Using deep neural networks to simulate the water surface surrounding an obstacle using Unreal Engine 4

Paweł Kowalski

https://github.com/p4vv37/ueflow
https://www.youtube.com/watch?v=oB-kbE85IRU
https://www.pkowalski.com
10.11.2023

Chapter 1

Introduction

This work aims to develop a neural network-based interactive simulation of water surface dynamics that meets specific requirements tailored to the simulation needs for gaming purposes:

- High performance, allowing for a satisfactory frame rate. The minimum acceptable value is set at 30 frames per second (fps). This implies that the simulation calculation time, data transfer, and any other operations necessary before generating each frame must not exceed 33.3 ms.
- The ability to achieve the desired surface style (artistic direction) at the expense of the physical accuracy of the simulation.
- The capability for interaction with the water surface.

The project assumes the utilization of neural networks. Several types of networks were analyzed during the project. It was decided that the network would generate data as a two-dimensional matrix representing the water's surface height. The input data for the network would include information about interaction with the fluid surface and the water surface height at the moment preceding the simulated one. If a frame other than the first is generated, the water height used as input data was previously generated by the same network for the preceding frame.

To train the network, it was decided to prepare a fluid simulation using tools available in the SideFX Houdini application. The application was chosen due to the ease of implementing the data generation process and the availability of the FLIP fluid physics engine. The simulation involves the interaction of a moving object with the water surface, where the object is partially submerged. The simulation was sampled at a rate of thirty samples per second, corresponding to the assumed thirty frames of simulation per second.

The generated examples contain information analogous to that provided by the game engine to the interactive application during operation, including:

- Information about the obstacle shape is stored in the form of a Signed Distance Field (SDF).
- Information about the direction of the object's movement.
- Information about the speed of the object's movement.
- Current water surface height.
- Current distribution of foam on the water surface.

All information is processed before being input into the network in such a way that it takes the form of two-dimensional matrices. This allows for the use of convolutional neural networks. The network is based on a modified U-Net architecture [5]. A network serving as a discriminator, similar to the one in GAN networks, was employed as part of the loss function.

An interactive presentation was prepared, including the simulation. The simulation corresponds to the one performed to generate samples in the SideFX Houdini program but utilizes the trained network as the source of water surface height information.

Chapter 2

Work Conducted

This chapter delineates the endeavors directed toward the development of an interactive water simulation through the application of neural networks.

The work is divided into two stages. It was decided that initially, a very simple form of simulation would be considered. Using this simplified simulation, a dataset was generated, which was then utilized to train a basic neural network. This approach allowed for making decisions regarding simulation details, identifying potential issues and limitations, confirming the feasibility of the project, and preparing and testing the necessary tools. The next stage involved expanding the simulation.

2.1 Water surface simulation

2.1.1 Water Surface Simulation Preparation

The first scenario involved simulating a water surface within fixed boundaries, where a cylindrical obstacle moved within the designated area. Reflections of waves from the area boundaries were intentionally omitted. A schematic representation of the simulation is shown in Figure 1.

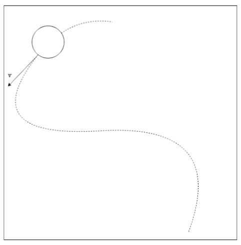


Figure 1. Schematic representation of the basic simulation. Source: author's own.

The simulation was conducted using the SideFX Houdini application, employing the Flat Tank tool [6]. This tool utilizes the FLIP algorithm (Fluid-Implicit-Particle, based on the Lagrange method) for simulating fluid within a specified region. The simulation focused on a limited water area as part of a larger tank. This approach eliminated the need to define side walls or a bottom, maintaining a constant water level automatically. The animation of obstacle movement was generated using a Python script creating a curve and the application's tool for object movement along the path during animation progression. Multiple simulations were performed, and script parameters were empirically adjusted to achieve the desired wave characteristics.

To expedite the simulation calculation process, GPU parallelization using OpenCL in the application was employed.

2.1.2 Dataset Generation

The dataset was represented using several arrays. Each array stored specific values from the simulated frames as they progressed. The collected data included:

- A two-dimensional matrix representing the water surface height at a given point determined by X and Y coordinates (Figure 2), representing the final effect of the considered frame,
- Three float values representing the obstacle's position in the frame with index N.

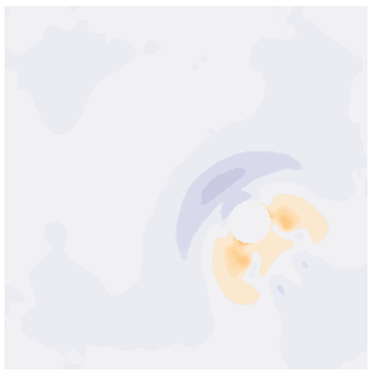
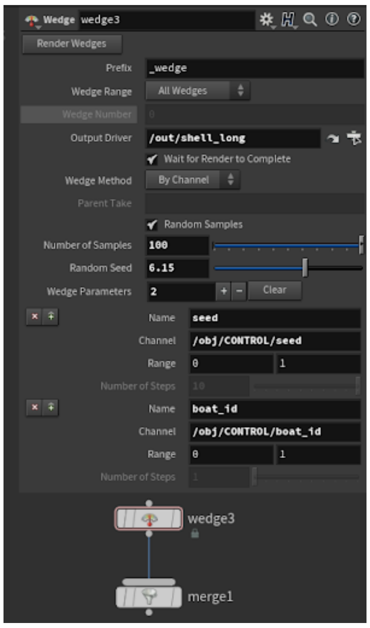


Figure 2. Visualization of the matrix representing the water surface height. Source: author's own.

To obtain the simulation state at frame index N, one needed to select the element at index N from each generated array. The obstacle's position required only two values; representing its position perpendicular to the flat water surface was unnecessary, always having a zero value. Optimization was planned by transforming the position representation into a two-dimensional array. Simulations were performed for various path scenarios with a fixed length of 1000 frames each.

Parallel execution of multiple simulations was planned to make full use of the computer's computational power. This could be achieved by running more than one application instance, a method successfully applied in past projects. However, due to the impossibility of using the Procedural Dependency Graph (PDG) tool, the decision was made to run a single application instance and use the wedge tool, executing selected tasks for different parameter values within a single thread. A graph example utilizing the wedge tool is shown in Figure 3.



Wedge wedge3

Render Wedges

Prefix	_wedge
Wedge Range	All Wedges
Wedge Number	0
Output Driver	/out/shell_long
	✓ Wait for Render to Complete
Wedge Method	By Channel
Parent Take	
	✓ Random Samples
Number of Samples	100
Random Seed	6.15
Wedge Parameters	2

Name	seed	
Channel	/obj/CONTROL/seed	
Range	0	1
Number of Steps	10	
Name	boat_id	
Channel	/obj/CONTROL/boat_id	
Range	0	1
Number of Steps	1	

Figure 3. Example graph utilizing the wedge tool. Source: author's own.

Experiments revealed significant memory consumption by arrays containing input data and simulation results. Cleaning these arrays and saving new files containing them every 100 simulation frames was implemented to address this issue.

The generated data needed to be transformed into a format usable by the neural network. Several representation types were prepared, and their performance was examined. Jupyter Notebook was employed for this task, allowing visualization and examination of the preparation process. TensorFlow Dataset was utilized to prepare input pipelines for the neural network.

Various neural network architectures were examined with different input data representations to find the most promising combination. For this purpose, functions creating different network models were prepared, and their training was executed for 15 epochs. Subsequently, the mean squared error value and empirically generated results were compared.

The data were utilized in the form of sequences of input and output data frames succeeding each other, similar to the recommendation in the TensorFlow documentation [7]. A CustomModel class based on the keras.Model class was prepared to handle the training mechanism. The training scheme is schematically presented in Figure 4.

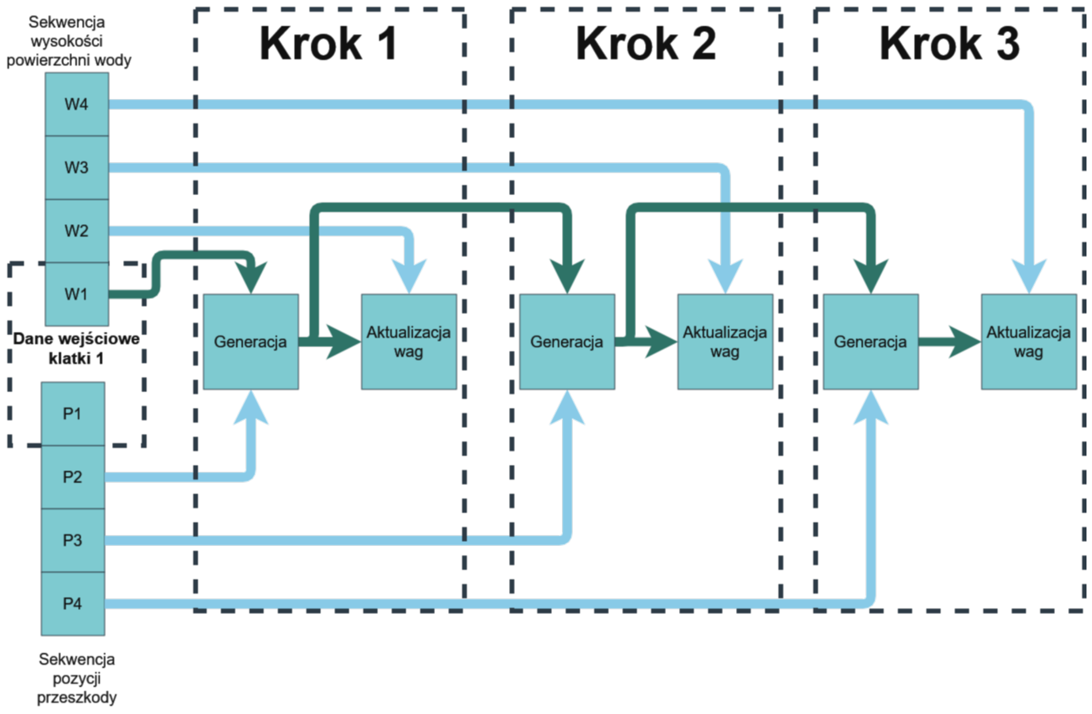


Figure 4. Graph used for processing and training the neural network. Source: author's own.

Separate Tensorflow Dataset pipelines were utilized for both types of input data. These pipelines were then combined using the Dataset.zip function, followed by the use of the shuffle operation to randomize the data at each epoch. Data generated in the animation process were stored in ṅp files containing Numpy arrays of length 100, each representing a corresponding value read at the frame corresponding to its index.

The first step in the code section responsible for data processing was to locate all files in the specified disk location containing the generated data. This was accomplished by parsing their names. The filtered list of file names was then transformed into a data pipeline. The load_file function was executed on this pipeline using the map method. The map method performs a convolution action: it runs the specified function for each element of the pipeline, creating a new pipeline element that is an array of results from the mentioned function. The designated function opened the file specified by the pipeline element name, and then returned it, resulting in the pipeline elements being one hundred-element arrays of values generated during the simulation. The window method of the pipeline was then used to create shorter sequences. Using the flat_map method, one dimension of the pipeline was removed, transforming the pipeline containing a table of such sequences in each element into a pipeline containing a sequence.

Passing the water surface height in the preceding frame as a two-dimensional matrix was an obvious choice. However, the format in which the data regarding the obstacle's position and its displacement between frames should be represented was not clear. Several forms of representation and network architectures were considered.

Initially, the decision was made to pass only information about the obstacle's position in the considered frame at index N and the water surface height in the frame preceding it at index N - 1. In the case of a recurrent network, the height of the water surface in frame N-2 was also passed. Although information about the change in position or information about the position in frame N-1 may seem necessary, it was decided that during these initial experiments, it would not be transmitted. It was speculated that perhaps the network is able to obtain this information by comparing the position of the wave front, essentially at a specific distance equal to the radius of the cylinder from the center of the obstacle, to the obtained data about its position in the considered frame. At the same time, water closer to the center of the obstacle than its radius must directly affect the rise of the wave front. Another argument was that a satisfying result of the network at this stage would be its ability to prolong the motion of existing waves. Therefore, it was decided that additional information would be taken into account in the input to the network in subsequent stages of work if the results prove promising.

Many convolutional neural network variants were prepared, which took information about the obstacle's position in the frame at index N and the water surface height in the frame at index N-1 as input. Several versions of a recurrent neural network based on the GRU architecture were also explored, which took information about the obstacle's position in the frame at index N and the water surface height in the frames at indices N-1 and N-2 as input. A variant of the network was also tested in which convolutional neural networks were combined with recurrent elements. A schematic representation of this variant is shown in Fig. 5.

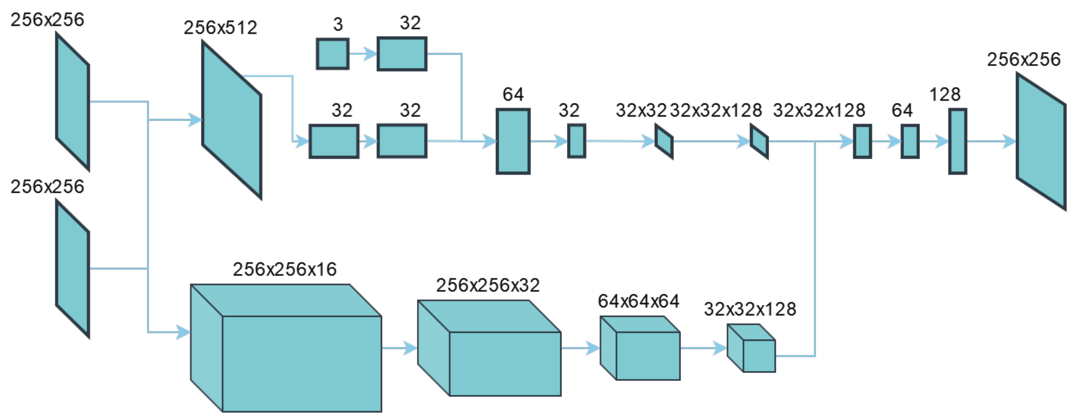


Figure 5. Schematic representation of the recursive variant of the network. Source: author's own work.

The mean squared error was utilized as the loss function. Monitoring the changes in the mean squared error value and conducting empirical evaluations were employed to assess the quality of the generated values. Regardless of the tested network, the squared error value stabilized after a few epochs around the number 0.0013. Empirical investigations into the results of network generation revealed that the networks did not generate waves; instead, they gradually approached zero values. It was concluded that the network would not be able to simulate the given scenario, leading to the decision to modify it.

2.1.3 Introduced Modifications and Achieving Satisfactory Results

To identify the reasons for failure, existing solutions were examined. It was observed that many of them [3][1] are based on the presence of a stationary obstacle. It can be speculated that this allows the network to understand the structure of the considered space. It can be hypothesized that the network had a challenging task in understanding the relationship between the position of a point in space and the fluid behavior at that point. A stationary obstacle may facilitate the network in recognizing phenomena occurring at specific locations. Another identified problem was the representation of data regarding the position of the obstacle in space. A three-element vector is a structure that is challenging to use when working with convolutional neural networks. Therefore, a more complex data processing approach was chosen for the data obtained from the simulation. To determine the position of each point in space relative to the obstacle and convey information about the direction and speed of the obstacle's movement, the following representations of values were used:

- A matrix of values indicating the distance of each point in space from the center of the obstacle (values smaller than the radius of the obstacle were set to 0),
- Values of trigonometric functions determining the angle between the vector, with its beginning at the center of the obstacle and its end at the considered point,
- Gradient determining the direction and speed of the obstacle's movement (the direction of the gradient descent indicates the direction of its movement, while the rate of change of its value represents its speed).

It was decided that the simplified simulation would be suitable for simulating a small water area around a moving cylinder. The simulation area was also limited, allowing for a more detailed simulation. The simulation schematic is shown in Fig. 6, and an example of the obtained data can be seen in Fig. 7.

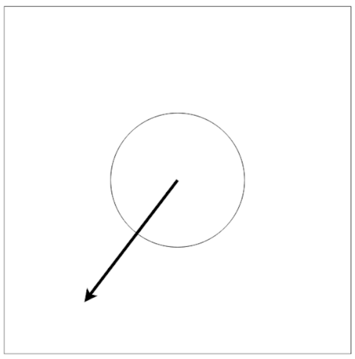


Figure 6. Schematic representation of the modified simple simulation. Source: author's own work.

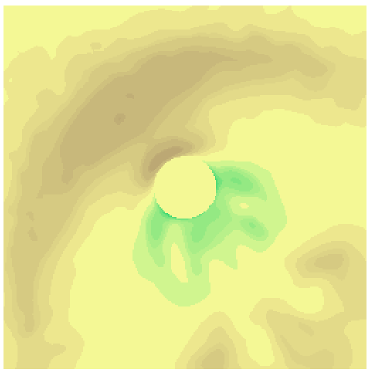


Figure 7. Visualization of the matrix representing the water surface height. Source: author's own work.

Multiple network architectures were investigated, and the most promising one was selected. The Pyradox library [4] was utilized to create a network based on the U-Net architecture [2]. Experiments showed that this network produced the best results among the considered options.

The network exhibited a tendency to exhibit a phenomenon known as Mode Collapse, as shown in Fig. 8. It was decided to prepare a complex loss function penalizing such behavior.

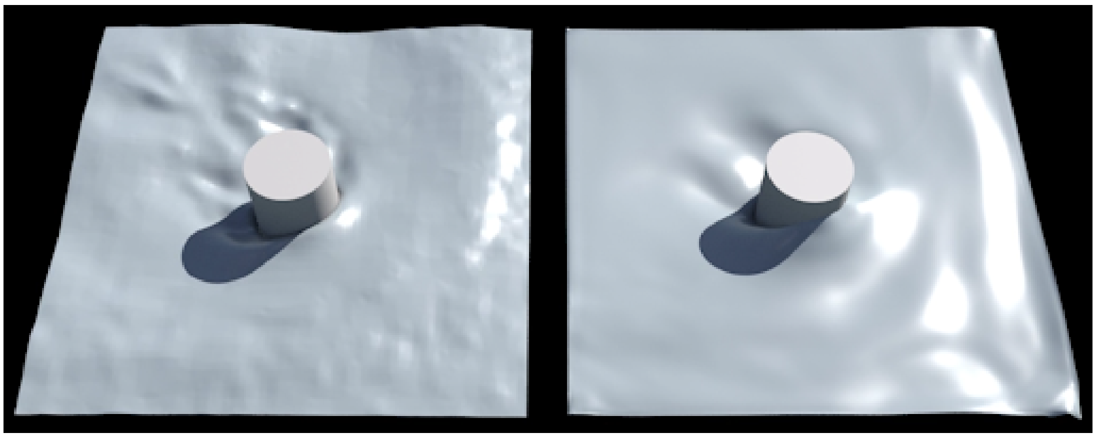


Figure 8. An example result of the network operation, where only mean squared error was used as the loss function. On the left are reference data, and on the right is the result of the network's operation. Source: author's own work.

The first penalty was calculated by determining the deviation of the expected value from the value corresponding to a flat, undisturbed water surface. It involved multiplying the square of this value by the square of the difference between the generated value and the expected value. As a result of this operation, in places where waves were expected but not generated by the network, the error value was significantly increased. The code responsible for calculating this penalty is presented below:

```
loss += tf.reduce_mean((1 - tf.square(2 * y_true - 1,0)) * (tf.square(y_true - y_pred)))
```

Listing 2.1. Code responsible for calculating the penalty related to creating a too flat water surface (simplified for clarity).

The next penalty involved comparing the maximum and minimum values of the generated matrix with the largest and smallest expected values. This penalty forced the network to generate a value range closer to the expected values and was particularly high in cases where the network generated a completely flat surface, even though it should be wavy.

These introduced penalties significantly improved the network's performance. The network was less cautious, and the generated data was not flat. Another encountered problem was the very blurry shape of the generated surface. They lacked sharp edges, a behavior typical of networks using mean squared error. Attempts were made to counteract this effect by using, in addition to mean squared error, structural similarity index (SSIM) error. However, the empirically assessed result was not significantly better.

The described actions increased the mean squared error value but improved the perception of the network's effect by humans. In the case of the studied issues, the quality of the network's operation is not easy to assess. The final evaluation is subjective because, in the considered example, the recognition of the surface as looking correct by a human observer is more important than the accuracy of simulating physics. This does not necessarily mean its physical correctness. The significance of this distinction can be noticed by analyzing the existing solutions for simulating water surfaces mentioned at the beginning of this work. Many of them are based on mechanisms unrelated to physically correct simulation. A network trained using mean squared error and relying solely on approximating the result of its operation to previously generated values tends to ignore them, creating a blurry result. A result close to the expected one but lacking small details may be perceived as too smooth or blurry.

Significant improvement was achieved by applying a more complex architecture, utilizing mechanisms typical of GAN networks. A network acting as a discriminator was used, and experiments established that its use compelled the network to generate the mentioned details, even if they negatively influenced the mean squared error. If the discriminator considered them more suitable than the reference data, the result was subjectively better. According to the author, this has a positive impact on the subjective quality of the network's operation.

To implement a network using a discriminator, the previously described CustomModel class was abandoned. Instead of overriding the method of the Model class, it was necessary to prepare several complex functions responsible for training the discriminator model. A new callback was also created, executed at the beginning of each epoch, whose task was to run this function. Its mechanism involved obtaining input data for the generator, using it to generate data based on it, and then passing this data to the discriminator, whose task was to recognize the generated data from the reference data. The trained discriminator was used to calculate the value of the loss function during the training of the generator. A generator loss function was prepared, aiming to increase the discriminator's error.

The result of the operation of such a trained network is presented in Fig. 9. Sharper edges and a higher wave height can be observed compared to previous examples. However, it is essential to note that these waves are also significantly higher than those in the reference data. This is a network error that can be corrected by adjusting the weights used in the loss functions.

The obtained results were considered satisfactory, demonstrating the feasibility of the project. Therefore, the decision was made to expand the simulation.

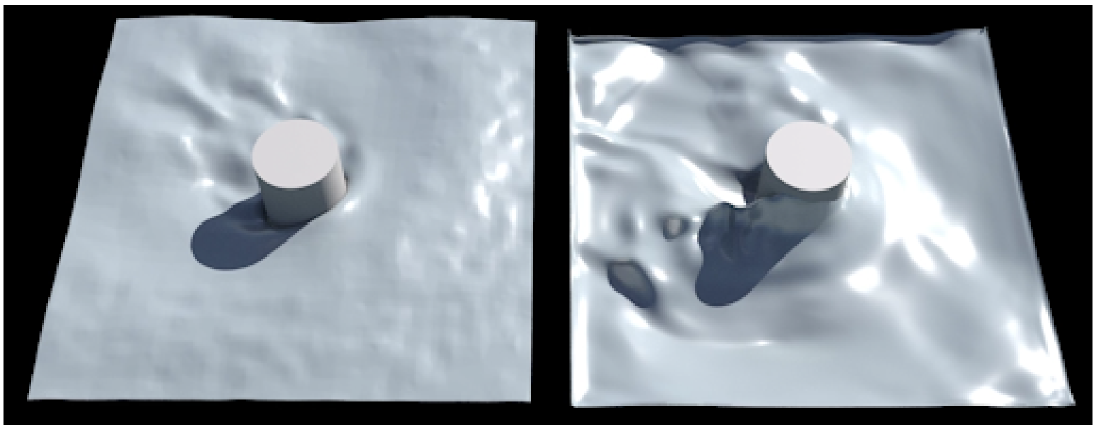
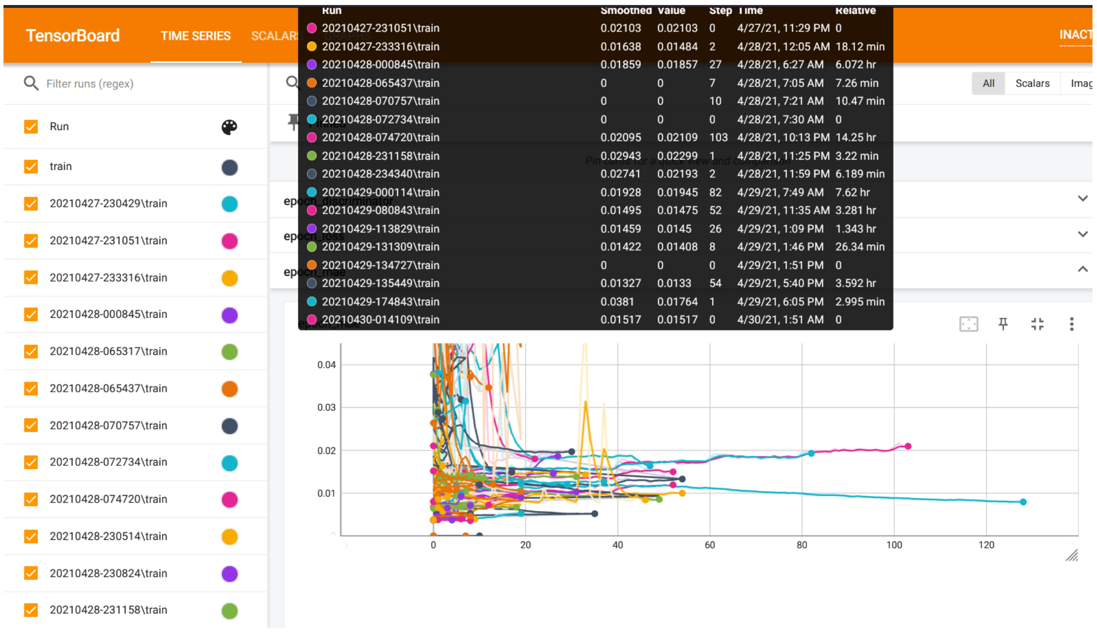


Figure 9. An example result of the network operation, using an architecture with a discriminator network and a complex loss function. On the left are reference data, and on the right is the result of the network's operation. Source: author's own work.

2.2 Training the network

The network training was conducted using the Jupyter Notebook. During the conducted research, over twenty versions of the training script were tested. These versions differed in the data preparation process, the type of data passed to the network, and the details of its architecture. The final version of the script consisted of over 700 lines of code responsible for data preparation, along with visualizations enabling the verification of their correctness, the creation of the network, the training loop, the logging of the experiment progress, and the execution of backups, as well as the presentation of results at various stages of the process. Tensorboard application was employed to monitor the process, as shown in Fig. 10, presenting a screenshot of the application displaying part of the logged training processes. The length of the training process varied because the training was interrupted when the results did not show improvement. Subsequently, an analysis was performed, the script or training parameters were modified, and the process was repeated. It is worth noting at this point that the mean squared error does not accurately reflect the quality of the results generated by the network. In the considered project, the crucial aspect is the human perception of the results, and subjective feelings induced by the generated outcomes are paramount.



	Run	Smoothed	Value	Step	Time	Relative
●	20210427-231051\train	0.02103	0.02103	0	4/27/21, 11:29 PM	0
●	20210427-233316\train	0.01638	0.01484	2	4/28/21, 12:05 AM	18.12 min
●	20210428-000845\train	0.01859	0.01857	27	4/28/21, 6:27 AM	6.072 hr
●	20210428-065437\train	0	0	7	4/28/21, 7:05 AM	7.26 min
●	20210428-070757\train	0	0	10	4/28/21, 7:21 AM	10.47 min
●	20210428-072734\train	0	0	0	4/28/21, 7:30 AM	0
●	20210428-074720\train	0.02095	0.02109	103	4/28/21, 10:13 PM	14.25 hr
●	20210428-231158\train	0.02943	0.02299	1	4/28/21, 11:25 PM	3.22 min
●	20210428-234340\train	0.02741	0.02193	2	4/28/21, 11:59 PM	6.189 min
●	20210429-000114\train	0.01928	0.01945	82	4/29/21, 7:49 AM	7.62 hr
●	20210429-080843\train	0.01495	0.01475	52	4/29/21, 11:35 AM	3.281 hr
●	20210429-113829\train	0.01459	0.0145	26	4/29/21, 1:09 PM	1.343 hr
●	20210429-131309\train	0.01422	0.01408	8	4/29/21, 1:46 PM	26.34 min
●	20210429-134727\train	0	0	0	4/29/21, 1:51 PM	0
●	20210429-135449\train	0.01327	0.0133	54	4/29/21, 5:40 PM	3.592 hr
●	20210429-174843\train	0.0381	0.01764	1	4/29/21, 6:05 PM	2.995 min
●	20210430-014109\train	0.01517	0.01517	0	4/30/21, 1:51 AM	0

Figure 10. Screenshot from the Tensorboard application showing part of the logs of the training process for various network variants. Source: author's own work.

2.3 Implementation the Unreal Engine 4

The following tools were utilized:

- Unreal Engine 4.27.2,
- CPPFlow 2 library,

- Tensorflow libraries for the C language in version 2.7.0.

A scene was prepared using the Unreal Engine editor, consisting of a graphical representation of the obstacle and a graphical interface through which the user can modify simulation parameters. The code responsible for handling the neural network is encapsulated in a class named ATensor-FlowNetwork, with an object representing the water surface instantiated during the constructor's execution.

The demonstration allows visualization of the effects of both previously described versions of the network: a simple one, generating only the water surface height map, and a complex one, generating both the water surface height map and the foam density map. Blueprints were employed to initialize the neural network, link interface value changes to class field value changes, and establish the execution of the function generating a new set of data through the network thirty times per second, in accordance with the adopted assumptions. The results of the network's operation are saved as textures, which are then utilized by the water surface Shader. The modification of vertex positions in the geometry occurs during the Vertex Shader operation. The material graph representing the material is shown in Figure 11.

The use of the CPPFlow 2 library necessitated a change in the language standard to C++ 17. This was achieved by modifying the project's configuration script.

In conclusion, the preparation of the plugin required writing over 370 lines of C++ code, all of which are available in the repository shared on the GitHub platform at: https://github.com/p4vv37/ueflow

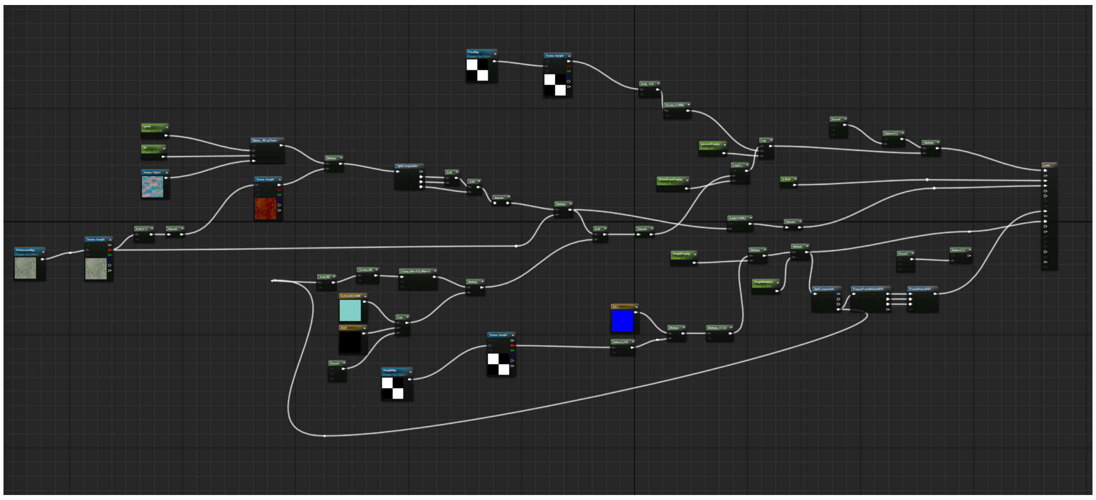


Figure 11. Material graph representing the water surface material. Source: author's own work.

2.4 Presentation and Analysis of Results

2.4.1 Presentation of Results

The interactive demonstration of the neural network's operation is capable of generating a water surface responsive to changes in parameters such as obstacle speed or direction. The essential elements of the demonstration interface are depicted in Fig. 12. Elements for controlling simulation parameters are visible:

- obstacle speed (a),
- obstacle movement direction represented by the angle of deviation from the X-axis (b),
- display mode selection (c). Five display modes have been implemented and presented in Fig. 14:
  - final result – view of the water surface considering all its parameters and characterized by a complex material,
  - water height – representation of water height in the form of color applied to a flat surface,
  - foam – representation of foam density in the form of color applied to a flat surface,
  - values of the cosine function of twice the angle between the vector, whose origin was the center of the obstacle, and the considered point, and the vector indicating the bow's direction of the boat (e),
  - values of the cosine function of twice the angle between the vector, whose origin was the center of the obstacle, and the considered point, and the vector indicating the bow's direction of the boat (e),
- selection between complex and simple network variants (d),
- obstacle shape (e).

Examples of generated water surfaces and the influence of simulation parameters on their shape using both a simple and a complex simulation variant are presented in Fig. 13 and Fig. 14.

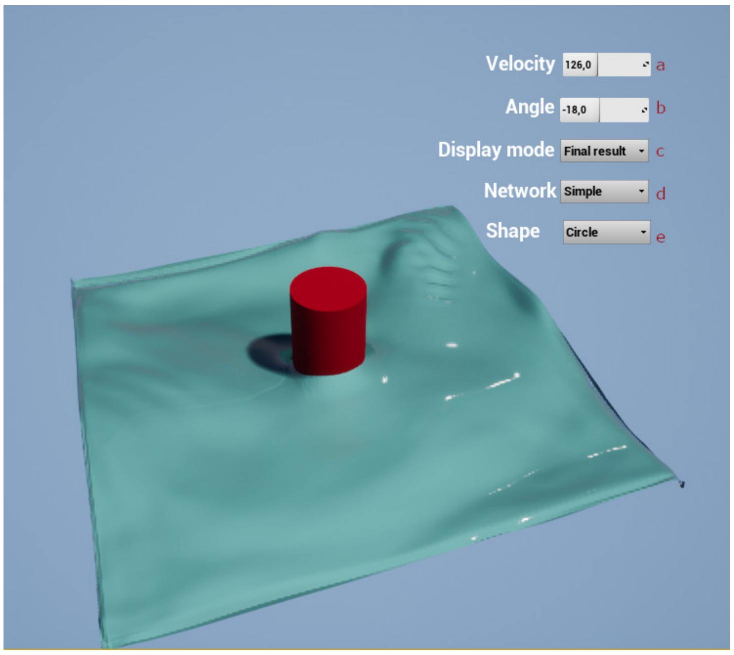
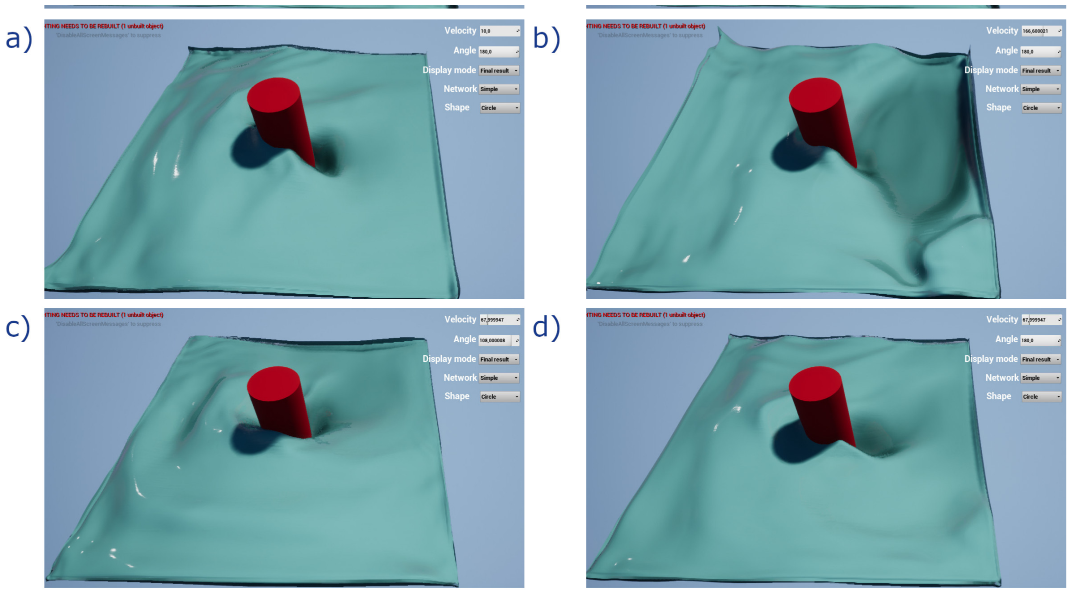
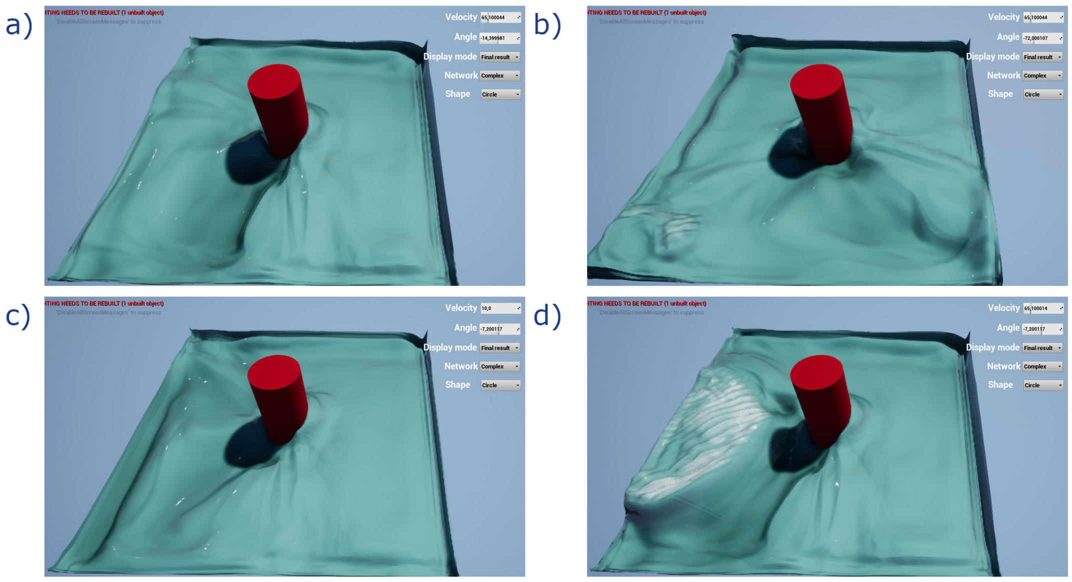

Velocity 126,0 a
Angle -18,0 b
Display mode Final result c
Network Simple d
Shape Circle e

Figure 12. Interactive demonstration screen. Source: author's own work.



a) Velocity 10,0; Angle 180,0; Display mode Final result; Network Simple; Shape Circle
b) Velocity 166,600021; Angle 180,0; Display mode Final result; Network Simple; Shape Circle
c) Velocity 67,999947; Angle 108,000008; Display mode Final result; Network Simple; Shape Circle
d) Velocity 67,999947; Angle 180,0; Display mode Final result; Network Simple; Shape Circle

Figure 13. Impact of various simulation parameter values on the appearance of the water surface generated using a simple neural network: images a and b show the effect of changing the obstacle speed while maintaining the direction, images c and d show the effect of changing the obstacle direction while maintaining its speed. Source: author's own work.


a)
b)
c)
d)
TING NEEDS TO BE REBUILT (1 unbuilt object)
'DisableAllScreenMessages' to suppress

a) Velocity 65,100044; Angle -14,399981; Display mode Final result; Network Complex; Shape Circle

b) Velocity 65,100044; Angle -72,000107; Display mode Final result; Network Complex; Shape Circle

c) Velocity 10,0; Angle -7,200117; Display mode Final result; Network Complex; Shape Circle

d) Velocity 65,100014; Angle -7,200117; Display mode Final result; Network Complex; Shape Circle

Figure 14. Impact of various simulation parameter values on the appearance of the water surface generated using a complex neural network: images a and b show the effect of changing the obstacle direction while maintaining its speed, images c and d show the effect of changing the obstacle speed while maintaining the direction. Source: author's own work.

It was tested whether the network could generalize results for obstacle shapes other than the circular one it was trained on. Simulations were conducted for a square-shaped obstacle, and the results were compared with those obtained for a circular obstacle. The analysis of results leads to the conclusion that the network cannot correctly generalize results for a square-shaped obstacle. The results for this obstacle contain more errors and areas with shapes deviating from expectations than the results for a circular obstacle. However, it is worth noting that the obtained results partially reflect the new shape of the obstacle cross-section. The comparison of results is presented in Fig. 15.

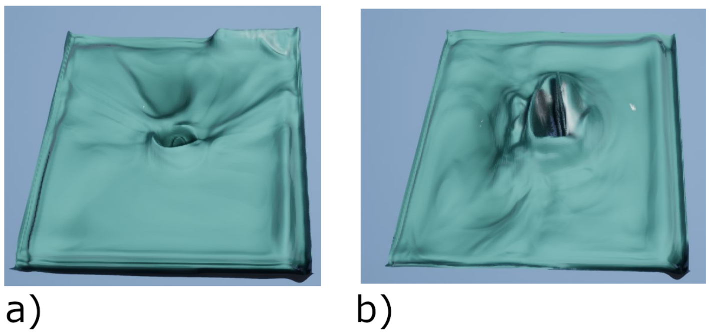
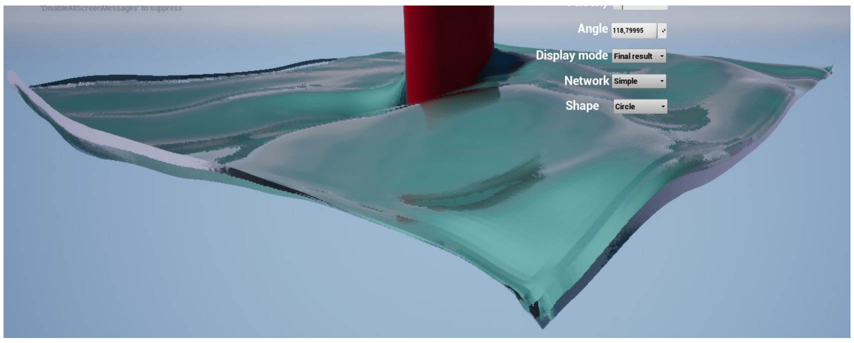


Figure 15. Comparison of simulation results for an obstacle with a circular cross-section (a), on which simulations used during its training were performed, and an obstacle with a square cross-section (b). The network partially takes into account the new cross-sectional shape, but a noticeable deterioration in the quality of the result is evident. Source: author's own work.

2.4.2 Qualitative Analysis of Results

Since the most important aspect of the project is the human perception of the result, there is no specific and obvious way to objectively assess the results. For example, an unnaturally smooth surface would achieve a higher score in the project than a complex and turbulent one, visually closer to the reference. Another challenge in such an evaluation would be the dynamic and occasionally turbulent nature of the simulation. Due to the inability to provide objective values representing the quality of the obtained results, it was decided to leave this matter to the subjective assessment of the recipient. However, it is essential to describe and analyze in detail the areas where clear errors in its operation are visible. These will be listed later in this chapter.

Regardless of the selected simulation parameters, artifacts are visible on the edges of the generated height map. This effect is presented in Fig. 16. This error can be easily concealed by reducing the displayed surface area.


'DisableAllScreenMessages' to suppress

Angle 118,79995
Display mode Final result
Network Simple
Shape Circle

Figure 16. Zoom-in on the edges of the simulated surface, showing artifacts in this area. Source: author's own work.

Exceeding the parameter values used during the generation of the training database results in noticeable errors. These errors are particularly noticeable in areas in front of and behind the obstacle in its direction of movement, while in areas parallel to the direction of movement, the generated surface looks correct. Examples of this effect for both simple and complex network variants are visible in Fig. 17 and Fig. 18. The presence of this effect indicates the network's low ability to generalize.

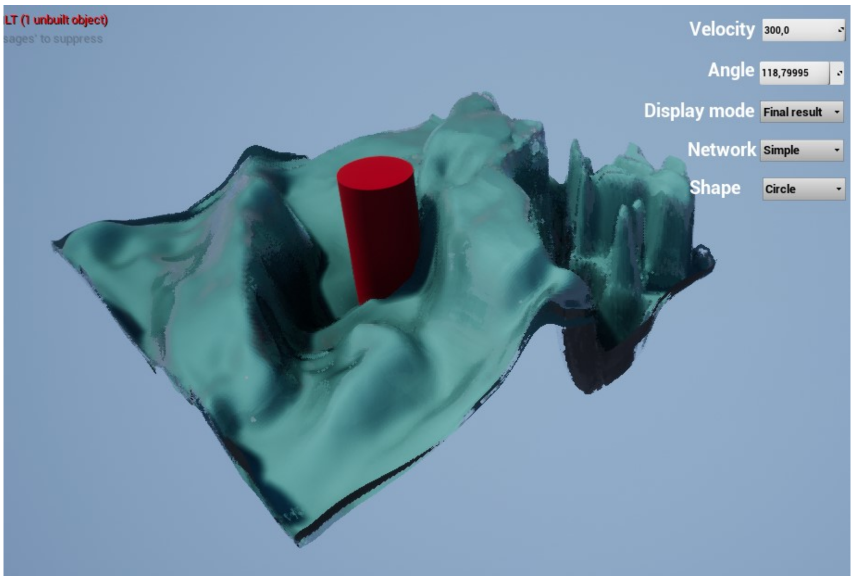

Velocity 300,0
Angle 118,79995
Display mode Final result
Network Simple
Shape Circle

Figure 17. Example of artifacts generated by the simple network variant after a significant increase in obstacle speed values beyond the range used during network training. Source: author's own work.

The simulation generated by the complex network variant quickly stabilizes, turning into a static surface resembling a single frame of the simulation. This effect does not occur in the case of the simple network.

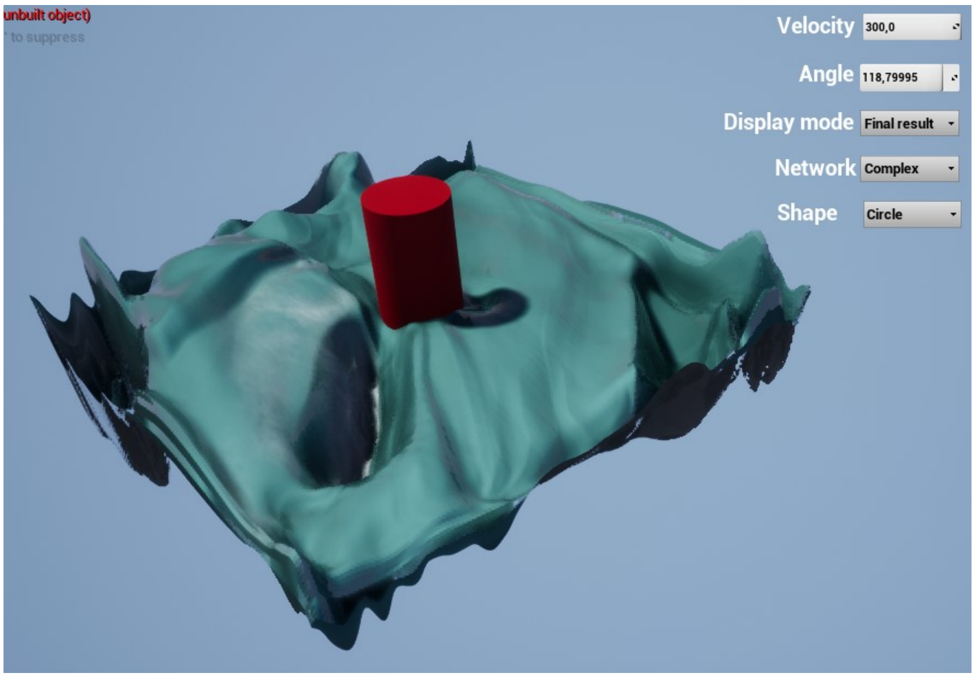
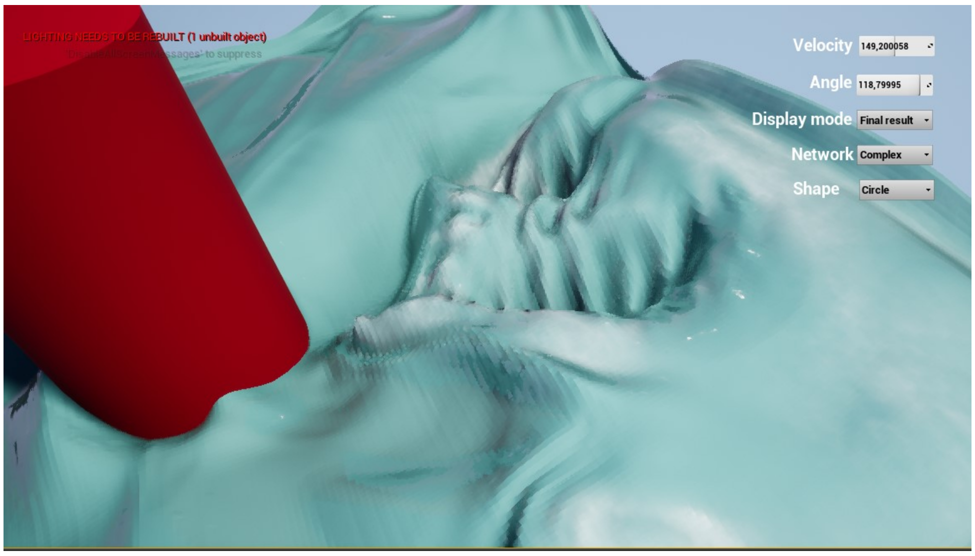

Velocity 300,0
Angle 118,79995
Display mode Final result
Network Complex
Shape Circle

Figure 18. Example of artifacts generated by the complex network variant after a significant increase in obstacle speed values beyond the range used during network training. Source: author's own work.

The areas of the network covered with dense foam often exhibit numerous dynamically changing details with a high frequency that were not observed in the data used in the training process. This effect is shown in Fig. 19.


LIGHTING NEEDS TO BE REBUILT (1 unbuilt object)
'DisableAllScreenMessages' to suppress

Velocity 149,200058
Angle 118,79995
Display mode Final result
Network Complex
Shape Circle

Figure 19. Example of artifacts occurring in areas of the surface covered with dense foam. Source: author's own work.

2.4.3 Performance Analysis of Results

To assess performance, the tool nvidia-smi was used, and the time taken for the network to make predictions during interactive demonstration was measured. Measurements were taken using two workstations. In both cases, the neural network used a graphics accelerator. The stations were equipped with graphics cards with the following parameters [8][8]:

- NVIDIA GeForce RTX 3080,
    - Ampere architecture,
    - 10 GB GDDR6X memory,
    - memory bus width: 320 bits,
    - memory bandwidth: 760.3 GB/s,
    - 8 nm manufacturing process,
    - number of CUDA Cores: 8704,
    - number of Tensor Cores: 272,
    - 28.3 billion transistors,
    - clock speed: 1440 MHz,
    - boost clock speed: 1440 MHz,
    - memory clock speed: 1188 MHz,
    - TDP: 320 W,
    - manufacturer-specified maximum floating-point operations per second: 29.77.

- NVIDIA GeForce RTX 2060 Mobile,
  - Turing architecture,
  - 6 GB GDDR6 memory,
  - memory bus width: 192 bits,
  - memory bandwidth: 336 GB/s,
  - 12 nm manufacturing process,
  - 10.8 billion transistors,
  - number of CUDA Cores: 1920,
  - number of Tensor Cores: 240,
  - clock speed: 960 MHz,
  - boost clock speed: 1200 MHz,
  - memory clock speed: 1188 MHz,
  - maximum measured power consumption during network operation: 91 W,
  - manufacturer-specified maximum floating-point operations per second: 9.216.

It should be noted that determining the performance difference between the cards based solely on comparing their parameters is not possible because there are significant differences between the Turing and Ampere architectures. One of the main reasons for this could be the manufacturer-declared twice the throughput of Tensor Cores in cards based on the newer architecture.

The results of the measurements are included in Table 1. Achieving the assumed thirty refreshes per second requires performing calculations in less than 33.30 ms. The highest recorded value was 29 ms. This means that both tested workstations were able to perform calculations fast enough to achieve the assumed thirty refreshes per second.

The graphics cards used significantly differ in parameters, which is noticeable in a significant difference in the evaluation time of the network on the tested workstations. In both cases, the cards had a significant amount of free memory. In the case of the 3080 card, the declared TDP value is known. Comparing it with the values measured during operation, it can be observed that it was exceeded. This suggests that the computational power of the cards is the limiting factor, not the memory or bandwidth of any of the elements.

Table 1. Zmierzone dane opisujące wydajność działania sieci na kartach GeForce RTX 3080 oraz GeForce RTX 2060 Mobile dla dwóch wariantów symulacji.

Wariant	Karta	Czas obliczeń [ms]	Użycie GPU[%]	Użycie pamięci[%]	Taktowanie GPU[MHz]	Moc[W]
Prosty	3080	7,46	81.64	26.53	1923	334
Prosty	2060 Mobile	28,00	90.30	41.65	1680	89
Złożony	3080	4.85	76.52	19.29	1918	327
Złożony	2060 Mobile	12.07	88.39	24.44	1727,5	87

The graph shown in Fig. 20 presents the relationship between the evaluation time of the network and the number of simulations performed simultaneously. The increase in the number of simultaneous simulations was achieved by expanding the input data. The data was collected only using a workstation equipped with a GeForce RTX 3080 card. Anomalies visible in the graph, consisting of a temporary increase in the evaluation time, probably result from a temporary decrease in the clock speed of the graphics card due to its excessive load.

There is a clearly linear relationship between these values. The linear approximation of the values obtained for the network variant responsible for complex simulation is determined by the formula 1, and the linear approximation of the values obtained for the network variant responsible for simple simulation is determined by the formula 2.

$$\mathrm{y} = 3.66 * 10^{-2}\mathrm{x} + 3.26 * 10^{-3} \tag{1}$$

$$\mathrm{y} = 5.23 * 10^{-3}\mathrm{x} + 2.26 * 10^{-2} \tag{2}$$

This means that increasing the number of simultaneously performed simulations from one to two increases the evaluation time by 19

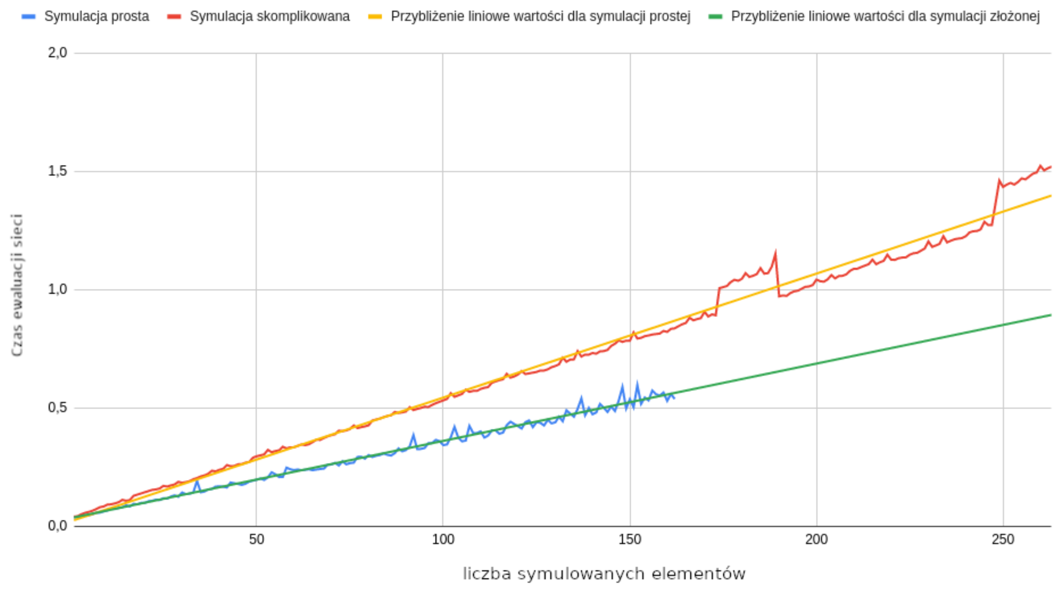


Figure 20. Presentation of the relationship between the evaluation time of the network and the number of simulations performed simultaneously. Source: author's own work.

Chapter 3

Conclusion

3.1 Results Presentation

This master's thesis proposes a solution for simulating water surfaces around obstacles using deep neural networks and the Unreal Engine 4. The project successfully achieved its main goal, which was to create an application with an interactive water surface simulation, and the research goal of employing neural networks for simulation. The application serves as a hidden goal, enabling a comparison of the proposed solution with existing ones.

The initial sections of the thesis provide background knowledge on water surface simulation, including historical context and contemporary solutions. Neural network principles are explained, outlining their current applications in fluid simulation and other domains. Game engines, particularly Unreal Engine 4, and their integration with neural networks are discussed.

The subsequent sections detail the steps taken to achieve the thesis goals. This includes preparing a reference simulation for training the neural network, integrating the Tensorflow library with Unreal Engine 4 using the cppflow library, and creating and training the neural network. The interactive application, developed as the hidden goal, is described in detail. The source code for the application is available at https://github.com/p4vv37/ueflow.

The thesis also presents the results and analysis of the achieved outcomes. Water surface simulation results using the neural network are compared with results obtained through other simulation methods. The data indicates that the neural network implementation, executed on a graphics card, provides sufficient performance for practical applications. The network evaluation time, ranging from 4.85 ms to 7.45 ms, is significantly lower than the frame evaluation time at a 60 frames-per-second refresh rate (16.66 ms). This allows for a higher refresh rate than the initially targeted 30 frames per second.

Based on the results, the proposed solution allows for high-quality water surface simulation while maintaining adequate performance. The application developed as a hidden goal facilitates interactive simulation exploration and result comparison with other water surface simulation methods.

Bibliography

[1] Hennigh, O., *Computational-fluid-dynamics-machine-learning-examples*, https://github.com/loliverhennigh/Computational-Fluid-Dynamics-Machine-Learning-Examples data dostępu: 14.12.2021.

[2] Kowalski, P., *Zmiany dokonane w module pyradox na potrzeby wykonania pracy.* https://github.com/Ritvik19/pyradox/commit/88eafb5dcd44c332f92c4f6d6da756016a2d6518 data dostępu: 13.09.2023.

[3] Rao, C., *Pinn-laminar-flow*, https://github.com/Raocp/PINN-laminar-flow data dostępu: 14.12.2021.

[4] Rastogi, R., *Repozytorium modułu pyradox zawierającego implementacje architektur sieci neuronowych.* https://github.com/Ritvik19/pyradox data dostępu: 13.09.2023.

[5] Ronneberger, O., Fischer, P., and Brox, T., „U-net: Convolutional networks for biomedical image segmentation,” *arXiv:1505.04597*, 2015.

[6] SideFX, *Opis narzędzia flat tank*, https://www.sidefx.com/docs/houdini/shelf/flattank.html data dostępu: 13.09.2023.

[7] TensorFlow, *Dokumentacja tenssorflow: Prognozowanie szeregów czasowych*, https://www.tensorflow.org/tutorials/structured_data/time_series data dostępu: 13.09.2023.

[8] www.techpowerup.com, *Nvidia geforce rtx 3080.* https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621 data dostępu: 13.09.2023.