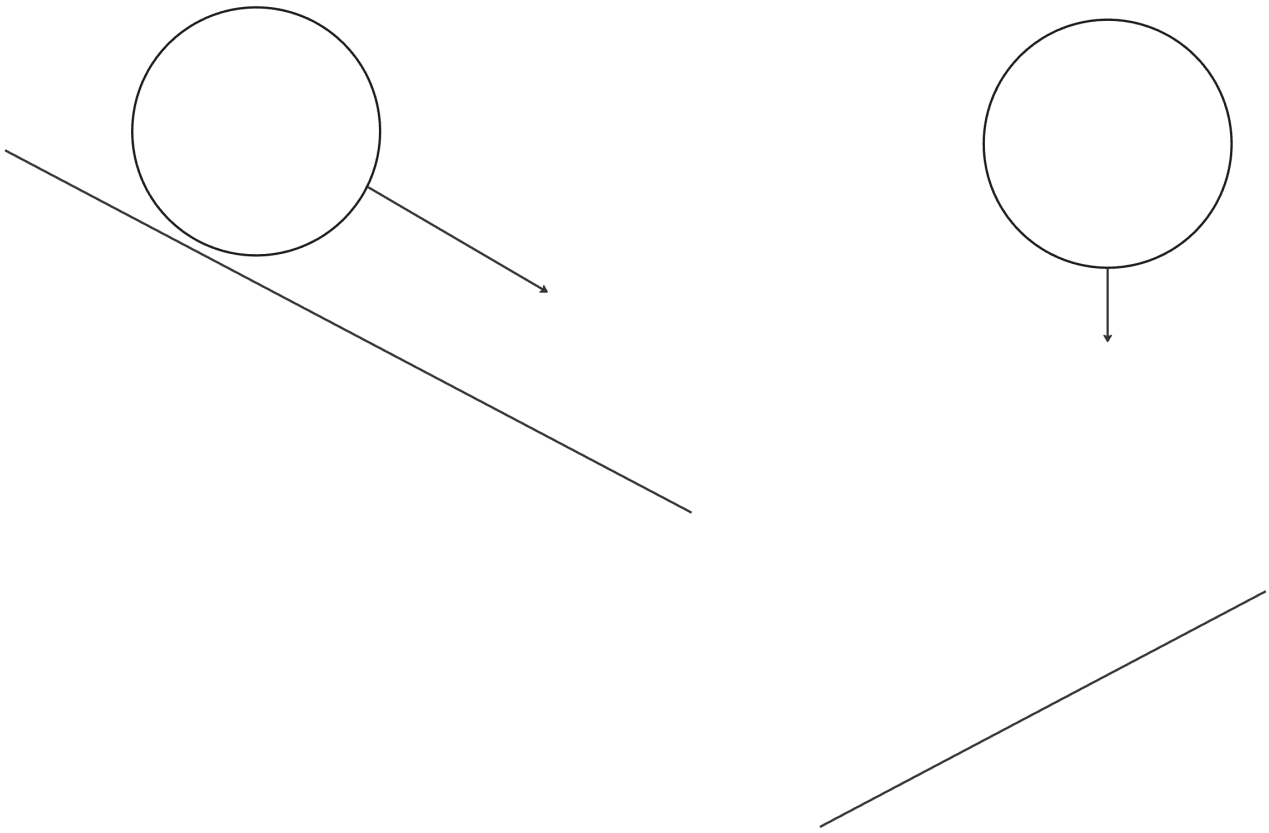
Merlin Roller Cpp Takehome

Merlin Labs Take-home interview: Free-Roller Sim in C++

Please do not post online or share
Last updated: July 6th, 2023 for Jackie Chan



Object-oriented multiple balls multiple slopes discrete time scenario. Your are creating a library for modeling multiple balls impacting multiple slopes.

Logistics

1. Thank you for taking the time to do this test. This test forgoes "leet code" that you might otherwise encounter during an onsite.
2. You can contact your interviewer with questions and they should get back to you within 2 hours (work days), or 6 hours (weekends).

3. Please email your zip to your interviewer and recruiter by the deadline listed in your email.
4. Interviewers will review your results before your onsite. Questions about your work will be a major part of an onsite. You can still expect technical lines of questioning during your onsite.

High-level

1. Note: We're still calibrating this assignment, but the ultimate goal is for this to take not much more than one and half weekend days of effort. Help us by tracking your time spent, please!
2. Implement the program in C++, any ISO flavor.
3. The program can only depend on the C and/or C++ standard libraries.
4. Any consumer of your code should be able to instantiate balls and slopes and attach to a simulation.
5. When you turn in, please set to run at 50hz `delta_t` and print the debug description for all balls at each time step for 10 seconds. Have it configured for a scenario with two balls and two slopes.
6. If the code is implemented in multiple source files it include a CMakeLists for compilation of it.
    1. You can implement it in one source or one source and one header file, if you like.
7. Use git commits as you would in your normal development environment.
8. Zip the whole folder and turn it in.
9. We ask that you keep this PDF confidential and not ask other humans to review or contribute to your work.
10. You may use generative text systems (Co-Pilot, Llamas, Chat etc.).
11. Please include a readme with instructions for building and running
    1. Please also explain the tools you utilized in the creation of your project (e.g. what did you use Chat or Co-pilot for). Also, what environment did you use to build and test?
    2. What limitations does your solution contain? If able, please comment on discrete-time errors relative to continuous time (i.e. real life) with varying discretization size. Notice anything odd? How would you characterize the error. (Hint: Pay particular attention to how a ball starts rolling and describe that.)
    3. We are calibrating this test: if you have feedback on its length or enjoyment relative to other tests, or suggestions, we welcome them. Thank you!

World objects

1. The world is 2 dimensional. Define Y is positive up (i.e. gravity is negative) and X is positive to the right (bottom left corner of your screen would be `0,0`).
2. It's not necessary to provide support for modifying the locations of the world objects.
3. Include a print method that prints object's location, and other state properties.

Slopes

1. Defined with two world-coordinate points (in meters `[m]`), a start and an end coordinate.
2. Have zero thickness.
3. Only provide normal force on balls perpendicular to their slope. There is no need for special forces computed for balls impacting the end-points of the slopes.
4. Are not affected by gravity or impacts and are never moved.

Balls

1. Defined with a starting coordinate (`[m,m]`) and a radius (`[m]`) and a mass (`[kg]`).
2. Have a linear velocity (in `m/s`)
3. Have an angular velocity (in `radians/s`)
4. Include a qualification print description matching: `ball # (x=xx, y=yy)(vx=vx, vy=vy)(wz=v_angular)`

Simulation

1. Should be initialized with the world objects and `delta_t`.
2. Run time length should be managed somehow within the simulation.
3. Please record the first and last time each ball impacts each slope. A method should accept a ball and print the times in seconds `ball # slope # (first=s, last=s)`.

Physics Equations

- The linear acceleration of a ball depends on the sum of forces acting on the ball
  - `a = f_sum/mass` f_sum_x what are the axes?
  - `v_new = v_last + a * delta_t`.
  - Perform on each axis independently, e.g. `a_x = f_sum_x / mass`

- The angular acceleration of the ball depends on the sum of torques acting on it
  - `a_angular = torque_sum/inertial_moment`
  - `intertial_moment = 2/5 * mass * radius^2`
  - `torque = radius * force` or `torque = radius.cross_product(force)` and see the accompanying free body diagrams for more details.
  - `v_angular_new = v_angular_old + a_angular * delta_t`
- The ball is accelerated down by gravity (use `9.8 m/s^2`)
- Normal force `f_normal` is perpendicular to a slope and prevents a ball from moving through the slope.
  - `f_normal = normal.dot(f_sum) / normal.length` ref
  - `normal = (-slope.dy, slope.dx)` length = magnitude aka sqrt(dx^2 + dy^2)
- Static friction exists when the ball is not slipping (velocity at the surface of the ball equals the linear speed of the ball) and serves to prevent sliding motion. It depends on `f_normal` and is no more than the gravitational force along the slope.
  - `v_surface = v_angular * radius`
  - `f_friction_static_max = f_normal * k_friction_static` (use steel on steel `.74` ref)
  - `f_friction = min(f_friction_static_max, f_gravity_along_slope)`
- Dynamic friction exists when the ball is slipping (velocity at the surface of the ball is not equal to the linear speed of the ball) and serves to reduce sliding motion
  - `f_friction_dynamic_max = f_normal * k_friction_dynamic` (use steel on steel `.42`)
  - `f_friction_dynamic = min(f_friction_dynamic, f_friction_such_that_sliding_will_be_eliminated_this_timestep)`
- Friction at the ball's radius will become torque that accelerates the ball's rotation
  - IF static friction is greater than the component of gravity along the slope AND the velocity at the surface of the ball equals the linear speed of the ball THEN the ball rolls without slipping and only static friction applies;
  - OTHERWISE, only dynamic "slipping" friction can apply.
  - Note: even with contact with a slope, a ball might not experience friction if e.g. the ball is rolling without slipping and not accelerating.
  - Note: Because of numerical considerations in discrete-time, you will probably want to use `abs(v.length - v_angular * radius) < k_difference_same_number` or similar to see if the linear speed of the ball matches its speed at the surface.
- When a ball impacts the surface of a slope (e.g. `height_above_surface < k_difference_same_number`), the velocity into the slope (normal to the slope) is

canceled (i.e., the energy is absorbed by deformation of a metal ball and slope)

- There's a geometric calculation for `height_above_surface`
  - `height_above_surface = closest_point_on_slope.distanceTo(ball.center) - ball.radius`
  - `closest_point_on_slope = distance_closest_point_on_slope * slope.slope + slope.start`
  - `distance_closest_point_on_slope = slope.slope.dot(segment_slope_start_and_point) / slope.length` ref
  - `segment_slope_start_and_point = line(slope.start, ball.center)`
- `f_normal = f_n_stopping + f_n_resisting_gravity + f_n_restoring`
  - The direction of `f_normal` is out of the slope... see the free body diagrams.
- `f_n_stopping` can be calculated as the momentary normal force (over one timestep) that will cancel the velocity into the slope at that timestep
  - `f_n_stopping = -1 * m * (slope.normal.dot(v) / slope.normal.length) / delta_t`
- `f_n_resisting_gravity` can also be included in `f_normal`
  - `f_n_resisting_gravity = -m * slope.normal.dot(g) / slope.normal.length`
- Due to the dynamics of the simulation, this force is also an opportunity to restore the ball to the surface of a slope that may have been trespassed due to numerical considerations in discrete time
  - `f_n_restoring = k_restoring * min(0, height_above_surface)`
  - You can start by trying a `k_restoring` of `(1) [kgm/s^2/m]`
- Note that increased static or dynamic friction (and thus torque and angular acceleration) will apply due to the application of this force.

Suggestions

1. Make classes for all fundamental types including points/coordinates. This test evaluates object-oriented thinking, and concepts including interface privacy.
2. Make a few test cases for yourself. Here are two examples:
   1. Initialize a ball with velocity along a flat surface; the ball should roll forever after angular acceleration picks up to match rotation at the surface with a linear speed.
   2. Balls of different inertial moments should roll different speeds down a slope.

3. Some of the names of world objects (especially slope) are confusing because they shadow mathematical conventions. You can name in your code as you please!

Example Animated SVG

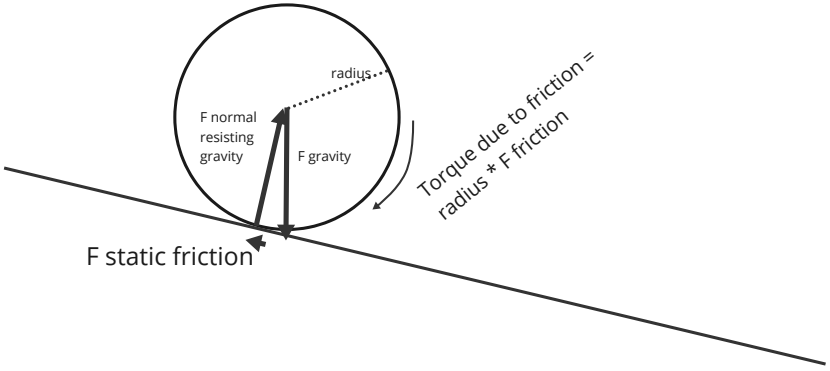
Here's an example `.SVG` you can open in (at least) *Chrome*. You could render `svg`'s yourself in the simulation to visualize your work, minus ball rotation (or perhaps you could figure out how to represent that! Color?).

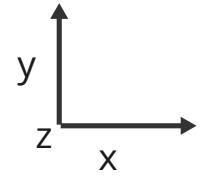
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg width="100%" height="100%" viewBox="0 0 120 120"
preserveAspectRatio="xMidYMid meet" xmlns="http://www.w3.org/2000/svg">
<g transform="translate(0,100)">
<g transform="scale(1,-1)">
<line x1="0" y1="50" x2="100" y2="0" stroke="black" stroke-width="1" />
<line x1="0" y1="0" x2="100" y2="50" stroke="black" stroke-width="1" />
<circle id="ball-0" cx="10" cy="50.9961" r="1" fill="red">
<animate attributeName="cx" from="10" to="15" dur="1.02s" begin="0.02s">
</animate>
<animate attributeName="cy" from="51" to="45" dur="1.02s" begin="0.02s">
</animate>
<animate attributeName="cx" from="15" to="26" dur="1.02s" begin="1.04s">
</animate>
<animate attributeName="cy" from="45" to="38" dur="1.02s" begin="1.04s">
</animate>
<animate attributeName="cx" from="26" to="26" dur="0s" begin="2.08s">
</animate>
<animate attributeName="cy" from="38" to="38" dur="0s" begin="2.08s">
</animate>
</circle>
</g>
</g>
</svg>
```

Other ***highly*** optional tasks

If you somehow have more energy/time:

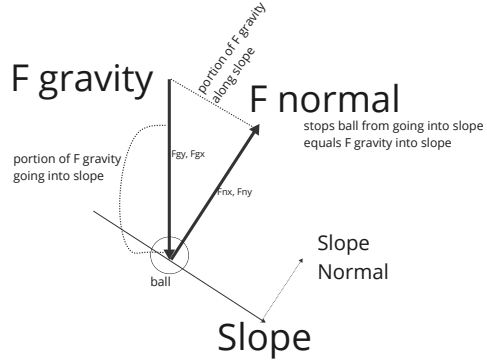
1. Make the collisions of ball on slopes elastic instead of energy dissipating. If you do this, please make sure there's a way to turn the feature off as well. :-)





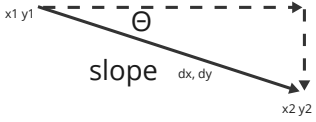
z is toward you!

Forces have just direction and magnitude



Slopes have start and end points

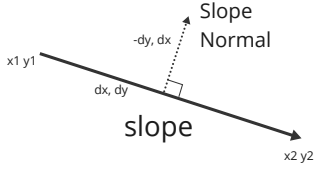
diff x = x2 - x1 = cos (Θ) * length



diff y = y2 - y1 =
sin (Θ) * length

length = sqrt(dx^2 + dy^2)

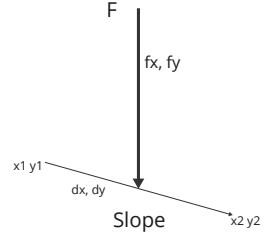
Normals are a direction defined like this

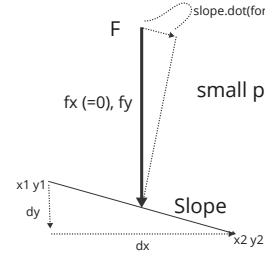


directions just have the difference in x & y

Get the amount of a force along a direction like this

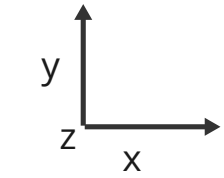
amount of force along direction = direction.dot_product(force) / direction.length =
(dx * fx + dy * fy) / sqrt(dx^2 + dy^2)



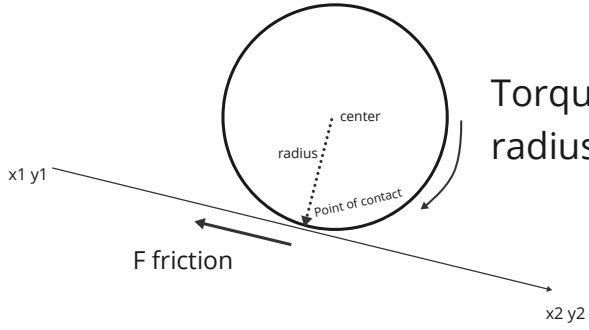


small proportion along slope in this case

There's a more general way to calculate sign of torque (+ or -) using cross products

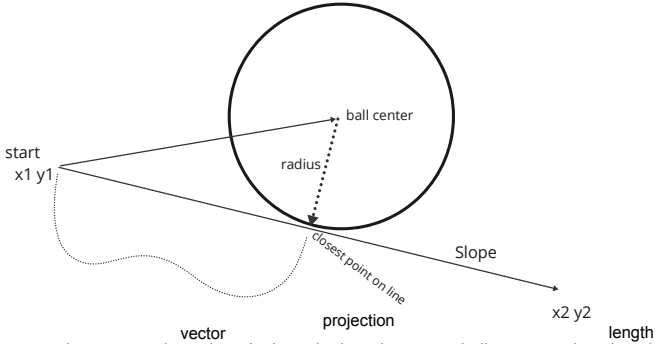


z is toward you!

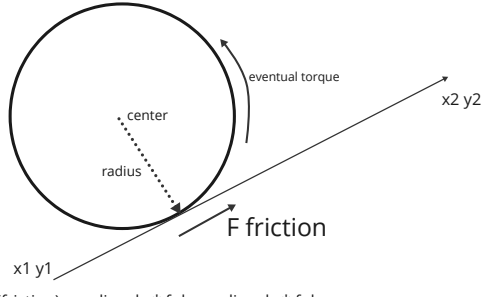


Torque due to friction = radius X F friction

Calculate closest point on line then draw the line from center radius distance to the point



closest point = slope.start + slope.slope * (slope.dot(line(slope.start, ball.center)) / slope.length)
radius line = line(ball.center, closest point).unit_direction * ball.radius
closest point - ball center



torque = radius.cross(friction) = radius.dx * f.dy - radius.dy * f.dx