# Lab 1: Setting up Software Emulator for Graphics Pipe in OpenGL

## Topics Covered

- Pixel buffer objects
- Asynchronous streaming from CPU to GPU - this is especially useful for streaming data for terrain rendering, water simulation, particle systems, skinning, implementing level of detail, movie or scripted animation playback
- Review OpenGL facilities for rendering texture mapped triangle primitives

## Prerequisites

- Understand how to set up a Visual Studio solution with property pages to quickly build projects that use GLFW, GLEW, GLM, FreeImage, and Dear ImGui.
- Understand viewport transform that maps from Normalized Device Context (NDC) coordinate system to window coordinate system.
- Be able to set up a Vertex Array Object with the state necessary to define a full-window, textured rectangle in NDC coordinates.
- Be able to implement texture mapping using 2D texture images.
- Be able to write, compile, and link vertex and fragment shader programs to render a full-window, textured rectangle specified in NDC coordinates.
- Please speak to me if you need help with any of the prerequisites.

## Getting Started

1. Set up a Visual Studio solution as described in Lab-1 from CS 200. External libraries such as GLFW and GLEW must be stored in `$(SolutionDir)lib` directory. GLEW has been update for OpenGL 4.6 - we'll be using `glew-2.1.0` rather than `glew-2.0.0`. Move the unzipped `glew-2.1.0` directory into `$(SolutionDir)lib`. Update your old property pages (or create new ones) to use `glew-2.1.0` rather than `glew-2.0.0`.

2. `./include/app.h` and `./src/app.cpp` contain the declarations and code required to create simple OpenGL applications - the code initializes an GL context with a double buffered framebuffer consisting of 32-bit (RGBA) color buffer and a 24-bit depth buffer. Adhere to this directory structure: source files will be located in `$(ProjectDir)src` while headers files will be located in `$ProjectDir)include`. This means that `app.cpp` must provide the full path to any local header files, as in

   ```
   1 #include "../include/app.h"
   ```

   The other alternative is to update the project pages to tell Visual Studio to look in `$(ProjectDir)include` to find certain header files. But it is better to assume nothing - instead specify the complete relative path so that the compiler can successfully find the headers irrespective of whether the grader has updated their property pages.

3. `./include/glshader.h` and `./src/glslshader.cpp` encapsulate the nitty-gritty details of reading shader source from files, compiling shader source, linking shader objects into a shader program, validating the shader program, and assigning values to `uniform` variables. All of the functions are defined except `GLboolean CompileShaderFromFile(GLenum shader_type, std::string const& file_name);` Use the description provided above this function's declaration in `./include/glslshader.h` to provide the correct implementation in `./src/glslshader.cpp`. ***This is important information:*** All shader source files must be located in `$(SolutionDir)shaders`. Otherwise, your submissions will not work and therefore cannot be graded.
4. Use `./src/main.cpp` as is.
5. Using `./include/app.h`, `./include/glslshader.h`, `./src/app.cpp`, `./src/glslshader.cpp`, and `./src/main.cpp`, compile, link, and execute - your code should display a window painted red similar to the sample executable `./Samples/lab-1-without.exe`.

# Conceptual Model for CS 250 Emulator

CS 250 is concerned with emulating some of the mathematical elements and algorithms implemented by fixed-function stages of the graphics hardware. To do this, the application must bypass much of the programmable and fixed-function stages in graphics hardware and yet display images to a window. One option is to render images to a Windows window using the GDI API. One drawback of this method is that the application is completely cut off from both graphics hardware and the GL API. Ideally, we'd like to display images from the emulator to one viewport while images generated by the graphics hardware are displayed in an adjacent viewport within the same window. This will allow comparison of images generated by the emulator and GLSL shaders from a GL-based application to detect any shortcomings in our understanding of the mathematical, algorithmic, and implementation details related to the graphics pipe and its abstraction through GL.

The method we use is common to data-intensive 3D applications which send large quantities of data from client memory to graphics memory every frame. Possible reasons for these data transfers include the implementation of skinning on skeletal objects, rendering large environments such as terrain and water, to stream triangle mesh information for level of detail, to compute physics simulations, setting uniform parameters for shaders with uniform buffers, and to implement streaming texture updates for movie playback, and so on. The method can be explained in the following steps:

1. Initialize a texture object with graphics memory storage for an image that will have the same dimensions and memory requirements as the framebuffer's colorbuffer. The contents of the image store are uninitialized but will be updated by the subsequent steps described below. This is done through GL commands `glCreateTextures` and `glTextureStorage2D` (see texture mapping lab for more information about these calls).
2. Reserve a chunk of graphics memory (similar to vertex and element buffers) with the same dimensions and memory requirements as the texture image from the previous step (and also equivalent to the framebuffer's colorbuffer). This is done through GL commands `glCreateBuffers` and `glNamedBufferStorage`.
3. Get a pointer to this chunk of graphics memory. This is done through GL command `glMapNamedBuffer`.
4. Program the emulator so that it uses this pointer to write color values directly to the chunk of graphics memory as though it were the emulator's framebuffer using a function such as `set_pixel` to write to

specific locations in the chunk of memory or use either `std::fill` or `std::memset` if performing an action similar to `glClear(GL_COLOR_BUFFER_BIT)`.

5. After the emulator has completed rendering a frame, release the pointer to graphics memory. This is done through GL command `glUnmapNamedBuffer`.
6. Copy the image from the chunk of graphics memory to the texture image store allocated in the first step. This is done through GL command `glTextureSubImage2D`.
7. Now, with the entire work done by the emulator specified as an image attached to a texture object, we only have to use a simple vertex shader program to render a full-window quad and a simple fragment shader to sample the texture image generated by the emulator to determine the color at each fragment.

These steps are conceptualized in Figure 1.

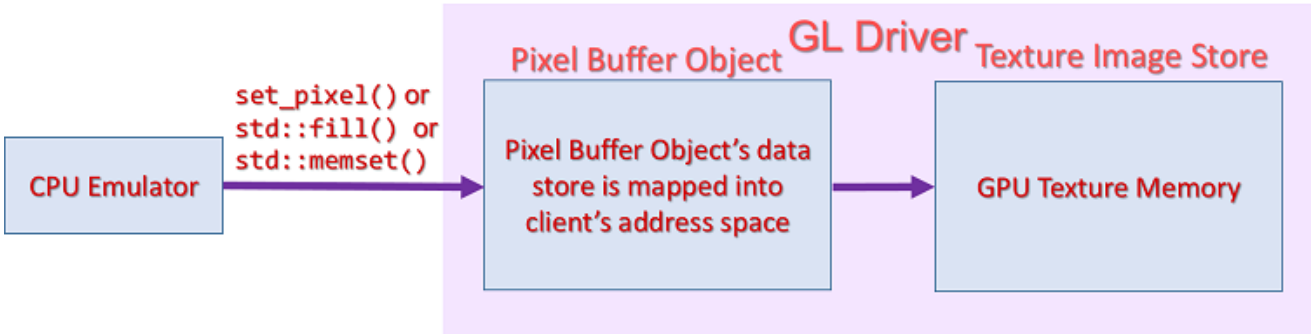


**Figure 1: Conceptual view of pixel data upload from emulator to graphics memory**

# What is a Pixel Buffer Object?

Recall that buffer objects conceptually are nothing more than chunks of memory. GL commands can source data from a buffer object by binding the buffer object to a given target (such as `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, and so on) and then overloading a certain set of GL commands' pointer arguments to refer to offsets inside the buffer, rather than pointers to client memory. In order to permit buffer objects to be used not only with vertex array data, but also with pixel data, later versions of GL added two new targets to which buffer objects can be bound: `GL_PIXEL_PACK_BUFFER` and `GL_PIXEL_UNPACK_BUFFER`. When a buffer object is bound to the `GL_PIXEL_PACK_BUFFER_TARGET`, commands such as `glReadPixels` *pack* (meaning *write*) their data into the buffer object. When a buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target, GL commands such as `glTextureSubImage2D` *unpack* (meaning *read*) their data from the buffer object. In our case, we're only interested in the `GL_PIXEL_UNPACK_BUFFER` target since the application is filling the buffer object with pixel data which the texture object is unpacking (or reading) into its image store. While a single buffer object can be bound for both vertex arrays and pixel commands, the designations vertex buffer object (VBO) and pixel buffer object (PBO) indicate their particular usage in a given situation. Otherwise, there's no difference between them.

Recall that in GL 4.5, a VBO is initialized with vertex attribute data in the following manner:

```
// client memory store for vertex attributes of quad
std::array<glm::vec2, 4> pos_vtx, st_vtx;
// assume application program fills these array with appropriate stuff

GLuint vbo_hdl;
glCreateBuffers(1, &vbo_hdl); // create a buffer object
// get graphics memory store for this buffer object
```

```
glNamedBufferStorage(vbo_hdl,
                     sizeof(glm::vec2)*(pos_vtx.size() + st_vtx.size()),
                     nullptr, GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glNamedBufferSubData(vbo_hdl,
                     0,
                     sizeof(glm::vec2) * pos_vtx.size(),
                     pos_vtx.data());
glNamedBufferSubData(vbo_hdl,
                     sizeof(glm::vec2)*pos_vtx.size(),
                     sizeof(glm::vec2) * st_vtx.size(),
                     st_vtx.data());

// store element array in graphics memory ...
// set up VAO ...
```

Setting up a PBO is not much different:

```
// compute number of bytes required to store RGBA pixel data for GL window
// assign this value to byte_cnt
glCreateBuffers(1, &pboid);
glNamedBufferStorage(pboid,
                     byte_cnt,
                     nullptr,
                     GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
```

Once the PBO is initialized, pixel data must be written by the emulator by grabbing a pointer to the graphics memory encapsulated by the PBO and mapping it into client address space. The next section will discuss these implementation details.

# Streaming Texture Updates: Implementation Details

The functionality required to generate, stream, and display images is declared in `./include/pbo.h`. You'll have to provide the implementation in `./src/pbo.cpp` . Scan through the declaration of type `Pbo` and identify the `static` data members. Begin your implementation by providing definitions to these `static` data members in `pbo.cpp`. You don't have to implement the `static` member functions in the order described but it would be the most convenient.

1. Define overloaded functions `Pbo::set_clear_color()` that set `static` data member `Pbo::clear_clr` with the RGBA values specified by function parameters. These functions emulate the behavior of the GL command `glClearColor()`.
2. Define `Pbo::clear_color_buffer()` - this function emulates GL command `glClear(GL_COLOR_BUFFER_BIT)` by using pointer `Pbo::ptr_to_pbo` to fill the PBO (that `Pbo::ptr_to_pbo` points to) with the RGBA value in `Pbo::clear_clr`. Since millions of bytes are being written to, think of how this can be done as efficiently as possible - potential candidates from the standard library include `std::fill()` and `std::memset()`.
3. Implement `Pbo::init()`:
    1. Begin by assigning appropriate values to `static` data members `Pbo::width`, `Pbo::height`, `Pbo::pixel_cnt`, and `Pbo::byte_cnt` so that the PBO and texture object will have the same

dimensions as the GL context's framebuffer. This means that `Pbo::width` and `Pbo::height` must be equivalent to `App::width` and `App::height`. `Pbo::pixel_cnt` will be equivalent to the number of pixels in the PBO while `Pbo::byte_cnt` is the number of total bytes in the PBO.
2. Set the color in data member `Pbo::clear_clr()` to white (or whatever color you wish) through `Pbo::set_clear_color()`.
3. Initialize `Pbo::pboid` by creating a PBO with an image store having dimensions `Pbo::width` $\times$ `Pbo::height` each pixel (I use the term pixel rather than texel to conform to the GL spec) having a 32-bit RGBA value. Research GL commands `glCreateBuffers()` and `glNamedBufferStorage()`.
4. Set `Pbo::ptr_to_pbo` to point to the PBO's address by calling `glMapNamedBuffer()`.
5. Call `Pbo::clear_color_buffer()` to fill PBO memory store with the value in `Pbo::clear_color`.
6. After `Pbo::clear_color_buffer()` returns, release PBO's pointer in `Pbo::ptr_to_pbo` back to the GPU driver by calling `glUnmapNamedBuffer()`.
7. **Important Note:** Steps 4 through 6 are the necessary steps for the emulator to write to graphics memory - the writes by the emulator to PBO must always be fenced by calls to `glMapNamedBuffer()` and `glUnmapNamedBuffer()`.
8. Initialize `Pbo::texid` by creating a texture object with storage for a texture image having the same dimensions as the PBO (which are both equivalent to the dimension of the GL context's colorbuffer). Research GL commands `glCreateTextures()` and `glTextureStorage2D()`.
9. Now, initialize the texture image with the contents of the PBO. This is tricky - you'll need to carefully research GL command `glTextureSubImage2D()`.
10. Before the texture image can be rendered, `Pao::vaoid` must be initialized with vertex buffers containing 2D position and texture coordinates for a full-window quad. The position vertices must be defined in NDC coordinates. This has been thoroughly investigated in previous labs.
11. There's one last object that must be initialized before the texture image can be rendered - `Pbo::shdr_pgm` must be initialized with a shader program object. Read the interface for type `GLSLShader` in `./include/glslshader.h`. Remember, shader source files must always be located in `$(SolutionDir)shaders`. I've included sample vertex and fragment shaders that should suffice for this Lab. The vertex shader `./shader/pass_thru_pos2d_tex2d.vert` doesn't do much - it just writes an input NDC vertex to `gl_Position` and passes the texture coordinate unchanged to subsequent stages. The fragment shader `./shader/basic-texture.frag` obtains a color by sampling the texture image attached to `Pbo::texid`. Please refer to Lab 12 (from CS 200) for additional details on texture mapping.

4. Now, we're ready to render by adding the appropriate functionality to `Pbo::render()`.
    1. In the future, this function will be augmented with the 3D graphics pipe emulator. For now, to get images similar to `./Samples/lab-1-with-pbo.exe`, compute the RGBA value for each frame using time (obtained from GLFW) as the domain to `sin()` and `cos()` functions whose outputs (which can be negative values) are mapped to range $[0, 255]$. The mapped values are assigned to the R and G components with B component set to zero. You can do whatever you like with these color components with the only condition being that the color displayed by the window must change every frame.
    2. Use `Pbo::set_clear_color()` to set the color to fill the PBO. Next, map PBO address to client address space by calling `glMapNamedBuffer()` and assigning the return value to `Pbo::ptr_to_pbo`.
    3. Call `Pbo::clear_color_buffer()` to fill the PBO with the previously computed color.

    4. Release the address returned by `glMapNamedBuffer()` back to the graphics driver by calling `glUnmapNamedBuffer()`.
    5. Use `glTextureSubImage2D()` to DMA the image in PBO to `Pbo::texid`'s texture image store.
5. Conclude the assignment by providing a definition for `Pbo::cleanup()`. The implementation must return VAO, PBO, and texture object resources back to the graphics driver.
6. Comment the preprocessor macro `#define NOT_USING_PBO` in `main.cpp`. Follow this macro in `main.cpp` to get an understanding of how the streaming of images is being implemented by the application. Compare your results with the sample executable `./Samples/lab-1-with-pbo.exe`.

# References

There is not much documentation available on the web about best practices involving streaming data from the client into the GL server. The method described in this document is a good first attempt - however, your games might require a more efficient streaming technique. Luckily, the chapter from OpenGL Insights dealing with transfers between GL clients and servers is available for download from here. This is the best resource available if you wish to investigate this particular topic in greater detail for your current or future games.

# Submission Guidelines

1. Check the course web page for the submission deadline.
2. You must submit source files `glslshader.cpp` and `pbo.cpp`.
3. Make sure to not alter any of the other files (except `glslshader.cpp` and `pbo.cpp`) that have been provided as part of this Lab. Any changes may make your code not compile, link, and/or execute - in which case, I'll be unable to assign a grade.
4. Open the rubric page (in Word format or in PDF format) and check the portions that are complete. Sign and date the bottom of the page. Scan the page into a PDF or JPEG or Word document. I cannot grade your assignment if the signed rubrics page is missing from your submission.
5. Copy the submission source files and the signed rubric page into a submission folder that is named using the following convention: <**student login name><class><lab#>.zip**. Assuming your login is **foo**, the course obviously is **cs250**, the lab is **1**, then the zipped folder would be named: **foo_cs250_lab1.zip**.