Computer Physics Communications 248 (2020) 106966

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Celeris Base: An interactive and immersive Boussinesq-type nearshore wave simulation software☆

Sasan Tavakkol *, Patrick Lynett

Department of Civil and Environmental Engineering, University of Southern California, Los Angeles, CA, USA

ARTICLE INFO

Article history:
Received 20 July 2019
Received in revised form 13 September 2019
Accepted 25 September 2019
Available online 28 September 2019

Keywords:
Celeris
Boussinesq
Wave modeling
Immersive
Interactive
GPU

ABSTRACT

We introduce our interactive and immersive coastal wave simulation software, Celeris Base, which is the successor to Celeris Advent. Celeris Base is an open source software developed in the Unity3D game engine and in C# language. It supports an interactive environment and allows users to view the simulations in a virtual reality headset. Celeris Base solves the same equations as Celeris Advent, the extended Boussinesq equations, using our hybrid finite volume–finite difference method. These equations are solved on the GPU using compute shaders, written in HLSL. Celeris Base has several new features such as 360° video capturing, geographic map overlays, built-in real-time gauge plotters, etc. It also improves the implementation of the sponge layer boundary condition by introducing new damping equations. Celeris Base is designed and implemented using the best software engineering practices in the hope that it will be a base for further developments of the Celeris software series by researchers around the globe. We validate Celeris Base against experimental results in this paper.

Program summary

Program Title: Celeris Base
Program Files doi: http://dx.doi.org/10.17632/jdx7tddcxz.1
Licensing provisions: MIT License
Programming language: C#, HLSL
Nature of problem: Celeris Advent enabled researchers and engineers for the first time to simulate nearshore waves with a Boussinesq-type model, faster than real-time and in an interactive environment. However, its development platform and implementation complexity hindered researchers from developing it further and made adding new features to the software a daunting task. The software used graphics shaders to solve scientific equations which could be confusing for many. The visualization environment was wired from scratch which made it very difficult to add features such as virtual reality.
Solution method: A new software is developed completely from scratch following Celeris Advent, called Celeris Base. This software uses the same hybrid finite volume–finite difference scheme to solve the extended Boussinesq equations, but using a variant of shaders called compute shaders, removing possible barriers for other researchers to understand the code and develop it further. The software is developed in Unity3D, a popular game engine with a large and helpful community as well as thousands of ready to use plugins. Celeris Base is equipped with virtual reality and is the first nearshore simulation software to provide this feature.

1. Introduction

Recent disastrous hurricanes such as Hurricane Sandy (2012), Hurricane Harvey (2017), and Hurricane Michael (2018) and catastrophic tsunamis such as the Tohoku Earthquake and Tsunami in Japan (2011), Sulawesi Earthquake and Tsunami (2018), and Sunda Strait Tsunami (2018) have raised the global awareness for the urgent need to understand the response of developed coastal regions to tsunamis and wind waves. While aspects of these natural disasters are unpreventable, the following catastrophes can be avoided by careful engineering and thorough simulations [1]. Numerical simulations are, arguably, the best tools that engineers can utilize to design safer structures or to predict the risk of coastal disasters before they hit the coast. However, the supercomputing facilities required to

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Correspondence to: Google Research, New York, NY, USA.
E-mail address: tavakkol@usc.edu (S. Tavakkol).

https://doi.org/10.1016/j.cpc.2019.106966
0010-4655/

run high-order and accurate numerical simulations are expensive and mostly exclusive to large government organizations and universities. Furthermore, utilizing these models requires researcher-level skills often not available in small engineering firms. We defined the Celeris research project to overcome these issues with the following mission statement:

> *Democratizing high-order and high-performance coastal wave modeling by leveraging the GPU in a user-friendly software and modernizing it by leveraging the state-of-the-art visualization technologies [2]*

We developed Celeris Advent, the first faster than real-time and interactive Boussinesq-type wave model, and introduced it in an article published in 2017 [3]. Celeris Advent is a tremendously fast Boussinesq-type wave solver running on the GPU, with a compelling 3D visualizer and an interactive environment. It runs on off-the-shelf Windows laptops, eliminating the need for expensive supercomputing facilities. Celeris Advent has been downloaded over 2000 times from users spanning 50+ countries and its user's manual is translated to several languages by independent users. This software is now widely used in research, education, and engineering projects. As an example use case, we developed a website that provides a five-day forecast of wave conditions at several US coasts by periodically running Celeris Advent. This website is available at http://coastal.usc.edu/waves/.

Celeris Advent is developed in C++ and HLSL using the Direct3D framework. While such a native development environment has some advantages such as, potentially, a higher performance, it is not without drawbacks. The main issue that we encountered was the difficulty of adding new features to the software. Furthermore, Celeris Advent was developed with a focus on the core scientific functionality and less attention to recommended software architectural patterns such as modularity and other object-oriented programming best practices. Finally, we observed that other researchers were not successful in developing the software further or even recompiling it after a year passed from the initial release of the open source code, perhaps due to its complexity. These issues made us to revamp the software.

Celeris has a lot in common with video games, for example, GPU accelerated physics, attractive visualization, and interactivity. Therefore, we redeveloped Celeris Base from scratch in a game engine called Unity3D. This redevelopment has a lot of advantages. There are hundreds of ready to use libraries (called "assets") available to download or purchase on Unity3D Asset Store which can help adding new features to the software, such as virtual reality (VR) capabilities. We did not use the physics engine of Unity3D and developed our own engine for the wave simulation based on the same equations used in Celeris Advent.

Using a popular development platform such as Unity3D provides developers with community support which was not available for Celeris Advent. Furthermore, we paid more attention to software design in the revamped software and employed state-of-the-art engineering practices to develop a modular codebase. We are optimistic that these features and the introduced ease of development will make scientists and researchers embrace the software as a base for further developments; hence, we call the new series of our simulation software, Celeris Base.

The key difference between Celeris Base and Celeris Advent is in their implementations. Celeris Base is developed in Unity3D using C# while Celeris Advent is developed in C++. The computations are done, in both software, using shaders, but we used a different kind of shaders in Celeris Base, called *compute shaders*. As the naming suggests, these shaders are designed to handle computations, unlike the regular shaders used in Celeris Advent, which are designed to render graphics. Compute shaders are also written in HLSL, however, there is no need to build a dummy graphics pipeline to run them. They can be simply run on the GPU using a "Dispatch" call. This makes them suitable for general purpose programming and make it easier for scientists and researchers to understand the code. For visualizations, we use regular shaders in Celeris Base just like Celeris Advent, though Unity3D shaders are written in a variant of HLSL. Furthermore, we paid more attention to the software architecture of Celeris Base and used recommended techniques in object-oriented programming such as polymorphism, design patterns, dependency injection, etc. This careful engineering makes the software a true base for further developments. Finally, Celeris Base has several new features compared to Celeris Advent. For example, the user can put a gauge on a point to show the water surface diagrams in real time or they can record a 360° video of the experiment (e.g., https://youtu.be/tJeGviPzwEs). We also discuss how Celeris Base can be configured to support virtual reality.

Although Celeris Base is a significant improvement upon Celeris Advent from a developers-perspective, it is not meant to replace it. Celeris Advent has a few features that are not yet implemented in Celeris Base. As mentioned before, our goal of developing Celeris Base was making it easier for other researchers to develop the code further. For practical use cases, we recommend researchers to continue using Celeris Advent until new versions of Celeris are developed on top of Celeris Base. Celeris Base is also released as an open source software under the permissive software license, MIT license. The third-party libraries used in Celeris Base have their own corresponding licenses.

2. Mathematical model

The Boussinesq-type equations are a powerful tool for the study of nearshore dynamics, including both nonlinear and dispersive effects. A significant effort in the nearshore wave model community towards developing Boussinesq models has occurred in the past decades [4,5]. Assuming that both nonlinearity and frequency dispersion are weak and are in the same order of magnitude, Peregrine [6] derived the "standard" Boussinesq equations for variable depth in terms of the depth-averaged velocity and the free surface displacement. Numerical results based on the standard Boussinesq equations or the equivalent formulations have been shown to give predictions that compared quite well with field data and laboratory data. Many modified forms of Boussinesq-type equations have been introduced (*e.g.,* [7–9]) to extend the validity zone of these equations.

FUNWAVE [9] and COULWAVE [10] are examples of successful and widely used numerical implementations of the highly-nonlinear Boussinesq-type equations. These models have been applied to a wide variety of topics, including rip and longshore currents [11], wave runup [12], wave–current interaction [13], and wave generation by underwater landslides [14], among many others. While these high-order models provide a better representation of physical processes in theory, they can suffer from two important drawbacks. First, as the second-order corrections become more complex, the computational requirements, and thus time, to solve the system substantially increases. Often for practical cases, simulations using these models run at a fraction of real time on hundreds of cores. Secondly, while it can be clearly shown that the high-order models do agree better with analytical solutions and controlled laboratory experiments, the impact of these corrections becomes blurred for application in real field cases. The inherent errors and uncertainties when hindcasting or forecasting a field site can overwhelm the high-order corrections. In these cases, there is little, if any, practical justification for using a high order and computationally expensive model. Bearing these facts in mind, we adopted the extended Boussinesq

equations [15] which are only weakly nonlinear, yet, accurate enough for practical cases. The interested readers can learn more about different kinds of Boussinesq-type models and inter-model analyses in references such as [4,16]. Discretized by a hybrid finite volume–finite difference (FVM–FDM) scheme, we found the extended Boussinesq equations very robust to use in Celeris as a fast and interactive solver. These equations for 2DH flow read as:

$$\begin{bmatrix} h \\ P \\ Q \end{bmatrix}_t + \begin{bmatrix} P \\ P^2/h + gh^2/2 \\ PQ/h \end{bmatrix}_x + \begin{bmatrix} Q \\ PQ/h \\ Q^2/h + gh^2/2 \end{bmatrix}_y + \begin{bmatrix} 0 \\ ghz_x + \psi_1 + f_1 \\ ghz_y + \psi_2 + f_2 \end{bmatrix} = 0 \tag{1}$$

where h is the total water depth, P and Q are the depth-integrated mass fluxes in x and y directions respectively (x-y plane makes the horizontal solution field). Subscripts x and y denote spatial differentiation, and subscript t denotes temporal differentiation. z is the bottom elevation measured from a fixed datum. f_1 and f_2 are the bottom friction terms and g is the gravitational acceleration coefficient. ψ_1 and ψ_2 are the modified dispersive terms defined as:

$$\begin{aligned} \psi_1 = &-\left(B+\frac{1}{3}\right)d^2\left(P_{xxt}+Q_{xyt}\right) - Bgd^3\left(\eta_{xxx}+\eta_{xyy}\right) \\ &- dd_x\left(\frac{1}{3}P_{xt}+\frac{1}{6}Q_{yt}+2Bgd\eta_{xx}+Bgd\eta_{yy}\right) \\ &- dd_y\left(\frac{1}{6}Q_{xt}+Bgd\eta_{xy}\right) \end{aligned} \tag{2}$$

$$\begin{aligned} \psi_2 = &-\left(B+\frac{1}{3}\right)d^2\left(P_{xyt}+Q_{yyt}\right) - Bgd^3\left(\eta_{yyy}+\eta_{xxy}\right) \\ &- dd_y\left(\frac{1}{3}Q_{yt}+\frac{1}{6}P_{xt}+2Bgd\eta_{yy}+Bgd\eta_{xx}\right) \\ &- dd_x\left(\frac{1}{6}P_{yt}+Bgd\eta_{xy}\right) \end{aligned} \tag{3}$$

where d is the still water depth and η is the water surface elevation measured from the still water surface elevation. B is the calibration coefficient for dispersion properties of the equations. We use $B = 1/15$ as suggested in the original article [7] introducing the extended Boussinesq equations and widely adopted thereafter.

3. Numerical model

We discretize the extended Boussinesq equations using a hybrid FVM–FDM explained in [3] and referred to as TL17. This scheme is developed following [17] by rearranging the terms such that we can rewrite Eq. (1) as ODE's in time. Then, following [18,19] we use a hybrid FVM–FDM discretization to solve these equations on a uniform Cartesian grid. The spatial domain is discretized by fixed-size rectangular cells of $\Delta x \times \Delta y$. Each cell is a control volume for the FVM discretization, while the cell centers and their corresponding cell averages are used as the grid points in FDM. The advective terms are discretized using a second-order well-balanced positivity preserving central-upwind scheme introduced in [20], known as KP07, which is a FVM to solve the Saint–Venant system of shallow water equations. The rest of the terms are discretized using second order central FDM. Time integration is performed by the third-order fixed timestep Adams–Bashforth scheme.

Celeris Base, similar to Celeris Advent, does not have a direct treatment for wave breaking. However, our numerical experiments show that the numerical dissipation incorporated in the FVM model using a minmod limiter, successfully imitates the wave breaking. Kirby et al. [21] also discuss that in models with shock-capturing schemes the implementation of an explicit formulation for breaking wave dissipation might be unnecessary. MOST model is another example in which numerical dissipation mimics wave breaking [22,23].

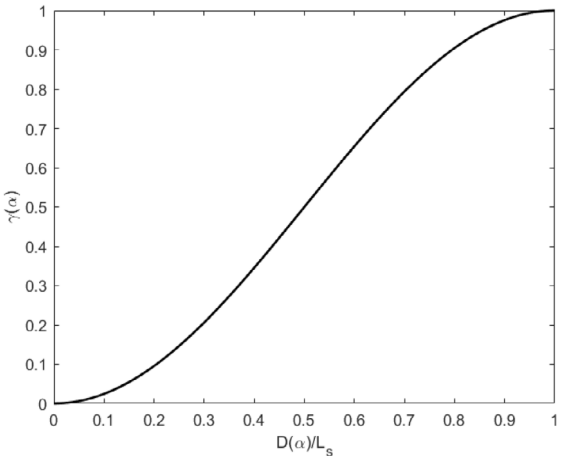

Fig. 1. Diagram of $\gamma(\alpha)$, Eq. (4), inside the sponge layer.

3.1. Boundary conditions

We add two layers of ghost cells at each side of the solutions field to implement the boundary conditions. This number agrees with the size of the numerical stencil. Four types of boundary conditions are implemented in Celeris Base: fully reflective solid wall, sinewave maker, sponge layer, and irregular wavemaker. These boundary conditions can be applied to any of the four boundaries of the field, either from the GUI or from the input file. The implementations of solid walls and sinewave maker are explained in [3] and the irregular wavemaker is discussed in [24]. They all follow their implementations in Celeris Advent. However, we improved the sponge layer implementation in Celeris Base which we discuss next.

3.1.1. Sponge layer

The sponge layer boundary condition in Celeris Base is an adjusted version of the method introduced by Tonelli and Petti [19] and used in Celeris Advent. The implementation proposed in [19] was done in Celeris Advent by multiplying the values of η, P, and Q by a damping coefficient, $\gamma(\alpha)$, defined as

$$\gamma(\alpha) = \frac{1}{2}\left(1+\cos\left(\pi\frac{L_s - D(\alpha)}{L_s}\right)\right) \tag{4}$$

where α is substituted with x or y for boundaries perpendicular to the x-axis or y-axis, L_s is the width of the sponge layer, and $D(\alpha)$ is the normal distance to the absorbing boundary. The damping is only applied to the cells which are located inside the sponge layer. Fig. 1 shows the value of $\gamma(\alpha)$ inside the sponge layer.

Tonelli and Petti [19] do not clarify whether this coefficient is applied every time-step, or in certain time intervals. We implemented this method in Celeris Advent v1.0.0, applying the coefficient on the flow parameters every time-step and observed that this implementation has an undesired steep damping effect causing spurious reflections. Applying this coefficient in every timestep has a compound effect in damping the waves. Consider the crest of a solitary wave with waveheight of H_0 on the edge of a west-side sponge layer and ready to enter the damping area. In

the first time-step that the crest of the wave enters the sponge layer, it gets damped by $\gamma(L_s - \delta)$, where δ is the distance of the crest from the edge of the sponge layer. This is equal to the distance the wave has traveled in one timestep, Δt. In the next time step, the already damped crest gets further damped by an even smaller coefficient of $\gamma(L_s - 2\delta)$. This compound effect makes the crest of the wave to diminish very quickly. The damping rate in this implementation is a function of the timestep size, and it is higher for smaller timesteps, which is not expected from a well-behaved sponge layer.

The correct approach is to damp the waveheight (and other flow parameters) such that the locus of the wave crest follows a shape similar to the one shown in Fig. 1. To achieve this goal the flow parameters must be multiplied by a coefficient, like $\lambda(\alpha)$, at each time-step, such that the compound effect of this new coefficient resembles the shape of $\gamma(\alpha)$. In other words:

$$\gamma\left(L_s - n\delta\right) = \prod_{i=0}^{n} \lambda\left(L_s - i\delta\right) \tag{5}$$

We can write:

$$\gamma\left(L_s - n\delta\right) = \lambda\left(L_s - n\delta\right) \times \gamma\left(L_s - (n-1)\,\delta\right) \tag{6}$$

or

$$\lambda\left(L_s - n\delta\right) = \frac{\gamma\left(L_s - n\delta\right)}{\gamma\left(L_s - (n-1)\,\delta\right)} \tag{7}$$

Substituting $L_s - n\delta$ with a newly defined variable such as α, we have:

$$\lambda\left(\alpha\right) = \frac{\gamma\left(\alpha\right)}{\gamma\left(\alpha + \delta\right)} \tag{8}$$

As mentioned earlier, δ is the distance the wave crest travels in Δt. Therefore, assuming the wave celerity, c, we have:

$$\lambda\left(\alpha\right) = \frac{\gamma\left(\alpha\right)}{\gamma\left(\alpha + c\Delta t\right)} \tag{9}$$

Eq. (9) shows that the damping coefficient for each wave depends on its celerity and therefore an ideal sponge layer must treat each wave frequency differently, which cannot be easily achieved. As a reasonable approximation, we use $c = \sqrt{gd}$ for c in Eq. (9) and damp the flow parameters by $\lambda(\alpha)$ at each time-step.

4. Software documentation

4.1. Source files

A Unity3D project generally consists of several folders and hundreds of files, however, the *Assets* folder is the one which contains most of the important files. Under the Assets folder of Celeris Base, the *Celeris* folder contains most of the code done to build the software. Fig. 2 shows the directory paths and files in each subdirectory of this folder. The *Scripts* folder hosts most of the codes written to create the solver and rest of the software. Most of the files implement only one class which handles one specific action or feature in the software. The main flow of the software is controlled by `GameManager.cs`. The Boussinesq solver is driven by `TL17Driver.cs`. There is also a driver for an NLSW solver, called `KP07Driver.cs`. Both driver classes are derived from a base class implemented in `KP07Base.cs`. The base class contains the codes to solve the common advective terms between the two solvers. This base class itself is a derived class of `Solver.cs`, which connects the solver classes to the rendering infrastructure and compute shaders. The Solver class also contains shared features such as logging. It also implements virtual functions such as TimeStep() which are overridden by the child classes.

Two main compute shaders, `TL17.compute` and `KP07.compute`, contain the kernels (GPU codes) for the Boussinesq and NLSW solvers, respectively. These two files import (include) a compute shader, called `base_KP07.compute`, which like the driver classes, contains the common kernels between two algorithms. Furthermore, each main compute shader includes other compute shaders such as `time_integration.compute`, `tools.compute`, etc.

The folder *GPGPU* contains the rendering shaders and non-solver compute shaders. The rendering shaders have a `.shader` extension and may import files ending in `.cginc`. Shaders are applied to objects through files called "Material", which are in the *Materials* folder. The entry point in Unity3D projects is called "scenes". The only scene in Celeris Base is `Main.unity` which should be opened from the Unity3D editor to start and run the software in development mode.

4.2. Implementation

Unity3D is a popular game engine which provides the developer with a rendering engine, scripting support, etc. along with a powerful visual editing tool. Most of the contents in a Unity3D project, such as an avatar, the camera, a popping sound, the logic, etc. are instances of the *GameObject* class. The behavior of a GameObject instance is defined and controlled by its *Components*. For example, a piece of code must be attached to the camera GameObject, as a Component, to control its movement.

The entry point to a Unity project is a *scene*. A scene is the collection of all the GameObjects, Components, and their connections. Unity scenes are written in YAML (rhymes with camel), which is a data serialization language. The number of lines in the aggregated YAML file of Celeris Base is over 140,000. Luckily, one rarely, if at all, needs to edit the YAML files of a scene. Instead, Unity3D has a visual editor where all these objects and their relationship are shown and can be edited. Unity3D 2018 for windows comes with the Visual Studio 2018 as the IDE. Unity3D supports C# as its main programming language, which is used in Celeris Base.

4.2.1. Structure

With the brief explanation of the Unity3D game engine, we can now discuss the structure of the Celeris Base scene. There are four major GameObject's in the scene: Main Camera, GameManager, Engine, and Canvas. The Main Camera renders what the user will see in the field. A few components are attached to this object, where the most important one is a script to control the movement of the camera by the user.

GameManager drives the software flow. This object also has several components, but the most important one is a script called `GameManager.cs`. The GameManager class uses the singleton design pattern, which means that only one instance of the class can exist. This is useful for GameManger to act as the sole coordinator of actions across the system. Furthermore, the one possible instance of the class is easily accessible from other classes just by calling the static public property of the class called "Instance". GameManger defines the entire framework of the software, such as the CML loader, solver, rendering infrastructure, etc. It also contains the "Update" function, which is called every frame to run the solver for a pre-defined number of steps.

The Engine GameObject is the rendering engine of Celeris Base. It has several children (sub-objects) to take care of rendering the water surface and the terrain. We will explain this object in more detail in the coming sections. Canvas is a GameObject that all UI elements must be a child of. Canvas always has the EventSystem object which takes care of the messaging system. Several scripts are attached to the UI elements to apply the required logic.

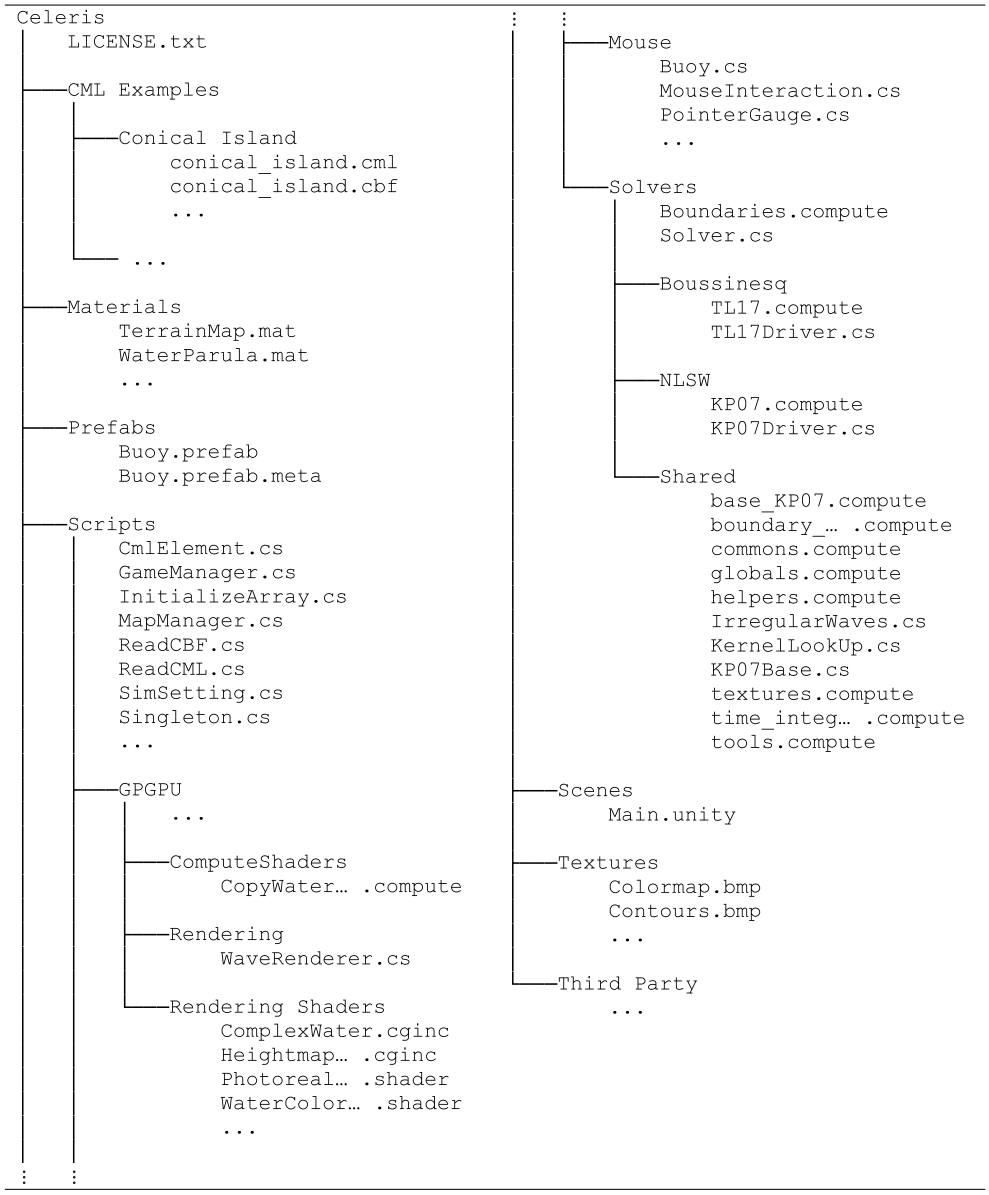

Fig. 2. Some of the important source files in Celeris Base.

For example, a script called "BoundaryPanelScript" is attached to the boundary tab UI element, which connects the GUI to the simulation engine. It takes the values of the parameters for the boundary from the GUI and updates the simulation parameters accordingly.

4.2.2. Solver

Celeris Base has two fluid dynamics models, one for solving the NLSW equations and one for the extended Boussinesq equations. The NLSW solver is based on a finite-volume model developed by Kurganov and Petrova (2007) [21], and therefore called KP07. Our Boussinesq solver, as introduced in [3], is call TL17.

Celeris Base has a core *Solver* class which is implemented in `Solver.cs` and defines the foundation for running a model on the GPU. For example, *RenderTexture*'s are defined in this class. It also has methods to hook the simulation parameters (coming from a CML file) to the model as well as data logging functions. However, it does not implement the functions to solve any equations. It is in fact, a parent class with a virtual *TimeStep()* method which should be overridden by a child class implementing this function and solving the motion equations. As explained before, TL17 uses KP07 to solve the advective terms of the extended Boussinesq equations and therefore the NLSW and Boussinesq solvers share several functions. We refactored the common functions of the two models into a class called *KP07Base* which inherits from Solver. Two classes called *TL17Driver* and *KP07Driver* are derived from *KP07Base* to handle their corresponding models.

Fig. 3 shows the UML diagram of Solver, KP07Base, KP07Driver, and TL17Driver classes. Some major or representative variables and methods are shown in this diagram, as well as the inheritance hierarchy. The Solver class has a reference to a ComputeShader, called Compute, which is set to either a KP07 or TL17 compute shader by the subclasses. It also has an instance of a ComputeShader called renderComputeShader, which connects the model results to the visualization tools. The *simTextureManager*, hosts all the RenderTexture's in the software and takes care of creating and destroying them as necessary to prevent memory leaks.

The KP07Base class defines the shared functions between KP07 and TL17 models to solve the advective terms. For example, *pass1*

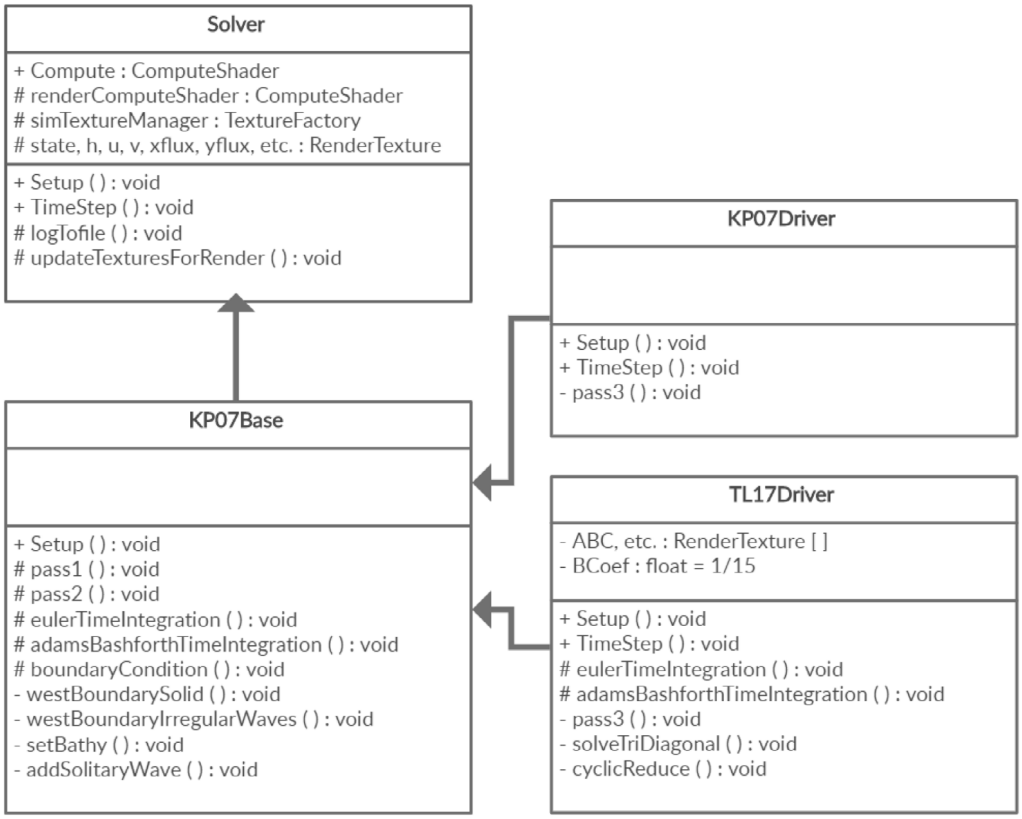

Fig. 3. Class UML diagram of Celeris Base solvers.

and *pass2* functions take care of reconstructing physical values and computing fluxes at cell interfaces. It also implements the time integration algorithms as virtual functions. This class defines, implements, and applies all the boundary conditions in the software. Furthermore, it handles bathymetry as well as applying the initial conditions such as solitary waves.

KP07Driver is a relatively thin class derived from KP07Base. It defines the final step in solving the NLSW equations as a function called *pass3*, which sums up the fluxes and source terms to calculate the time derivatives of the flow parameters. It also implements the *TimeStep* function by calling *pass1*, *pass2*, *pass3* and then a time integration method. *TimeStep* also calls the *boundaryCondtion* and the base *TimeStep* functions from KP07Base.

TL17Driver is the class which drives the TL17 model and solves the extended Boussinesq equations. It implements a function called *pass3* which, similar to KP07Driver, aggregates the results from *pass2* with source and dispersive terms. The class also defines crucial functions such as those to solve the tridiagonal matrices using the cyclic reduction algorithm. TL17Driver overrides the time integration functions to add the necessary changes, yet calls the base time integration functions within. Finally, it implements the *TimeStep* function by calling *pass1*, *pass2*, *pass3*, time integration, and tridiagonal solver functions. Note that all the computational functions in the Solver class and its children, are drivers to run compute shaders on the GPU, and do not implement any CPU solutions.

4.2.3. Compute shaders

Celeris Base has a key difference with Celeris Advent in leveraging the power of GPU. We use compute shaders in Celeris Base, in contrast to pixel shaders that we used in Celeris Advent. Compute shaders are very similar to pixel shaders in language and syntax, however they are designed for the primary purpose of computation and therefore fit better for our purpose. Unlike using pixel shaders for general purpose programming, which requires setting up a dummy graphics pipeline, driving the compute shaders is simply done with a function called *Dispatch*. Compute shaders of Unity3D closely match Microsoft's DirectCompute technology. They also have some similarities with the CUDA programming language which perhaps is more popular in the coastal engineering research community. We hope that using compute shaders instead of the pixel shaders in Celeris Base will remove the barriers for coastal researchers to further develop the mathematical and numerical model of the software.

All the files with the extension of `.compute` are compute shaders. Most of them are located at Assets/Celeris/Scripts/Solver/Shared folder. These compute shaders implement the shared functions between TL17 and KP07 models, which, themselves, are implemented in `TL17.compute` and `KP07.compute` files.

4.2.4. Rendering mesh generation

One of the challenges in developing Celeris Base was rendering the wave and terrain surfaces. Unity3D allows mesh generation from heightmaps, however, it only takes square shapes, i.e., a heightmap with equal number of grid points in x and y directions. The generated mesh can be scaled independently in x and y directions to form a rectangle, however, for shapes with larger difference in number of grid points, the precision of the mesh in one direction need to be either sacrificed or overloaded. To overcome this limitation, we adapted a visualization system from a commercial library called Surface Waves by Code Animo. Although this library was designed to render only square domains, it had many useful tools which helped us design and implement our own surface rendering tools.

In Celeris Base, we tile a rectangular grid with smaller square grids of size 128×128, and then slightly scale up the squares in one direction to fit the domain. Fig. 4 shows this infrastructure

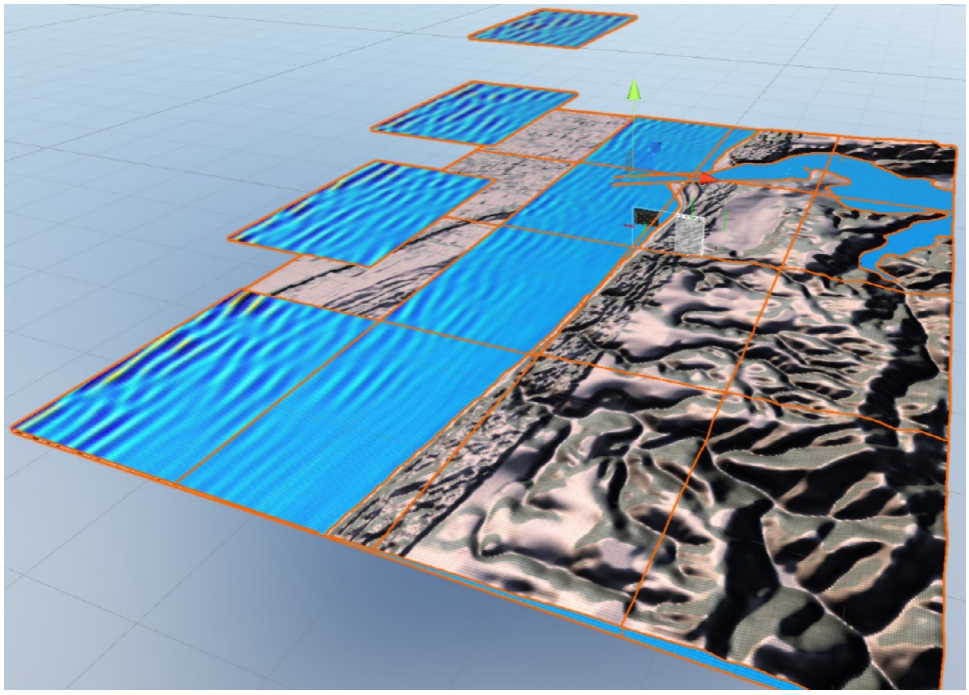

Fig. 4. Rendering infrastructure in Celeris Base with disassembled tiles.

Fig. 5. Textures of grid (a), jet colormapping (b), and terrain colormapping (c), used in Celeris Base renderings.

for a rectangular domain with disassembled tiles for a better illustration. The number of squares in each direction is calculated by dividing the number of simulation cells in that direction (i.e., n_x or n_y) by the size of square tiles (i.e., 128), and then rounding the result to the nearest integer number. To avoid low visualization resolution, we use at least two tiles in each direction. This system introduces a new challenge: the number of the grid points in the underlying computational model differs from the number in the rendering infrastructure, and therefore there is not a 1:1 match between the mathematical mesh and visualization mesh in the software. We solve this problem by defining a linear UV mapping between two meshes and sampling the compute textures for visualization purposes.

4.2.5. Shading and materials

Rendering in Unity3D is done by using materials, shaders and textures. Materials are components attached to a GameObject with a mesh component that define its rendering properties. A material has references to the textures it uses, tiling information, color tints, etc. It also uses a shader which takes care of the math of calculating the color of each pixel. A shader normally takes one or more textures as its input and uses them for calculating the surface color. Celeris Base uses several materials, shaders, and textures to deliver its vivid visualization. We briefly explain photorealistic and colormap shading in this section and leave rendering the geographic map of the domain to the next section.

Shaders in Celeris Base are used for more than just coloring the pixels. We use vertex shaders to displace the mesh points every frame to create the water surface. The vertex shader is where the sampling which we mentioned in the previous section happens. The vertex shader runs for each vertex of the visualization mesh and samples the water surface texture (populated by compute shaders) to displace the vertex according to the water surface elevation.

Photorealistic shading in Celeris Base is done by the `photorealistic.shader` file, which is attached to the `SimpleWater.mat` material. This shader adds the effects from light and skybox reflection to the albedo color of a pixel. It also has a reference to a grid texture (Fig. 5a). This grid texture is sampled, scaled, color reversed, and then added to the pixel's color to render the grid visual effect in Celeris Base. Fig. 6 shows a scene rendered with the photorealistic shader with two different skyboxes and two grid modes.

Colormapping is also done using surface shaders. Two colormapping shaders exist in Celeris Base: `WaterColormap.shader` for the water surface and `TerrainColormap.shader` for the terrain. In colormap shaders, a colormapped texture is given to the shader as an input. For example, Fig. 5b shows the jet

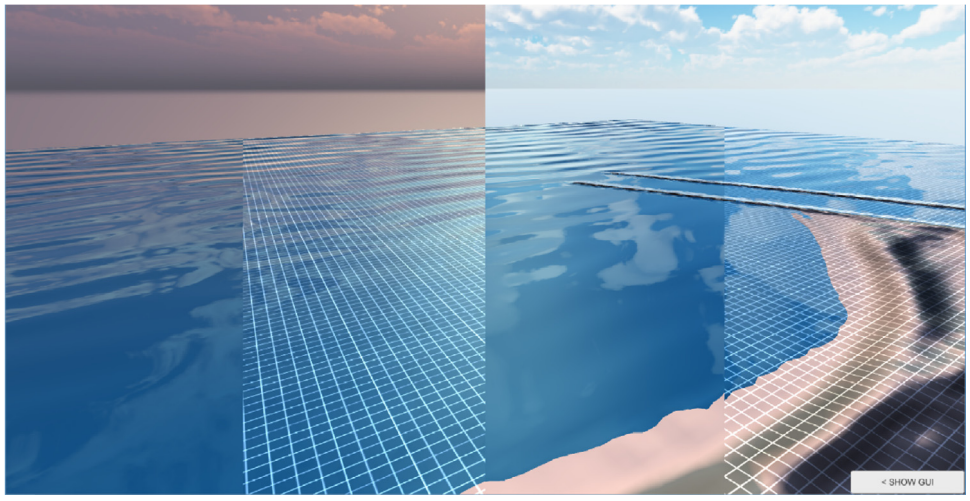

Fig. 6. Sample photorealistic rendering in Celeris Base with different sky and grid options; this figure is consisted of four snapshots stitched to each other.

colormap used for shading the water surface according to a flow variable and a linear mapping specified by max and min values. In water colormapping, the shader gets the value of the shading variable from the vertex shader (e.g., water surface elevation), then calculates the UV coordinates from

$$U = \frac{C - C_{min}}{C_{max} - C_{min}}; \quad V = 0.5 \tag{10}$$

where C is the value of the shading variable. Since the given textures are constant in the V direction, any value between 0 and 1 can be used for V. These UV coordinate values are then used to sample the color from the colormapping texture. The terrain colormap works similarly with a subtle difference. In case of the terrain, two distinct linear mapping functions are used for heights above and below the sea level. Eq. (11) shows the equations used to calculate the U and V values in the terrain colormapping shader. Terrain colormapping shader takes the texture shown in Fig. 5c as an input.

$$\begin{cases} U = \dfrac{C - C_{min}}{C_{min}}, & C \le 0 \\ U = \dfrac{C}{C_{max}}, & C > 0 \end{cases}; \quad V = 0.5 \tag{11}$$

4.2.6. Map rendering

Celeris Base has a new feature which enables it to automatically download the geographic map of the simulation location and render it on the terrain. To specify the geographic location of the simulation, the latitude and longitude of the center of the field must be given in the CML input file. The map of the location is then downloaded by sending a request to the Google Maps Static API. This API takes several variables to set the visual effects of the map along with the latitude and longitude of the map center and its zoom level. Celeris Base must calculate the required zoom level such that the retrieved image covers the entire simulation field, and then crop the image appropriately. To explain how we implemented these calculations in Celeris Base, we first need to explain how web-based geodatabases map the spherical earth surface on a cylinder.

All the major web map services (e.g., Google, Bing, OpenStreetMap, etc.) use a mapping technique called Web Mercator (or Google Web Mercator) which is a variant of Mercator projection. This projection maps the locations from a sphere (earth) to a cylinder. The cylinder is then unwrapped to form the well-known planar map we see on our screens or mobile phones.

In Celeris Base, a class called GoogleStaticMap handles downloading and other calculations related to the geographic maps. This class is part of a Unity3D asset, Lean Go Maps, developed by the author and published on the Unity Asset store under the MIT License. The asset is available at https://github.com/SasanTV/Lean-Go-Maps.

Let us define the M_x and M_y as the coordinates in the Web Mercator such that for the full earth map both coordinates span from -1 to 1. The following equations map the longitude and latitude to M_x and M_y

$$M_x = \frac{lon}{\pi} \tag{12}$$

$$M_y = \frac{1}{\pi} \ln\left(tan\left(\frac{\pi}{4} + \frac{lat}{2}\right)\right) \tag{13}$$

where *lon* and *lat* are longitude and latitude in radians. Since the Mercator projects poles to infinity, the poles cannot be shown on the map, and therefore the map is cut off at $\pm 85.051129°$. This latitude is where we have $M_y = \pm 1$.

The Google Maps Static API receives an integer value called zoom level. This value is used to calculate the tile size which will be cropped from the cylinder and returned to the user. If *zoom* denotes this value, 2^{zoom} is the total number of the tiles in each direction. For example, $zoom = 0$ corresponds to the entire map, and $zoom = 2$ divides the full map into $2^2 \times 2^2 = 16$ tiles. Given the lat/lon of a location and the zoom level, the API calculates the tile size and returns an image of that size. In Celeris Base, we do the reverse calculation to find the right zoom level. The first step in the reverse calculation is finding the *lat*/*lon* of corners of the field by using the haversine formula to convert distances in meters (width and length) to *lon* and *lat*.

After the required zoom level is calculated, Celeris Base assembles and sends a REST request to the Google Maps Static API. By default, Celeris Base Sends the request for the satellite imagery map, but it can be configured to request other kinds of maps. The response is a tile of the map centered in the location specified in the request. This image is given to a shader defined in *TerrainMap.shader*. The shader receives four parameters (two in each direction) to form a linear UV mapping between the domain and its geographic map. Remember, the geographic map returned by Google is larger than the simulation domain. Fig. 7 shows the geographic map of the coast of Newport, OR, retrieved from Google Maps Static API by Celeris Base to render the terrain in an example experiment in the area. Fig. 8 shows this map projected on the terrain in the simulation.

Fig. 7. Geographic map of the coast of Newport, OR, retrieved from Google Maps Static API by Celeris Base.

4.2.7. Mouse pointer gauge

The mouse pointer works like a gauge in Celeris Base, such that by hovering it over any point in the simulation field, the software shows the flow parameters and the position of the point in the field. To find the location of the pointer on the field, the software casts a ray from the pointer and in the heading direction of the camera. Then, it calculates the intersection of this ray (a line in the 3D space) with a horizontal plane placed on the mean sea-level. There is some error associated with this approach since the water surface elevation might be lower or higher than the mean sea-level and therefore the ray intersection with the water surface does not necessarily match with its intersection with the mean sea-level plane. However, the error is not significant, especially since the position of the gauge is also reported using the same technique, the flow parameters always correspond to the gauge location, even if the location is not precisely under the mouse pointer. Finding the exact intersection of the ray and water surface is computationally expensive and, considering the small error margin, not practically important. After finding the position of the mouse pointer on the field, the software reads the flow data from the textures using the *GetPixel* method (see Fig. 9).

4.2.8. Buoy

Celeris Base implements an interesting feature, where the user can deploy a "buoy" in the field and observe the readings from the buoy for the water height in real time. This feature is implemented using the tools used to implement the mouse pointer gauge feature. The values read from the GPU at the location of the buoy using these tools, are then plugged into a third party plotter asset that is shown on the screen (see Fig. 10). A maximum of five buoys can be deployed by clicking on the scene. They can also be removed by clicking on the plotter legend while pressing the Ctrl button.

4.3. Input and output files

Celeris Base uses the same format for input and output files as in Celeris Advent. The input setup for a specific experiment is given as an XML file, which is called the input CML file. The bathymetry of the domain is given as a text grid file in CBF format. A CBF file starts with tags which determine the resolution of the bathymetry. Then a matrix of numbers is followed in which each row corresponds to the z value of the cells in a row of the solution field. If the resolutions of the bathymetry (given in the CBF file) and the experiment (given in the CML file) are different, Celeris Base will use interpolation.

4.4. Compilation

Unity3D is a cross-platform game engine, and therefore it can build projects for several platforms. However, using compute shaders imposes some limitations in building the project for different platforms. We developed Celeris Base on a Windows machine with the intention of running it on Windows machines. Although Unity supports compute shaders on several platforms there is not a guarantee that Celeris Base will run on all of these platforms. At the early stages of developing Celeris Base, we successfully compiled and ran the software on a Linux machine. We expect that our software can still be built for Linux machines,

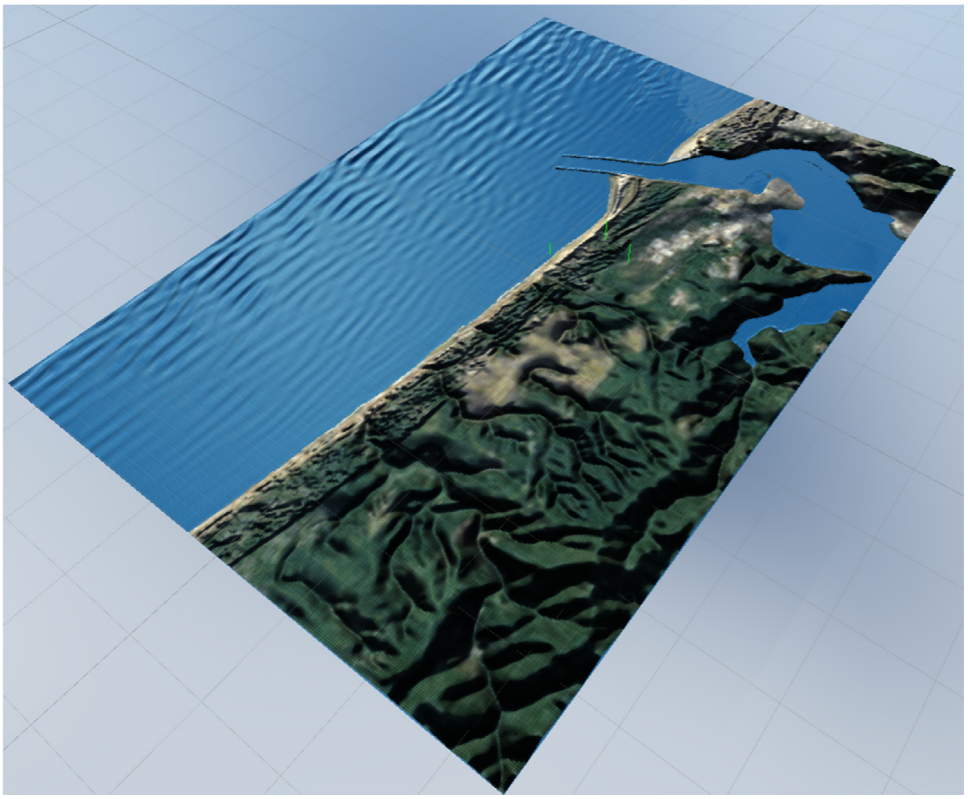

Fig. 8. Simulation of the coast of Newport, OR, using Celeris Base with rendering the geographic map of the location.

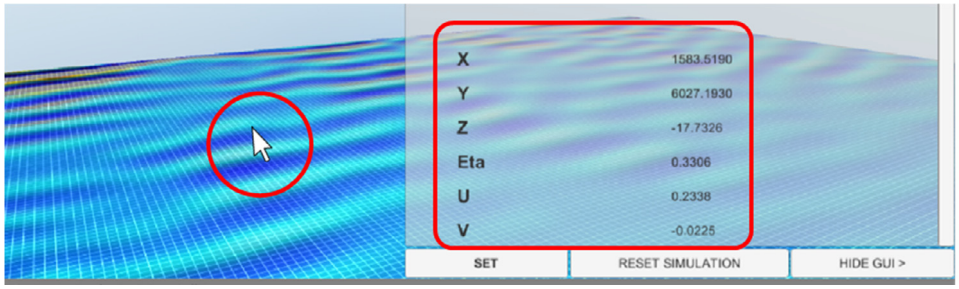

Fig. 9. Mouse pointer acts as a gauge in Celeris Base.

with some minor modifications. We believe that the software can be built for different platforms and encourage researchers to do so, but the process will require some necessary changes in the software. For example, Windows' DirectX gracefully handles out-of-bound access in textures by returning zero, while this may cause a crash on some platforms. Such cases must be found and removed from Celeris Base to prepare for compilation for different platforms. To build Celeris for Windows, you should first set the build platform to "PC, Mac & Linux Standalone" from File > Build Setting, then, set the target platform to Windows in the same setting window and build the project.

4.5. Running Celeris Base

Running Celeris Base on Windows is similar to running Celeris Advent and is done by running the executable file. Celeris Base first looks at the `setting.init` file located in the same folder as the executable file itself, and if a path to a CML file is not provided, it opens a file browser window, where the user can select the input CML file. The GUI of Celeris Base also resembles Celeris Advent, with some minor differences. In order to use the geographic map feature of Celeris Base, a Google Static Maps API key must be provided in the `GoogleStaticMap.cs` file.

4.6. Immersive visualization

Applications of virtual reality (VR) in science span from research in medical sciences and surgery to molecular physics [25,26]. The most common VR technology uses VR headsets which generate realistic images, audio and other sensations in accordance with user's real time movement such that they simulate a user's physical presence in a virtual environment. For example, a user can move in a virtual museum and observe different items, or with the controllers they can grab objects and interact with the environment. Despite the existence of several commercial applications of VR and AR in 3D modeling and design, practical applications in numerical scientific simulation and visualization are rare. These novel visualization techniques can significantly help coastal researchers to have a better understanding of complex processes in a visual format.

Virtual reality can help coastal researchers and engineers by taking them into a virtual world where they can interact with the

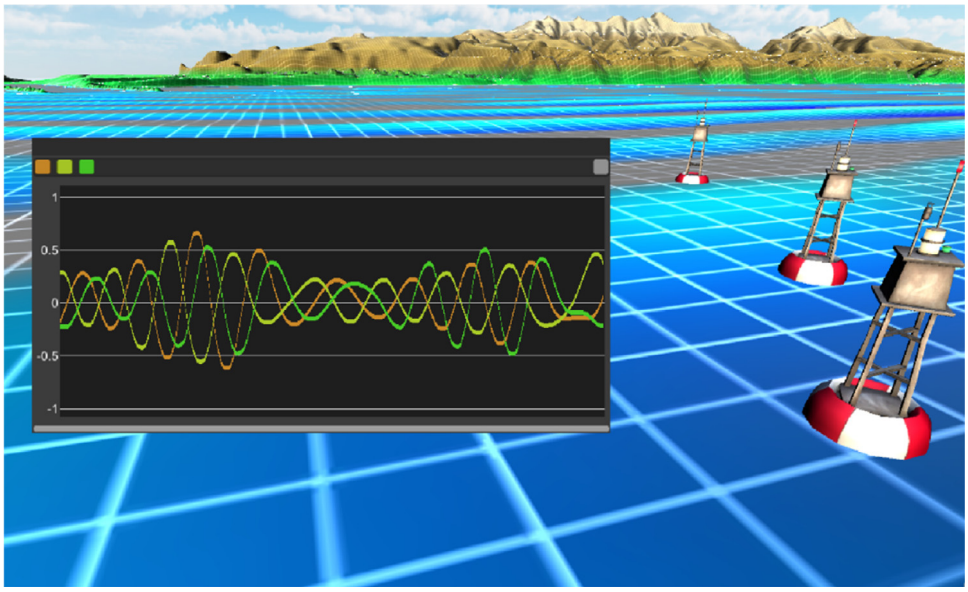

Fig. 10. Deploying virtual buoys in Celeris Base.

waves and observe them in an immersive way. The potential of such technology is endless. For instance, imagine entering a harbor on a boat and experiencing the waves first hand in VR, before even the harbor is built, or setting up a virtual laboratory with several flumes and wave tanks where students and researchers can collaborate. Despite the huge potential of applications of VR in coastal research, to the best of our knowledge, there is no prior work in this area. One reason is that developing a VR coastal simulation software requires super-fast interactive models, which did not exist before development of Celeris Advent. In this section we discuss Celeris VR, which is built on top of Celeris Base and provides the users with an immersive environment. We demonstrated Celeris VR for the first time in [27] and later in [28].

4.6.1. Implementation

We targeted our software for Oculus Rift headsets, which are one of the most advanced and popular VR headsets as of 2019. The VR support in Unity3D is simply enabled by checking a checkbox in Edit > Project Settings > Player. It is not an exaggeration to assert that integrating VR into Celeris Advent, could have taken as much time as developing the entire Celeris Base from scratch. Although, getting started with the VR in Unity3D is simple, there still is need for some development and integration efforts. For example, the player movement and interaction in the field using the controllers need some coding, which can also be eased by plugging in and modifying ready to use libraries and Unity assets.

4.6.2. Running Celeris VR

Celeris VR looks very similar to Celeris Base, except that it lets the user mount an Oculus Rift headset, jump into the virtual world, and move around as a flying object. Fig. 11 shows the controllers to move the player in the field. The left controller lets the user move in different directions and the right one lets them rotate. Note that the user can also rotate their view in the field by physically rotating in the room, however the rotation controller helps them to adjust their view and direction of movement easier. Fig. 12 shows a user (the author) working with Celeris VR, simulating a coastal area. In this scene, two buoys are deployed in the field and their real-time plots are shown on the screen.

The immersive environment in Celeris VR provides a unique experience for users. The feel of presence inside a numerical

Fig. 11. Movement control in Celeris VR.

coastal simulation is unprecedented and it lets the user observe the coastal phenomenon as if in person, but with the addition of powerful visualization tools such as buoys, colormapping, etc. Celeris VR is also released as an open source software and under MIT License. The external libraries used in the software, carry their own licenses.

5. Numerical validation

Since Celeris Base uses the same mathematical and numerical model as Celeris Advent, validations of Celeris Advent can be accounted for Celeris Base as well. Celeris Advent was validated for wave and current simulation by the authors in [3,24,29] and by several independent researchers [30–32]. However, in order to ensure that we have correctly re-implemented the solver in Celeris Base, we validate the model against one of the experiments of Briggs et al. [33] for solitary wave interaction around a conical island which is frequently used to validate numerical models [12,34–36]. In these experiments, a circular island with 7.2 m base diameter and side slope is located in a 30 m x 25 m wave tank with 0.32 m depth. We only simulate the case with target relative wave heights of $H/d = 0.20$ which has a higher non-linearity and wave breaking condition, therefore is expected to be more challenging for a wave model.

Our numerical setup for the conical island experiments consists of a 30 m $\times$ 30 m domain with the conical island in the center and a solitary wave placed as an initial condition near

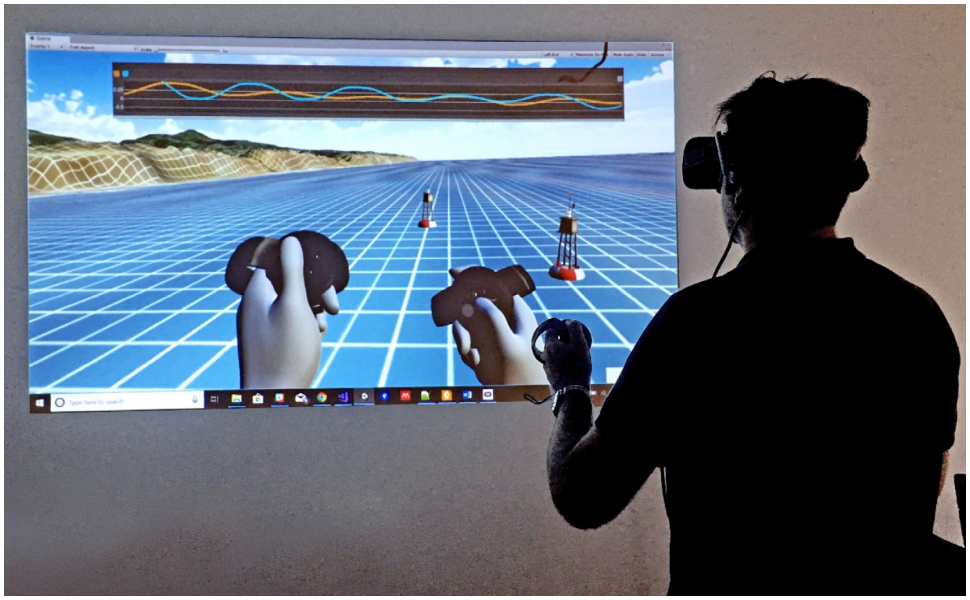

Fig. 12. Simulating a coastal area with Celeris VR.

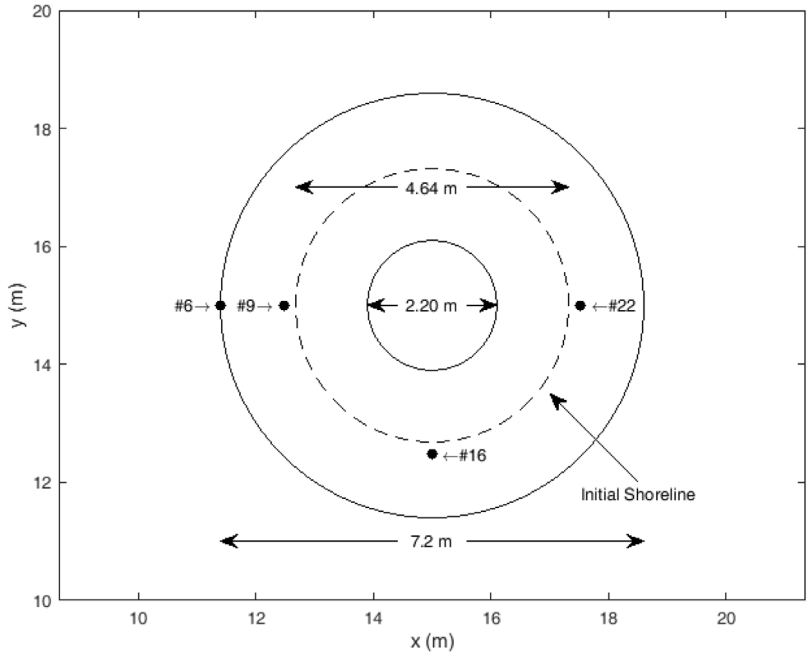

Fig. 13. Experimental setup of the conical island. The gauge locations are shown by dots and the wave approaches the island from the left.

the west boundary. The west and east boundaries are set to the sponge layer condition, while the north and south boundaries are fully reflective solid walls. The domain is discretized by 301 $\times$ 301 cells and a timestep of 0.0025 s. Bottom friction is not applied. The test case is performed with a slightly smaller relative wave height at $H/d = 0.18$ which is used in several other studies such as [12,36], and [35], as it is closer to the wave height ratio observed downstream of the wavemaker.

Fig. 13 shows the gauges locations in the experimental setup. Two gauges (#6 and #9) are in front of the island, while one (#16) is on the side, and one (#22) is behind the island. Figs. 14 and 15 compare the experimental and numerical results for the time-series at gauges and maximum horizontal runup on the island, respectively. The initial waveheight and subsequent drawdown are predicted well in Fig. 14, which is consistent with numerical results from Celeris Advent [3] as well as results from other Boussinesq-type solvers [12,34–36]. The maximum horizontal runup on the island is also predicted very well by Celeris Base and is identical to results from Celeris Advent.

6. Conclusion

We introduce our open source software for coastal wave simulation, called Celeris Base. This software is a revamped version the widely used Celeris Advent Boussinesq-type model. Celeris

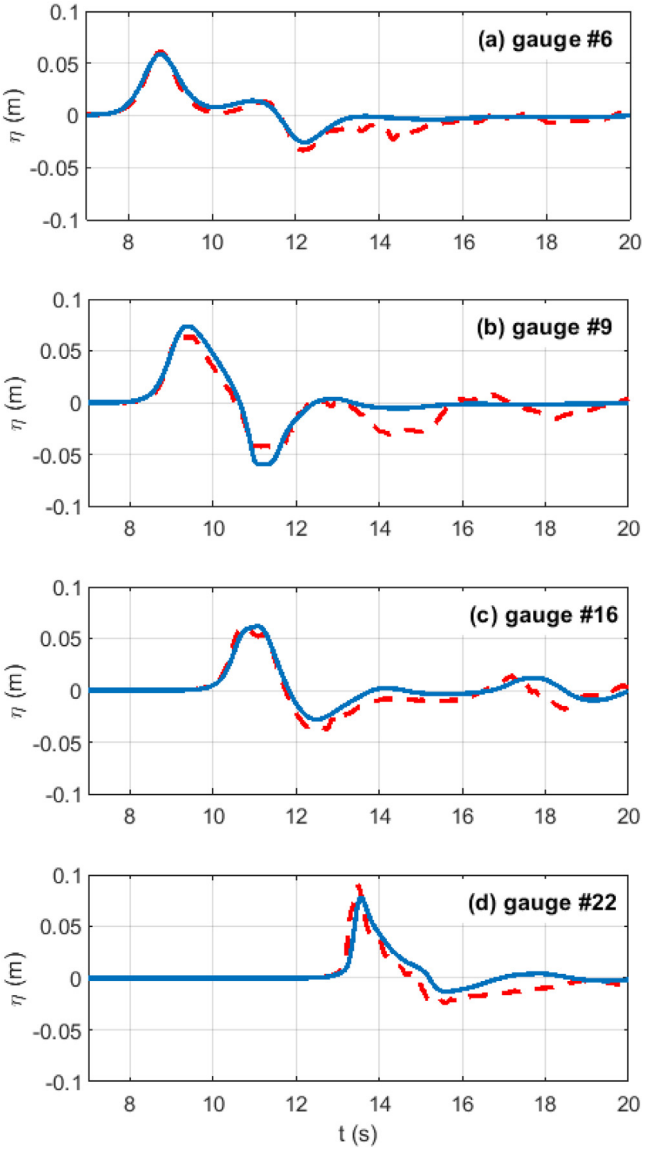

Fig. 14. Experimental (– –) and numerical (–) time series for the interaction of a solitary wave with H/d = 0.18 on a conical island, at gauges #6, #9, #16, and #22 (a–d), simulated by Celeris Base.

Base is implemented in Unity3D using C# and HLSL. The scientific equations are solved using compute shaders, unlike Celeris Advent which uses regular pixel shaders. The discretization of the extended Boussinesq equations is done using our hybrid finite volume–finite difference scheme and implemented on GPU. The structure of the software and source files are explained to help researchers develop the software further. We introduced several new features of Celeris Base such as its ability to overlay geographic maps and its immersive virtual reality environment. A compiled version of Celeris Base is distributed along with its source codes under terms of the MIT License.

Acknowledgments

This research was partially funded by the Office of Naval Research (ONR) award number N00014-17-1-2878. We acknowledge the helpful comments and suggestions of an anonymous reviewer.

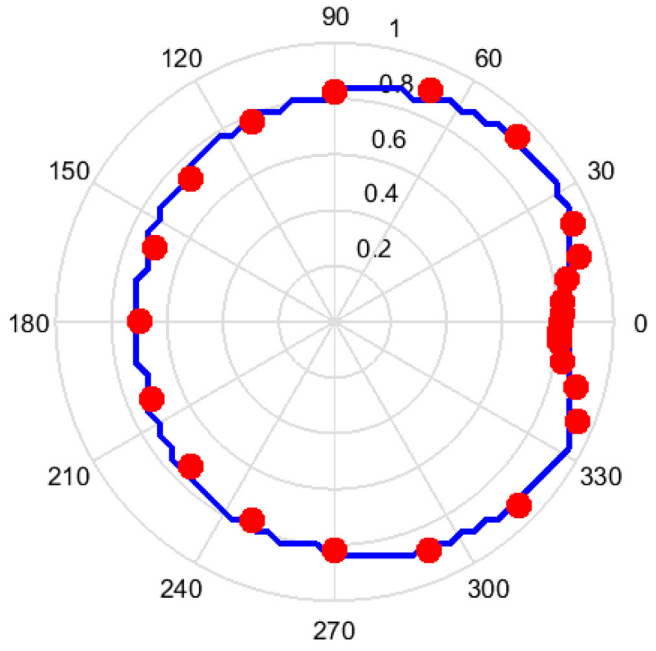

Fig. 15. Numerical (solid line) and measured (•) maximum horizontal run-up for H/d = 0.018 simulated by Celeris Base.

References

[1] C. Synolakis, U. Kanoğlu, Phil. Trans. R. Soc. A 373 (2015) 20140379.
[2] S. Tavakkol, Interactive and Immersive Coastal Hydrodynamics, University of Southern California, 2019.
[3] S. Tavakkol, P. Lynett, Comput. Phys. Comm. 217 (2017) 117–127.
[4] M. Brocchini, Proc. R. Soc. A 469 (2013) 20130496.
[5] J.T. Kirby, J. Waterw. Port Coast. Ocean Eng. 142 (2016).
[6] D.H. Peregrine, J. Fluid Mech. 27 (1967) 815–827.
[7] P.A. Madsen, R. Murray, O.R. Sorensen, Coastal Eng. 15 (1991) 371–388.
[8] O. Nwogu, J. Waterw. Port Coast. Ocean Eng. 119 (1993) 618–638.
[9] Y. Chen, P.L.-F. Liu, J. Fluid Mech. 288 (1995) 351–381.
[10] P. Lynett, P.L.F. Liu, K.I. Sitanggang, D. Kim, Modeling Wave Generation, Evolution, and Interaction with Depth-Integrated, Dispersive Wave Equations COULWAVE Code Manual Cornell University Long and Intermediate Wave Modeling Package V. 2.0, Cornell University, Itacha, New York, 2008.
[11] Q. Chen, R.A. Dalrymple, J.T. Kirby, A.B. Kennedy, M.C. Haller, J. Geophys. Res. Ocean 104 (1999) 20617–20637.
[12] P.J. Lynett, T.-R. Wu, P.L.-F. Liu, Coastal Eng. 46 (2002) 89–107.
[13] S. Ryu, M.H. Kim, P.J. Lynett, Comput. Mech. 32 (2003) 336–346.
[14] P. Lynett, P.L.-F. Liu, in: Proc. R. Soc. London A Math. Phys. Eng. Sci. 2002, pp. 2885–2910.
[15] P.A. Madsen, O.R. Sørensen, Coastal Eng. 18 (1992) 183–204.
[16] P.J. Lynett, K. Gately, R. Wilson, L. Montoya, D. Arcas, B. Aytore, Y. Bai, J.D. Bricker, M.J. Castro, K.F. Cheung, C.G. David, G.G. Dogan, C. Escalante, J.M. González-Vida, S.T. Grilli, T.W. Heitmann, J. Horrillo, U. Kanoğlu, R. Kian, J.T. Kirby, W. Li, J. Macías, D.J. Nicolsky, S. Ortega, A. Pampell-Manis, Y.S. Park, V. Roeber, N. Sharghivand, M. Shelby, F. Shi, B. Tehranirad, E. Tolkova, H.K. Thio, D. Velioğlu, A.C. Yalçıner, Y. Yamazaki, A. Zaytsev, Y.J. Zhang, Ocean Model. 114 (2017) 14–32.
[17] G. Wei, J.T. Kirby, J. Waterw. Port Coast. Ocean Eng. 121 (1995) 251–261.
[18] K.S. Erduran, S. Ilic, V. Kutija, Internat. J. Numer. Methods Fluids 49 (2005) 1213–1232.
[19] M. Tonelli, M. Petti, Coastal Eng. 56 (2009) 609–620.
[20] A. Kurganov, G. Petrova, Commun. Math. Sci. 5 (2007) 133–160.
[21] F. Shi, J.T. Kirby, J.C. Harris, J.D. Geiman, S.T. Grilli, Ocean Model. 43–44 (2012) 36–51.
[22] V.V. Titov, C.E. Synolakis, J. Waterw. Port Coas. Ocean Eng. 121 (1995) 308–316.
[23] V. Titov, U. Kanoğlu, C. Synolakis, J. Waterw. Port Coast. Ocean Eng. 142 (2016).

[24] S. Tavakkol, P. Lynett, in: Proc. Coast. Eng. Conf. 2016.
[25] M.A. Spicer, M.L.J. Apuzzo, Neurosurgery 52 (2003) 489–498.
[26] A. Sharma, A. Nakano, R.K. Kalia, P. Vashishta, S. Kodiyalam, P. Miller, W. Zhao, X. Liu, T.J. Campbell, A. Haas, Presence Teleoper. Virtual Environ. 12 (2003) 85–95.
[27] P. Lynett, S. Tavakkol, in: 36th Int. Conf. Coast. Eng. 2018.
[28] P. Lynett, S. Tavakkol, in: AGU Fall Meet. Abstr. 2018.
[29] S. Tavakkol, S. Son, P. Lynett, ArXiv Prepr. ArXiv:1909.04153 (2019).
[30] M. Queijeiro Rilo, Estudio Del Clima Marítimo Y Diseño de Una Protección Del Litoral de San Andrés, Tenerife, University of Cantabria, 2018.
[31] G. Pérez González, Predicción Del Remonte Del Oleaje (Ru2%) En Diques En Talud Con Un Modelo de Boussinesq, University of Cantabria, 2018.
[32] V.A. Bheeroo, Long Wave Amplification in a Coral-Reef Lagoon, Oregon State University, 2019.
[33] M.J. Briggs, C.E. Synolakis, G.S. Harkins, D.R. Green, Pure Appl. Geophys. 144 (1995) 569–593.
[34] V.V. Titov, C.E. Synolakis, ASCE J. Waterw. Port Coast. Ocean Eng. 124 (1998) 157–171.
[35] D.R. Fuhrman, P. a Madsen, Coast. Eng. – Amsterdam 55 (2008) 139–154.
[36] M. Tonelli, M. Petti, Ocean Eng. 37 (2010) 567–582.