Comp4900L - Final Project Report

Stormy Sea

Group Members:

Jason Xu 101220729

David Simonov 101232041

Video Link: https://youtu.be/VP4QUTvhGwM

[A] Introduction. Explain what you were trying to do and why it is a worthwhile undertaking -- in short, why anyone would care about your topic. (I know that you care about it yourself -- explain what would be interesting to someone outside your group.) Describe your goals in enough detail that a reader would be able to independently assess whether or not you achieved them. Give a high-level description of your approach and preview your results.

This project document details the design, development, and assessment of the real-time visualization of a stormy sea using openFrameworks. This project aimed at developing a static, high-quality visualization of a dark storm-lit ocean illuminated by thick, heavy storm clouds. The concepts include procedural shading, Gerstner wave displacement, and aggregated geometry.

Realistic simulation of water and cloud effects has been a topic of interest for the computer graphics community for several years. Water simulation needs view-dependent reflectance, wave patterns, foam simulation, and atmospheric effects, and the simulation of storm clouds needs volumetric lighting effects and realistic silhouettes. This project will work on providing a feasible, GPU-based approach that is capable of generating realistic images without the need for off-line rendering and volumetric ray marching.

The objective of the project was:

1. Create the ocean surface mesh using the sum of Gerstner waves with vertex displacement.
2. Provide the following environment effects: implement storm-appropriate lighting, including controlled specular highlights, Fresnel reflectance, and distance fog.
3. To generate thick, billowed, and cloudy clouds that do not have visible primitive boundaries, use procedural noise.

The final implementation produces a still image with dark, turbulent ocean waves illuminated by the light of the distant light source, muted water in the distance, fog, and cloud masses rendered using layered primitives. The project will contain a src code (ofApp.cpp, ofApp.h, main.cpp), shaders (cloud.vert, cloud.frag, ocean.frag, ocean.vert) , youtube video, and a README.txt.

[B] Previous Work / Background. Describe relevant related work, especially work that you directly built on. Do not assume that the reader has read any of the papers; your target audience should be a senior computer science student who is in the first month of this class. If your work relies heavily on a single paper, describe that paper in some detail. Do not cite only using URLs, but give author names, paper title, forum, and year of publication. (If a web-only source, do include the URL, but do your best to extract the other information and include it as well.)

Waves

This approach immediately leverages the existing knowledge of these two mutually supportive traditions of modeling the sea surface. The first of these is the classic Gerstner wave model, originally formulated in the 19th century by Franz J. von Gerstner, Gerstner, F. J. (1802). Theorie der Wellen. *Annalen der Physik*, who developed the analytic trochoidal wave solution for the direct displacement of vertex geometry. The other is the current literature on ocean rendering, that demonstrates how several components of the spectrum can be combined for the realistic modeling of ocean swell and chop, the canonical reference for which is Tessendorf, J. This implementation includes the physically motivated version that is better suited for cloudy weather.

Lighting, textures and noise

Our implementation of small-scale roughness and foam procedural textures is well within current procedural-texture practice. Procedural textures using the noise and fractal Brownian motion processes, developed by Ken Perlin for the paper “An Image Synthesizer” (SIGGRAPH, 1985), are the prevailing method of achieving natural, high variation on a surface, the fractal Brownian motion itself being the standard procedural-texture technique for modeling the spray of foam and chop. For the shading and reflectance modeling (grazing angle reflectance and specular components), it was simply the paper “An Inexpensive BRDF Model for Physically-Based Rendering” by Christophe Schlick, published in 1994, that provided the

Schlick approximation for Fresnel components. For physically-based ocean rendering, the angle-dependent reflectance model used is the Fresnel equation. However, for real-time rendering, Schlick's approximation is employed. This project uses the Blinn-Phong model with a Fresnel function that prevents the scene from looking bright when it is rendered at a farther distance.

Comp4900L - Mesh Modelling and Rendering slides/lecture + shader/lighting demo code

For the generation and formation of meshes, especially when creating the cloud meshes from the combination of multiple 'ofSpherePrimitives', the mesh rendering lecture explained the concept of depth, z-buffering, order of drawing, lighting, and shading each object which is implemented in this project. The shaderdemo code was referenced when creating our own shaders for the clouds and ocean.

Key References:

Perlin, K. (1985). An image synthesizer. Computer Graphics. https://s.dou.ua/storage-files/Perlin_noise_original_article.pdf

Prof. David Mould, "Intro to real-time rendering and mesh modeling", Course Lecture; https://brightspace.carleton.ca/d2l/le/content/369342/viewContent/4449595/View

Prof. David Mould, "lighting-shader", Course Demo Code: https://brightspace.carleton.ca/d2l/le/content/369342/viewContent/4468284/View

Schlick, C. (1994). An inexpensive BRDF model for physically-based rendering. Computer Graphics Forum.

Tessendorf, J. (2001). Simulating ocean water. SIGGRAPH 2001 Course Notes. ACM SIGGRAPH. https://people.computing.clemson.edu/~jtessen/reports/papers_files/coursenotes2002.pdf

GLSL Versioning and Compatibility Issues

Compatibility problems are very common on integrated GPUs, for example, on Intel HD/UHD. This project tries to use OF_GLSL_SHADER_HEADER and shader syntax but older systems will run into issues.

Linux, and older systems, or lenovo's with linux sub-systems can potentially have

misplaced or lack correct versions of OpenGL for 3d rendering leading to shaders lacking functionality. The best solution is making sure drivers are up to date and that your device is using the correct gpu for rendering 3d images. In rare cases with really out of date systems, shaders will not be functional for your 3d images, however, there is a wireframe fallback that can be enabled/set to true in the ofApp.cpp void setup called “Enable_WF” to take effect and display/draw without the inactive shaders.

[C] Implementation. Say what you did. This section is highly flexible depending on exactly what your project looks like. Aim for precision of explanation; often, introducing symbols and then using equations and pseudocode will be the most precise way of communicating your meaning. Ideally (an ideal rarely actually achieved in practice, but directionally correct) a motivated reader should be able to reimplement your project from the description you give, plus adequate background knowledge. List third-party assets, libraries, and software that you made use of in creating your project.

C.1 High-level architecture

System: openFrameworks 0.12.1, developed on Windows with Visual Studio 2022.
Files involved: Key files involved were main.cpp, ofApp.h, ofApp.cpp, and shader files stored in data: ocean.vert, ocean.frag, cloud.vert, cloud.frag.
Rendering loop: OF standard setup(), update(), and draw(). It features ofEasyCam (camera) and rendering of the ocean mesh and cloud geometry/shaders within cam.begin() and cam.end().
Shader management: Shaders are loaded through ofShader::load(), and errors are checked. In-depth console output lists shader uniforms and attributes for alignment-related errors.

Header Functionality Consisted of:

- Cloud structure, consisting of the cloudMesh, position, and overall size, allowing for easy multi-cloud generation along with the vector list of clouds.
- Ocean Mesh along with the Gerstner wave parameters including;
    > Amplitudes.
    > Wavelengths.
    > Speeds.
    > Directions.
- The grid resolution and size (which the ocean and cloud generation follow).

- The camera, which allows for you to move around the scene and watch it from different angles and positions.
- Ocean and Cloud shaders.
- Ocean and cloud colors, for consistency upon shading.
- The lighting values such as position, color, and fog.

Setup functionality:

- Sets the grid values of N and L representing resolution and size.
- Ocean values such as amplitudes, wavelengths, speeds, and directions being set appropriately to influence generation.
- Setting oceanMesh and each cloudMesh to “OF_PRIMITIVE_TRIANGLES” and enabling normals for each.
- Setting the camera up for the user.
- Shader loading.

C.2 Ocean mesh generation

The ocean mesh starts with a regular rectangular grid defined on the CPU, where the scalars are the number of grid cells, defined by the variable N, and the size of the ocean, defined by the variable L, with the following code: N=2048, and L=2000.0f. Vertices defined by the indices i=z*N+x, with u=x/(N-1) and v=z/(N−1) for the 2D texture coordinate. Two triangles, thus 6 indices, are defined for each cell (x,z), meaning that the following code is used:

i, i+1, i+N and i+1, i+N+. In pseudocode:
for z in range(0, N)
for x in range(0, N)
pos.x = (x − N/2)*L/N
pos.y = 0
pos.z = (z − N/2)*L/N
addVertex(pos); addTexCoord(x/(N−1), z/(N−1))
after vertices: build the indices as above.

The code calls oceanMesh.enableNormals(), which causes the mesh to store vertex normals and the code applies the displacement to each vertex once, calling gerstnerWave(.) for each vertex and storing the result with oceanMesh.setVertex(i,vertices[i]). For the final

technique, normals were computed using vertex shader analytical normals when the displacement was GPU-based, and mesh.enableNormals(), along with ofMesh helpers, when CPU-based.

Moving on is the integration of the wave math and vertex displacement. For vertex height, it is the sum of several Gerstner terms along with the chop and the modulation. To express the jth component, it will need the amplitude a_j, wavelength λ_j, wave number k_j = 2π/λ_j, speed c_j, and unit direction vector d_j. To express the contribution for a vertex at horizontal position p = (x,z) at time t, it is given by:

y_g = Σ_j (a_j' * sin(k_j * dot(d_j', p) − c_j * t + φ_j))

where a_j' and d_j' add a slight spatial randomness using ofNoise(.) (the code multiplies the amplitude by (0.8 + 0.4 * noise) and varies the direction by a small angle offset of the noise). A high-frequency chop adds:

y_chop = A_chop(p,t) * sin(k_chop * (x + z) − ω_chop * t)

with `A_chop` and `k_chop` varied by local noise. Finally, the overall vertex height is varied by large-scale height modulation `heightMod(p) = 0.8 + 0.4 * noise(p * 0.005)`. The C++ code for the `gerstnerWave` function, given below, adheres to the above formulation: Iterate through the specified amplitude/wavelength/speed vectors, calculate per-component `k`, phase, dot product, and sum `y`; add `chop` and scale by `heightMod`.

C.3 Ocean shader design

In general shaders take vec3 positions and calculate the per-vertex displacement using the math of Gerstner. It calculates a displaced position and sends the transformed normal and also the world position to the fragment shader. The ocean.frag shader contains code for the local lighting technique, along with procedural foam and fog. The essential variables used for the shader are explained below. They are all used in the fragment shader. They are:

N: This variable holds the surface normal. V: (view vector = −fragPosition normalized).L: (light direction = normalize(lightPosition − fragPosition)), and H (half-vector = normalize(L+V)). H: (half-vector = normalize(L+V)).

The following are a list of lighting terms for the Blinn-Phong lighting model::

ambient = 0.15 * lightColor * distanceFactor

diffuse: max(dot(N, L),0) * lightColor *

specular = k_spec * pow(max(dot(N,H),0), shininess) * lightColor * distanceFactor where shineiness is 150 and k_spec ~ 1.2.

The Fresnel reflectance is approximated using F = mix(0.01, 0.25, pow(1 - max(dot(N,V),0),4)) * distanceFactor, which somewhat enhances grazing reflectivity.

Water color is varied according to depth modulation ranging from deep to shallow with depthFactor clamping the range of (waveHeight+25)/50 between 0 and 1, and waterColor = mix(waterColorDeep, waterColorShallow, depthFactor).

Foam is produced using two features: crest threshold and surface steepness. For waveHeight > 18.0, crestFoam is defined as (waveHeight-18.0)/20.0, along with foamNoise fbm(fragPosition.xz * 0.1 + time * 0.1), and foamAmount is crestFoam * foamNoise * 0.8. For steepness determined using shader derivatives steepness = 0.5 * length(vec2(dFdx(waveHeight), dFdy(waveHeight))), when steepness > 0.35, foamAmount is max(foamAmount, (steepness-0.35)1.5). Lastly, foamAmount is clamped between 0 and 0.7, then combine with litColor = waterColor * (ambient + diffuse) + specular and computed with finalColor = mix(litColor, foamColor, foamAmount).

C.4 Cloud system

Created a cloud structure that holds the mesh, position, and size of the cloud.
The list of clouds is stored in “clouds” which allows quick and easy management of multiple, especially when drawing and applying the cloudShader to them all.
The construction of the Mesh from the cloud structure was done with the build_Cloud() function, which sets the appropriate values and creates a list of cloud layers. These layers represent the multiple ofSpherePrimitive shapes used to form the cloud itself, with the first layer being the centre.

- The amount of layers for each cloud mesh is chosen between 12-20 spheres for optimization.
- Each layer is given a random size, rotation, position, and offset with respect to the centre, staying close enough to be a part of the same cloud forming that puffy cloud effect rather than a single sphere.
- These layers (multiple ofSpherePrimitives) are then traversed and get their vertices, normals, and indices added to the cloud mesh with their appropriate offsets by using “addVertex()”, “addNormal()”, and “addIndex()”, forming a wireframe that we later fill in by our cloud_light shaders.
- The grid/range of the clouds being generated follows along the same grid size and range of the ocean itself, so that we don’t have any random outliers. The “ofRandom()” function was used to fill up the sky appropriately, and “ofSeedRandom()” was used to propagate

true random cloud filling by setting a unique seed (which is currently set to ‘0’ for consistency).

In the setup, a random seed is chosen so that the position, size, count, and other various attributes of the sky filled with clouds is random each time. Also placed modifiers in the setup that can modify this and be changed to any amount or range for different results. For consistency, the base random seed chosen is ‘0’.

With these modifiers, you can change the level of detail in the clouds by increasing or decreasing the size range of layers/clouds, the amount of layers, and the amount of layers in general.

For greater detail, you can have smaller clouds, but a higher count of layers to fill it with, resulting in more realistic clouds. However, to generate this many ofSpherePrimitives can cause performance issues on low-end devices. To counteract this, each cloud shares the same fragment shader, and avoids filling in each layer itself, and instead, applying the shader to the mesh itself.

C.5 Cloud Shader (cloud.frag)

Makes use of the coarse 3D noise/FBM sampled for each fragment position to calculate turbulence.

Smooth threshold function (smoothstep) is used for converting turbulence intensity into the occupancy/density map. The fragment color varies between very dark core shadow and medium storm gray, depending on turbulence.

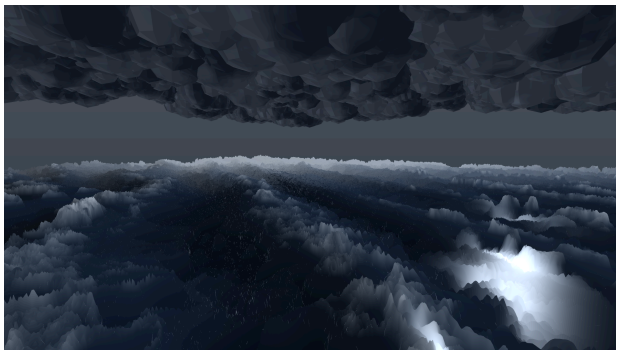
Lighting is subtle, with a rim highlight and a very small directional light contribution. This will prevent the clouds from shining.

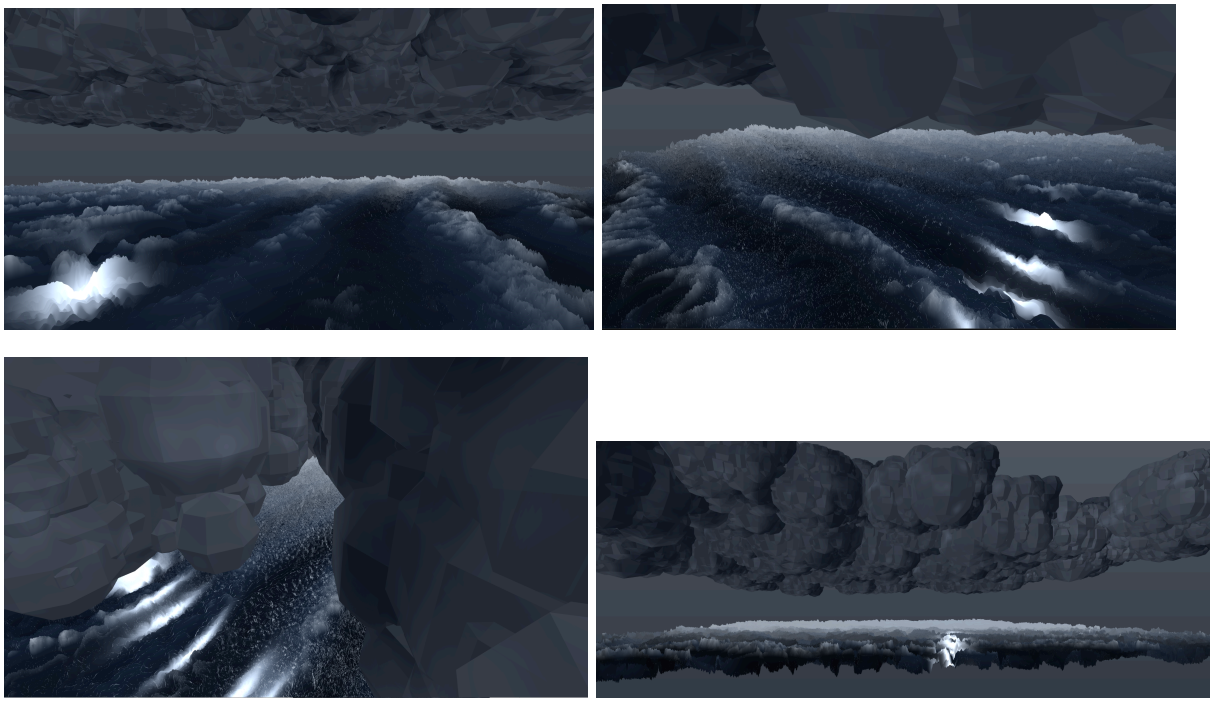
Texture: clouds are opaque (0.88-0.98 alpha) with a slight smoothing near the edges. Edge smoothing applies a smoothstep that is dependent on the density and clamped.

This shader calculates the approximate normal for lighting using dFdx/dFdy. C.5 GLSL & OpenGL compatability work OF macro use: Added OF_GLSL_SHADER_HEADER for support among various GLSL versions on different systems. Attribute bindings: Verified that the mesh supplies the mandatory attributes. If texcoord was not used, it was removed from the shader, otherwise the presence of mesh.addTexCoord() was verified. Shader Error Messages: The

application will print shader compile and log messages, along with active attributes and uniforms.

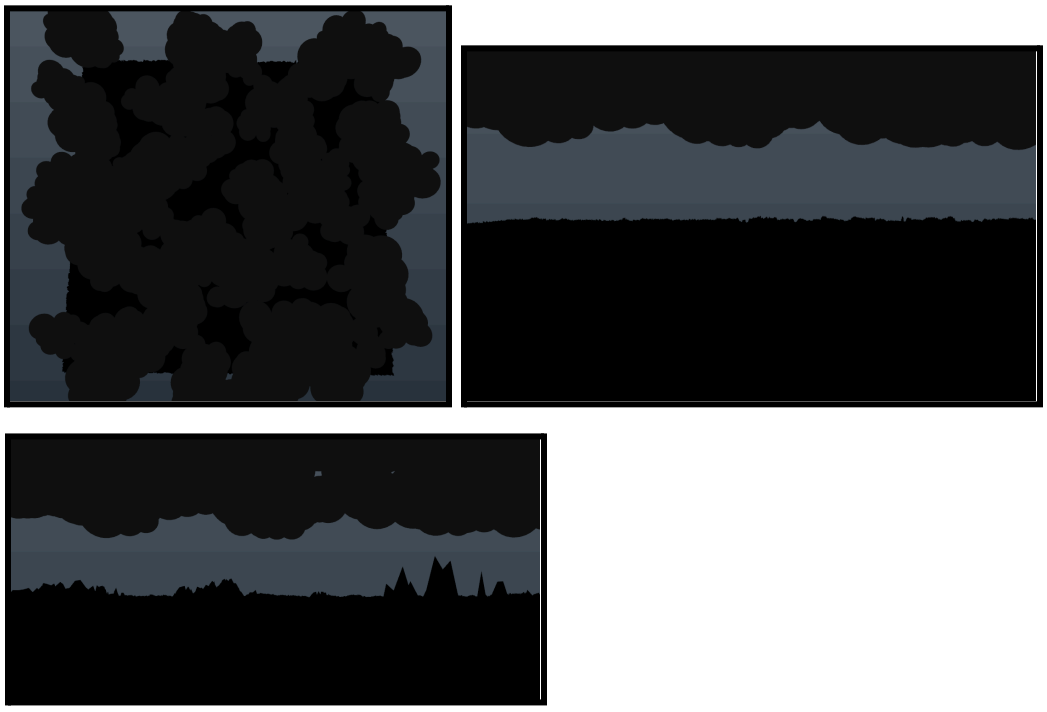
[D] Results and Discussion. Show output from your approach and assess it. Be specific; you may want to call attention to details of interest in particular images. Compare with other approaches, if applicable. Are there classes of output that were in your original target that you cannot do well? Characterize the cases where your method works especially well or especially poorly. Give performance figures -- "running on a machine with X hardware, my implementation required Y minutes to generate an image with Z resolution."
Probably this section will contain many images. I suggest showing 12 to 20 images on a page, perhaps arranged into a 3x4 or 4x5 grid. Readers can zoom in to see details. Comment on and critique the images you show, at least generally (you probably lack space to discuss each image individually).



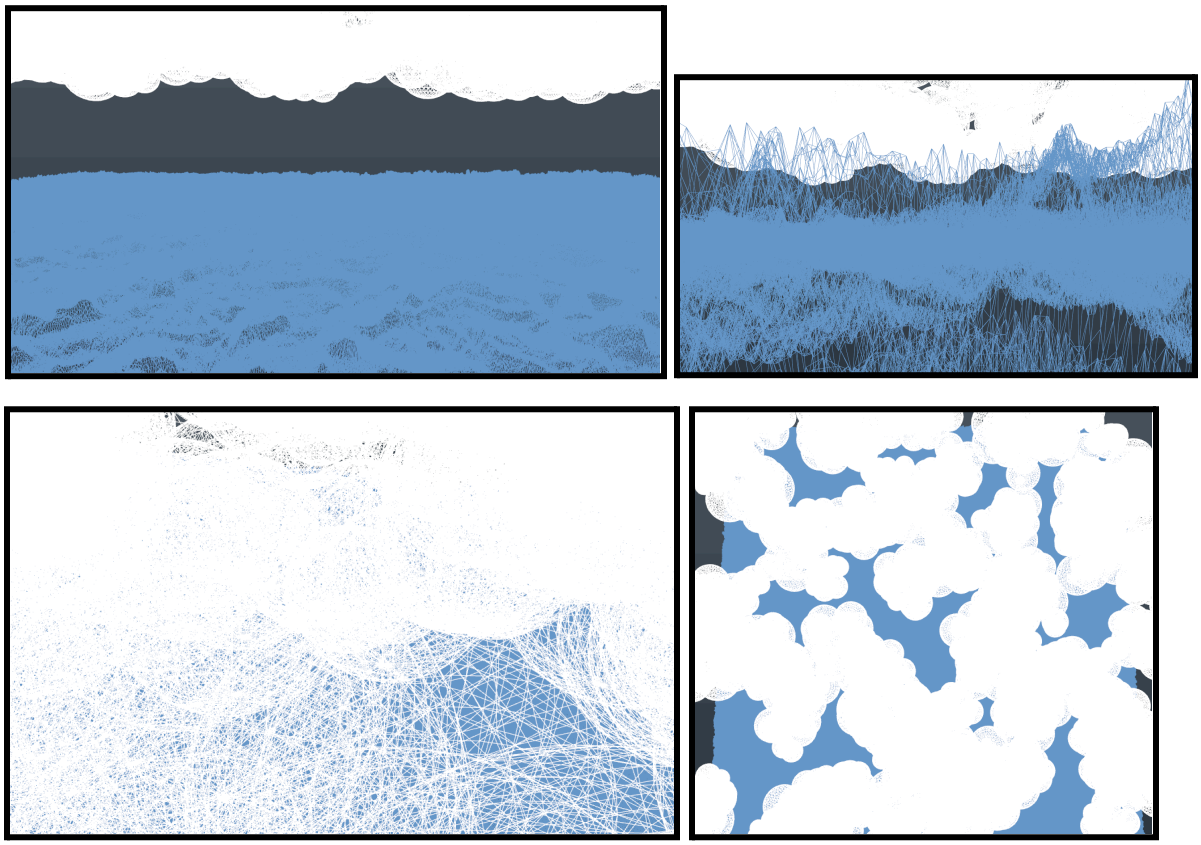



**Note more angles of the image can be found in the video.*

Failed/Missing OpenGL Result:



Wireframe Fallback (Enabled by shaders not loading correctly or setting Enable_WF to true):



The rendered image demonstrates a very stylized scene of a storm, rendered using a CPU-displaced Gerstner-wave ocean and sphere clouds: the code underlying the project, in ofApp.cpp, creates a high-resolution grid (N, 2048) where the vertex height is calculated using a multi-component Gerstner wave sum (additive, with randomized wavelength/amplitude and a “chop”), and also calculates the shade and foam effects using ocean.frag, where the code uses the FRampNoise FB for the foam, thresholded where waveHeight > 18.0, steeping-dependent foam, Fresnel effects, and dark linear fog. The clouds are created in the build_Cloud routine, where thousands of sphere primitives of sizes 3-6 are reduced and shaded with cloud.frag or the simplified water.frag.

Important features of the image correspond well with the implementation, including large polygons and silhouettes on cloud undersides (low sphere res), horizontal banding and dim sky reflections from the dark fog, and discretely colored bright whitecaps where the crest threshold and slope test trigger the foam. The presence of the smoke/atmosphere and the reduced Fresnel allow realistic moonlit shadows but restrict realistic sky reflections, and the foam

appears rough and blocky because it is rendered by a harsh height threshold together with FBM.

Notes on performance and resource requirements are also significant: For N=2048, the mesh will contain about 4,194,304 vertices and around 25 million indices, which historically suggests hundreds of megabytes of GPU buffer space, very roughly around 200-400+ megabytes, not accounting for textures and other assets. It will take several seconds (often 5-30s, depending on the CPU and the allocator) to build the mesh on the CPU, and sustained rendering of such a dense, expensive shader on a mid-range GPU (like the GTX 10/20 series) could get into single or lower double digits per frame for interactive frames, or simply not work on lower-end GPUs. Can also modify the cloud variables which are indicated by “MODIFIABLE” comment in the void setup of ofApp.cpp and reducing the amount of clouds, or going to build_Clouds and reducing the layer range that forms each cloud, will result in a smoother simulation as there will be less shapes to render.

In short, the current system is excellent for large-scale, artistically rendered storm silhouettes and dark studies, but not for the finer details of foam filigree, volumetric cloud scattering, nor real-time operations at the current image resolution. The highest-return modifications and improvements will come from using a GPU vertex shader for displacement (enabling smaller base mesh sizes and animating them in real time), the implementation of level of detail or clip mapping, and using curvature or slope-based, or particle-based, foam.

[E] Postmortem. Discuss the project in general. What are the most impressive aspects of your project? Are you happy with the results overall? What is the weak point in your approach? What would you do differently if you had to do it again, and what might you add if you had more time?

The most interesting parts of this project, and what I think are the big accomplishments here, are the large-scale visual effects that are accomplished through fairly simple approaches: the multi-component Gerstner wave sum, computed in `ofApp.cpp`, provides realistic and controllable patterns for the swell, and the procedural cloud generator (many sphere primitives together) provides highly sculptural cloud silhouettes with very minimal assets. The optimized fragment shader code in `ocean.frag`, integrating FBM-based foam, steepness testing, and a dark distance fog, provides a cohesive and atmospheric visual identity that scales well.

On the whole, I’m satisfied with the artistic result: the image depicts the presence of a stormy environment, pronounced silhouettes, and a distinct style, and the system is very easy to

control (lists of the wave components, wave directions, fog, and light parameters are easy to adjust). The deterministic cloud seeding and parameterization schemes allow for fast iterative processing of the still images, and the mesh and pixel processes are separated, which allows for swift examination of the look dev effects in the shaders using the ocean.frag and water.frag files.

The weakest aspects are the technology: large-scale construction and storage of a CPU-created 2048×2048 mesh is computationally expensive, and the foam model is simply driven by the hard height map threshold and FBM, which is very patchy, not fine and streaky. Clouds, being surface geometry, do not have volumetric lighting transport and have visible polygons because the sphere level of details are not high. Also, the scene is not dynamic, with animation off, and not using temporal filtering, which is essential for the smooth animation of the scene that will not flicker.

If I were to do this again, I'd build on displacement on the GPU (vertex displacement, or heightfield textures), and implement level of detail and clip maps so that the environment provides artist control but also allows for interactivity, and simply doing that will offer the greatest possible speedup. Next, I'd add curvature-based or screen space foam, along with particle spray, and for clouds, I'd EITHER use volumetric ray marching (for realism) OR impostors with softly edged alpha (for speed). Finally, if I had the time, I'd also implement profiling tools within ofApp.cpp (mesh construction, frames), GPU queries, and a toggle for comparing the three sizes of the N grid. With more time, we would add randomly generating lightning, animated clouds and ocean, changing the still image to a full blown environment. The lightning would emit its own light and interact with the waves it hits. Clouds would signal when they would fire off lightning by lighting up, and changing shape temporarily.

[F] Conclusions. Describe the main outcomes. What is the takeaway? Review the key points of the report. Point to directions for followup work, if you have ideas for some.

The project illustrates that, using a comparatively straightforward pipe, Gerstner wave sum on the CPU in ofApp.cpp, FBM-based foam and dark fog on the ocean surface in ocean.frag, and sphere clouds aggregated using build_Cloud, is sufficient for achieving a dramatic, ominous, and visually interesting storm scene with realistic large-scale silhouettes. The results of the project are a dramatic still image that conveys scale and environment, a

parameter set that is easy to control for wave and lighting effects, and a content pipe that is very good for still image look development.

The trade-offs are simulation complexity and detail: the strategy used is very heavy on the CPU and memory for N=2048 (millions of vertices, tens of millions of indices), the foam is threshold-based, hence very coarse, and the clouds are surface mesh representations, not volumetrics, which reduces the simulated scattering of soft clouds and small wisps. The lessons learned are that the code works very well for stylized, controlled storms and art for stills, but for real-time and higher physical accuracy, one should write the displacement map on the GPU (vertex shader or heightfield), use LOD/clipmap, lower the base map sizes, use curvature/slope-based or particle systems for the foam, and go for volumetrics or impostors for the clouds. For the follow-up project, I would have focused on: (1) adding light profiling for estimating mesh build and frame rates on the target platform, (2) adding a GPU-displacement vertex shader with a lower-resolution base grid and comparing the performance vs. quality, and (3) enhancing the quality of the foam/cloud: (a) curvature-based foam & particle spray, or (b) volumetric cloud raymarching & impostors.