CIS 441 Project 3C

5 May, 2021

Thomas Mitchell

Choices made in my program: I chose the radii to be .3 because if they were bigger, the balls would collide with each other, and unlike in 3B, we weren't asked to implement collisions, so this would look weird. I also chose to color the balls according to which third of distance away from the camera they were, which made it easy to know I implemented the sort correctly, and also gives my program some color in the end. The only drawback is that because this is discrete, the color change is abrupt. Although the instructions didn't specify this, I decided to make my final program re-spawn each ball from the height where it initially came from. I made this decision to "make the program mine" as well as because I think it looks more visually pleasing that way. I was not able to test this on the provided Linux VM because the framerates were extremely low (in many cases below one frame per second), but I don't foresee any compatibility/portability issues.

Initial observation:

Using the settings from the spec, my average FPS is about 36. It depends, however, on how many balls are within view of the camera, I believe. For instance, at the very beginning, when the balls are initially dropped, that's when I see my highest FPS of 42. This must indicate that even though all balls (and therefore triangles) are ordered to be drawn by OpenGL, the GPU doesn't waste the effort. These figures are for dropping 100 balls. Since my GPU is low-end, I would need to choose a much lower number for it to be binding at 60 fps at this level of quality. My overall throughput must be 36 * (2 + 33.3 * (4^7 + 4^6 + 4^5) = 25,779,067 triangles per second. I'll explain in experiment 3 how I determined the per-sphere formula. Naturally, to get the units to be in per second, I needed to multiply by the FPS; I added 2 for the plane, which is constant.

Experiment 0:

Before changing the number of balls dropped, I think it would be more interesting to check if setting all of the balls to the highest default level of recursion (7) would cause a noticeable drop in fps. I was quite surprised at how much slower my framerate was after making this change. It falls to an average of just below 17, after an initial fps of about 19.5 (when balls are mostly out of view of the camera for those measured frames). The throughput with these settings would be just under 17 * (2 + 100*4^7) = 27,852,834. This is consistent with my result from the initial observation, since I rounded up in this one.

Experiment 1:

Let's reverse things: low-quality models for all balls. Other than seeing the fps improve, I will also try to judge whether the quality has decreased in my eyes (I could not see a difference in the last test). This caused the program to max out its frames, and I could not notice a difference in the quality of the balls. It is worth noting at this point that my balls each have a radius of .3, so maybe they're too small to see a difference on. I'll take a look at this in my next test.

Experiment 2:

I'll be taking a look at if using larger balls while maintaining the low-quality model for all balls will cause my program to not hit the FPS cap, as well as judge if the balls closer to the camera look sloppy after using exclusively the low-quality model. After doubling the radii to .6, this test proved to be uninteresting. Still hits FPS cap, and I personally can't see any indication that the balls (even closer ones) would look better using a higher-quality model. I decided to run another variant of this test (original spec, but with larger radius) since it was inconclusive whether the radius had an impact on FPS. It does not, which indicates that the number of triangles slows us down without much regard to how many pixels must be filled in for each one. Knowing a model of a ball that uses more triangle recursions doesn't use more pixels, this makes sense, but when we remember our VTK projects, bigger triangles would lead to more iterations of our loops. The mechanism by which the GPU and OpenGL driver operate must, therefore, not spend much of its time actually writing pixels to a (frame)buffer. More likely, this is done by different components/pipelines concurrently, so it is of no time consequence for the video output process as a whole.

Experiment 3:

I noticed that changing the recursion level of the balls seemed to have an exponential effect on the FPS between my first and second experiments. This is easy to reconcile: every level of recursion means a triangle is split into four, so the number of triangles per sphere should be 4^(levels of recursion). To prove exceeding 7 levels of recursion, even if only for the closest third of the balls, will cause the model to become much slower, I will test if changing just the L3 definition from 7 to 9 will cause a significant drop versus the initial observation. I will be changing the radii back to .3.

After my program seems to have hanged after changing the L3 definition to 9 (although I think it would continue if given enough time), I decided to experiment with an L3 of 8 instead. This resulted in an FPS average approaching 11.5 after an initial of 12.5. This is MUCH lower than the average of 36 for the L3 = 7 model. Triangle throughput is about 11.5 * (2 + 33.3 * (4^5 + 4^6) + 33.3 * 4 ^ 8) = 27,057,715.2, consistent with the other experiments so far (FPS was a bit more than 11.5).

Experiment 4:

Another experiment I'd like to conduct is seeing what the minimum level of recursion is for the closer balls before I am no longer satisfied with their quality. Seeing as rendering all balls to the L1 quality of 5, I will be examining setting the closest third of balls to 3 levels of recursion. To my surprise, I ended up needing to try 2 levels of recursion instead because I could not see a lack in quality after setting the levels of recursion to 3. This trend continued until I set the level to just 1. The close balls looked like hexagons after that. I can safely assume that with zero levels, they would be just triangles. The FPS cleanly maxed-out at 60, which served as a reminder that the closest balls, despite having just two levels of recursion more than the next closest third, constitutes a majority of the balls' rendering effort (I left L2 and L1 as their default levels of 5 and 3, respectively for this test, and only modified L3).

Experiment 5:

The last experiment I'm conducting is to see if doubling the number of balls from 100 to 200 while keeping the other settings the same as my initial observation has a linear effect on FPS. The number of triangles should change linearly with the number of balls/spheres, but we also have to render the plane, as well as additional overhead of just running the program. Nonetheless, I predict the FPS will be about half after doubling the number of balls, especially since the plane's triangle count of 2 is insignificant. With an average approaching just under 19.5, the 200 ball model operates at about 54% the speed, indicating a linear relationship. This suggests that the program, before any triangles are added, has more overhead than I predicted (since the loss was only 46%; just under half). In my hypothesis, I predicted the overhead to not be significant enough to measure, so it was slightly wrong. The triangle throughput here would be 19.5 * (2 + 66.6 * (4^5 + 4^6 + 4^7)) = 27,927,283.8. This is almost a million more than the average of the other experiments.