LSU EE 7700-2 **Homework 3** Solution **Due: 31 March 2008**

The solution template for Problem 1 of this assignment is available through SVN at https://svn.ece.lsu.edu/svn/gp/hw/2008/hw3 .

Problem 1: Modify the code in the solution template so that it uses triangle strips to specify the triangles making up the tube. (See the OpenGL documentation for use of triangle strips.) The homework template is written so that it will be easy to switch between triangle strips and independent triangles (once the triangle strip code is written). The triangle vertices using triangle strips (the added code) must be at the same coordinates as those with the individual triangles (the existing code). That is, when switching between the two (using the "t" key) there should be no change in the appearance of the tube.

- The code must support all three buffering methods (see the `opt_v_buffering switch` statement).
- It will probably be necessary to emit multiple strips, each for a ring of triangles.
- The code should continue to write all vertices in array `coor_buffer` before passing them to OpenGL, however vertices might be written in a different order when triangle strips are in use.
- For `opt_v_buffering` options 1 and 2 the entire `coor_buffer` and `norm_buffer` should be given to OpenGL, even if multiple triangle strips are needed. (See the documentation for `DrawArrays` in the OpenGL specification.)

For the solution see the code in the repository at https://svn.ece.lsu.edu/svn/gp/hw/2008/hw3/hw3sol.cc.

Problem 2: Consider the performance benefit of using triangle strips in the previous problem.

(*a*) Show the change in performance between triangle strips and the original code for each of the three methods of vertex transfer.

The table below shows the timing for the six combinations of vertex buffering and triangle specification. The GPU column shows the time per frame used by the GPU, the CPU column shows the time per frame for the CPU. `IT` indicates individual triangle and `TS` indicates triangle strips. See the solution of the next part for a Normalized Time definition.

SOLUTION TABLE

Vertex Buffering		GPU μs	CPU ms	Data MB/s	Normalized CPU Time (TS/IT)
Individual	IT	2.8	3.4	26.3	
Individual	TS	2.2	2.6	8.8	0.767
Cl Array	IT	2.5	3.0	26.3	
Cl Array	TS	2.1	2.5	8.8	0.833
Buffer Obj	IT	2.2	2.4	0	
Buffer Obj	TS	2.1	2.4	0	1.000

(*b*) For each of the three methods of vertex buffering, either explain the observed change in performance or if you find the change in performance inexplicable, explain the change in performance

that should have been obtained. If you haven't gotten the first problem working just explain what you thing the change in performance should be and why.

Since the CPU takes so much more time it will be used to measure performance changes.

The Normalized Time column shows the normalized CPU time for the triangle strip code. The 0.767 indicates that when specifying vertices individually the time needed using triangle strips is 76.7% of that using whole triangles. For client arrays, the triangle time is 83.3% of whole triangles, a smaller improvement. For buffer objects there is no improvement at all.

In all cases the number of triangles rendered is the same and so performance changes must be due to other differences.

The use of triangle strips reduces the number of vertices specified by a factor of 3 (since each vertex is part of 3 triangles).

For individual vertices, triangle strips reduces the number of glVertex calls by a factor of 3, for the other vertex buffering schemes there is no change in the number of OpenGL calls. The overhead in handling vertices one at a time might be the reason individual vertices buffering enjoys the largest reduction in time, by a factor of 0.767.

For both individual vertices and the client array, triangle strips reduce the amount of data sent to the GPU by a factor of 3. (The buffer object size will also be reduced by a factor of 3, but since that's only sent to the GPU once the transfer should not affect performance.) That would probably account for the 0.833 reduction enjoyed by the client array buffering.

With buffer object buffering there is no change in CPU time with triangle strips, probably because the use of triangles strips changes only the command send from the CPU to the GPU. The GPU shows an improvement of $0.1\,\mu$s, which is probably due to random variation but could be due to a reduction in the amount of data read from GPU memory or perhaps to the GPU transforming each shared vertex just once instead of 3 times. (See the next problems.)

Problem 3: Using triangle strips will certainly reduce the amount of data passed to OpenGL API calls and it will certainly not reduce the amount of data written to the frame buffer.

(*a*) By what factor does the use of triangle strips reduce data passed to OpenGL API calls for the first problem? (An answer of 0.5 indicates that half the data is transferred?) (This part is easy.)

By a factor of 3 for individual and client array buffing. For the buffer object the initial data transfer is reduced by a factor of 3 but that is only during initialization, after that there is no change.

(*b*) Consider a GPU and OpenGL implementation in which the amount of data passed from the CPU to the GPU is the same whether or not triangle strips are used.

How might that be the fault of the OpenGL implementation (say, if the the OpenGL implementation was grudgingly developed by a company with its own proprietary GPU API)?

The OpenGL code running on the CPU might have converted the triangle strips to individual triangles and sent those over to the GPU, eliminating any reduction in data transfer. For example, for a call to `glVertex` the implementation would re-send the last two vertices plus the one specified in the `glVertex` call (instead of just sending over the coordinates specified in the `glVertex` call).

How might that be due to the design of the GPU? Assume that the GPU is moderately well designed but either old or targeted at a particular part of the market. Be reasonably specific on why a GPU might make it impossible to reduce the amount of data passed.

A reasonable GPU would implement triangle strips by doing what the OpenGL implementation was described as doing in the last part. That is, the GPU would remember the last two vertices received. When a new one arrived it would send the last two vertices plus the new one down the rendering pipeline.

A low-cost GPU might not have the storage or a smart enough control processor to do this and so the OpenGL implementation would have to re-send vertices from the client.

Problem 4: Consider the GPU designs presented in class and the use of triangle strips from the first problem. The use of triangle strips will certainly reduce the amount of data passed to OpenGL and certainly **not** reduce the data written to the frame buffer. How far down the GPU hardware (rendering pipeline) can the reduction be seen? That is, at what point will the reduction disappear?

A well designed pipeline would not duplicate vertices until primitive assembly (after vertex processing and before clipping and interpolation). One way to do this would be to write the output of the vertex processor to three different primitives. If this were done the reduction in work would be seen until just before clipping.