Ripple Simulation on Water Surface

Group 70
Chujie Chen, Shuran Wen, Siying Cen

https://github.gatech.edu/scen9/CSE6730

https://colab.research.google.com/drive/1bn290T7k9IYhFoCuNi4vuaeUAYDiAPSD?usp=sharing#scrollTo=XB9H38d-ChRO

Abstract

In this project, we aim to simulate the formation procedure of ripples when droplets drop on a water surface. The dynamics of the water surface is represented by the change of relative height and velocity of each point.

Introduction

Wave is propagation of vibration. The wave propagation direction which is perpendicular to the particle vibration direction is called a transverse wave, while the same is called a longitudinal wave, and a water wave is a superposition of a transverse wave and a longitudinal wave.

There have been many methods proposed for water wave simulation. One classic class of simulation methods is wave superposition, including *Sinusoids* wave superposition and *Gerstner*'s wave [1] superposition. Other classes include FFT methods [2], wave particle methods [3] and flow map methods.

The nature of waves are vibrations and energy transmission. As a result, the distribution of energy is a representation of the distribution of waves, which can be represented by wave amplitudes.

The shallow-water equations are a set of hyperbolic partial differential equations that describe the flow below a pressure surface in a fluid, bounded from below by the bottom topography and from above by a free surface [5]. Shallow-water equations can be used to model waves in rivers and oceans. Considering shear stress, viscous forces or frictions in the water system, using shallow-water equations to describe the velocity and height deviation of water surface is more accurate and realistics.

Description of the system

A water system of 1km x 1km x 1m (x, y, z) is studied in this project. Initially, two drops are created on this water system; at each step of simulation, the velocity and height deviation of each point on the water surface is simulated and recorded. After certain steps of simulation, the final water system state will be visualized in 2D and 3D animation.

Conceptual model of the system

- **Initial model:**

Due to the wave propagation characteristics, the vibration at a certain point of the next moment is affected by the combination of the vibration of the surrounding particles and its own vibration. To simplify this, we assume that the amplitude *Ai* of point *Xi* is not only affected by itself, but also affected by four points (*Xi−W, Xi+W, Xi−1, Xi+1*) around it. We also assume that the influence of the four points on the Xi point is equal and linearly superposition.

Thus, the formula of calculating amplitude Ai of point Xi after one step can be summarized as:

$$A_i^{'} = \alpha(A_{i-w} + A_{i+w} + A_{i-1} + A_{i+1}) + bA_i$$

After approximation, this formula can be described as [4]:

$$A_i^{'} = \frac{1}{2}(A_{i-w} + A_{i+w} + A_{i-1} + A_{i+1}) - A_i$$

Besides, we need to take attenuation into consider, or the water surface will keep waving when a wave source is added. Therefore, we also add a dampening factor in our model, so that after each iteration, the amplitude will be reduced by a certain percentage from the ideal value:

$$A_i^{'} = dampening\ factor\ *\ A_i^{'}$$

Repeating this one-step update, we can simulate the amplitudes of each point on the water surface after *n* steps.

When visualizing, because of the refraction of water, when the water surface is not perpendicular to our line of sight, the underwater scenery we see is not directly below the observation point, but has a certain offset. To handle this offset, we simply do a linear approximation, which in particular means by calculating the difference in amplitude between the up,down, left and right points of a certain point to represent the deviation of the underwater scenery shift.

We store the pixel information of the original image in two arrays, one is used to store the original image data, and the other is used to store the updated data in each iteration. We also use an inverter to handle the image restoration problem when updating the image by giving each point an offset in the opposite direction. At last, copy each pixel on the original image to the updated data according to the offset.

In order to form a wave, we add a wave source on a calm water surface like throwing a stone into a pool, and the size and energy of the wave source formed are related to the radius of the stone and the force you throw at it. In order to simulate the wave source, we modify the wave amplitude array around the center point at the beginning.The radius of the wave source is controlled by modifying the amplitude in the circle around the center point with a certain radius.

- **Advanced model:**

We implemented the following 2D of shallow-water equations to solve the two dimensional velocity of each point on the water surface [6]:

$$du/dt - fv = -g*d(eta)/dx + tau\_x/(rho\_0*H) - kappa*u$$
$$dv/dt + fu = -g*d(eta)/dy + tau\_y/(rho\_0*H) - kappa*v$$
$$d(eta)/dt + d((eta + H)*u)/dx + d((eta + H)*u)/dy = sigma - w$$

u: the velocity in the x direction

v: the velocity in the y direction
f: coriolis coefficient associated with the coriolis force
tau: wind stress
g: the acceleration due to gravity
kappa: friction
rho_0: Density of fluid
H: height of the water
eta: the height deviation of the surface from its mean height H

In our model, we can choose to set wind stress, friction, coriolis force to true or false.

At each step of shallow-water equation simulation, ‘eta’ value of each point will be collected and saved to visualization functions.
Repeat “take-one-step” for thousands of times to get the final simulation result and save the animation of simulation procedure.

Platform

We use Python to develop our model and visualize the result. In particular, we implement the data structure definition and detail logic in python files, and use the jupyter notebook to initialize the objects and call the functions.

Codes

We defined four main classes (‘liquid.py’, ‘drop.py’ and ‘visualize.py’) including functions in our codes implementation.

- **The “drop” class defines drops:**

```
class drop:
    def __init__(self, x, y, width=20, amplitude=1.0):
        self.x = x
        self.y = y
        self.width = width
        self.amplitude = amplitude
```

- **The ”liquid” class defines the initialization state of water surface and ‘update_one_step’, ‘update_n_step’ methods:**

```
def __update_one_step(self):

    u_np1 = np.zeros((self.N_x, self.N_y))  # To hold u at next time step
```

```
v_np1 = np.zeros((self.N_x, self.N_y))  # To hold v at enxt time step
eta_np1 = np.zeros((self.N_x, self.N_y))  # To hold eta at next time step

# ------------ Computing values for u and v at next time step ---------------
u_np1[:-1, :] = self.u_n[:-1, :] - self.g * self.dt / self.dx * (self.eta_n[1:, :] - self.eta_n[:-1, :])
v_np1[:, :-1] = self.v_n[:, :-1] - self.g * self.dt / self.dy * (self.eta_n[:, 1:] - self.eta_n[:, :-1])

# Add friction if enabled.
if (self.use_friction is True):
    u_np1[:-1, :] -= self.dt * self.kappa[:-1, :] * self.u_n[:-1, :]
    v_np1[:-1, :] -= self.dt * self.kappa[:-1, :] * self.v_n[:-1, :]

# Add wind stress if enabled.
if (self.use_wind is True):
    u_np1[:-1, :] += self.dt * self.tau_x[:] / (self.rho_0 * self.H)
    v_np1[:-1, :] += self.dt * self.tau_y[:] / (self.rho_0 * self.H)

# Use a corrector method to add coriolis if it's enabled.
if (self.use_coriolis is True):
    u_np1[:, :] = (u_np1[:, :] - self.beta_c * self.u_n[:, :] + self.alpha * self.v_n[:, :]) / (1 + self.beta_c)
    v_np1[:, :] = (v_np1[:, :] - self.beta_c * self.v_n[:, :] - self.alpha * self.u_n[:, :]) / (1 + self.beta_c)

v_np1[:, -1] = 0.0  # Northern boundary condition
u_np1[-1, :] = 0.0  # Eastern boundary condition
# -------------------------- Done with u and v -----------------------------

# Temporary variables (each time step) for upwind scheme in eta equation
h_e = np.zeros((self.N_x, self.N_y))
h_w = np.zeros((self.N_x, self.N_y))
h_n = np.zeros((self.N_x, self.N_y))
h_s = np.zeros((self.N_x, self.N_y))
uhwe = np.zeros((self.N_x, self.N_y))
vhns = np.zeros((self.N_x, self.N_y))

# --- Computing arrays needed for the upwind scheme in the eta equation.----
h_e[:-1, :] = np.where(u_np1[:-1, :] > 0, self.eta_n[:-1, :] + self.H, self.eta_n[1:, :] + self.H)
h_e[-1, :] = self.eta_n[-1, :] + self.H

h_w[0, :] = self.eta_n[0, :] + self.H
h_w[1:, :] = np.where(u_np1[:-1, :] > 0, self.eta_n[:-1, :] + self.H, self.eta_n[1:, :] + self.H)

h_n[:, :-1] = np.where(v_np1[:, :-1] > 0, self.eta_n[:, :-1] + self.H, self.eta_n[:, 1:] + self.H)
h_n[:, -1] = self.eta_n[:, -1] + self.H
```

```
        h_s[:, 0] = self.eta_n[:, 0] + self.H
        h_s[:, 1:] = np.where(v_np1[:, :-1] > 0, self.eta_n[:, :-1] + self.H, self.eta_n[:, 1:] + self.H)

        uhwe[0, :] = u_np1[0, :] * h_e[0, :]
        uhwe[1:, :] = u_np1[1:, :] * h_e[1:, :] - u_np1[:-1, :] * h_w[1:, :]

        vhns[:, 0] = v_np1[:, 0] * h_n[:, 0]
        vhns[:, 1:] = v_np1[:, 1:] * h_n[:, 1:] - v_np1[:, :-1] * h_s[:, 1:]
        # ------------------------- Upwind computations done -------------------------

        # ----------------- Computing eta values at next time step -------------------
        eta_np1[:, :] = self.eta_n[:, :] - self.dt * (uhwe[:, :] / self.dx + vhns[:, :] / self.dy)  # Without
source/sink

        # Add source term if enabled.
        if (self.use_source is True):
            eta_np1[:, :] += self.dt * self.sigma

        # Add sink term if enabled.
        if (self.use_sink is True):
            eta_np1[:, :] -= self.dt * self.w
        # ------------------------------ Done with eta --------------------------------

        eta_np1 = eta_np1 * (1/np.max(eta_np1))

        self.u_n = np.copy(u_np1)  # Update u for next iteration
        self.v_n = np.copy(v_np1)  # Update v for next iteration
        self.eta_n = np.copy(eta_np1)  # Update eta for next iteration

        self.u_list.append(self.u_n)
        self.v_list.append(self.v_n)
        self.eta_list.append(self.eta_n)

        return


    def update_n_step(self, n=1):
        for i in range(n):
            self.__update_one_step()
        return
```

- **Codes for visualization:**

Three animation functions (2D, 3D, velocity field) are implemented inside

```python
def eta_animation(X, Y, eta_list, frame_interval, filename):
    fig, ax = plt.subplots(1, 1)
    plt.xlabel("x [m]", fontname = "serif", fontsize = 12)
    plt.ylabel("y [m]", fontname = "serif", fontsize = 12)
    pmesh = plt.pcolormesh(X, Y, eta_list[0], vmin = -0.7*np.abs(eta_list[int(len(eta_list)/2)]).max(),
        vmax = np.abs(eta_list[int(len(eta_list)/2)]).max(), cmap = plt.cm.RdBu_r, shading='auto')
    plt.colorbar(pmesh, orientation = "vertical")

    def update_eta(num):
        ax.set_title("Surface elevation $\eta$ after t = {:.2f} hours".format(
            num*frame_interval/3600), fontname = "serif", fontsize = 16)
        pmesh.set_array(eta_list[num][:-1, :-1].flatten())
        return pmesh,

    anim = animation.FuncAnimation(fig, update_eta,
        frames = len(eta_list), interval = 10, blit = False)
    mpeg_writer = animation.FFMpegWriter(fps = 24, bitrate = 10000,
        codec = "libx264", extra_args = ["-pix_fmt", "yuv420p"])
    anim.save("{}.mp4".format(filename), writer = mpeg_writer)
    return anim

def eta_animation3D(X, Y, eta_list, frame_interval, filename):
    fig = plt.figure(figsize = (12, 12), facecolor = "white")
    ax = fig.add_subplot(111, projection='3d')

    surf = ax.plot_surface(X, Y, eta_list[0], cmap = plt.cm.RdBu_r)

    def update_surf(num):
        ax.clear()
        surf = ax.plot_surface(X/1000, Y/1000, eta_list[num]*10, cmap = plt.cm.RdBu_r)
        ax.set_title("Surface elevation $\eta(x,y,t)$ after $t={}$ seconds".format(
            num*frame_interval), fontname = "serif", fontsize = 19, y=1.04)
        ax.set_xlabel("x [m]", fontname = "serif", fontsize = 14)
        ax.set_ylabel("y [m]", fontname = "serif", fontsize = 14)
        ax.set_zlabel("$\eta$ [cm]", fontname = "serif", fontsize = 16)
        ax.set_xlim(X.min()/1000, X.max()/1000)
        ax.set_ylim(Y.min()/1000, Y.max()/1000)
        ax.set_zlim(-3, 7)
        plt.tight_layout()
        return surf,

    anim = animation.FuncAnimation(fig, update_surf,
                                   frames = len(eta_list), interval = 10, blit = False)
    mpeg_writer = animation.FFMpegWriter(fps = 24, bitrate = 10000,
                                         codec = "libx264", extra_args = ["-pix_fmt", "yuv420p"])
    anim.save("{}.mp4".format(filename), writer = mpeg_writer)
    return anim
```

```
def velocity_animation(X, Y, u_list, v_list, frame_interval, filename):
    fig, ax = plt.subplots(figsize = (12, 12), facecolor = "white")
    plt.title("Velocity field $\mathbf{u}(x,y)$ after 0.0 days", fontname = "serif", fontsize = 19)
    plt.xlabel("x [km]", fontname = "serif", fontsize = 16)
    plt.ylabel("y [km]", fontname = "serif", fontsize = 16)
    q_int = 3
    Q = ax.quiver(X[::q_int, ::q_int]/1000.0, Y[::q_int, ::q_int]/1000.0, u_list[0][::q_int,::q_int], v_list[0][::q_int,::q_int],
        scale=0.2, scale_units='inches')

    def update_quiver(num):
        u = u_list[num]
        v = v_list[num]
        ax.set_title("Velocity field $\mathbf{{u}}(x,y,t)$ after t = {} seconds".format(
            num*frame_interval/3600), fontname = "serif", fontsize = 19)
        Q.set_UVC(u[::q_int, ::q_int], v[::q_int, ::q_int])
        return Q,

    anim = animation.FuncAnimation(fig, update_quiver,
        frames = len(u_list), interval = 10, blit = False)
    mpeg_writer = animation.FFMpegWriter(fps = 24, bitrate = 10000,
        codec = "libx264", extra_args = ["-pix_fmt", "yuv420p"])
    fig.tight_layout()
    anim.save("{}.mp4".format(filename), writer = mpeg_writer)
    return anim
```

- **Codes for running simulation tests (in colab):**

```
from main.water.drop import drop
from main.surface.liquid import liquid
```

```
canvas = liquid(N_x=150, N_y=150)
```

```
dp1 = drop(20, 20, width=10, amplitude=10.0)
dp2 = drop(80, 80, width=2, amplitude=3.0)
```

```
canvas.take_drops([dp1, dp2])
```

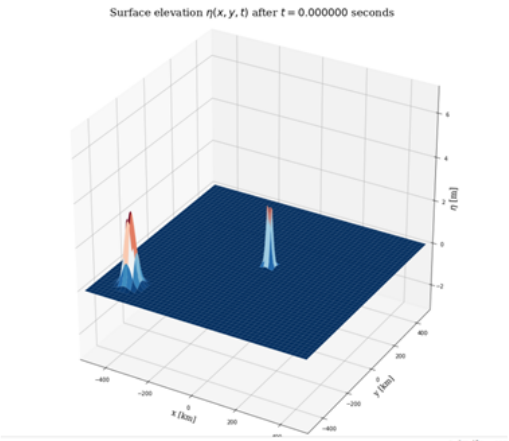
```
canvas.update_n_step(1000)
```

- **Time interval definition:**

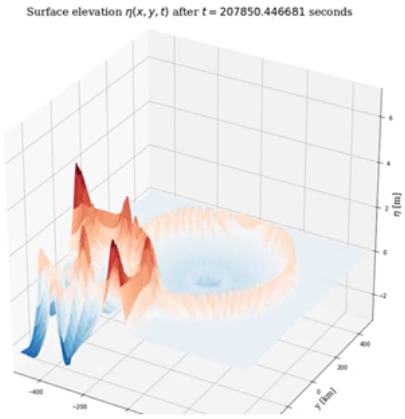
```
ax.set_title("Surface elevation $\eta(x,y,t)$ after $t={:2f}$ seconds".format(
    num*frame_interval), fontname = "serif", fontsize = 19, y=1.04)
```

Result

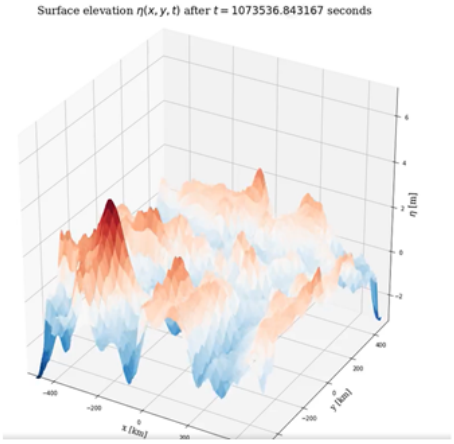
- **3D simulation animation:**



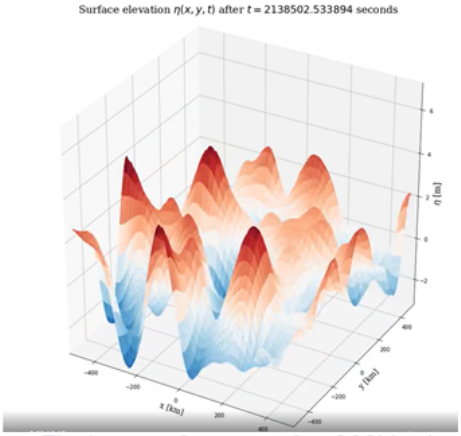
Initial state of water system at t = 0



Simulation state of system after 60h



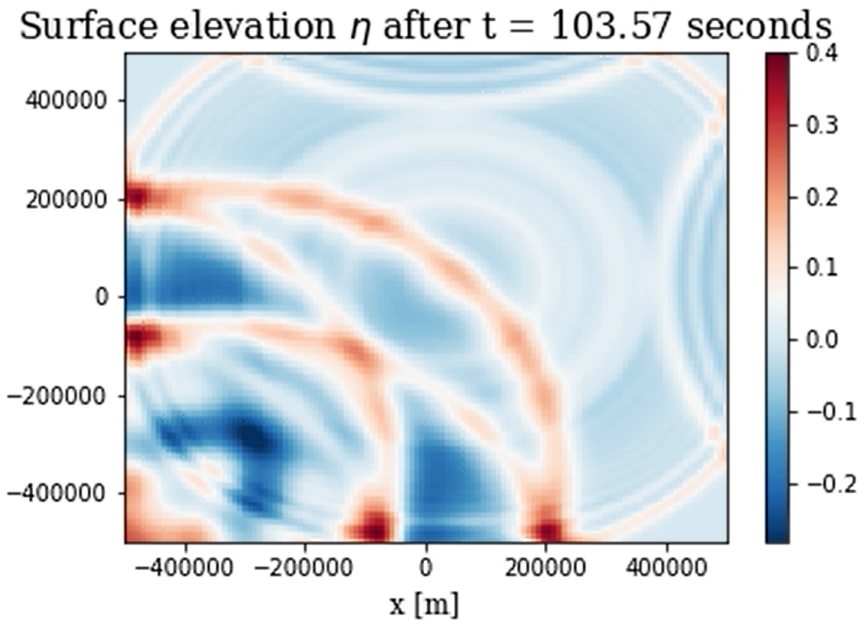
Simulation state of system after 300h



Final state of system after 600h's simulation

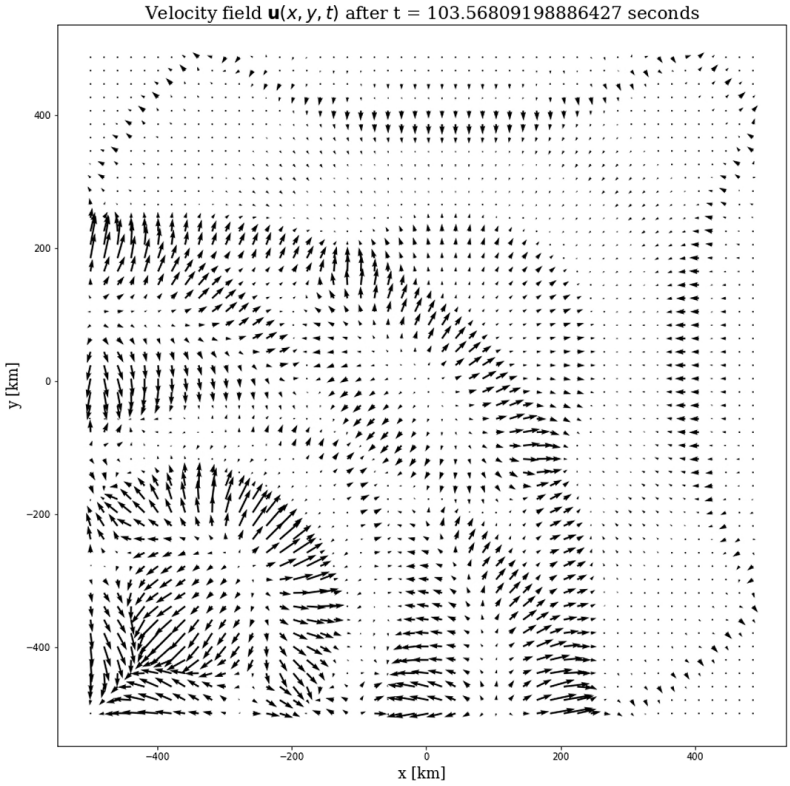
The whole 3D simulation process can be viewed in:
surface_simulation_3d.mp4

- **2D simulation animation:**



The whole 2D simulation process can be viewed in:
surface_simulation_2d.mp4

- **Velocity field simulation:**



The whole velocity simulation process can be viewed in:
velocity_simulation.mp4

From our simulation results, the initial two drops created two ripples; when ripples diffuse, the vibrations transmit to the whole surface of the water system.

Division of labor

- Chujie Chen: part of code implementation; create colab
- Shuran Wen: explore other methods; part of code implementation; simulation demo
- Siying Cen: explore other methods; write report

Reference

[1] Fournier, Alain, and William T. Reeves. "A simple model of ocean waves." Proceedings of the 13th annual conference on Computer graphics and interactive techniques. 1986.
[2] Mastin, Gary A., Peter A. Watterberg, and John F. Mareda. "Fourier synthesis of ocean scenes." IEEE Computer graphics and Applications 7.3 (1987): 16-23.
[3] 2010, Yuksel C. Real-time water waves with wave particles[M]. Texas A&M University
[4] https://www.cnblogs.com/youyoui/p/8546178.html
[5] https://empslocal.ex.ac.uk/people/staff/gv219/codes/shallowwater.pdf
[6] https://github.com/jostbr/shallow-water