The Technical Art of Sea of Thieves

Nigel Ang
Rare Ltd.
daang@microsoft.com

Andrew Catling
Rare Ltd.
ancatlin@microsoft.com

Francesco Cifariello Ciardi
Rare Ltd.
fracif@microsoft.com

Valentine Kozin
Rare Ltd.
valkozin@microsoft.com

ABSTRACT

Sea of Thieves posed a unique challenge - developing a stylised, open world game in Unreal Engine 4, a demanding and contemporary game engine focused on photo-realistic rendering. Our game contains a large number of dynamic elements and is designed to run on hardware ranging from integrated GPUs on a laptop, to the most powerful modern gaming PCs. Over the course of development, we have come up with a number of innovative techniques focused both on keeping an open world game like ours performant and visually appealing.

We introduced several techniques that we used to stylise and supplement the look of our FFT water implementation for the game's oceans. We also created a new cloud rendering and simulation system for this game, allowing for fast rendering of three-dimensional, art-directed cloudscapes without using expensive raymarching techniques.

To bring the world to life, we also developed other graphical features, including a physically-based system of rendering rope-and-pulley systems, our use of baking simulation data to textures and real-time surface fluid simulations to model incidental water behaviour on the GPU.

CCS CONCEPTS

• **Computing methodologies → Rasterization**; **Non-photorealistic rendering**; **Mesh geometry models**;

KEYWORDS

SIGGRAPH Talks, games, real-time, water, ropes, lightning, clouds, Unreal Engine

ACM Reference Format:
Nigel Ang, Andrew Catling, Francesco Cifariello Ciardi, and Valentine Kozin. 2018. The Technical Art of Sea of Thieves. In *Proceedings of SIGGRAPH '18 Talks, August 12-16, 2018, Vancouver, BC, Canada*, 2 pages. https://doi.org/10.1145/3214745.3214820

1 WATER RENDERING

1.1 Ocean

The underlying ocean water simulation is an implementation of the FFT technique described in [Tessendorf 2001].

The water colour is based on scattering approximations. We blend between a deep water colour and a sub-surface water colour based on a combination of view angle, sun direction and a wave peak mask. The wave peak mask is generated from the FFT choppiness vertex offsets. Where the choppiness offset is greater, this corresponds to wave peaks, which show more sub-surface due to shorter distance traveled by light through the water.

Foam is generated at wave peaks using the method described in the reference paper. It is also added around objects that intersect the water surface within a camera centered window using depth buffer comparisons. We progressively blur the result of the foam buffer with feedback to simulate the foam dispersing and to give us a softer mask, more in keeping with the style of the game. The resulting mask is blended with artist-authored textures to give a more stylized appearance to the foam.

Foam generation, dispersion and blending with the artist-authored textures is modified based on whether the water is calm, normal or stormy. Stormy water will have more foam to give the impression of the churn created by the more violent waves, whereas calm water will only show foam generation around intersecting objects.

We also apply an area specular highlight to allow for a large low sun reflection using the closest point on sphere approximation[Karis and Games 2013]. When looking at the water surface from below, we apply a Snell's Window effect to show the scene above the water surface.

1.2 Shallow Water Details

For shallow water features, such as water splashing on the deck of a ship, we use a GPU water surface simulation based on [Mei et al. 2007]. For water features such as waterfalls and streams we supplement this by projecting the depth buffer from the perspective of the camera onto the surface of a mesh, into the texture space of its shallow water simulation. This allows e.g. a character intersecting with a waterfall to occlude the falling stream of water, and a character standing in a running stream to have foam interacting with their feet where they intersect with the water surface.

2 CLOUDSCAPE RENDERING

The art direction in our game called for volumetric cloudscapes, storms and skull-shaped clouds that would appear to be part of the game world. Because of the performance requirements of the system and the high degree of artistic control desired, we chose not to pursue a ray-marched approach as favoured by modern game titles aiming at photo-realism. Instead, we developed a system that renders opaque geometry with simple per-vertex illumination that approximates sub-surface scattering.

In a technique similar to and partially inspired by that detailed in [Bahnassi and Bahnassi 2007], we render the cloud geometry into a separate off-screen buffer. The image is then downsampled to a quarter resolution, where the RGB channels undergo a Gaussian blur. In addition, we save out the depth information to the alpha channel and apply a box blur. We then project a quad in front of the player's camera which reads this texture and composites the computed image back into the primary scene colour buffer.

The compositing shader uses the depth information to calculate an approximate world-space position for every pixel, which allows us to apply additional fogging, translucency blending and other effects onto the image when it is being rendered. We sample a distortion map texture to give the clouds a fluffy appearance and we blend between low-frequency and high-frequency noise to further give the impression of depth, based on the distance of the pixel from the camera. For very distant clouds we apply an alpha threshold to sharpen their edges to give them a more distinct and cartoon-like appearance compared to the softer clouds directly overhead.

The cloud meshes are pre-distributed across a square several miles across. During runtime, we use the wind direction to offset all of the cloud meshes and wrap them around the player's position to give the impression of a continuous, moving cloudscape. We synchronise this offset vector across all clients, allowing every player to see the same clouds in the same relative locations to them.

We are then able to specify radial areas of high and low pressure that can modify the distribution of clouds. This allows level artists vary the amount of cloud cover throughout the world, using high pressure zones to create gaps in the cloud cover that fade out and push the cloud meshes out from inside them, and low pressure zones that do the opposite.

3 VERTEX ANIMATION

3.1 Ropes

The rigging on our ships demanded many ropes being rendered at the same time. We required high fidelity animated ropes that would respond to the movement of the ship, to the players individually controlling the sails and hold up when seen up close in first-person view. We did not have the computational resources to physically simulate so many rope segments, so we attempted to find an analytical solution.

Our system allows artists to define a rope system with a start point, an end point and any number of attachment points with pulleys in-between. An artist can set up tolerance for how much slack these ropes should be allowed to have. For each rope segment we are then able to calculate the desired length of that rope. On the CPU we use this value to solve the catenary equation and find the parameters for the hyperbola describing that rope. These parameters are passed into the vertex shader, which deforms an arbitrary length of tube into the correct curve. Rope offsets are tracked through the system to create the impression of one long rope.

3.2 Simulation

To add fidelity to Kraken tentacle animations wrapping around a ship, we use Houdini to simulate additional deformation and bake it down to vertex animation textures. We authored the Kraken tentacle wraps through our key-framed animation pipeline, then allowed those animations to drive FEM simulations. This extra deformation is written out to a texture that is used for vertex position, normal and tangent look-up in the final game. We blend several states and loops for a given type of tentacle attack into a single texture and by having a number of matching frames, we're able to implement simple state machines for the tentacle, transitioning to and from different attacks entirely within the GPU shader. This allowed us to trade CPU cost for memory usage, as well as reducing the animators' work as they could reuse general animation curves without having to animate precisely down to the geometry of the ships.

Houdini also found use in generating 3D shapes for forked lightning in storms. We start with a randomised L-System, then pick out the longest path. We then bake data into each vertex of the structure, encoding information about whether that vertex is part of the main branch or not, and how far it has traveled from the origin. For the final asset, we rotate and scale the mesh so that the end point is exactly one unit up above the start point. This representation allows a vertex shader to animate the the lightning strike, rendering the complex, realistic dynamics you might see in a slow-motion video of a lightning bolt to add an additional, dramatic effect to the event while keeping it believable.

REFERENCES

Homam Bahnassi and Wessam Bahnassi. 2007. Volumetric Clouds and Mega-Particles. In *ShaderX^5 Advanced Rendering Techniques*, Wolfgang Engel (Ed.). ShaderX Series, Vol. 5. Charles River Media, Boston, Chapter 5.3, 275–302.

Brian Karis and Epic Games. 2013. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice* (2013).

Xing Mei, Philippe Decaudin, and Bao-Gang Hu. 2007. Fast hydraulic erosion simulation and visualization on GPU. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. IEEE, 47–56.

Jerry Tessendorf. 2001. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH* 1, 2 (2001), 5.