CG Programming II – Assignment #1 (Vertex Fetcher)

In this assignment you will use several pieces of functionality of the OpenGL API and of the hardware vertex fetcher. This will include the use of vertex array objects, instanced rendering, and asynchronous buffer mapping.

Part 1: Due on 16-January-2013 at the start of class

- Download the base code and get it to compile.
- Modify the provided vertex shader, `simpler.vert`, to use `mvp` and `mv_normal` as attributes (using the `in` modifier) instead of as uniforms. Use the `layout` qualifier to place these attributes at locations 4 and 8, respectively.
- Modify the `Init` function in `main.cpp` to create a new buffer object that is large enough to hold all of the instance transformation data that is calculated in `Redisplay`. These are the `transforms` and `normal_transforms` arrays.
- Modify the `Redisplay` function in `main.cpp` to use `glBufferSubData` to copy the instance transformation data into the buffer object created in `Init`.
- Modify `Redisplay` to set the attribute pointers for the 8 attributes using `glVertexAttribPointer`. Be *very* careful about the values used for the attribute base offset and stride.
- Modify `Redisplay` to set the instance divisor to 1 for each of the 8 attributes using `glVertexAttribDivisor`.
- Replace the `glDrawElements` loop in `Redisplay` with a single call to `glDrawElementsInstanced`.

Part 2: Due on 16-January-2013 at the end of class

- Modify `Init` to create and initialize a single vertex array object to encapsulate all of the attribute settings.
- Use the new vertex array object in `Redisplay` instead of resetting all of the data every frame.

Part 3: Due on 23-January-2013 at the start of class

At each step rendering should work and look the same. The only thing that may change is the performance.

- Modify the buffer object created to hold instance data to be large enough to hold instance data for several frames (e.g., make it 3 or 4 times as large as it currently is).
- Modify `Redisplay` to map a subrange of the where data for the current frame should be placed using `glMapBufferRange`. The `access` parameter should be `GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_RANGE_BIT`. Initially, map the buffer once per frame, and always use an offset of 0. This allows you to leave the rest of the code unmodified. Write the transformation data directly to the mapping. Before rendering be sure to call `glUnmapBuffer`!
- Modify the code to write the data for each frame to a previously unused location in the buffer. To do this, track the offset in the buffer of the first free location. Initially this will be zero, and each frame it will increment by the amount of data written to the buffer. When the offset is too close to the end of the buffer to hold all of the data, reset it back to zero.

  Since the location of the instance data in the buffer changes each frame, the attribute pointers will also need to be modified each frame.
- Modify the previous change to only be used if the implementation does not supprot the `GL_ARB_base_instance` extension. If the implementation does support that extension, use `glDrawElementsInstancedBaseInstance` instead of `glDrawElementsInstance`. Carefully set the `baseinstance` parameter so that the correct transformation data is used.

- Modify the `glMapBufferRange` call to also use `GL_MAP_UNSYNCHRONIZED_BIT` for every mapping *except* when the buffer offset wraps around to zero.
- Modify the code the calculates the transformation data to split the calculation into parts. First generate data for and render one small set of instances. Then generate data for and render the remaining groups of instances. Play with the sizes (and count) of the groups to see what happens to overall performance. For example, there are 42 total instances. Generate data for 5 instances and start them drawing. Then divide the remaining 37 instances into two groups of 13 and one group of 11. In pseudo code, you can probably implement this like

```
const unsigned total_instances = 42;
const unsigned first_group_size = 5;
const unsigned other_group_size = 13;

for (unsigned i = 0; i < total_instances; /* empty */) {
  unsigned group_size = (i == 0) ? first_group_size : other_group_size;

  if (i + group_size > total_instances)
    group_size = total_instances - i;

  for (unsigned j = 0; j < group_size; j++) {
    /* generate transformation data for this group. */
  }

  glDrawElementsInstanced( ..., group_size);

  i += group_size;
}
```

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).