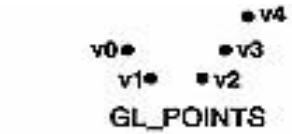
Real-time Graphics

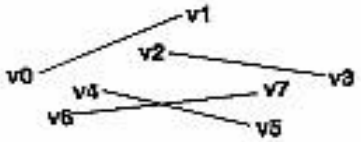
2. Buffer Objects, FBO

Martin Samuelčík

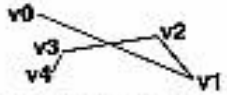
Geometry entry



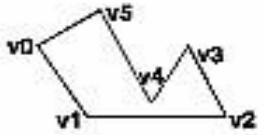
GL_POINTS



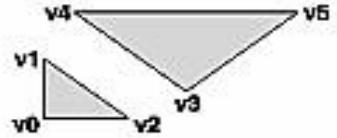
GL_LINES



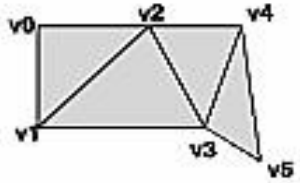
GL_LINE_STRIP



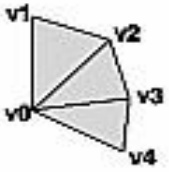
GL_LINE_LOOP



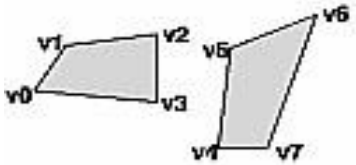
GL_TRIANGLES



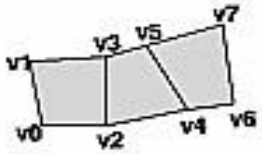
GL_TRIANGLE_STRIP



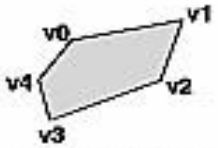
GL_TRIANGLE_FAN



GL_QUADS



GL_QUAD_STRIP



GL_POLYGON

Geometry entry

- Intermediate Mode
  - *glBegin* / *glEnd* block
  - Each vertex given by *glVertex*
  - Slow, Depreciated
- Vertex arrays
  - Lots of data in large buffers
  - Minimalization of function calls
  - Buffers for vertex attributes (coordinates, normals, texture coordinates, …)

Display lists

- Group of commands stored for later execution in compiled form
- No later evaluation and data transmitting
- Efficient for static data
- Can be shared between contexts
- After compilation, can’t be modified - bad for dynamic data
- Client related commands can’t be stored (vertex arrays)

Vertex arrays

- Solve sharing of vertex data between polygons, separate vertex and polygon
- Arrays of vertex attributes – coordinates, normals, colors, tex. coordinates, …
- Arrays of indices for creating polygons
- Arrays are in client memory
- Arrays are transmitted each frame

Setting vertex arrays

- Set data in client's memory as vertex attributes
- *void glVertexPointer (GLint size, GLenum type, GLsizei stride, const GLvoid* pointer)*
  - *size* - 2, 3, 4
  - *type* - GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
  - *stride* - byte offset between consecutive vertices
  - *pointer* - data in client memory
- *glColorPointer*, *glTexCoordPointer*, ...
- Enable: *void glEnableClientState (GLenum cap)*

Vertex arrays – OGL 2.0

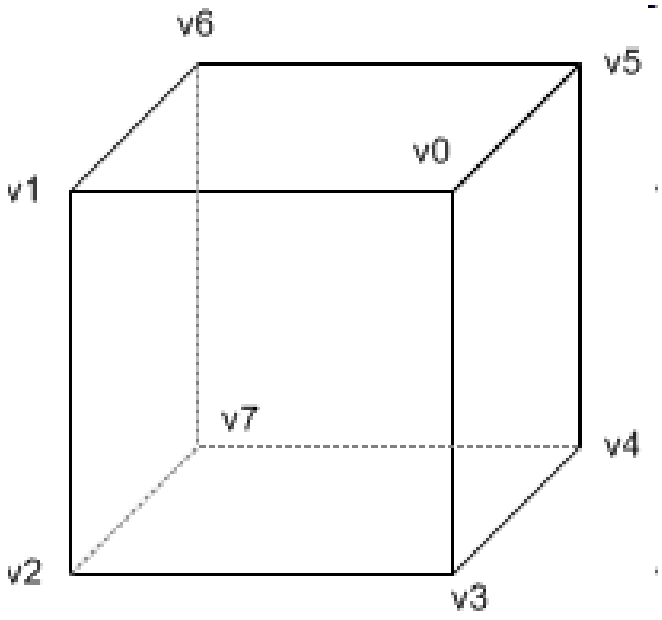
- Passing arbitrary vertex attributes to vertex shader
- *void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* pointer)*
  - *index* - location of attribute in shader program
  - *size* - number of components – 1,2,3,4
  - *type* - data type of each component
  - *normalized* - integer values mapped to [-1,1] or [0,1]
  - *stride* - byte offset between consecutive attributes
  - *pointer* – data
- Enable: *void glEnableVertexAttrib(GLuint index)*

Vertex arrays drawing

- *void glDrawArrays (GLenum mode, GLint first, GLsizei count)*
  - *mode* - GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, GL_POLYGON
  - *first* - specifies starting index
  - *count* - specifies the number of used indices
- *void glDrawElements (GLenum mode, GLsizei count, GLenum type, const GLvoid *indices)*
  - *type* - type of each index in indices array - GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT
  - *indices* - array of indices to be used for primitives

Vertex arrays

```
GLfloat vertices[] = {1,1,1,    -1,1,1,  -1,-1,1,   1,-1,1,          // v0-v1-v2-v3
                      1,-1,-1, 1,1,-1, -1,1,-1,  -1,-1,-1};        // v4-v5-v6-v7
GLfloat colors[] = {1,1,1,   0,1,1, 0,0,1,  1,0,1,                 // c0-c1-c2-c3
                    1,0,0,   1,1,0,   0,1,0,  0,0,0};              // c4-c5-c6-c7
GLubyte indices[] = {0,1,2,3, 0,3,4,5, 0,5,6,1,                    // f0-f1-f2
                     1,6,7,2,   7,4,3,2,  4,7,6,5};                // f3-f4-f5

// activate and specify pointers to vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, colors);

// draw a cube
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```



Vertex arrays – OGL 2.0

```
GLfloat vertices[] = {1,1,1,   -1,1,1,  -1,-1,1,   1,-1,1, 1,-1,-1, 1,1,-1, -1,1,-1,  -1,-1,-1};
GLfloat colors[] = {1,1,1,  0,1,1,  0,0,1,  1,0,1,  1,0,0,   1,1,0,   0,1,0,  0,0,0};
GLubyte indices[] = {0,1,2,3,  0,3,4,5,  0,5,6,1,  1,6,7,2,   7,4,3,2,   4,7,6,5};

// get location, index of attributes in shader
GLuint vertexLoc = glGetAttribLocation(programID, "InVertex");
GLuint colorLoc = glGetAttribLocation(programID, "InColor");

// activate and specify pointers to vertex attribute arrays
glEnableVertexAttrib(vertexLoc);
glVertexAttribPointer(vertexLoc, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), vertices);
glEnableVertexAttrib(colorLoc);
glVertexAttribPointer(colorLoc, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), colors);

// draw a cube
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);

// deactivate vertex arrays after drawing
glDisableVertexAttrib(vertexLoc);
glDisableVertexAttrib(colorLoc);
```

```
attribute vec4 InVertex;
attribute vec3 InColor;
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * InVertex;
  gl_FrontColor = vec4(InColor, 1.0);
}
```

OpenGL buffer objects

- Unified framework for work with buffers containing data of various types, server manages best location for data
- Each buffer object is represented by identifier, "name" – GLuint
- *void glGenBuffers{ARB} (GLsizei n, GLuint* bufs)*
  - generate *n* buffer object "names" (IDs)
  - *bufs* – array of size *n* for new buffer IDs
- *void glDeleteBuffers{ARB} (GLsizei n, const GLuint* bufs)*
  - delete *n* "named" buffer objects, *bufs* is array of IDs

Current buffer object

- Only one active buffer objects of given type at a time
- Setting active buffer object with ID = *bufID*
- *void glBindBuffer{ARB} (GLenum target, GLuint bufID)*
  - *target* – type of active buffer:
    - GL_ARRAY_BUFFER
    - GL_ELEMENT_ARRAY_BUFFER
    - GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER
    - GL_UNIFORM_BUFFER
    - GL_TRANSFORM_FEEDBACK_BUFFER
    - …

Buffer object data

- Creates and initializes memory for active buffer object's data, fills memory with given data
- *void glBufferData{ARB} (GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage)*
  - *target*: GL_ARRAY_BUFFER, …
  - *size* – number of bytes, size of data
  - *data* – client memory block to be copied into buffer object, NULL – no copying, just allocating
  - *usage*
    - GL_STREAM_DRAW, _READ, _COPY
    - GL_STATIC_DRAW, _READ, _COPY,
    - GL_DYNAMIC_DRAW, _READ, _COPY

Modifying data

- Map content of buffer object to part of client's memory for reading or writing
- *void* glMapBuffer{ARB}* *(GLenum target, GLenum access)*
  - *access* - GL_READ_ONLY, GL_WRITE_ONLY, GL_READ_WRITE
- Now application can read or modify data in client memory given by returned pointer

Modifying data

- Finishing with modification, changes are written to buffer object
- *GLboolean glUnmapBuffer{ARB} (GLenum target)*
- Getting parameters of buffer object
- *void glGetBufferParameteriv{ARB} (GLenum target, GLenum value, GLint * data);*
  - *value* - GL_BUFFER_ACCESS, GL_BUFFER_MAPPED, GL_BUFFER_SIZE, or GL_BUFFER_USAGE
  - *data* – returned parameter value

Vertex Buffer Objects

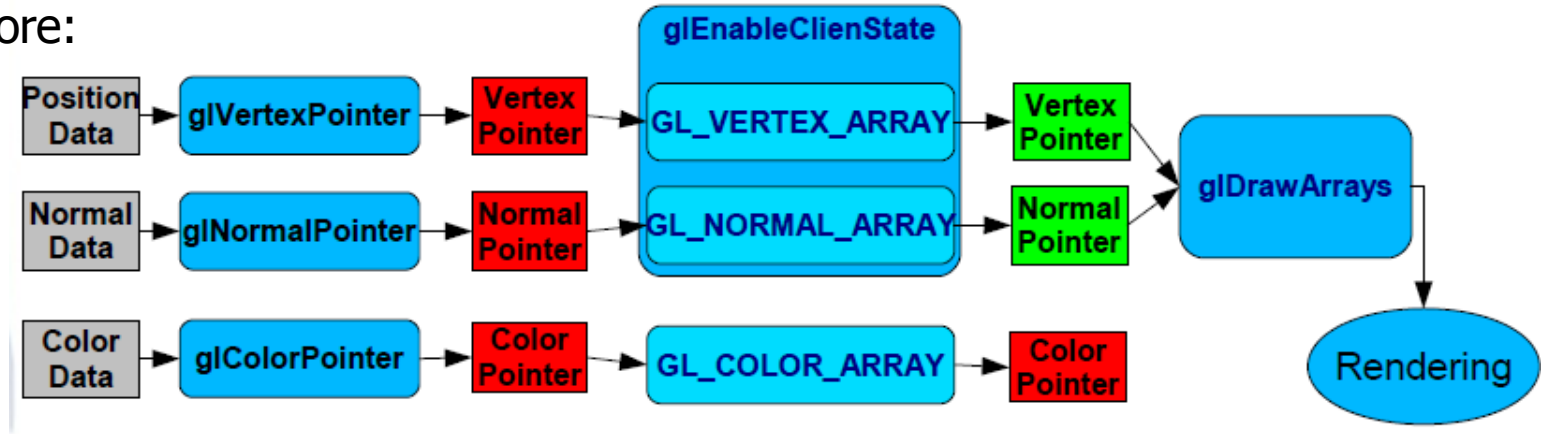
- Enhanced vertex arrays
- Vertex attributes and indices are copied to server memory only once as buffer objects
- Instead of vertex array or index array, buffer is attached
- Extension - GL_ARB_vertex_buffer_object
- From OpenGL 1.5
- Buffers can be shared between contexts

Using VBO

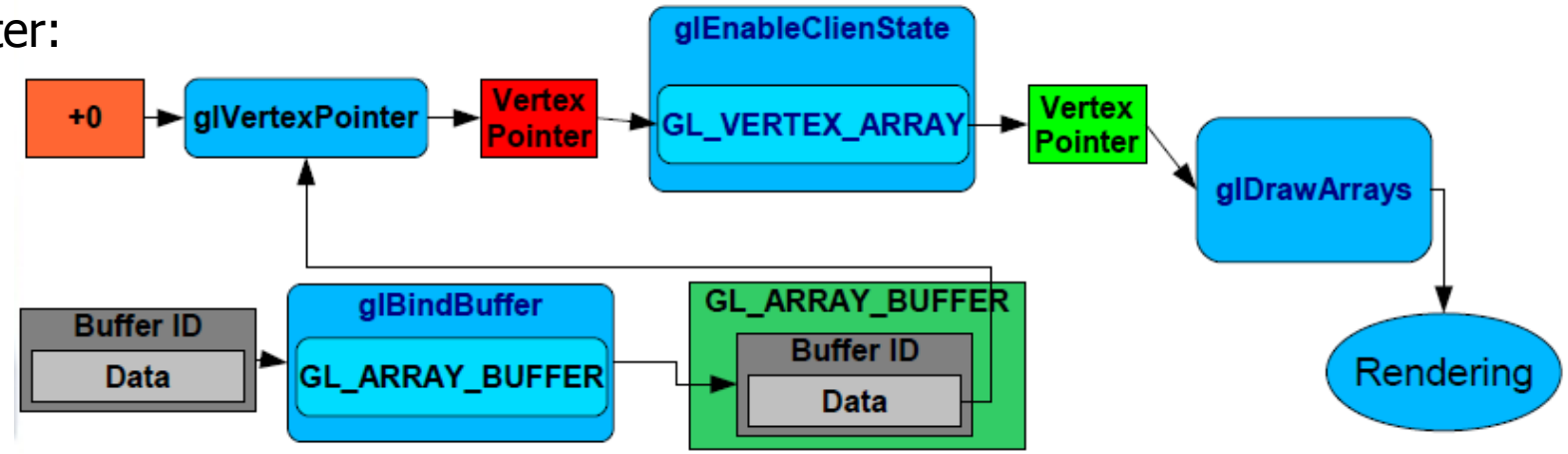
- Use vertex arrays as usual, instead of setting pointer to client memory, bind prepared buffer object and set pointer to 0
- Data from buffer will be used
- Before:
  - *glVertexPointer (3, GL_FLOAT, 0, vertices)*
- After:
  - *glBindBuffer (GL_ARRAY_BUFFER, uiID)*
  - *glVertexPointer (3, GL_FLOAT, 0, NULL)*

Using VBO

Before:

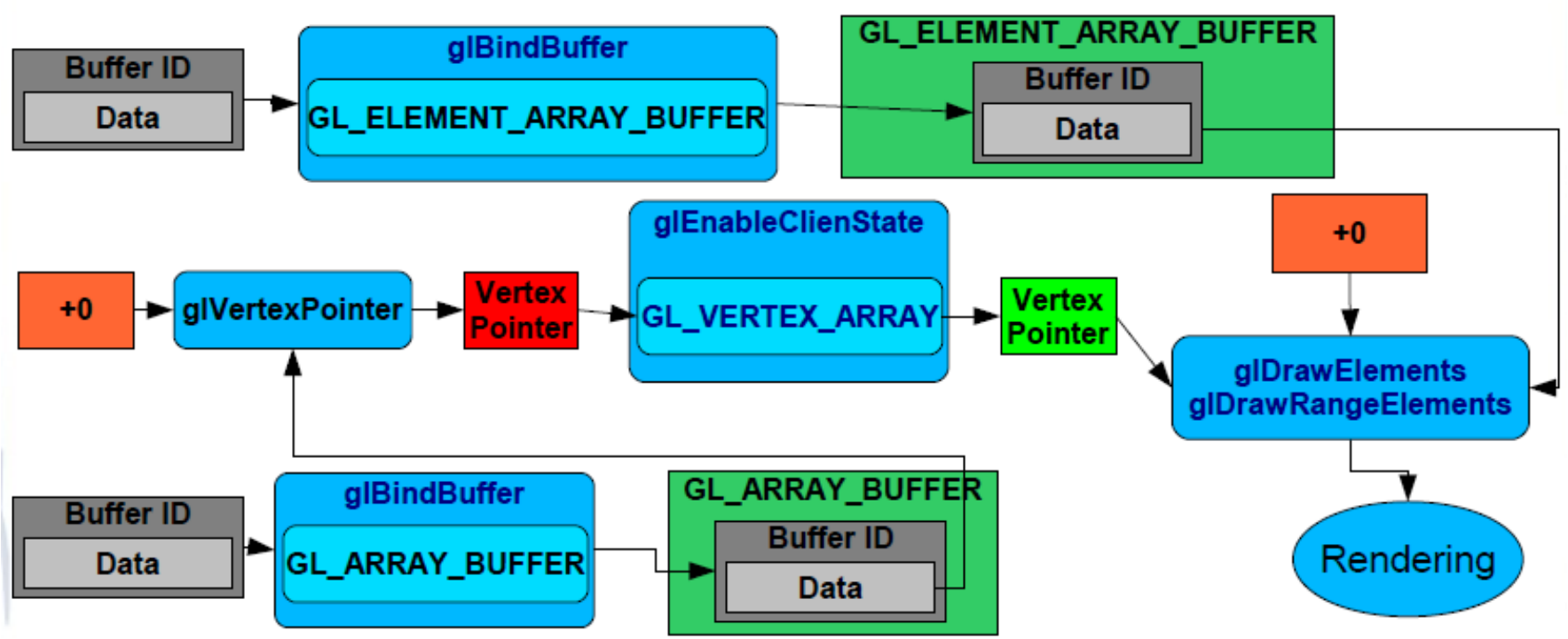


After:



Drawing with VBO

- *glDrawElements* is retrieving indices from actual binded element buffer



VBO example - init

```
GLfloat vertices[] = {1,1,1,   -1,1,1,  -1,-1,1,   1,-1,1, 1,-1,-1, 1,1,-1, -1,1,-1,  -1,-1,-1};
GLfloat colors[] = {1,1,1,   0,1,1,  0,0,1,  1,0,1, 1,0,0,   1,1,0,   0,1,0,  0,0,0};
GLubyte indices[] = {0,1,2,3,  0,3,4,5,  0,5,6,1,  1,6,7,2,   7,4,3,2, 4,7,6,5};

// prepare used buffer objects, in init phase of application
// buffer object with coordinates
glGenBuffers(1, &g_uiCoordBuffer);
glBindBuffer(GL_ARRAY_BUFFER, g_uiCoordBuffer);
glBufferData(GL_ARRAY_BUFFER, 24 * sizeof(GLfloat), vertices, GL_STATIC_DRAW);

// buffer object with colors
glGenBuffers(1, &g_uiColorBuffer);
glBindBuffer(GL_ARRAY_BUFFER, g_uiColorBuffer);
glBufferData(GL_ARRAY_BUFFER, 24 * sizeof(GLfloat), colors, GL_STATIC_DRAW);

// buffer object with indices
glGenBuffers(1, &g_uiIndexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_uiIndexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 24 * sizeof(GLubyte), indices, GL_STATIC_DRAW);
```

VBO example - draw

```
// activate and specify pointers to vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, g_uiCoordBuffer);
glVertexPointer(3, GL_FLOAT, 0, NULL);

glEnableClientState(GL_COLOR_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, g_uiColorBuffer);
glColorPointer(3, GL_FLOAT, 0, NULL);

// draw a cube
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_uiIndexBuffer);
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, NULL);

// deactivate vertex arrays and VBO after drawing
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

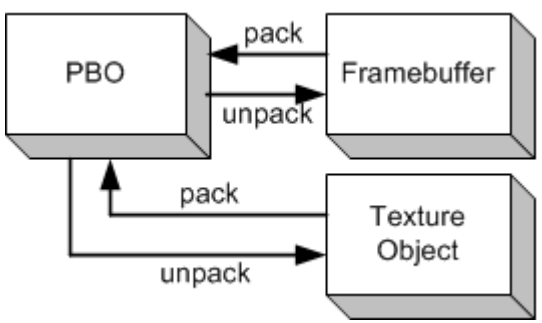
VBO draw – OGL 2.0

```
// get location, index of attributes in shader
GLuint vertexLoc = glGetAttribLocation(programID, "InVertex");
GLuint colorLoc = glGetAttribLocation(programID, "InColor");

// activate and specify buffers to vertex attribute arrays
glEnableVertexAttrib(vertexLoc);
glBindBuffer(GL_ARRAY_BUFFER, g_uiCoordBuffer);
glVertexAttribPointer(vertexLoc, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), NULL);
glEnableVertexAttrib(colorLoc);
glBindBuffer(GL_ARRAY_BUFFER, g_uiColorBuffer);
glVertexAttribPointer(colorLoc, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), NULL);

// draw a cube
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_uiIndexBuffer);
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, NULL);

// deactivate vertex arrays after drawing
glDisableVertexAttrib(vertexLoc);
glDisableVertexAttrib(colorLoc);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

```
// vertex shader using arbitrary VBO attributes
attribute vec4 InVertex;
attribute vec3 InColor;
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * InVertex;
  gl_FrontColor = vec4(InColor, 1.0);
}
```

Pixel Buffer Objects

- Extension ARB_pixel_buffer_object
- Storing pixel data in buffer objects
- Fast pixel data transfer to and from a graphics card using DMA without CPU
- Replaces usage of client memory buffers for pack and unpack functions
- OpenGL 2.1



Pixel Buffer Objects

- Unpack (read): *glBitmap*, *glColorSubTable*, *glColorTable*, *glCompressedTexImage1D*, *glCompressedTexImage2D*, *glCompressedTexImage*, *glCompressedTexSubImage1D*, *glCompressedTexSubImage2D*, *glCompressedTexSubImage3D*, *glConvolutionFilter1D*, *glConvolutionFilter2D*, *glDrawPixels*, *glPixelMapfv*, *glPixelMapuiv*, *glPixelMapusv*, *glPolygonStipple*, *glSeparableFilter2D*, *glTexImage1D*, *glTexImage2D*, *glTexImage3D*, *glTexSubImage1D*, *glTexSubImage2D*, *glTexSubImage3D*
- Pack (write): *glGetCompressedTexImage*, *glGetConvolutionFilter*, *glGetHistogram*, *glGetMinmax*, *glGetPixelMapfv*, *glGetPixelMapuiv*, *glGetPixelMapusv*, *glGetPolygonStipple*, *glGetSeparableFilter*, *glGetTexImage*, *glReadPixels*— read a block of pixels from the frame buffer

```
// buffer object for storing pixel data
glGenBuffers(1, &g_uiPixelBuffer);
glBindBuffer(GL_PIXEL_PACK_BUFFER , uiPixelBuffer);
glBufferData(GL_PIXEL_PACK_BUFFER , 1024 * 768 * 4, 0, GL_STATIC_READ);
glReadPixels(0, 0, 1024, 768, GL_RGBA, GL_UNSIGNED_BYTE, 0);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);
```

Uniform buffer objects

- Sending uniform variables to shader programs, using block of uniforms
- Extension GL_ARB_uniform_buffer_object
- OpenGL 3.1

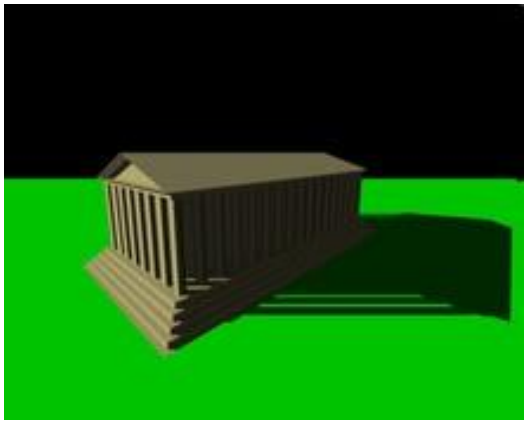
```
// Create and initialize
glGenBuffers(1, &UniformBufferTransformName);
glBindBuffer(GL_UNIFORM_BUFFER, UniformBufferTransformName);
glBufferData(GL_UNIFORM_BUFFER, GLsizei(sizeof(MVP)), &MVP[0][0], GL_DYNAMIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
UniformTransform = glGetUniformBlockIndex(ProgramName, "transform");
glUseProgram(ProgramName);
glBindBufferBase(GL_UNIFORM_BUFFER, 1, UniformBufferTransformName);
glUniformBlockBinding(ProgramName, UniformTransform, 1);
glUseProgram(0);

// Render, set the value of MVP uniform.
glUseProgram(ProgramName);
glBindBuffer(GL_UNIFORM_BUFFER, UniformBufferTransformName);
glBufferSubData(GL_UNIFORM_BUFFER, 0, GLsizei(sizeof(MVP)), &MVP[0][0]);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Transformation feedback

- Primitives processed by a Vertex, Tesselation, Geometry Shader will be written to buffer objects
- Rasterizer can be switched off
- Fast processing of transformations
  - Update of particle systems
  - Tesselation
  - …
- GL_EXT_transform_feedback

Rendering to texture

- Fragment data copied also to textures
- Off-screen rendering to several buffers, textures
- Crucial for most effects, for multi-pass rendering
- Shadow maps, post-processing (HDR, gloom, filtering), reflections, SSAO, GPGPU, ……

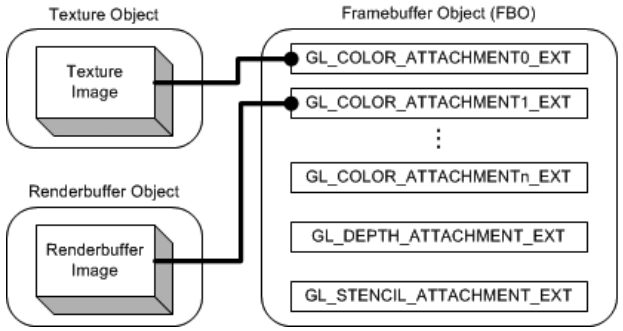


Rendering to texture

- glReadPixels() -> glTexImage*()
  - slow, related to window size
  - PBO
- glCopyTexImage*()
  - better, related to window size
- glCopyTexSubImage*()
  - better, related to window size
- P-buffer
  - fast, new context must be created
  - Z-buffer only on Nvidia

Framebuffer objects

- GL_ARB_framebuffer_object, OpenGL 3.0
- Additional non-displayable framebuffers
- Redirect the rendering output to the application-created framebuffer
- Framebuffer-attachable images:
  - textures and renderbuffers
- FBO contains a collection of rendering destinations:
  - color, depth, stencil



FBO management

- Several framebuffer objects, each with integer identifier
- *void glGenFramebuffersEXT(GLsizei n, GLuint * framebuffers)*
  - generate *n* framebuffer object names
- *void glDeleteFramebuffersEXT(GLsizei n, const GLuint * framebuffers)*
  - delete named framebuffer objects
- *void glBindFramebufferEXT(GLenum target, GLuint framebuffer)*
  - bind a named framebuffer object
  - *target* must be GL_FRAMEBUFFER_EXT

FBO texture images

- *void glFramebufferTexture2DEXT(GLenum target, GLenum attachment, GLenum textarget, GLuint texture, Glint level)*
  - *target* - must be GL_FRAMEBUFFER_EXT
  - *attachment*
    - GL_COLOR_ATTACHMENT0 .. n – n color textures
    - GL_DEPTH_ATTACHMENT – one depth texture
    - GL_STENCIL_ATTACHMENT – one stencil texture
  - *textarget*
    - GL_TEXTURE_2D
    - GL_TEXTURE_CUBE_MAP_POSITIVE_X, _Y, _Z,
    - GL_TEXTURE_CUBE_MAP_ NEGATIVE_X, _Y, _Z
  - *texture* - texture name generated by glGenTextures and set by glTexImage2D
  - *level* - mipmap level to attach from texture to attachment

FBO renderbuffer images

- *void glGenRenderbuffersEXT (GLsizei n, GLuint * renderbuffs)*
  - generate renderbuffer object names
- *void glDeleteRenderbuffersEXT (GLsizei n, const GLuint * renderbuffs)*
  - delete named renderbuffer objects
- *void glBindRenderbufferEXT (GLenum target, GLuint renderbuffer)*
  - bind a named renderbuffer object
- *void glRenderbufferStorageEXT (GLenum target, GLenum internalformat, GLsizei width, GLsizei height)*
  - target - must be GL_RENDERBUFFER_EXT
  - internalformat - GL_RGBA4, GL_RGB565, GL_RGB5_A1, GL_DEPTH_COMPONENT16, 24, GL_STENCIL_INDEX8,...

FBO renderbuffer images

- *void glFramebufferRenderbufferEXT (GLenum target, GLenum attachment, GLenum renderbuffertarget, GLuint renderbuffer)*
  - target - must be GL_FRAMEBUFFER_EXT
  - attachment -
    - GL_COLOR_ATTACHMENT0..n
    - GL_DEPTH_ATTACHMENT
    - GL_STENCIL_ATTACHMENT
  - renderbuffertarget - must be GL_RENDERBUFFER_EXT
  - renderbuffer - name generated by glGenRenderbuffersEXT

FBO status

- Validating current FBO
- *GLenum glCheckFramebufferStatusEXT (GL_FRAMEBUFFER_EXT)*
- Should be GL_FRAMEBUFFER_COMPLETE_EXT
- Rules for textures:
  - The width and height of framebuffer-attachable image must be not zero.
  - If an image is attached to a color attachment point, then the image must have a color-renderable internal format. (GL_RGBA, GL_DEPTH_COMPONENT, GL_LUMINANCE, etc)
  - If an image is attached to GL_DEPTH_ATTACHMENT_EXT, then the image must have a depth-renderable internal format. (GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT24_EXT, etc)
  - If an image is attached to GL_STENCIL_ATTACHMENT_EXT, then the image must have a stencil-renderable internal format. (GL_STENCIL_INDEX, GL_STENCIL_INDEX8_EXT, etc)
  - FBO must have at least one image attached.
  - All images attached a FBO must have the same width and height.
  - All images attached the color attachment points must have the same internal format.

FBO example - init

```
// create a texture object
GLuint textureId;
glGenTextures(1, &textureId);
glBindTexture(GL_TEXTURE_2D, textureId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, TEXTURE_WIDTH, TEXTURE_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, 0);
glBindTexture(GL_TEXTURE_2D, 0);

// create a renderbuffer object to store depth info
GLuint rboId;
glGenRenderbuffersEXT(1, &rboId);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, rboId);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, TEXTURE_WIDTH,  TEXTURE_HEIGHT);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);

// create a framebuffer object
GLuint fboId;
glGenFramebuffersEXT(1, &fboId);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

// attach the texture to FBO color attachment point and the renderbuffer to depth attachment point
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, textureId, 0);
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, rboId);

// check FBO status
GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
if(status != GL_FRAMEBUFFER_COMPLETE_EXT) fboUsed =  false;
```

FBO example - draw

```
// create a framebuffer object
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

// set attachment to draw to
// if no color attachment is attached, call
//        glDrawBuffer(GL_NONE);
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);

// ..... Render scene to texture here .....

// switch back to window-system-provided framebuffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

Multiple render targets

- Extension GL_ARB_draw_buffers
- Multiple color attachments to store additional rendering info
- *glGetIntegerv(GL_MAX_COLOR_ATTACHMENTS_EXT, &maxColorAttachments)*
- *void glDrawBuffers(GLsizei n, const GLenum * bufs)*
  - *n* - number of color attachments to draw
  - *bufs* - array of color attachments
- Fragment shader - gl_FragData[i] is output variable that will be written to i-th draw attachment

MRT example

```
// create a framebuffer object
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

// get number of color attachments
GLint maxColorAttachments;
glGetIntegerv(GL_MAX_COLOR_ATTACHMENTS_EXT, &maxColorAttachments);

// set attachments to draw to
// these attachments must be prepared as textures or renderbuffers
GLenum drawbuffers[3] = {GL_COLOR_ATTACHMENT0_EXT,
                         GL_COLOR_ATTACHMENT1_EXT,
                         GL_COLOR_ATTACHMENT2_EXT};
If (maxColorAttachments >= 3)
      glDrawBuffers(3, drawbuffers);
else
      glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);

// ..... Render scene to textures here .....

// switch back to window-system-provided framebuffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

```
// MRT fragment shader
void main(void)
{
   gl_FragData[0] = vec4(1.0, 0.0, 0.0, 1.0);
   gl_FragData[1] = vec4(0.0, 1.0, 0.0, 1.0);
   gl_FragData[2] = vec4(0.0, 0.0, 1.0, 1.0);
}
```

MRT - Deferred shading

- Render scene to multiple render targets storing basic info about pixels (material, normal, …) – G-buffer
- Compute shading only for window pixels in screen space
- Difficult transparency & HW anti-aliasing

DS	Depth (24bit integer)	Stencil
RT0	Lighting accumulation RGB	Glow
RT1	View space normals XY (RG FP16)	
RT2	Motion vectors XY	Roughness, spec. intensity
RT3	Albedo RGB	Sun shadow

Deferred shading

[Killzone 2]

Deferred lighting

- Deferred lighting pipeline
  - In first pass, scene is rendered and only normal and depth data are stored in G-buffer
  - In second pass, lighting (+shadows) is computed using normal texture, reconstructed eye position and lights parameters in screen space. Output is texture containing diffuse and specular values of accumulated lighting for each pixel
  - In third pass, render scene again and combine computed diffuse and specular lighting from texture (from second pass) with materials using some local lighting model. Post-process effects are added to result.
- No need for complicated G-buffer
- One more rendering pass for whole scene

Deferred shaders – 1. pass

- Creating G-buffer – buffers with diffuse, specular material and normal data
- Can be extended with other material properties

```
// Deferred shading – 1.pass – vertex shader
varying vec4 N_eye;
varying vec2 vTexCoord;

void main(void)
{
   vTexCoord = vec2(gl_MultiTexCoord0);
   N_eye = vec4(gl_NormalMatrix * gl_Normal, 0.0);
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
// Deferred shading – 1.pass - fragment shader
uniform sampler2D diffuseMap;
varying vec4 N_eye;
varying vec2 vTexCoord;

void main(void)
{
   vec4 mat = texture2D(diffuseMap, vTexCoord);
   mat.a = gl_FrontMaterial.specular.r;
   gl_FragData[0] = mat;
   gl_FragData[1] = 0.5 * (normalize(N_eye) + 1);
   gl_FragData[1].a = gl_FrontMaterial.shininess / 255.0;
}
```

Deferred shaders – 2. pass

- Rendering full-screen quad ([0,0,-1], [1,0,-1], [1,1,-1], [0,1,-1])
- Only one built-in light in scene, can be extended for many lights
  - Rendering light volumes for each light
  - Or rendering screen quad containing area of influence for each light
  - Rendering full-screen quad for directional light

```
// Deferred shading – 2.pass – vertex shader
varying vec2 vTexCoord;

void main(void)
{
   vTexCoord = vec2(gl_Vertex);
   gl_Position = 2 * gl_Vertex - 1;
}
```

Deferred shaders – 2. pass

```
// Deferred shading – 2.pass - fragment shader
uniform sampler2D materialMap;
uniform sampler2D normalMap;
uniform sampler2D depthTexture;
uniform mat4 projMatrix;
uniform mat4 invProjMatrix;
uniform vec2 viewport_dim;
varying vec2 vTexCoord;

void main(void)
{
  // compute eye space position of fragment
  // from depth and window space position
  vec4 pos_ndc;
  pos_ndc.x = 2 * gl_FragCoord.x / viewport_dim.x – 1;
  pos_ndc.y = 2 * gl_FragCoord.y / viewport_dim.y – 1;
  pos_ndc.z = 2 * texture2D(depthTexture, vTexCoord).r - 1;
  if (pos_ndc.z == -1) discard;
  float T1 = projMat[2][2];
  float T2 = projMat[2][3];
  float E1 = projMat[3][2];
  vec4 pos_clip;
  pos_clip.w = T2 / (pos_ndc.z – T1 / E1);
  pos_clip.x = pos_ndc.x * pos_clip.w;
  pos_clip.y = pos_ndc.y * pos_clip.w;
  pos_clip.z = pos_ndc.z * pos_clip.w;
  vec4 pos_eye = invProjMat * pos_clip;
```

```
  // get vectors for lighting computation
  vec4 N_spec = texture2D(normalMap, vTexCoord);
  float shininess = 255 * N_spec.a; N_spec.a = 0;
  vec4 N_eye = normalize(2 * N_spec - 1);
  vec4 L_eye = normalize(gl_LightSource[0].position – pos_eye);
  vec4 V_eye = normalize(-pos_eye);

  // compute coefficients for components
  float diffuse = clamp(dot(L_eye, N_eye), 0.0, 1.0);
  vec4 R_eye = reflect(-L_eye, N_eye);
  float specular = sgn(diffuse)*pow(clamp(dot(R_eye, V_eye), 0.0, 1.0),  shininess);

  // get material parameters from G-buffer texture
  vec4 mat = texture2D(materialMap, vTexCoord);

  // compute final color of fragment
  gl_FragColor = diffuse * gl_LightSource[0].diffuse * vec4(mat.xyz, 1) +
                 specular * gl_LightSource[0].specular * vec4(mat.a);
}
```

Deferred lighting, shading

- Battlefield 3
- Crackdown
- Crysis 2
- Dead Space and Dead Space 2
- Dungeons
- Grand Theft Auto IV
- Halo Reach
- inFamous
- Killzone 2 and Killzone 3
- LittleBigPlanet
- Mafia 2
- Metro 2033
- Stalker: Shadow of Chernobyl, Clear Sky and Call of Prypiat
- Red Dead Redemption
- StarCraft II
- Assassin's Creed 3
- Almost every new game

Questions?