Crest

Release 4.16

Wave Harmonic & Contributors

Nov 22, 2022

ABOUT

CHAPTER

ONE

INTRODUCTION

Crest is a technically advanced ocean system for Unity.

It is architected for performance and makes heavy use of Level Of Detail (LOD) strategies and GPU acceleration for fast update and rendering. It is also highly flexible and allows any custom input to the water shape/foam/dynamic waves/etcetera, and has an intuitive and easy to use shape authoring interface.

This documentation is for *Crest* 4.16 and targets BIRP (Built-in Render Pipeline).

You can view this documentation online.

This documentation is for *Crest* 4.16 and targets HDRP (High Definition Render Pipeline).

You can view this documentation online.

This documentation is for *Crest* 4.16 and targets URP (Universal Render Pipeline).

You can view this documentation online.

1.1 Sponsorship

Sponsor

Sponsor Wave Harmonic on GitHub Sponsors to increase development time on Crest.

Throughout the documentation, you will see sponsor admonitions like this one for features where only expanded funding can help cover development costs.

Sponsor Us

1.2 Social

- **YouTube** https://www.youtube.com/channel/UC7_ZKKCXZmH64rRZqe-C0WA
- **Twitter** https://twitter.com/@crest_ocean
- **Discord** https://discord.gg/g7GpjDC

CHAPTER

TWO

KNOWN ISSUES

We keep track of issues on GitHub for all pipelines. Please see the following links:

- Issues on GitHub.
- BIRP specific issues on GitHub.
- HDRP specific issues on GitHub.
- URP specific issues on GitHub.

If you discover a bug, please open a bug report or mention it on the bugs channel on our Discord.

2.1 Unity Bugs

There are some Unity issues that affect *Crest*. Some of these may even be blocking new features from being developed. If you could vote on these issues, that would be greatly appreciated:

- Gizmos render over opaque objects with Post-Processing stack. *BIRP*

2.2 Unity Features

There are upcoming features being developed by Unity which will greatly help *Crest*. If you could vote on these features, that would be greatly appreciated:

- Post Processing Custom Effects *URP*

2.3 Prefab Mode Not Supported

Crest does not support running in prefab mode which means dirty state in prefab mode will not be reflected in the scene view. Save the prefab to see the changes.

CHAPTER

THREE

ROADMAP

Sponsor

This will help us expand our roadmap and achieve goals sooner. Certain sponsor tiers allows one to vote on roadmap items to guide our priorities.

Sponsor Us

Visit our roadmap to see where Crest's development is heading:

Trello Board

CHAPTER

FOUR

RELEASE NOTES

4.16

Breaking

- Set minimum Unity version to 2020.3.40.
- Set minimum render pipeline package version to 10.10. *HDRP URP*

Changed

- Add support for multiple cameras to the *Underwater Renderer*. One limitation is that underwater culling will be disabled when using multiple *Underwater Renderer*s.
- ShapeFFT/Gerstner can now take a mesh renderer as an input.
- Add *Crest/Inputs/Shape Waves/Sample Spectrum* shader which samples the spectrum using a texture.
- Ocean inputs provided via the *Register* components now sort on sibling index in addition to queue, so multiple inputs with the same queue can be organised in the hierarchy to control sort order.
- Add ability to alpha blend waves (effectively an override) instead of only having additive blend waves. Set *Blend Mode* to *Alpha Blend* on the *ShapeFFT* or *ShapeGerstner* to use. It's useful for preventing rivers and lakes from receiving ocean waves.
- Add *Water Tile Prefab* field to *Ocean Renderer* to provide more control over water tile mesh renderers like reflection probes settings.
- Warn users that edits in prefab mode will not be reflected in scene view until prefab is saved.
- Validate that no scale can be applied to the *OceanRenderer*.
- Viewpoint validation has been removed as it was unnecessary and spammed the logs.
- Whirlpool now executes in edit mode.
- *Visualise Ray Trace* now executes in edit mode.
- *Render Alpha On Surface* now executes in edit mode.
- Only report no Shape component validation as help boxes (ie no more console logs).
- Remove outdated lighting validation.
- Validate layers to warn users of potential build failures if *Crest* related renderers are not on the same layer as the *OceanRenderer.Layer*.
- No longer log info level validation to the console.

- Add info validation for tips on using reflection probes when found in a scene.
- Set *Ocean Renderer Wind Speed* default value to the maxmimum to reduce UX friction for new users.
- Also search *Addressables* and *Resources* for ocean materials when stripping keywords from underwater shader.
- Add *Ocean Renderer* > *Extents Size Multiplier* to adjust the extents so they can be increased in size to meet the horizon in cases where they do not.
- Greatly improve performance when many SphereWaterInteraction components are used by utilising GPU Instancing.
- Improve example scenes.
- Improve *Ocean Depth Cache* capture performance by excluding all render features. *URP*

Fixed

- Fix FFTs incorrectly adding extra foam.
- Limit minimum phase period of flow technique applied to waves to fix objectionable phasing issues in flowing water like rivers.
- Fix some components breaking in edit mode after entering/exiting prefab mode.
- Fix *Build Processor* deprecated/obsolete warnings.
- Fix spurious "headless/batch mode" error during builds.
- Greatly improve spline performance in the editor.
- Fix PSSL compiler errors.
- Fix incompatibility with EasySave3 and similar assets where water tiles would be orphaned when exiting play mode.
- Fix ocean tiles being pickable in the editor.
- Fix several memory leaks.
- Fix *Sea Floor Depth Data* disabled state as it was still attenuating waves when disabled.
- No longer execute when building which caused several issues.
- Fix self-intersecting polygon (and warning) on Ferry model.
- Fix *Examples* scene UI not scaling and thus looking incorrect for non 4K resolution.
- Fix build failure for *main* scene if reflection probe is added that excluded the *Water* layer.
- Prevent bad values (NaN etc) from propagating in the *Dynamic Waves* simulation. This manifested as the water surface disappearing from a singlar point.
- Fix shader include path error when moving *Crest* folder from the standard location.
- No longer disable the *Underwater Renderer* if it fails validation.
- Fix *Underwater Curtain* lighting not matching the water surface causing a visible seam at the far plane. *BIRP URP*
- Fix "mismatching output texture dimension" error when using XR SPI (Single-Pass Instanced). *BIRP URP*
- Fix caustics not rendering in XR SPI when shadow simulation is disabled. *BIRP*
- Fix XR spectator camera breaking if shadow simulation enabled. *BIRP*
- Fix shadow simulation executing for all cameras which could cause incorrect shadows. *BIRP*

- Fix underwater effect not rendering properly if spectator camera is used with XR SPI. *BIRP*
- Fix ocean moving in edit mode when *Always Refresh* is disabled. *HDRP*
- Fix ocean not rendering if no active *Underwater Renderer* is present. *HDRP*
- Fix *Clip Surface* adding negative alpha values when *Alpha Clipping* is disabled on the ocean material. *HDRP*
- Fix *Sort Priority* on the ocean material not having an effect. *HDRP*
- Improve performance by removing duplicated pass when using shadow simulation. *HDRP*
- Improve XR MP (Multi-Pass) performance by removing shadow copy pass from the right eye. *HDRP*
- Fix Unity 2022.2 shader compilation errors. *HDRP*
- Fix Unity 2023.1 script compilation errors. *HDRP*
- Fix *Underwater Renderer* incompatibility with SSAO (Screen-Space Ambient Occlusion). *URP*
- Fix Unity 2022.2 obsolete warnings. *URP*

Attention: The history has been trimmed but the full history can be viewed online.

CHAPTER

FIVE

GETTING STARTED

This section has steps for importing the *Crest* content into a project, and for adding a new ocean surface to a scene.

Warning: When changing Unity versions, setting up a render pipeline or making changes to packages, the project can appear to break. This may manifest as spurious errors in the log, no ocean rendering, magenta materials, scripts unassigned in example scenes, etcetera. Often, restarting the Editor fixes it. Clearing out the *Library* folder can also help to reset the project and clear temporary errors. These issues are not specific to *Crest*, but we note them anyway as we find our users regularly encounter them.

To augment / complement this written documentation we published a video available here:

BIRP

https://www.youtube.com/watch?v=qsgeG4sSLFw

Fig. 5.1: Getting Start with Crest for BIRP

HDRP

https://www.youtube.com/watch?v=FE6l39Lt3js

Fig. 5.2: Getting Start with Crest for HDRP

URP

https://www.youtube.com/watch?v=TpJf13d_-3E

Fig. 5.3: Getting Start with Crest for URP

5.1 Requirements

- Unity Version: 2020.3.40
- Shader compilation target 4.5 or above
- Crest does not support OpenGL or WebGL backends

BIRP

- The *Crest* example content uses the post-processing package (for aesthetic reasons). If this is not present in your project, you will see an unassigned script warning which you can fix by removing the offending script.

HDRP

- The minimum HDRP package version is 10.10

URP

- The minimum URP package version is 10.10

5.2 Importing *Crest* files into project

The steps to set up *Crest* in a new or existing project are as follows:

5.2.1 Pipeline Setup

BIRP

Ensure that BIRP is setup and functioning, either by setting up a new project using the BIRP template or by configuring your current project. This is beyond the scope of this documentation so please see the Unity documentation for more information.

Switch to Linear space rendering under *Edit* → *Project Settings* → *Player* → *Other Settings*. If your platform(s) require Gamma space, the material settings will need to be adjusted to compensate. Please see the Unity documentation for more information.

HDRP

Ensure that HDRP is setup and functioning, either by setting up a new project using the HDRP template or by configuring your current project. This is beyond the scope of this documentation so please see the Unity documentation for more information.

URP

Ensure that URP is setup and functioning, either by setting up a new project using the URP template or by configuring your current project. This is beyond the scope of this documentation so please see the Unity documentation for more information.

Switch to Linear space rendering under *Edit* → *Project Settings* → *Player* → *Other Settings*. If your platform(s) require Gamma space, the material settings will need to be adjusted to compensate. Please see the Unity documentation for more information.

5.2.2 Importing Crest

Import the *Crest* package into project using the *Asset Store* window in the Unity Editor.

Note: The files under Crest-Examples are not required by our core functionality, but are provided for illustrative purposes. We recommend first time users import them as they may provide useful guidance.

BIRP

Import *Crest* assets by either:

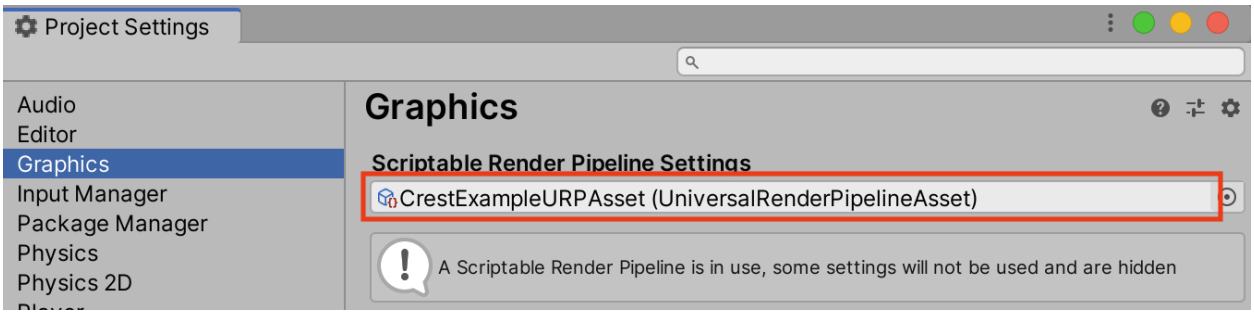
- Currently we do not prepare release packages. However, we do tag each asset store version, so the zip corresponding to each version can be downloaded by clicking the desired version on the Releases page.
- Getting latest by either cloning this repository or downloading it as a zip, and copying the *Crest/Assets/Crest/Crest* folder into your project. Be sure to always copy the .meta files.

Note: The *Crest/Assets/Crest/Development* folder is only used to develop *Crest* and should be skipped.

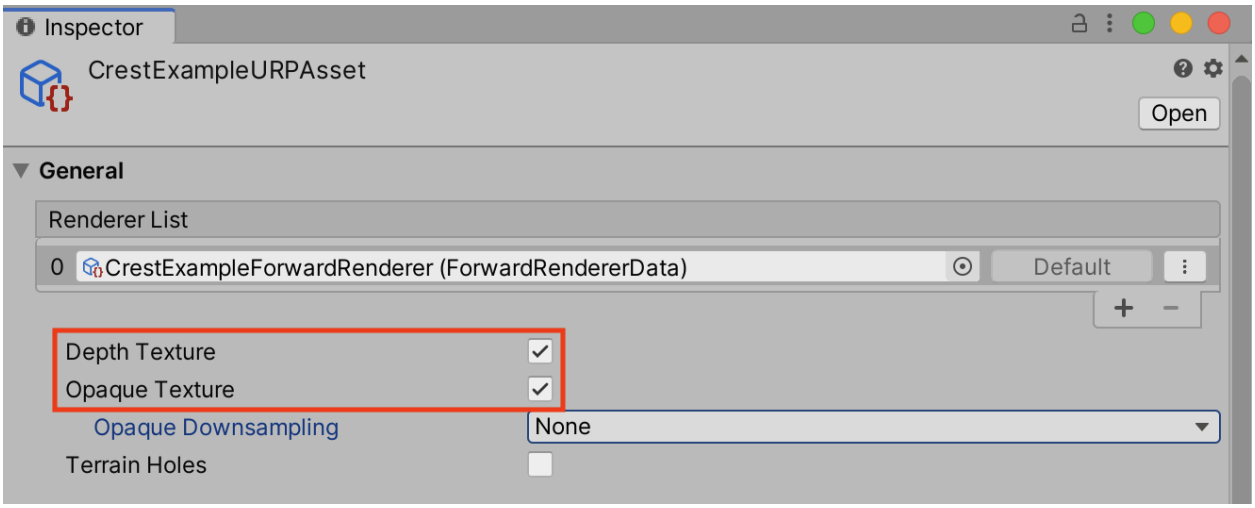
URP

Transparency

To enable the water surface to be transparent, two options must be enabled in the URP configuration. To find the configuration, open *Edit/Project Settings/Graphics* and double click the *Scriptable Render Pipeline Settings* field to open the render pipeline settings. This field will be populated if URP was successfully installed.



After double clicking the graphics settings should appear in the Inspector. Transparency requires the following two options to be enabled, *Depth Texture* and *Opaque Texture*:



Note: If you are using the underwater effect, it is recommended to set *Opaque Downsampling* to *None*. *Opaque Downsampling* will make everything appear at a lower resolution when underwater. Be sure to test to see if recommendation is suitable for your project.

Read Unity's documentation on the URP Asset for more information on these options.

Tip: If you are starting from scratch we recommend creating a project using a template in the Unity Hub.

5.3 Adding *Crest* to a Scene

The steps to add an ocean to an existing scene are as follows:

- Create a new *GameObject* for the ocean, give it a descriptive name such as *Ocean*.
  - Assign the *OceanRenderer* component to it. This component will generate the ocean geometry and do all required initialisation.
  - Assign the desired ocean material to the *OceanRenderer* script - this is a material using the *Crest/Ocean* shader.
  - Set the Y coordinate of the position to the desired sea level.
- Tag a primary camera as *MainCamera* if one is not tagged already, or provide the *Camera* to the *View Camera* property on the *OceanRenderer* script. If you need to switch between multiple cameras, update the *ViewCamera* field to ensure the ocean follows the correct view.
- Be sure to generate lighting if necessary. The ocean lighting takes the ambient intensity from the baked spherical harmonics. It can be found at the following:

  Window → *Rendering* → *Lighting Settings* → *Debug Settings* → *Generate Lighting*

  Tip: You can check *Auto Generate* to ensure lighting is always generated.

- To add waves, create a new GameObject and add the *Shape FFT* component. See *Wave Conditions* section for customisation.
- Any ocean seabed geometry needs set up to register it with *Crest*. See section *Shorelines and Shallows*.
- If the camera needs to go underwater, the underwater effect must be configured. See section *Underwater* for instructions.

5.4 Frequent Setup Issues

The following are kinks or bugs with the install process which come up frequently.

5.4.1 Errors present, or visual issues

Try restarting Unity as a first step.

5.4.2 Compile errors in the log, not possible to enter play mode, visual issues in the scene

Verify that render pipeline is installed and enabled in the settings. See the follow for documentation:

Upgrading to HDRP

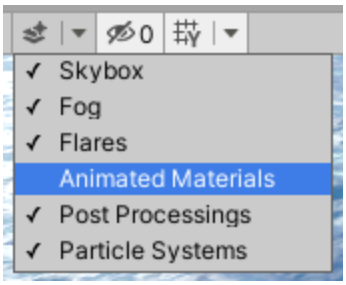
Installing URP into a project

5.4.3 Possible to enter play mode, but errors appear in the log at runtime that mention missing 'kernels'

Recent versions of Unity have a bug that makes shader import unreliable. Please try reimporting the *Crest/Shaders* folder using the right click menu in the project view. Or simply close Unity, delete the Library folder and restart which will trigger everything to reimport.

5.4.4 Ocean framerate low in edit mode

By default, the update speed is intentionally throttled by Unity to save power when in edit mode. To enable real-time update, enable *Animated Materials* in the Scene View toggles:



See the Unity Documentation for more information.

5.4.5 Ocean reflections/lighting/fog looks wrong *HDRP*

If reflections appear wrong, it can be useful to make a simple test shadergraph with our water normal map applied to it, to compare results. We provide a simple test shadergraph for debugging purposes - enable the *Apply test material* debug option on the *OceanRenderer* component to apply it. If you find you are getting good results with a test shadergraph but not with our ocean shader, please report this to us.

5.4.6 Changes made in prefab mode are not reflected in the scene view

Crest does not support running in prefab mode which means dirty state in prefab mode will not be reflected in the scene view. Save the prefab to see the changes.

CHAPTER

SIX

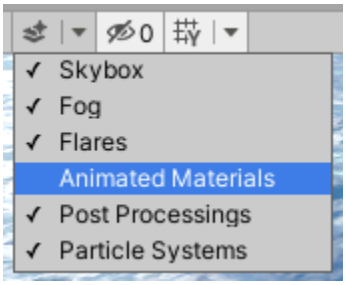
CONFIGURATION

Some quick start pointers for changing the ocean look and behaviour:

- Ocean surface appearance: The active ocean material is displayed below the *OceanRenderer* component. The material parameters are described in section *Material Parameters*. Turn off unnecessary features to maximize performance.
- Animated waves / ocean shape: Configured on the *ShapeFFT* script by providing an *Ocean Wave Spectrum* asset. This asset has an equalizer-style interface for tweaking different scales of waves, and also has some parametric wave spectra from the literature for comparison. See section *Wave Conditions*.
- Shallow water: Any ocean seabed geometry needs set up to register it with *Crest*. See section *Shorelines and Shallows*.
- Ocean foam: Configured on the *OceanRenderer* script by providing a *Sim Settings Foam* asset.
- Underwater: If the camera needs to go underwater, the underwater effect must be configured. See section *Underwater* for instructions.
- Dynamic wave simulation: Simulates dynamic effects like object-water interaction. Configured on the *OceanRenderer* script by providing a *Sim Settings Wave* asset, described in section *Simulation Settings*.
- A big strength of *Crest* is that you can add whatever contributions you like into the system. You could add your own shape or deposit foam onto the surface where desired. Inputs are generally tagged with the *Register* scripts and examples can be found in the example content scenes.

All settings can be changed at run-time and live authored. When tweaking ocean shape it can be useful to freeze time (from script, set *Time.timeScale* to 0) to clearly see the effect of each octave of waves.

Tip: By default, the update speed is intentionally throttled by Unity to save power when in edit mode. To enable real-time update, enable *Animated Materials* in the Scene View toggles:



See the Unity Documentation for more information.

6.1 Material Parameters

6.1.1 Normals

Overall Normal Strength Strength of the final surface normal (includes both wave normal and normal map)
Use Normal Map Whether to add normal detail from a texture. Can be used to add visual detail to the water surface *BIRP URP*
Normal Map Normal map and caustics distortion texture (should be set to Normals type in the properties)
Normal Map Scale Scale of normal map texture
Normal Map Strength Strength of normal map influence

6.1.2 Scattering

Scatter Colour Base Base colour when looking straight down into water.
Scatter Colour Grazing Base colour when looking into water at shallow/grazing angle. *BIRP URP*
Enable Shadowing Changes colour in shadow. Requires 'Create Shadow Data' enabled on OceanRenderer script. *BIRP URP*
Scatter Colour Shadow Base colour in shadow. Requires 'Create Shadow Data' enabled on OceanRenderer script.

6.1.3 Subsurface Scattering

Enable Whether to to emulate light scattering through the water volume. *BIRP URP*
SSS Tint Colour tint for primary light contribution.
SSS Intensity Base Amount of primary light contribution that always comes in.
SSS Intensity Sun Primary light contribution in direction of light to emulate light passing through waves.
SSS Sun Falloff Falloff for primary light scattering to affect directionality.

6.1.4 Shallow Scattering

Deprecated

Shallow Scattering will be removed in a future version. A properly tweaked *Depth Fog Density* achieves better results at lower cost. Consider copying over the value from our materials.

The water colour can be varied in shallow water (this requires a depth cache created so that the system knows which areas are shallow, see section *Shorelines and Shallows*).

Enable Enable light scattering in shallow water. *BIRP URP*
Scatter Colour Shallow Scatter colour used for shallow water.
Scatter Colour Depth Max Maximum water depth that is considered 'shallow', in metres. Water that is deeper than this depth is not affected by shallow colour.
Scatter Colour Depth Falloff Falloff of shallow scattering, which gives control over the appearance of the transition from shallow to deep.
Scatter Colour Shallow Shadow Shallow water colour in shadow (see comment on Shadowing param above). *BIRP URP*

6.1.5 Reflection Environment

Specular Strength of specular lighting response.

Occlusion Strength of reflection. *HDRP*

Smoothness Smoothness of surface. *HDRP URP*

Vary Smoothness Over Distance Helps to spread out specular highlight in mid-to-background. From a theory point of view, models transfer of normal detail to microfacets in BRDF. *URP*

Smoothness Far Material smoothness at far distance from camera. *HDRP URP*

Smoothness Far Distance Definition of far distance. *HDRP URP*

Smoothness Falloff How smoothness varies between near and far distance. *HDRP URP*

Roughness Controls blurriness of reflection *BIRP*

Softness Acts as mip bias to smooth/blur reflection. *URP*

Light Intensity Multiplier Main light intensity multiplier. *URP*

Fresnel Power Controls harshness of Fresnel behaviour. *BIRP URP*

Refractive Index of Air Index of refraction of air. Can be increased to almost 1.333 to increase visibility up through water surface. *BIRP URP*

Deprecated
The *Refractive Index of Air* property will be removed in a future version.

Refractive Index of Water Index of refraction of water. Typically left at 1.333.

Planar Reflections Dynamically rendered 'reflection plane' style reflections. Requires OceanPlanarReflection script added to main camera. *BIRP URP*

Planar Reflections Distortion How much the water normal affects the planar reflection. *BIRP URP*

Override Reflection Cubemap Whether to use an overridden reflection cubemap (provided in the next property). *BIRP*

Reflection Cubemap Override Custom environment map to reflect. *BIRP*

6.1.6 Add Directional Light

Enable Add specular highlights from the the primary light. *BIRP*

Boost Specular highlight intensity. *BIRP*

Falloff Falloff of the specular highlights from source to camera. *BIRP*

Vary Falloff Over Distance Helps to spread out specular highlight in mid-to-background. *BIRP*

Far Distance Definition of far distance. *BIRP*

Falloff At Far Distance Same as "Falloff" except only up to "Far Distance". *BIRP*

6.1.7 Procedural Skybox

Enable Enable a simple procedural skybox. Not suitable for realistic reflections, but can be useful to give control over reflection colour - especially in stylized/non realistic applications. *BIRP URP*

Base Base sky colour. *BIRP URP*

Towards Sun Colour in sun direction. *BIRP URP*

Directionality Direction fall off. *BIRP URP*

Away From Sun Colour away from sun direction. *BIRP URP*

6.1.8 Foam

Enable Enable foam layer on ocean surface.
Foam Foam texture.
Foam Scale Foam texture scale.
Foam Feather Controls how gradual the transition is from full foam to no foam.
Foam Tint Colour tint for whitecaps / foam on water surface. *BIRP URP*
Light Scale Scale intensity of lighting. *BIRP URP*
Shoreline Foam Min Depth Proximity to sea floor where foam starts to get generated. *BIRP URP*
Foam Albedo Intensity Scale intensity of diffuse lighting. *HDRP*
Foam Emissive Intensity Scale intensity of emitted light. *HDRP*
Foam Smoothness Smoothness of foam material. *HDRP*

6.1.9 Foam 3D Lighting

Enable Generates normals for the foam based on foam values/texture and use it for foam lighting. *BIRP URP*
Foam Normal Strength Strength of the generated normals.
Specular Fall-Off Acts like a gloss parameter for specular response. *BIRP URP*
Specular Boost Strength of specular response. *BIRP URP*

6.1.10 Foam Bubbles

Foam Bubbles Color Colour tint bubble foam underneath water surface.
Foam Bubbles Parallax Parallax for underwater bubbles to give feeling of volume.
Foam Bubbles Coverage How much underwater bubble foam is generated.

6.1.11 Transparency

Enable Whether light can pass through the water surface. *BIRP URP*
Refraction Strength How strongly light is refracted when passing through water surface.
Depth Fog Density Scattering coefficient within water volume, per channel.

6.1.12 Caustics

Enable Approximate rays being focused/defocused on underwater surfaces.
Caustics Caustics texture.
Caustics Scale Caustics texture scale.
Caustics Texture Grey Point The 'mid' value of the caustics texture, around which the caustic texture values are scaled.
Caustics Strength Scaling / intensity.
Caustics Focal Depth The depth at which the caustics are in focus.
Caustics Depth Of Field The range of depths over which the caustics are in focus.
Caustics Distortion Texture Texture to distort caustics. *HDRP*
Caustics Distortion Strength How much the caustics texture is distorted.
Caustics Distortion Scale The scale of the distortion pattern used to distort the caustics.

6.1.13 Underwater

Enable Whether the underwater effect is being used. This enables code that shades the surface correctly from underneath. *BIRP URP*

Cull Mode Ordinarily set this to *Back* to cull back faces, but set to *Off* to make sure both sides of the surface draw if the underwater effect is being used.

6.1.14 Flow

Example

Flow is demonstrated in the *whirlpool* example scene.

Enable Flow is horizontal motion in water. 'Create Flow Sim' must be enabled on the OceanRenderer to generate flow data.

6.2 Lighting

6.2.1 General

BIRP

Sponsor

Sponsoring us will help increase our development bandwidth which could work towards improving this feature.

Sponsor Us

Crest BIRP does not support additional lights due to bugs in the pipeline and performance concerns. Please see #382 and #383 for more details.

TODO

This section is a work in progress.

HDRP

As other shaders would, the ocean will get its lighting from the primary directional light (AKA sun). Like other mesh renderers, this can be masked by setting the *Rendering Layer Mask* property on the *Ocean Renderer*. Please see the HDRP documentation on light layers for more information on setup and usage.

But some lighting will come from the light set as the *Primary Light* on the *Ocean Renderer*. This includes the sub-surface scattering colour.

Lighting can also be overriden with the *Indirect Lighting Controller*. Please see the HDRP documentation on volume overrides for more information on setup and usage.

For the ocean to have lighting completely separate from everything else, you would need to do all of the above.

URP

Sponsor

Sponsoring us will help increase our development bandwidth which could work towards improving this feature.

Sponsor Us

Crest URP currently does not support additional lights.

TODO

This section is a work in progress.

6.2.2 Reflections

Reflections contribute hugely to the appearance of the ocean. The look of the ocean will dramatically changed based on the reflection environment.

The Index of Refraction setting controls how much reflection contributes for different view angles.

BIRP

The base reflection comes from a one of these sources:

- Unity's specular cubemap. This is the default and is the same as what is applied to glossy objects in the scene. It will support reflection probes, as long as the probe extents cover the ocean tiles, which enables real-time update of the reflection environment (see Unity documentation for more details).
- Override reflection cubemap. If desired a cubemap can be provided to use for the reflections. For best results supply a HDR cubemap.
- Procedural skybox. Developed for stylized games, this is a simple approximation of sky colours that will give soft results.

This base reflection can then be overridden by dynamic planar reflections. This can be used to augment the reflection with 3D objects such as boat or terrain. This can be enabled by applying the *Ocean Planar Reflections* script to the active camera and configuring which layers get reflected (don't include the *Water* layer). This renders every frame by default but can be configured to render less frequently. This only renders one view but also only captures a limited field of view of reflections, and the reflection directions are scaled down to help keep them in this limited view, which can give a different appearance. Furthermore 'planar' means the surface is approximated by a plane which is not the case for wavey ocean, so the effect can break down. This method is good for capturing local objects like boats and etcetera.

A good strategy for debugging the use of Unity's specular cubemap is to put another reflective/glossy object in the scene near the surface, and verify that it is lit and reflects the scene properly. Crest tries to use the same inputs for lighting/reflections, so if it works for a test object it should work for the water surface as well.

HDRP

Crest makes full use of the flexible lighting options in HDRP (it is lit the same as a shadergraph shader would be).

Planar Reflection Probes

HDRP comes with a *Planar Reflection Probe* feature which enables dynamic reflection of the environment at run-time, with a corresponding cost. See Unity's documentation on Planar Reflection Probes. At time of writing we used the following steps:

- Create new GameObject
- Set the height of the GameObject to the sea level.
- Add the component from the Unity Editor menu using *Component/Rendering/Planar Reflection Probe*
- Set the extents of the probe to be large enough to cover everything that needs to be reflected. We recommend starting large (1000m or more as a starting point).
- Ensure water is not included in the reflection by deselecting *Water* on the *Culling Mask* field
- Check the documentation linked above for details on individual parameters

HDRP's planar reflection probe is very sensitive to surface normals and often 'leaks' reflections, for example showing the reflection of a boat on the water above the boat. If you see these issues we recommend reducing the *Overall Normal Strength* parameter on the ocean material.

The planar reflection probe assumes the reflecting surface is a flat plane. This is not the case for for a wavey water surface and this can also produce 'leaky' reflections. In such cases it can help to lower the reflection probe below sea level slightly.

Screen-Space Reflections

HDRP has a separate setting for transparents to receive SSR (Screen-Space Reflections) and it is not enabled by default. It is important that you understand the basics of HDRP before proceeding.

1. Enable *Screen Space Refection* and the *Transparent* sub-option in the Frame Settings.
2. Add and configure the SSR Volume Override
    - Please learn how to use the *Volume Framework* before proceeding as covering this is beyond the scope of our documentation:

      https://www.youtube.com/watch?v=vczkfjLoPf8

      Fig. 6.1: Adding Volumes to HDRP (Tutorial)

3. Enable *Receives Screen-Space Reflections* on the ocean material.

URP

The base reflection comes from a one of these sources:

- Unity's specular cubemap. This is the default and is the same as what is applied to glossy objects in the scene. It will support reflection probes, as long as the probe extents cover the ocean tiles, which enables real-time update of the reflection environment (see Unity documentation for more details).
- Procedural skybox. Developed for stylized games, this is a simple approximation of sky colours that will give soft results.

This base reflection can then be overridden by dynamic planar reflections. This can be used to augment the reflection with 3D objects such as boat or terrain. This can be enabled by applying the *Ocean Planar Reflections* script to the active camera and configuring which layers get reflected (don't include the *Water* layer). This renders every frame by default but can be configured to render less frequently. This only renders one view but also only captures a limited field of view of reflections, and the reflection directions are scaled down to help keep them in this limited view, which can give a different appearance. Furthermore 'planar' means the surface is approximated by a plane which is not the

case for wavey ocean, so the effect can break down. This method is good for capturing local objects like boats and etcetera.

A good strategy for debugging the use of Unity's specular cubemap is to put another reflective/glossy object in the scene near the surface, and verify that it is lit and reflects the scene properly. Crest tries to use the same inputs for lighting/reflections, so if it works for a test object it should work for the water surface as well.

6.2.3 Refractions

Refractions sample from the camera's colour texture. Anything rendered in the transparent pass or higher will not be included in refractions.

See *Transparent Object In Front Of Ocean Surface* for issues with Crest and other refractive materials.

6.3 Orthographic Projection

Crest supports orthographic projection out-of-the-box, but it might require some configuration to get a desired appearance.

Crest uses the camera's position for the LOD system which can be awkward for orthographic which uses the size property on the camera. Use the *Viewpoint* property on the *Ocean Renderer* to override the camera's position.

Underwater effects do *not* currently support orthographic projection.

6.4 Ocean Construction Parameters

There are a small number of parameters that control the construction of the ocean shape and geometry:

- **Lod Data Resolution** - the resolution of the various ocean LOD data including displacement textures, foam data, dynamic wave sims, etc. Sets the 'detail' present in the ocean - larger values give more detail at increased run-time expense.
- **Geometry Down Sample Factor** - geometry density - a value of 2 will generate one vert per 2x2 LOD data texels. A value of 1 means a vert is generated for every LOD data texel. Larger values give lower fidelity surface shape with higher performance.
- **Lod Count** - the number of levels of detail / scales of ocean geometry to generate. The horizontal range of the ocean surface doubles for each added LOD, while GPU processing time increases linearly. It can be useful to select the ocean in the scene view while running in editor to inspect where LODs are present.
- **Max Scale** - the ocean is scaled horizontally with viewer height, to keep the meshing suitable for elevated viewpoints. This sets the maximum the ocean will be scaled if set to a positive value.
- **Min Scale** - this clamps the scale from below, to prevent the ocean scaling down to 0 when the camera approaches the sea level. Low values give lots of detail, but will limit the horizontal extents of the ocean detail.

6.5 Advanced Ocean Parameters

These parameters are found on the *Ocean Renderer* under the *Advanced* heading.

- **Surface Self-Intersection Mode** - How Crest should handle self-intersections of the ocean surface caused by choppy waves which can cause a flipped underwater effect. When not using the portals/volumes, this fix is only applied when within 2 metres of the ocean surface. *Automatic* will disable the fix if portals/volumes are used and is the recommended setting.
- **Underwater Cull Limit** - Proportion of visibility below which ocean will be culled underwater. The larger the number, the closer to the camera the ocean tiles will be culled.

CHAPTER

SEVEN

OCEAN SIMULATION

The following sections cover the major elements of the ocean simulation. All of these can be directly controlled with user input, as covered in this video:

https://www.youtube.com/watch?v=sQIakAjSq4Y

Fig. 7.1: Basics of Adding Ocean Inputs

Note: Inputs only execute the first shader pass (pass zero). It is recommended to use unlit shader templates or unlit *Shader Graph* (URP only) if not using one of ours.

Tip: For inputs, you are not limited to only using a MeshRenderer. Almost any renderer can be used like a TrailRenderer, LineRenderer or ParticleSystem.

The following shaders can be used with any ocean input:

- **Scale By Factor** scales the ocean data between zero and one inclusive. It is multiplicative, which can be inverted, so zero becomes no data and one leaves the data unchanged.

7.1 Animated Waves

7.1.1 Overview

The Animated Waves simulation contains the animated surface shape. This typically contains the ocean waves (see the Wave conditions section below), but can be modified as required. For example, parts of the water can be pushed down below geometry if required.

The animated waves sim can be configured by assigning an Animated Waves Sim Settings asset to the OceanRenderer script in your scene (*Create* → *Crest* → *Animated Wave Sim Settings*).

The waves will be dampened/attenuated in shallow water if a *Sea Floor Depth* LOD data is used (see *Sea Floor Depth*). The amount that waves are attenuated is configurable using the *Attenuation In Shallows* setting.

7.1.2 Simulation Settings

All of the settings below refer to the *Animated Waves Sim Settings* asset.

- **Attenuation In Shallows** - How much waves are dampened in shallow water.
- **Shallows Max Depth** - Any water deeper than this will receive full wave strength. The lower the value, the less effective the depth cache will be at attenuating very large waves. Set to the maximum value (1,000) to disable.
- **Collision Source** - Where to obtain ocean shape on CPU for physics / gameplay.
- **Max Query Count** - Maximum number of wave queries that can be performed when using ComputeShader-Queries.
- **Ping Pong Combine Pass** - Whether to use a graphics shader for combining the wave cascades together. Disabling this uses a compute shader instead which doesn't need to copy back and forth between targets, but it may not work on some GPUs, in particular pre-DX11.3 hardware, which do not support typed UAV loads. The fail behaviour is a flat ocean.
- **Render Texture Graphics Format** - The render texture format to use for the wave simulation. Consider using higher precision (like R32G32B32A32_SFloat) if you see tearing or wierd normals. You may encounter this issue if you use any of the *Set Water Height* inputs.

7.1.3 User Inputs

Preview

Splines now support animated waves. Please see *Wave Splines* for general spline information.

To add some shape, add some geometry into the world which when rendered from a top down perspective will draw the desired displacements. Then assign the *Register Anim Waves Input* script which will tag it for rendering into the shape. This is demonstrated in Fig. 7.1

There is an example in the *boat.unity* scene, GameObject *wp0*, where a smoothstep bump is added to the water shape. This is an efficient way to generate dynamic shape. This renders with additive blend, but other blending modes are possible such as alpha blend, multiplicative blending, and min or max blending, which give powerful control over the shape.

The following input shaders are provided under *Crest/Inputs/Animated Waves*:

- **Add From Texture** allows any kind of shape added to the surface from a texture. Can ether be a heightmap texture (1 channel) or a 3 channel XYZ displacement texture. Optionally the alpha channel can be used to write to subsurface scattering which increases the amount of light emitted from the water volume, which is useful for approximating light scattering.
- **Scale By Factor** scales the waves by a factor where zero is no waves and one leaves waves unchanged. Useful for reducing waves.
- **Set Base Water Height Using Geometry** allows the sea level (average water height) to be offset some amount. The top surface of the geometry will provide the water height, and the waves will apply on top.
- **Push Water Under Convex Hull** pushes the water underneath the geometry. Can be used to define a volume of space which should stay 'dry'.
- **Set Water Height Using Geometry** snaps the water surface to the top surface of the geometry. Will override any waves.
- **Wave Particle** is a 'bump' of water. Many bumps can be combined to make interesting effects such as wakes for boats or choppy water. Based loosely on http://www.cemyuksel.com/research/waveparticles/.

7.2 Dynamic Waves

7.2.1 Overview

Crest includes a multi-resolution dynamic wave simulation, which allows objects like boats to interact with the water.

To turn on this feature, enable the *Create Dynamic Wave Sim* option on the *OceanRenderer* script, and to configure the sim, create or assign a *Dynamic Wave Sim Settings* asset on the *Sim Settings Dynamic Waves* option.

The dynamic wave simulation is added on top of the animated FFT waves to give the final shape.

The dynamic wave simulation is not suitable for use further than approximately 10km from the origin. At this kind of distance the stability of the simulation can be compromised. Use the *FloatingOrigin* component to avoid travelling far distances from the world origin.

7.2.2 Adding Interaction Forces

Dynamic ripples from interacting objects can be generated by placing one or more spheres under the object to approximate the object's shape. To do so, attach one or more *SphereWaterInteraction* components to children with the object and set the *Radius* parameter to roughly match the shape.

The following settings can be used to customise the interaction:

- **Radius** - The radius of the sphere from which the interaction forces are calculated.
- **Weight** - Strength of the effect. Can be set negative to invert.
- **Weight Up Down Mul** - Multiplier for vertical motion, scales ripples generated from a sphere moving up or down.
- **Inner Sphere Multiplier** - Internally the interaction is modelled by a pair of nested spheres. The forces from the two spheres combine to create the final effect. This parameter scales the effect of the inner sphere and can be tweaked to adjust the shape of the result.
- **Inner Sphere Offset** - This parameter controls the size of the inner sphere and can be tweaked to give further control over the result.
- **Velocity Offset** - Offsets the interaction position in the direction of motion. There is some latency between applying a force to the wave sim and the resulting waves appearing. Applying this offset can help to ensure the waves do not lag behind the sphere.
- **Compensate For Wave Motion** - If set to 0, the input will always be applied at a fixed position before any horizontal displacement from waves. If waves are large then their displacement may cause the interactive waves to drift away from the object. This parameter can be increased to compensate for this displacement and combat this issue. However increasing too far can cause a feedback loop which causes strong 'ring' artifacts to appear in the dynamic waves. This parameter can be tweaked to balance this two effects.

Non-spherical objects can be approximated with multiple spheres, for an example see the *Spinner* object in the *boat.unity* example scene which is composed of multiple sphere interactions. The intensity of the interaction can be scaled using the *Weight* setting. For an example of usages in boats, search for GameObjects with "InteractionSphere" in their name in the *boat.unity* scene.

7.2.3 Simulation Settings

All of the settings below refer to the *Dynamic Wave Sim Settings* asset.

The key settings that impact stability of the simulation are the **Damping** and **Courant Number** settings described below.

- **Simulation Frequency** - Frequency to run the dynamic wave sim, in updates per second. Lower frequencies can be more efficient but may limit wave speed or lead to visible jitter. Default is 60 updates per second.
- **Damping** - How much energy is dissipated each frame. Helps sim stability, but limits how far ripples will propagate. Set this as large as possible/acceptable. Default is 0.05.
- **Courant Number** - Stability control. Lower values means more stable sim, but may slow down some dynamic waves. This value should be set as large as possible until sim instabilities/flickering begin to appear. Default is 0.7.
- **Horiz Displace** - Induce horizontal displacements to sharpen simulated waves.
- **Displace Clamp** - Clamp displacement to help prevent self-intersection in steep waves. Zero means unclamped.
- **Gravity Multiplier** - Multiplier for gravity. More gravity means dynamic waves will travel faster.
- **Attenuation in Shallows** - How much waves are dampened in shallow water.

The *OceanDebugGUI* script gives the debug overlay in the example content scenes and reports the number of sim steps taken each frame.

7.2.4 User Inputs

The recommended approach to injecting forces into the dynamic wave simulation is to use the *SphereWaterInteraction* component as described above. This component will compute a robust interaction force between a sphere and the water, and multiple spheres can be composed to model non-spherical shapes.

However for when more control is required custom forces can be injected directly into the simulation. The following input shader is provided under *Crest/Inputs/Dynamic Waves*:

- **Add Bump** adds a round force to pull the surface up (or push it down). This can be moved around to create interesting effects.

7.3 Foam

7.3.1 Overview

Crest simulates foam getting generated by choppy water (*pinched*) wave crests) and in shallow water to approximate foam from splashes at shoreline. Each update (default is 30 updates per second), the foam values are reduced to model gradual dissipation of foam over time.

To turn on this feature, enable the *Create Foam Sim* option on the *OceanRenderer* script, and ensure the *Enable* option is ticked in the Foam group on the ocean material.

To configure the foam sim, create a *Foam Sim Settings* asset by right clicking the a folder in the *Project* window and selecting *Create/Crest/Foam Sim Settings*, and assigning it to the OceanRenderer component in your scene.

7.3.2 User Inputs

Preview

Splines now support foam. Please see *Wave Splines* for general spline information.

Crest supports inputing any foam into the system, which can be helpful for fine tuning where foam is placed. To place foam, add some geometry into the world at the area where foam should be added. Then assign the *RegisterFoamInput* script which will tag it for rendering into the shape, and apply a material with a shader of type *Crest/Inputs/Foam/...*. See the *DepositFoamTex* object in the *whirlpool.unity* scene for an example.

The process for adding inputs is demonstrated in this Fig. 7.1.

The following input shaders are provided under *Crest/Inputs/Foam*:

- **Add From Texture** adds foam values read from a user provided texture. Can be useful for placing 'blobs' of foam as desired, or canbe moved around at runtime to paint foam into the sim.
- **Add From Vert Colours** can be applied to geometry and uses the red channel of vertex colours to add foam to the sim. Similar in purpose to *Add From Texture*, but can be authored in a modelling workflow instead of requiring at texture.
- **Override Foam** sets the foam to the provided value. Useful for removing foam from unwanted areas.

7.3.3 Simulation Settings

General Settings

- **Foam Fade Rate** - How quickly foam dissipates. Low values mean foam remains on surface for longer. This setting should be balanced with the generation *strength* parameters below.

Wave foam / whitecaps

Crest detects where waves are 'pinched' and deposits foam to approximate whitecaps.

- **Wave Foam Strength** - Scales intensity of foam generated from waves. This setting should be balanced with the *Foam Fade Rate* setting.
- **Wave Foam Coverage** - How much of the waves generate foam. Higher values will lower the threshold for foam generation, giving a larger area.

Shoreline foam

If water depth input is provided to the system (see **Sea Floor Depth** section below), the foam sim can automatically generate foam when water is very shallow, which can approximate accumulation of foam at shorelines.

- **Shoreline Foam Max Depth** - Foam will be generated in water shallower than this depth. Controls how wide the band of foam at the shoreline will be. Note that this is not a distance to shoreline, but a threshold on water depth, so the width of the foam band can vary based on terrain slope. To address this limitation we allow foam to be manually added from geometry or from a texture, see the next section.
- **Shoreline Foam Strength** - Scales intensity of foam generated in shallow water. This setting should be balanced with the *Foam Fade Rate* setting.

Developer Settings

These settings should generally be left unchanged unless one is experiencing issues.

- **Simulation Frequency** - Frequency to run the foam sim, in updates per second. Lower frequencies can be more efficient but may lead to visible jitter. Default is 30 updates per second.

7.4 Sea Floor Depth

This simulation stores information that can be used to calculate the water depth. Specifically it stores the terrain height, which can then be differenced with the sea level to obtain the water depth. This water depth is useful information to the system; it is used to attenuate large waves in shallow water, to generate foam near shorelines, and to provide shallow water shading. It is calculated by rendering the render geometry in the scene for each LOD from a top down perspective and recording the Y value of the surface.

The following will contribute to ocean depth:

- Objects that have the *RegisterSeaFloorDepthInput* component attached. These objects will render every frame. This is useful for any dynamically moving surfaces that need to generate shoreline foam, etcetera.
- It is also possible to place world space depth caches. The scene objects will be rendered into this cache once, and the results saved. Once the cache is populated it is then copied into the Sea Floor Depth LOD Data. The cache has a gizmo that represents the extents of the cache (white outline) and the near plane of the camera that renders the depth (translucent rectangle). The cache should be placed at sea level and rotated/scaled to encapsulate the terrain.

When the water is e.g. 250m deep, this will start to dampen 500m wavelengths, so it is recommended that the sea floor drop down to around this depth away from islands so that there is a smooth transition between shallow and deep water without a visible boundary.

7.5 Clip Surface

https://www.youtube.com/watch?v=jXphUy__J0o

Fig. 7.2: Water Bodies and Surface Clipping

This data drives clipping of the ocean surface, as in carving out holes. This can be useful for hollow vessels or low terrain that goes below sea level. Data can come from primitives (signed-distance), geometry (convex hulls) or a texture.

To turn on this feature, enable the *Create Clip Surface Data* option on the *OceanRenderer* script, and ensure the *Enable* option is ticked in the *Clip Surface* group on the ocean material.

The data contains 0-1 values. Holes are carved into the surface when the value is greater than 0.5.

7.5.1 Simulation Settings

All of the settings below refer to the *Clip Surface Sim Settings* asset.

- **Render Texture Graphics Format** - The render texture format to use for the clip surface simulation. Consider using higher precision (like *R16_UNorm*) if you are using *Primitive* mode for even more accurate clipping.

7.5.2 User Inputs

Primitive Mode

Clip areas can be added using signed-distance primitives which produces accurate clipping and supports overlapping. Add a *RegisterClipSurfaceInput* script to a *GameObject* and set *Mode* to *Primitive*. The position, rotation and dimensions of the primitive is determined by the *Transform*. See the *FloatingOpenContainer* object in the *boat.unity* scene for an example usage.

Geometry Mode

Clip areas can be added by adding geometry that covers the desired hole area to the scene and then assigning the *RegisterClipSurfaceInput* script and setting *Mode* to *Geometry*. See the *RowBoat* object in the *main.unity* scene for an example usage.

To use other available shaders like *ClipSurfaceRemoveArea* or *ClipSurfaceRemoveAreaTexture*: create a material, assign to renderer and disable *Assign Clip Surface Material* option. For the *ClipSurfaceRemoveArea* shaders, the geometry should be added from a top-down perspective and the faces pointing upwards.

The following input shaders are provided under *Crest/Inputs/Clip Surface*:

- **Convex Hull** - Renders geometry into clip surface data taking all dimensions into account. An example use case is rendering the convex hull of a vessel to remove the ocean surface from within it.

  Example

  See the *RowBoat* object in the *main.unity* scene for an example usage.

  Note: Overlapping or adjacent meshes will not work correctly in most cases. There will be cases where one mesh will overwrite another resulting in the ocean surface appearing where it should not. The mesh is rendered from a top-down perspective. The back faces add clip surface data and the front faces remove from it which creates the convex hull. With an overlapping mesh, the front faces of the sides of one mesh will clear the clipping data creating by the other mesh. Overlapping boxes which are not rotated on the X or Z axes will work well whilst spheres will have issues. Consider using *Primitive* mode which supports overlapping.

- **Include Area** - Removes clipping data so the ocean surface renders.
- **Remove Area** - Adds clipping data to remove the ocean surface.
- **Remove Area Texture** - Adds clipping data using a texture to remove the ocean surface.

7.6 Shadows

The shadow data consists of two channels. One is for normal shadows (hard shadow term) as would be used to block specular reflection of the light. The other is a much softer shadowing value (soft shadow term) that can approximately variation in light scattering in the water volume.

This data is captured from the shadow maps Unity renders before the transparent pass. These shadow maps are always rendered in front of the viewer. The Shadow LOD Data then reads these shadow maps and copies shadow information into its LOD textures.

BIRP

To turn on this feature, enable the *Create Shadow Data* option on the *OceanRenderer* script, and ensure the *Shadowing* option is ticked on the ocean material.

HDRP

To turn on this feature, enable the *Create Shadow Data* option on the *OceanRenderer* script.

Specular (direct) lighting on the ocean surface is not shadowed by this data. It is shadowed by the pipeline. But we still use the data to shadow anything not covered by the pipeline like caustic shadows.

To create this setup from scratch, the steps are the following.

1. On the HDRP asset (either the asset provided with Crest *Assets/Crest/CrestExampleHDRPAsset*, or the one used in your project), ensure that *Custom Pass* is enabled.
2. Shadow maps must be enabled in the frame settings for the camera.
3. Enable shadowing in Crest. Enable *Create Shadow Data* on the OceanRenderer script.
4. On the same script, assign a *Primary Light* for the shadows. This light needs to have shadows enabled, if not an error will be reported accordingly.
5. If desired the shadow sim can be configured by assigning a *Shadow Sim Settings* asset (*Create/Crest/Shadow Sim Settings*).

URP

To turn on this feature, enable the *Create Shadow Data* option on the *OceanRenderer* script, and ensure the *Shadowing* option is ticked on the ocean material.

To create this setup from scratch, the steps are the following.

1. In the shadow settings of the URP asset, ensure that shadow cascades are enabled. *Crest* requires cascades to be enabled to obtain shadow information.
2. Enable shadowing in Crest. Enable *Create Shadow Data* on the OceanRenderer script.
3. On the same script, assign a *Primary Light* for the shadows. This light needs to have shadows enabled, if not an error will be reported accordingly.
4. If desired the shadow sim can be configured by assigning a *Shadow Sim Settings* asset (*Create/Crest/Shadow Sim Settings*).
5. Enable *Shadowing* on the ocean material to compile in the necessary shader code

The shadow sim can be configured by assigning a Shadow Sim Settings asset to the OceanRenderer script in your scene (*Create/Crest/Shadow Sim Settings*). In particular, the soft shadows are very soft by default, and may not appear for small/thin shadow casters. This can be configured using the *Jitter Diameter Soft* setting.

There will be times when the shadow jitter settings will cause shadows or light to leak. An example of this is when trying to create a dark room during daylight. At the edges of the room the jittering will cause the ocean on the inside of the room (shadowed) to sample outside of the room (not shadowed) resulting in light at the edges. Reducing the

Jitter Diameter Soft setting can solve this, but we have also provided a *Register Shadow Input* component which can override the shadow data. This component bypasses jittering and gives you full control.

7.7 Flow

7.7.1 Overview

Flow is the horizontal motion of the water volumes. It does not affect wave directions, but transports the waves horizontally. This horizontal motion also affects physics.

Example

See the *whirlpool.unity* example scene where flow is used to rotate the waves and foam around the vortex.

7.7.2 User Inputs

Preview

Splines now support flow. Please see *Wave Splines* for general spline information.

Crest supports adding any flow velocities to the system. To add flow, add some geometry into the world which when rendered from a top down perspective will draw the desired displacements. Then assign the *RegisterFlowInput* script which will tag it for rendering into the flow, and apply a material using one of the following shaders.

The following input shaders are provided under *Crest/Inputs/Flow*:

The *Crest/Inputs/Flow/Add Flow Map* shader writes a flow texture into the system. It assumes the x component of the flow velocity is packed into 0-1 range in the red channel, and the z component of the velocity is packed into 0-1 range in the green channel. The shader reads the values, subtracts 0.5, and multiplies them by the provided scale value on the shader. The process of adding ocean inputs is demonstrated in Fig. 7.1.

7.8 Albedo

Preview

This feature is currently in preview.

7.8.1 Overview

The Albedo feature allows a colour layer to be composited on top of the water surface. This is useful for projecting colour onto the surface.

This is somewhat similar to decals, except the colour only affects the water.

HDRP has a Decal Projector feature that works with the water, and the effect is more configurable and may be preferred over this feature. When using this feature be sure to enable Affects Transparent.

URP 2022 has a decal system but it does not support transparent surfaces like water.

7.8.2 User Inputs

Note: Inputs only execute the first shader pass (pass zero). It is recommended to use unlit shader templates or unlit *Shader Graph* (URP only) if not using one of ours. Shaders provided by *Unity* generally will not work as their primary pass is not zero - even for unlit shaders.

Any geometry or particle system can add colour to the water. It will be projected from a top down perspective onto the water surface.

To tag GameObjects to render onto the water, attach the *RegisterAlbedoInput* component.

CHAPTER

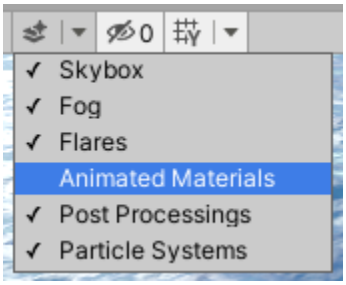
EIGHT

WAVE CONDITIONS

The following sections describe how to define the wave conditions.

Tip: It is useful to see the animated ocean surface while tweaking the wave conditions.

By default, the update speed is intentionally throttled by Unity to save power when in edit mode. To enable real-time update, enable *Animated Materials* in the Scene View toggles:



See the Unity Documentation for more information.

8.1 Wave Systems

The *ShapeFFT* component is used to generate waves in Crest.

For advanced situations where a high level of control is required over the wave shape, the *ShapeGerstner* component can be used to add specific wave components.

8.2 Authoring

To add waves, add the *ShapeFFT* component to a GameObject (comparison of the two options above).

The appearance and shape of the waves is determined by a *Wave Spectrum*. A default wave spectrum will be created if none is specified. To author wave conditions, click the *Create Asset* button next to the *Spectrum* field. The resulting spectrum can then be edited by expanding this field.

The spectrum can be freely edited in Edit mode, and is locked by default in Play mode to save evaluating the spectrum every frame (this optimisation can be disabled using the *Spectrum Fixed At Runtime* toggle). The spectrum has sliders for each wavelength to control contribution of different scales of waves. To control the contribution of 2m wavelengths, use the slider labelled '2'. Note that the wind speed may need to be increased on the *OceanRenderer* component in order for large wavelengths to be visible.

There is also control over how aligned waves are to the wind direction. This is controlled via the *Wind Turbulence* control on the *ShapeFFT* component.

Another key control is the *Chop* parameter which scales the horizontal displacement. Higher chop gives crisper wave crests but can result in self-intersections or 'inversions' if set too high, so it needs to be balanced.

To aid in tweaking the spectrum, we provide a standard empirical wave spectrum model from the literature, called the 'Pierson-Moskowitz' model. To apply this model to a spectrum, select it in the *Empirical Spectra* section of the spectrum editor which will lock the spectrum to this model. The model can be disabled afterwards which will unlock the spectrum power sliders for hand tweaking.

Tip: Notice how the empirical spectrum places the power slider handles along a line. This is typical of real world wave conditions which will have linear power spectrums on average. However actual conditions can vary significantly based on wind conditions, land masses, etc, and we encourage experimentation to obtain visually interesting wave conditions, or conditions that work best for gameplay.

Together these controls give the flexibility to express the great variation one can observe in real world seascapes.

8.3 Wave Splines

Preview

This feature is in preview and may change in the future.

---

https://www.youtube.com/watch?v=JRzPcUP5aaA

Fig. 8.1: Wave Splines

Wave Splines allow flexible and fast authoring of how waves manifest in the world. A couple of use cases are demonstrated in the video above.

If the *Spline* component is attached to the same GameObject as a *ShapeFFT* component, the waves will be generated along the spline. This allows for quick experimentation with placing and orienting waves in different areas of the environment.

The *Spline* component can also be combined with the *RegisterHeightInput* to make the water level follow the spline, and with the *RegisterFlowInput* to make water move along the spline.

8.4 Custom Shader

Shape components can receive input from a *Mesh Renderer* which allows for a custom shader. It is recommended to use an upwards facing quad.

The *Crest/Inputs/Shape Waves/Sample Spectrum* shader is provided to sample from the spectrum using a texture. The RG channels are the wave direction and together they make the magnitude. The values are 0-1 where 0.5 is zero magnitude (ie no waves).

CHAPTER

NINE

SHORELINES AND SHALLOWS

Changed in version 4.9: *Layer Names* (String[]) has changed to *Layers* (LayerMask).

Crest requires water depth information to attenuate large waves in shallow water, to generate foam near shorelines, and to provide shallow water shading. The way this information is typically generated is through the *OceanDepthCache* component, which takes one or more layers, and renders everything in those layers (and within its bounds) from a top-down orthographic view to generate a heightfield for the seabed. These layers could contain the render geometry/terrains, or it could be geometry that is placed in a non-rendered layer that serves only to populate the depth cache. By default this generation is done at run-time during startup, but the component exposes other options such as generating offline and saving to an asset, or rendering on demand.

The seabed affects the wave simulation in a physical way - the rule of thumb is *waves will be affected by the seabed when the water depth is less than half of their wavelength*. So for example when the water is 250m deep, this will start to dampen 500m wavelengths from the spectrum, so it is recommended that the seabed drop down to at least 500m away from islands so that there is a smooth transition between shallow and deep water without a 'step' in the sea floor which appears as a discontinuity in the surface waves and/or a line of foam. Alternatively, there is *Shallows Max Depth* on the *Sim Settings Animated Waves* asset which smooths the attenuation to a provided maximum depth where waves will be at full strength.

9.1 Setup

https://www.youtube.com/watch?v=jcmqUlboTUk

Fig. 9.1: Depth Cache usage and setup

One way to inform *Crest* of the seabed is to attach the *RegisterSeaFloorDepthInput* component. *Crest* will record the height of these objects every frame, so they can be dynamic.

The *main.unity* example scene has an example of a cache set up around the island. The cache GameObject is called *IslandDepthCache* and has a *OceanDepthCache* component attached. The following are the key points of its configuration:

- The transform position X and Z are centered over the island
- The transform position y value is set to the sea level
- The transform scale is set to 540 which sets the size of the cache. If gizmos are visible and the cache is selected, the area is demarcated with a white rectangle.
- The *Camera Max Terrain Height* is the max height of any surfaces above the sea level that will render into the cache. If gizmos are visible and the cache is selected, this cutoff is visualised as a translucent gray rectangle.
- The *Layers* field contains the layer that the island is assigned to (*Terrain* in our project). Only objects in these layer(s) will render into the cache.

- Both the transform scale (white rectangle) and the *Layers* property determine what will be rendered into the cache.

By default the cache is populated in the *Start()* function. It can instead be configured to populate from script by setting the *Refresh Mode* to *On Demand* and calling the *PopulateCache()* method on the component from script.

Once populated the cache contents can be saved to disk by clicking the *Save cache to file* button that will appear in the Inspector in play mode. Once saved, the *Type* field can be set to *Baked* and the saved data can be assigned to the *Saved Cache* field.

9.2 Shoreline Waves

Modelling realistic shoreline waves efficiently is a challenging open problem. We discuss further and make suggestions on how to set up shorelines with *Crest* in the following video.

https://www.youtube.com/watch?v=Y7ny8pKzWMk

Fig. 9.2: Tweaking Shorelines

9.3 Troubleshooting

Crest runs validation on the depth caches - look for warnings/errors in the Inspector, and in the log at run-time, where many issues will be highlighted.

To inspect the contents of the cache, look for a child GameObject parented below the cache with the name prefix *Draw_*. It will have a material with a *Texture* property. By double clicking the icon to the right of this field, one can inspect the contents of the cache. The cache will appear black for dry land and red for water that is at least 1m deep.

CHAPTER

TEN

OCEANS, RIVERS AND LAKES

Preview

The features described in this section are in preview and may evolve in future versions.

10.1 Oceans

By default Crest generates an infinite body of water at a fixed sea level, suitable for oceans and very large lakes.

10.2 Lakes

Crest can be configured to efficiently generate smaller bodies of water, using the following mechanisms.

- The waves can be generated in a limited area - see the *Wave Splines* section.
- The *WaterBody* component, if present, marks areas of the scene where water should be present. It can be created by attaching this component to a GameObject and setting the X/Z scale to set the size of the water body. If gizmos are enabled an outline showing the size will be drawn in the Scene View.
- The *WaterBody* component turns off tiles that do not overlap the desired area. The *Clip Surface* feature can be used to precisely remove any remaining water outside the intended area. Additionally, the clipping system can be configured to clip everything by default, and then areas can be defined where water should be included. See the *Clip Surface* section.
- If the lake altitude differs from the global sea level, create a spline that covers the area of the lake and attach the *RegisterHeightInput* component which will set the water level to match the spline (or click the *Set Height* button in the *Spline* inspector). It is recommended to cover a larger area than the lake itself, to give a protective margin against LOD effects in the distance.

Example

The *LakesAndRivers.unity* scene contains an example of a lake connected by a river.

Another advantage of the *WaterBody* component is it allows an optional override material to be provided, to change the appearance of the water. Since this feature cannot be applied partially to an ocean tile, and an ocean tile can overlap two water bodies, this feature does not work well with bordering water bodies. If you use this feature and want to still have an ocean, then disable *Water Body Culling* on the *Ocean Renderer*.

10.3 Rivers

Splines can also be used to create rivers, by creating a spline at the water surface of the river, and attaching the following components:

- *RegisterHeightInput* can be used to set the water level to match the spline.
- *RegisterFlowInput* can be used to make the water move along the spline.
- *ShapeFFT* can be used to generate waves that propagate along the river.

The *Add Feature* section of the *Spline* inspector has helper buttons to quickly add these components.

Example

The *LakesAndRivers.unity* scene contains an example of a river connecting two lakes.

CHAPTER

ELEVEN

COLLISION SHAPE FOR PHYSICS

The system has a few paths for computing information about the water surface such as height, displacement, flow and surface velocity. These paths are covered in the following subsections, and are configured on the *Animated Waves Sim Settings*, assigned to the OceanRenderer script, using the Collision Source dropdown.

The system supports sampling collision at different resolutions. The query functions have a parameter *Min Spatial Length* which is used to indicate how much detail is desired. Wavelengths smaller than half of this min spatial length will be excluded from consideration.

To simplify the code required to get the ocean height or other data from C#, two helpers are provided, *SampleHeightHelper* and *SampleFlowHelper*. Use of these is demonstrated in the example content.

Research

We use a technique called *Fixed Point Iteration* to calculate the water height. We gave a talk at GDC about this technique which may be useful to learn more: http://www.huwbowles.com/fpi-gdc-2016/.

The *Visualise Collision Area* debug component is useful for visualising the collision shape for comparison against the render surface. It draws debug line crosses in the Scene View around the position of the component.

11.1 Collision API Usage

The collision providers built intout our system perform queries asynchronously; queries are offloaded to the GPU or to spare CPU cores for processing. This has a few non-trivial impacts on how the query API must be used.

Firstly, queries need to be registered with an ID so that the results can be tracked and retrieved later. This ID needs to be globally unique, and therefore should be acquired by calling *GetHashCode()* on an object/component which will be guaranteed to be unique. A primary reason why *SampleHeightHelper* is useful is that it is an object in itself and there can pass its own ID, hiding this complexity from the user.

Important: Queries should only be made once per frame from an owner - querying a second time using the same ID will stomp over the last query points.

Secondly, even if only a one-time query of the height is needed, the query function should be called every frame until it indicates that the results were successfully retrieved. See *SampleHeightHelper* and its usages in the code - its *Sample()* function should be called until it returns true. Posting the query and polling for its result are done through the same function.

Finally due to the above properties, the number of query points posted from a particular owner should be kept consistent across frames. The helper classes always submit a fixed number of points this frame, so satisfy this criteria.

11.2 Compute Shape Queries (GPU)

This is the default and recommended choice for when a GPU is present. Query positions are uploaded to a compute shader which then samples the ocean data and returns the desired results. The result of the query accurately tracks the height of the surface, including all wave components and depth caches and other Crest features.

11.3 Baked FFT Data (CPU)

Preview

This feature is in preview and may change in the future.

In scenarios where a GPU is not present such as for headless servers, a CPU option is available.

To use this feature, select a *Shape FFT* component that is generating the waves in a scene and enable the **Enable Baked Collision**. Next configure the following options:

- **Time Resolution** - Frames per second of baked data. Larger values may help the collision track the surface closely at the cost of more frames and increase baked data size.
- **Smallest Wavelength Required** - Smallest wavelength required in collision. To preview the effect of this, disable power sliders in spectrum for smaller values than this number. Smaller values require more resolution and increase baked data size.
- **Time Loop Length** - FFT waves will loop with a period of this many seconds. Smaller values decrease data size but can make waves visibly repetitive.

Next click **Bake to asset and assign to current settings** and select a path and filename for the result. After the bake completes the current active *Animated Waves Sim Settings* will be configured to use this data.

Important: There are currently a few key limitations of this approach:

- Only a single set of waves from one *Shape FFT* component is supported. This collision does not support multiple sets of waves.
- The *Depth Cache* components are not supported. In order to get a one to one match between the visuals and the collision data, depth caches should not be used.
- Varying water levels such as rivers flowing down a gradient or lakes at different altitudes is not supported. This feature assumes a fixed sea level for the whole scene.

Sponsor

Sponsoring us will help increase our development bandwidth which could work towards solving the aforementioned limitations.

Trello Card

Sponsor Us

11.4 Gerstner Waves CPU (deprecated)

Deprecated

The Gerstner wave system in Crest is now deprecated. A CPU query path for the FFT waves is being worked on.

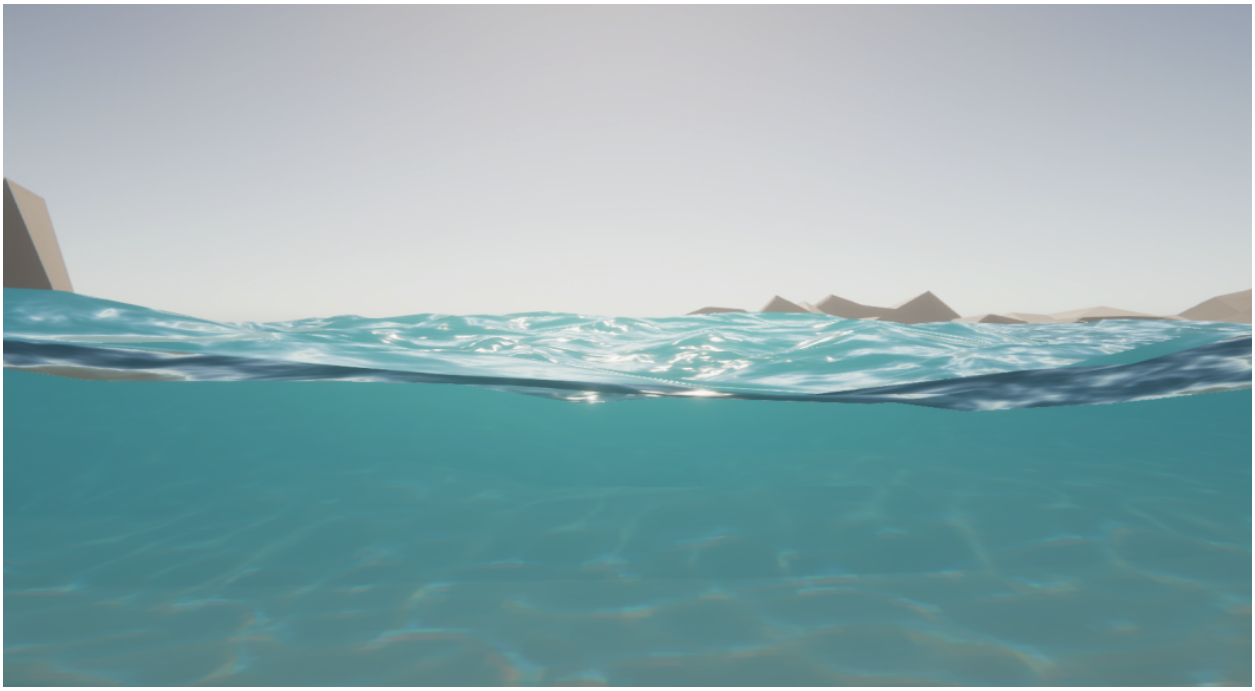
This collision option is serviced directly by the *GerstnerWavesBatched* component which implements the *ICollProvider* interface, check this interface to see functionality. This sums over all waves to compute displacements, normals, velocities, etc. In contrast to the displacement textures the horizontal range of this collision source is unlimited.

A drawback of this approach is the CPU performance cost of evaluating the waves. It also does not include wave attenuation from water depth or any custom rendered shape. A final limitation is the current system finds the first *GerstnerWavesBatched* component in the scene which may or may not be the correct one. The system does not support cross blending of multiple scripts.

CHAPTER

TWELVE

UNDERWATER



Crest supports seamless transitions above/below water. It can also have a meniscus which renders a subtle line at the intersection between the camera lens and the water to visually help the transition. This is demonstrated in the *main.unity* scene in the example content. The ocean in this scene uses the material *Ocean-Underwater.mat* which enables rendering the underside of the surface.

Out-scattering is provided as an example script which reduces environmental lighting with depth underwater. See the *UnderwaterEnvironmentalLighting* component.

For performance reasons, the underwater effect is disabled if the viewpoint is not underwater. Only the camera rendering the ocean surface will be used.

Tip: Use opaque or alpha test materials for underwater surfaces. Transparent materials may not render correctly underwater. See *Transparent Object Underwater* for possible workarounds.

12.1 Underwater Renderer

Note: You can enable/disable rendering in the scene view by toggling fog in the scene view control bar.

The *Underwater Renderer* component executes a fullscreen underwater effect between the transparent pass and post-processing pass.

It is similar to a post-processing effect, but has the benefit of allowing other renderers to execute after it and still receive post-processing. An example is to add underwater fog correctly to semi-transparent objects.

This is the current underwater solution used for the example scenes, and is the simplest to setup.

12.1.1 Setup

BIRP

- Configure the ocean material for underwater rendering. In the *Underwater* section of the material params, ensure *Enabled* is turned on and *Cull Mode* is set to *Off* so that the underside of the ocean surface renders. See *Ocean-Underwater.mat* for an example.

HDRP

- Configure the ocean material for underwater rendering. Ensure that *Double-Sided* is enabled under *Surface Options* on the ocean material so that the underside of the ocean surface renders. See *Ocean-Underwater.mat* for an example.

URP

- Configure the ocean material for underwater rendering. In the *Underwater* section of the material params, ensure *Enabled* is turned on and *Cull Mode* is set to *Off* so that the underside of the ocean surface renders. See *Ocean-Underwater.mat* for an example.
- Add the *Underwater Renderer* component to your camera game object.

12.1.2 Parameters

- **Mode:** How the underwater effect (and ocean surface) is rendered:
  - **Full-Screen:** Full screen effect.
  - **Portal:** Renders the underwater effect and ocean surface from the geometry's front-face and behind it.
  - **Volume:** Renders the underwater effect and ocean surface from the geometry's front-face to its back-face.
  - **Volume (Fly-Through):** Renders the underwater effect and ocean surface from the geometry's front-face to its back-face - even from within the volume.
- **Depth Fog Density Factor:** Reduces the underwater depth fog density by a factor. Useful to reduce the intensity of the fog independently from the ocean surface.

12.1.3 Detecting Above or Below Water

The *OceanRenderer* component has the *ViewerHeightAboveWater* property which can be accessed with `OceanRenderer.Instance.ViewerHeightAboveWater`. It will return the signed height from the ocean surface of the camera rendering the ocean. Internally this uses the *SampleHeightHelper* class which can be found in *SamplingHelpers.cs*.

There is also the *OceanSampleHeightEvents* example component (requires example content to be imported) which uses UnityEvents to provide a scriptless approach to triggering changes. Simply attach it to a game object, and it will invoke a UnityEvent when the attached game object is above or below the ocean surface once per state change. A common use case is to use it to trigger different audio when above or below the surface.

12.1.4 Portals & Volumes

Preview

This feature is in preview and may change in the future.

The underwater effect can be rendered from a provided mesh which will effectively become a portal (2D) or volume (3D). Change the *Mode* property to one of your choosing and set the *Volume Geometry* to a *Mesh Filter* (it will use its transform). This feature also clips the ocean surface to match. A common use case would be a window on a boat.

12.1.5 Underwater Shader API

Preview

This feature is in preview and may change in the future.

The underwater effect uses opaque depth and thus will not render correctly for transparent objects. Too much fog will be applied as it is as if the transparent object does not exist.

The most effective approach is to render the transparent objects after the underwater effect and apply the underwater effect as part of the shader for the transparent object (basically the same way Unity fog is applied).

The *Shader API* needs to be enabled on the *Underwater Renderer* (located under the *Shader API* heading).

BIRP

Once the *Shader API* is enabled, the underwater effect will be rendered before the transparent pass instead of after it, and the global shader properties will be populated. This means that when a transparent object is rendered, it will already have underwater fog behind it. It is then just a matter of applying the underwater fog to the transparent object.

Example

We have an example *Surface Shader* which you can use as a reference:
Crest/Crest-Examples/Shared/Shaders/ExampleUnderwaterTransparentSurfaceShader.shader

Furthermore, you can view the shader in action in the *Transparent Object Underwater* example in the *Examples* scene.

Setting up a shader can be broken down to the following:

1. Including our includes file:
   Crest/Crest/Shaders/Underwater/UnderwaterEffectIncludes.hlsl
2. Adding optional keywords (see example shader)
3. Use the *CrestApplyUnderwaterFog* function to apply the fog to the final color

Here is the important part from *ExampleUnderwaterTransparentSurfaceShader.shader*:

```
float2 positionNDC = IN.screenPos.xy / IN.screenPos.w;
float deviceDepth = IN.screenPos.z / IN.screenPos.w;

if (!CrestApplyUnderwaterFog(positionNDC, IN.worldPos, deviceDepth, _FogMultiplier,␣
↪color.rgb))
{
   UNITY_APPLY_FOG(IN.fogCoord, color);
}
```

HDRP

Once the *Shader API* is enabled, any transparent object in the correct layer and using a modified shader (more on that later) will have its above water pixels rendered in the transparent pass and below water pixels rendered after the underwater pass.

In a perfect world, we would render the underwater pass before the transparent pass, and then apply the underwater effect to the final color of each transparent object using the *CrestNodeApplyUnderWaterFog* node. But *Shader Graph* does not allow modification of the final color.

The workaround is in the example node *CrestNodeApplyUnderwaterFogExample*. This node uses the *CrestNodeApplyUnderWaterFog* node and does a few things to get around this problem:

- Apply the underwater effect only to the Emission input to bypass *Unity*'s lighting
- Reduce the alpha and the color by distance from the camera

The end result is that the effect is inconsistent with the underwater pass. Despite that we believe it is a decent enough approximation until *Unity* improves this area.

Example

We have an example *Surface Shader* which you can use as a reference:
Crest/Crest-Examples/Shared/Shaders/LitTransparentWithUnderwaterFog.shadergraph

Furthermore, you can view the shader in action in the *Transparent Object Underwater* example in the *Examples* scene.

Setting up a graph can be broken down to the following:

1. Add optional keywords (see example graph)
2. Add the *CrestNodeApplyUnderwaterFogExample* node
3. Connect *Fogged Color* (and alpha) and *Fogged Emission* outputs to the Master Stack
4. Multiply *Factor* output with any properties except *Ambient Occlusion*
5. Enable *Alpha Clipping*

For best results using the Lit Shader graph:

- Keep *Preserve Specular Lighting* disabled as this will cause the object to be visible from any distance
- Do not enable *Receive Fog* as this will write over the emission and thus underwater fog
- Be mindful of what features on the *Shader Graph* you enable as it might affect the underwater fog

URP

Once the *Shader API* is enabled, any transparent object in the correct layer and using a modified shader (more on that later) will have its above water pixels rendered in the transparent pass and below water pixels rendered after the underwater pass.

In a perfect world, we would render the underwater pass before the transparent pass, and then apply the underwater effect to the final color of each transparent object using the *CrestNodeApplyUnderWaterFog* node. But *Shader Graph* does not allow modification of the final color.

The workaround is in the example node *CrestNodeApplyUnderwaterFogExample*. This node uses the *CrestNodeApplyUnderWaterFog* node and does a few things to get around this problem:

- Apply the underwater effect only to the Emission input to bypass *Unity*'s lighting
- Reduce the alpha and the color by distance from the camera

The end result is that the effect is inconsistent with the underwater pass. Despite that we believe it is a decent enough approximation until *Unity* improves this area.

Example

We have an example *Surface Shader* which you can use as a reference:
Crest/Crest-Examples/Shared/Shaders/LitTransparentWithUnderwaterFog.shadergraph

Furthermore, you can view the shader in action in the *Transparent Object Underwater* example in the *Examples* scene.

Setting up a graph can be broken down to the following:

1. Add optional keywords (see example graph)
2. Add the *CrestNodeApplyUnderwaterFogExample* node
3. Connect *Fogged Color* (and alpha) and *Fogged Emission* outputs to the Master Stack
4. Multiply *Factor* output with any properties except *Ambient Occlusion*
5. Enable *Alpha Clipping*

12.2 Underwater Curtain *BIRP URP*

Deprecated

The *Underwater Curtain* will be removed in a future Crest version. It has been replaced by the *Underwater Renderer*.

12.2.1 Setup

- Configure the ocean material for underwater rendering. In the *Underwater* section of the material params, ensure *Enabled* is turned on and *Cull Mode* is set to *Off* so that the underside of the ocean surface renders. See *Ocean-Underwater.mat* for an example.
- Place *UnderWaterCurtainGeom* and *UnderWaterMeniscus* prefabs under the camera (with cleared transform).

12.3 Underwater Post-Process *HDRP*

Deprecated

The *Underwater Post-Process* will be removed in a future Crest version. It has been replaced by the *Underwater Renderer*.

Renders the underwater effect at the beginning of the post-processing stack.

12.3.1 Setup

Steps to set up underwater:

1. Ensure Crest is properly set up and working before proceeding.
2. Enable Custom Pass on the HDRP Asset and ensure that Custom pass on the camera's Frame Settings is not disabled.
3. Add the custom post-process (*Crest.UnderwaterPostProcessHDRP*) to the *Before TAA* list. See the Custom Post Process documentation.
4. Add the *Crest/Underwater* Volume Component.
    - Please learn how to use the *Volume Framework* before proceeding as covering this is beyond the scope of our documentation:

      https://www.youtube.com/watch?v=vczkfjLoPf8

      Fig. 12.1: Adding Volumes to HDRP (Tutorial)

5. Configure the ocean material for underwater rendering. Ensure that *Double-Sided* is enabled under *Surface Options* on the ocean material so that the underside of the ocean surface renders. See *Ocean-Underwater.mat* for an example.

CHAPTER

THIRTEEN

TIME CONTROL

By default, *Crest* uses the current game time given by Time.time when simulating and rendering the water. In some situations it is useful to control this time, such as an in-game pause or to synchronise wave conditions over a network. This is achieved through what we call *TimeProviders*, and a few use cases are described below.

Note: The *Dynamic Waves* simulation must progress frame by frame and can not be set to use a specific time, and also cannot be synchronised accurately over a network.

13.1 Supporting Pause

One way to pause time is to set Time.timeScale to 0. In many cases it is desirable to leave Time.timeScale untouched so that animations continue to play, and instead pause only the water. To achieve this, attach a *TimeProviderCustom* component to a GameObject and assign it to the *Time Provider* parameter on the *OceanRenderer* component. Then time can be paused by setting the *_paused* variable on the *TimeProviderCustom* component to *false*.

The *TimeProviderCustom* also allows driving any time to the system which may give more flexibility for specific use cases.

A final alternative option is to create a new class that implements the *ITimeProvider* interface and call *OceanRenderer.Instance.PushTimeProvider()* to apply it to the system.

13.2 Network Synchronisation

A requirement in networked games is to have a common sense of time across all clients. This can be specified using an offset between the clients Time.time and that of a server.

This is supported by attaching a *TimeProviderNetworked.cs* component to a GameObject, assigning it to the *Time Provider* parameter on the *OceanRenderer* component, and at run-time setting *TimeProviderNetworked.TimeOffsetToServer* to the time difference between the client and the server.

If using the Mirror network system, set this property to the network time offset.

13.3 Timelines and Cutscenes

One use case for this is for cutscenes/timelines when the waves conditions must be known in advance and repeatable. For this case you may attach a *Cutscene Time Provider* component to a GameObject and assign it to the *Ocean Renderer* component. This component will take the time from a Playable Director component which plays a cutscene Timeline. Alternatively, a *Time Provider Custom* component can be used to feed any time into the system, and this time value can be keyframed, giving complete control over timing.

CHAPTER

FOURTEEN

OTHER FEATURES

14.1 Floating Origin

Crest has support for 'floating origin' functionality, based on code from the *Unity Community Wiki*. See the original Floating Origin wiki page for an overview and original code.

By default the *FloatingOrigin* script will call *FindObjectsOfType()* for a few different component types, which is a notoriously expensive operation. It is possible to provide custom lists of components to the "override" fields, either by hand or programmatically, to avoid searching the entire scene(s) for the components. Managing these lists at run-time is left to the user.

14.2 Buoyancy

Note: Buoyancy physics for boats is not a core focus of *Crest*. For a professional physics solution we recommend the DWP2 (Dynamic Water Physics 2) asset which is compatible with *Crest*.

With that said, we do provide rudimentary physics scripts.

SimpleFloatingObject is a simple buoyancy script that attempts to match the object position and rotation with the surface height and normal. This can work well enough for small water craft that don't need perfect floating behaviour, or floating objects such as buoys, barrels, etc.

BoatProbes is a more advanced implementation that computes buoyancy forces at a number of *ForcePoints* and uses these to apply force and torque to the object. This gives more accurate results at the cost of more queries.

BoatAlignNormal is a rudimentary boat physics emulator that attaches an engine and rudder to *SimpleFloatingObject*. It is not recommended for cases where high animation quality is required.

14.2.1 Adding boats

Setting up a boat with physics can be a dark art. The authors recommend duplicating and modifying one of the existing boat prefabs, and proceeding slowly and carefully as follows:

1. Pick an existing boat to replace. Only use *BoatAlignNormal* if good floating behaviour is not important, as mentioned above. The best choice is usually *BoatProbes*.
2. Duplicate the prefab of the one you want to replace, such as *crest/Assets/Crest/Crest-Examples/BoatDev/Data/BoatProbes.prefab*
3. Remove the render meshes from the prefab, and add the render mesh for your boat. We recommend lining up the meshes roughly.

4. Switch out the collision shape as desired. Some people report issues if the are multiple overlapping physics collision primitives (or multiple rigidbodies which should never be the case). We recommend keeping things as simple as possible and using only one collider if possible.
5. We recommend placing the render mesh so its approximate center of mass matches the center of the collider and is at the center of the boat transform. Put differently, we usually try to eliminate complex hierarchies or having nested non-zero'd transforms whenever possible within the boat hierarchy, at least on or above physical parts.
6. If you have followed these steps you will have a new boat visual mesh and collider, with the old rigidbody and boat script. You can then modify the physics settings to move the behaviour towards how you want it to be.
7. The mass and drag settings on the boat scripts and rigdibody help to give a feeling of weight.
8. Set the boat dimension:
    - BoatProbes: Set the *Min Spatial Length* param to the width of the boat.
    - BoatAlignNormal: Set the *Boat Width* and *Boat Length* params to the width and length of the boat.
    - If, even after experimenting with the mass and drag, the boat is responding too much to small waves, increase these parameters (try doubling or quadrupling at first and then compensate).
9. There are power settings for engine turning which also help to give a feeling of weight.
10. The dynamic wave interaction is driven by the object in the boat hierarchy called *SphereWaterInteraction*. It can be scaled to match the dimensions of the boat. The *Weight* param controls the strength of the interaction.

The above steps should maintain a working boat throughout - we recommend testing after each step to catch issues early.

CHAPTER

FIFTEEN

RENDERING

15.1 Transparency

Crest is rendered in a standard way for water shaders - in the transparent pass and refracts the scene. The refraction is implemented by sampling the camera's colour texture which has opaque surfaces only. It writes to the depth buffer during rendering to ensure overlapping waves are sorted correctly to the camera. The rendering of other transparent objects depends on the case, see headings below. Knowledge of render pipeline features, rendering order and shaders is required to solving incompatibilities.

15.1.1 Transparent Object In Front Of Ocean Surface

Normal transparent shaders should blend correctly in front of the water surface. However this will not work correctly for refractive objects. *Crest* will not be available in the camera's colour texture when other refractive objects sample from it, as the camera colour texture will only contain opaque surfaces. The end result is *Crest* not being visible behind the refractive object.

15.1.2 Transparent Object Behind The Ocean Surface

Alpha blend and refractive shaders will not render behind the water surface. Other transparent objects will not be part of the camera's colour texture when *Crest* samples from it. The end result is transparent objects not being visible behind *Crest*.

On the other hand, alpha test / alpha cutout shaders are effectively opaque from a rendering point of view and may be usable in some scenarios.

15.1.3 Transparent Object Underwater

Note: See the *Underwater Shader API* (in preview) for a more accurate solution.

This is tricky because the underwater effect uses the opaque scene depths in order to render the water fog, which will not include transparents.

The following only applies to the *Underwater Renderer*.

BIRP

Transparents will need to be rendered after the underwater effect. The underwater effect is rendered at the CameraEvent.AfterForwardAlpha event. They can be rendered after the underwater effect using Command Buffers. Transparents rendered after the underwater effect will not have the underwater water fog shading applied to them. The effect

of the fog either needs to be faked by simply ramping the opacity down to 0 based on distance from the camera, or the water fog shader code needs to be included and called from the transparent shader.

HDRP

The Submarine example scene demonstrates an underwater transparent effect - the bubbles from the propellors when the submarine is in motion. This effect is from the *Bubbles Propellor* GameObject, which is assigned a specific layer *TransparentFX*. The particles need to be rendered between the underwater and post-processing passes which is achieved using a *Custom Pass Volume* component attached to the *CustomPassForUnderwaterParticles* GameObject. It is configured to render the *TransparentFX* layer in the *Before Post Process* injection point with a priority of "-1" (which orders it to render after the underwater pass). Transparents rendered after the underwater effect will not have the underwater water fog shading applied to them. The effect of the fog either needs to be faked by simply ramping the opacity down to 0 based on distance from the camera, or the water fog shader code needs to be included and called from the transparent shader. The shader *UnderwaterEffectPassHDRP.shader* is a good reference for calculating the underwater effect. This will require various parameters on the shader like fog density and others.

URP

Transparents will need to be rendered after the underwater effect. The underwater effect is rendered at the *BeforeRenderingPostProcessing* event. They can be rendered after the underwater effect using the Render Objects Render Feature set to *BeforeRenderingPostProcessing*. Transparents rendered after the underwater effect will not have the underwater water fog shading applied to them. The effect of the fog either needs to be faked by simply ramping the opacity down to 0 based on distance from the camera, or the water fog shader code needs to be included and called from the transparent shader.

15.2 Render Order *BIRP URP*

A typical render order for a frame is the following:

- Opaque geometry is rendered, writes to opaque depth buffer.
- Sky is rendered, probably at zfar with depth test enabled so it only renders outside the opaque surfaces.
- Frame colours and depth are copied out for use later in postprocessing.
- Ocean renders early in the transparent queue (queue = 2510).
  - Queue = Geometry+510 *BIRP*. Queue = Transparent-100 *URP*.
  - It samples the postprocessing colours and depths, to do refraction.
  - It reads and writes from the frame depth buffer, to ensure waves are sorted correctly.
  - It stomps over sky - sky is at zfar and will be fully fogged/obscured by the water volume.
- Particles and alpha render. If they have depth test enabled, they will clip against the surface.
- Postprocessing runs with the postprocessing depth and colours.
  - If enabled, underwater postprocess constructs a screenspace mask for the ocean and uses it to draw the underwater effect over the screen.

CHAPTER

SIXTEEN

PERFORMANCE

The foundation of *Crest* is architected for performance from the ground up with an innovative LOD system. It is tweaked to achieve a good balance between quality and performance in the general case, but getting the most out of the system requires tweaking the parameters for the particular use case. These are documented below.

16.1 Quality parameters

These are available for tweaking out of the box and should be explored on every project:

- See *Ocean Construction Parameters* for parameters that directly control how much detail is in the ocean, and therefore the work required to update and render it. These are the primary quality settings from a performance point of view.
- The ocean shader has accrued a number of features and has become a reasonably heavy shader. Where possible these are on toggles and can be disabled, which will help the rendering cost (see *Material Parameters*). A potential idea would be to change materials on the fly from script, for example to switch to a deep water material when out at sea to avoid shallow water calculations
- Our wave system uses an inefficient approach to generate the waves to avoid an incompatibility in older hardware. If you are shipping on a limited set of hardware which you can test the waves on, you may try disabling the *Ping pong combine* option in the *Animated Wave Settings* asset.

CHAPTER

SEVENTEEN

TECHNICAL DOCUMENTATION

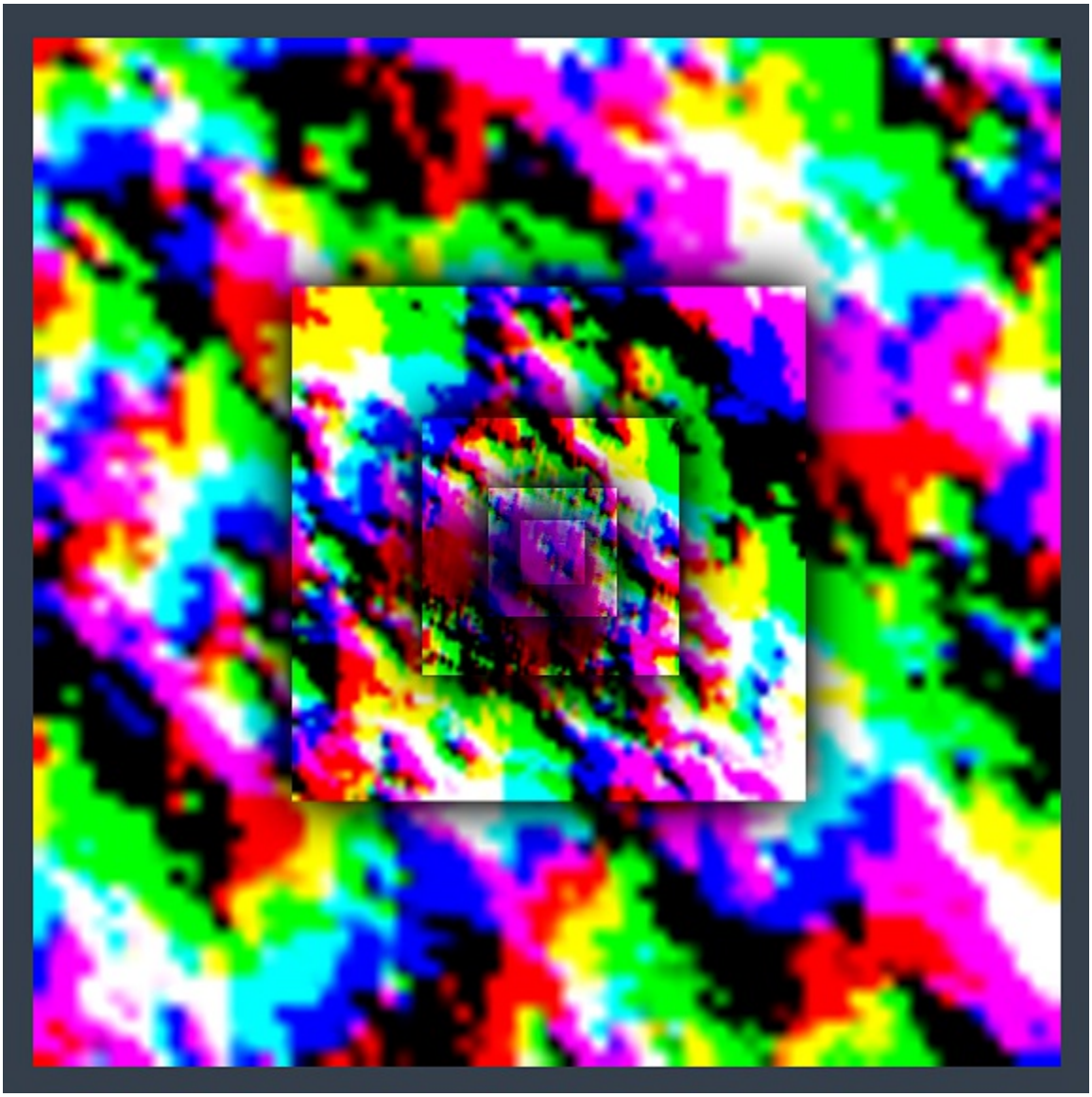
We have published details of the algorithms and approaches we use. See the following publications:

- Crest: *Novel Ocean Rendering Techniques in an Open Source Framework*, Advances in Real-Time Rendering in Games, ACM SIGGRAPH 2017 courses http://advances.realtimerendering.com/s2017/index.html
- *Multi-resolution Ocean Rendering in* Crest Ocean System, Advances in Real-Time Rendering in Games, ACM SIGGRAPH 2019 courses http://advances.realtimerendering.com/s2019/index.htm

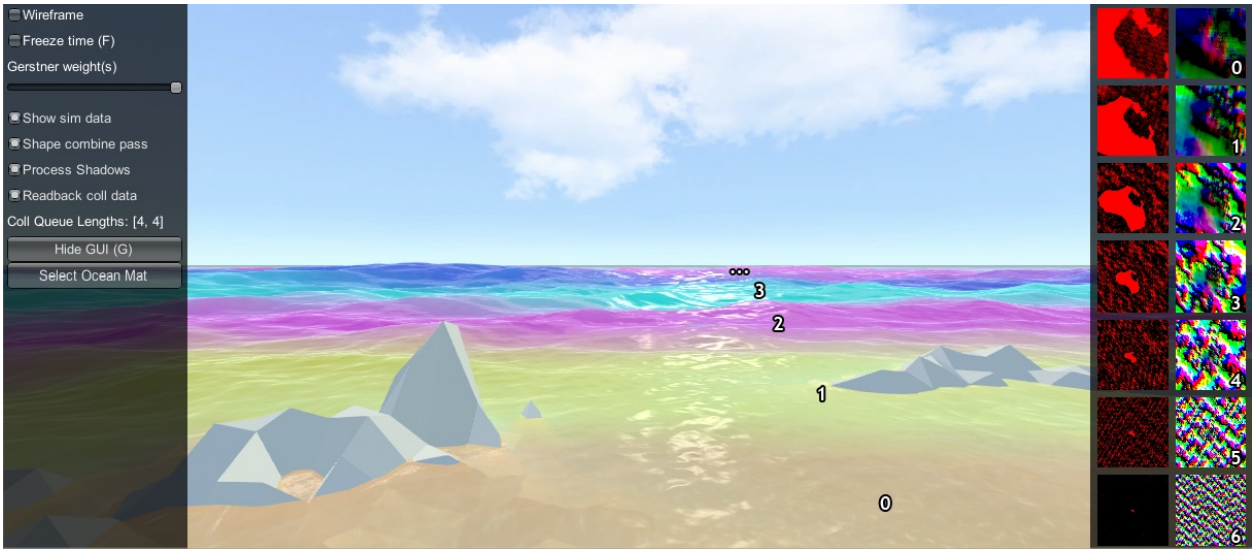
17.1 Core Data Structure

The backbone of *Crest* is an efficient Level Of Detail (LOD) representation for data that drives the rendering, such as surface shape/displacements, foam values, shadowing data, water depth, and others. This data is stored in a multi-resolution format, namely cascaded textures that are centered at the viewer. This data is generated and then sampled when the ocean surface geometry is rendered. This is all done on the GPU using a command buffer constructed each frame by *BuildCommandBuffer*.

Let's study one of the LOD data types in more detail. The surface shape is generated by the Animated Waves LOD Data, which maintains a set of *displacement textures* which describe the surface shape. A top down view of these textures laid out in the world looks as follows:

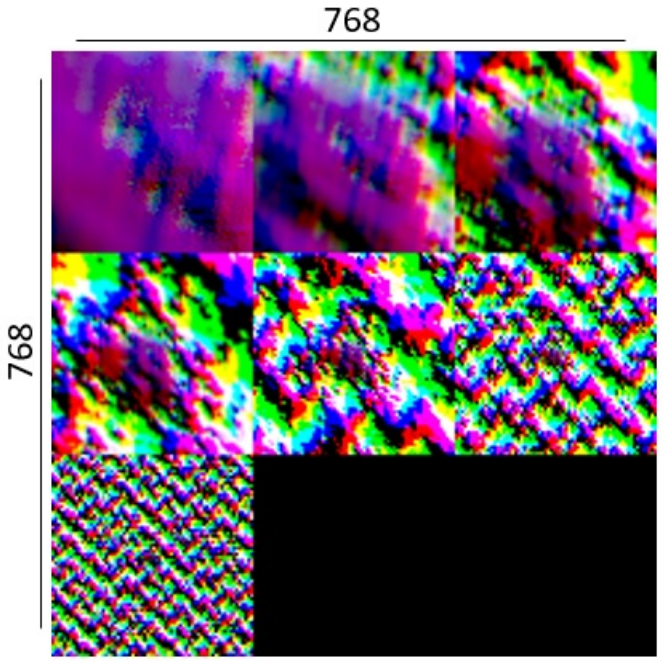


Each LOD is the same resolution (256x256 here), configured on the *OceanRenderer* script. In this example the largest LOD covers a large area (4km squared), and the most detail LOD provides plenty of resolution close to the viewer. These textures are visualised in the Debug GUI on the right hand side of the screen:



In the above screenshot the foam data is also visualised (red textures), and the scale of each LOD is clearly visible by looking at the data contained within. In the rendering each LOD is given a false colour which shows how the LODs are arranged around the viewer and how they are scaled. Notice also the smooth blend between LODs - LOD data is always interpolated using this blend factor so that there are never pops are hard edges between different resolutions.

In this example the LODs cover a large area in the world with a very modest amount of data. To put this in perspective, the entire LOD chain in this case could be packed into a small texel area:



A final feature of the LOD system is that the LODs change scale with the viewpoint. From an elevated perspective, horizontal range is more important than fine wave details, and the opposite is true when near the surface. The *OceanRenderer* has min and max scale settings to set limits on this dynamic range.

When rendering the ocean, the various LOD data are sample for each vert and the vert is displaced. This means that the data is carried with the waves away from its rest position. For some data like wave foam this is fine and desirable.

For other data such as the depth to the ocean floor, this is not a quantity that should move around with the waves and this can currently cause issues, such as shallow water appearing to move with the waves as in #96.

17.2 Implementation Notes

On startup, the *OceanRenderer* script initialises the ocean system and asks the *OceanBuilder* script to build the ocean surface. As can be seen by inspecting the ocean at run-time, the surface is composed of concentric rings of geometry tiles. Each ring is given a different power of 2 scale.

At run-time, the ocean system updates its state in *LateUpdate*, after game state update and animation, etc. *OceanRenderer* updates before other scripts and first calculates a position and scale for the ocean. The ocean GameObject is placed at sea level under the viewer. A horizontal scale is computed for the ocean based on the viewer height, as well as a *_viewerAltitudeLevelAlpha* that captures where the camera is between the current scale and the next scale ($\times 2$), and allows a smooth transition between scales to be achieved.

Next any active ocean data are updated, such as animated waves, simulated foam, simulated waves, etc. The data can be visualised on screen if the *OceanDebugGUI* script from the example content is present in the scene, and if the *Show shape data* on screen toggle is enabled. As one of the ocean data types, the ocean shape is generated using an FFT and copied into the animated waves data. Each wave component is rendered into the shape LOD that is appropriate for the wavelength, to prevent over- or under- sampling and maximize efficiency. A final pass combines the shape results from the different FFT components together. Disable the *Shape combine pass* option on the *OceanDebugGUI* to see the shape contents before this pass.

Finally *BuildCommandBuffer* constructs a command buffer to execute the ocean update on the GPU early in the frame before the graphics queue starts. See the *BuildCommandBuffer* code for the update scheduling and logic.

The ocean geometry is rendered by Unity as part of the graphics queue, and uses the *Crest/Ocean* shader. The vertex shader snaps the verts to grid positions to make them stable. It then computes a *lodAlpha* which starts at 0 for the inside of the LOD and becomes 1 at the outer edge. It is computed from taxicab distance as noted in the course. This value is used to drive the vertex layout transition, to enable a seamless match between the two. The vertex shader then samples any required ocean data for the current and next LOD scales and uses *lodAlpha* to interpolate them for a smooth transition across displacement textures. Finally, it passes the LOD geometry scale and *lodAlpha* to the ocean fragment shader.

The fragment shader samples normal and foam maps at 2 different scales, both proportional to the current and next LOD scales, and then interpolates the result using *lodAlpha* for a smooth transition. It combines the normal map with surface normals computed directly from the displacement texture.

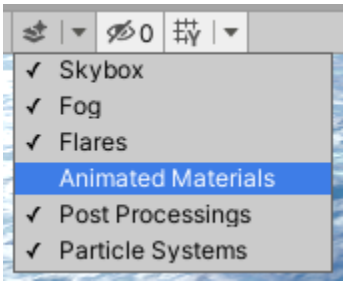
CHAPTER

EIGHTEEN

Q & A

Why does the ocean not update smoothly in edit mode?

By default, the update speed is intentionally throttled by Unity to save power when in edit mode. To enable real-time update, enable *Animated Materials* in the Scene View toggles:



See the Unity Documentation for more information.

Why aren't my prefab mode edits not reflected in the scene view?

Crest does not support running in prefab mode which means dirty state in prefab mode will not be reflected in the scene view. Save the prefab to see the changes.

Is *Crest* well suited for medium-to-low powered mobile devices?

Crest is built to be performant by design and has numerous quality/performance levers. However it is also built to be very flexible and powerful and as such can not compete with a minimal, mobile-centric ocean renderer such as the one in the *BoatAttack* project. Therefore we target *Crest* at PC/console platforms.

Which platforms does *Crest* support?

Testing occurs primarily on Windows.

We have users targeting the following platforms:

- Windows
- Mac
- Linux
- PS4 **
- XboxOne **
- Switch * **
- iOS/Android * ** *BIRP URP*
- Quest * ** *BIRP URP*

* Performance is a challenge on these platforms. Please see the previous question.
** We do not have access to these platforms ourselves.

Crest also supports VR/XR Multi-Pass and Single Pass Instanced rendering.

For additional platform notes, see Platform Support.

Is *Crest* well suited for localised bodies of water such as lakes?

Currently Crest is targeted towards large bodies of water. This area is being actively developed. Please see *Oceans, Rivers and Lakes* for current progress.

Can *Crest* work with multiplayer?

Yes, the animated waves are deterministic and easily synchronized. See discussion in #75. However, the dynamic wave sim is not synchronized over the network and can not currently be relied upon in networked situations.

For more information, see the following sections relevant to networking and server environments:

- *Network Synchronisation*
- *Baked FFT Data (CPU)*
- *Gerstner Waves CPU (deprecated)*

Errors are present in the log that report *Kernel 'xxx.yyy' not found*

Unity sometimes gets confused and needs assets reimported. This can be done by clicking the *Crest* root folder in the Project window and clicking *Reimport*. Alternatively the *Library* folder can be removed from the project root which will force all assets to reimport.

Can I push the ocean below the terrain?

Yes, this is demonstrated in Fig. 7.1.

Does *Crest* support multiple viewpoints?

Currently only a single ocean instance can be created, and only one viewpoint is supported at a time. We hope to support multiple simultaneous views in the future.

Can I sample the water height at a position from C#?

Yes, see SampleHeightHelper class in SamplingHelpers.cs. The OceanRenderer uses this helper to get the height of the viewer above the water, and makes this viewer height available via the ViewerHeightAboveWater property.

Can I trigger something when an object is above or under the ocean surface without any scripting knowledge?

Yes. Please see *Detecting Above or Below Water*.

Does Crest support orthographic projection?

Yes. Please see *Orthographic Projection*.

How do I disable underwater fog rendering in the scene view?

You can enable/disable rendering in the scene view by toggling fog in the scene view control bar.

Can the density of the fog in the water be reduced?

The density of the fog underwater can be controlled using the *Fog Density* parameter on the ocean material. This applies to both above water and underwater. The *Depth Fog Density Factor* on the *Underwater Renderer* can reduce the density of the fog for the underwater effect.

Does Crest support third party sky assets? *BIRP URP*

We have heard of Crest users using TrueSky, AzureSky. These may require some code to be inserted into the ocean shader - there is a comment referring to this, search Ocean.shader for 'Azure'.

Please see the Community Contributions section in our Wiki for some integration solutions.

Can I remove water from inside my boat?

Yes, this is referred to as 'clipping' and is covered in section *Clip Surface*.

How to implement a swimming character?

As far as we know, existing character controller assets which support swimming do not support waves (they require a volume for the water or physics mesh for the water surface). We have an efficient API to provide water heights, which the character controller could use instead of a physics volume. Please request support for custom water height providers to your favourite character controller asset dev.

Can I render transparent objects underwater?

See *Transparent Object Underwater*.

Can I render transparent objects in front of water?

See *Transparent Object In Front Of Ocean Surface*.

Can I render transparent objects behind the ocean surface?

See *Transparent Object Behind The Ocean Surface*.