Assignment 1: Moving Circles

Department : 소프트웨어학과
Student Id : 2020312914
Name : Minho Baek

Algorithm

Change the number of circles

init more than 16 solid circles

The number of circles is managed through the global variable `NUM_CIRCLES`. A global `std::vector` of the `circle_t` structure is used to store the parameters of each circle. During the `render()` step, the VAO is bound, and each circle is rendered individually.

change the number of circles up to 32

In the keyboard mapping function `keyboard()`, keyboard input is used to modify `NUM_CIRCLES`. When either `GLFW_KEY_KP_ADD` or `GLFW_KEY_EQUAL` is pressed, `NUM_CIRCLES` is increased, while pressing `GLFW_KEY_MINUS` decreases its value.

Initialize circle parameter

init circle random radii, color, location

To prevent certain parameters from becoming zero, a bias was added to the random source for all values except color. This was done to avoid zero-minima. The standard method used was `rand() / float(RAND_MAX)`, although using a utility function like `randf2`, as introduced during live coding, would provide a more convenient approach. For location, the expression `float ratio = float(window_size.x) / window_size.y` was used to determine the (x, y) coordinates.

init circle random velocities

Although velocity can be represented using the x and y components of a `vec2`, in my case, I additionally used separate values for speed and 2d-direction `theta` to simplify physical collision calculations. The speed is initialized as a random value with a lower bound to avoid being too small, and `theta` ranges $(0, 2\pi)$.

speed consistency across different computers

At the end of each frame, the current time was recorded using `t = float(glfwGetTime())`, and at the beginning of the `update()` function, a timestamp was calculated as `ts = float(glfwGetTime()) - t`. This timestamp `ts` was then used in all movement updates to maintain speed consistency across frames.

Circle movement

avoid initial collisions

To avoid initial collisions or overlaps, a strategy was implemented using the `initCollision()` function. This function checks whether a new circle's position and radius would place it outside the screen bounds or cause it to intersect with any existing circles. A `while` loop repeatedly generates random positions until a valid, non-overlapping location is found.
To prevent the possibility of an infinite loop due to overcrowding, the minimum and maximum radius values (`r_min` and `r_max`) were set proportional to $\frac{1}{\sqrt{N}}$, ensuring that circle sizes decrease as the number of circles increases.

collision detection

Instead of using a simple boolean flag to detect collisions, the `object_collision()` function accumulates overlap values against all other circles and updates the motion only when the current overlap exceeds the previously recorded maximum. This allows for smoother and more continuous collision response across frames, helping to mitigate issues typically encountered in discrete collision detection methods such as missed or one-off reactions.
Here's simple pseudo code.

```
function object_collision(self, circles):
    max_overlap ← 0

    for each other in circles:
        if self is other: continue

        overlap ← compute_overlap(self, other)
        if overlap > max_overlap:
            max_overlap ← overlap
            update_velocity_and_theta(self, other)

    self.overlap ← max_overlap
```

Data Structure

- VAO
  Each circle is rendered using a shared VAO (Vertex Array Object), which stores the vertex attribute configurations. The VAO is bound once before rendering, and reused for drawing each circle. Within the render loop, `glDrawElements` is called to draw the tessellated triangles representing the circle. This setup enables efficient rendering of multiple circles with minimal state changes.
- circle_t
  The `circle_t` structure represents a 2D circle with properties required for both rendering and physics interactions. It includes position (`center`), size (`radius`), orientation (`theta`), color (`color`), velocity (`velosity`), and a transformation matrix (`model_matrix`) used for modeling. An `overlap` field stores the current collision state. Member functions such as `update()` , `object_collision()` , and `get_overlap()` define the circle's behavior and interaction logic.