Code!

Hello to all who have found themselves here. You can find my and Dorian's Manim code below. I've also included a link here to the shadertoy project which I used to make the high resolution version of the "flip" image for Noah's section, the code for which is also listed here.

Manim

```
# note: import * is there for ease of use, specific imports for lsp
from manim import *
from manim import (
    np,
    config,
    MovingCameraScene,
    LEFT,
    RIGHT,
    UP,
    DOWN,
    ValueTracker,
    Dot,
    Line,
    linear,
    TracedPath,
    VMobject,
    ManimColor,
)
import random

# rendering constants
runtime = 60.0  # runtime of animation
simulation_time = 90.0  # time experienced by simulation
dt = simulation_time / (config.frame_rate * runtime)
dissipate_after = -1.0  # -1.0 for no dissipation
num_pends = 5
mode = "single"  # sequential, random, single

# physical constants
g = 9.81
l1, l2 = 2.0, 1.0
m1, m2 = 1.0, 1.0
theta1_i, theta2_i = 0.75 * np.pi, 0.75 * np.pi
dtheta1_i, dtheta2_i = 0.0, 0.0

#  #  #  #  #  #  #  #


# get derivative info for runge-kutta
def diff(state):
    theta1 = state[0]
    dtheta1 = state[1]
    theta2 = state[2]
    dtheta2 = state[3]

    delta_theta = theta1 - theta2
    M = m1 + m2
    alpha = m1 + m2 * (np.sin(delta_theta) ** 2)
```

```
    eq6 = (
        -m2 * l2 * (dtheta2**2) * np.sin(delta_theta)
        - m2 * l1 * (dtheta1**2) * np.sin(delta_theta) * np.cos(delta_theta)
        - M * g * np.sin(theta1)
        + m2 * g * np.sin(theta2) * np.cos(delta_theta)
    )
    ddtheta1 = eq6 / (l1 * alpha)

    eq7 = (
        M * l1 * (dtheta1**2) * np.sin(delta_theta)
        + m2 * l2 * (dtheta2**2) * np.sin(delta_theta) * np.cos(delta_theta)
        + M * g * np.sin(theta1) * np.cos(delta_theta)
        - M * g * np.sin(theta2)
    )
    ddtheta2 = eq7 / (l2 * alpha)

    return np.array((dtheta1, ddtheta1, dtheta2, ddtheta2))


# rk4 yass
def update_pendulum(theta1, dtheta1, theta2, dtheta2):
    state = np.array((theta1[-1], dtheta1[-1], theta2[-1], dtheta2[-1]))

    k1 = diff(state)
    k2 = diff(state + dt * (k1 / 2))
    k3 = diff(state + dt * (k2 / 2))
    k4 = diff(state + dt * k3)

    state += (dt / 6) * (k1 + 2 * k2 + 2 * +k3 + k4)

    theta1.append(state[0])
    dtheta1.append(state[1])
    theta2.append(state[2])
    dtheta2.append(state[3])


# double pendulum object
class DoublePendulum(VMobject):
    def __init__(self, theta1_i, theta2_i, color, **kwargs):
        VMobject.__init__(self, **kwargs)

        # values to keep track of
        self.theta1 = ValueTracker(theta1_i)
        self.hist_theta1 = [self.theta1.get_value()]
        self.theta2 = ValueTracker(theta2_i)
        self.hist_theta2 = [self.theta2.get_value()]

        self.dtheta1 = ValueTracker(dtheta1_i)
        self.hist_dtheta1 = [self.dtheta1.get_value()]
        self.dtheta2 = ValueTracker(dtheta2_i)
        self.hist_dtheta2 = [self.dtheta2.get_value()]

        # update function called once per frame
        def custom_update(self, **kwargs):
            update_pendulum(
                self.hist_theta1, self.hist_dtheta1, self.hist_theta2,
```

```
self.hist_dtheta2
            )

            self.theta1.set_value(self.hist_theta1[-1])
            self.theta2.set_value(self.hist_theta2[-1])

            self.dtheta1.set_value(self.hist_dtheta1[-1])
            self.dtheta2.set_value(self.hist_dtheta2[-1])

        self.add_updater(custom_update)

        # draw mass and arm
        self.mass1 = Dot(color=ManimColor(color))
        self.mass1.add_updater(
            lambda mob: mob.move_to(
                (l1 * np.sin(self.theta1.get_value())) * RIGHT
                + (l1 * np.cos(self.theta1.get_value())) * DOWN
            )
        )

        self.add(self.mass1)

        self.mass2 = Dot(color=ManimColor(color))
        self.mass2.add_updater(
            lambda mob: mob.move_to(
                self.mass1.get_center()
                + (l2 * np.sin(self.theta2.get_value())) * RIGHT
                + (l2 * np.cos(self.theta2.get_value())) * DOWN
            )
        )

        self.add(self.mass2)

        self.trace = TracedPath(
            self.mass2.get_start,
            stroke_color=ManimColor(color),
            dissipating_time=(dissipate_after if (dissipate_after != -1.0) else
None),
        )
        self.add(self.trace)

        self.origin = Dot()

        self.line1 = Line(color=ManimColor(color))
        self.line1.add_updater(
            lambda mob: mob.put_start_and_end_on(
                self.origin.get_center(), self.mass1.get_center()
            )
        )
        self.line2 = Line(color=ManimColor(color))
        self.line2.add_updater(
            lambda mob: mob.put_start_and_end_on(
                self.mass1.get_center(), self.mass2.get_center()
            )
        )
```

```
        self.add(self.line1, self.line2, self.origin)


# simulation happens here
class Simulation(MovingCameraScene):
    def construct(self):
        # print out some basic info about sim
        print("frame rate: " + str(config.frame_rate))
        print("runtime: " + str(runtime))
        print("simulation_time: " + str(simulation_time))
        print("dt: " + str(dt))

        # scale frame to zoom out
        self.camera.frame.scale(((l1 + l2) * 2 * 1.15) / (self.camera.frame.height))

        # time tracker needed for animation length
        time = ValueTracker(0)

        match mode:
            case "sequential":
                for i in range(num_pends):
                    color = hex(int((theta1_i + i * 0.00075) / (2 * np.pi) * 2**24))
                    pend = DoublePendulum(
                        theta1_i + i * 0.01, theta2_i + i * 0.01, str(color)
                    )
                    self.add(pend)
            case "random":
                for i in range(num_pends):
                    color = hex(random.randrange(0, 2**24))
                    theta1 = random.uniform(0, 2 * np.pi)
                    theta2 = random.uniform(0, 2 * np.pi)
                    pend = DoublePendulum(theta1, theta2, str(color))
                    self.add(pend)
            case "single":
                pend = DoublePendulum(theta1_i, theta2_i, "#c95792")
                self.add(pend)

        self.play(
            time.animate.set_value(simulation_time), rate_func=linear,
run_time=runtime
        )

        print("yayay")
        print()
```

Fragment Shader

```
#define PI 3.14159
#define TAU PI*2.
// ex: height of 5 -> [-2.5, 2.5]
#define SCREEN_HEIGHT TAU

#define CENTER_X 0.
#define CENTER_Y 0.
#define SCALE_INV 1.

#define DT 0.01
#define g 9.8067

#define L1 1.
#define L2 2.
#define M1 1.
#define M2 2.

// T = number of steps (per frame)
// NOTE: if modifying T_MAX, also modify color scaling below manually
#define T_MAX 2. * max(1000. * sqrt(L1 / g), 1000. * sqrt(L2 / g))

const vec3 black = vec3(0., 0., 0.);
const vec3 red   = vec3(0.2392, 0.2118, 0.3608);
const vec3 green = vec3(0.7882, 0.3412, 0.5725);
const vec3 blue  = vec3(0.7882, 0.3412, 0.5725);
const vec3 grey  = vec3(1., 0.6353, 0.);
const vec3 white = vec3(0.9804, 0.8588, 1.0);

// this function is pulled pretty much entirely from Noah's code,
// which pulls from https://physics.umd.edu/hep/drew/pendulum2.html
// state.x = theta1, state.y = dtheta1, state.z = theta2, state.w = dtheta2
vec4 diff(vec4 state) {
    float theta1  = state.x;
    float dtheta1 = state.y;
    float theta2  = state.z;
    float dtheta2 = state.w;

    float delta_theta = theta1 - theta2;
    float M = M1 + M2;
    float alpha = M1 + M2 * pow(sin(delta_theta), 2.);

    float eq6 = (- M2 * L2 * pow(dtheta2, 2.) * sin(delta_theta)
                 - M2 * L1 * pow(dtheta1, 2.) * sin(delta_theta) * cos(delta_theta)
                 - M * g * sin(theta1)
                 + M2 * g * sin(theta2) * cos(delta_theta));
    float ddtheta1 = eq6 / (L1 * alpha);

    float eq7 = (M * L1 * pow(dtheta1, 2.) * sin(delta_theta)
                 + M2 * L2 * pow(dtheta2, 2.) * sin(delta_theta) *
cos(delta_theta) // sign is confusing
                 + M * g * sin(theta1) * cos(delta_theta)
                 - M * g * sin(theta2));
    float ddtheta2 = eq7 / (L2 * alpha);
```

```
    return vec4(dtheta1, ddtheta1, dtheta2, ddtheta2);
}

// RK4 step
// Runge-Kutta methods are designed to solve the IVP: dy/dt = f(t, y), y(0) = y_0
// algorithm from https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
vec4 RK4_step(vec4 state) {
    float t = 0.;

    vec4 k1 = diff(state);
    vec4 k2 = diff(state + DT * (k1 / 2.));
    vec4 k3 = diff(state + DT * (k2 / 2.));
    vec4 k4 = diff(state + DT * k3);

    return state + (DT / 6.) * (k1 + 2. * k2 + 2. * k3 + k4);
}

// also stolen from Noah mostly
// base return value is vec2(T_MAX + 1000., T_MAX + 1000.)
// returns time it took the first pendulum to flip in t_f[0],
//         time it took the second pendulum to flip in t_f[1],
// note: will return whenever either of the pendulums flip, stay same if not
vec2 flip_time(vec4 state) {
    vec2 t_f = vec2(T_MAX + 1000., T_MAX + 1000.);

    vec4 state_i = state;

    float t = 0.;
    while (t < T_MAX) {
        state = RK4_step(state);

        if (abs(state_i.x - state.x) >= TAU) {
            t_f.x = t;
            break;
        } else if (abs(state_i.z - state.z) >= TAU) {
            t_f.y = t;
            break;
        } else {
            t++;
        }
    }

    return t_f;
}

// given a min, max, and value between them, returns a value
// between 0. and 1. representing the value's placement between them
float fake_lerp(float range_min, float range_max, float val) {
    return (val - range_min) / (range_max - range_min);
}


void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
    // scale to x = (-?, ?), y = (.5, .5) to avoid stretching, then scale by
SCREEN_HEIGHT
    // theta.x = theta1, theta.y = theta2
```

```
    vec2 theta = (SCREEN_HEIGHT) * (fragCoord/iResolution.y - .5 *
vec2(iResolution.x / iResolution.y, 1.));

    // color bar
    if ((theta.x <  (1.15 * SCREEN_HEIGHT / 2.))
     && (theta.x >  (1.05 * SCREEN_HEIGHT / 2.))
     && (theta.y <  (0.85 * SCREEN_HEIGHT / 2.))
     && (theta.y > -(0.85 * SCREEN_HEIGHT / 2.))) {
        if ((theta.x <  (1.13 * SCREEN_HEIGHT / 2.))
         && (theta.x >  (1.07 * SCREEN_HEIGHT / 2.))
         && (theta.y <  (0.83 * SCREEN_HEIGHT / 2.))
         && (theta.y > -(0.83 * SCREEN_HEIGHT / 2.))) {
            float scale = (theta.y + (0.85 * SCREEN_HEIGHT / 2.)) / (2. * (0.85 *
SCREEN_HEIGHT / 2.));
            if (scale <= (1. / 6.)) {
                float r = scale * 6.;
                fragColor = vec4(mix(black.x, red.x, r), mix(black.y, red.y, r),
mix(black.z, red.z, r), 1.);
                return;
            }
            if (scale <= (2. / 6.)) {
                float r = (scale - (1. / 6.)) * 6.;
                fragColor = vec4(mix(red.x, green.x, r), mix(red.y, green.y, r),
mix(red.z, green.z, r), 1.);
                return;
            }
            if (scale <= (3. / 6.)) {
                float r = (scale - (2. / 6.)) * 6.;
                fragColor = vec4(mix(green.x, blue.x, r), mix(green.y, blue.y, r),
mix(green.z, blue.z, r), 1.);
                return;
            }
            if (scale <= (4. / 6.)) {
                float r = (scale - (3. / 6.)) * 6.;
                fragColor = vec4(mix(blue.x, grey.x, r), mix(blue.y, grey.y, r),
mix(blue.z, grey.z, r), 1.);
                return;
            }
            if (scale <= (5. / 6.)) {
                float r = (scale - (4. / 6.)) * 6.;
                fragColor = vec4(mix(grey.x, white.x, r), mix(grey.y, white.y, r),
mix(grey.z, white.z, r), 1.);
                return;
            }
            if (scale <= 1.) {
                float r = (scale - (5. / 6.)) * 6.;
                fragColor = vec4(white, 1.);
                return;
            }
        }
        fragColor = vec4(1., 1., 1., 1.);
        return;
    }

    // theta.y will always be in range, but set theta.x out of range to black
    if (abs(theta.x) > (SCREEN_HEIGHT / 2.)) {
```

```
        fragColor = vec4(0., 0., 0., 0.);
        return;
    }

    // transform to focus on desired region, could make live updating later
    // state.x = theta1, state.y = dtheta1
    // state.z = theta2, state.w = dtheta2
    vec4 state = vec4(SCALE_INV * theta.x + CENTER_X, 0., SCALE_INV * theta.y +
CENTER_Y, 0.);

    // t_f.x = time for first pendulum to flip
    // t_f.y = time for second pendulum to flip
    vec2 t_f = flip_time(state);

    // color based off of time it took to flip, on a "log" scale, each pend different
    const float t1_ref1 = 100. * sqrt(L1 / g);
    const float t2_ref1 = 100. * sqrt(L2 / g);
    const float t1_ref2 = 400. * sqrt(L1 / g);
    const float t2_ref2 = 400. * sqrt(L2 / g);
    const float t1_ref3 = 750. * sqrt(L1 / g);
    const float t2_ref3 = 750. * sqrt(L2 / g);
    const float t1_ref4 = 1000. * sqrt(L1 / g);
    const float t2_ref4 = 1000. * sqrt(L2 / g);
    const float t1_ref5 = 2300. * sqrt(L1 / g);
    const float t2_ref5 = 2300. * sqrt(L2 / g);


    if (t_f.x <= t1_ref1 || t_f.y <= t2_ref1) {             // "black" flip before
first ref
        float r = fake_lerp(
            0.,
            max(t1_ref1, t2_ref1),
            min(t_f.x, t_f.y)
        );
        fragColor = vec4(mix(black.x, red.x, r), mix(black.y, red.y, r), mix(black.z,
red.z, r), 1.);
    } else if (t_f.x <= t1_ref2 || t_f.y <= t2_ref2) {      // "red" flip between ref
1 and 2
        float r = fake_lerp(
            max(t1_ref1, t2_ref1),
            max(t1_ref2, t2_ref2),
            min(t_f.x, t_f.y)
        );
        fragColor = vec4(mix(red.x, green.x, r), mix(red.y, green.y, r), mix(red.z,
green.z, r), 1.);
    } else if (t_f.x <= t1_ref3 || t_f.y <= t2_ref3) {      // "green" flip between
ref 2 and 3
        float r = fake_lerp(
            max(t1_ref2, t2_ref2),
            max(t1_ref3, t2_ref3),
            min(t_f.x, t_f.y)
        );
        fragColor = vec4(mix(green.x, blue.x, r), mix(green.y, blue.y, r),
mix(green.z, blue.z, r), 1.);
    } else if (t_f.x <= t1_ref4 || t_f.y <= t2_ref4) {      // "blue" flip between
ref 3 and 4
```

```
        float r = fake_lerp(
            max(t1_ref3, t2_ref3),
            max(t1_ref4, t2_ref4),
            min(t_f.x, t_f.y)
        );
        fragColor = vec4(mix(blue.x, grey.x, r), mix(blue.y, grey.y, r), mix(blue.z,
grey.z, r), 1.);
    } else if (t_f.x <= t1_ref5 || t_f.y <= t2_ref5) {      // "grey" flip between
ref 4 and 5
        float r = fake_lerp(
            max(t1_ref4, t2_ref4),
            max(t1_ref5, t2_ref5),
            min(t_f.x, t_f.y)
        );
        fragColor = vec4(mix(grey.x, white.x, r), mix(grey.y, white.y, r),
mix(grey.z, white.z, r), 1.);
    } else {
        fragColor = vec4(white, 1.);
    }
}
```