Shallow Water Equations on a Trefoil Knot Surface in Blender

Pirmin Neyer, Nina Fieldhouse, Anais Schneider

August 31, 2024

Abstract

The following report investigates the simulation of Shallow Water Equations (SWE) on a trefoil knot surface. The SWE, a simplified form of the Navier-Stokes equations for shallow fluids, are widely used in atmospheric dynamics, hydraulic engineering, and oceanography. Due to the complex geometry of the knot, unique challenges were introduced while traditional SWE simulations are typically conducted on simpler surfaces, such as spheres and tori. The simulations were conducted using Blender, where a high-resolution trefoil knot mesh was created. To accustom for the geometry, custom gravity vectors and boundary conditions were introduced. A staggered grid approach was employed to improve accuracy and stability while accounting for non-uniform grid spacing. A central finite difference scheme was implemented for spatial derivatives, coupled with forward Euler integration in time. To further ensure numerical stability, the Courant-Friedrichs-Lewy (CFL) criterion the time step was dynamically adjusted. Key results include the effect of grid resolution on computational load and the visualization of SWE variables through animated UV maps. The methods used improved simulation stability, enhanced computational efficiency, and provided insightful visualizations of fluid behavior on complex geometries.

1 Introduction

1.1 The shallow water equations

The shallow water equations (SWE) are a system of equations for fluid dynamics in shallow fluids. They are a simplified version of the Navier-Stokes equations since they are applied in scale situations where the horizontal scale is much larger than the vertical scale. For this simplification they are a very important set of equations in a variety of science and engineering fields, because they are computationally much simpler than the Navier-Stokes equations but still provide an accurate depiction of the flow field in a shallow water situation. Examples for their applications are hydraulic engineering, modelling tides and coastal currents, sediment transport and most importantly for us they are crucial for simulating atmospheric dynamics. We can derive them first by thinking of shallow water flow on a rotating plane. We first consider a shallow layer of an incompressible and homogeneous fluid. The governing equations are then the 3 momentum equations in each dimension, as well as the fact of non-divergence in an incompressible flow:

$$\frac{Du}{Dt} - fv = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\nabla^2 u \tag{1}$$

$$\frac{Dv}{Dt} + fu = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\nabla^2 v \tag{2}$$

$$\frac{Dw}{Dt} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g + \nu\nabla^2 w \tag{3}$$

$$div(\mathbf{u}) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{4}$$

With u, v and w being the velocities in the x, y and z dimensions, f being the coriolis parameter, ν being the viscosity and D/Dt being the material derivative. Now since the horizontal scale is to be considered much larger than the vertical scale, scale analysis can be performed on this set of equations. This transformation yields the conclusions that the system is assumed to be inviscid, the vertical momentum equation can be approximated by the hydrostratic relationship and that the pressure force in the horizontal is independent on z, meaning that if the horizontal velocities are initially constant, they will remain constant over the fluid depth. Finally after this scale transformation the shallow water equations are made of the following set of equations:

$$\frac{Du}{Dt} - fv = -g\frac{\partial}{\partial x}(\eta + h) \tag{5}$$

$$\frac{Du}{Dt} - fv = -g\frac{\partial}{\partial x}(\eta + h) \tag{6}$$

$$\frac{Dh}{Dt} = -h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \tag{7}$$

With h being height from the surface to the top of the shallow layer and η being the height of an obstacle.

1.2 Shallow water dynamics on a trefoil knot

A trefoil knot is the simplest nontrivial knot in mathematics. Spanning a two-dimensional surface along it creates an object in three-dimensional space, fulfilling the criteria of closedness, similar in that respect to a torus. Using this more complex geometrical body as a grid for a physical simulation such that of SWE results in interesting differences compared to the same simulation on a sphere. Like the torus, it could be viewed in simplified terms as a two-dimensional surface in 3D space with boundary conditions on every side of the grid. A non-uniform spacing of the grid-points results in dx and dy which are not constant, which in turn affects the simulation.

For simplification of the problem we have decided to run the simulation on a non-rotating trefoil knot, meaning that we don't account for Coriolis or other rotating forces. However, the a special kind of gravity vector was constructed through an addition of forces to two different centres of mass, and is different for each grid point.

2 Methods

2.1 The creation of a trefoil knot in Blender

The creation of the mesh on which the simulation would run was done in a couple of steps. First, the knot curve itself is created, using the parametric equations for the trefoil knot:

$$x(t) = \mathrm{a} \cdot (\sin(t) + 2 \cdot \sin(2t))$$
$$y(t) = \mathrm{a} \cdot (\cos(t) - 2 \cdot \cos(2t))$$
$$z(t) = \mathrm{a} \cdot (-\sin(3t))$$

where t is the parameter that ranges from 0 to 4π and a represents the overall scale of the object. Here, an adjustable resolution was used to make varying intervals for t. Further, a curve circle with a second custom resolution was then used as a second step as a profile for the knot curve. It was aligned orthogonally to the local curve tangent such that the resulting mesh is a tubular surface. As a second step, the following variables were stored as attributes on each grid point, to enable later procedures:

Attribute	**Description**
x_index	i-coordinate on a 2D grid
y_index	j-coordinate on a 2D grid
h	Water height variable in SWE
u	Horizontal velocity in i-direction in SWE
v	Horizontal velocity in j-direction in SWE
position	Cartesian coordinate of grid point
g_vector_x, *g_vector_y*, *g_vector_z*	Gravity force components in x-, y-, and z-direction

Table 1: Grid point variables

2.1.1 Initial conditions

While *x_index* and *y_index* could be obtained by performing mathematical operations on the vertex indices, the initial conditions in terms of h, u and v also had to be defined. This was to represent initial perturbations in the SWE variables, and was thus done by evaluating a noise function with a certain scale at every grid point. Values for the three variables were determined like this and normalized to be in a range of $[-1, 1]$. The choice of using noise rather than completely randomized values was to have some spatial correlation between the vertices, mimicking a more natural setting of a body of water.

2.1.2 Gravity

The gravity vectors in the code are calculated based on two main factors:

1. **Attraction to the closest point on the knot curve** (denoted as `A to B`): This simulates a kind of "gravitational pull" that attracts vertices towards the knot's curve.

2. **Attraction to the world origin** (denoted as `A to C`): This simulates another force that pulls vertices towards the origin of the 3D space $(0, 0, 0)$.

The gravity vector, denoted as `F`, is a blend of these two directional vectors, each normalized and then scaled by respective strengths (`gravity_ratio` and `gravity_strength`).

The vectors are computed as follows:

- `D`: A unit vector pointing from the vertex position $\mathbf{A}$ to the closest point on the curve `B`.
- `E`: A unit vector pointing from the vertex position `A` to the world origin `C`, scaled by `gravity_ratio`.
- `F`: The final gravity vector is the normalized sum of `D` and `E`, which is then scaled by `gravity_strength`.

This combination ensures that each vertex is influenced both by the shape of the trefoil knot and the central origin of the scene.

2.2 Boundary Conditions

The code handles boundary conditions effectively by employing a combination of techniques to ensure the geometry remains smooth and continuous across the entire mesh.
Firstly, the trefoil knot curve is generated using a parametric equation. Since the curve is periodic, it repeats smoothly, forming a closed loop. This property inherently resolves boundary conditions at the start and end points of the curve. The curve's periodicity ensures that the final point naturally connects back to the initial point, thereby avoiding discontinuities at the boundaries of the curve. Consequently, no special treatment for the start or end of the curve is necessary.
In the process of calculating the tangents for the vertices along the curve, the function `get_tangent()` handles boundary conditions explicitly. For any point along the curve that is not the last point, the tangent vector is computed by taking the difference between the current point and the next point on the curve. Mathematically, this is represented as:

$$\text{tangent} = \mathbf{P}_{i+1} - \mathbf{P}_i,$$

where $\mathbf{P}_i$ is the position of the current point and $\mathbf{P}_{i+1}$ is the position of the subsequent point. For the last point on the curve, since there is no subsequent point, the tangent is computed using the difference between the last point and the preceding point, as follows:

$$\text{tangent} = \mathbf{P}_i - \mathbf{P}_{i-1}.$$

This ensures that even at the boundary (i.e., the last point), the tangent vector remains properly defined and smoothly aligned with the rest of the curve, preventing abrupt changes in orientation that could distort the geometry.
The mesh generation and profile wrapping are handled in the function `create_tubular_mesh_with_attributes()`. This function constructs the tubular mesh by lofting a circular profile along the knot curve. Each circular profile is connected to the next by creating faces between corresponding vertices. To manage the boundaries of the circular profile, modular arithmetic is employed. Specifically, the vertices are connected in a loop by using the operation:

$$\text{bm.faces.new}(\mathbf{v}_j, \mathbf{v}_{(j+1) \mod N}, \mathbf{v}'_{(j+1) \mod N}, \mathbf{v}'_j),$$

where N is the number of vertices in the profile. This operation ensures that the vertices of the circular profile wrap around smoothly, thereby preventing any discontinuities at the boundary of the profile and maintaining a consistent connection between vertices. The geometry thus remains continuous, without visible artifacts at the edges of the mesh.
Additionally, the code adds custom attributes such as noise fields (h, u, v) and gravity vectors to the vertices. These attributes are computed for each vertex individually, based on its position

in space. The noise fields are scaled and normalized across the mesh, while the gravity vectors are calculated by considering both the vertex's position relative to the curve and its position relative to the world origin. Since the attributes depend solely on the vertex positions and not on the overall structure of the mesh, they are applied uniformly across the entire geometry. Consequently, no special handling of boundary conditions is required for these attributes. The noise and gravity values are interpolated smoothly across the entire mesh, contributing to the overall consistency of the geometry.

2.3 Staggered grid

Instead of a collocated grid we have implemented a staggered grid, which other than storing the variables at the same location stores the variables at different positions relative to the grid points. Specifically the variable height (h) is stored at the center of each grid cell while horizontal velocity component (u) is stored between grid points horizontally and the vertical velocity component (v) is stored between grid points vertically. This way the simulation updates the velocity components based on height and the other way around. This is implemented in the update equations of h, u and v: `h_new[1:-1, 1:-1]`, `u_new[1:-1, 1:-1]` and `v_new[1:-1, 1:-1]`. The update equations assume that the grid is implicitly staggered by using neighboring h values when updating u and v and vice versa. This coupling of the variables makes the simulation physically more realistic, by also better conserving mass and momentum which helps increase the stability of the simulation. The staggered grid also helps reduce numerical oscillations and improves the calculation of gradients and fluxes, making them more accurate.

2.4 Discretization of the shallow water equations

The `update_swe()` function uses a finite difference method scheme to approximate the solution to the shallow water equations on a discrete grid. Specifically, it employs a central difference scheme in space and a forward difference scheme in time.
The spatial derivatives for both the height (h) and velocity components (u and v) are approximated using central differences. For example, the height update due to the advection term is computed using:

$$\frac{\partial h}{\partial x} \approx \frac{h[i+1,j] - h[i-1,j]}{2 \cdot \Delta x}, \quad \frac{\partial h}{\partial y} \approx \frac{h[i,j+1] - h[i,j-1]}{2 \cdot \Delta y}.$$

These terms are used to calculate the changes in height (h) due to horizontal advection. Similarly, for the velocity update related to the pressure gradient term:

$$\frac{\partial h}{\partial x} \approx \frac{h[i+1,j] - h[i-1,j]}{2 \cdot \Delta x}, \quad \frac{\partial h}{\partial y} \approx \frac{h[i,j+1] - h[i,j-1]}{2 \cdot \Delta y}.$$

These terms represent the pressure gradient forces affecting the velocity components u and v. The central difference scheme is second-order accurate, meaning that the error decreases quadratically as the grid resolution increases.
In terms of time derivatives, the scheme uses a forward difference approach, also known as explicit Euler integration. The height is updated using:

$$h_{\text{new}}[i,j] = h[i,j] - \Delta t \cdot \text{advection terms},$$

and the velocity components are updated as follows:

$$u_{\text{new}}[i,j] = u[i,j] - \Delta t \cdot \text{pressure gradient term}, \quad v_{\text{new}}[i,j] = v[i,j] - \Delta t \cdot \text{pressure gradient term}.$$

The forward Euler scheme is first-order accurate in time, meaning that the error decreases linearly as the time step size decreases.

2.4.1 Non-uniform grid point spacing

In the simulation code, non-uniform grid spacing is implemented by dynamically calculating the distances between grid points based on their positional attributes. This approach ensures that the finite difference method adapts to the spatial variations in the mesh geometry, particularly for complex structures like the trefoil knot. The non-uniform grid spacing is computed using the function `calculate_distances_from_attribute()`, which calculates the distances between adjacent grid points in both the x- and y-directions.
The position of each grid point is stored in the `position_attr` attribute, from which the distances are derived. For each grid point (i, j), the distance to its neighboring point in the x-direction, denoted as `dx_grid[i, j]`, is calculated as the Euclidean distance between their respective positions. Similarly, the distance in the y-direction, `dy_grid[i, j]`, is determined in the same manner. These distance values are stored in the arrays `dx_grid` and `dy_grid`, allowing the grid spacing to vary across the simulation domain. At the boundaries of the grid, where there are no adjacent points, the spacing is set equal to the spacing of the previous point, or a very small default value (e.g., 1×10^{-6}) is used to prevent division by zero.
The calculated non-uniform grid spacings are utilized in the discretization of the shallow water equations. These spacings affect the finite difference calculations for the spatial derivatives of height and velocity. In particular, the function `update_swe_staggered()` uses the `dx_grid` and `dy_grid` values when calculating differences between adjacent grid points in the velocity and height fields. This ensures that the finite difference updates properly account for the non-uniformity in grid spacing.
The use of non-uniform grid spacing improves the accuracy of the simulation by adapting the spatial resolution to better represent the underlying geometry. In regions where the grid is denser, the simulation captures finer details, while in coarser regions, computational resources are conserved without sacrificing significant accuracy. Additionally, the stability of the simulation is maintained by adjusting the time step size based on the smallest local grid spacing, as dictated by the Courant-Friedrichs-Lewy (CFL) condition. This is achieved by computing a CFL factor that depends on the grid spacing and the local gravitational strength. The time step is then adjusted dynamically to ensure that the CFL condition is satisfied, preventing instability during the simulation.

2.5 Optimizing runtime

In regards to the computational cost of the simulation, different approaches as to how to handle the data in Blender internally were explored. For instance, storing h, u and v attributes on all vertices for each simulation step by keyframing them at each simulation step was tested. Though this would have been a useful approach, in which individual frames could have been inspected very freely, it proved to be way to costly. Thus, the alternative approach of generating UV maps at each simulation step was taken and turned out to be much more fruitful, as real-time animations of even higher-resolution trefoil surfaces could be visualised. For example, for a low-resolution (3.2k vertices) mesh, the average simulation step time is 0.06 seconds on our device, but of course increases with the number of vertices.

3 Results

3.1 Stability

The stability of the simulation is mainly ensured by the CFL-criterion that is implemented in the function `update_swe_staggered()`. This condition makes sure that the time step Δt is small enough to avoid numerical instabilities. During the simulation this is executed with the time step being adjusted dynamically based on local gravitation and grid spacing. This condition effectively prevented instabilities such as unphysical oscillations. The CFL condition

is frequently violated at $\Delta t > 0.01$, leading to an automatic adjustment to smaller time steps, which in turn also slows down the simulation, as more computation time is needed. With $\Delta t < 0.005$ it can be observed that the simulation remains stable.

3.2 Resolution and Simulation Speed

The speed of a single simulation step is measured by timing the loop execution within the `simulation_uv_maps_animation` function. As expected the computation time increases linearly with the number of grid points and increasing grid resolution. Decreasing values of `grid_size_i` and `grid_size_j` result in a quadratic increase of computational load. Doubling the resolution results in around 4 times the computation time per simulation step. This increase in runtime is a small cost compared to the increase of accuracy of the shallow water equations acquired instead.

It needs to be noted that the resolution parameters of the `generate_knot_mesh` function are `half_x_resolution` and `half_y_resolution`. This was to ensure that the x-resolution and y-resolution of the grid are definitely even numbers to prohibit any possible complications in the simulation.

3.3 Visualization of Results

The materials that are automatically assigned by the "Simulate_SWE.py" script function are as follows. For h_map, u_map and v_map (see Figure 1), the UV map that is updated each simulation step is used as input, is mapped from the $[-1, 1]$ range onto $[0, 1]$. It is then declared that values ≤ 0.4 are represented as blue, while values ≥ 0.6 are visualised as red and values in between are depicted as white. This is to show the variable anomalies (such as the water height anomaly) in an simple way.

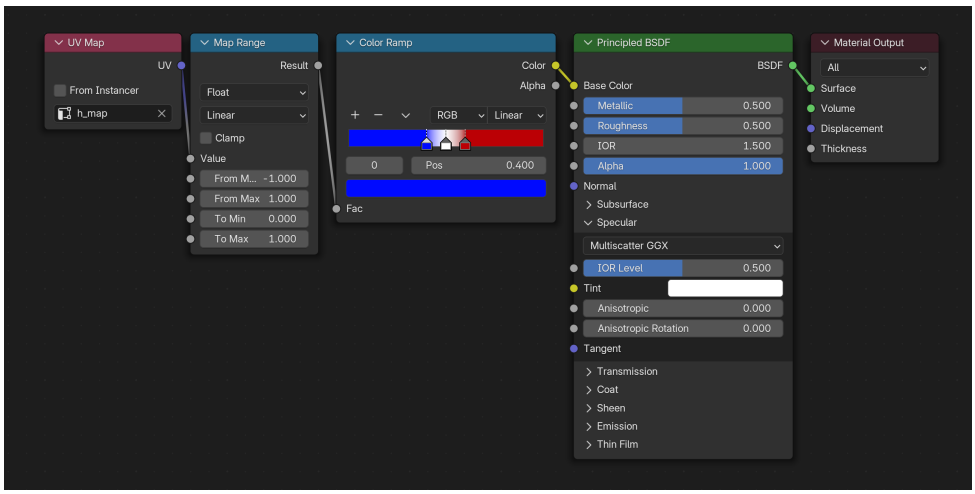


Figure 1: Shader setup for the h_map material (same approach with u_map and v_map)

For huv_map this is slightly different (see figure 2), as h, u and v values are combined to a 3D vector after being mapped. This has the effect that h, u and v correspond to the R, G and B values respectively. Further the colour is then simply adjusted in terms of saturation and brightness for better visibility. The huv_map material, while being less insightful on the evolution of individual variables, was created additionally as to visualise the effect of all variables in a concise way.
How these materials look applied to the trefoil knot surface can then be seen in figure 3. Here, the state after 20 simulation steps with $dt = 0.005$ is shown. Interestingly, u-, and v-values tend to deviate more strongly from 0 than h does, as there, lower horizontal gradients can be seen.

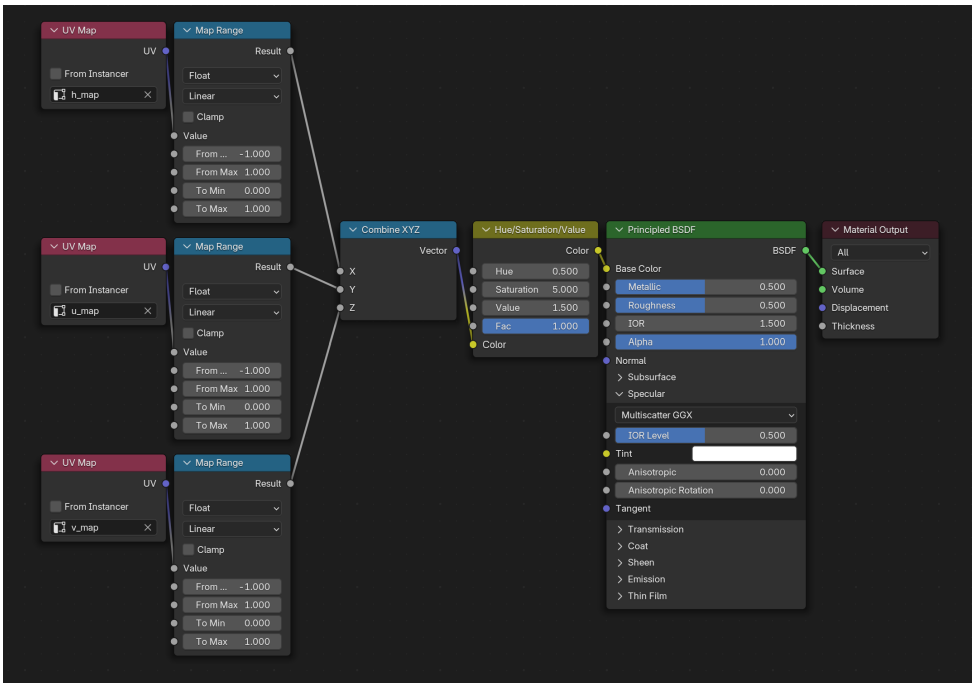
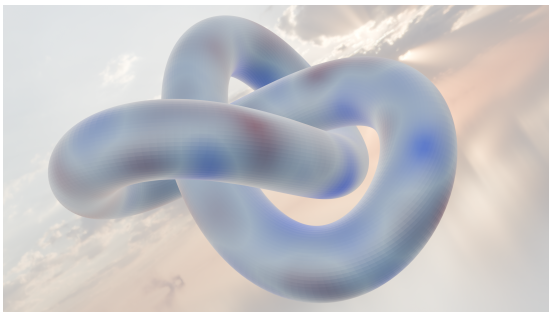
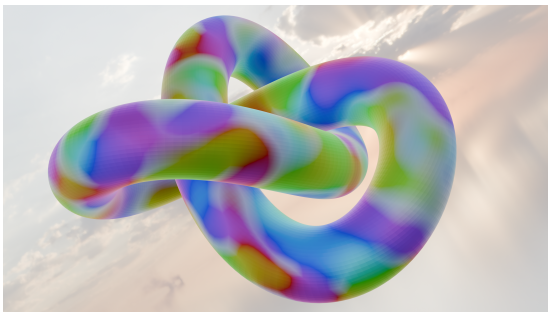


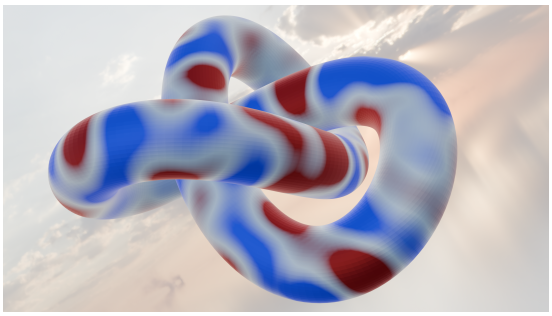
Figure 2: Shader setup for the huv_map material



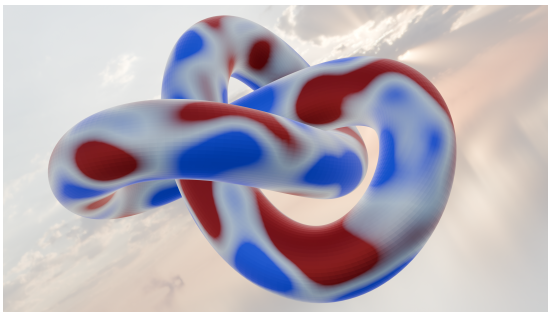
(a) h-values



(b) huv-representation



(c) u-values



(d) v-values

Figure 3: SWE variables after 20 simulation steps with $dt = 0.005$, on a trefoil surface with 38.4k vertices (resolution parameters: 300 x 32).

Further, a quick overview for the evolution of the h, u and v fields can be seen also here, in this gif or video (make sure to set the quality to 1080p). For this, 80 simulation steps with $dt = 0.005$ were applied and relatively few vertices used (3.2k, as the resolution are 50 x 16). The arrangement of the objects in terms of the applied materials is the same as in figure 3.

4 Conclusion

The Navier-Stokes equations were simplified fit a shallow water problem, also known as the shallow water equations, on a trefoil knot. The knot was created by first creating the knot curve itself using the parametric equations for a trefoil knot with adjustable resolution. The initial conditions had to be defined for the variables h, u and v using a noise function to represent a body of water more naturally. One of the main challenges and most important adjustments to this geometric body compared to a sphere was the representation of gravity. The gravity for the trefoil knot was based on two main factors, the attraction to the closest point on the knot curve as well as to the world origin. This was implemented using a gravity vector, blending these two directional vectors. The boundary conditions were handled based on the smooth calculations of the periodic curve and the tangent vectors to avoid discontinuities. As for the grid, a staggered grid was chosen instead of a collocated grid to ensure more stability and to make the simulation physically more realistic. The shallow water equations were discretized using a central difference scheme in space and a forward difference scheme in time. Further non-uniform grid spacing was used to increase the accuracy of the simulation by adapting spatial resolution to better represent the geometry. Stability was additionaly maintained by dynamically adjusting the time step size based on local grid spacing ensuring that the CFL criterion is not violated. The successful simulation showed that higher grid resolutions lead to a significant increase in computation time per simulation step. However this trade-off is justified by the improved accuracy which was visualized through various shader setups.

Performing this computation on a complex geometric surface such as the trefoil knot successfully highlights the diversity of the problems that the shallow water equations can simulate and that a couple simple adjustments can significantly increase the simulation accuracy and performance time. Also it is clear that this mesh creation set-up can be easily generalized or used with any other complex mathematical curve that is cyclical, i.e. closed (e.g. like a cinquefoil or septafoil knot). Using the trefoil knot for this purpose was an arbitrary assumption and just to show this.