Computer Graphics

Assignment 1: Moving Circles

First, refer to the general instruction for assignment submission, provided separately on the course web. If somethings do not meet the general requirements, you will lose corresponding points or the entire credits for this assignment.

1. Objective

In computer graphics, the majority of drawing primitives are triangles. The goal of this assignment is to learn how to draw and animate such primitives (here, 2D). Precisely, you need to draw and move 2D colored circles on the screen, while avoiding the collisions among them and screen boundaries (as walls).

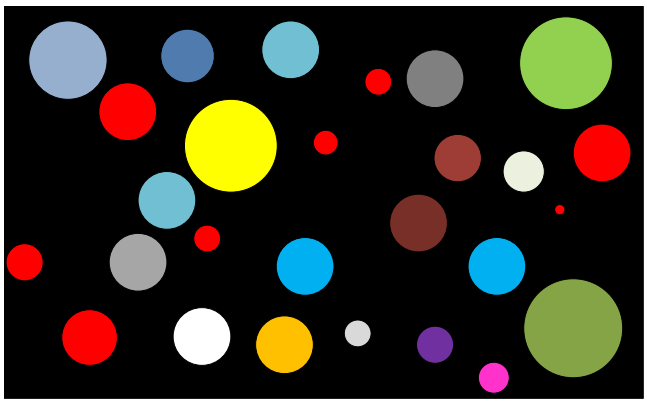

Figure 1: An example screenshot of a single animation frame.

2. Triangular Approximation of Circle

In general, a circle can be represented by its center position $\mathbf{p}_0 = (x_0, y_0)$ and radius r, for example:

$$(x - x_0)^2 + (y - y_0)^2 = r^2, \tag{1}$$

which we call an *implicit function*. Such an implicit representation, however, is not directly supported in GPU.

Hence, we first need to convert it to a finite set of triangle primitives. A common way of doing this is to approximate the circle by subdividing it to a number of equal-size triangles. This process is called the "tessellation" or "subdivision."

Figure 2 illustrates how to approximate a circle using eight triangles, resulting in an octagon. For your implementation, use more triangles (e.g., 36 or 72 triangles) to obtain a smoother boundary; the more triangles you use, the smoother appearance you get.

The positions of edge vertices can be found as follows. Given the origin $\mathbf{p}_0$—typically $(0, 0)$—and r, a boundary vertex $\mathbf{p}$ can be:

$$\mathbf{p} = \mathbf{p}_0 + r(\cos\theta, \sin\theta),$$

where θ indicates the angle at the xy-plane. To pass the position to your shader, you also need to set $z = 0, w = 1$, because you vertex shader recognizes only a 4D position. You will learn later (in the Transformations) why we set the fourth component $w = 1$; for the moment, do not care about it.

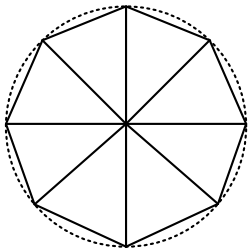

Figure 2: An octagonal approximation of a circle.

Also, recall that the default viewing volume is provided with $[-1, 1]^3$ for the x, y, and z axes. So, when you define a unit circle, use $r = 1$. Then, transform its vertices using a model matrix in the vertex shader.

3. Requirements

Requirements for the assignment are as follows. You may start from "cgcirc" project, when available from the course web; if so, many of them are already available in the sample.

- Initialize the window size whose aspect ratio (width/height) is $16/9$ (e.g., 1280×720). As long as your window is not a square, it is alright.
- The program has more than 20 solid circles (Figure 1) (10 pt).
- The initialization of circles uses random radii, colors, locations (10 pt).
- The initialization of circles uses random velocities (10 pt). Your circles should not be significantly slow.
- The circles' movement speeds should be consistent across different computers; use a time difference between consecutive frames to define the movement of circles (10 pt).
- The initial positions of circles avoid overlaps and collisions with the other circles and walls (10 pt).
- When a circle meet other circles or side walls, most of them (more than 90% of circles) can avoid collisions well (20 pt).
- You can avoid wrong collisions perfectly. This should be tested against the condition: *Rapidly moving 100 circles can avoid collisions without problems in 30 seconds*. "rapidly" means that the speeds of the circles are similar to the distributed sample. Your program should be able to set the number of circles to 100 with '+/-' key, and pass this test (20 pt).

Read the followings for additional descriptions and hints.

How to Avoid Initial Collision You probably define initial positions of circles randomly. Whenever you create a new circle instance, test if it collides with others. If the collision is found, repeat this step until no collisions are found.

However, when you create many circle instances, you may suffer from an infinite looping (meaning collision is always found). This may result from that you define a constant bound of circle sizes. In other words,your bound $[r_{\text{min}}, r_{\text{max}}]$) are fixed, regardless of the number of circles N.

One way to avoid the problem is scaling the bound depending upon the number of circles N. The effective areas of the circles are proportional to N^2. So, when you define the bound as follows, you will be able to avoid the problem:

$$[r_{\min}, r_{\max}] \propto \frac{1}{\sqrt{N}}. \tag{2}$$

Consistent Animation Speed When you animate circles, it is better to compute the amounts of movements based on the *time* difference between the current and previous frames rather than simply using a frame counter. When GLFW is used, use `glfwGetTime()` function to retrieve a timestamp. This way ensures that your application runs at the same speed in different machines.

It is also good to test your application in a different (e.g., your friend's or TA's) machine before submission. When you want to test your program in the TA's machine, contact the TA before submission, and make an appointment.

Physically-Based Elastic Collision Rather than relying solely on the random control over velocities, you can well handle the collision using a physically-based elastic collision. Refer to the link:

https://en.wikipedia.org/wiki/Elastic_collision

Better Collision Detection Without robust handling of continuous collision detection, you may end up with a few problems in the collision detection. The main reason you face with is the use of discrete collision detection. When circles are moving rapidly and their discrete sampling is sparse in time, your circles would penetrate others or do not escape from the collisions, leading to trembling of circles. In particular, when you simply reverse the velocities of circles, this becomes pronounced. Avoiding this problem completely is challenging and beyond the level of this class, because you need to find precise intersections predictively.

Instead, I suggest two simpler improvements.

- Use finer in-frame time steps.
- Use real-number measurement for the collision detection; i.e., do not use a *boolean flag* for the collision detection.

Read the following descriptions for more details.

The first approach is not using the time interval directly (derived from the time difference between the previous and current frames). Depending on the speeds of the circles, even a single frame can make huge leaps in their movements. So, subdivide the in-frame time interval into finer steps, and test collisions repeatedly, which may call update() more than once. This would reduce the problems, but trades performance for the quality of simulation.

The second approach is using a real number (e.g., floating-point number) that indicates how much circles overlap. Do not trivially reverse the velocities for a collision. Instead, use a continuous metric, while allowing penetration to some extents. For the collision test, you evaluate two conditions for the current frame and next frame. If the next frame has more overlaps, you need to change the velocities. Otherwise, just leave them as they move; the overlaps will be resolved automatically. The moments are short, so we will not perceive the little penetrations.

4. Mandatory Requirements

What is listed below is mandatory. So, if anything is missing, you lose 50 pt for each.

- Instancing should be applied for a single static vertex array object (VAO) or vertex buffer (VB). Your VAO should stay constant at run time, which means positioning and coloring of the circles should be implemented using *uniform* variables.

  In other words, do not create multiple or dynamic vertex buffers whose sizes and positions are defined in the host. Instead, you create a single VAO with a unit size (i.e., $r = 1$) located at the origin, and change its size and position using uniform variables in the vertex shader.
- In `render()` function, you need to call an OpenGL `draw` function as many as the number of circles, while you change the size, position, and other attributes for each circle.

Actually, all the distributed samples already use this strategy. So, if you start from the sample and use the same program structure, you will not have problems with this mandatory requirements. However, if you make your own way, be careful with this.

5. Advanced Topics for Extra Credits

The full score for the requirements and report is 100 pts for every assignment, but extra credits would be given when implementing what you did not learn in the course. The amounts of extra credits depend on its difficulty and importance. In general, adding new mathematical models is better than trivial implementation improvements; trivial improvements may not get any credits. In such a case, please emphasize them in the report so that we do not miss them. This policy applies to all the regular assignments, except team projects.

6. What to Submit

- Source, project (or makefile), and executable files (90 pt)
- A PDF report file: YOURID-YOURNAME-A1.pdf (10 pt)
- Compress all the files into a single archive and rename it as YOURID-YOURNAME-A1.7z.
- Use i-campus to upload the file. You need to submit the file at the latest 23:59, the due date (see the course web page).