Computer Graphics, 2020

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. In addition, you circles may avoid collision with other circles and screen walls.

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 *implicit function*. However, such an implicit representation is not directly supported in graphic hardware.

Hence, to draw the circle in raster-based graphics, 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 1 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.

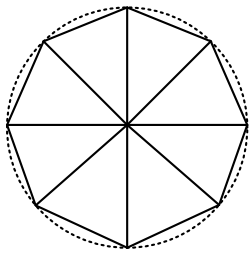

Figure 1: An octagonal approximation of a circle.

Given the $\mathbf{p}_0$ and r, you can compute the coordinates of each boundary vertex $\mathbf{p}$ as

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

where θ indicates the angle at the xy-plane. In your shader program, you will need to use 4D coordinates for `gl_Position`; in that case, use `vec4(`$x_0 + r\cos\theta$`,` $y_0 + r\sin\theta$`, 0, 1)`, where 0 indicates the z coordinate. For the moment, do not care about the last component 1; this will be explained later in the (transformation) lecture.

How to assemble the vertices to form triangles are already covered in the lecture.

Also, recall that the default viewing volume is provided with $[-1, 1]^3$ for the x, y, and z axes.

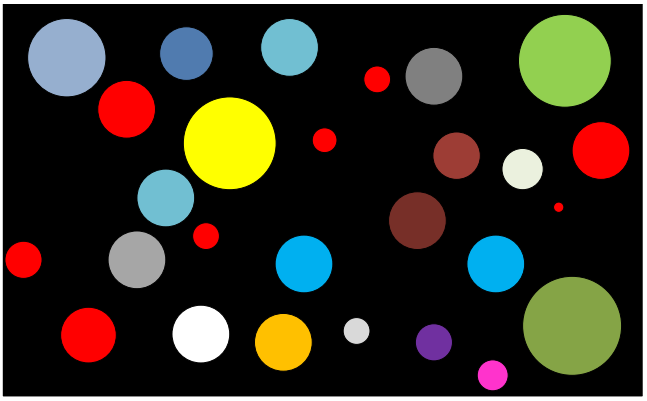

Figure 2: An example screenshot of a single animation frame.

3. Requirements

You may start from "cgbase" or "cgcirc" project, when available from the course web. Requirements for the assignment are as follows.

- If ESCAPE or 'q' keys are pressed, the program is terminated.
- Initialize the window size to 720×480.
- The program has more than 20 solid circles (Figure 2) (20 pt).
- Your circle should be a circle (*not an ellipse*), even when the window is being resized (10 pt).
- The circles are initialized with random position, radius, color, and speed (20 pt).
- The circles' initial positions require avoiding collisions or overlaps (10 pt).
- The circles are moving in random directions (10 pt).
- When a circle meet other circles or side walls, its direction and speed are changed to avoid overlaps (20 pt).
  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 below:

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

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.

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 buffer. Your vertex buffer 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 vertex buffers whose sizes and positions are defined in the host. Instead, you create a single vertex buffer 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.

5. 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).