CIS 441/541 Project 3C

Level-of-Detail: Tradeoff Between Performance and Rendering Quality

May 30, 2021

Introduction and Motivation

As discussed extensively this term, rasterization is a process which takes triangles as input and creates as output an image that has pixels colored based on triangle overlaps. Since this process is roughly O(#triangles), the more triangles, the longer this process takes.

For this project you will consider performance when rendering 100 spheres. The spheres are approximated with triangles, and the number of triangles used can vary. Approximations with more triangles have finer facets and look more sphere-like. That said, more triangles increases rendering time. In all, there is a tradeoff between quality and performance — more triangles may mean long render times, but fewer triangles may look bad. How can we balance this?

A common technique is to adapt how many triangles are used based on the sphere position. In other words, spheres close to the camera should be rendered at a high resolution, and those far away with low resolution. This will ensure that spheres that appear the closer to the camera appear more "sphere-like" without degrading the performance.

In this project, you will be testing this trade-off with a number of bouncing balls and a moving camera. Specifically, you will be sorting the balls into 3 levels: L1, L2, and L3. The closest third of the balls to the camera will be assigned L3, the second third to L2, and the furthest third will be given L1.

Physics of a bouncing ball

Disclaimer: You do not have to understand the physics of a bouncing ball - just use the equation provided

This project will have you implement a part of a bouncing ball. The physics of an actual bouncing ball is quite complicated, but the following equations should get you a decently accurate bouncing ball. The starter code includes a `struct Ball_physics` with the following parameters: $h_0, v, g, t, dt, rho, tau, hmax, h,\ htop, freefall, t\_last, vmax$. This struct is then wrapped in a `struct Ball` that contains other information about the ball. Here's some info about the ball physics parameters.

- h_0: initial height

- v: current velocity
- g: gravitational acceleration
- t: the starting time
- dt: time step
- rho: coefficient of restitution
- tau: contact time for the bounce
- $hmax$: the maximum height of current bounce
- h: current height
- $hstop$: the stopping height (i.e. if height is less than stopping height assume that the bouncing is over)
- $freefall$: boolean for whether the ball is bouncing or not
- t_last: time we would have launched to get the ball at h_0 at t.
- $vmax$: max velocity

Initializing the ball physics

In this project, you will be tasked with initializing the ball physics parameters of the ball. Below are equations that you will need.

- $v, g, t, dt, rho, tau, hmax, htop$ are pre-populated so you don't have to worry about them (you are free to play with them if you want)
- When you initialize your ball to some height, say X...
  - set $h = h_0 = hmax = X$
  - set $t\_last = -1 \cdot \sqrt{2 * h_0/g}$
  - set $vmax = \sqrt{2 \cdot hmax \cdot g}$

Updating the ball physics

There are a couple of scenarios we have to cover when updating the ball physics. 1) The ball is in free ball (imagine when the ball is first dropped), 2) The ball is not in free ball and is in contact with the ground (imagine the exact moment when the ball bounces), 3) The ball is not in free fall and is bouncing up, 4) The ball is at rest on the ground.

Sorting through the physics can all get quite confusing, so the part of the code to update the height and its associated parameters **while the ball is in flight** is provided. However, you will have to add the code to make a new ball after the ball has stopped bouncing. The exact implementation is up to you but, in essence, you will have to reset ball physics and re-initialize it with new starting parameters.

Instructions

1. Download `project3C.cxx, CMakeLists.txt`. This will be your starter code.
2. Make sure you can compile it! Run the program and make sure you see something like the below image. Also, you should also see FPS output in your console.
3. Copy over your Phong lighting code from 2A into the shader program.
4. Unlike Project 2 where the RenderManager had one sphere, this code will have three different types of spheres of differing resolutions. In other words, instead of calling `RenderManager::SPHERE`, you will be calling `RenderManager::SPHERE1`, `RenderManager::SPHERE2`, or `RenderManager::SPHERE3`. Note that `SPHERE3` should be the most detailed and `SPHERE1` should be the least detailed.
    - Notice the macros defined at the top, `L1,L2,L3,numBalls`
    - `L1,L2,L3` describe how detailed the balls should be at each level. `L3` defines the level of detail for `SPHERE3` etc. Note that a higher number means more detail. So, setting `L3=8` instead of `L3=7` will render spheres with more triangles and thus with more detail when `rm.Render(RenderManager::SPHERE3...)` is called.
    - `numBalls` should be self-explanatory, but it will define how many balls to have in your environment.
5. Complete the code to initialize the balls. This will amount to populating the `std::vector<Ball> balls`.
    - For the `x,z` fields of the ball, pick a random number between -10 and 10
    - For the initial height pick a random number between 10 and 40.
    - C++ has a number of random number generators - search engines are your friend!
6. Complete the second half of the `UpdateBallPhysics()` function. This is when you will need to replace balls that have stopped bouncing.
7. Complete the `BounceBall()` function. Here you should update each ball's location according to its `ball_physics` and render them accordingly. Remember, the closest third to the camera should be rendered at `L3`, the second third at `L2`, and the farthest third at `L1`.

8. At this point, you should have a compile-able program, when run, shows a number of bouncing balls with a rotating camera.

9. Now, increase `L3` and change `numBalls`. Is there a point where the FPS changes? Can you reason about the trade-off between resolution and rendering time?

    - The FPS that you see will depend on your system, GPU etc. If you are using a virtual box, the initial setup may already be laggy.
    - **NOTE!:** GLFW caps the framerate at 60FPS. There are ways around it but for this project, we will keep the cap at 60FPS. So, you shouldn't expect to see FPS higher than 60FPS. But, FPS will drop under 60 at higher loads.

10. Submit two things:

    (a) the completed code `project3C.cxx`

    (b) a short write-up of your findings on FPS. Were you able to hit a maximum number of triangles per second? Any relationships where you could increase the geometry size and the frame rate would drop accordingly? This writeup should be about a paragraph and should include at least five experiments. Each experiment should vary some combination of the number of balls, the resolution for L1, L2, and L3, or the proportion of balls that are rendered at various resolutions. Note that you may need to work for a while to build a mental model of how many triangles your GPU can render in a second. Finally, your report should be meaningful. Do not just run a few tests, collect numbers, and report the numbers. Instead, try to learn about GPU performance and show how your experiments inform that performance. If your report is not meaningful, then you should expect a significant deduction in points.

What you should see first