Assignment 2 – Graphics hardware algorithms

The purpose of this exercise is to implement some algorithms in a software framework that could be used in graphics hardware.

1. Getting started

Download the code from the assignment's website, and unpack the zip-file. Start Microsoft Visual Studio by clicking on the `LUR.sln` file. In the file `rasterizer.cpp`, take a look at the function `rasterizeTriangle()`, and follow to the functions `inside()`(and look also in `edgefunc.cpp`) and `perFragment()`. This is a very basic, very simple rasterization algorithm. Note that `setup()` is called before `rasterizeTriangle()` renders the triangle.

Note that you can press the R key or press the right mouse button and select "toggle Rasterizer" in the menu to switch back and forth between the software rasterizer we have developed for this assignment and using the graphics hardware you have in the PC. This function can be used to make it simpler to solve the tasks below. However, it should be noted that hardware rendering has been "slowed down" so that it is possible to compare the two modes.

For problem I, II, and III you only need to add code in `rasterizer.*` and `edgefunc.*`.

2. Problem I

Compile and run the program. You should see a window with 64 triangles emanating from the center of the screen. This simple scene is currently rendered using the code that you looked at in "Getting Started" above. One artifact from the software rasterizer should be very apparent. It is your task to identify the artifact, and to correct for this in the code. You should solve this so that the artifacts disappear.

3. Problem II

Start the program again. Press the key "2" to switch to an even simpler scene consisting of 10 triangles. Now press space in order to make these triangles translate slowly to the right. Compare to hardware rendering again. Your task is to identify a new artifact, and to correct for that in the code. It may be impossible to get exactly the same behaviour, but the annoying artifacts should be removed.

4. Problem III

Start the program again. Press the key "3" to switch to a scene containing two triangles textured with a checkerboard image. Press "space". Identify a third artifact, and make the necessary changes in the code to give the rendering the right appearance.

5. Reducing texture bandwidth usage

The purpose of this task is to reduce bandwidth usage to texture (i.e., texture reads) using a texture cache. If you are not familiar with caching, read the text about "General Caching", Section 5, available online. Read also the following two papers:

1. "The Design and Analysis of a Cache Architecture for Texture Mapping" by Hakura and Gupta, ISCA'97. http://graphics.stanford.edu/papers/texture_cache/
2. "Prefetching in a Texture Cache Architecture", by Igehy et al. Workshop on Graphics Hardware 1998. http://graphics.stanford.edu/papers/texture_prefetch/

Start the program, and press the key "4" to switch scene. Press space to start the animation. Let it run until it is finished, and look at the bandwidth measurements. Explain the ratio of "Color buffer BW" divided by "Texture reads". Note that a color write costs 32 bits, and a texel costs also 32 bits. The system does not currently use a cache.

Now, add a texture cache. Do this in glstate.cpp. There is code that you can uncomment in order to do this. Run the program again. Note that the size of the cache is written into the console window. In the following experiments you should keep the size of the cache constant. The constructor used to create a texture cache is:

```
cTextureCache(int nSets, int nEntries)
```

This class uses a random replacement strategy, and caches 4x4 texels per entry, and it uses 6D blocking as described by Igehy et al (see reference above). You can change the associativity by changing the constructor parameters. Remember to keep the cache size constant. Watch what happens to the texture read bandwidth (BW).

Your task is to reduce the texture read bandwidth as much as possible without increasing the size of the texture cache. The parameters you can change are:

- Associativity of the cache (done using the constructor as shown above).
- Use a tile-based rasterizer instead of a scan-line-based rasterizer. In LUR.cpp, you can change from using the class cRasterizer() to cTileRasterizer()
- Change tile size for the rasterizer (also done in LUR.cpp). You need to use powers of two for the resolution of the tile.

You need to be able to explain your results. To make it simpler, note your results for different parameters somewhere.

In order to pass, you need to be able to reduce texture bandwidth to (about) only 13.5% (<1.9 MB) of a non-cached system. If you're good you can make it come below 11.5% (< 1.6 MB).