600086 Lab Book

# Week 6 – Lab F

Date: 16th Mar 2022

## Q1. Particles

### Question: Create a particle system that can manage moving particles

### Solution:

Opened the empty particles project and created the Particle and Particle system structs these are as follows.

#[derive(Debug, Copy, Clone)]

struct Particle {

    x\_dir : f32,

    y\_dir : f32,

    x\_pos : f32,

    y\_pos : f32,

    velocity: f32

}

impl Particle {

    pub fn new( xd : f32, yd :f32, x:f32 , y: f32, v:f32) -> Particle {

        Particle{

            x\_dir : xd,

            y\_dir : yd,

            x\_pos : x,

            y\_pos : y,

            velocity : v,

        }

    }

    pub fn move\_particle(&mut self) {

        let x2 = self.x\_dir \* self.velocity;

        let y2 = self.y\_dir \* self.velocity;

        let test = self.x\_pos + x2;

        if test < 0.0 || test > 10.0 {

            self.x\_dir \*= -1.0;

        }

        let test = self.y\_pos + y2;

        if test < 0.0 || test > 10.0 {

            self.y\_dir \*= -1.0;

        }

        self.x\_pos += x2;

        self.y\_pos += y2;

    }

}

struct ParticleSystem

{

    particles: Vec<Particle>,

}

impl ParticleSystem

{

    pub fn new() -> ParticleSystem

    {

        ParticleSystem

        {

            particles: Vec::new(),

        }

    }

    pub fn run\_system(&mut self)

    {

        for particle in self.particles.iter\_mut()

        {

            let p = particle;

            p.move\_particle();

        }

    }

}

I then implemented the open GL code from last week in order to display the particles and show the movement. The implementation can be seen below, note only the render loop is shown

// Begin render loop

        // Animation counter

        delta\_t += 0.005;

        if delta\_t > 0.7 {

            delta\_t = -1.4;

        }

        // Create a drawing target

        let mut target = display.draw();

        // Clear the screen to black

        target.clear\_color(0.0, 0.0, 0.0, 1.0);

        //run particle system

        if T.elapsed().as\_secs()<10

        {

            p.run\_system();

        }

        // Iterate over the 10 triangles

        let particle\_list = &p.particles;

        let mut i = 0;

        for particle in particle\_list

        {

            // Calculate the position of the triangle

            let pos\_x : f32 = (particle.x\_pos - (WINDOW\_WIDTH/2.0))/(WINDOW\_WIDTH/2.0);

            let pos\_y : f32 = (particle.y\_pos - (WINDOW\_HEIGHT/2.0))/(WINDOW\_HEIGHT/2.0);

            let pos\_z : f32 = 0.0;

            //calculate colors

            let mut r = rand::thread\_rng();

            let red  : f32 = r.gen\_range(0.0..1.0);

            let green: f32 = r.gen\_range(0.0..1.0);

            let blue : f32 = r.gen\_range(0.0..1.0);

            // Create a 4x4 matrix to store the position and orientation of the triangle

            let uniforms = uniform! {

                matrix: [

                    [1.0, 0.0, 0.0, 0.0],

                    [0.0, 1.0, 0.0, 0.0],

                    [0.0, 0.0, 1.0, 0.0],

                    [pos\_x, pos\_y, pos\_z, 1.0],

                ],

                color: [red,green,blue,1.0 as f32]

            };

            i += 1;

            // Draw the triangle

            target.draw(&vertex\_buffer, &indices, &program, &uniforms, &Default::default()).unwrap();

        }

        // Display the completed drawing

        target.finish().unwrap();

        // End render loop

The position of the particles was kept within a 10 by 10 enclosure defined using the following global variables

const WINDOW\_HEIGHT: f32 = 10.0;

const PARTICLE\_SPEED:f32 = 0.04;

const WINDOW\_WIDTH: f32 = 10.0;

const LIMIT :u32 = 100;

this is then scaled to be represented on the screen as can be seen in the pos\_x variable set in the previous code.

### Test data:

N/A

### Sample output:

Can’t be shown as it is video but will present in Lab.

### Reflection:

Was fairly difficult handling the various mut, &mut, refs for the self variable. I kept getting ownership issues until I came across iter\_mut() function for the VEWc<Particle>

### Metadata:

### Further information:

N/A

## Q2. Threaded Particles

### Question:

Adapt the code from Q1 to utilise multiple scoped threads for calculating the particle movement

### Solution:

Added and adapted the following functions to the code in order to run the movement via scoped threads.

    let mut pool = scoped\_threadpool::Pool::new(NUM\_OF\_THREADS as u32);

…

// Limit the scope of the reads to this section of code

        pool.scoped(|scope| {

            for slice in p.particles.chunks\_mut(PARTICLES\_PER\_THREAD) {

                scope.execute(move || thread\_main(slice,10.0));

            }

        });

        // Implicit join here, where all threads go out of scope.

…

fn thread\_main (list: &mut [Particle], enclosure\_size: f32)

{

    for particle in list.iter\_mut()

    {

        let p = particle;

        p.move\_particle();

    }

}

This code splits the list of particles into slices with each slice being managed by a different thread.

### Test data:

### Sample output:

Can’t be shown as it is video but will present in Lab.

### Reflection:

This was a lot easier to implement then the part 1 was however there was some confusion about the variable names given in the example code such as the chunk size parameter of the chunks\_mut() function being a variable named NUMBER\_OF\_CHUNKS etc.

Between the two versions the running is barely noticeable between the two the threaded version seems fractionally faster but it is hard to tell.

### Metadata:

OpenGL

### Further information:

N/A