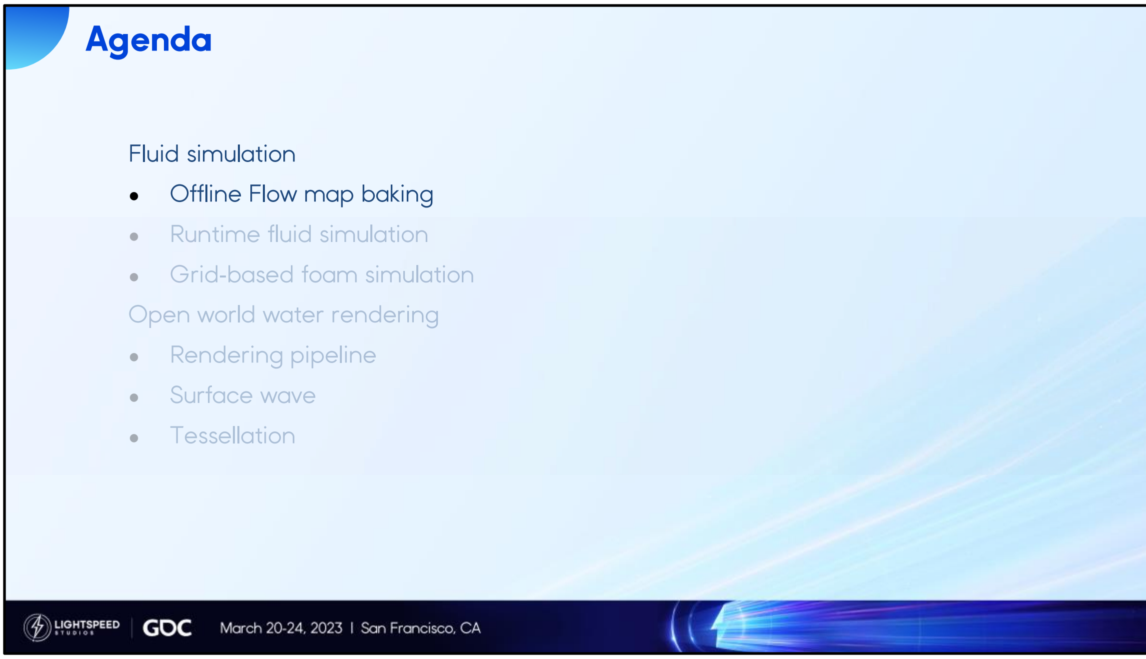

LIGHTSPEED STUDIOS
GDC
MEET
LIGHTSPEED STUDIOS
AT GDC2023
March 20-24, 2023 | San Francisco, CA

Hello everyone, this is Kui Wu. Today, Zhenyu and I are going to present our inhouse water system for unreal engine, Photon Water System, for open world water rendering and real-time simulation. I will first present the simulation part and then Zhenyu will introduce the components on rendering side.

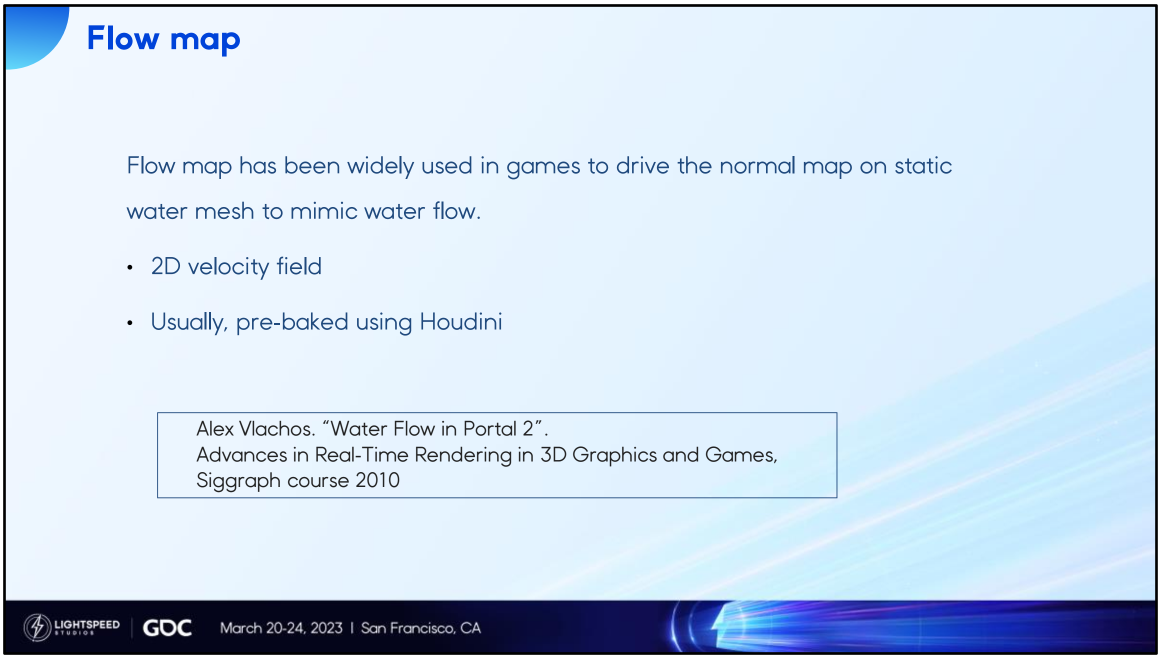
Before getting into the details, let me briefly introduce myself. My name is Kui. I got my phd in compute graphics from the University of Utah in 2019. before joining lightspeed studios as a senior researcher, I spent 1 year at MIT as a posdoc.

Water is a critical feature in video games, especially open-world games. Traditional game solutions mostly use pre-generated mesh and flow maps. Runtime fluid simulation solutions are limited to a small simulation domain and require a high-end GPU. However, we need a solution to support various projects and has a capability of scale from mobile platform to next gen console. Based on this request, we are inspired by several existing tools and plugins, such as Unreal Engine water system and water rendering frameworks in other games. We develop an inhouse water system in Unreal engine to support several games under development in our studios.

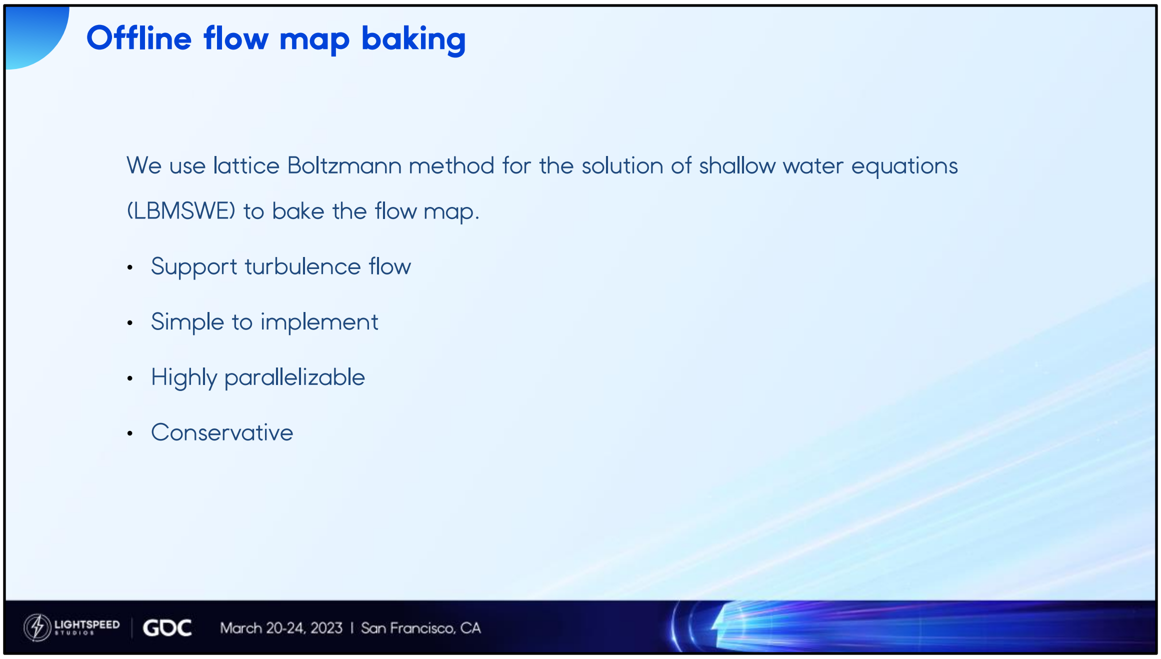
We call it as photon Water System internally, which is an Unreal Engine plugin for a unified open world water solution, including a full tool set from content creation for river, lake and ocean. As shown several screenshot below about some of our features, our system also supports offline Flowmap baking, runtime fluid simulation. buoyancy and boat physics , underwater volumetric lighting, adaptive water mesh tessellation..



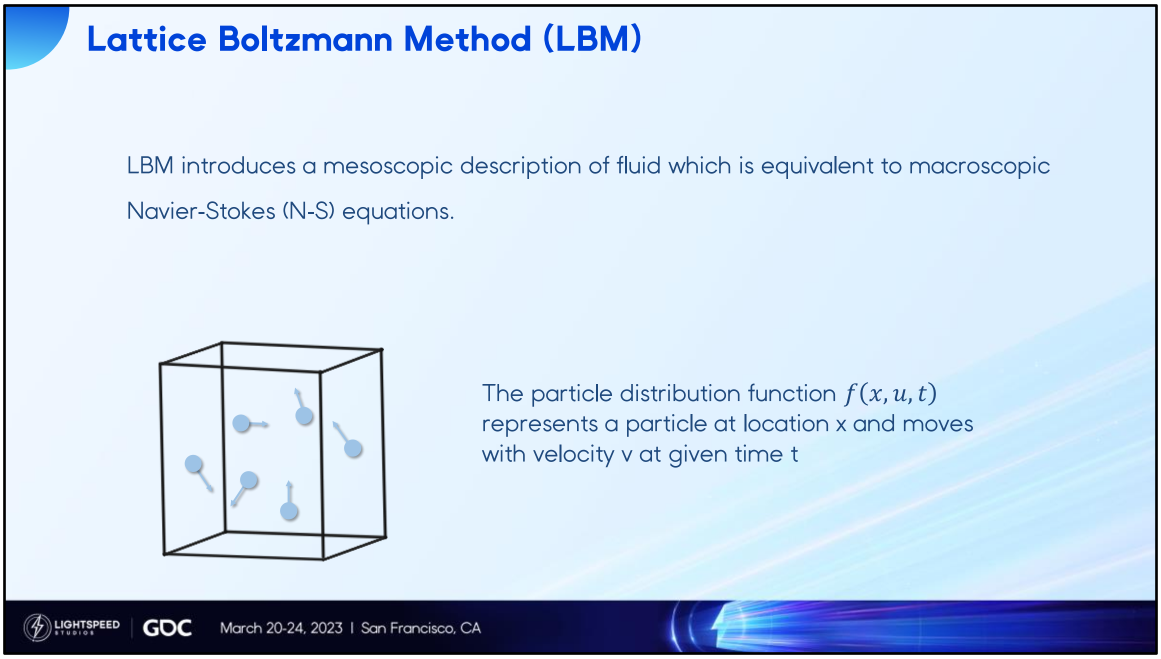
Here is the agenda today. I will first introduce the components for fluid simulation



Flow map has been widely used in games to drive the normal map on static water mesh to mimic water flow. We refer to siggraph course in 2010 for more details about how to use the flow map. Basically, the flowmap is pre-computed using physical-based simulation tool in existing third party software, such as Houdini. And the velocity field is stored as a 2d vector map and fetch at runtime. However, the existing tool is still inconvenience for artists to use and cannot support turbulent flow.



Therefore, we implement a lattice Boltzmann method for the solution of shallow water equations (LBMSWE) in the unreal engine to bake a physics driven flow map. We choose lbm because of its several advantages, such as Support turbulence flow, Simple to implement, Highly parallelizable, Conservative



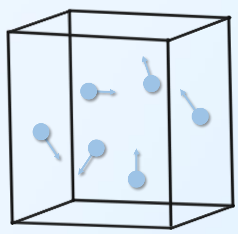
LBM introduces a mesoscopic description of fluid which is equivalent to Macroscopic Navier-Stokes (N-S) equations. It considers a collection of particles as a unit, and accurately model average behavior of these particles as macro-scale behavior. $f(x, v, t)$ is the particle distribution function that there is a particle at location x and moves with velocity v at given time t.

Lattice Boltzmann Method (LBM)

LBM introduces a mesoscopic description of fluid which is equivalent to macroscopic Navier-Stokes (N-S) equations.

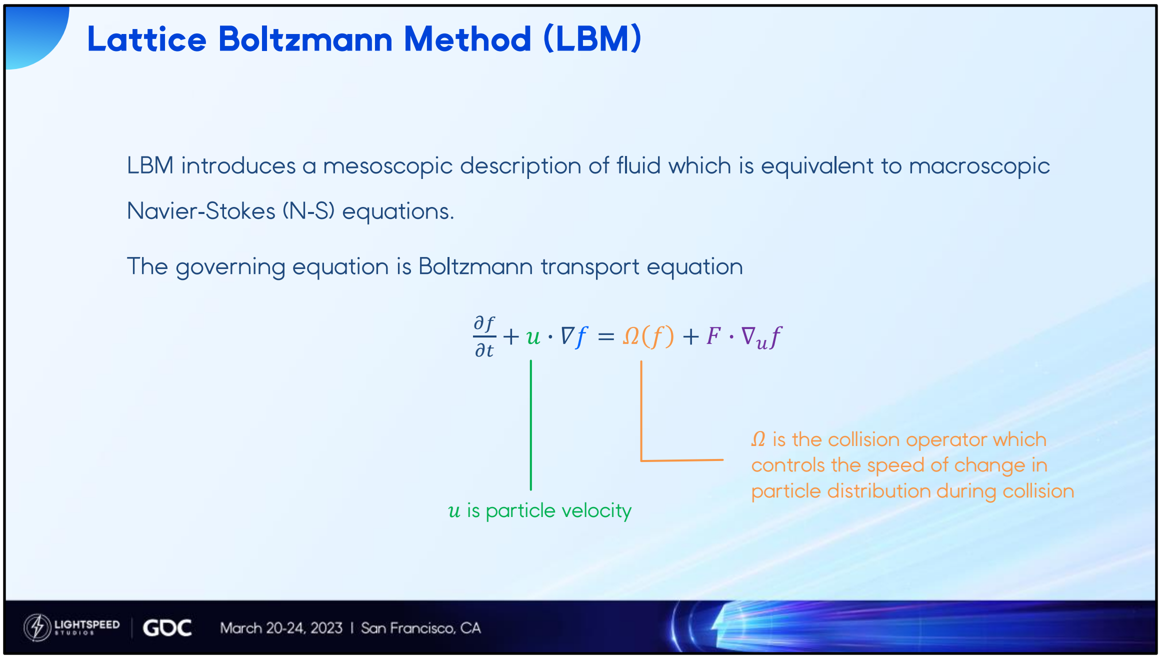
The evolution of function f is defined by the Boltzmann transport equation

$$\frac{\partial f}{\partial t} + u \cdot \nabla f = \Omega(f) + F \cdot \nabla_u f$$



LIGHTSPEED STUDIOS | GDC March 20-24, 2023 I San Francisco, CA

The evolution of function f is defined by the Boltzmann transport equation,



u is particle velocity, Ω is the collision operator which controls the speed of change in particle distribution during collision.

Lattice Boltzmann Method (LBM)

LBM introduces a mesoscopic description of fluid which is equivalent to macroscopic Navier-Stokes (N-S) equations.

The governing equation is Boltzmann transport equation

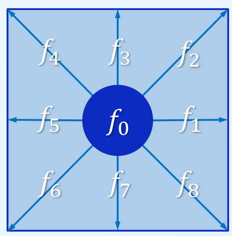
$$\frac{\partial f}{\partial t} + u \cdot \nabla f = \Omega(f) + F \cdot \nabla_u f$$

external force term

Note that the purple term can incorporate with other force such as the force from shallow water assumption.

Lattice Boltzmann Method (LBM)

LBM uses 9-speed square lattice pattern that each cell contains 9 distribution functions to indicate the particle motions.



Direction

```
const float2 e[9] =
{
    float2( 0.0f,  0.0f),
    float2( 1.0f,  0.0f),
    float2( 1.0f,  1.0f),
    float2( 0.0f,  1.0f),
    float2(-1.0f,  1.0f),
    float2(-1.0f,  0.0f),
    float2(-1.0f, -1.0f),
    float2( 0.0f, -1.0f),
    float2( 1.0f, -1.0f),
};
```

Weight

```
const float w[9] =
{
    0.0f,
    1.0f,
    0.25f,
    1.0f,
    0.25f,
    1.0f,
    0.25f,
    1.0f,
    0.25f
};
```


LBM uses 9-speed square lattice pattern that each cell contains 9 distribution functions to indicate the particle motions. **There are also two arrays to store the direction and weight for each** distribution function.

Lattice Boltzmann Method (LBM)

Discretization and time update of distributions:

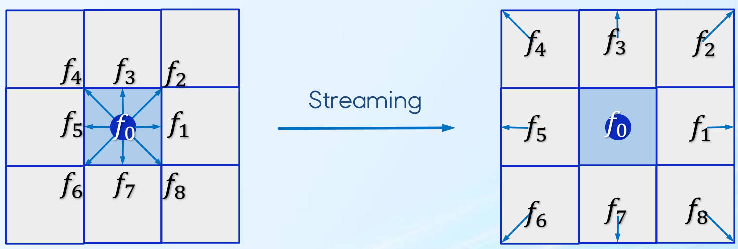
$$f_\alpha^{x+1,t+1} - f_\alpha^t = \underbrace{-\frac{1}{\tau}\left(f_\alpha^t - f_\alpha^{eq}\right)}_{\text{BGK Collision Operator}} + F \cdot \nabla_u f$$

For Discretization and time update of distributions, we can rewrite the update function as, with BGK collision operator.

Lattice Boltzmann Method (LBM)

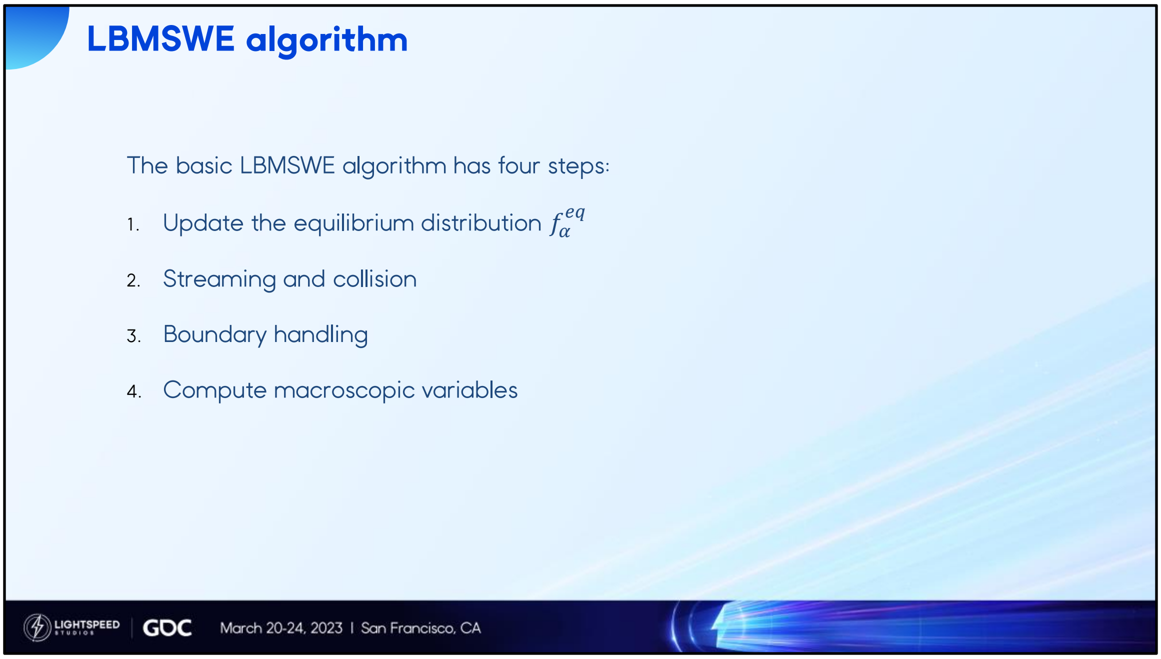
Discretization and time update of distributions:

$$f_\alpha^{x+1,t+1} - f_\alpha^{t} = \underbrace{-\frac{1}{\tau}\left(f_\alpha^{t} - f_\alpha^{eq}\right)}_{\text{BGK Collision Operator}} + F \cdot \nabla_u f$$



LIGHTSPEED STUDIOS | GDC March 20-24, 2023 | San Francisco, CA

In particular, the lbm update scheme is based on the streaming, that distribution function will stream to the neighboring cell as shown here.



To solve this equation, the basic LBMSWE algorithm only has four steps:

1. Update equilibrium distribution
2. Streaming and collision
3. Boundary handling
4. Compute macroscopic variables

Step 1: update f_α^{eq}

We first update the equilibrium distribution f_α^{eq} based on current h, u , v

$$
f_\alpha^{eq} = \begin{cases} h - \dfrac{5gh^2}{6e^2} - \dfrac{2h}{3e^2} u_i u_i & \alpha = 0, \\ \dfrac{gh^2}{6e^2} + \dfrac{h}{3e^2} e_{\alpha i} u_i + \dfrac{h}{2e^4} e_{\alpha i} e_{\alpha j} u_i u_j - \dfrac{h}{6e^2} u_i u_i & \alpha = 1,3,5,7, \\ \dfrac{gh^2}{24e^2} + \dfrac{3h}{12e^2} e_{\alpha i} u_i + \dfrac{h}{8e^4} e_{\alpha i} e_{\alpha j} u_i u_j - \dfrac{h}{24e^2} u_i u_i & \alpha = 2,4,6,8. \end{cases}
$$

```
void updateFeq(float u, float v, float h)
{
    float uu = u * u + v * v;

    f_eq[uint3(xy, 0)] = h * (1.0f - 5.0f/6.0f*g*h*InvE2 - 2.0f/3.0f*InvE2*uu);

    [unroll]
    for (uint a = 1; a < 9; a++)
    {
        float eu = (e[a].x * u + e[a].y * v) * E;
        f_eq[uint3(xy, a)] = w[a] * (g*h/6.0f + eu/3.0f + InvE2*eu*eu/2.0f – uu/6.0f)*h*InvE2;
    }
}
```


local equilibrium function plays an essential role in the lattice Boltzmann method. We first update current targeted equilibrium distribution f_α^{eq} based on current h, u , v, where f_α^{eq} is a model specified function. This is the definition for lbmswe. Alpha indicates different possibility along different directions.

Step 2: streaming and collision

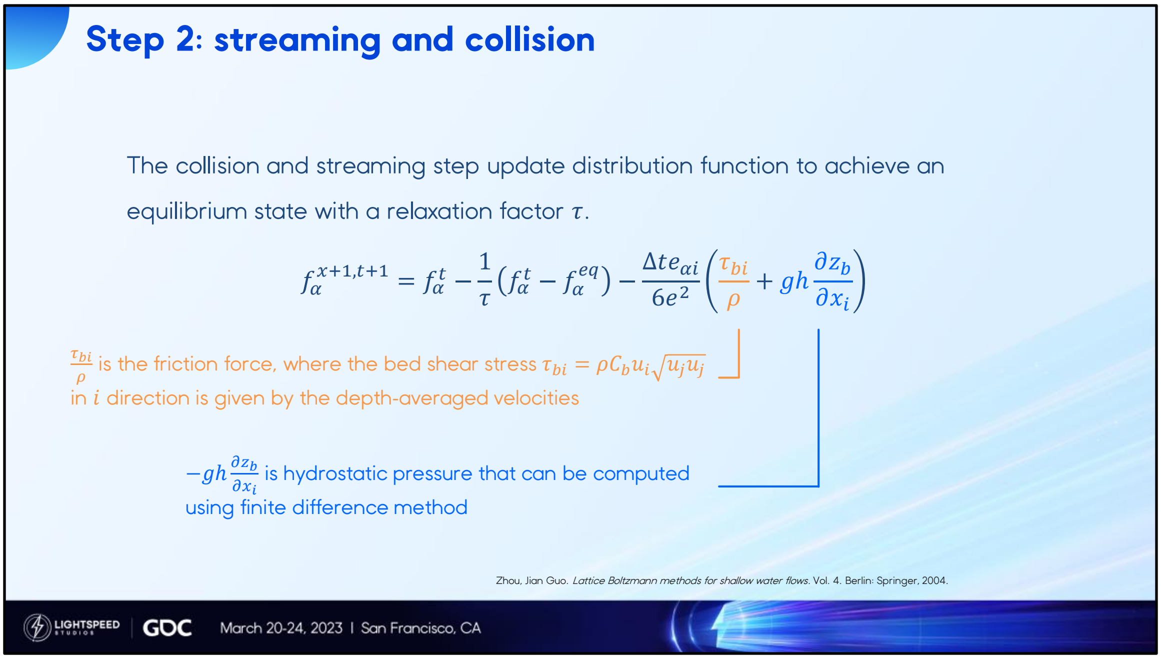
The collision and streaming step update distribution function to achieve an equilibrium state with a relaxation factor τ.

$$\boxed{f_\alpha^{x+1,t+1} = f_\alpha^t - \frac{1}{\tau}\left(f_\alpha^t - f_\alpha^{eq}\right)} - \frac{\Delta t e_{\alpha i}}{6e^2}\left(\frac{\tau_{bi}}{\rho} + gh\frac{\partial z_b}{\partial x_i}\right)$$

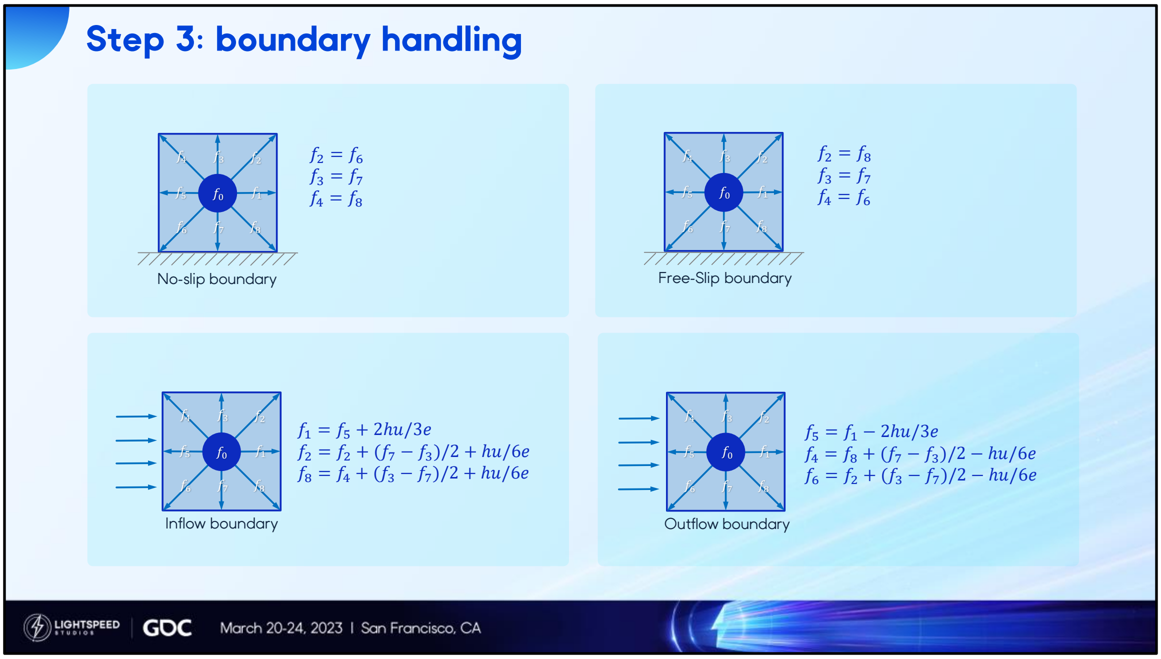
```
int x  = xy.x;
int y  = xy.y;
int xf = xy.x + 1;
int xb = xy.x - 1;
int yf = xy.y + 1;
int yb = xy.y - 1;

f_new[uint3(x,  y,  0)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 0)] + inv_Tau * f_eq[uint3(x, y, 0)];
f_new[uint3(xf, y,  1)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 1)] + inv_Tau * f_eq[uint3(x, y, 1)] + (e[1].x * ext[ 2] + e[1].y * ext[ 3]) * s;
f_new[uint3(xf, yf, 2)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 2)] + inv_Tau * f_eq[uint3(x, y, 2)] + (e[2].x * ext[ 4] + e[2].y * ext[ 5]) * s;
f_new[uint3(x,  yf, 3)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 3)] + inv_Tau * f_eq[uint3(x, y, 3)] + (e[3].x * ext[ 6] + e[3].y * ext[ 7]) * s;
f_new[uint3(xb, yf, 4)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 4)] + inv_Tau * f_eq[uint3(x, y, 4)] + (e[4].x * ext[ 8] + e[4].y * ext[ 9]) * s;
f_new[uint3(xb, y,  5)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 5)] + inv_Tau * f_eq[uint3(x, y, 5)] + (e[5].x * ext[10] + e[5].y * ext[11]) * s;
f_new[uint3(xb, yb, 6)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 6)] + inv_Tau * f_eq[uint3(x, y, 6)] + (e[6].x * ext[12] + e[6].y * ext[13]) * s;
f_new[uint3(x,  yb, 7)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 7)] + inv_Tau * f_eq[uint3(x, y, 7)] + (e[7].x * ext[14] + e[7].y * ext[15]) * s;
f_new[uint3(xf, yb, 8)] = (1.0 - inv_Tau) * f_old[uint3(x, y, 8)] + inv_Tau * f_eq[uint3(x, y, 8)] + (e[8].x * ext[16] + e[8].y * ext[17]) * s;
```

LIGHTSPEED STUDIOS | GDC March 20-24, 2023 | San Francisco, CA

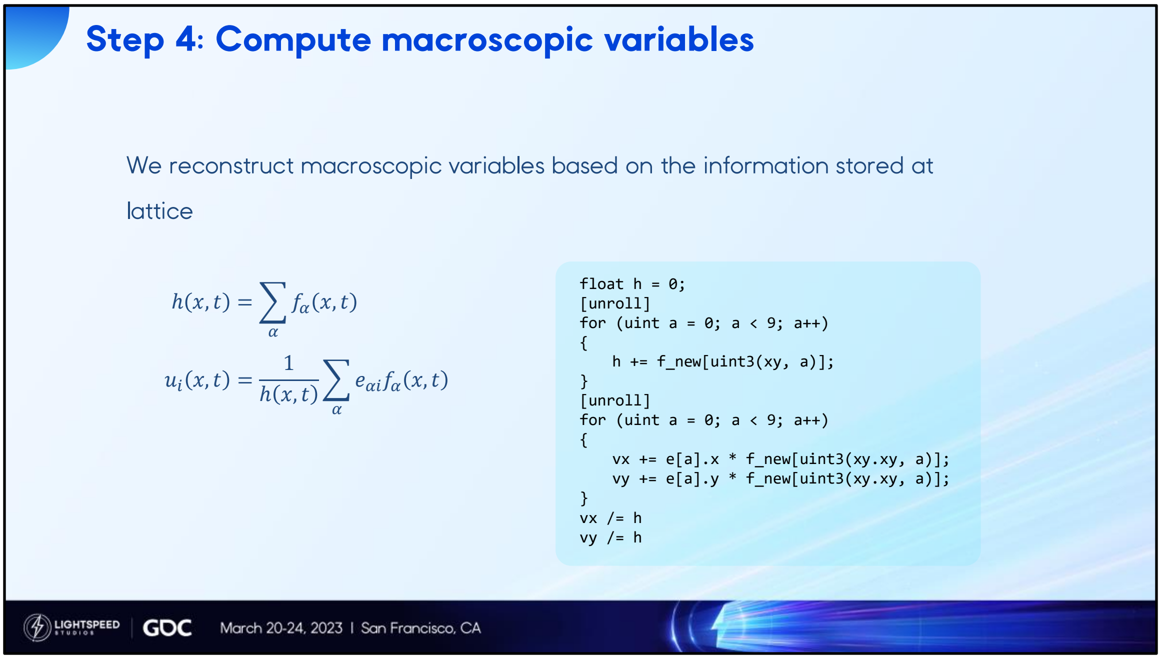
The highlight part is trivial to compute as shown in the pseudocode below, and same as the traditional LBM. We can use the current distribution function f, equilibrium distribution function feq and a relaxation factor τ to update the new distribution function



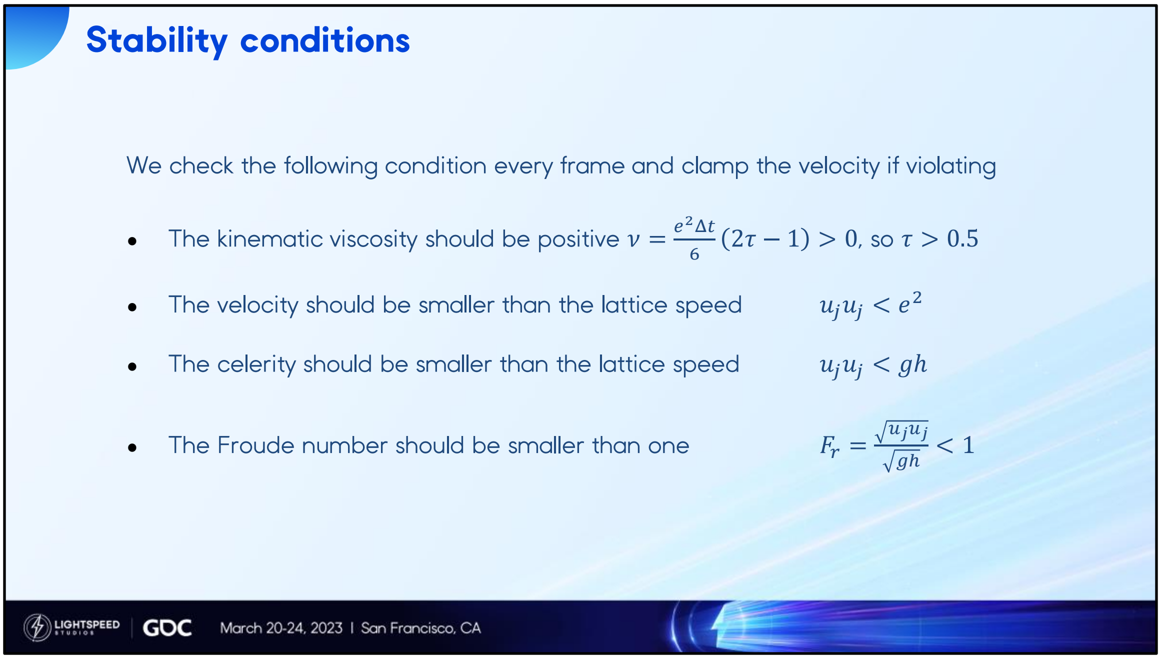
The last term is for the shallow water equation, including the friction force, where the bed shear stress in i direction is given by the depth-averaged velocities and is hydrostatic pressure. Due to limited time, please refer the book for more detailed definitions.



Then, we can handle the boundary by simply flipping the distribution function at certain direction, we can handle no-slip boundary, free-slip boundary, inflow boundary and outflow boundary.

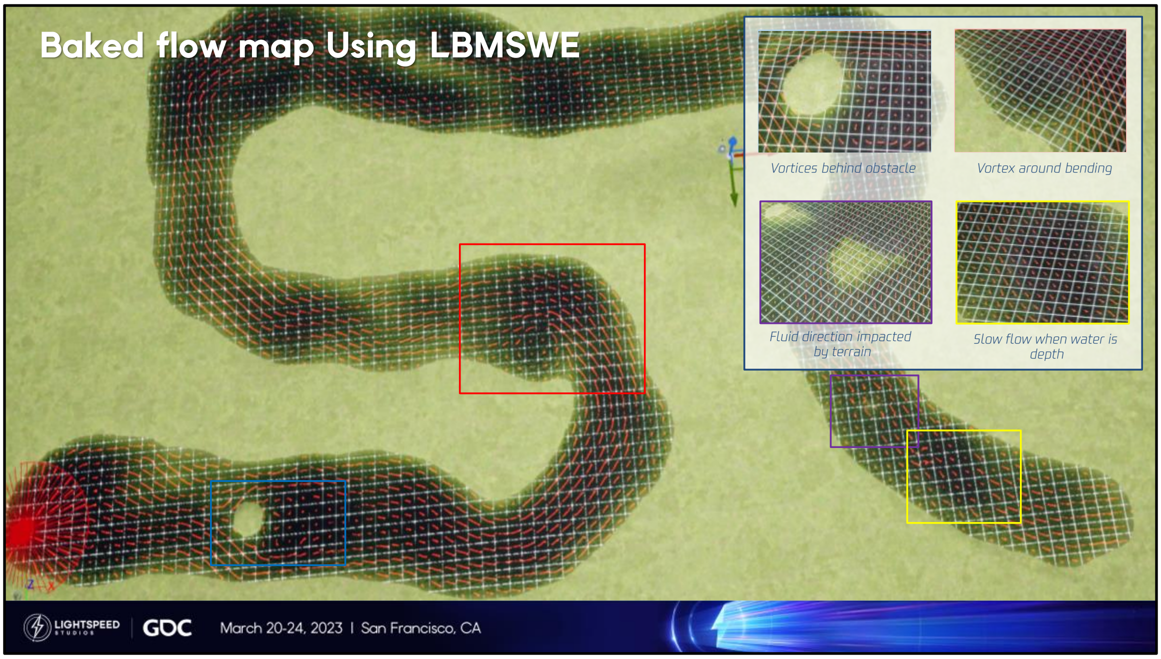


finally, we can compute Macroscopic quantities , such as the velocity and depth by accumulating the local values from 9 directions.

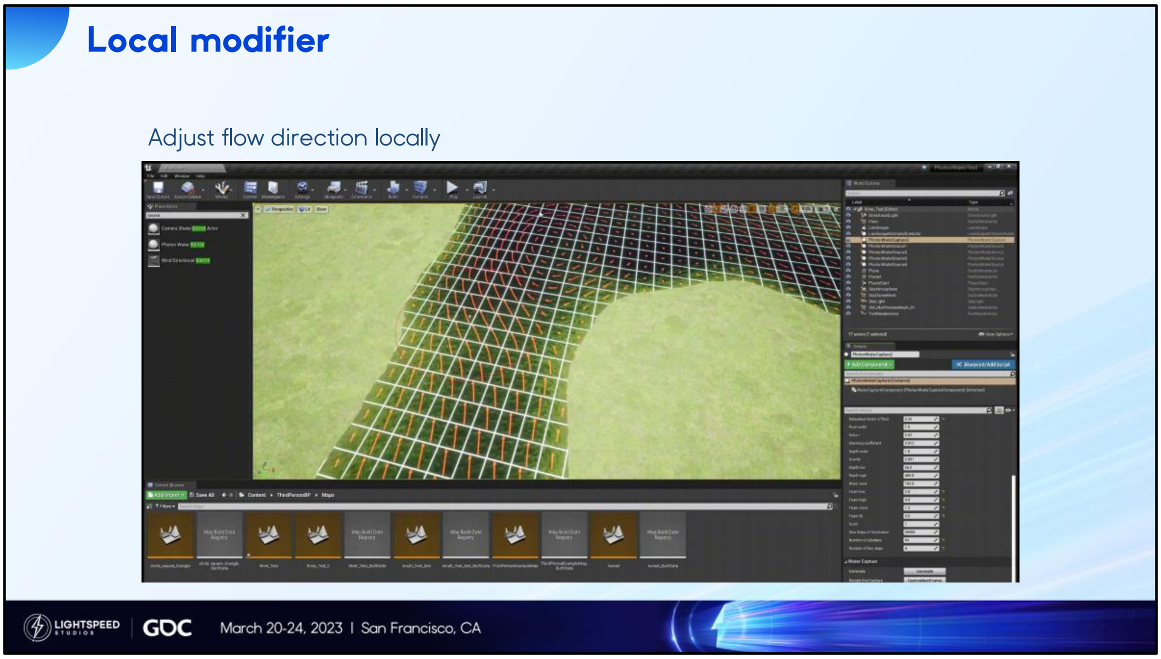


Since the lattice Boltzmann equation is a discrete form of a numerical method. It may suffer from a numerical instability like any other numerical methods. We check four the stability conditions as described below.

The kinematic viscosity should be positive, The velocity should be smaller than the lattice speed, The celerity should be smaller than the lattice speed, The Froude number should be smaller than one. We enforce those condition at the end of each step and clamp the velocity when it violates stability conditions.

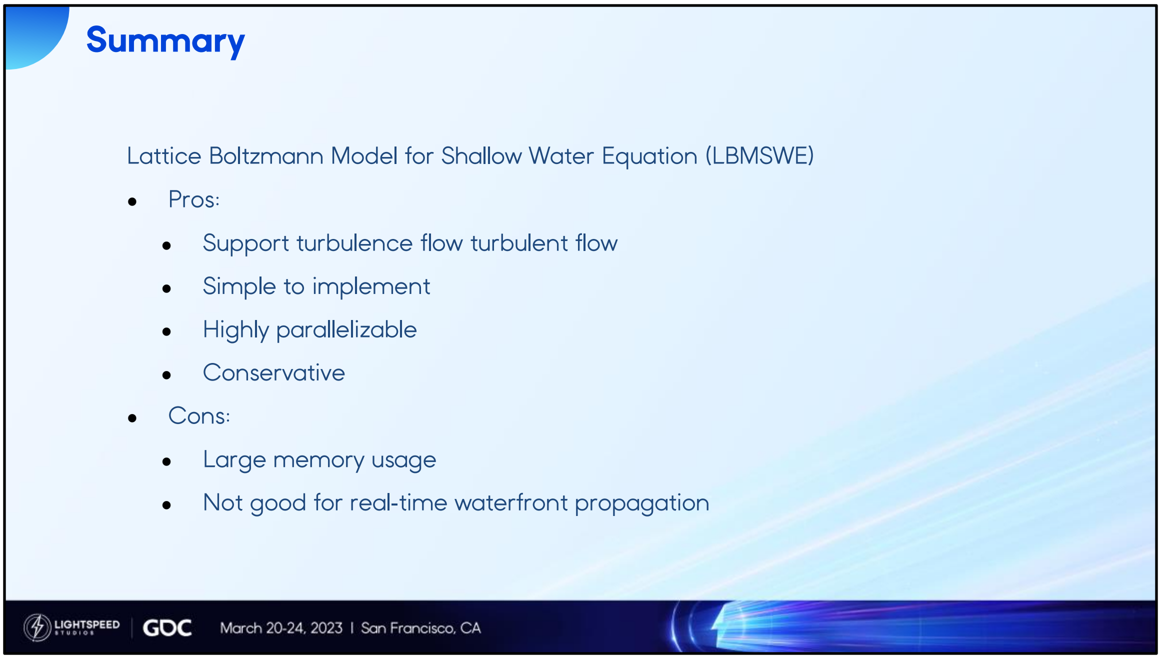


The LBMSWE is very easy to implemented in UE using compute shader. Here is a top-down view of a long river. As we can see, LBMSWE can create Vortex around bending and behind obstacle. Also, the Fluid direction impacted by underneath terrain as well as the water depth.



We also provide artist a set of tools to modify the result. Such as the example shown, artist can adjust the flow direction locally until they are happy with

Here is a close view. The red arrows show the flow map computed using LBMSWE.



To summarize, Lattice Boltzmann Model for Shallow Water Equation (LBMSWE) is good to generate flow map offline with several advantages. However, since each cell needs to store 3 distribution values for nine directions, which has a large memory cost. Also, since lbmswe is still a single phase method, it is not good for real-time waterfront propagation

Agenda

Fluid simulation

- Offline Flow map baking
- Runtime fluid simulation
- Grid-based foam simulation

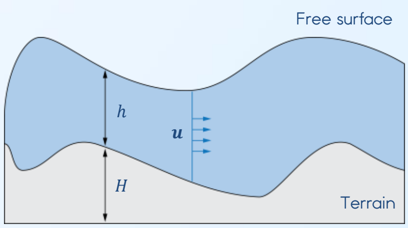
Open world water rendering

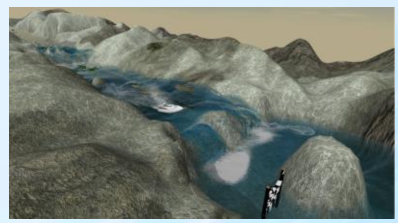
- Rendering pipeline
- Surface wave
- Tessellation

Runtime fluid simulation

We use Shallow Water Equation (SWE), a height-field based 2.5D fluid simulation

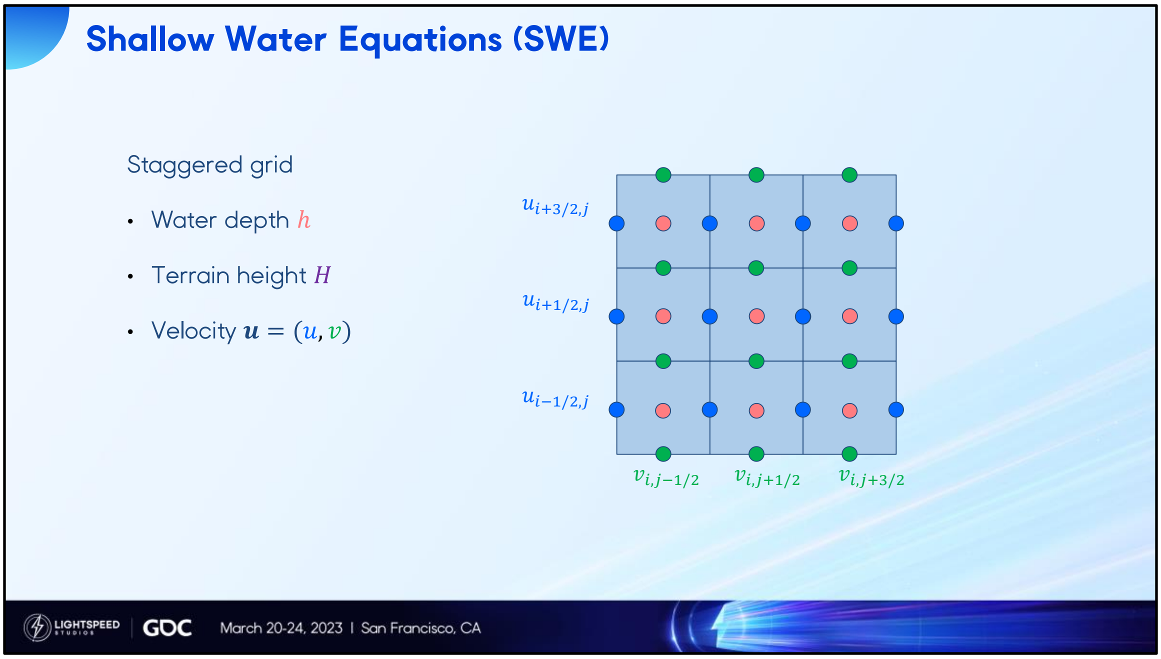
- $\frac{Dh}{Dt} + h\nabla \cdot \boldsymbol{u} = 0$ Conservation of mass
- $\frac{D\boldsymbol{u}}{Dt} = -g\nabla(h + H)$ Conservation of momentum





Chentanez, Nuttapong, and Matthias Müller.
Real-time Simulation of Large Bodies of Water with Small Scale Details.
Symposium on Computer Animation. 2010.

For runtime, we do a height field fluid simulation based on the shallow water assumption. The conservation of mass and momentum leads to the following well-known SWE governing equation, where h is the water depth, H is the terrain height. The horizontal velocity u contains the velocity components along the x and y directions. We borrow the idea of the previous academic paper, cited here.



We use a staggered grid to store the data. The height are stored at cell centers, and velocities are stored at edge centers.

Shallow Water Equations (SWE)

Splitting into 3 steps:

- Step 1: $\frac{\partial h}{\partial t} = -h\nabla \cdot \boldsymbol{u}$ Height integration
- Step 2: $\frac{\partial \boldsymbol{u}}{\partial t} = -\boldsymbol{u} \cdot \nabla \boldsymbol{u}$ Velocity advection
- Step 3: $\frac{\partial \boldsymbol{u}}{\partial t} = -g\nabla(h + H)$ Apply water pressure

Our fluid time integration algorithm is based on the splitting scheme consisting of three main steps, height integration, velocity advection, and apply water pressure.

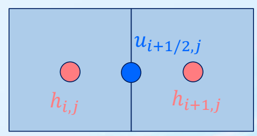
Step 1: Height integration

Discretization

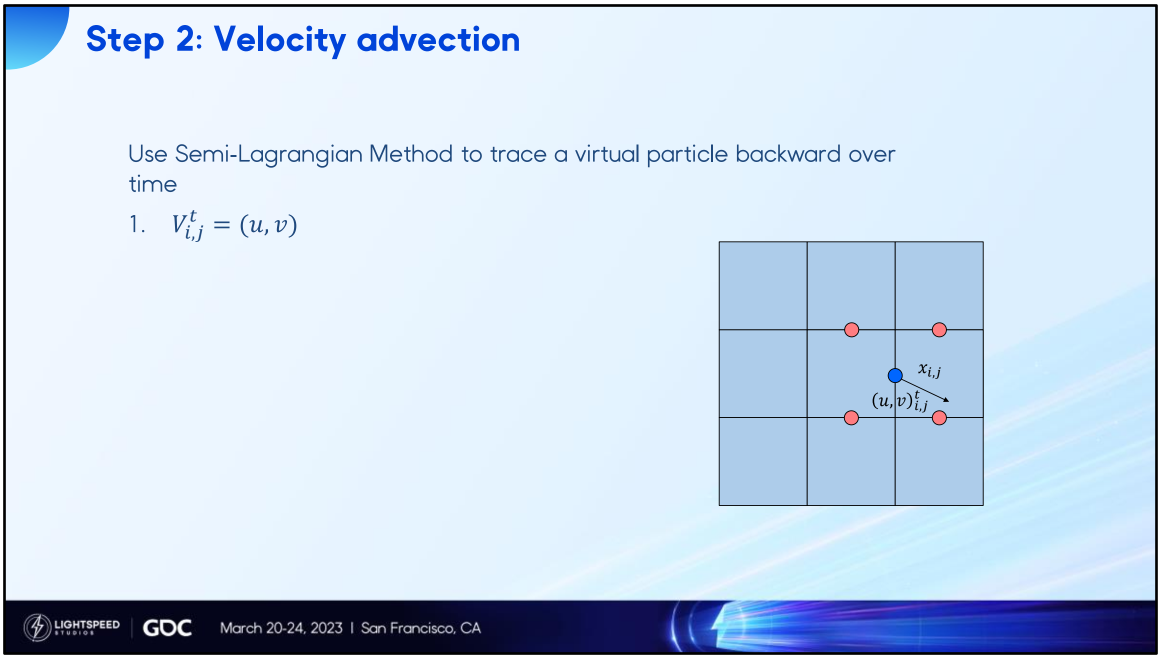
$$h_{i,j} \mathrel{+}= -\frac{\Delta t}{\Delta x}\left(\left(\bar{h}u\right)_{i+1/2,j} - \left(\bar{h}u\right)_{i-1/2,j} + \left(\bar{h}w\right)_{i,j+1/2} - \left(\bar{h}w\right)_{i,j-1/2}\right)$$

Up-winding advector

$$\left(\bar{h}u\right)_{i+1/2,j} = \begin{cases} h_{i+1,j}u_{i+1/2,j} & \text{if } u_{i+1/2,j} \le 0 \\ h_{i,j}u_{i+1/2,j} & \text{if } u_{i+1/2,j} > 0 \end{cases}$$



We update the height from velocity using the following mass-conserved equation. In particular, we use a conserved up-winding advection scheme to figure out which direction we should get the information based on the face velocity.

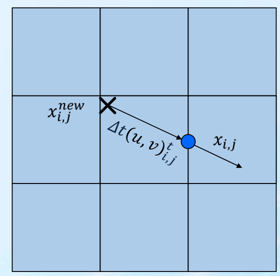


To discretize **velocity advection**, we first use an unconditionally stable Semi-Lagrangian advector to update the velocities. The key idea is very simple. First, we fetch the velocity at the current location

Step 2: Velocity advection

Use Semi-Lagrangian Method to trace a virtual particle backward over time

1. $V_{i,j}^{t} = (u, v)$
2. $x_{i,j}^{new} = x_{i,j} - \Delta t V_{i,j}^{t}$



LIGHTSPEED STUDIOS | GDC March 20-24, 2023 I San Francisco, CA

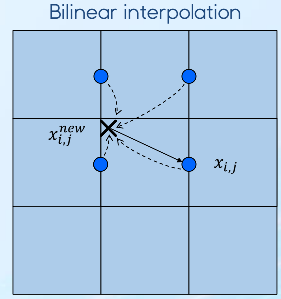
, and use a virtual particle back trace to a new location based on the current velocity,

Step 2: Velocity advection

Use Semi-Lagrangian Method to trace a virtual particle backward over time

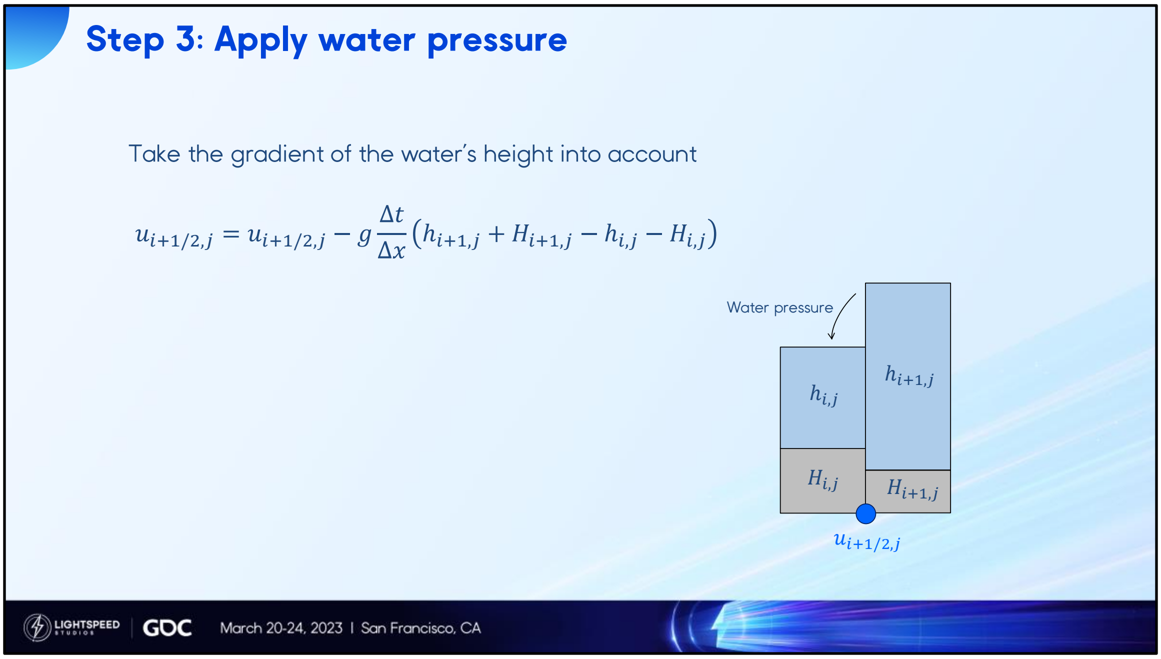
1. $V_{i,j}^{t} = (u, v)$
2. $x_{i,j}^{new} = x_{i,j} - \Delta t V_{i,j}^{t}$
3. $u_{i,j}^{t+1} = u^{t}\,(x_{i,j}^{new})$

Bilinear interpolation

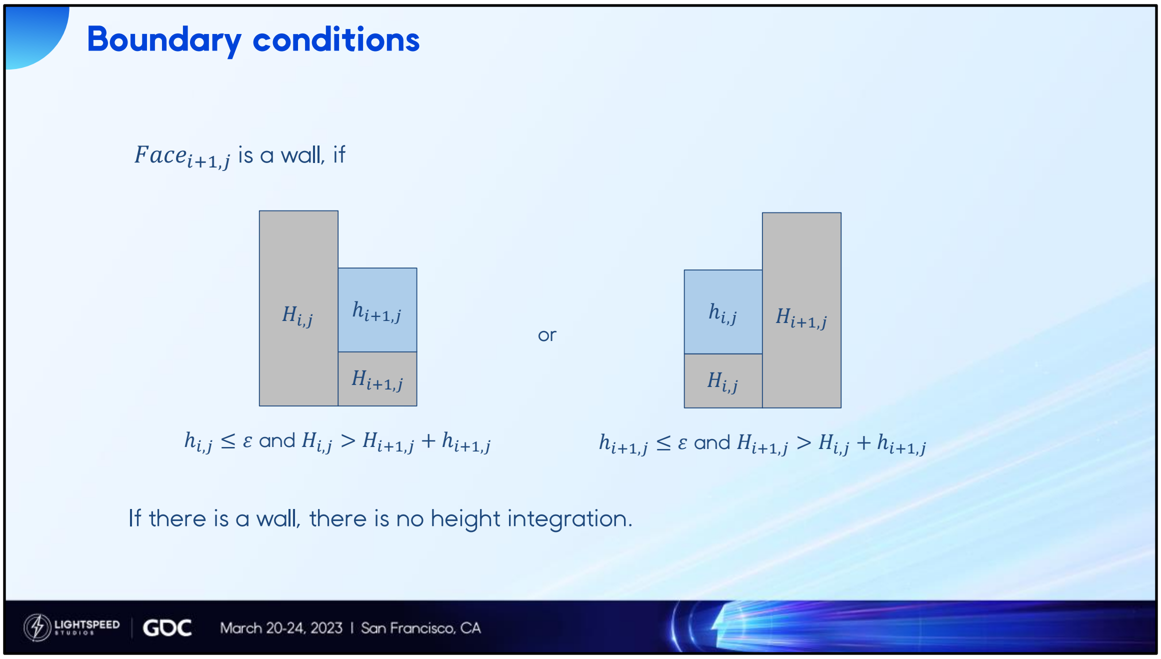



```
void Semi-LagrangianAdvection(float2 pos)
{
    float u = read_u(pos)
    float v = SRVInVelXY.SampleLevel(SRVSampler, pos, 0).x;
    float2 pos_new = pos + dt * float2(u, v);
    float u_new = SRVInVelXY.SampleLevel(SRVSampler, pos_new, 0).y;
}
```

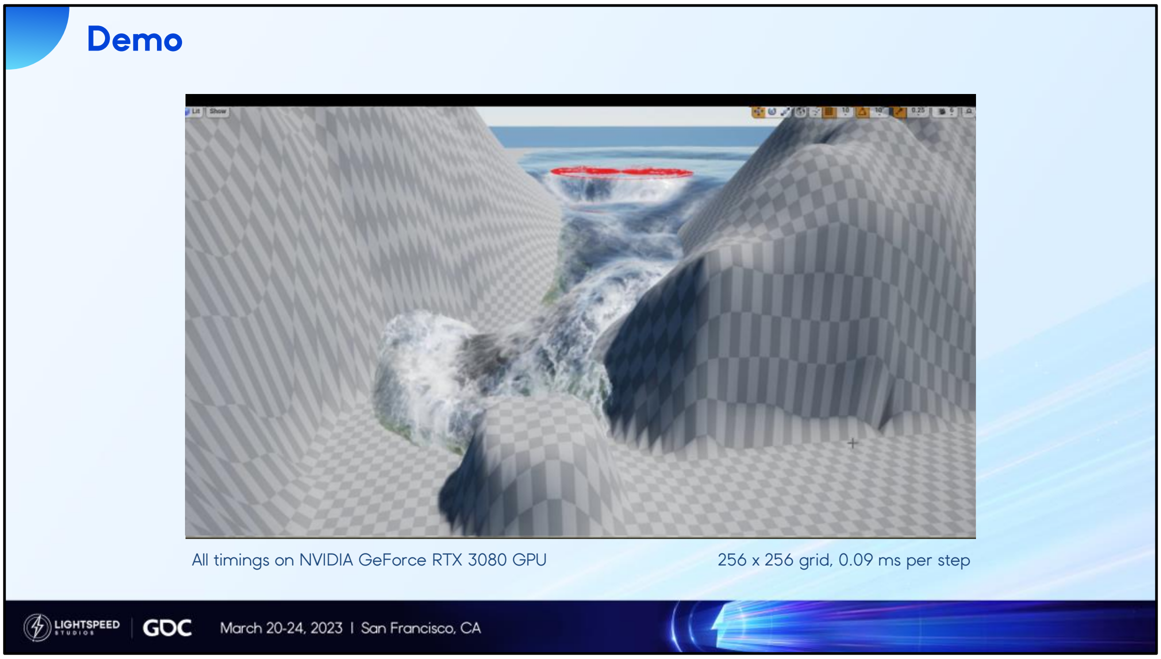

Last, we can fetch the velocity from the new location suing bilinear interpolation and write it to the original location.



Then, we explicitly update velocities by taking the gradient of the water height into account as follows. Note that the gradient computation should include the terrain height as well, since we are measuring the gap between neighboring's water levels.



We also need to check each face and see whether it is a wall based on the terrain height and water depth. In particular, if either of the following is true, we set the face i+1, j as the wall and set the velocity value as 0 at the end of every time step.

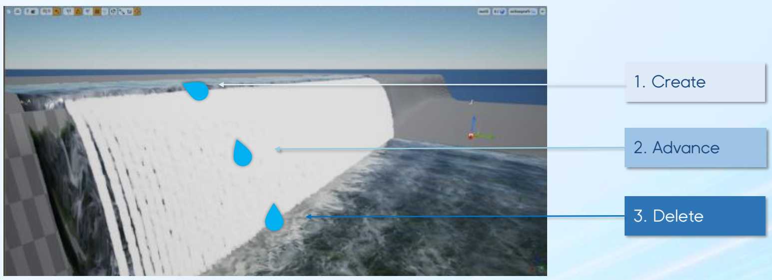


Here is an example of our shallow water simulation in unreal engine. The domain is 256x256 grid. In this demo, it only takes 0.09 ms per step. [13.5mins]

SWE + particles system

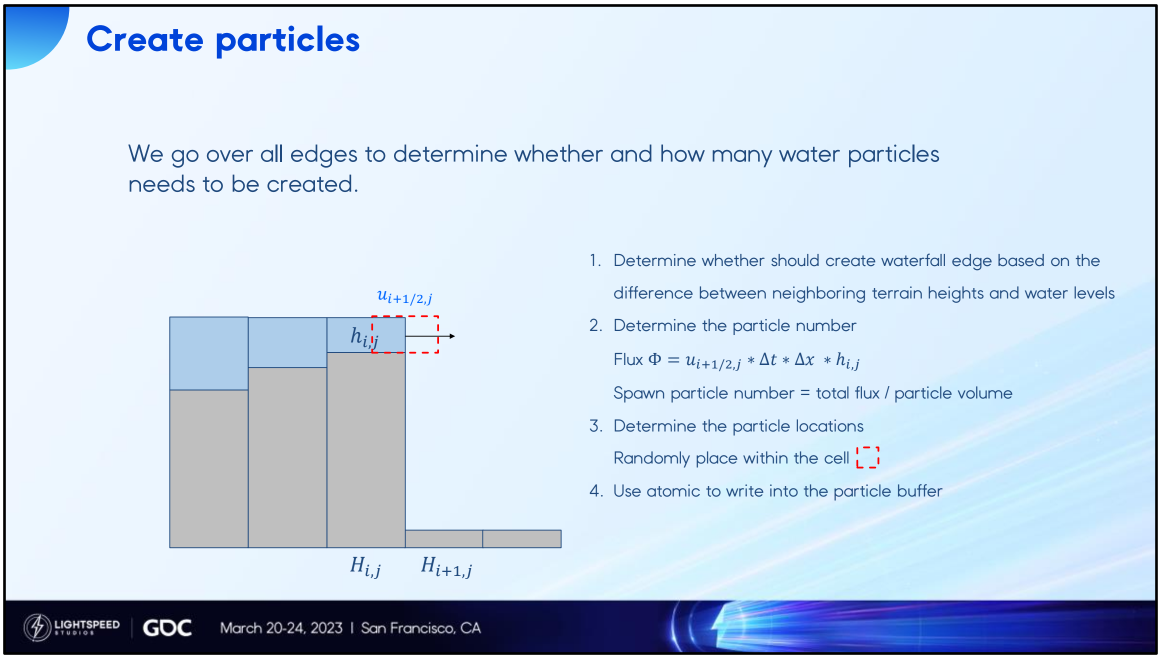
Problem: SWE does not support discontinuous terrain, such as waterfall

Solution: additional particle system, in which each particle carries water mass and velocity



LIGHTSPEED STUDIOS | GDC March 20-24, 2023 I San Francisco, CA

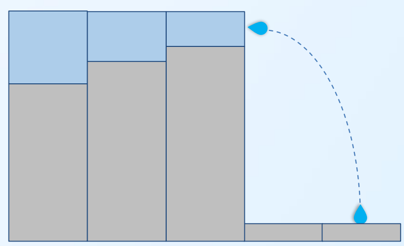
SWE is fast and can be parallelized easily, however, due to the shallow assumption and the limited representation of height field, it doesn't support terrain discontinuities, such as **waterfall** and breaking waves naturally. We follow the same idea and use the particle system. The surface discontinuities are detected automatically and the liquid volume in such locations are converted from SWE system into particles that carry mass and momentum of the height field across the discontinuity. Obviously, it contains three steps, create, advance, and delete



We first go over all edges to determine whether and how many water particles needs to be created. The spawn particles number can be computed based on the flux over the face. Then, we randomly shift the particle locations and use atomic to write into the particle buffer.

Advance particles

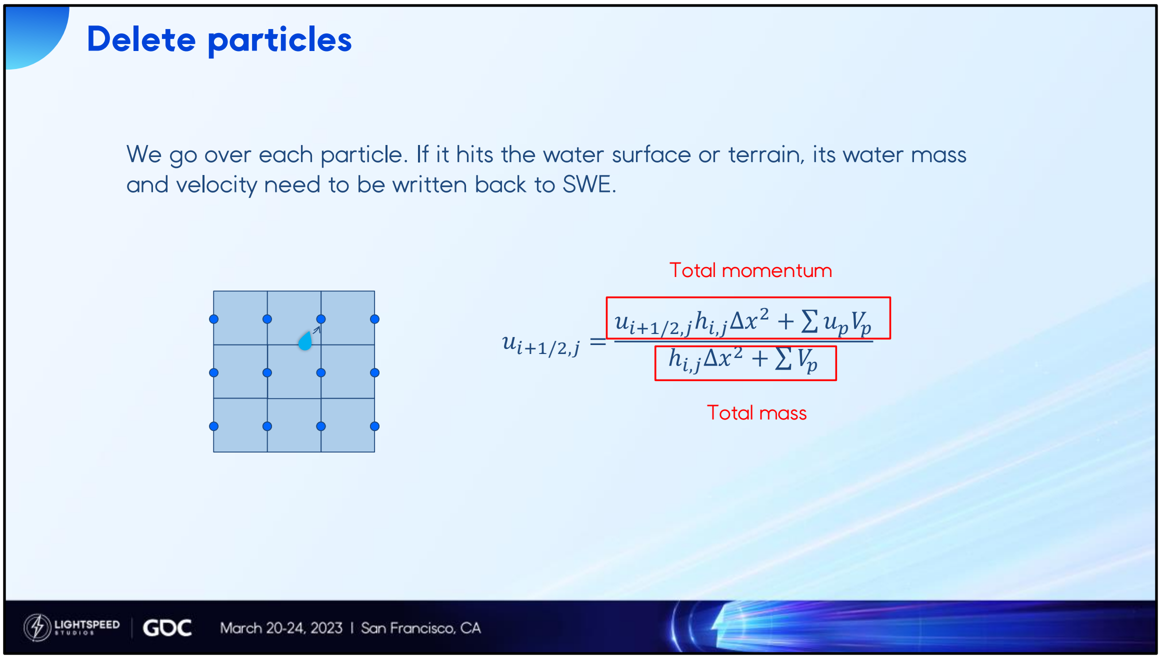
All water particles fall under the sole influence of gravity.



```
void advanceParticle(float3& vel, float3& pos)
{
    const float C = drag * length(vel);
    vel = (vel + float3(0, 0, -gravity * dt)) / (1 + C * dt);
    pos = pos + vel * dt;
}
```

LIGHTSPEED STUDIOS | GDC March 20-24, 2023 | San Francisco, CA

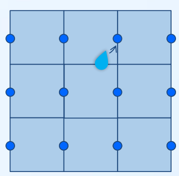
The advances is trivially moving all water particles under free fall by gravity.



when particle hits ground or water, **deleted particles** should contribute back to the SWE grid based. We first accumulate total momentum and velocity from particles to each cell. Then, the new velocity can be computed as the sum of momentum divided by total volume.

Delete particles

We go over each particle. If it hits the water surface or terrain, its water mass and velocity need to be written back to SWE.



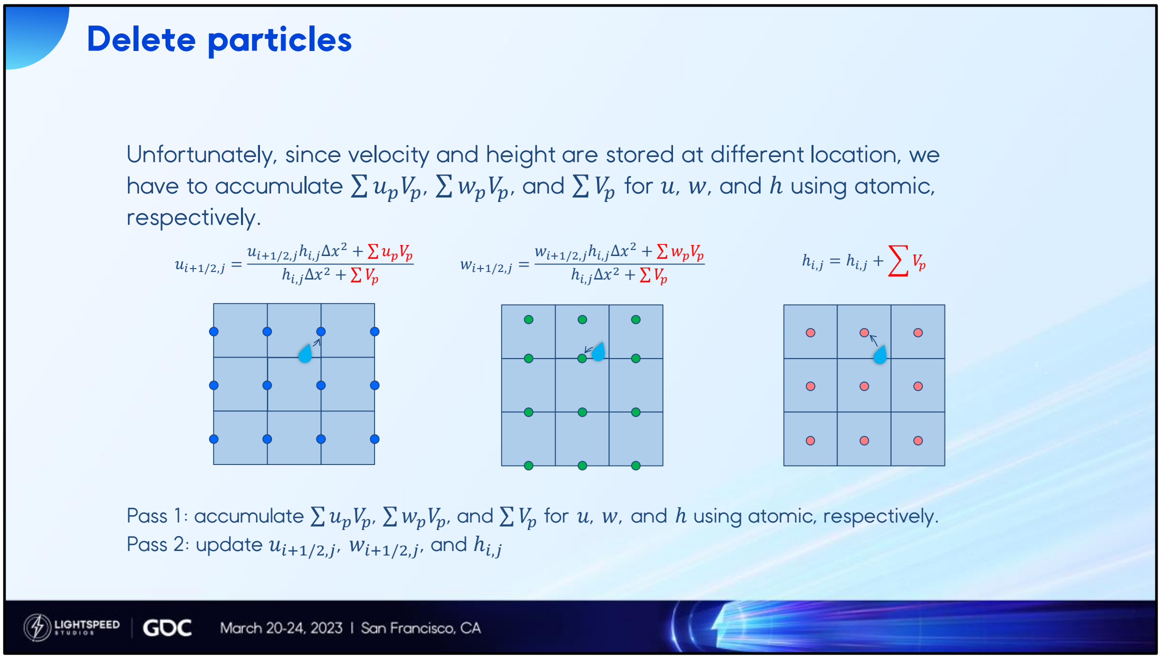
Total momentum

$$u_{i+1/2,j} = \frac{\boxed{u_{i+1/2,j} h_{i,j} \Delta x^2 + \sum u_p V_p}}{\boxed{h_{i,j} \Delta x^2 + \sum V_p}}$$

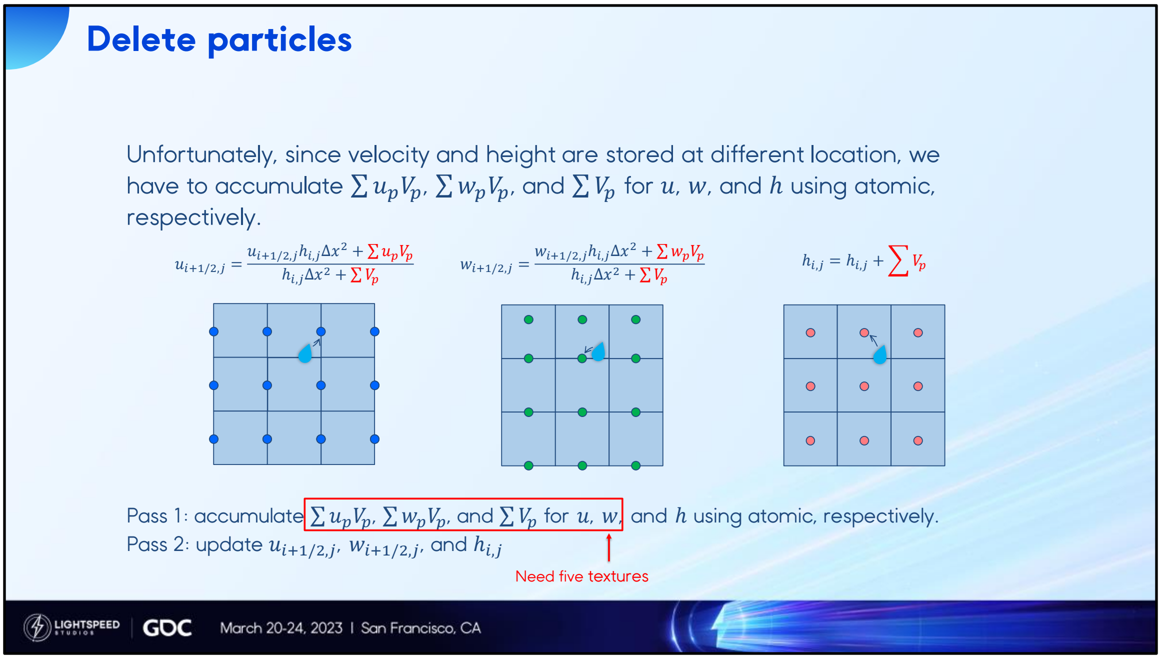
Total mass

Pass 1: accumulate $\sum u_p V_p$ and $\sum V_p$ using atomic, respectively.
Pass 2: update $u_{i+1/2,j}$, $w_{i+1/2,j}$, and $h_{i,j}$

Which require us to have two textures for grid momentum and mass



Unfortunately, since velocity and height are stored at different location, we have to accumulate $\sum u_p V_p$, $\sum w_p V_p$, and $\sum V_p$ for u, w, and h using atomic, respectively.



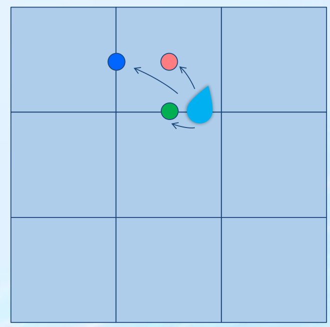
Unfortunately, since velocity and height are stored at different location, we have to accumulate $\sum u_p V_p$, $\sum w_p V_p$, and $\sum V_p$ for u, w, and h using atomic, respectively.

Delete particles

We couple u, w, and h together when writing.

Pass 1: accumulate $\sum u_p V_p$, $\sum w_p V_p$, and $\sum V_p$ for h, respectively.

Pass 2: update $u_{i+1/2,j}$, $w_{i+1/2,j}$, and $h_{i,j}$



LIGHTSPEED STUDIOS | GDC March 20-24, 2023 I San Francisco, CA

To simplify this operation, we couple the velocity and height together, by assuming the water particle will contribute to those three variables, simultaneously. Then, we can only accumulate momentum for the velocity at each direction and height, respectively. By that, we only need three textures and reuse the total mass when update the velocity at the second pass.

Delete particles

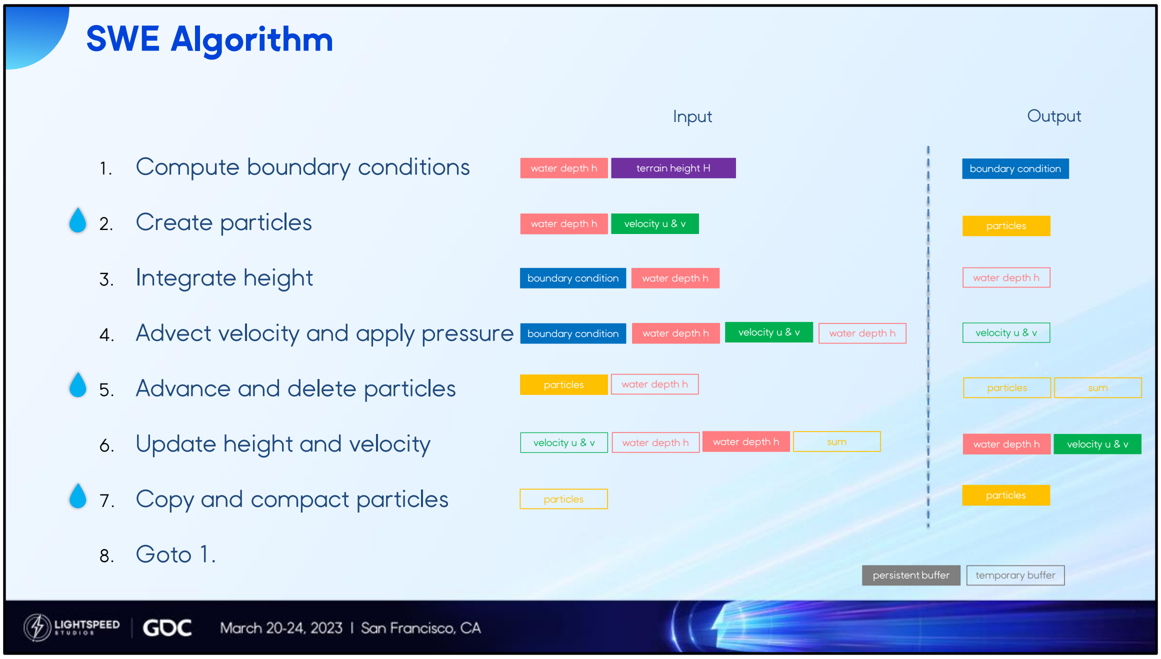
We couple u, w, and h together when writing.

Pass 1: accumulate $\sum u_p V_p$, $\sum w_p V_p$, and $\sum V_p$ for h, respectively.

Pass 2: update $u_{i+1/2,j}$, $w_{i+1/2,j}$, and $h_{i,j}$

By that, we only need three textures without much visual loss.

Here is the Full Algorithm for SWE with Particles. it is a small waterfall example that uses 200 x 200 grid and 20k+ particles and takes 0.11 ms per step. [17.5mins]



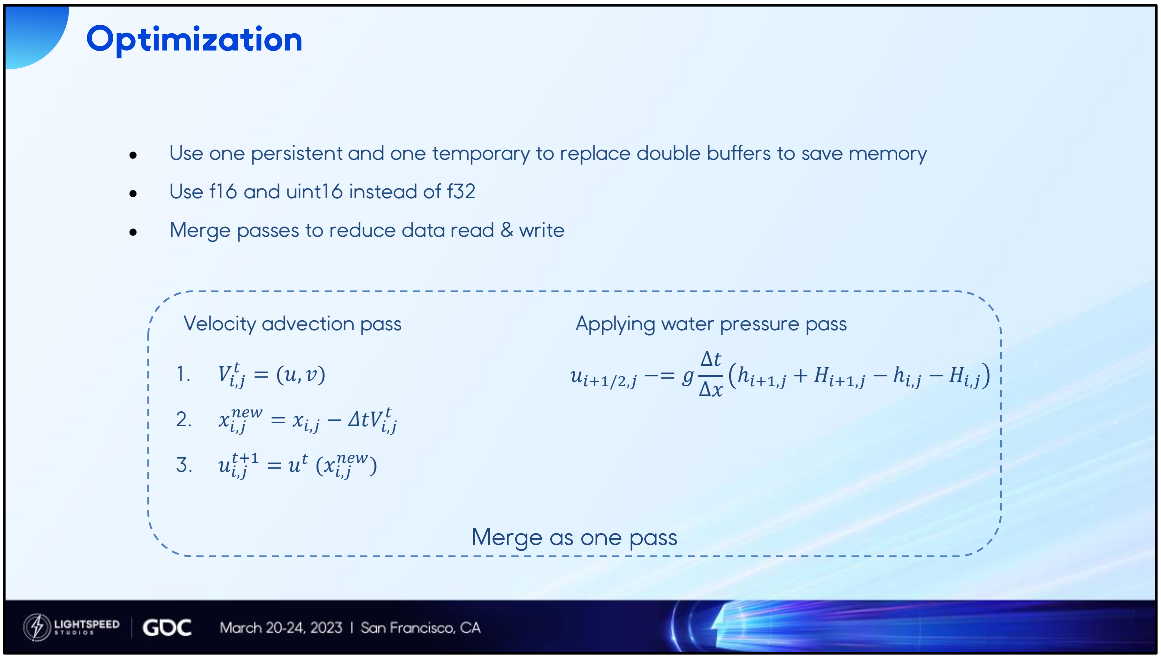
Here is the full swe algorithm. The passes labeled by the water droplet are for particles only. We also list the input and output for each pass.To save the memory usage, we use one persistent buffer and one temporary buffer to replace the double buffer. So, each frame the temporary buffer can be obtained from the texture pool.

Optimization

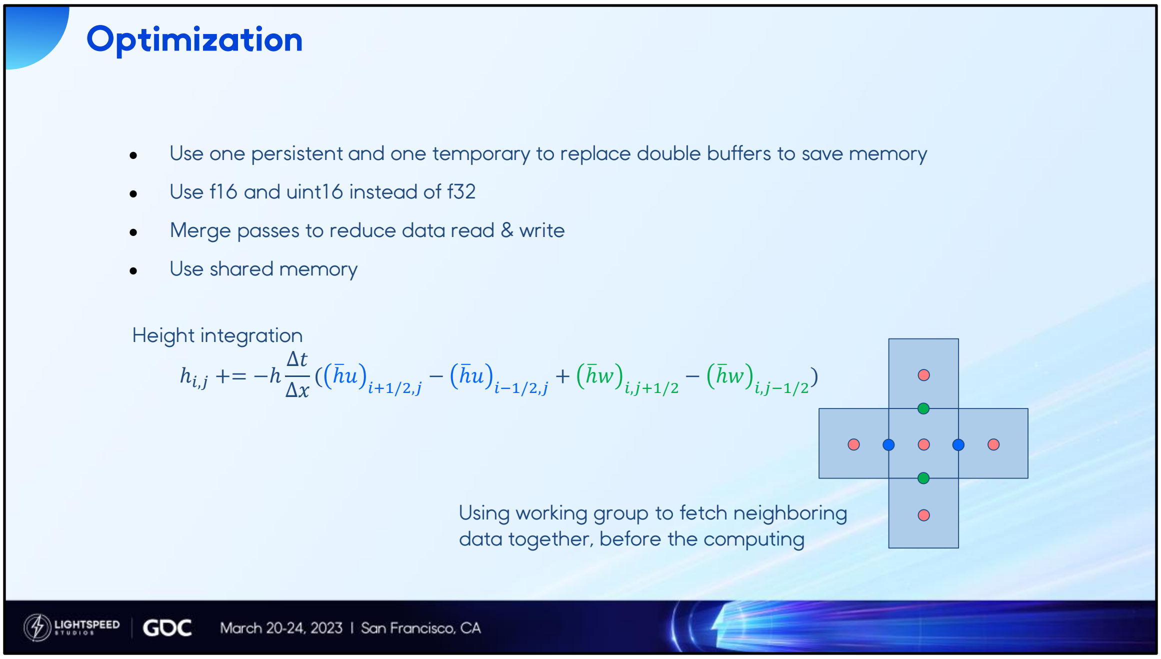
- Use one persistent and one temporary to replace double buffers to save memory
- Use f16 and uint16 instead of f32
  - N+1 x N+1 texture buffers x 4, including
    - PF_R16G16_FLOAT x 1 -- velocity u & v
    - PF_G16_FLOAT x 1 -- water depth h
    - PF_G16_UINT x 1 -- boundary condition
    - PF_G16_FLOAT x 1 -- terrain height H
  - 3 x 65536 texture buffers x 2, including
    - PF_R16G16B16_FLOAT x 1 -- particle position
    - PF_R16G16B16_FLOAT x 1 -- particle velocity
    - PF_G16_FLOAT x 1 -- particle lifetime

LIGHTSPEED STUDIOS | GDC March 20-24, 2023 | San Francisco, CA

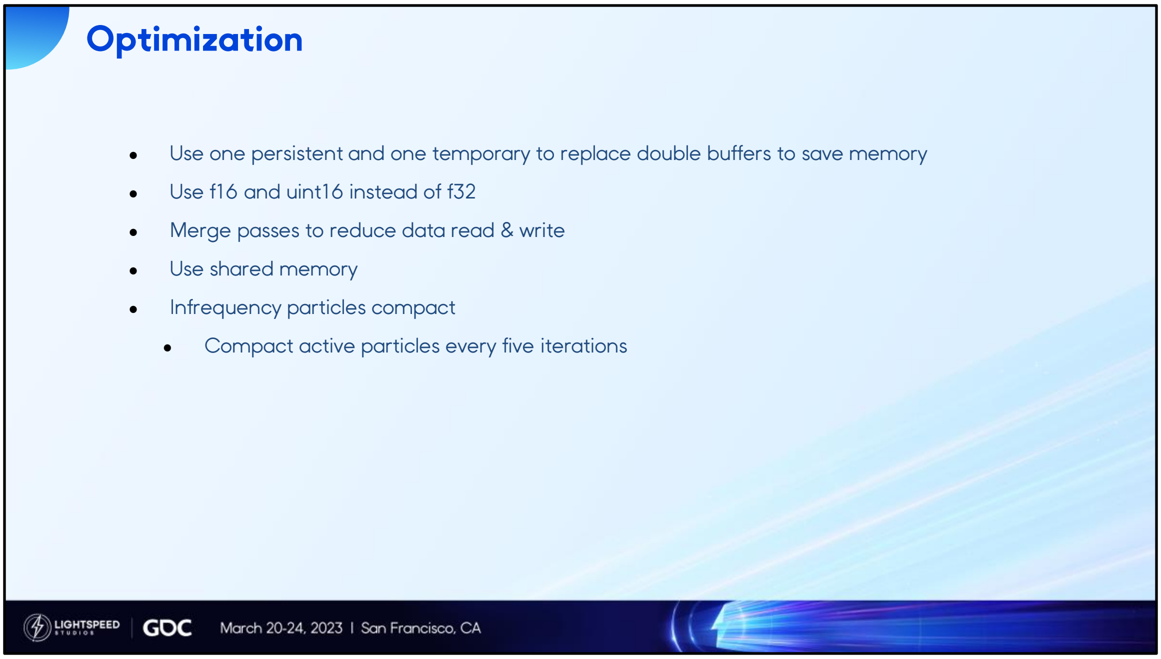
Besides, we also did lots of optimization, such as replace all f32 with 16bits float and integer. There are total four n+1 by n+1 texture needed, for velocity, water depth boundary condition and terrain height. For particles, we assume there are 65536 particle at max. and use three buffers for position, velocity and lifetime.



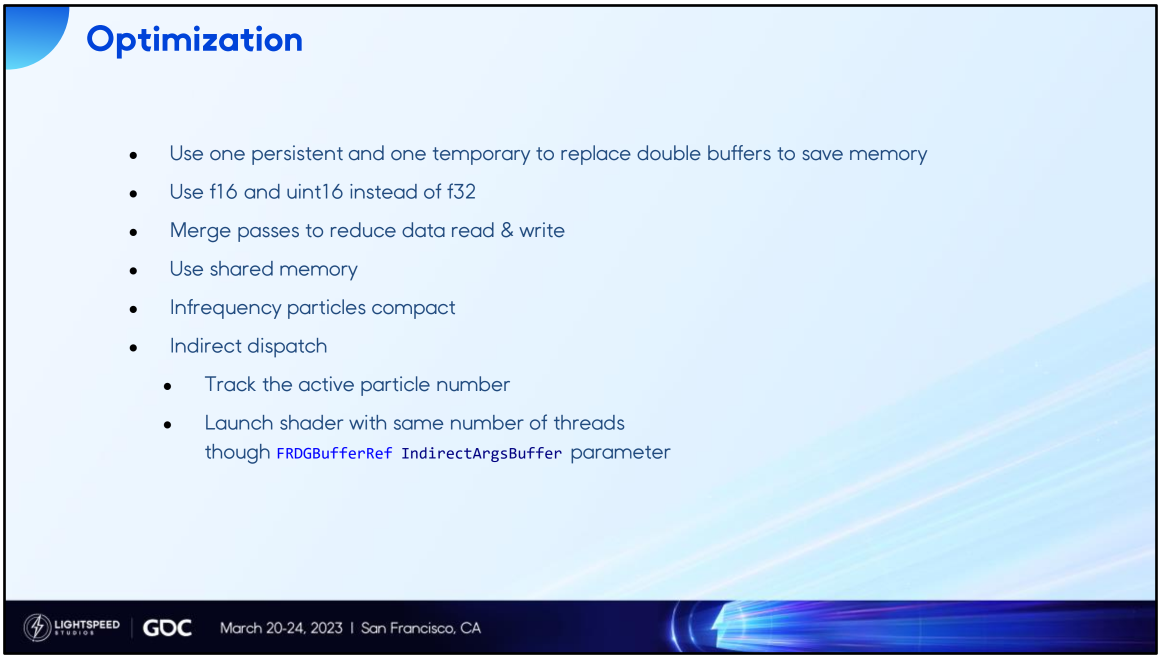
We also attempted to merge compute shader passes as many as possible to reduce the data read and write, such as the velocity advection pass and applying water pressure pass. Since they don't have any write or read conflicting, we can simply combine them together



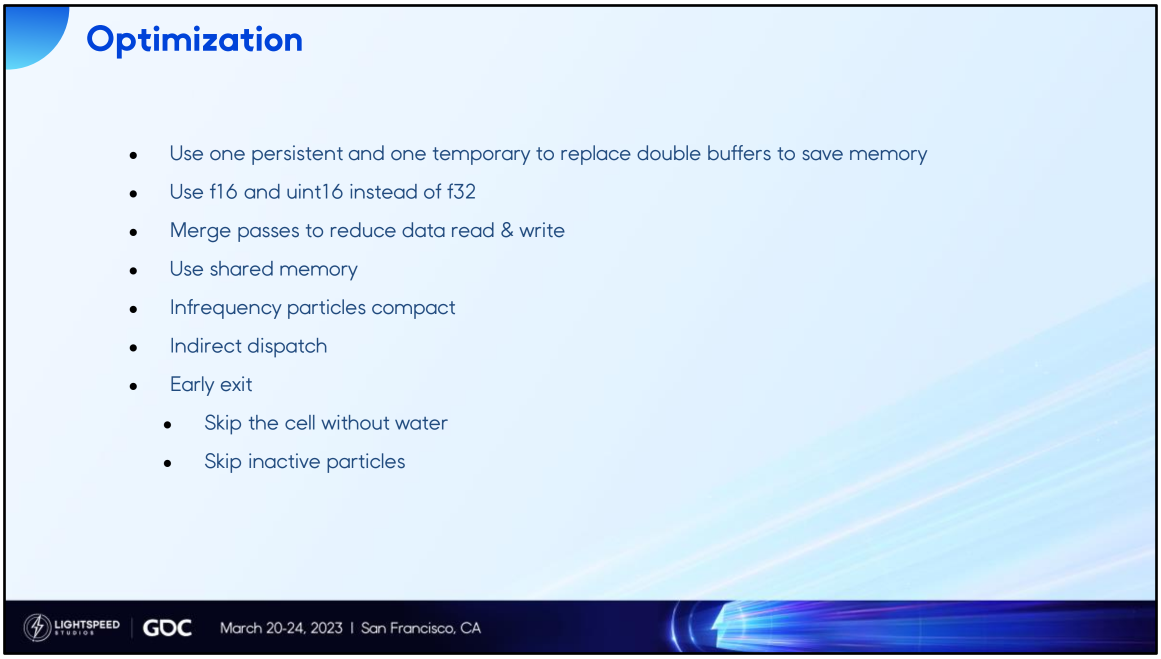
Another well-known optimization strategy is to use shared memory pre-fetch a chunk of data using working groups. Such as the height integration here, update height needs to data from all neighboring velocity and water height, which can be fetched together.



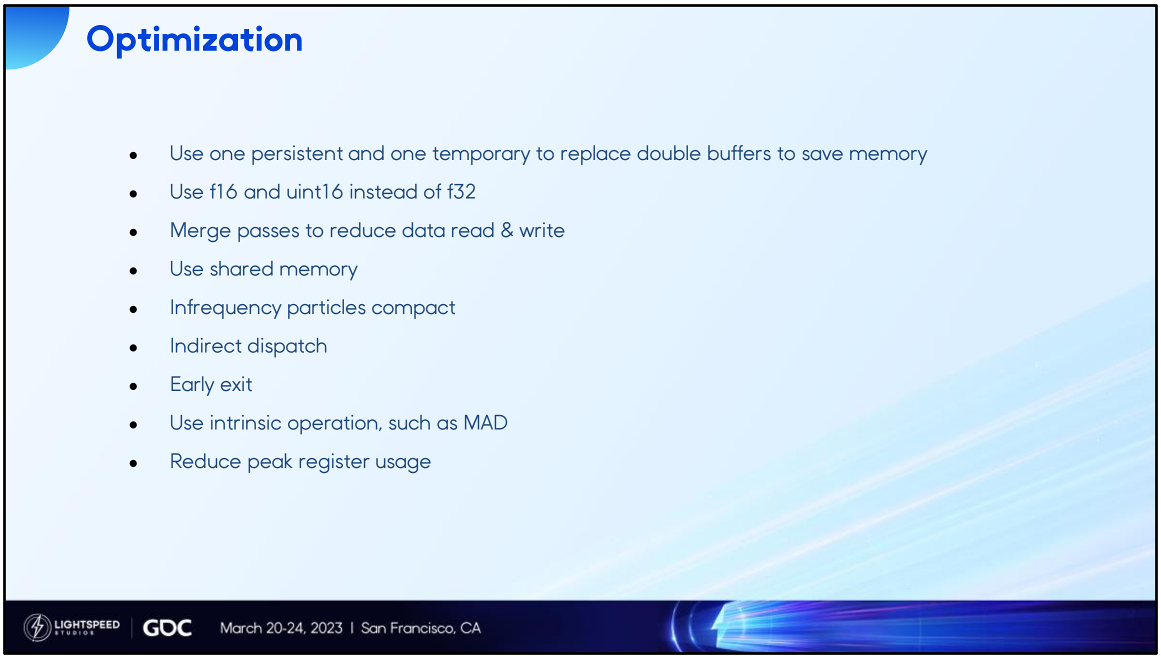
When particles hit the ground or water, we will mark the particles inactive. Ideally, we should compact the particle list every frame to save the memory. However, compacting particle is a very expensive operation. So, we compact active particles every five iterations



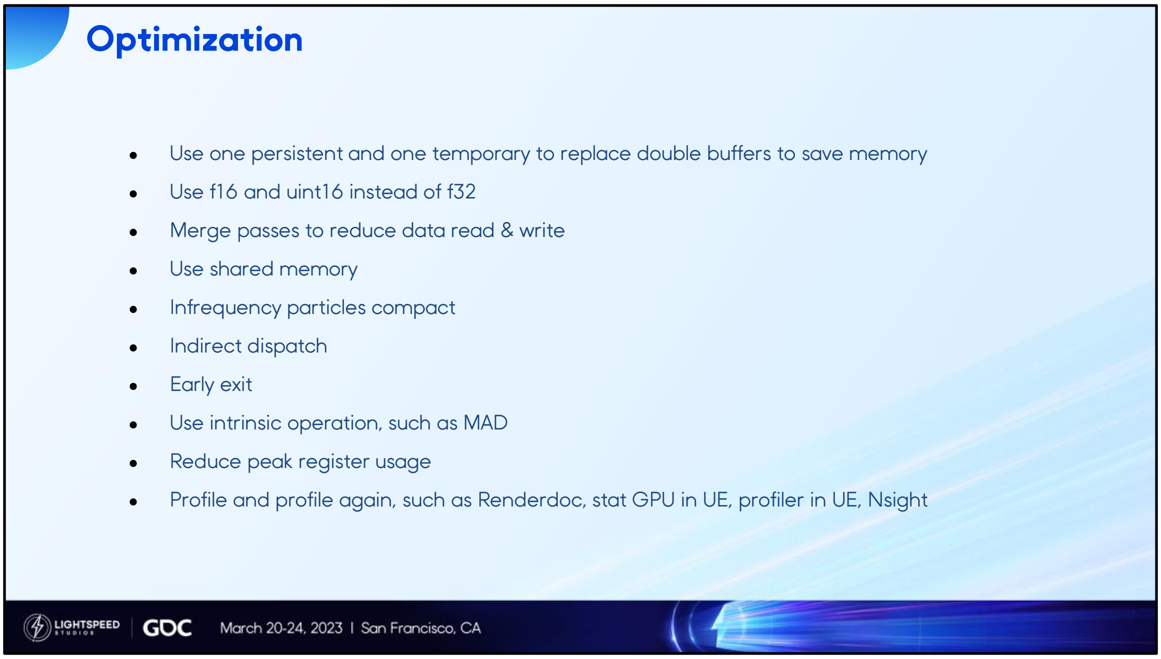
To avoid waster threads on inactive particles, we keep tracking the Track the active particle number particles, and launch shader with same number of threads as active particles have using unreal indirect dispatch feature.



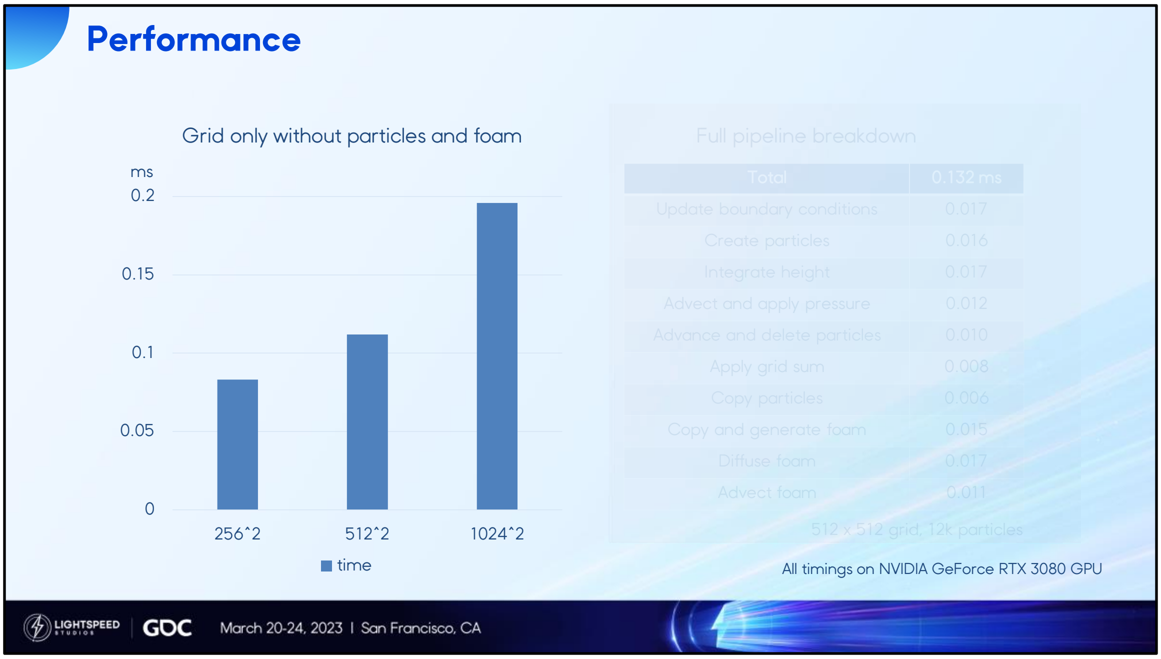
Also, we found early exit can benefit the performance a lot, such as Skip the cell without water and Skip inactive particles. When updating the particles, we mark the lifetime as -1 for inactive particles.



We also use intrinsic operation, such as MAD and profile the shader code to locate when is peak register usage and reduce the usage to increase number of warps launched



The final thing I'd like to mention is to keep profiling the shader code, such as Renderdoc, stat GPU in UE, profiler in UE, Nsight. That is the only way to improve the performance.



On the left, we test the scalability of our implementation with different domain size. As shown in the bar chart, even for the domain with 1024^2, our implementation still takes less than 0.3 ms per step.

Performance

Full pipeline breakdown

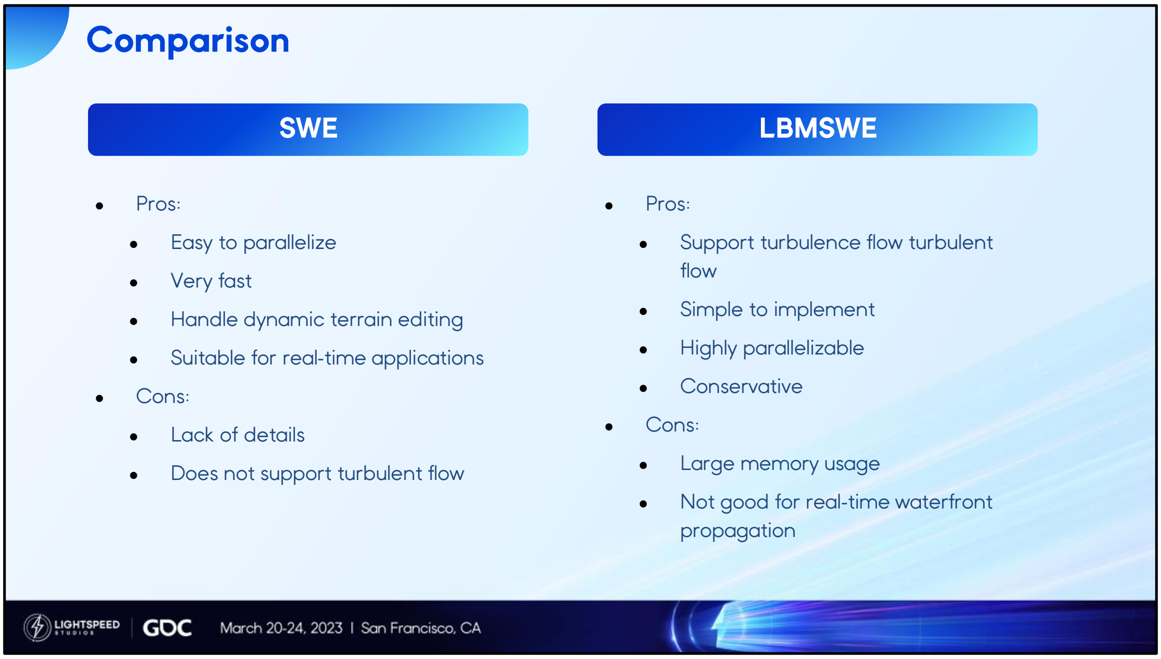
Total	0.132 ms
Update boundary conditions	0.017
Create particles	0.016
Integrate height	0.017
Advect and apply pressure	0.012
Advance and delete particles	0.010
Apply grid sum	0.008
Copy particles	0.006
Copy and generate foam	0.015
Diffuse foam	0.017
Advect foam	0.011

512 x 512 grid, 12k particles

All timings on NVIDIA GeForce RTX 3080 GPU

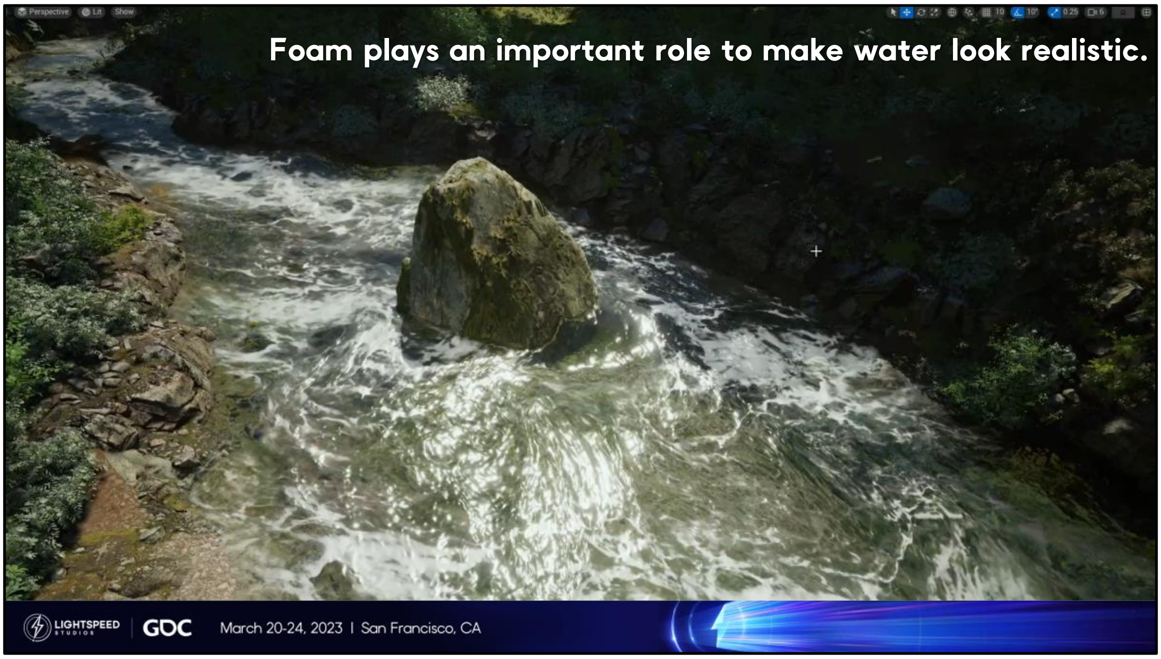
LIGHTSPEED STUDIOS | GDC March 20-24, 2023 I San Francisco, CA

On the right, there is the full pipeline breakdown table.



To summary, SWE and LBMSWE both have their own advantages and disadvantages. We plan to combine the real-time SWE and offline LBMSWE baking together in the future.

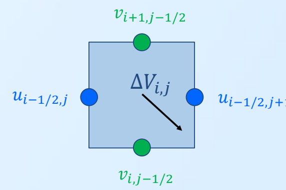

Agenda
Fluid simulation
Offline Flow map baking
Runtime fluid simulation
Grid-based foam simulation
Open world water rendering
Rendering pipeline
Surface wave
Tessellation
LIGHTSPEED STUDIOS
GDC
March 20-24, 2023 | San Francisco, CA


Foam plays an important role to make water look realistic.
LIGHTSPEED STUDIOS
GDC
March 20-24, 2023 | San Francisco, CA

Grid-based foam simulation

- Foam generation
  - Trap air – velocity difference within the face

$$\phi_{ta} = \left\|\left(u_{i-1/2,j+1} - u_{i-1/2,j}, v_{i+1,j-1/2} - v_{i,j-1/2}\right)\right\|$$




```
float GenFoamTrapAir(int2 xy)
{
    float Vx00 = ReadInVelX(xy + uint2(0, 0));
    float Vx10 = ReadInVelX(xy + uint2(1, 0));
    float Vy00 = ReadInVelY(xy + uint2(0, 0));
    float Vy01 = ReadInVelY(xy + uint2(0, 1));
    float2 VelDiff = float2(Vx10 - Vx00, Vy01 - Vy00);
    return FoamClampingFunc(length(VxDiff), FoamParas2.x, FoamParas2.y) * FoamParas2.z;
}

float FoamClampingFunc(const float I, const float tau_min, const float tau_max)
{
    return (min(tau_max, I) - min(tau_min, I)) / (tau_max - tau_min);
}
```

Ihmsen, Markus, et al. "Unified spray, foam and air bubbles for particle-based fluids." *The Visual Computer* 28 (2012).

We develop a **Grid-based foam** simulation. For foam generation, we use two heuristics. The first is in this regions have high turbulence that Air is trapped by impacts. We simply use the magnitude of relative velocities within the cell to determine how much foam can be generated. In here, we follow the previous work to use a clamping function to adjust the result

Grid-based foam simulation

- Foam generation
  - Trap air – velocity difference within the face

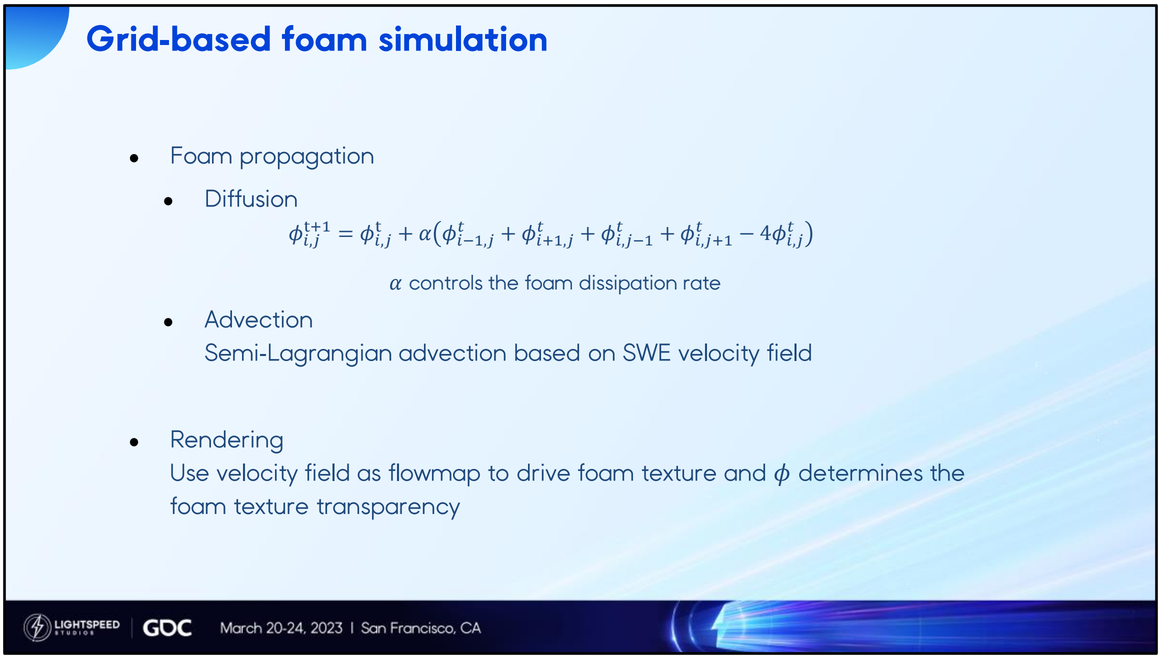
$$\phi_{ta} = \left\|\left(u_{i-1/2,j+1} - u_{i-1/2,j}, v_{i+1,j-1/2} - v_{i,j-1/2}\right)\right\|$$

  - Steep enough to break – gradient of water level

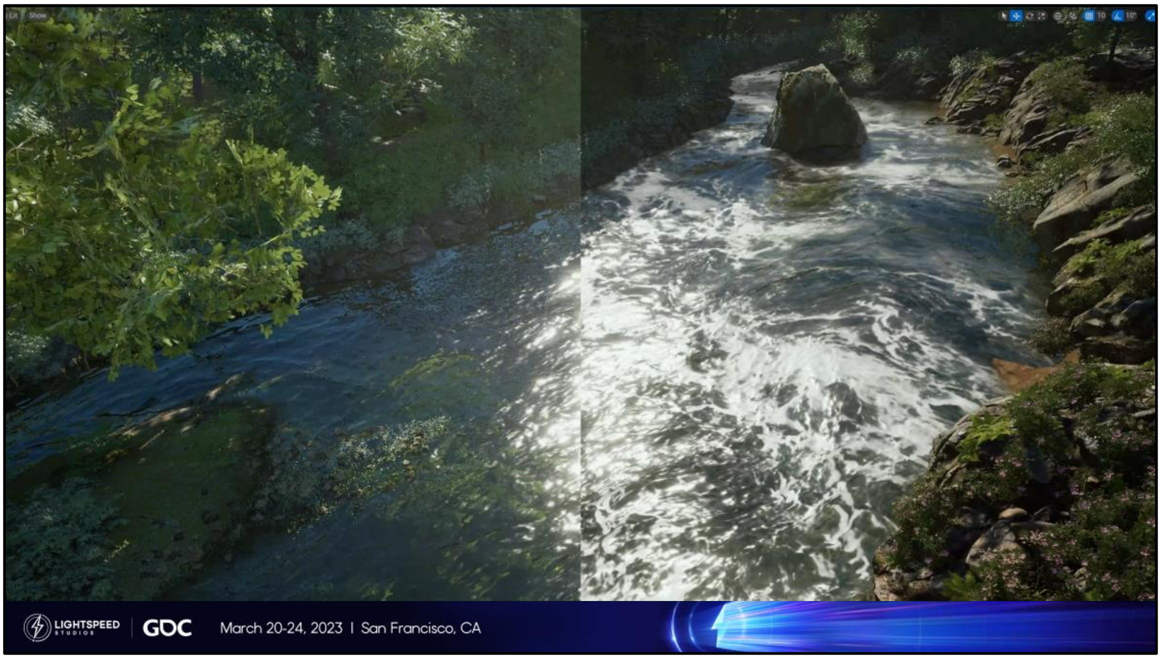
$$\phi_{steep} = \left\|\left(\frac{\partial H + h}{\partial x}, \frac{\partial H + h}{\partial y}\right)\right\|$$

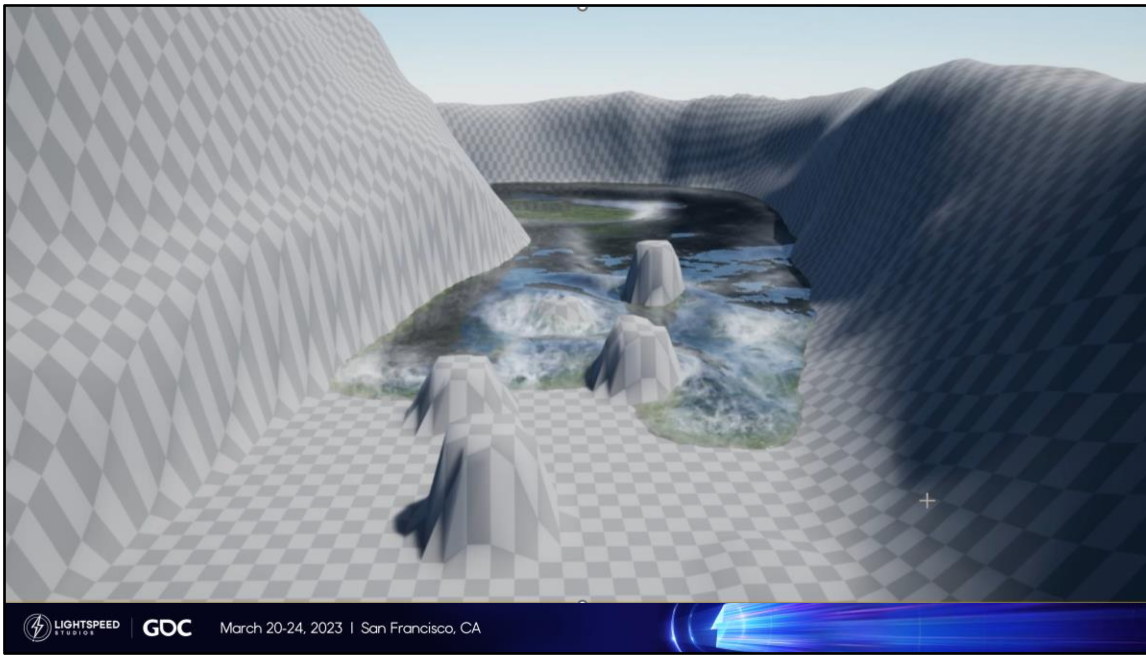
```
float GenFoamSteepToBreak(int2 xy)
{
    float Eta11 = h[xy + uint2( 0,  0)] + H[xy + uint2( 0,  0)];
    float Eta01 = h[xy + uint2(-1,  0)] + H[xy + uint2(-1,  0)];
    float Eta10 = h[xy + uint2( 0, -1)] + H[xy + uint2( 0, -1)];
    float Eta21 = h[xy + uint2( 1,  0)] + H[xy + uint2( 1,  0)];
    float Eta12 = h[xy + uint2( 0,  1)] + H[xy + uint2( 0,  1)];
    float2 GradEta = overdx * float2(abs(Eta21-Eta11)>abs(Eta11-Eta01)?Eta21-Eta11:Eta11-Eta01,
                                     abs(Eta12-Eta11)>abs(Eta11-Eta10)?Eta12-Eta11:Eta11-Eta10);
    return FoamClampingFunc(length(GradEta), FoamParas2.x, FoamParas2.y) * FoamParas2.z;
 }
```

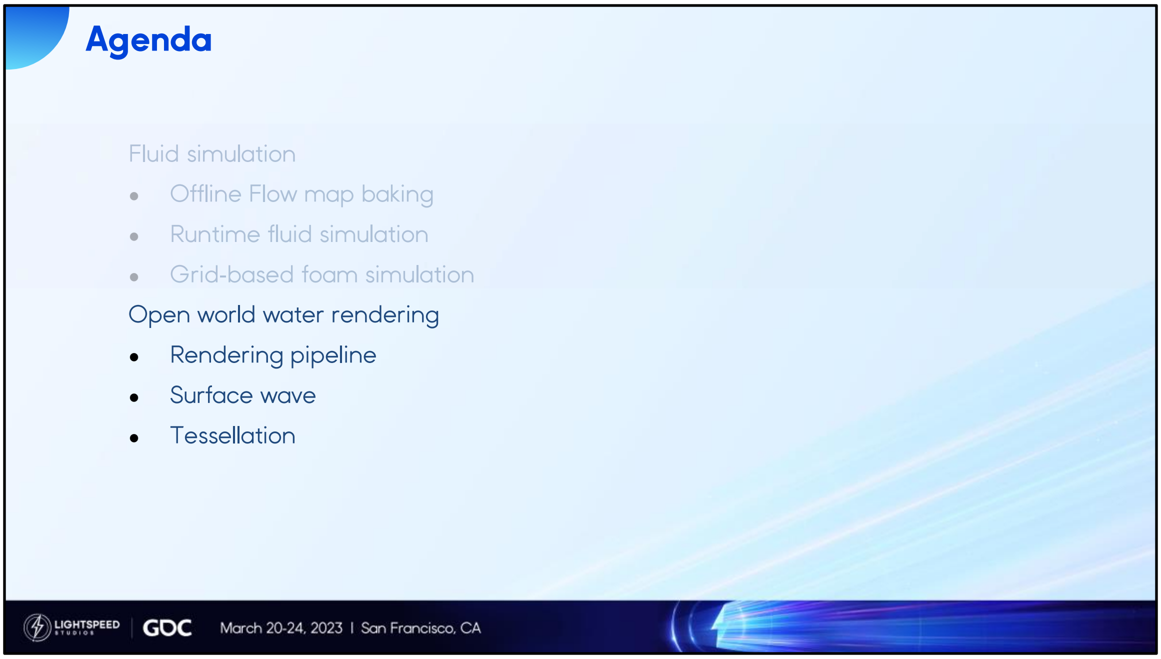

The second is In this regions have high curvature that the water is unable and steep enough to break. We use the magnitude of gradient of water level how much foam density can be generated



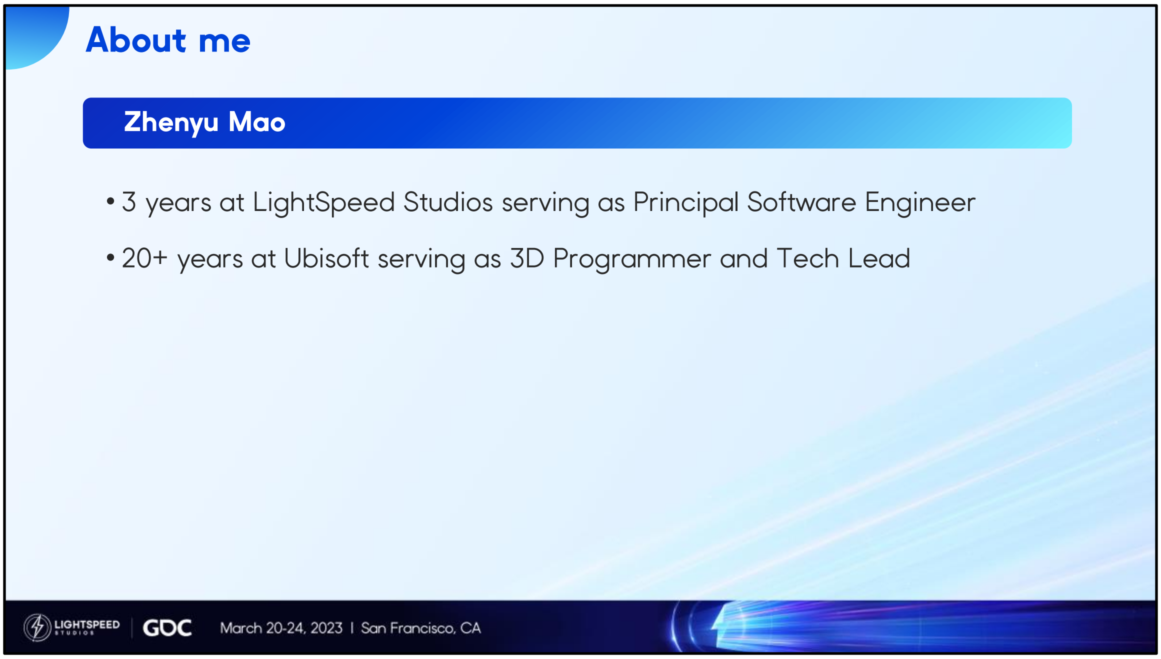
Foam simulation is similar to eularian grid fluid simulation. It contains a diffusion step that can be control by alpha for the foam dissipation rate. Luckly, we don't have to solve the imcompressible velocity field. Instead, we can reuse the velocity field generated by SWE to advection the foam. For rendering, we use velocity field as flowmap to drive foam texture and foam density to determine the foam texture transparency

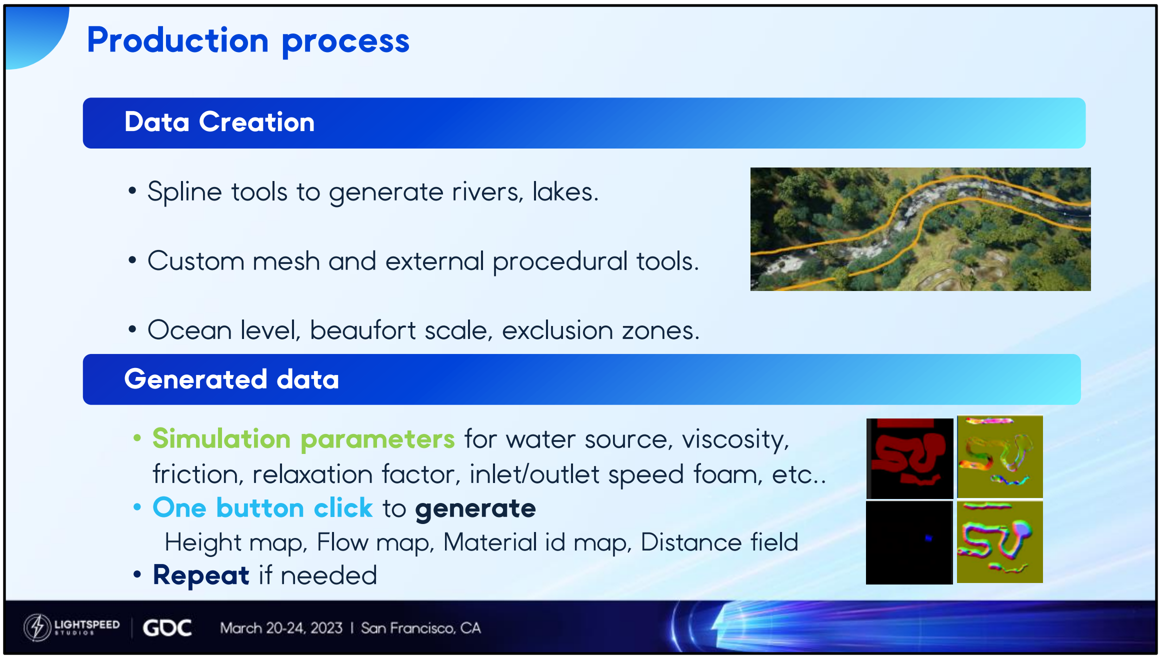

LIGHTSPEED STUDIOS
GDC
March 20-24, 2023 | San Francisco, CA


LIGHTSPEED STUDIOS
GDC
March 20-24, 2023 | San Francisco, CA



Here is the agenda today. I will first introduce the components for fluid simulation

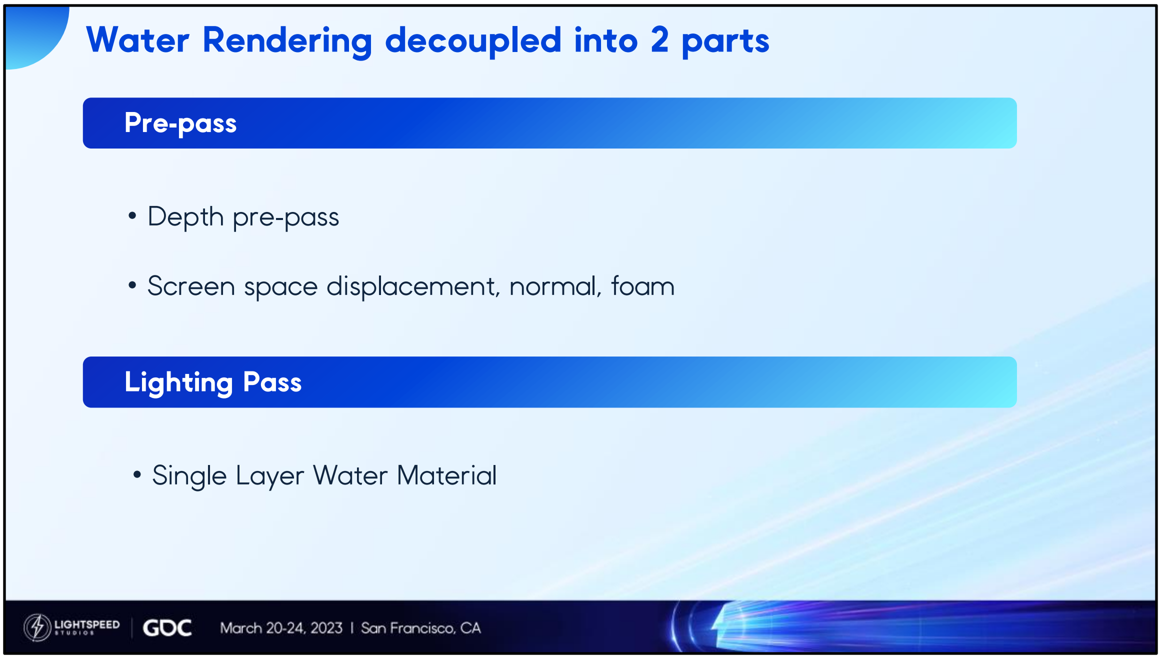

About me
Zhenyu Mao
• 3 years at LightSpeed Studios serving as Principal Software Engineer
• 20+ years at Ubisoft serving as 3D Programmer and Tech Lead
LIGHTSPEED STUDIOS
GDC
March 20-24, 2023 I San Francisco, CA



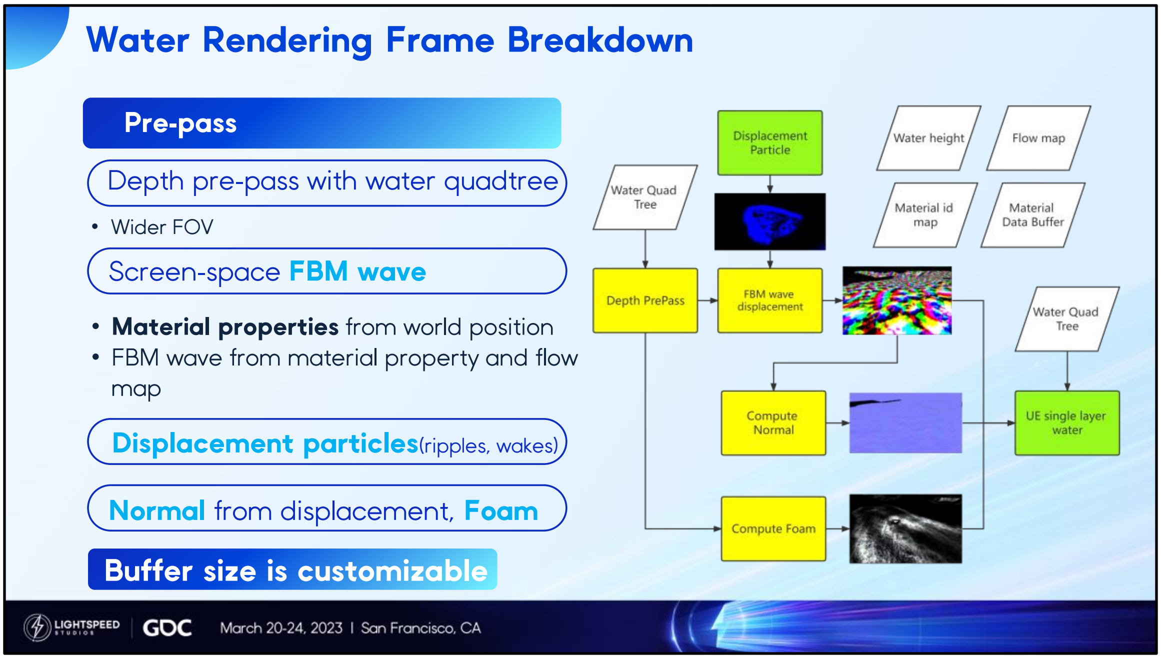
Before dive into the rendering, just a quick introduction about the production pipeline, without data we can not run any simulation or rendering.

- Artists can use spline tools to create rivers and lakes,
- or generate water procedurally using external tools like Houdini.
- Alternatively, they can assign the water material to any mesh.
- At the end. all the water meshes will be **baked into height maps**.
- For ocean, we just need to set the ocean level, and some exclusion zone to avoid some area get flooded.

Once the desired water sources have been added, the fluid simulation can be initiated with the click of a button.
The simulation results are saved as water height, flow maps, material id maps, and distance field will also be generated for rendering.
This process can be repeated until get a good result



Instead of squeeze in all the generated resources into one shader, we decoupled the rendering pass to 2 passes.
The first pass will the water displacement , normal and foam in 3 screen space buffer in 3 passes.



Here is a break down of how we render a frame for water

Pre-pass

- A **depth-only** pre-pass is first rendered with the water quadtree and height map.
- it uses a **simplified VS** to render a flat water surface.
- The pre-pass is rendered with a **wider FOV** to avoid the gap on the screen border after displacement is applied.
- After the depth pre-pass, we can un-project the depth to get world position for the following passes.

FBM wave

- We use FBM wave to simulate the **detailed waves**,
- FBM stands for **Fractional Brownian Motion**, which is a random motion wave computed by multiple iterations,
- the wave can be tweaked by **amplitude, frequency, and speed** to represent calm lake or rapid water.
- We store the FBM **wave parameter** in the water material data buffer.
- We can use the world position to get the material id from the material id map, then use it to index from the material data buffer to get the material

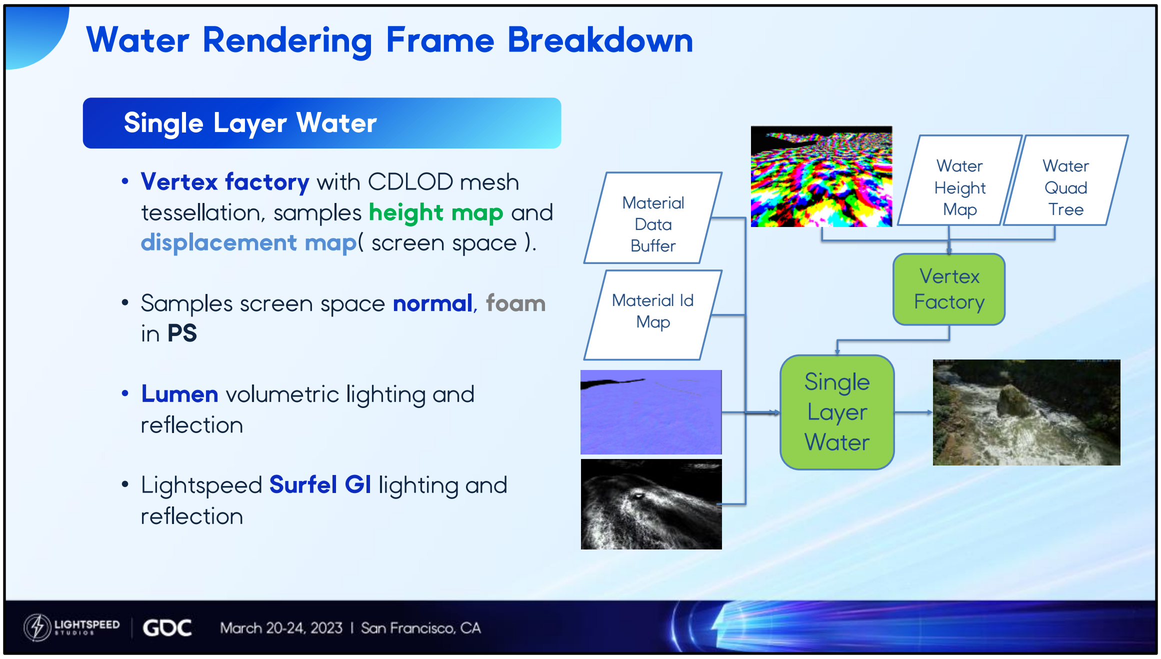
properties.
- Flow map is used to control the global movement of the fbm waves.

Displacement particle
- We added a **new material type** in the Unreal **Niagara** particle system.
- It will render particles to a **specified displacement buffer** instead of a color buffer.
- This buffer will be **combined with FBM** displacement as the final displacement.

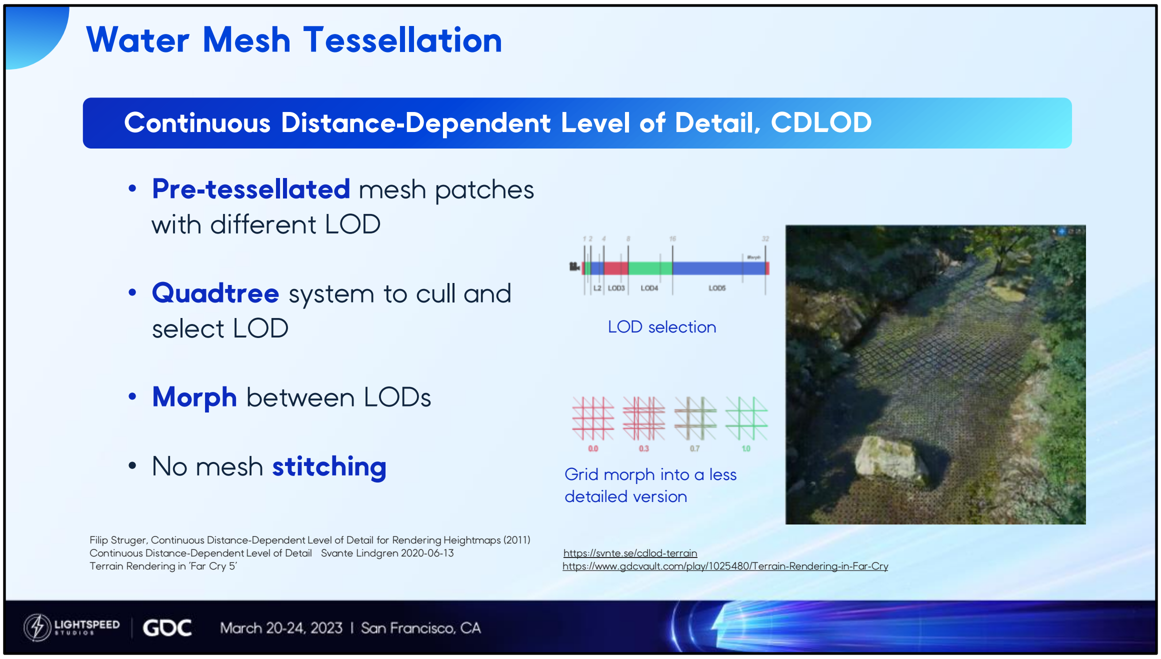
Normal is generated from the final displacement buffer.

Foam
Foam is also **de-coupled** and rendered to a separate buffer.

The pre-pass **buffer size** does not need to be a one-to-one match with the GBuffer size.
- This is the biggest reason we chose to **de-couple** the wave computation from the lighting pass,
- We can selectively use a smaller buffer size for the low-end platform.



A **custom vertex factory** is created for the **CDLOD** mesh tessellation, we applied height map and the **displacement** on it.
The **material shader** will sample from the **screen space** normal and foam, with **FOV corrected** of course.
As we use the single layer water, we can easily get benefit from the Lumen lighting and reflection.
We also support the lightspeed studio internal tech **Surfel GI** which provided a similar real-time GI as Lumen but at a lower cost.



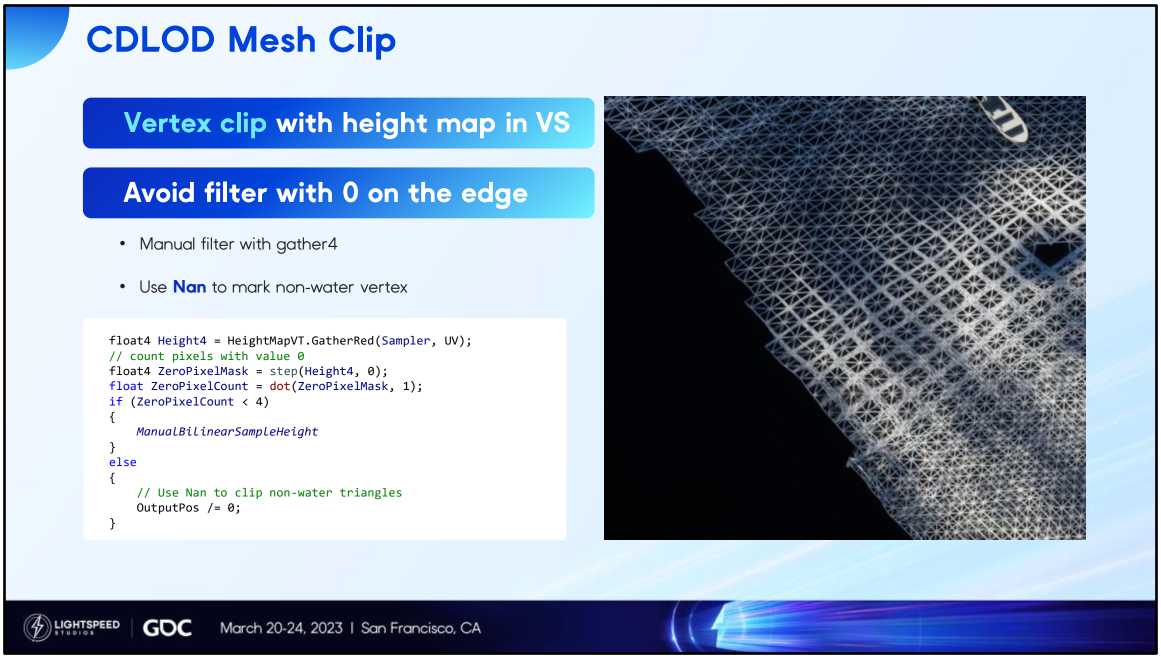
CDLOD is one of the popular methods used for **rendering height maps**.
It p**re-generated** multiple LODs of the mesh patch and organized the meshes in a **quadtree**.
At runtime, it **selects the LOD** based on the distance to the camera, so the high-density mesh is used close to the camera and lower LOD is used in distance.

Morph

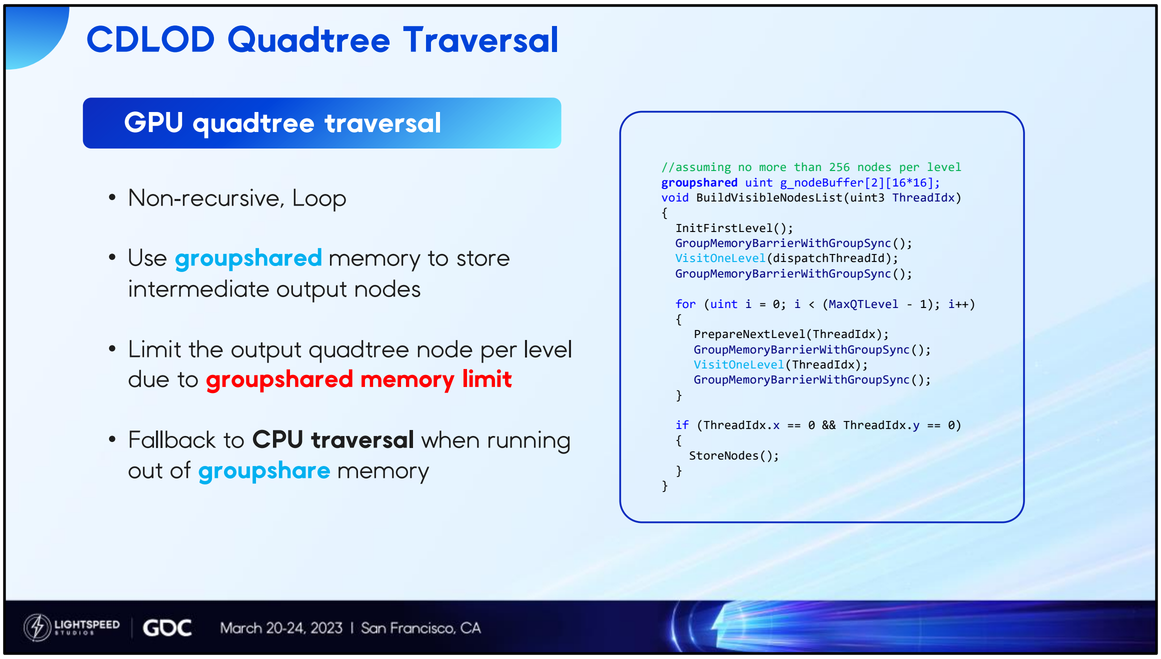
- **No stitching** between LODs.
- Just uses a 0 to 1 morph value to morph between LOD levels.
- The cost of the vertex shader is just like regular mesh rendering plus height map sampling.

CDLOD limit

Compare to the **stitching** method used in farcy 5 terrain, CDLOD is simpler but has its limit.
With CDLOD, the connected LODs can only have **one level difference**, while the **stitching method allows more than 2 levels** of difference, so the LOD level can drop faster.
With CDLOD, it may get unnecessary **high mesh density at a very far distance** which we will address later.



Unlike terrain height map rendering, a water height map only covers a part of the land and needs to **be clipped** at the shoreline.
We use a **bilinear filter** to sample the water height map to get **smooth** results on the slope,
but it will be wrong on the edge when **filtered with an invalid height value**, in our case is 0.
We use a **gather4** to do a manual filter which excludes the 0 value pixels.
if the height value is 0, we use the **divide-by-zero** trick to create a nan on the vertex position, the whole triangle will be **clipped**.
It is an undocumented feature but works surprisingly well and it works everywhere, even on the mobile platform.
The water mesh needs to be expanded so it will not leave a gap near the shore in low LOD.



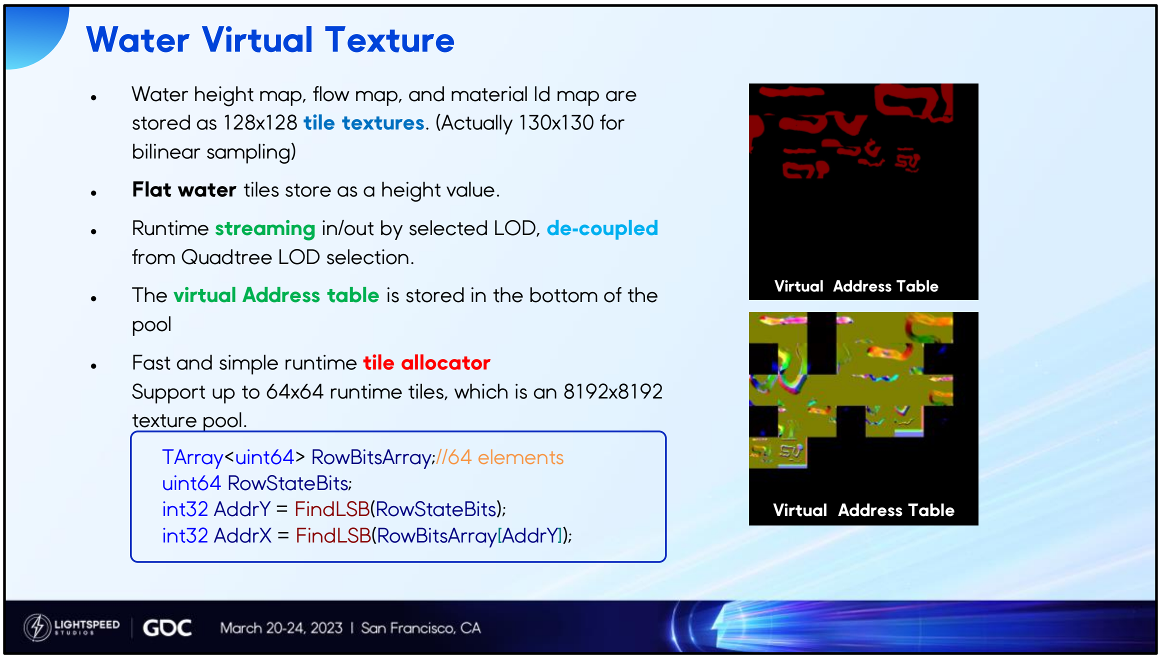
Quadtree traversal is done in **GPU**
We traverse the quadtree in **one compute shader dispatch.**

It uses a **loop** method to traverse one level per iteration,

- In each iteration, it compute the visible nodes and store into the **group shared memory buffer** for the next iteration,
- After all the quadtree level is visited, the group shared memory will be copied to the **output node buffer**,
- the **indirect draw argument** will be updated as well.

Per level Node Limit

- This compute-shader has a **group size of 16x16** with only **one group**.
- Each thread can access all the group shared memory.
- But there is a hardware limit of group memory size per thread, which becomes **the limit of the max node count** that can be written into the buffer, **per level**.
- In our test with 12k x 12k world, the **visible node buffer never runs out**.
- We added a **fallback** solution that once the node buffer is overflowed, the system will use **CPU traversal** in the next frame.



To support a large open world, we can not store the height map or flow map textures in **one large texture**,

Tile Texture

- We cut all the textures into **128x128 tile textures**.
- Actually, stored them as 130x130 textures with added one-pixel **border**, so the **bilinear** filter on the border of the tile texture not will go across the tile border.
- The **resolution** of the texture is one pixel per square meter at the highest LOD.
- **Lower LOD** texture is also exported in 128x128 textures but covers the bigger area, like LOD1 texture will cover 4 squared meters per pixel.
- **Tiles without water** will be skipped
- **Flat water tile**, if all the pixel in the tile has the same height, only a height value will be stored.

Runtime

- The tile textures will be **selected** based on the distance to the camera,
- then they are composed into the **virtual texture pool**.

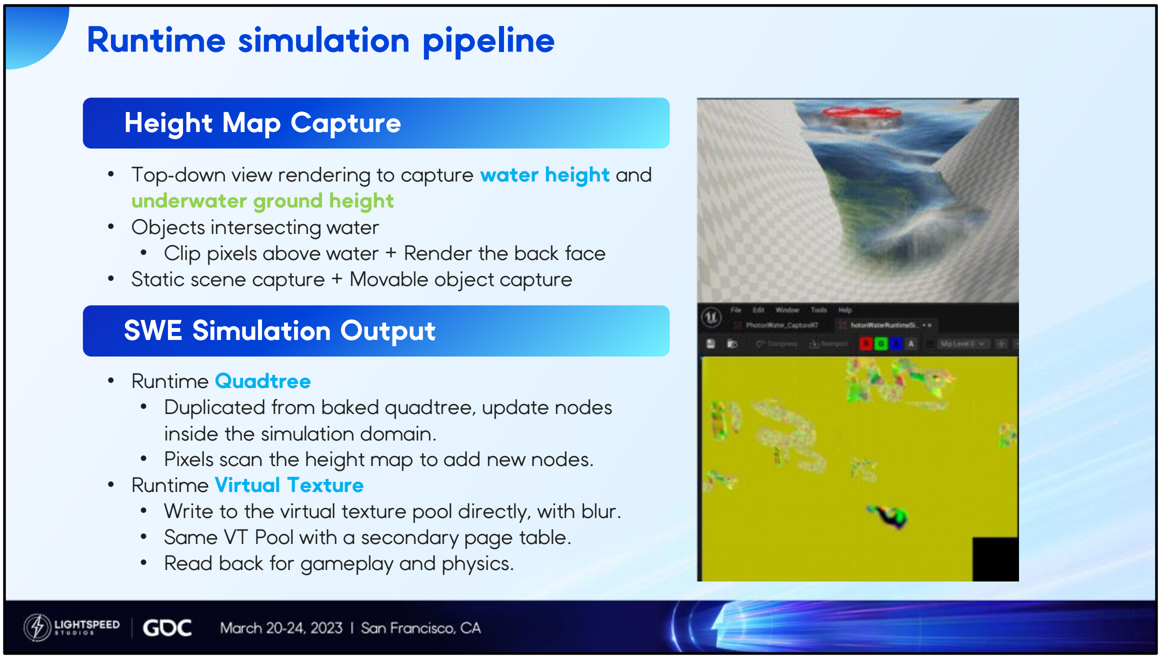
- The **texture pool size** can be scaled from 1k to 8k depending on the project requirement.

Virtual Address Table

Since the actual tile texture is 130x130 which is not the power of 2, there is **empty space** in the pool texture on the right and the bottom part. We use this space to store the virtual address table which maps world space position to VT Pool texture UV.

Tile allocator

- We implement a very fast and simple tile allocator.
- It assumes there are up to **64x64 tiles.**
- **W**e use a **64x 64bits bit array** to store if a tile is used or not
- And another **uint64** is used to store the **row bits**.
- The bit array is initialized as all 1, which means free.
- The allocation always happens on the **least significant non-zero bit**,
- We can use **2 intrinsics** like **_BitScanForward64** to get the address quickly, then mark the bit as 0(occupied) and return the address
- Each allocation will **only allocate one tile**,
- By this way, we can guarantee all the **allocations can find the first available tile**.
- To **Free** the tile**,** just mark the bit to 1.



To make the runtime simulation work, we need to first capture the height map for the simulation, then output the simulation results to the rendering engine. We wrote a **custom height map** capture shader instead of using the unreal scene capture which is very expensive.

- A top-down view rendering pass is used to render the water and terrain.
- Then render all the static meshes that intersecting water to cut holes on water, a regular top-down view rendering will not work for the case that water goes under bridge or cave, it will be occluded completely.
- We do a **software depth clip** to discard all the pixels above the water and render the scene mesh's **back face** to get the precise cut.
- A second pass is rendered for the below water part of the object.

Static scene will only be rendered once, movable objects will be rendered every frame.

The **runtime quadtree** is duplicated from the baked quadtree but with all the nodes inside simulation domain removed.
We do a pixel counting on the height map to get the valid quadtree nodes, and update the quadtree dynamically, in GPU.

The **height map and flow map** are **exported into the virtual texture pool directly**.
We pre-allocate VT tiles for the whole simulation domain to make things easier as VT allocator runs on CPU.
and use a secondary page table,
so, we can change back to the baked water at any time and **blend** the runtime and the baked water at the border.

The quadtree update , VT table update, VT texture write out are all done in **one compute shader pass with group shared memory** optimization.

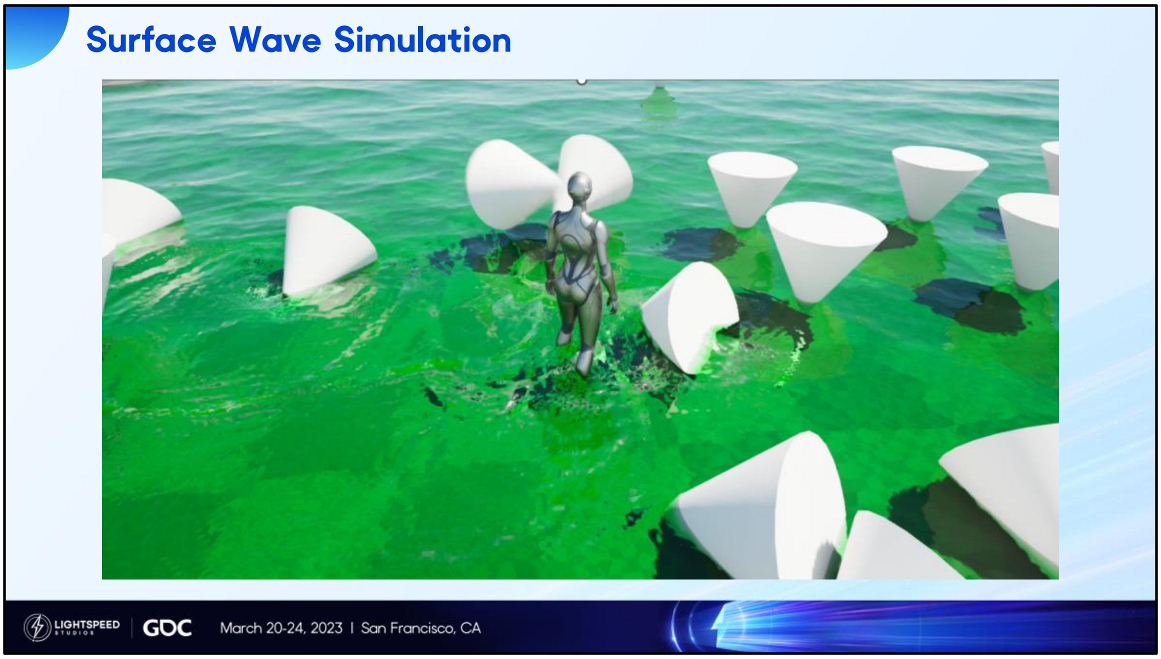
The Height Map and flow map are copied to staging texture for **readback in the next frame**, in tile mode, empty tiles will be skipped.

Performance

PS5

	SWE	Pre-Render	Single Layer Water
Time(ms)	0.5	0.5	1.8

- 4 sub steps in SWE, 512x512 simulation domain
- FBM wave displacement took 0.3ms in pre-render,
- Lumen reflection took 1ms+ in the single layer water



SWE can run on PC/console, we are still looking for a lightweight simulation that works for low-end and mobile platforms.
The idea is to take out all the time-consuming parts from the SWE, and only solve the pressure function to create an interactive wave.

Surface Wave Simulation

Only propagate the water height on static water surfaces for efficiency

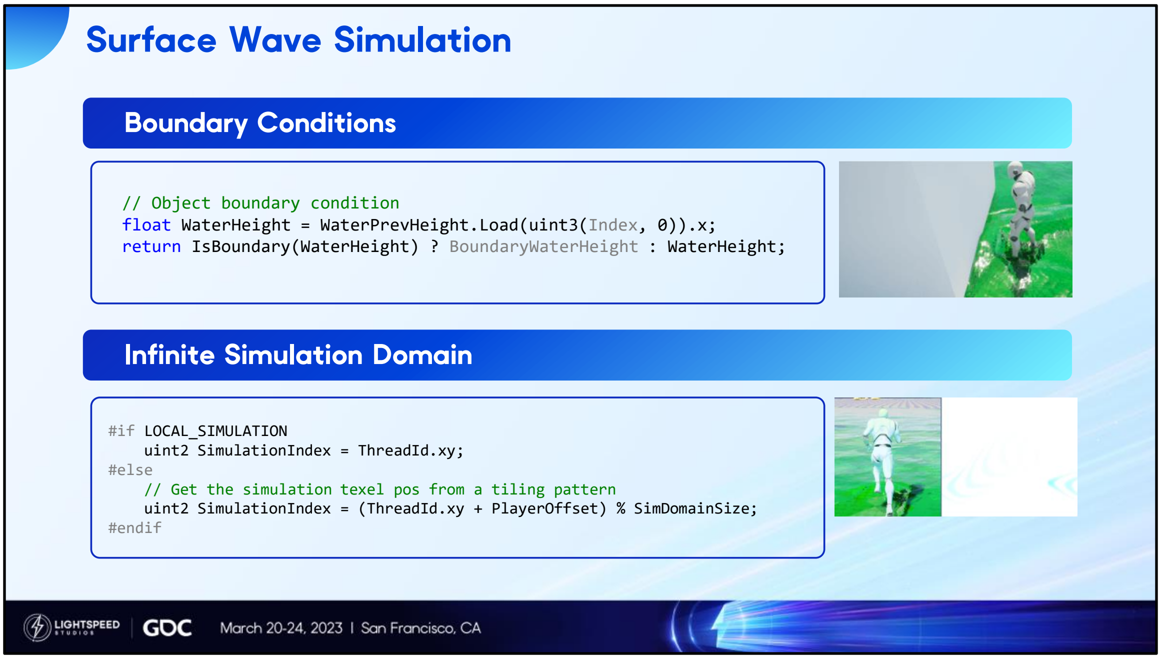
$$h^{t} = Damping * (h^{t-1} + \beta(h^{t-1} - h^{t-2}) + \alpha(h_N^{t-1} - 4h^{t-1}))$$

$\boldsymbol{\beta}$ is viscosity contant
$\boldsymbol{\alpha}$ is SWE constant
h_N^{t-1} is height sum of the nearest 4 points
t is the iteration number

- We only perform one cycle per frame to minimize computational cost and employ an additional damping factor for quick fading and to prevent explosion.

Kass, Michael, and Gavin Miller. "Rapid, stable fluid dynamics for computer graphics." *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990.

The wave simulation will only create a height map to displace the existing water surface visually, it will not change water height or flow map
The original implement will do multiple iterations in one frame to get a stable result.
But to get better performance, we only run one iteration per frame which cause some unstable issues and break the simulation.
That's why a damping value is used to avoid big height changes.



The boundary is easy to handle in surface wave simulation. The surface wave simulation use neumann boundary conditions to make waves bounce back. A neumann boundary condition specifies the values of the derivative applied at the boundary of the domain, which mean there should be no water exchange though the boundaries. When computing h_N^{t-1}, **if the pixel is outside the boundary, make its value to h^{t-1} to negate the water exchange**

We support two types of simulation domain

- One is a fixed simulation domain.
- Another is the tiling simulation domain that always attached to the player. We just need to map the player's position to the UV in the simulation texture in a tiling pattern

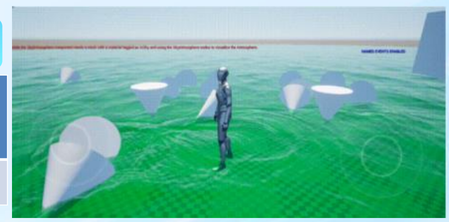
Surface Wave Simulation

Buffer Usage

- Three R16F textures for holding t-1, t-2 and current frame water height.
- Two R8 texture for dynamic object and boundary information.
- 1024x1024 texture for 80x80 meters simulation domain, tweakable.
- Rendered into screen space displacement buffer

Performance

	PS5 (1024x1024)	Snapdragon 865 (512x512)
Time(ms)	<0.1	1



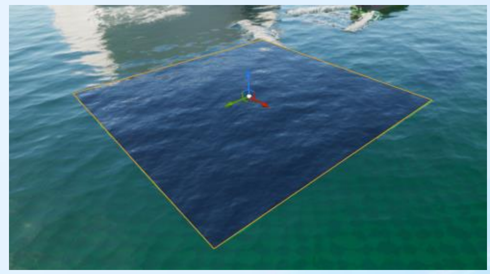
More Tessellation Methods

- Screen space tessellation
- Adaptive subdivision

Height Map Water Limitation

One layer

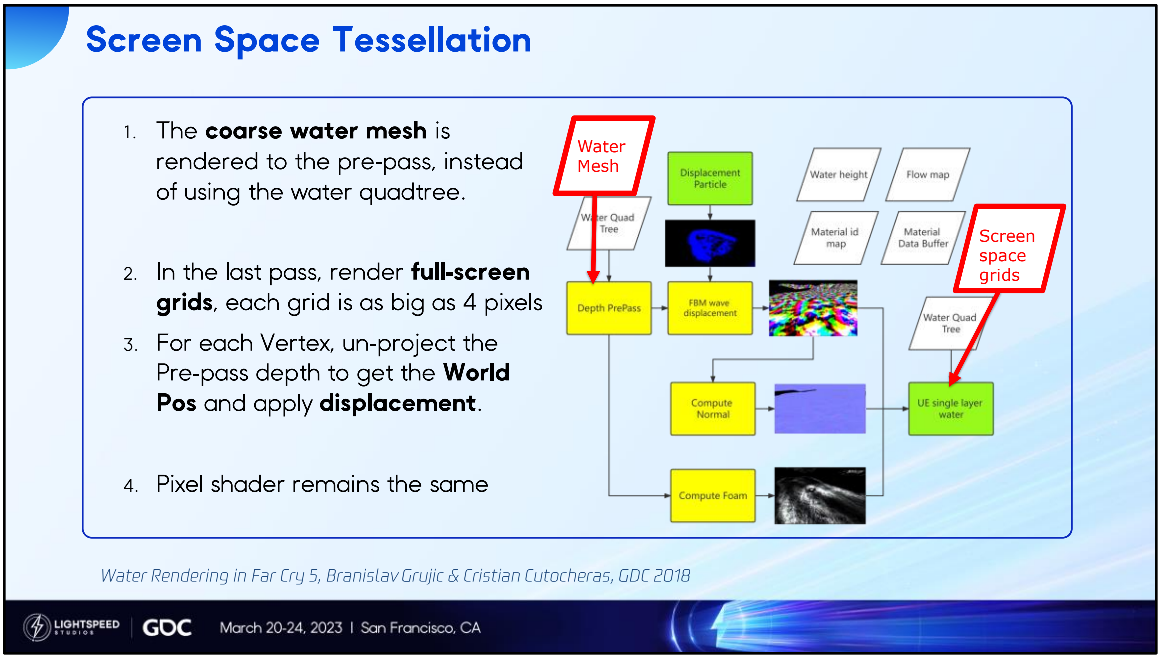
- Can’t render water over water



Solution

- Screen space tessellation for small pieces of water

Branislav Grujic & Cristian Cutocheras. “Water Rendering in Far Cry 5”, GDC 2018

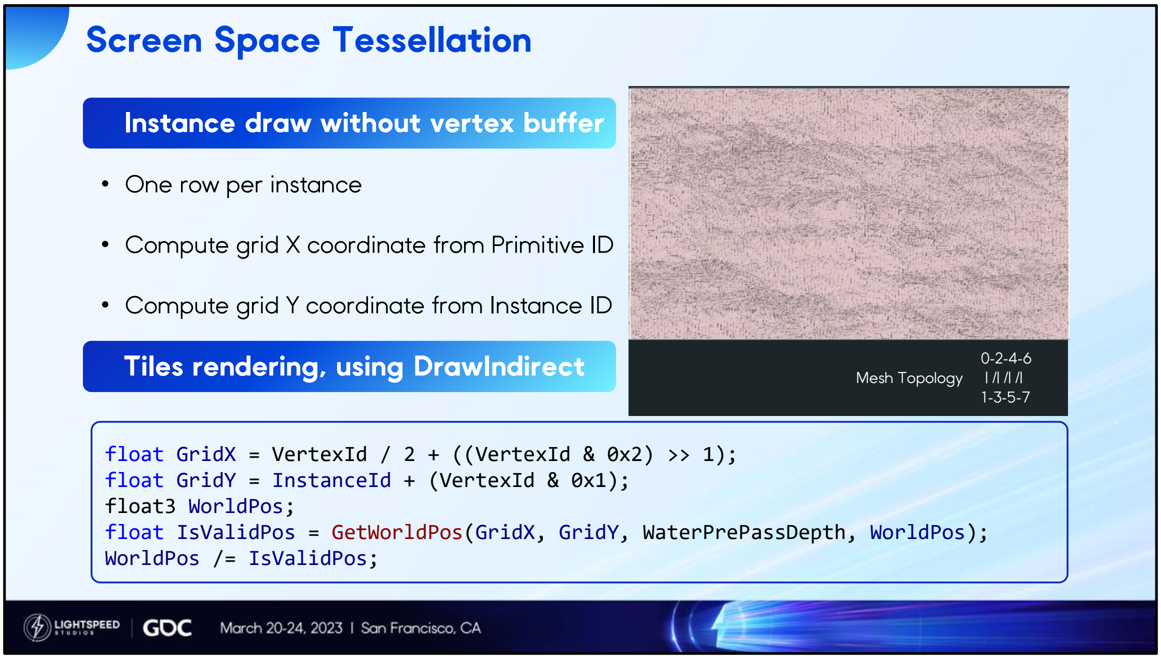


The pipeline looks similar to the water quadtree rendering in the pre-pass
Click to show new pipeline:

- Instead of using quadtree, we render the coarse water mesh into the pre-pass.
- in the lighting pass, full-screen grids are rendered instead of CDLOD quadtree mesh.
- the rest part remains the same.

The screen space grids contain many pixel-size quads. Each quad is as big as 4 pixels.
In **the vertex shader**, each vertex will sample the depth from pre-pass to get the world position and sample the screen space displacement buffer to get the final position.
The pixel shader remains the same.

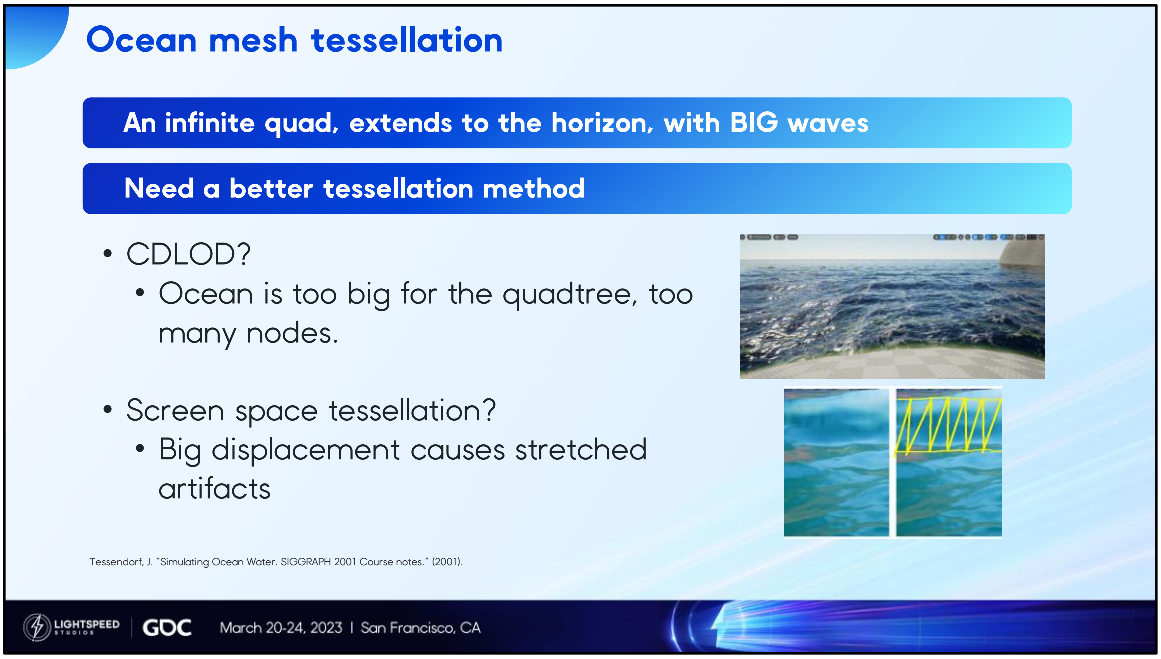
One thing to be noted. As both coarse mesh and quadtree mesh are rendered into the same pre-pass depth, we need an extra bit to mark each pixel if they are used for screen space or not, the final screen space grids will only pick up the pixel with a right mask.



We render the grids with instance draw without a vertex buffer
- Each draw contains a row of the grids
- The instance count will be the grids height.

This is the mesh view from rendering doc capture, each grid looks like a pixel.
- In VS, we use the same divide-by-zero trick to remove non-water pixels
- A tile categorization pass will be dispatched to filter out tiles containing no water.

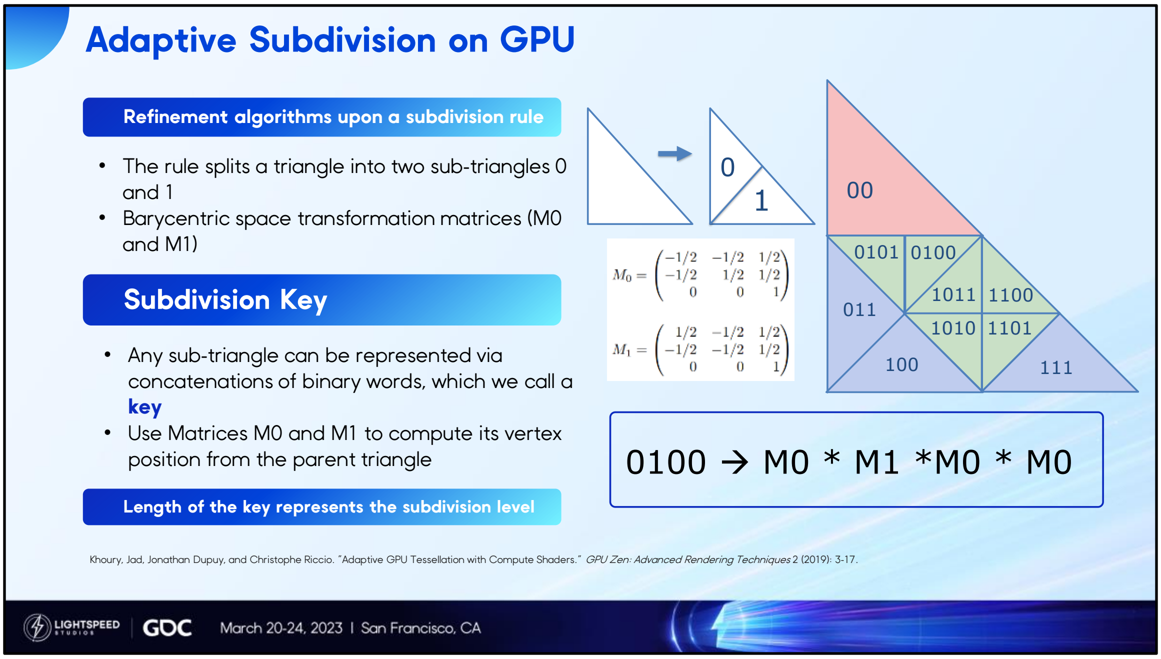
We could also use SV_PrimitveID here, but it is not supported on all platforms, especially mobile.



We use FFT waves for ocean. This tech is covered by many previous papers and game talks, we will focus on the tessellation method this time.
Ocean is basically a very big mesh that extends to the horizon with big waves.

CDLOD is not suitable for rendering a super large water body, the **quadtree nodes count** will be huge, and the **mesh density** is still high in the distance.

Screen space tessellation looks perfect when looking from the top, but when we apply the displacement from big ocean waves in screen space, and the camera is close to the ocean surface, it scrambled the vertices and connected them in the wrong order. Some pixels in the middle distance are connected with pixels in the far distance and form **stretched** triangles.



The algorithm is based on a binary triangle subdivision rule

Subdivision Key

The rule splits a triangle into two sub-triangles 0 and 1, each sub-triangle has its own barycentric space transformation matrices (**M0** and **M1)** that can be used to compute its vertex position from the parent triangle.

Any sub-triangle can be represented via concatenations of binary words, which we call a **key**.

We retrieve the subdivision matrix for each key through successive matrix multiplications with the same sequence as the binary key concatenates.

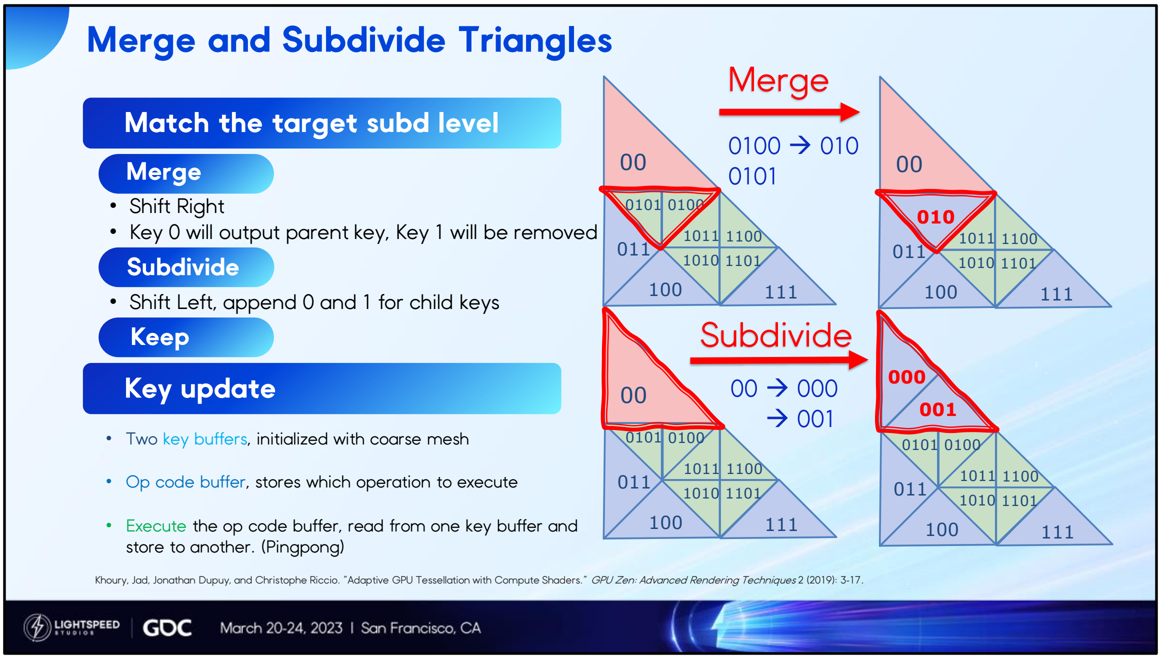
For example, the transformation matrix for key 0100 denotes as M0100, it can be concatenated as M0100 = M0 * M1 *M0 * M0.

Subdivision Level:

Natively, the length of the key represents the subdivision level that a triangle is applied. For example, key 0101 means the triangle is subdivided 4 times from the original triangle.

Performance issue?

It looks like a performance issue that a triangle with 10 subd level will do 10 matrix concatenation, but in reality, it is not that slow as the highly detailed mesh only exist in a small range close to the camera.
We have also tried pre-cache the matrix for each key, it doesn't improve the performance but only used more memory.



Implementation:
The subdivision starts from a coarse mesh which represents the entire ocean plane
For each frame, we process each key in the working buffer based on its distance to the camera to compute its target subdivision level.

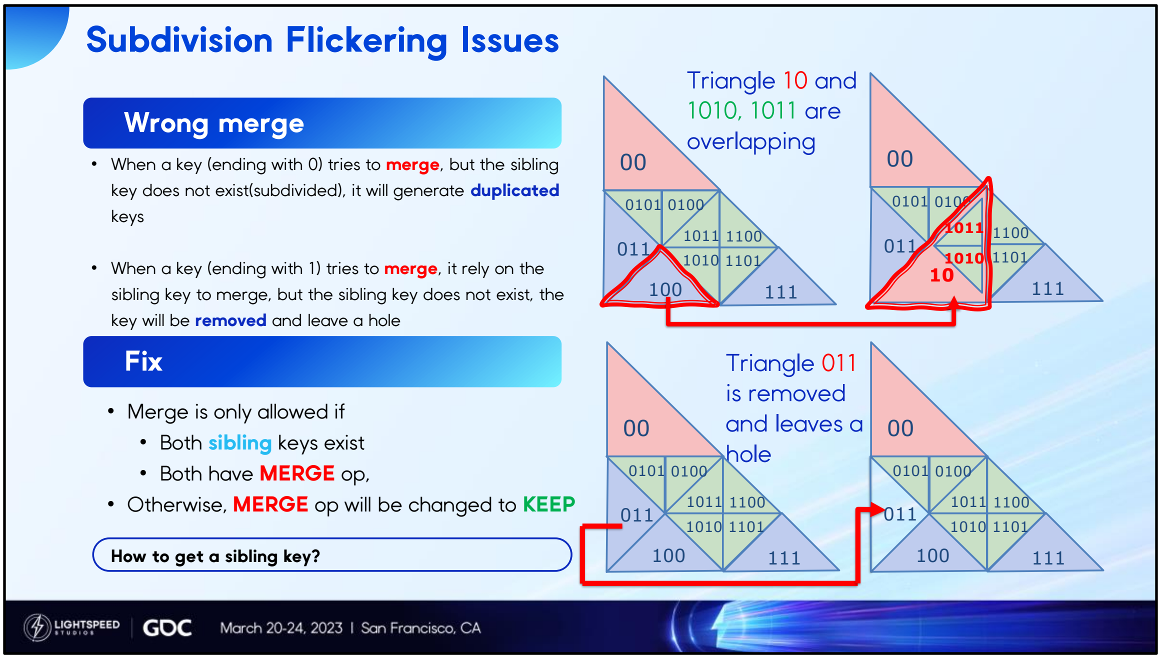
There are 3 cases:

- **Merge**
  If the target subdivision level is smaller than the current subdivision level, we will abandon this triangle but write the key to its parent triangle to the output buffer.
  The key of the parent triangle can be simply gotten by removing the right most bit.
  For example (Figure: Merge), key 0101 has parent key 010
  During the merge, only the keys ending with 0 will output its parent key, keys ending with 1 will be removed.
  So only one parent key is outputted from the 2 children after the merge.
- **Subdivide**
  If the target level is bigger then current level, we will need to subdivide

current triangle and output 2 sub-triangles to the output buffer.
It can be done by simply appending 0 and 1 on current key.
For example, key 00 can be split into 000 and 001. (Figure: Subdivide)

- **Keep**
  If target subdivision level is the same as the current level, simply write it to the output buffer.

In the implementation , the root key is 1, so the most significant bit is always 1.



We found that when the camera is moving fast, some random mesh flickering can be observed, and even worse, sometimes it leaves a permanent hole on the water mesh.

Debugging

We dump all the key buffers, sort them, check the diff, and found the reason.

Reason

In most cases, both sibling keys will get the same operation code as merge or subdivide, but there are cases they got a different operation, and then the next merge command will be an issue.

Subdivision Key Buffer Sorting

In a sorted key buffer, the siblings must be at adjacent pos

- Compute shader output is unordered. Sorting is required

Prefix sum sorting

- Our implementation uses **the in-place up/down sweep** in **LDS**
- The total is 3 passes of prefix sum and 2 passes of block sum add.
- Each pass uses a group size of 64 and it can support up to **262,144** items.
- Under **0.1ms** for subdivision and sorting.

https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

We realize that if the key is sorted, the siblings must be located beside each other.
As the keys are output from the computer shader in random order. we need a GPU sorting algorithm.
Prefix sum is a popular algorithm for parallel GPU sorting,

There was a Unity talk in 2021, they are using the same subvision algorithm on terrain, with concurrent binary tree.
We haven't tried that method, but prefix sum sorting in group share memory is very fast, it only took less than 0.1ms for both subdivision and sorting on PS5.

Subdivision Memory Issues

Running out of memory with 2x64M buffers

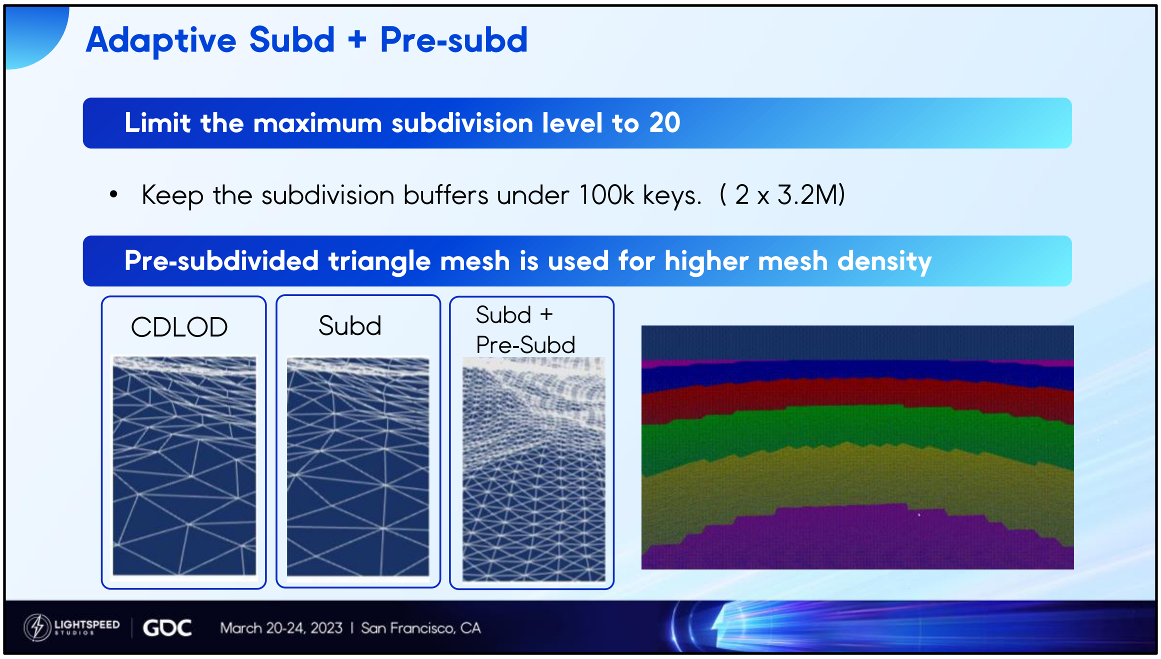
- 32 bits x 2 buffer to store the keys
- One triangle can be subdivided up to 30 times (some bits reserved)
- Ocean size 20km x 20km

Two Solutions

- Limit the max subdivision level
- Reduce the ocean mesh size

LIGHTSPEED STUDIOS | GDC March 20-24, 2023 | San Francisco, CA

After implemented the subdivision, we quickly runs into memory issues.
We use 32bits key and double buffered, allow it to subdivide up to 30 times,
The ocean will cover 20km x 20km, the mesh center will keep following the player/camera.
It can quickly run out of the buffer.



By limiting the subdivision level to 20, we can control the key buffer under 100k keys.
It can reach the same mesh density as cdlod where close to the camera, but less triangles in distance.

Pre-subdivision
For the Adaptive sub, one key represents one triangle. we can pre-subdivided this one triangle into 2 or 4 or 8 triangles, this can further increase the mesh density without increasing the key buffer memory.

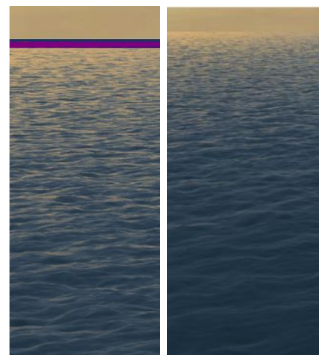
Adaptive Subd + Screen Space Tessellation

Reduce the Ocean mesh size

- How about just 10km x 10km?

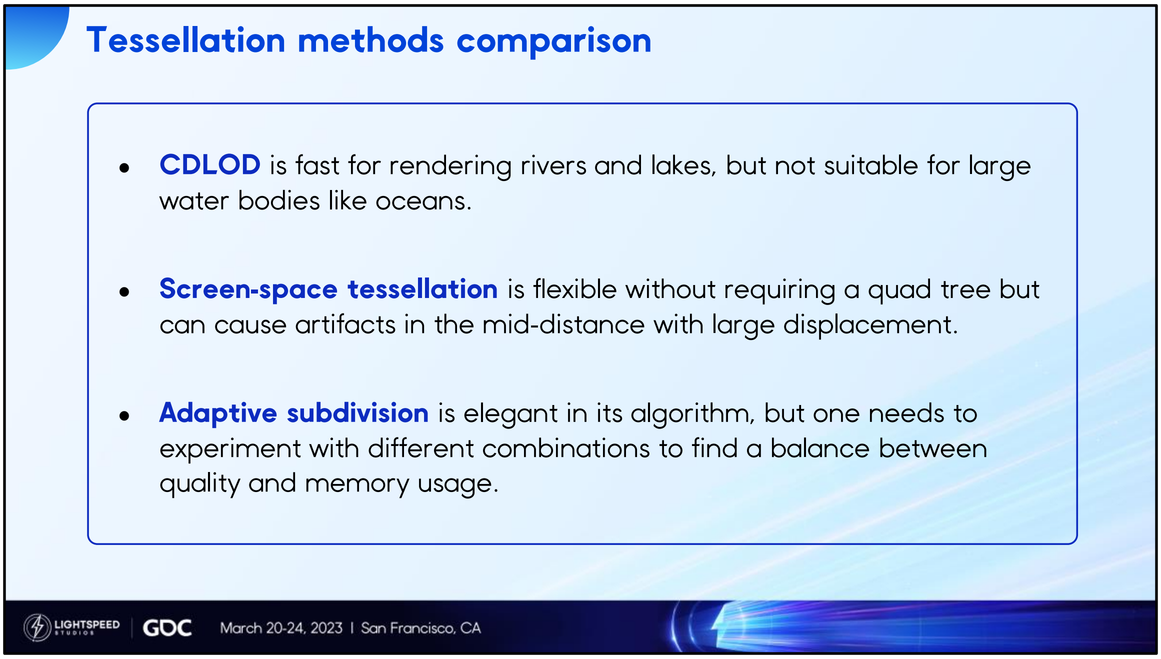
Solution

- Use screen space tessellation to fill the gap between horizon
- Render without displacement



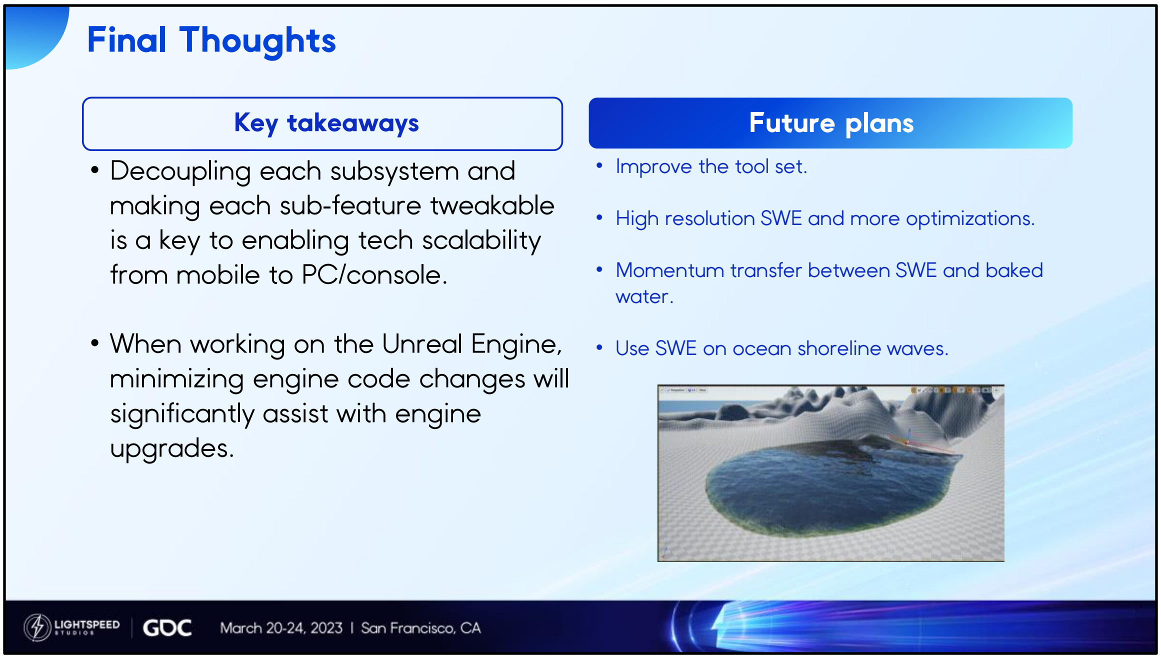
Left, Screen space mesh in debug view
Right, After screen space mesh applied

We experiment with reducing the ocean mesh size, surprisingly the ocean keeps extending to the horizon visually until the size is reduced to 10k. But there is a small gap.
That means a large area in this ocean mesh close to the horizon only contributes a few pixels on the screen.
We tried again with the screen space tessellation to **fill the gap**, and it works great.
We rendered the screen space grids to cover the entire ocean but skipped all the pixels already been rendered. With the tile categorization.
As we will fade out the displacement at the horizon anyway, only normal is applied, there is no displacement



Using adaptive subdivision in ocean rendering is a relatively easier problem to solve if compared to terrain rendering.

- No height map
- No Lod tweak for slope or mountain
- Ocean mesh can be attached to the player, so we can render a much smaller mesh than the terrain



When creating a system that can be scaled from mobile to high-end pc, break the rendering pass to multiple passes gives us more freedom to tweak the LOD. We tried our best to put all the code in the Unreal plugin. All the engine modifications are wrapped with a MACRO.

Plane

- We plan to add more tools like custom wave, painting foam or algae layers, and paint a river channel to control the SWE simulation.
- Current SWE simulation runs at the resolution of 1 square meter per pixel, any objects small than this size will be ignored, we will try to increase the resolution to get more detailed waves.
- We have tried blending between SWE and baked water where they are connected, but the result it not good. We will try to get momentum from the baked height map and flow map for more realistic effects.
- We have tested moving the water source up and down to create a shoreline wave, but still need more optimization and figuring out how to extend it to a large area.

Acknowledgment

Fengquan Wang

Yang Zhang

Yong Ding

Zhen Luo

Siyu Zhang

Roger Law

Wei Li

Reference

Branislav Grujic & Cristian Cutocheras. "Water Rendering in Far Cry 5", GDC 2018

Jeremy Moore . "Terrain Rendering in Far Cry 5", GDC 2018

Strugar, Filip. "Continuous distance-dependent level of detail for rendering heightmaps." Journal of graphics, GPU, and game tools 14.4 (2009): 57-74.

Svante Lindgren . "Continuous Distance-Dependent Level of Detail" 2020-06-13

Tessendorf, Jerry. "Simulating ocean water." Simulating nature: realistic and interactive techniques. SIGGRAPH 1.2 (2001): 5.

Khoury, Jad, Jonathan Dupuy, and Christophe Riccio. "Adaptive GPU Tessellation with Compute Shaders."

Macklin, Miles, and Matthias Müller. "Position based fluids." ACM Transactions on Graphics (TOG) 32.4 (2013): 1-12.

Wu, Kui, et al. "Fast fluid simulations with sparse volumes on the GPU." Computer Graphics Forum. Vol. 37. No. 2. 2018.

Yuksel, Cem, Donald H. House, and John Keyser. "Wave particles." ACM Transactions on Graphics (TOG) 26.3 (2007): 99-es.

Chentanez, Nuttapong, and Matthias Müller. "Real-time Simulation of Large Bodies of Water with Small Scale Details." Symposium on Computer Animation. 2010.

Kass, Michael, and Gavin Miller. "Rapid, stable fluid dynamics for computer graphics." Proceedings of the 17th annual conference on Computer graphics and interactive techniques. 1990.

Zhou, Jian Guo. "Lattice Boltzmann methods for shallow water flows". Vol. 4. Berlin: Springer, 2004.

LIGHTSPEED STUDIOS | GDC

THANKS

March 20-24, 2023 | San Francisco, CA

Website: https://www.lightspeed-studios.com/

Facebook： LightSpeedStudiosGames

Twitter： LIGHTSPEED STUDIOS

Youtube： LIGHTSPEED STUDIOS

Welcome to stop by our booth **S1069** if you would like to learn more about LIGHTSPEED STUDIOS!

WE'RE HIRING!