Tutorial 6: Setting up Software Emulator for Graphics Pipe in OpenGL

Learning Outcomes

- 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 viewport transform that maps from *Normalized Device Context* (NDC) coordinate system to window coordinate system.
- Be able to set up a Vertex Buffer 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.

Getting Started

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev with a new version available on the assessment's web page. Execute the batch file. Choose option **6 - Create Tutorial 6** to create a Visual Studio 2022 project tutorial-6.vcxproj in directory build with source in directory projects/tutorial-6 whose layout is shown below:

```
csd2101-opengl-dev/      # OpenGL sandbox directory
├─ build/                  # Build files will exist here
├─ projects/             # Tutorials and assignments
|  └─tutorial-6          # Tutorial 6 code exists here
|     └─ include          # Header files - *.hpp and *.h sfiles
|        └─ glhelper.h
|        └─ glpbo.h
|        └─ glslshader.h
|     └─ src              # Source files - *.cpp and .c files
|        └─ glhelper.cpp
|        └─ glpbo.cpp
|        └─ glslshader.cpp
|        └─ main-pbo.cpp
└─ csd2101.bat           # Automation Script
```

Source and header files are pulled into nested directory /projects/tutorial-6/src while header files are pulled into nested directory /projects/tutorial-6/include. Notice this tutorial In fact, this tutorial doesn't require the code in glapp.h and glapp.cpp. Instead the rendering that was earlier performed by `GLApp` will now be implemented by an emulator defined in `GLPbo` [in source file glpbo.cpp]. The purpose of this assignment is to provide a definition of `GLPbo` that can then

be used to rasterize lines and triangles. The main.cpp is now replaced by main-pbo.cpp. Compiling, linking, and executing your Tutorial 6 source code with main-pbo.cpp should display a window painted similar to the sample executable.

The remaining portion of this document explains the implementation of a functional off-screen color buffer in GPU memory that will be filled by line and triangle rasterizers implemented in `GLPbo`. Your task is to provide this implementation in header file glpbo.h and source file glpbo.cpp.

Conceptual Model for CSD2101 Emulator

CSD2101 is concerned with studying some of the mathematical elements and algorithms implemented by fixed-function stages of the graphics hardware. To implement these algorithms, 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 requiring the application to be 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, streaming 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. This is done through GL commands `glCreateTextures` and `glTextureStorage2D` [see texture mapping tutorial for more information about these calls]. Contents of the image store are uninitialized but will be updated by subsequent steps described below.
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`. This chunk of graphics memory will be our so called Pixel Buffer Object (PBO).
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` [which you'll define] to write to specific locations in the chunk of memory. To perform actions similar to `glClear(GL_COLOR_BUFFER_BIT)`, potential candidates from the C++ standard library include `std::fill` or `std::fill_n` or combinations of these fill functions with `std::memcpy`.

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.

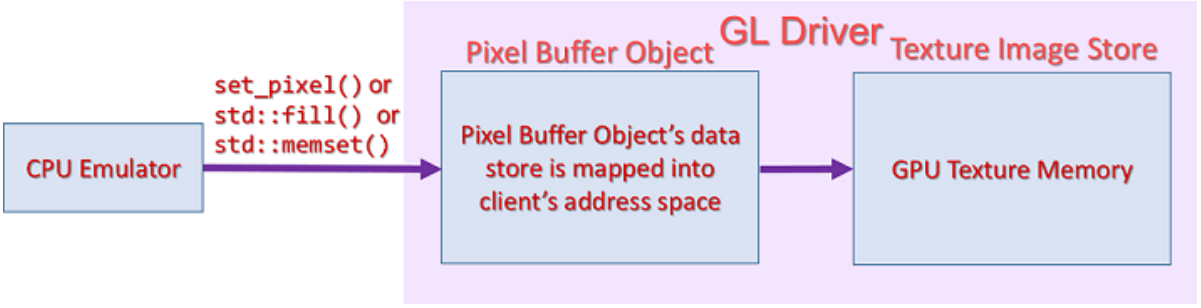

Figure 1: Conceptual view of pixel data upload from emulator to graphics memory

What is a Pixel Buffer Object?

Buffer objects are conceptually nothing more than chunks of server-side or GPU 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/glpbo.h. You'll have to provide the implementation in ./src/glpbo.cpp.

1. Scan through the definition of type `GLPbo` and identify `static` data members. Begin your implementation by providing definitions to these `static` data members in glpbo.cpp.
2. Define overloaded functions `GLPbo::set_clear_color` that set `static` data member `GLPbo::clear_clr` with 32-bit RGBA values specified by function parameters. These functions emulate the behavior of GL command `glClearColor`.
3. Define `GLPbo::clear_color_buffer`. This function emulates GL command `glClear(GL_COLOR_BUFFER_BIT)` by using pointer `GLPbo::ptr_to_pbo` to fill the PBO's data store with RGBA value in `GLPbo::clear_clr`. Since you'll write millions of bytes, think of how this can be done as efficiently as possible - potential candidates from the standard library include `std::fill` or `std::fill_n` or combinations of these fill functions with `std::memcpy`.
4. Implement function `GLPbo::init`:
    1. In this tutorial, we assume that each element of the PBO's data store represents a $32-$bit RGBA pixel with each component of size one byte. Likewise, we assume each texel of the texture object's texture image is a $32-$bit `GL_RGBA` value with each component of type `GL_UNSIGNED_BYTE`.

2. We further assume that the PBO's data store and the texture object's texture image will have the same dimensions as the GL context's framebuffer. Since all of these memories are storing values of the same type [$32-$bit RGBA], these three data stores will be equivalent except that the GL context's framebuffer is double-buffered.
3. Begin by assigning appropriate values to `static` data members `GLPbo::width`, `GLPbo::height`, `GLPbo::pixel_cnt`, and `GLPbo::byte_cnt`:
    1. `GLPbo::width` and `GLPbo::height` must be equivalent to `GLHelper::width` and `GLHelper::height`, respectively.
    2. Data member `GLPbo::pixel_cnt` specifies the number of pixels in the PBO's data store [and texels in the texture image] and should therefore be equivalent to $GLPbo :: width \times GLPbo :: height$.
    3. Data member `GLPbo::byte_cnt` specifies the size in bytes of the PBO's data store [and texture image] and should therefore be equivalent to $GLPbo :: pixel\_cnt \times 4$
4. Set the PBO fill color [to white or whatever other color you wish] by calling function `GLPbo::set_clear_clr`. Recall that this function will assign the fill color to data member `GLPbo::clear_clr`.
5. Create a texture object with storage for a texture image that is exactly equivalent to the size of the PBO's data store. Just as in $\text{Tutorial } 5$, first call GL command `glCreateTextures` to get a handle in data member `GLPbo::texid` to a texture object of type `GL_TEXTURE_2D`. Next, call GL command `glTextureStorage2D` to allocate storage for the texture image.
6. Create a PBO using the steps described here.
7. To render the texture image created from the colors written into the PBO's data store, data member `GLPbo::vaoid` must be initialized as a handle to a VAO that encapsulates a full-screen quad with appropriate position and texture coordinates. Define member function `GLPbo::setup_quad_vao` to implement this initialization. This setup has been thoroughly investigated in previous tutorials. The only thing of note is to remind you that the full-screen quad's position coordinates must be specified in NDC coordinate system.
8. Finally, you'll need to initialize the shader object `GLPbo::shdr_pgm` that will contain the shader program that will render the texture mapped full-screen quad. Define a function `GLPbo::setup_shdrpgm` to do just that. This setup has also been thoroughly investigated in previous tutorials. Since the shader program will perform a specific task of rendering a full-screen texture mapped quad, the vertex and fragment shaders don't require frequent edits. Therefore, you'll not be submitting text files containing vertex and fragment shader source files. Instead, it is simpler to define the shader source in C or C++ strings. The vertex shader 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 obtains a color by sampling the texture image attached to `GLPbo::texid`. The texture mapping tutorial provides the nitty-gritty details of writing these shaders.

5. Implement function `GLPbo::emulate`:

1. This function will be used by our graphics program to emulate the behaviors of vertex processor, rasterizer engine in graphics hardware, and fragment processor. This function will be called once per frame to write a color for every element in the PBO's data store [whose handle is `GLPbo::pboid`] and then to transfer the entire data store's contents to the texture image storage [whose handle is `GLPbo::texid`].
2. Use function `GLPbo::set_clear_color` to set the PBO fill color. You've implemented something similar in $\text{Tutorial } 1$ to smoothly vary the color between frames. One method is to provide a varying value such as the current time [easily obtained from GLFW] as the domain to functions `sin` and `cos` whose outputs [which can be negative values] are mapped to range $[0, 255]$. You can make up whatever color you wish - the only caveat is that the displayed color smoothly changes every frame. Run the sample to see what you're aiming for.
3. Introduce three key events: R, G, and B. While the background color is smoothly transitioning, pressing and holding key R should change the background color to **Red** 🔴. Similarly, pressing and holding key G should change the background color to **Green** 🟢, and pressing and holding key B should change the background color to **Blue** 🔵. Releasing any of these keys will revert the background to its default smooth color transition. Run the sample and interact with the application to see the desire key event behavior.
4. Map the PBO data store's address [meaning the address of the data store's first element] to client address space by calling `glMapNamedBuffer` and assigning the return value to `GLPbo::ptr_to_pbo`.
5. Call function `GLPbo::clear_color_buffer` to fill every pixel in the PBO's data store with the color previously assigned to `GLPbo::clear_color` in the previous steps.
6. After function `GLPbo::clear_color_buffer` returns, release PBO's pointer in `GLPbo::ptr_to_pbo` back to the graphics 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`. Otherwise, you could be writing to the PBO while the GPU driver is reading from it at the same time. This will cause images to be rendered with tears and other artifacts.
8. Now, initialize the texture image with the PBO's data store by using `glTextureSubImage2D` to DMA the image in PBO's data store to `GLPbo::texid`'s texture image store. This is tricky - you'll need to carefully research GL commands `glTextureSubImage2D` and `glBindBuffer`.

6. Implement function `GLPbo::draw_fullwindow_quad`:

    1. As implied by the function's name, this function will render a full-screen quad with the texture image generated by the final step of function `GLPbo::emulate`.
    2. Recall that `GLPbo::vaoid` is a handle to the full-screen quad's position and texture coordinates stored in a VBO. Also recall that `GLPbo::texid` is a handle to the texture object whose texture image will be generated every frame by function `GLPbo::emulate`. The details of rendering a full-screen quad have been extensively discussed in previous tutorials starting with $\text{Tutorial } 1$. The details of writing the vertex and fragment shader that will render a texture mapped triangle have been discussed in $\text{Tutorial } 5$. You'll combine these details from previous tutorials to render the image generated by function `GLPbo::emulate`.

3. Augment this function with what you've doing in earlier tutorials: printing information in the window toolbar that prints your name [in place of mine] but is otherwise *exactly* similar to the sample executable.

7. Conclude the assignment by providing a definition for function `GLPbo::cleanup`. The implementation must return VAO, PBO, and texture object resources back to the graphics driver.

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. This page gives an overview of the different streaming methods. Techniques for asynchronous transfers using double-buffered PBOs are briefly discussed here.

Submission

1. Create a copy of project directory tutorial-6 named <login>-<tutorial-6>. That is, if your Moodle student login is foo, then the directory should be named foo-tutorial-6. Ensure that directory foo-tutorial-6 has the following layout:

```
foo-tutorial-6          # You're submitting Tutorial 6
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
```

2. *Zip the directory* to create archive file foo-tutorial-6.zip.

⚠ Before Submission: Verify and Test ⚠

- Copy archive file foo-tutorial-6.zip into directory test-submissions. Unzip the archive file to create project directory foo-tutorial-6 by typing the following command in the command-line shell [which can be opened by typing cmd and pressing Enter in the Address Bar]:

```
powershell -Command "Expand-Archive -LiteralPath foo-tutorial-6.zip -DestinationPath ."
```

- After executing the command, the layout of directory test-submissions will look like this:

```
csd2101-opengl-dev    # Sandbox directory for all assessments
├─ test-submissions   # ⚠Test submissions here before uploading
|  └─ foo-tutorial-6 # foo is submitting Tutorial 6
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
└─ csd2101.bat              # Automation Script
```

- Delete the original copy of foo-tutorial-6 from directory projects to prevent duplicate project names during reconfiguration.
- Run batch file csd2101.bat and select option R to reconfigure the Visual Studio 2022 solution with the new project foo-tutorial-6.
- Build and execute project foo-tutorial-6 by opening the Visual Studio 2022 solution in directory build.

- Use the following checklist to ***verify and submit*** your submission:

Things to test before submission	Status
Assessment compiles without any errors	☐
All compilation warnings are resolved; there are zero warnings	☐
Executable generated and successfully launched in Debug and Release mode	☐
Directory is zipped using the naming conventions outlined in submission guidelines	☐
Zipped file is uploaded to assessment submission page	☐

> ℹ️ ***The purpose of this verification step is not to guarantee the correctness of your submission. Instead, it verifies whether your submission meets the most basic rubrics [it compiles; it doesn't generate warnings; it links; it executes] or not and thus helps you avoid major grade deductions. And, remember you'll need every single point from these programming assessments to ensure a noteworthy grade!!! Read the section on Grading Rubrics for information on how your submission will be assigned grades.***

Grading Rubrics

The core competencies assessed for this assessment are:

- [***core1***] Submitted code must build an executable file with ***zero*** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing. In other words, if your source code doesn't compile nor link nor execute, there is nothing to grade.
- [***core2***] This competency indicates whether you're striving to be a professional software engineer, that is, are you implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [are file and function headers properly annotated?]. Submitted source code must satisfy ***all*** requirements listed below. Any missing requirement will decrease your grade by one letter grade.
    - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
    - Source code files submitted is correctly named.
    - Source code files are *reasonably* structured into functions and *reasonably* commented. See next two points for more details.
    - If you've created a new source code file, it must have file and function header comments.
    - If you've edited a source code file provided by the instructor or from a previous tutorial, the file header must be annotated to indicate your co-authorship and the changes made to the original or previous document. Similarly, if you're amending a previously defined function, you must annotate the function header to document your amendments. You must add a function header to every new function that you've defined.
- [***core3***] Texture generation, transfer, and display is complete using pixel buffer objects.

- [***core4***] Print information in window toolbar *exactly* similar to sample executable .
- [***core5***] Cleanup tasks to return VAO, PBO, and texture object resources back to the graphics driver are correctly implemented in function `GLPbo::cleanup`.
- [***core6***] Framerate should be similar to my sample.

Mapping of Grading Rubrics to Letter Grades

The following table illustrates the mapping of core competencies listed in the grading rubrics to letter grades:

Grading Rubric Assessment	**Grade**
There is no submission.	F
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If ***core2*** rubrics are not satisfied, final letter grade will be decreased by one [e.g, from A to B].	
core3 rubric is complete.	B
core4 rubric is complete.	$B+$
core5 rubric is complete.	A
core6 rubric is complete.	$A+$