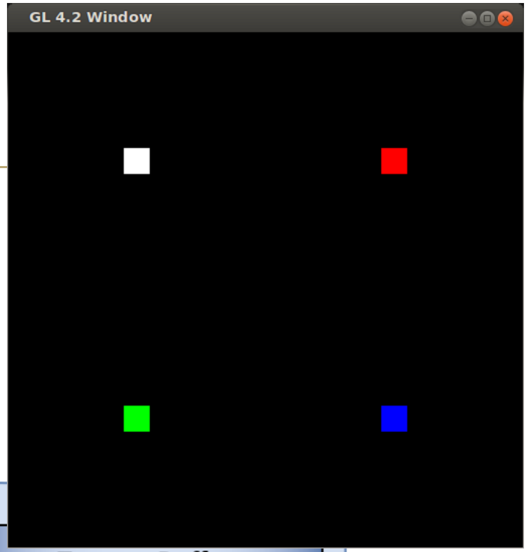

```
// Link mesh to vertex shader input variables
GLuint va;
glGenVertexArrays(1, &va);
glBindVertexArray(va);

glBindBuffer(GL_ARRAY_BUFFER, vb);
glEnableVertexAttribArray(glGetAttribLocation(sp, "inPosition"));
glVertexAttribPointer(glGetAttribLocation(sp, "inPosition"), 2, GL_FLOAT,
         GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, cb);
glEnableVertexAttribArray(glGetAttribLocation(sp, "inColor"));
glVertexAttribPointer(glGetAttribLocation(sp, "inColor"), 3, GL_FLOAT,
         GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Point Sprites– Host Code

```
// Draw
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(sp);
glBindVertexArray(va);
glDrawArrays(GL_POINTS, 0, 4);  // draw four points -> yields four quads
glBindVertexArray(0);
glUseProgram(0);
glXSwapBuffers(display, win);

// Cleanup
glDeleteProgram(sp);
glDeleteVertexArrays(1, &va);
glDeleteBuffers(1, &vb);
glDeleteBuffers(1, &cb);
```

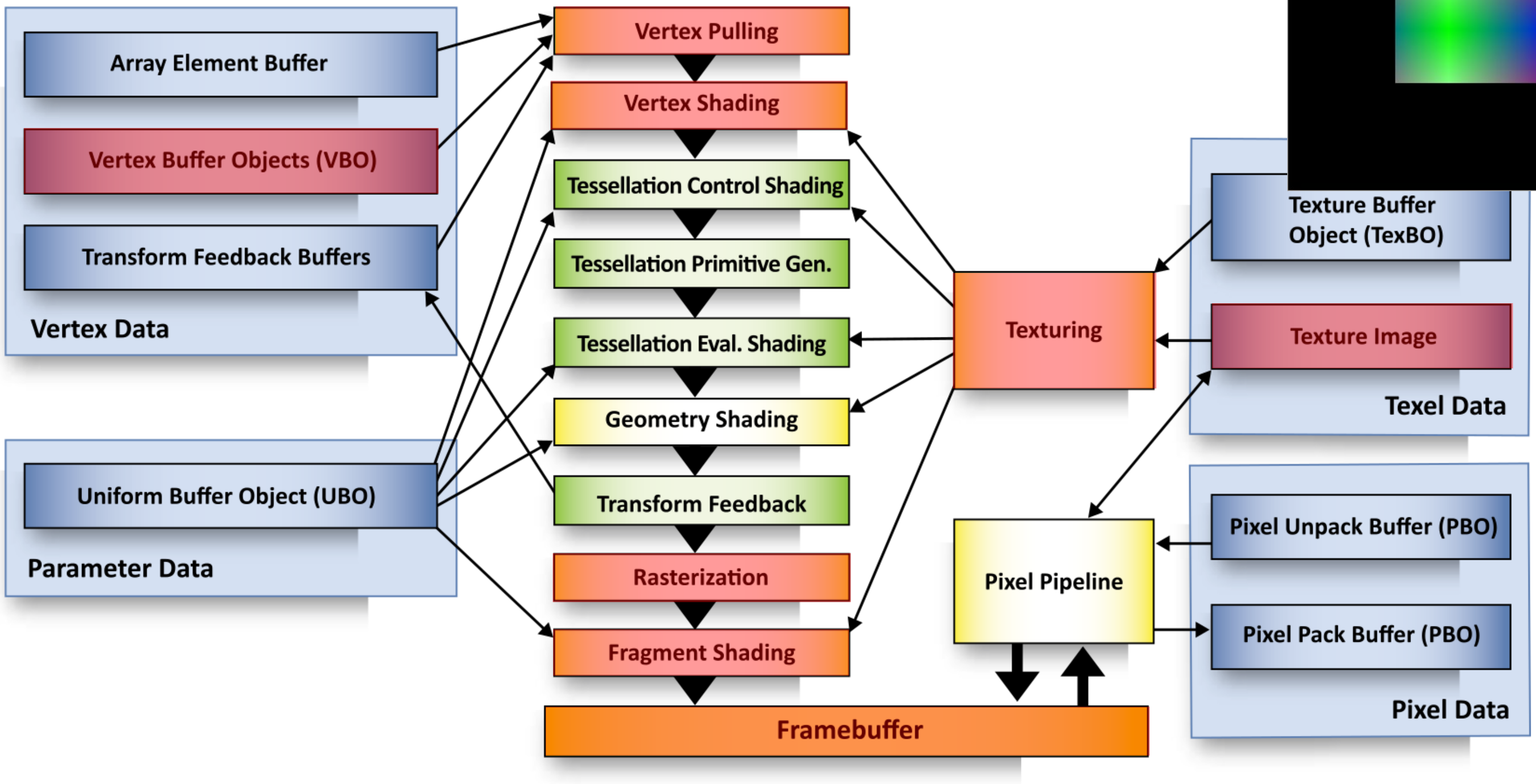
Point Sprites

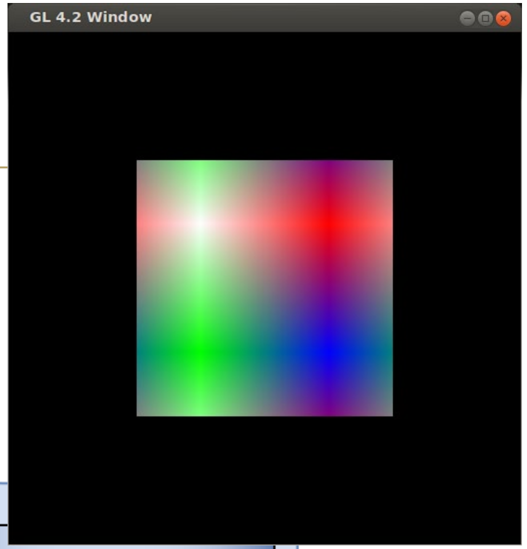




Textures

Textured Quad





Textured Quad - Idea

- Load texture
- Set up a texture sampler configuration
- Bind texture and sampler configuration to a texture unit

- Ship texture coordinates to the programmable pipeline with vertex data (as done with color before)
- Ship the texture unit number to the programmable pipeline as uniform variable

- The texture coordinates will be interpolated for each fragment during rasterization
- The fragment shader can use it directly to fetch the color from the texture.

Textured Quad – Vertex Shader

```
#version 420 core
in vec2 inPosition;
in vec2 inTexcoord;

out VertexData
{
    vec2 Texcoord;
} outData;

void main(void)
{
    gl_Position = vec4(inPosition, 0.0, 1.0);
    outData.Texcoord = inTexcoord;
}
```

Textured Quad – Fragment Sh.


```
#version 420 core
uniform sampler2D tex;

in VertexData
{
    vec2 Texcoord;
} inData;

out vec4 outColor;

void main(void)
{
    outColor = texture(tex, inData.Texcoord);
}
```

- `texture(tex, texcoord)` fetches a texture sample from the `tex`th texture unit at the given texture coordinates.

Textured Quad – Host Code

```
// Create shader
GLuint sp = createShaderProgram("Shaders/textured2d.vs", NULL, NULL, NULL,
        "Shaders/textured2d.fs");

// Create mesh
float mesh[] =
{   -0.5, 0.5, 0.5, 0.5, -0.5, -0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5};
float texCoords[] =
{   0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f};

GLuint vb;
glGenBuffers(1, &vb);
glBindBuffer(GL_ARRAY_BUFFER, vb);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 2 * 6, mesh, GL_STATIC_DRAW);

GLuint tb;
glGenBuffers(1, &tb);
glBindBuffer(GL_ARRAY_BUFFER, tb);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 2 * 6, texCoords,
        GL_STATIC_DRAW);
```

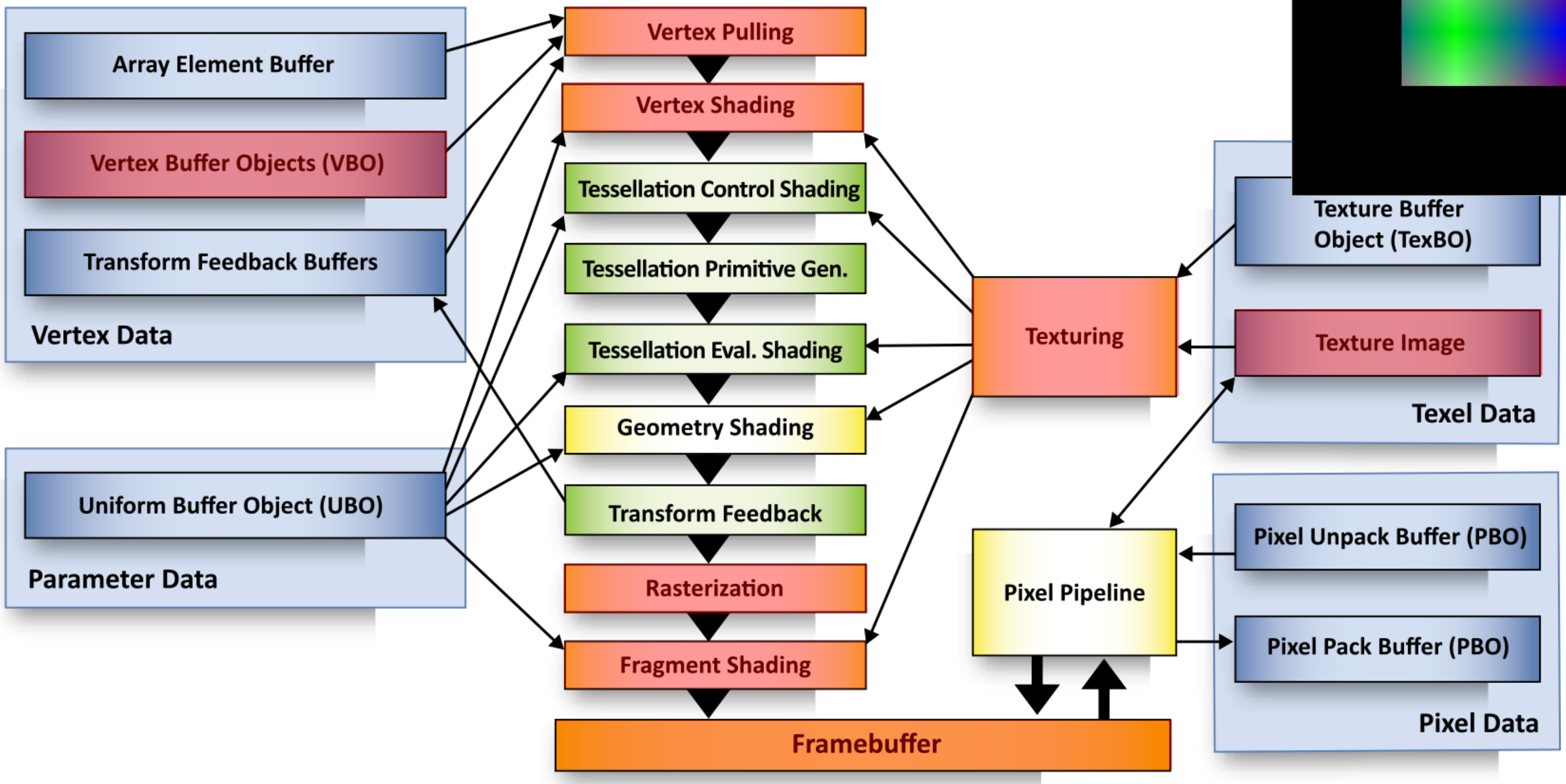
Textured Quad – Host Code

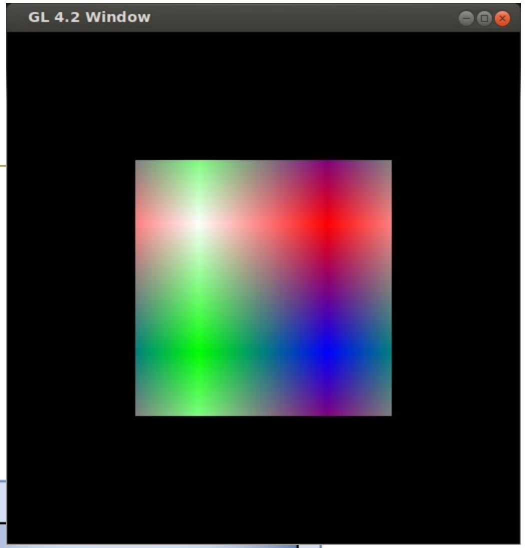
```
// Link mesh to vertex shader input variables
GLuint va;
glGenVertexArrays(1, &va);
glBindVertexArray(va);

glBindBuffer(GL_ARRAY_BUFFER, vb);
glEnableVertexAttribArray(glGetAttribLocation(sp, "inPosition"));
glVertexAttribPointer(glGetAttribLocation(sp, "inPosition"), 2, GL_FLOAT,
         GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, tb);
glEnableVertexAttribArray(glGetAttribLocation(sp, "inTexcoord"));
glVertexAttribPointer(glGetAttribLocation(sp, "inTexcoord"), 2, GL_FLOAT,
         GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

// Create the texture
float texels[] =
{    1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f};
GLuint tex;
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 2, 2, GL_FALSE, GL_RGB, GL_FLOAT,
         texels);
glGenerateMipmap(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, 0);
```

Textured Quad – Host Code

```
// Create and configure the texture sampler
GLuint sampler;
glGenSamplers(1, &sampler);
glSamplerParameteri(sampler, GL_TEXTURE_WRAP_S, GL_REPEAT);
glSamplerParameteri(sampler, GL_TEXTURE_WRAP_T, GL_REPEAT);
glSamplerParameteri(sampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(sampler, GL_TEXTURE_MIN_FILTER,
         GL_LINEAR_MIPMAP_LINEAR);

// Draw
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(sp);
glBindTexture(GL_TEXTURE_2D, tex);
glBindSampler(0, sampler);
glUniform1i(glGetUniformLocation(sp, "tex"), 0);
glBindVertexArray(va);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
glBindSampler(0, 0);
glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
glXSwapBuffers(display, win);

// Cleanup
glDeleteSamplers(1, &sampler);
glDeleteTextures(1, &tex);
glDeleteProgram(sp);
glDeleteVertexArrays(1, &va);
glDeleteBuffers(1, &vb);
glDeleteBuffers(1, &tb);
```

Textured Quad





Debugging

- Use an OpenGL Debugging Tool
  - BuGLe
  - gDEBugger
  - GLIntercept
  - …
- Write `GL_CHECK_ERROR` macro after EVERY gl* function call, define it like this:

```
void glCheckError(const char* file, unsigned int line)
{
    unsigned int error = glGetError();
    if (error != 0)
    {
        std::cout << "OpenGL error " << error << " in file " << file
                  << ", line " << line << std::endl;
        exit(1);
    }
}

#ifdef DEBUG
#define GL_CHECK_ERROR glCheckError(__FILE__, __LINE__)
#else
#define GL_CHECK_ERROR
#endif
```

Online Resources

http://www.khronos.org

- Offical home
- **Quick Reference: http://www.khronos.org/files/opengl42-quick-reference-card.pdf**

http://www.opengl.org

- start here; up to date specification and lots of sample code
- **Reference pages: http://www.opengl.org/sdk/docs/man/**

http://www.mesa3d.org/

- Brian Paul’s Mesa 3D (OpenGL in Software)

http://developer.nvidia.com

- Lots of examples, tutorials, tips& tricks

http://www.ati.com/developer/

- Lots of examples, tutorials, tips& tricks

http://www.sgi.com/software/opengl

- For historic purposes :-) .... but no longer active now

Books

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- OpenGL Programming Guide, 3rd Edition
- OpenGL Reference Manual, 3rd Edition
- OpenGL Programming for the X Window System
  - includes many GLUT examples
- Interactive Computer Graphics: A top-down approach with OpenGL, 2nd Edition

Overview

- Last lecture:
  - Rasterization
  - Clipping

- Today:
  - OpenGL
    - Vertex shader
    - Geometry shader
    - Fragment shader
    - Textures

- Next lecture:
  - Advanced OpenGL features