CSCI 5611 Instr.: Guy

HW 1: Collision Detection Library

Homework: Fri., Sept 22

Overview. This assignment is intended to prepare you for Project 1 by developing a library for detecting collisions between geometric primitives such as circles, squares, and lines. Knowing when objects are overlapping is one of the foundational steps for a wide variety of simulation and gaming tasks such as detecting when a player is touching something in their environment or determining when two objects in a physics simulation have collided and should bounce off each other.

This is an individual assignment. Each person most complete their own code.

For this homework, you must write a library that detects overlap (collisions) between the basic geometric primitives of line segments, circles, and axis-aligned boxes.

You may use any programming language you want (besides Python), but you may not important any libraries, modules, or existing code. This means you likely will want to write your own Vec2 functionality (fell free to use mine as a basis).

This HW has two parts:

A. Collision Detection Library

Write a collision detection library. This library must have separate functions for determining collisions between a pair of circles, line segments, and/or axis-aligned boxes. Additionally, there must be a function that can report any collisions that occur in a scene.

You may use any algorithm you like for detecting collisions, but make sure you consider any corner cases (e.g., a circle completing overlapping a square). If you use any external resources to develop your code, you must cite these resources and provide links within a *readme.txt* file.

B. Scene Processing

Ten files named *task1.txt* through *task10.txt* are available on the course webpage. Each of these files list one or more circles, lines, and boxes. Each of these primitives will be given a unique ID. You must write code that leverages your collision avoidance library (from Part A), in order to find any primitives that are colliding.

Your must output a file with the time it took to find all collisions (in milliseconds) along with a list of the IDs of every colliding object n the following format:

Line 1: “Duration: xxx ms”, xxx is the time in milliseconds
Line 2: “Num Collisions: xxx”, xxx is the number of unique colliding primitives
Line 3+: The ID of colliding primitives, in numerical order, one per line

For example, for the file *task1.txt* your results should look like

```
Duration: 0.000625 ms
Num Collisions: 2
0
1
```

because the circle (ID 0) and the line (ID 1) collide with each other. The file must be named *task[NUM]_solution.txt*, where *[NUM]* is the task number. The timing only needs to include the time needed to find collisions, it does not need to include loading the file or writing out the results.

Your code must work with any of the 10 files provided but does not need to generalize to any other file.

Submission [100 points]

Submit 3 separate files: readme.txt, solutions.txt, code.zip, solutions.zip.

1. The file *readme.txt* should have a short description of the approach you took to finding collisions, any resources you used, and a brief note of any optimizations you used.

2. The file, *code.zip,* should have both your code for computing the output of the collision detection library, and your code for finding the optimal value. [note: you do not need to use processing, but you must supply all files needed to compile your code]

3. The file *solutions.zip* should contain exactly 10 files, each named *task1_solution.txt* through *task10_solution.txt*, with each file following the exact format described in part B.

Extra Credit [up to 10 points]

Graph the time it takes your code to find all collisions as a function of the number of obstacles in the scene. For extra credit, you must demonstrate that you've implemented a method whose runtime grows sub-quadratically.

Grading

Your project will be graded on both speed and accuracy. Speed will account for 10% of the grade and for full credit, each of the provided tasks must complete in last than 1s (1000 ms). If your code is too slow, you can improve your performance by using a better algorithm, switching programming languages, or finding a faster computer. Accuracy will account for the other 90% of the grade. For grading purposes, both speed and accuracy will be determined by the results provided in your submitted *solutions.zip* file, so be sure you can reproduce these results exactly if asked.