Computer Graphics

- OpenGL-

Hendrik Lensch

Overview

- **Last lecture:**
  - Rasterization
  - Clipping
- **Today:**
  - OpenGL

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
  - 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 128 SIMD processors, >>80 GB/s memory access
    - Up to 1GB 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
    - Only real alternative today to Microsoft’s Direct3D

Introduction to OpenGL

- **What is OpenGL?**
  - Software interface for graphics hardware (API)
    - 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, ...
  - Focused on ***immediate-mode*** operation
    - Thin hardware abstraction layer – almost direct access to HW
    - Triangles as base primitives – directly submitted by application
    - More efficient batch processing with vertex arrays (and display lists)
  - Network-transparent protocol
    - GLX-Protocol – X Window extension (only in X11 environment!)
    - Direct (hardware access) versus indirect (protocol) rendering

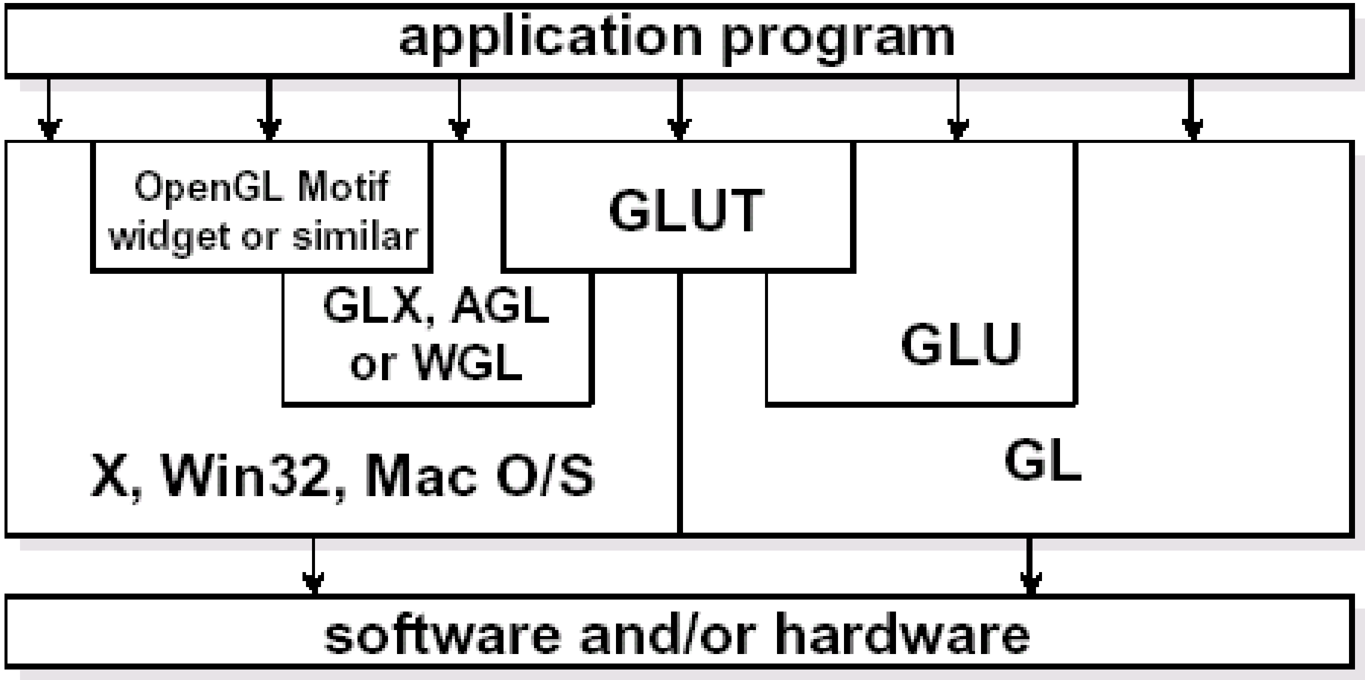
Introduction to OpenGL

- **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 ARP
      - No central control (by MS)
      - Could accelerate innovation
  - „No“ subsets (only one, plus many, many extensions :-)
    - Capabilities are well defined (but may not all be HW accelerated)
    - Exception: Imaging subset (and extensions)
    - But now OpenGL ES (for embedded devices)

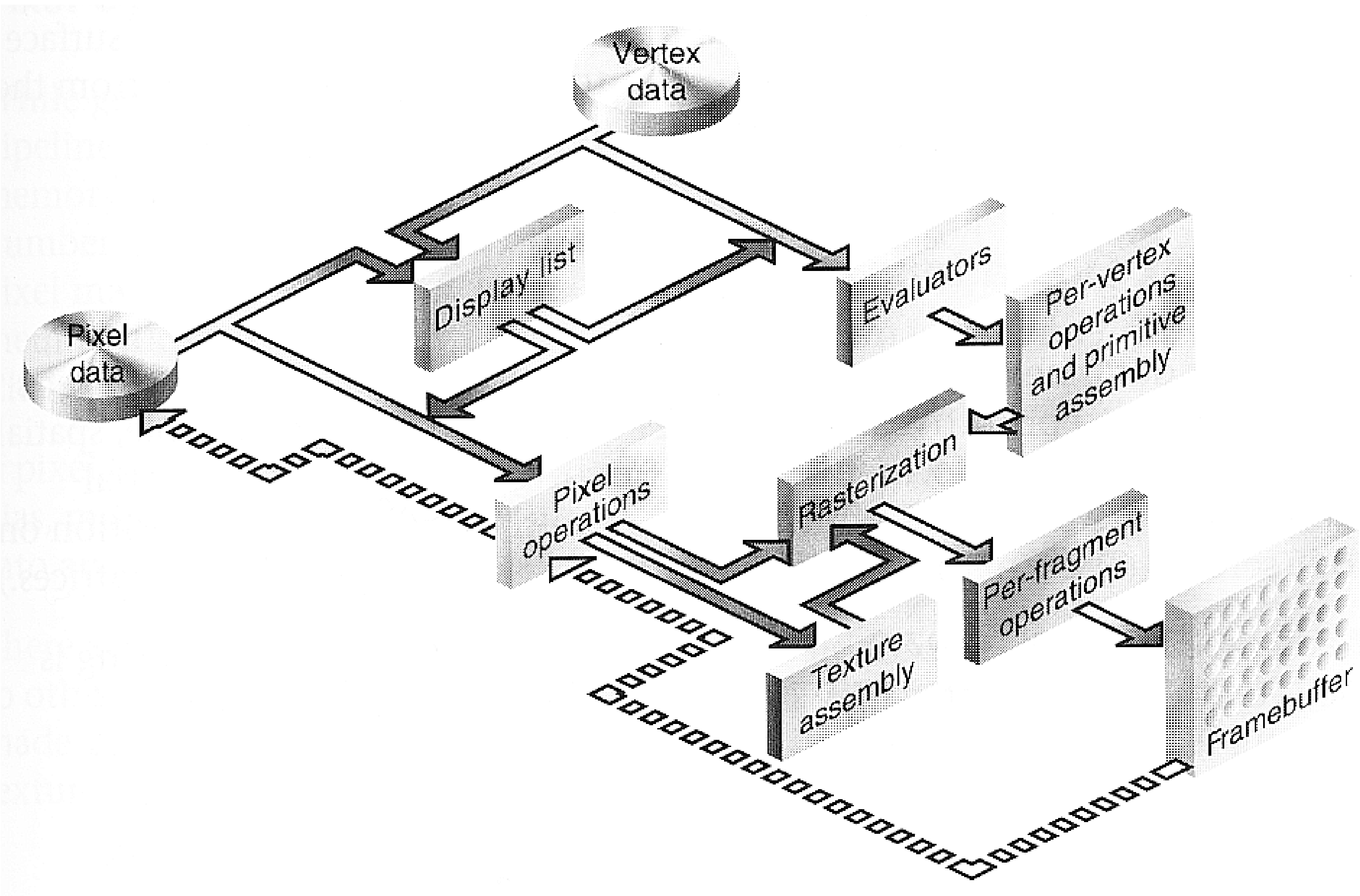
Related APIs

- **AGL, GLX, WGL**
  - glue between OpenGL and windowing systems
- **GLU (OpenGL Utility Library)**
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- **GLUT (OpenGL Utility Toolkit)**
  - portable windowing API
  - not officially part of OpenGL

OpenGL and related APIs



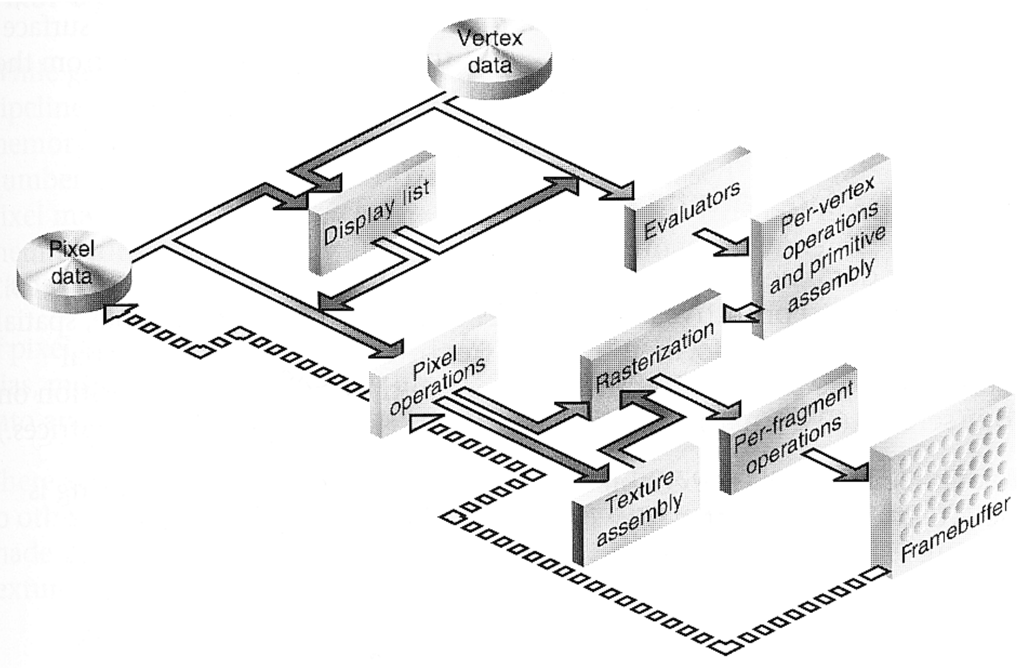
Overview



Vertex data
Pixel data
Display list
Evaluators
Per-vertex operations and primitive assembly
Rasterization
Pixel operations
Texture assembly
Per-fragment operations
Framebuffer

OpenGL Rendering

- **Geometric primitives**
  - Points, lines and polygons
- **Image primitives**
  - Images and bitmaps
- **Separate pipeline for images and geometry**
  - Linked through texture mapping
- **Rendering depends on state**
  - Colors, materials, light sources, etc.
- **Immediate Mode Rendering**



Immediate Mode Rendering

```
void initialize() {}
void redraw() {
   for(int i=0; i<N; i++)
      draw_primitive(i);
}
```

Immediate Mode

```
void initialize() {
   for(int i=0; i<N; i++)
      load_primitive(i);
}
void redraw() {}
```

Retained Mode

- **Immediate Mode**
  - Application maintains scene data
  - Execute drawing commands whenever window is repainted
- **Retained Mode**
  - Graphics system maintains scene data and handles redraw
  - OpenGL provides some retained mode functionality:
    - Display Lists: encapsulate and optimize immediate mode stream
    - Vertex Arrays: pass large array of geometry data in one function call
    - Vertex Buffer Objects: like vertex arrays with less overhead

OpenGL-Concepts

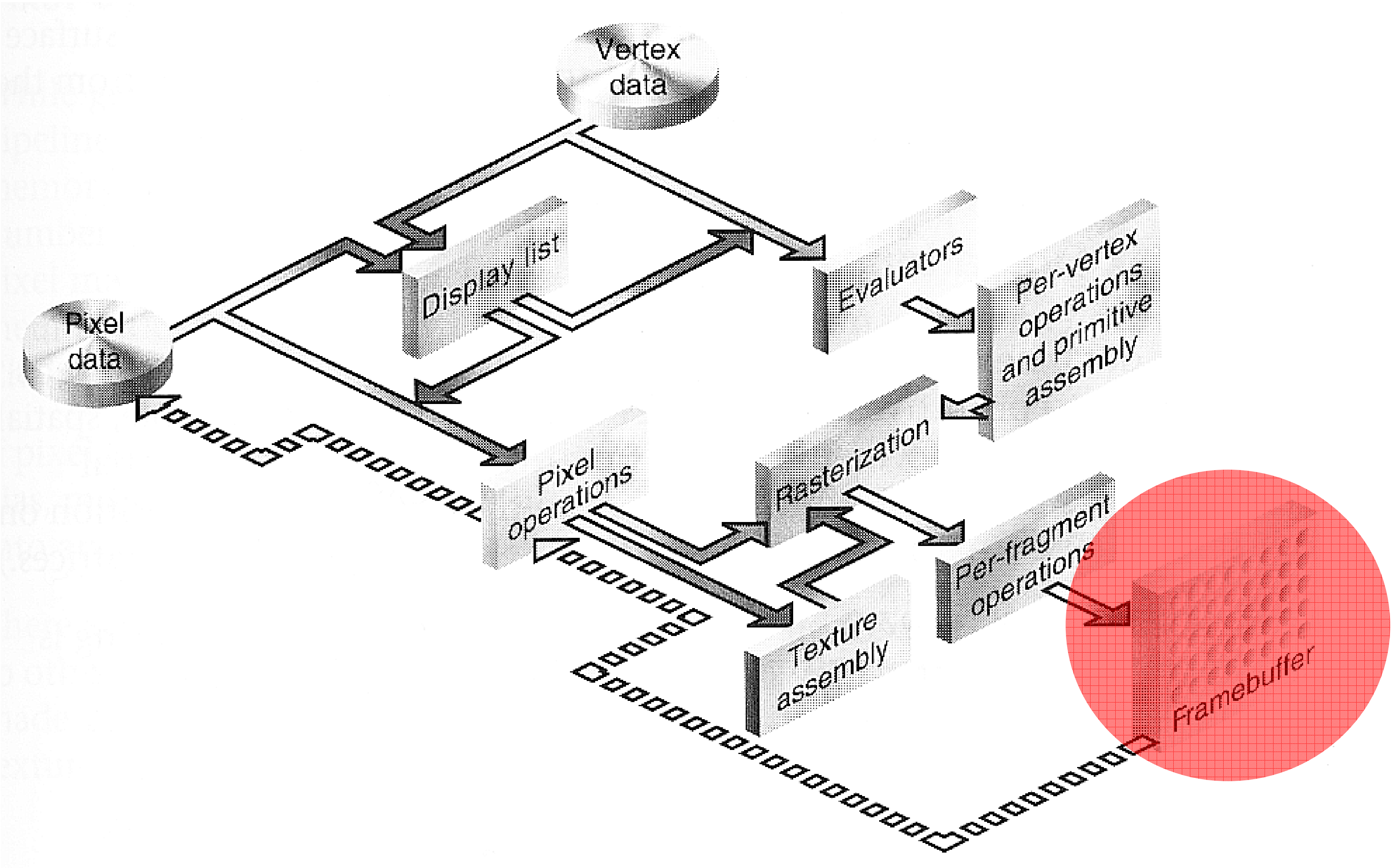
- **Rendering context**
- **Buffer**
- **Vertex operations**
- **Raster operations**
- **Rasterization**
- **Fragment operations**

- **Terminology: pixel, texel, and fragments**
  - Pixels are elements of the frame buffers (picture element)
  - Texels are elements of textures (images applied to geometry)
  - Fragments are
    - the output of rasterization and
    - the input to frame buffer operations (finally generating pixels)

OpenGL Rendering Context

- **Context**
  - Analogy: drawing tool
  - Maintains the OpenGL state that is applied to all later geometry
  - Must be compatible with underlying Window/Drawable
  - Always one current context (per thread)
- **Direct/indirect context**
  - Direct: Rendering directly to hardware (no GLX protocol)
    - Fallback to indirect rendering if no direct access is possible
  - Indirect: Rendering via network protocol GLX
    - limited to host's capabilities
- **Sharing between contexts**
  - Joint storage and usage von textures and display lists
- **Access to rendering context**
  - glXCreateContext()/glXDestroyContext
  - glXMakeCurrent()

OpenGL and Buffers



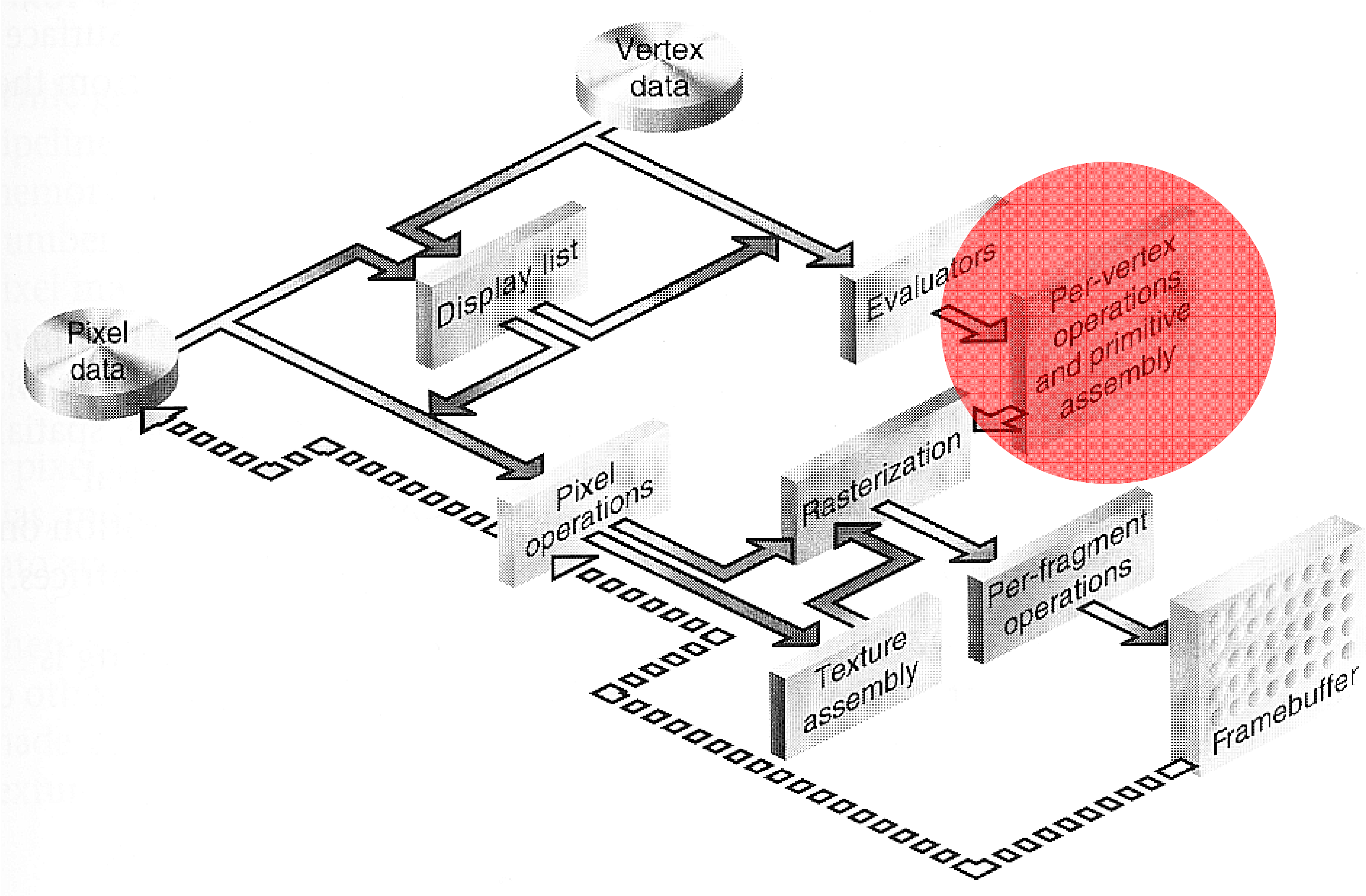
OpenGL and Buffers

- **OpenGL buffers**
  - Provide memory for storing data for every pixel
    - Color, depth (Z), stencil, accumulation, (window-id), and others
  - Format must be fixed before windows are opened
    - Window-System specific: glXGetConfig
- **Color buffers**
  - RGBA (RGB+Alpha) or index into a color table (hardly used)
    - *Alpha* stores transparency/coverage information
    - Today often 8/8/8(/8) bits
    - Latest chips also support 16 bit fix and 16/24/32 bit float components
  - Double buffering option (back- und front buffer)
    - Animations: draw into back, display front
    - Swap buffers during vertical retrace (glXSwapBuffers)
      - No flashing or tearing artifacts during display
  - Stereo option
    - Left and right buffers (also with DB), e.g. for two projectors
    - Requires support from GUI

OpenGL and Buffers

- **Depth/Z buffer**
  - Stores depth/Z coordinate of visible geometry per pixel
  - Used for occlusion test (Z-test)
- **Stencil buffer**
  - Small integer variable per pixel
  - Used for masking fragment operations
  - Write operations based on fragment tests
    - Set/increment/decrement variable
- **Accumulation buffer**
  - RGBA buffer with many bits per pixel (now obsolete with floats)
  - Supports special operations on entire images
    - glAccum(): weighted addition, multiplication
- **Other buffers**
  - Aux-buffers, window-ID buffers, off-screen buffers, P-buffers, DM-buffers, T-buffers, ...

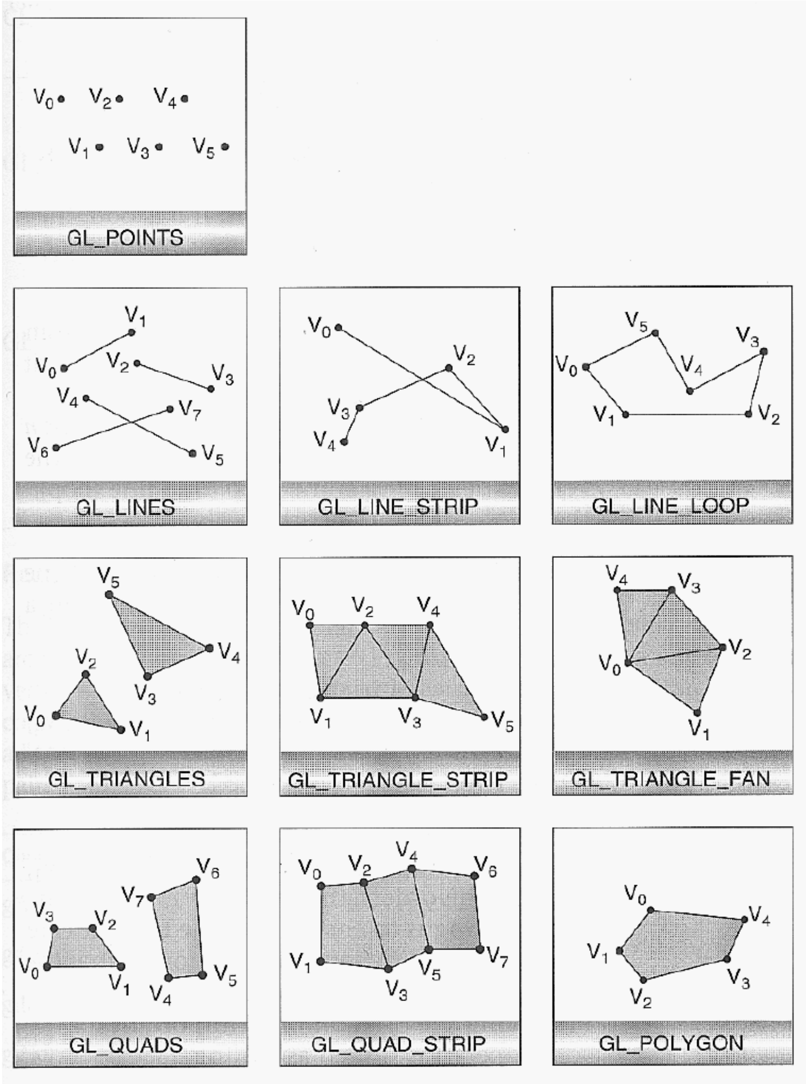
Overview



Vertex data
Pixel data
Display list
Evaluators
Per-vertex operations and primitive assembly
Rasterization
Pixel operations
Per-fragment operations
Texture assembly
Framebuffer

OpenGL Geometrie

- **Primitive:**



Vertex Operations

- **Sequence of Vertex Operations**
  - Input to vertex operations are vertices
    - Position, normal, colors, texture coordinates, …
  - Transformation of geometry with the model-view matrix (3D→3D)
  - Shading: Lighting computation can generate per vertex colors
  - Perspective projection: perspective transformation to 2-1/2D
  - Optional: generation of texture coordinates
  - Primitive assembly: generating primitives from vertices
  - Clipping: Cutting off off-screen parts of geometry
  - Back face culling: dropping geometry facing the wrong way
  - Output of vertex operations are primitives with vertex data
    - Position (2D plus Z), color, texture coordinates
    - Fed to rasterization unit

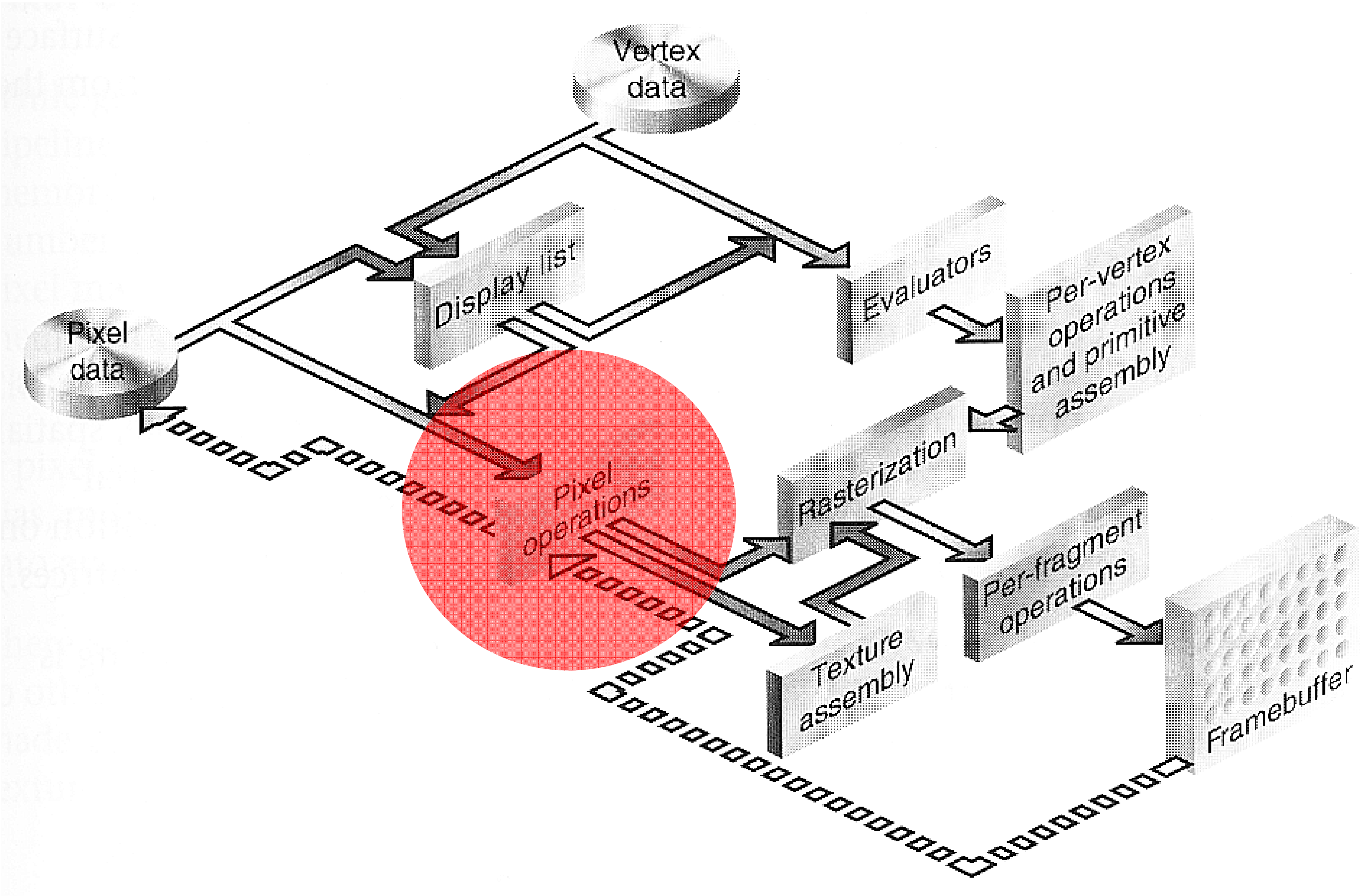
Shading

- **Lighting computation**
  - Definition of light sources
    - Position, direction, distance falloff, directional cutoff & exponent
    - Ambient, diffuse, specular, and emission color
      - Extended Phong model
  - Computes color for all vertices
    - Without lighting: directly specified by glColor()
    - With lighting: Determined by lighting computation from parameters
      - Light source, vertex colors, material/Phong, light model
- **Light source parameter**
  - glLightfv(GL_LIGHT0, GL_DIFFUSE, color4); // RGBA
  - glLightfv(GL_LIGHT0, GL_POSITION, pos4); // homogen
  - glEnable(GL_LIGHT0);
  - glEnable(GL_LIGHTING);
  - Light source parameter are part of the OpenGL state

Shading

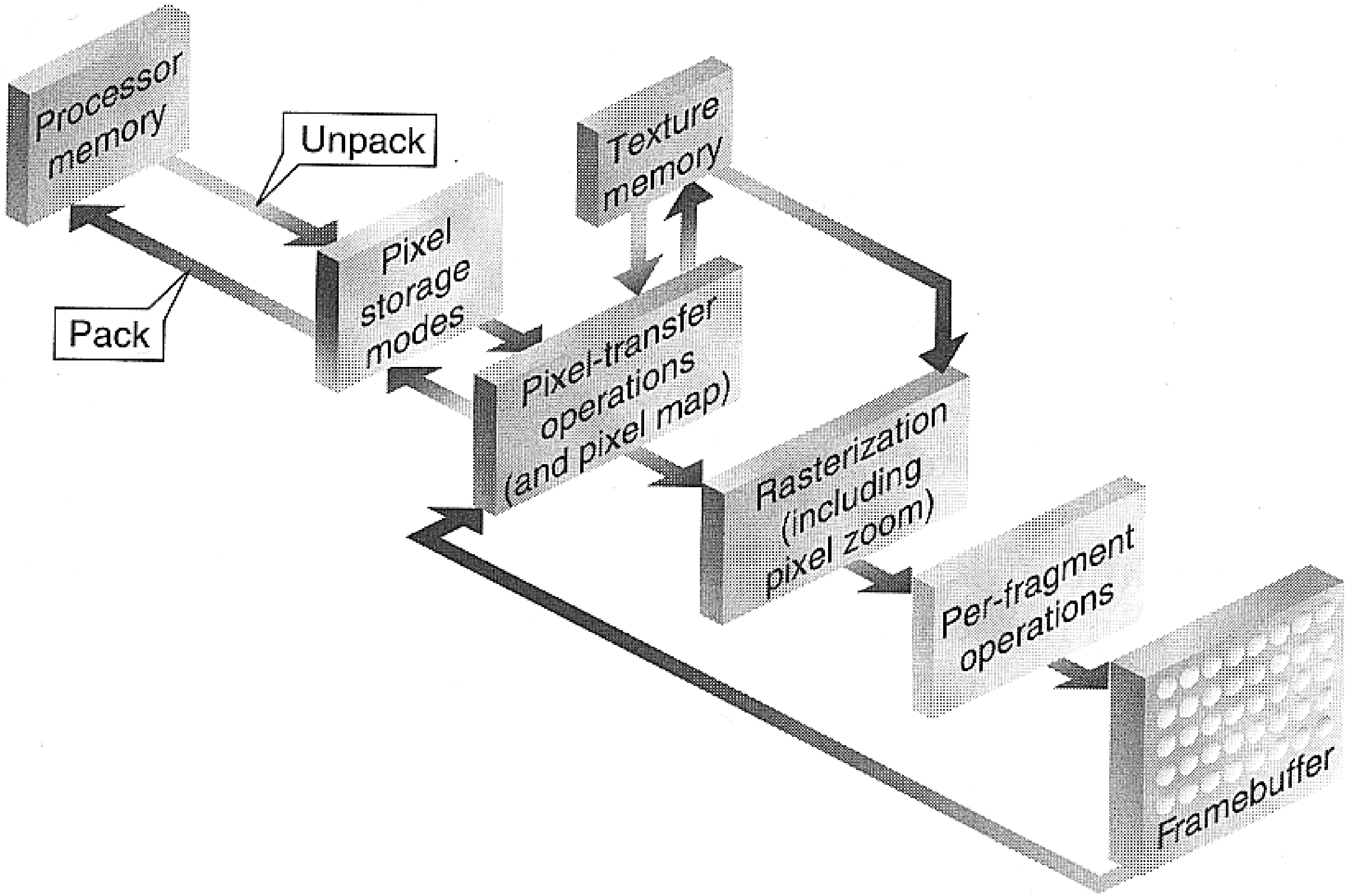
- **Material parameter**
  - glColor() sets both ambient and diffuse color by default
  - glMaterial{if}[v](GL_FRONT, GL_DIFFUSE, color4); …
  - glShadeModel(model);
    - GL_FLAT: constant color (defined by last vertex)
    - GL_SMOOTH: linear interpolation of color across primitive
  - Material and light parameter are only used by lighting
- **Changing material parameters**
  - Calling glMaterial() between two vertices (can be expensive)
  - Optimization: Bind glColor() to specific material parameter
    - glColorMaterial(GL_FRONT_AND_BACK, GL_SPECULAR);
      - Ambient, diffuse, specular, ambient & diffuse, and emission
    - Default: Ambient and diffuse
    - Must be enabled by glEnable(GL_COLOR_MATERIAL);

Overview



Vertex data
Pixel data
Display list
Evaluators
Per-vertex operations and primitive assembly
Rasterization
Pixel operations
Texture assembly
Per-fragment operations
Framebuffer

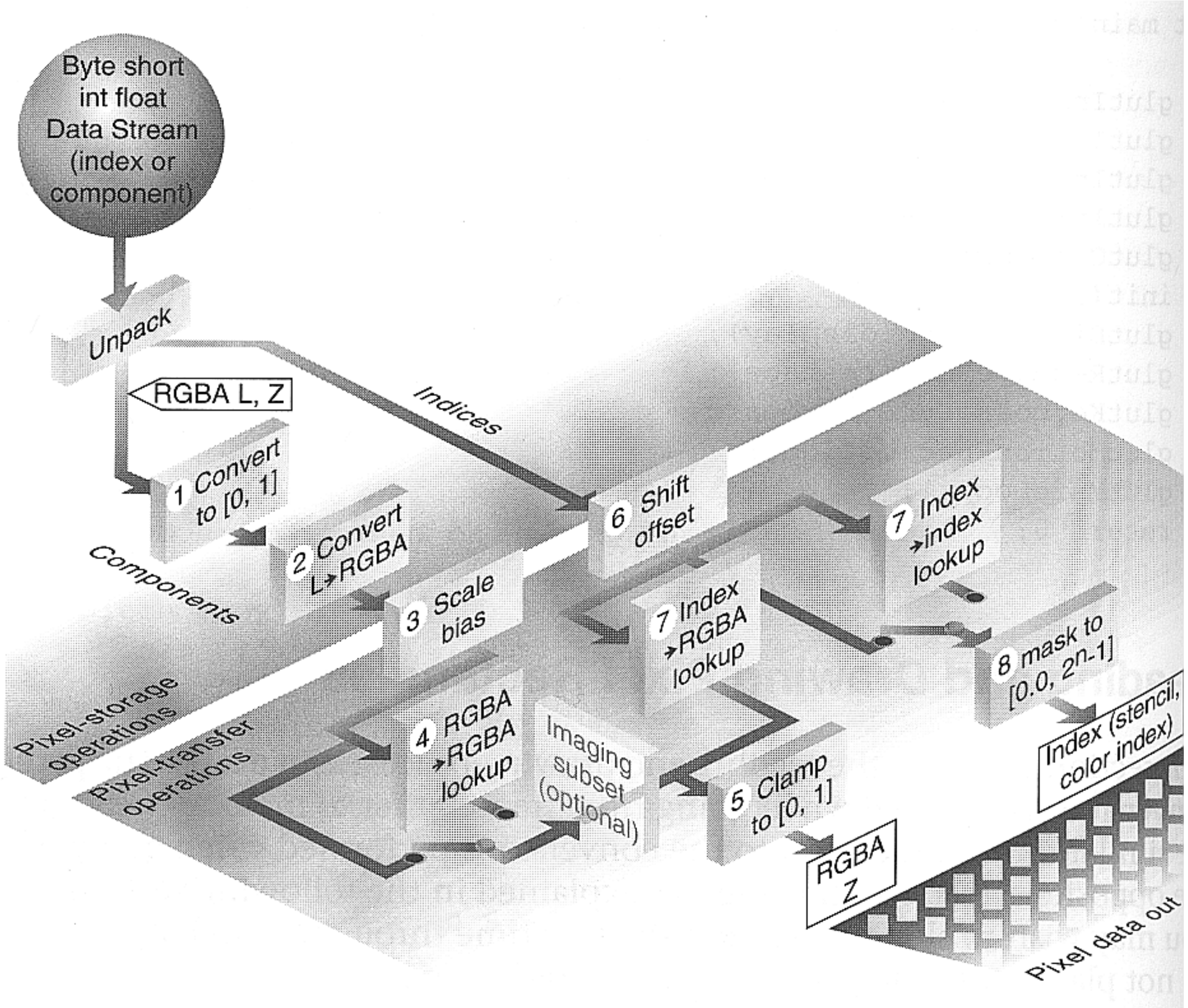
Pixel Operations



Pixel Operations

- **Pixel storage operations**
  - Conversion from/to external formats in main memory
    - Reformatting, Mapping gray tones $\leftrightarrow$ RGBA
  - glDrawPixels(), glReadPixels(), ...
- **Pixel transfer operations**
  - Scaling, offset, table lookup, clamping, etc.
  - Optional Imaging Subset
    - Additional lookup tables, convolution, color matrix, histogram, minmax
  - Applied during pixel transfer to rasterizer, texture memory, or main memory
- **Copying pixels**
  - Operations apply only during write stage
  - glCopyPixels(), glCopyTexImage(), ...

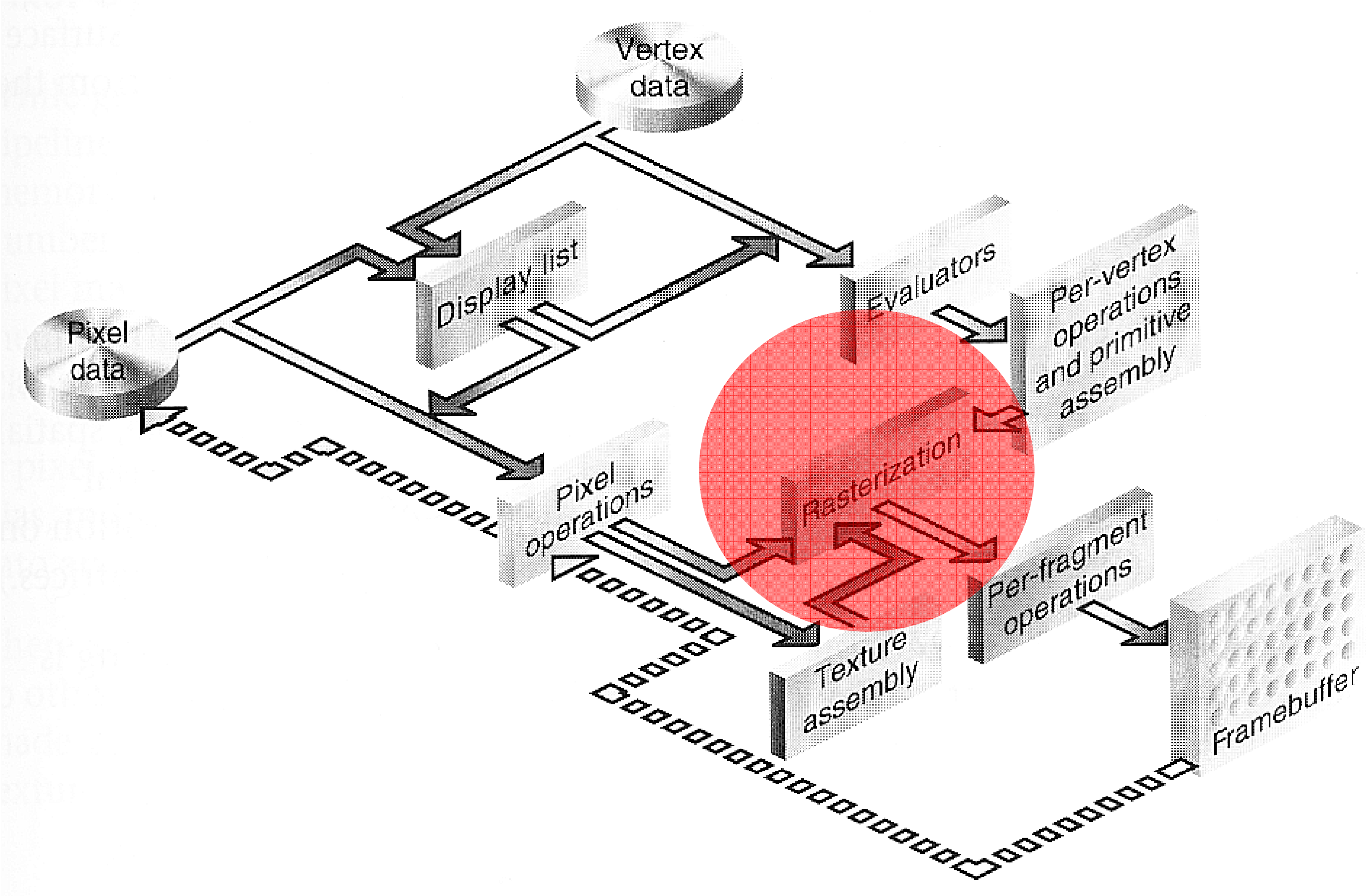
Pixel Operations



Pixel Operations

- **Performance remarks**
  - All standard OpenGL operations also apply to pixel data
    - E.g. rasterization & fragment operations
  - Drawing pixels can be very costly
  - Any unnecessary operations should be disabled
  - Natives formats should be used wherever possible

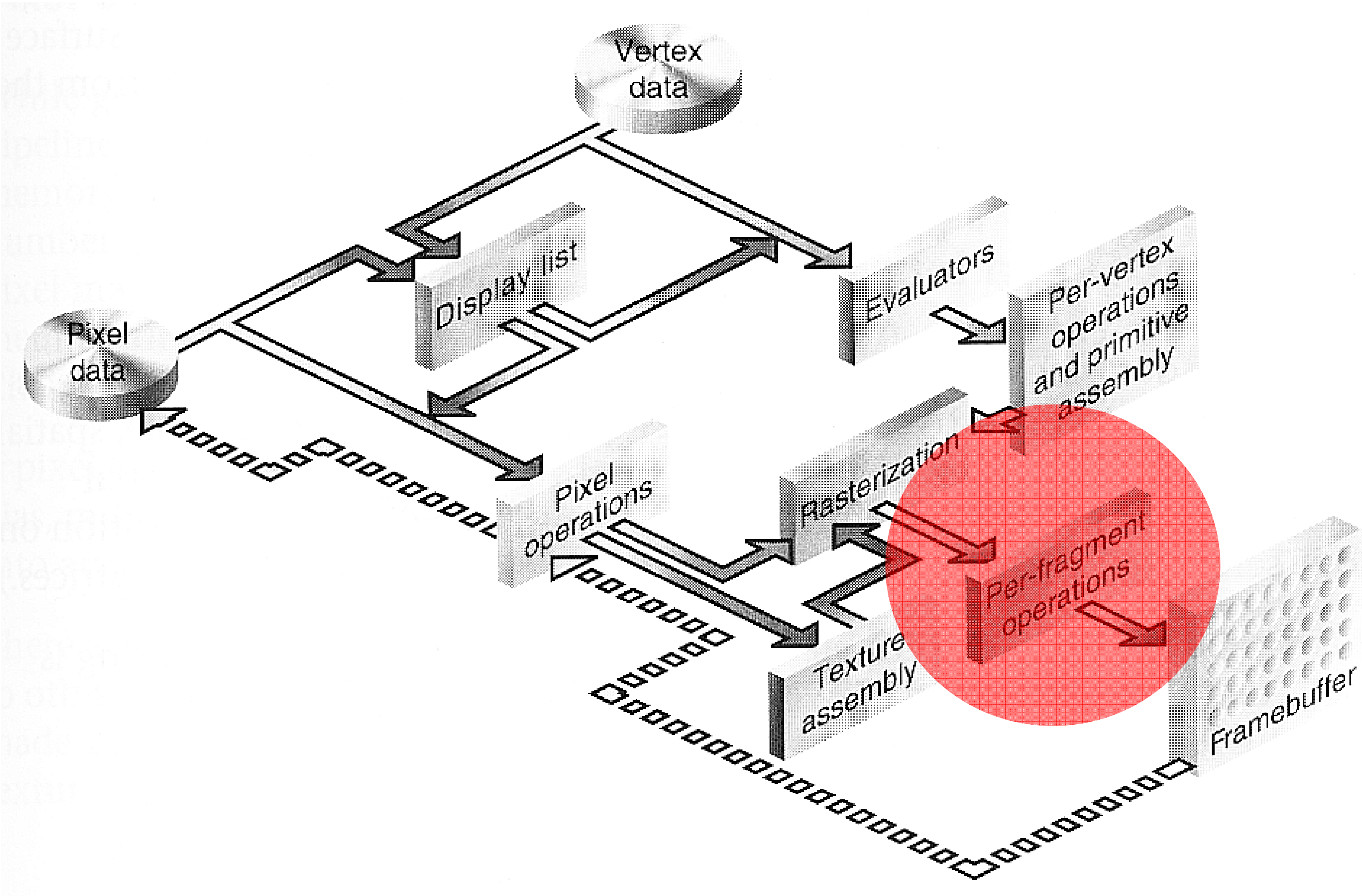
Overview


Vertex data
Pixel data
Display list
Evaluators
Per-vertex operations and primitive assembly
Rasterization
Pixel operations
Texture assembly
Per-fragment operations
Framebuffer

Rasterization

- **Rasterization:**
  - Generating fragments from geometric primitives
    - For every covered pixel
    - Determining fragment data
      - location, colors, texture coordinates, depth, …
    - Pixel data is also rasterized similarly
  - Applications of textures happens in a separate step
    - In modern card considered part of the fragment operations
- **Strict ordering**
  - Primitives are rasterized as they proceed through the pipeline
    - “Immediate mode rendering“
  - Pipeline may actually consist of multiple parallel pipelines
  - Primitives must be rasterized in order as send by application
    - Requires synchronization between pipelines
    - Complicates scalability questions

Overview


Vertex data
Pixel data
Display list
Evaluators
Per-vertex operations and primitive assembly
Pixel operations
Rasterization
Per-fragment operations
Texture assembly
Framebuffer

Fragment Processing

- **Consists of three sub-steps**
  - Fragment operations
    - Perform operations on fragments including texturing
  - Fragment test
    - Cull fragments conditionally
  - Blend operations
    - Merge fragments with content of the frame buffer

Fragment Operations

- **Much innovation in this part of the pipeline**
  - Simple texture mapping
    - Lookup of texel values
      - Requires memory access: Can potentially stall the pipeline
      - Requires careful design of graphics architecture
  - Fully programmable shading
    - Can use GPU for general purpose computation (“GPGPU”)
    - Predefined input an output registers
    - Exposes general assembly language for fragment operations
    - Various higher level shading languages (e.g. Cg, HLSL, GLSL)

Fragment Tests

- **Scissor test**
  - Culls fragments not in a 2D box on screen
- **Alpha test**
  - Compares fragment alpha with a constant
  - Culls fragments conditionally
- **Stencil test**
  - Compares value of stencil buffer with reference constant
  - Culls fragments conditionally
  - Can apply different operation to stencil value based mode
    - Stencil-fail/S-pass & Z-fail / S-pass & Z-pass
    - Operations: Set, increment, decrements, …
- **Depth test (visibility/occlusion test)**
  - Compares Z value with value from Z-buffer
  - Culls fragments conditionally, otherwise updates Z-buffer

Fragment Tests

- **Fragment tests**
  - Require per pixel read operations (high bandwidth)
  - May require per pixel write operations (stencil and Z-test)
    - Read-Modify-Write operations
    - Again synchronization issues with multiple pipelines
  - Tests occur late in the pipeline
    - Might have spend significant processing on the data already
    - Should perform tests earlier without violating OpenGL semantics
- **Occlusion culling**
  - At application level
    - Replicated visibility computation in the application (mostly coarse)
    - Avoids bandwidth to graphics engine completely, but uses CPU
  - Early Z test after rasterization
    - Can cull is fragments if known to be occludes (some addition cost)
    - Used bandwidth in upper pipeline already

Blend Operations

- **Merge fragments with frame buffer content**
- **Order of operations**
  - Blending operations (aka. compositing)
    - Weighted combination of fragment and pixel values
  - Dithering operation
    - Approximation of color by spatial averaging
    - Different rounding based pixel location
      - „Half-Toning“
  - Logical operations
    - 16 combinations of fragment and pixel values
      - NOT, AND, OR, XOR

OpenGL Guaranties

- **Non Guaranties**
  - No exact rule for implementation of graphics operations
    - Number of bits, coverage by a primitive, etc.
  - Different implementations can differ on a per-pixel basis
- **Invariants**
  - Invariants within an implementation
    - Same output when given the same input
    - Fragment values are independent of
      - Content of frame buffer
      - Active color buffer, ...
    - Independence of parameter values (e.g. for stencil / blending)
  - No invariance when switching options on and off
    - E.g. stencil, texturing, lighting, ...
    - On-screen versus off-screen buffers

OpenGL as an Instruction Set

- **Equivalence**
  - Frame buffer — Accumulator
  - Textures — Memory
  - Vertex/Fragment-Ops — ALUs (pipelined)
  - OpenGL-State — VLIW-Instruction
  - Geometry — Arguments
- **Example: Adding two vectors/arrays (as images)**
  - Render image A into frame buffer
  - Copy frame buffer $\rightarrow$ texture (glCopyTexImage)
  - Render image B into frame buffer
  - Render rectangle with texture into frame buffer
    - Use fragment operations (blending) to add fragments to pixels
  - Multi-pass computation
- **Mostly replaced by expressive shader support**

OpenGL Programming

OpenGL Preliminaries

Header Files

- #include <GL/gl.h>
- #include <GL/glu.h>
- #include <GL/glut.h>
  - Automatically includes gl.h, glu.h

Libraries

- -lopengl32 -lglu32 -lglut32

Enumerated Types

- OpenGL defines numerous types for compatibility
- GLfloat, GLint, GLenum, etc.

GLUT Basics

Application Structure

- **Configure and open window**
- **Initialize OpenGL state**
- **Register input callback functions**
  - render
  - resize
  - input: keyboard, mouse, etc.
- **Enter event processing loop**

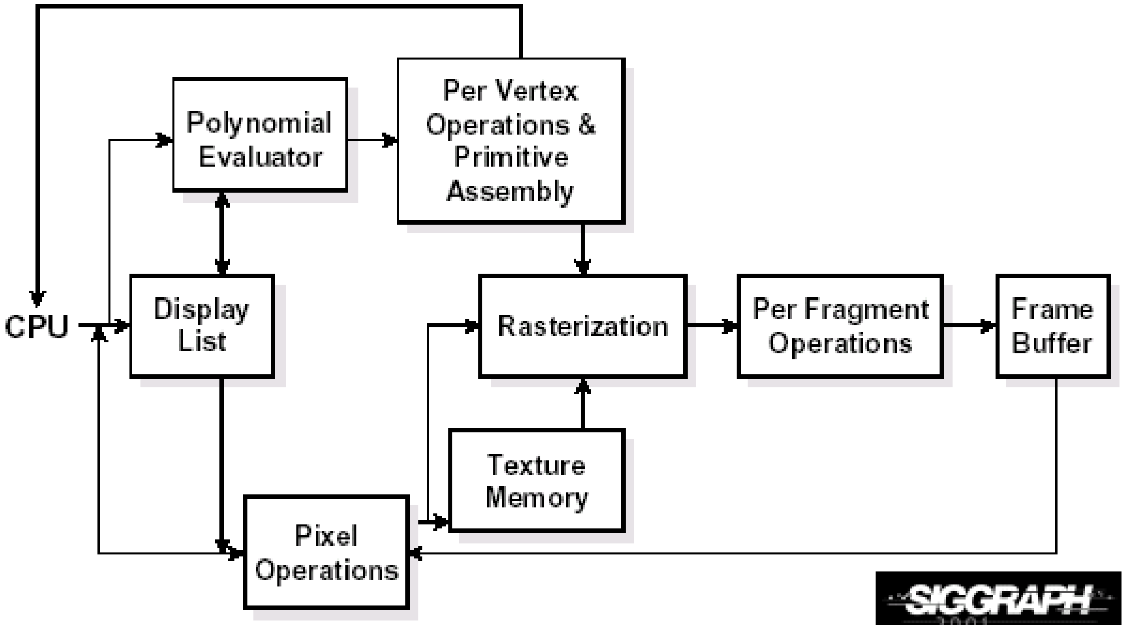
Main Template

```
int main(int argc, char** argv) {
    int mode = GLUT_RGB | GLUT_SINGLE;
    glutInit(&argc, argv);
    glutInitDisplayMode(mode);
    glutInitWindowSize(200, 200);
    glutInitWindowPosition(200, 200);
    glutCreateWindow("OpenGL Demo");
// …
    glutMainLoop();
    return 0;
}
```

OpenGL – State Machine

All rendering attributes are encapsulated in the OpenGL State

- rendering styles
- shading
- lighting
- texture mapping



Manipulating OpenGL State

Appearance is controlled by current state

```
for each ( primitive to render ) {
    update OpenGL state
    render primitive
}
```

Manipulating vertex attributes is most common way to manipulate state

glColor*() , glNormal*() , glTexCoord*(), …

Controlling the Current State

Setting State

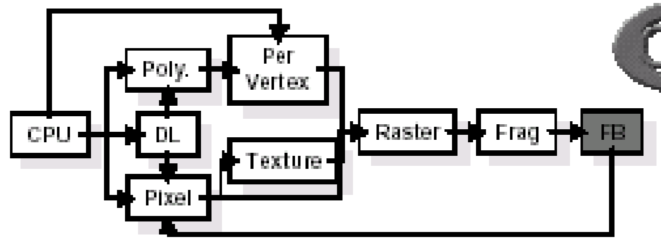
```
glPointSize( size );
glLineStipple( repeat, pattern );
glShadeModel( GL_ SMOOTH );
```

Enabling Features

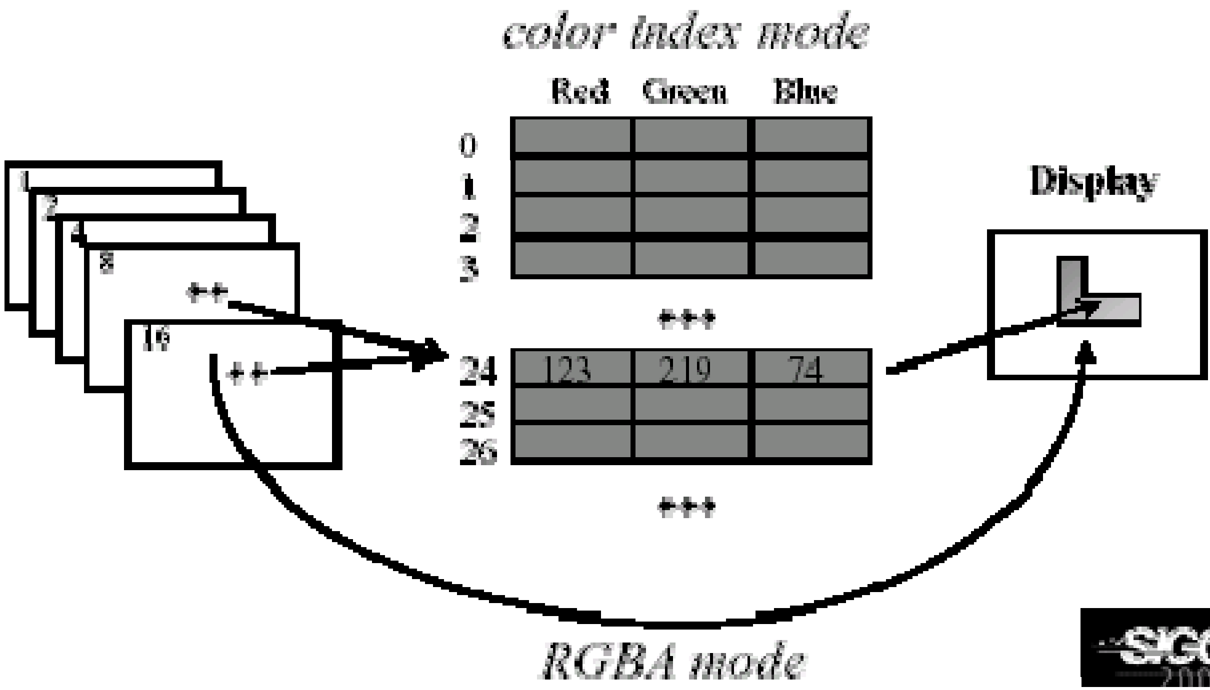
```
glEnable( GL_ LIGHTING );
glDisable( GL_TEXTURE_2D );
```

OpenGL Color Models

- **RGBA or Color Index**



  - glColor*() *or* glIndex*()
  - glutInitDisplayMode(GLUT_RGBA *or* GLUT_INDEX)



Initialization

Set up global state

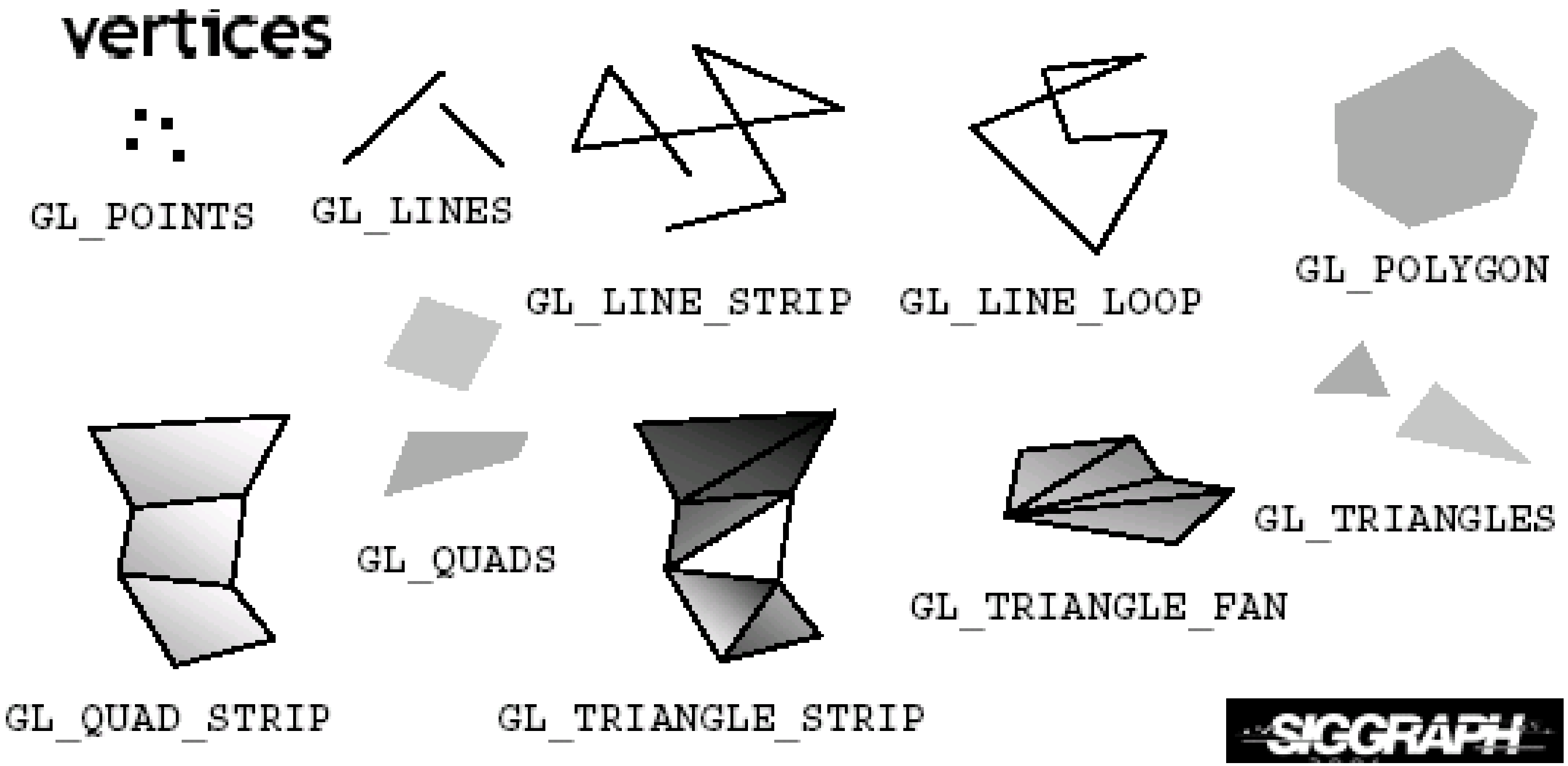
init();

– valid for entire execution time

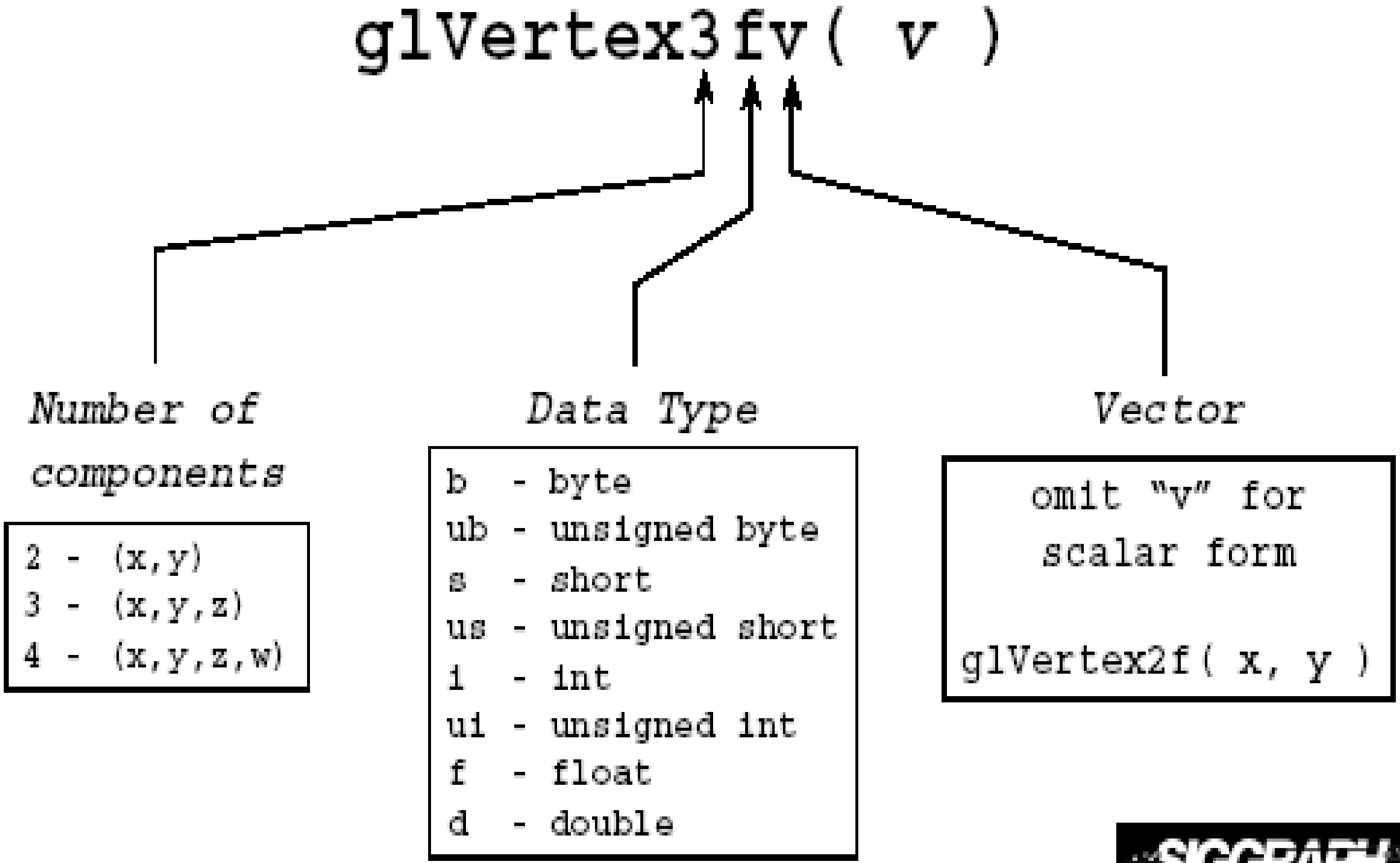
```
void init(void) {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glMatrixMode(GL_PROJECTION);      // two matrix stacks
    glLoadIdentity();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    …
}
```

Geometric Primitives

- All geometric primitives are specified by vertices
- Strips and fans save on the number of vertices



OpenGL Command Formats



Specifying Geometric Primitives

- **Primitives are specified using**
  glBegin(*primType*);
  glEnd();
  - *primType* determines how vertices are combined

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

OpenGL Primitive Types

- GL_POINTS
- GL_LINE_STRIP
- GL_LINES
- GL_LINE_LOOP
- GL_POLYGON
- GL_TRIANGLE_STRIP
- GL_TRIANGLES
- GL_TRIANGLE_FAN
- GL_QUADS
- GL_QUAD_STRIP

GLUT Callback Functions

Routine to call when something happens

- rendering
- user input
- animation
- window resize or redraw

“Register” callbacks with GLUT

```
glutDisplayFunc( display );
glutKeyboardFunc( keyboard );
glutIdleFunc( idle );
glutReshapeFunc( resize );
```

Rendering Callback

Do all of your drawing here

```
glutDisplayFunc(display);
```

void display(void) {

```
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_TRIANGLES);
        glColor3f(1, 0, 1); glVertex3f(-0.5, -0.5, 0.0);
        glColor3f(0, 0, 1); glVertex3f(-0.5, 0.5, 0.0);
        glColor3f(1, 0, 0); glVertex3f(0.5, 0, 0.0);
    glEnd();

    glFlush();
}
```

User Input Callback

React to key strokes

```
glutKeyboardFunc( keyboard );
```

void keyboard(unsigned char key, int x, int y) {

```
  switch (key) {
    case 27:
      exit(0); break;
    case '[':
      col = col < 0. ? 0. : col-0.1; glutPostRedisplay(); break;
    case ']':
      col = col > 1. ? 1. : col+0.1; glutPostRedisplay(); break;
  }
}
```

Global variable GLfloat col=0.;

In display() glColor3f(1, col, 0); glVertex3f(0.5, 0, 0.0);

Idle Callbacks

Use for animation and continuous update

```
glutIdleFunc( idle );
```

void idle(void) {

```
    t +=dt;
    glutPostRedisplay();
}
```

Global variables

```
GLfloat t = 0;
GLfloat dt= 0.001;
```

In display()

```
glColor3f( 0.5+0.5*cos(t), 0,1);
```

Callback Functions

- **glutDisplayFunc()**
  - called when pixels in the window need to be refreshed
- **glutReshapeFunc()**
  - called when the window changes size
- **glutKeyboardFunc()**
  - called when a key is struck on the keyboard
- **glutMouseFunc()**
  - called when the user presses a mouse button on the mouse
- **glutMotionFunc()**
  - called when the user moves the mouse while a mouse button is pressed
- **glutPassiveMouseFunc()**
  - called when the mouse is moved regardless of mouse button state
- **glutIdleFunc()**
  - called when nothing else is going on; very useful for animations

Online Resources

http://www.khronos.org

– Offical home

http://www.opengl.org

– start here; up to date specification and lots of sample code

http://www.mesa3d.org/

– Brian Paul's Mesa 3D (OpenGL in Software)

http://www.cs.utah.edu/~narobins/opengl.html

– GLUT & interactive tutorials

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

- **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**