GPU Computing for Games

Cem Cebenoyan

Overview

- GPU Computing in games case studies
- Just Cause 2
  - CUDA C Bokeh
  - CUDA C Water
- Metro 2033
  - DirectCompute Depth of Field
- JX3 Online
  - CUDA C Animation

GPU Computing for Games

- What is *GPU Computing for Games?*

- Using a general purpose language to enable and accelerate game algorithms
  - Languages like CUDA C, DirectCompute, OpenCL
  - Algorithms like post processing, animation, simulation, and much more

- Enables new classes of algorithms, and easier access to massive parallel horsepower of GPUs

- This presentation focuses on visual effects

Just Cause 2 - Background

- Dev: Avalanche, Stockholm
- Pub: Square Enix
- 3rd person action shooter; huge sandbox world

Just Cause 2 – Original

Just Cause 2 – With Bokeh

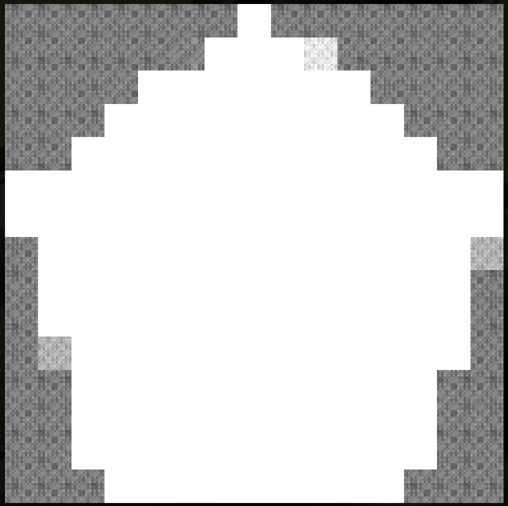
Why Bokeh?

- Provide artistic, filmic quality to depth of field
- Movie examples:

- Convolving with 8-bit, LDR scene doesn’t work
- Needs small, sharp, high-contrast points

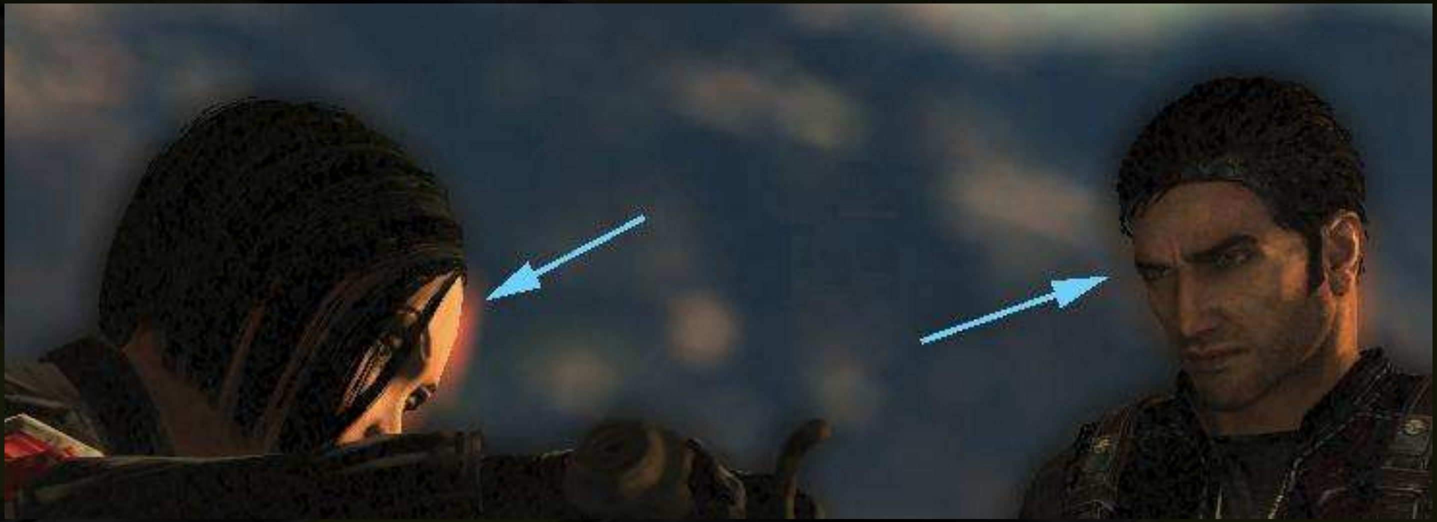
CUDA C Bokeh Blur

- Replace existing, usual PS blur
- No other changes to Depth of Field
- Brute-force, image-space convolution kernel
- First downscale scene 2x2 for perf
- 15x15 kernel gives good shape definition:
  - Hence 30x30 at frame-buffer res



Issues: Blur Leakage

- Blur leakage
- Exists in original – less obvious
- Large kernel width with bokeh – more obvious



- Fix: cross bilateral using focus amount
  - Ignore samples with distinctly different focal values
  - Requires focal value – pack into alpha channel

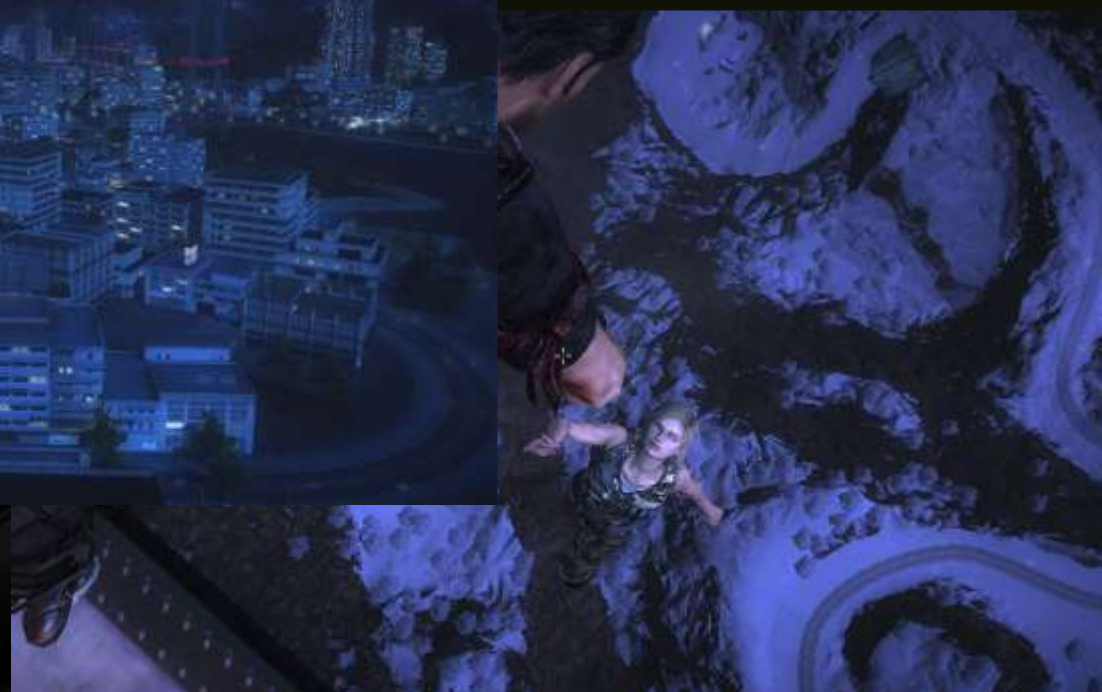
Cross Bilateral Results

Highlight Exaggeration

- Typical LDR problem
- Need to extract more contrast from R8G8B8
- Used Photoshop Lens Blur as reference

Highlight Discrimination

- Apparently bright images similar to dark ones
- Typical LDR problem
- Histograms similar



Incorrect Highlights

- **Huge highlights wrong places**
- **Snow - big problem**

Incorrect Highlights

- **Another example – cut scene**

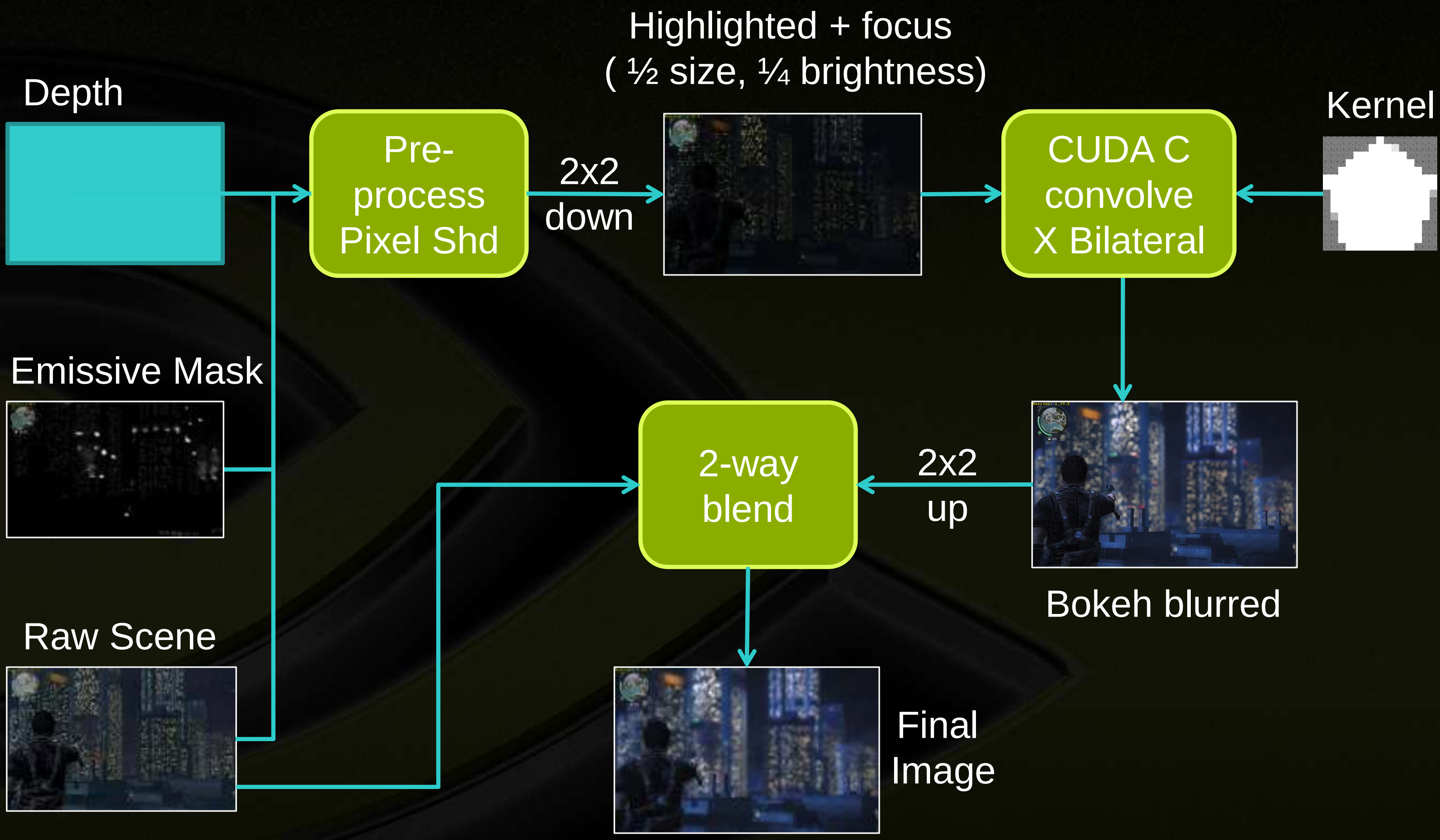
Emissive Masking

- Indicate emissive pixels in scene alpha
- Apply highlight exaggeration to emissive only
- Much more control
- Dual-source blending required

Emissive Masking – Bokeh Input

Emissive Masking – Bokeh Output

Bokeh Pipeline Summary

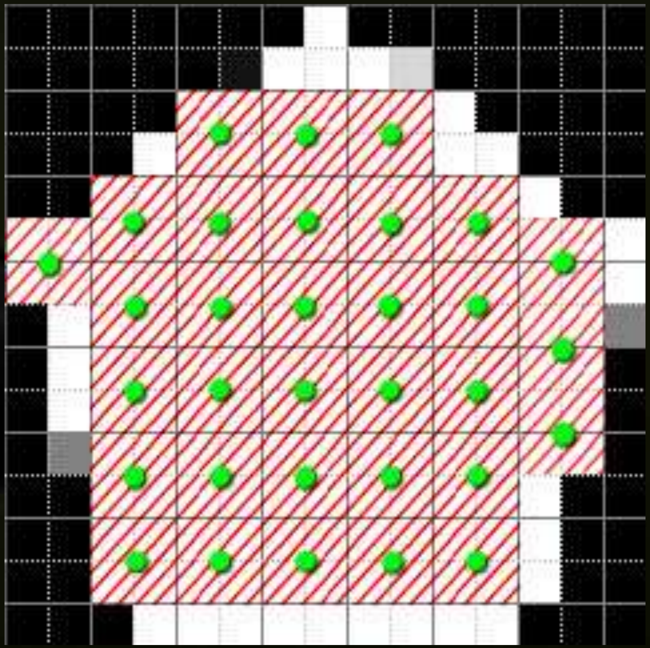


Bokeh CUDA Performance

- 15x15 kernel = 225 samples per pixel
- Early, simple versions:
  - ~ mad per input sample
  - Texture sampling of input
- Cross-bilateral:
  - $\exp(k * (f_i - f_o)^2)$ per input sample
  - Less texture bottleneck

Bokeh Optimizations

- Generate CUDA C code off line:
  - Unroll kernel loop
  - Skip kernel samples with zero weight
- Skip 100% in-focus output pixels
- Reduce kernel radius as focus increases
- Use linear sampling



Final Bokeh Perf

- **Scene-specific optimizations:**
  - **Function of how much in-focus**
  - **Cost highly variable – CUDA kernel times on GT200:**

Normal

Aim

- **Add ~2ms for D3D interop & context switches**

Just Cause 2 - Bokeh Video

gameworks.nvidia.com

Just Cause 2 - CUDA water

- **Game already contained large areas of open water (seas, harbors and estuaries)**

CUDA Water Overview

- Based on Jerry Tessendorf's paper "Simulating Ocean Water"
  - Statistic based, not physics based
  - Generate wave distribution in frequency domain, then perform inverse FFT
  - Widely used in movie CGIs since 90s, and in games since 2000s
- In movie CG: the size of height map is large
  - 2048x2048 is typical
- In games: the size of height map is small
  - Often 32x32 or 64x64 at most
  - Cost of CPU simulation is high

Performance Issues

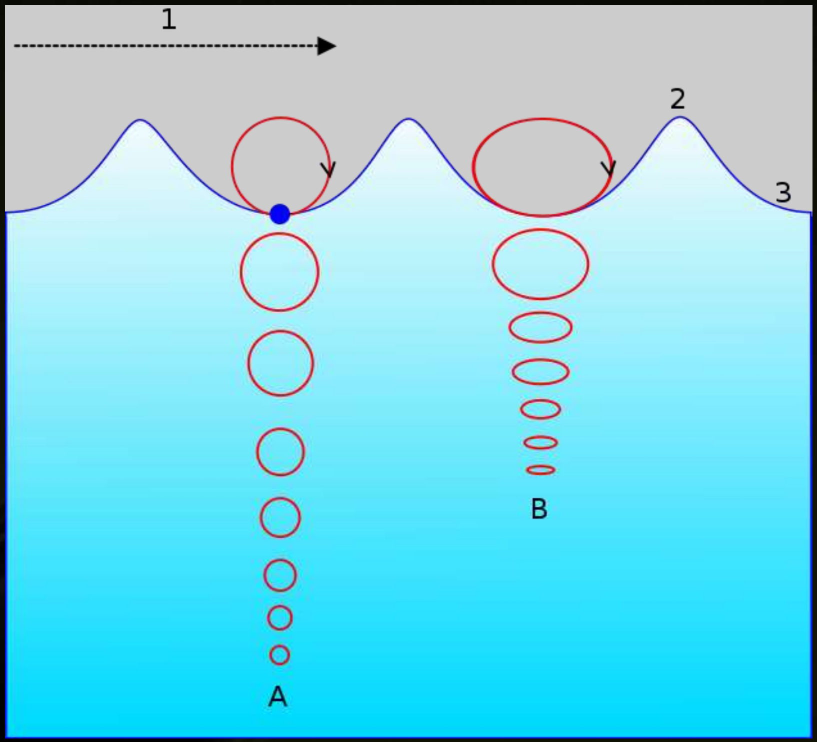
- Required to generate a displacement map in real-time
- Large displacement map gives better looking water
  - High cost on CPU FFT
  - Takes long time on CPU-GPU data transfer
- Perform FFT with GPU computing
  - Multiple 512x512 transform can be performed in trivial time
  - 1024x1024 transforms are affordable on high-end GPUs

The Algorithm: Wave Composition

- **Assumption: the ocean surface is composed by enormous simple waves**
- **Each simple wave is a hybrid sine wave, called Gerstner wave**
  - **A mass point on the surface is doing vertical circular motion**

$$\mathbf{x} = \mathbf{x}_0 - (\mathbf{k}/k)A\sin(\mathbf{k}\cdot\mathbf{x} - \omega t)$$

$$z = A\cos(\mathbf{k}\cdot\mathbf{x} - \omega t)$$



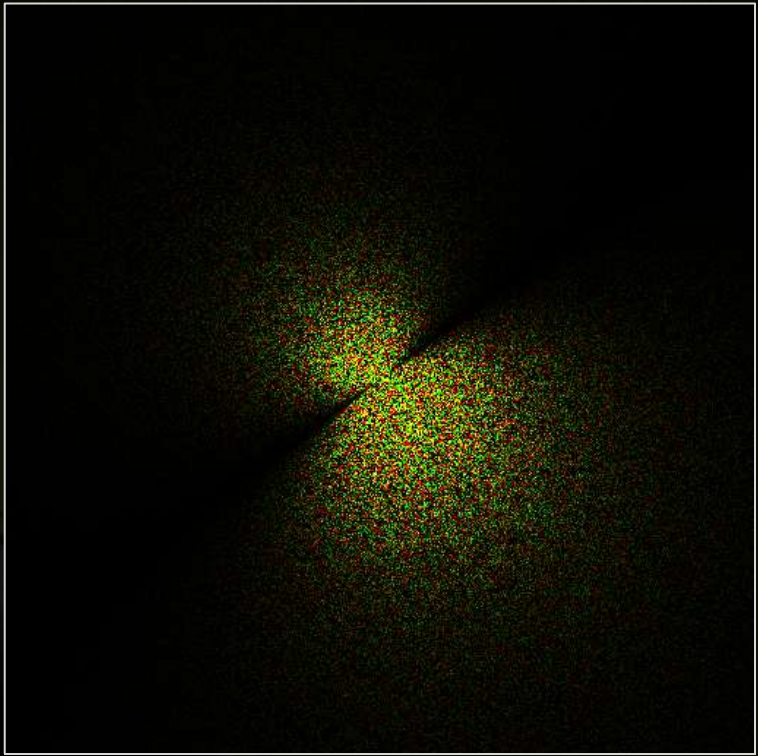
The Algorithm: Statistic Model

- **Distribution of wave length, speed and amplitude are following a statistic models**
  - **Phillips spectrum model:**

$$P_h(\mathbf{k}) = \frac{A}{k^4} \left|\mathbf{k} \cdot \mathbf{w}\right|^2 e^{-\frac{1}{k^2 L^2}}$$

- **Generated in frequency domain at the initial time**

$$\tilde{H}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} \tilde{\xi}(\mathbf{k}) \sqrt{P_h(\mathbf{k})}$$

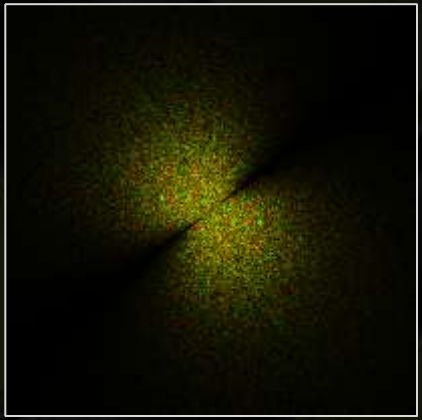


The Algorithm: Runtime

- Update three spectrums for XYZ directions per frame

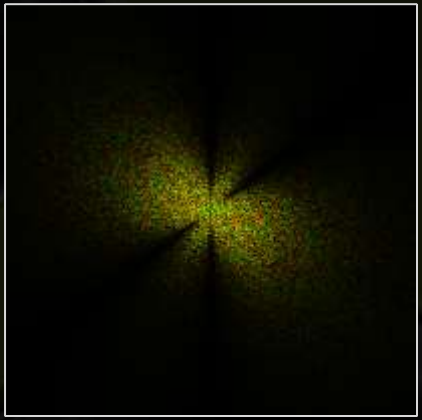
Z (height field)

$$\tilde{H}(\mathbf{k},t) = \tilde{H}_0(\mathbf{k})e^{i\omega t} + \tilde{H}_0^*(-\mathbf{k})e^{-i\omega t}$$



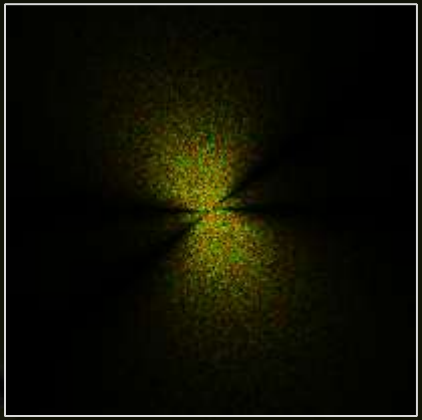
X (choppy field)

$$\tilde{\mathbf{D}}_x(\mathbf{k},t) = i\frac{\mathbf{k}.x}{k}\tilde{H}(\mathbf{k},t)$$



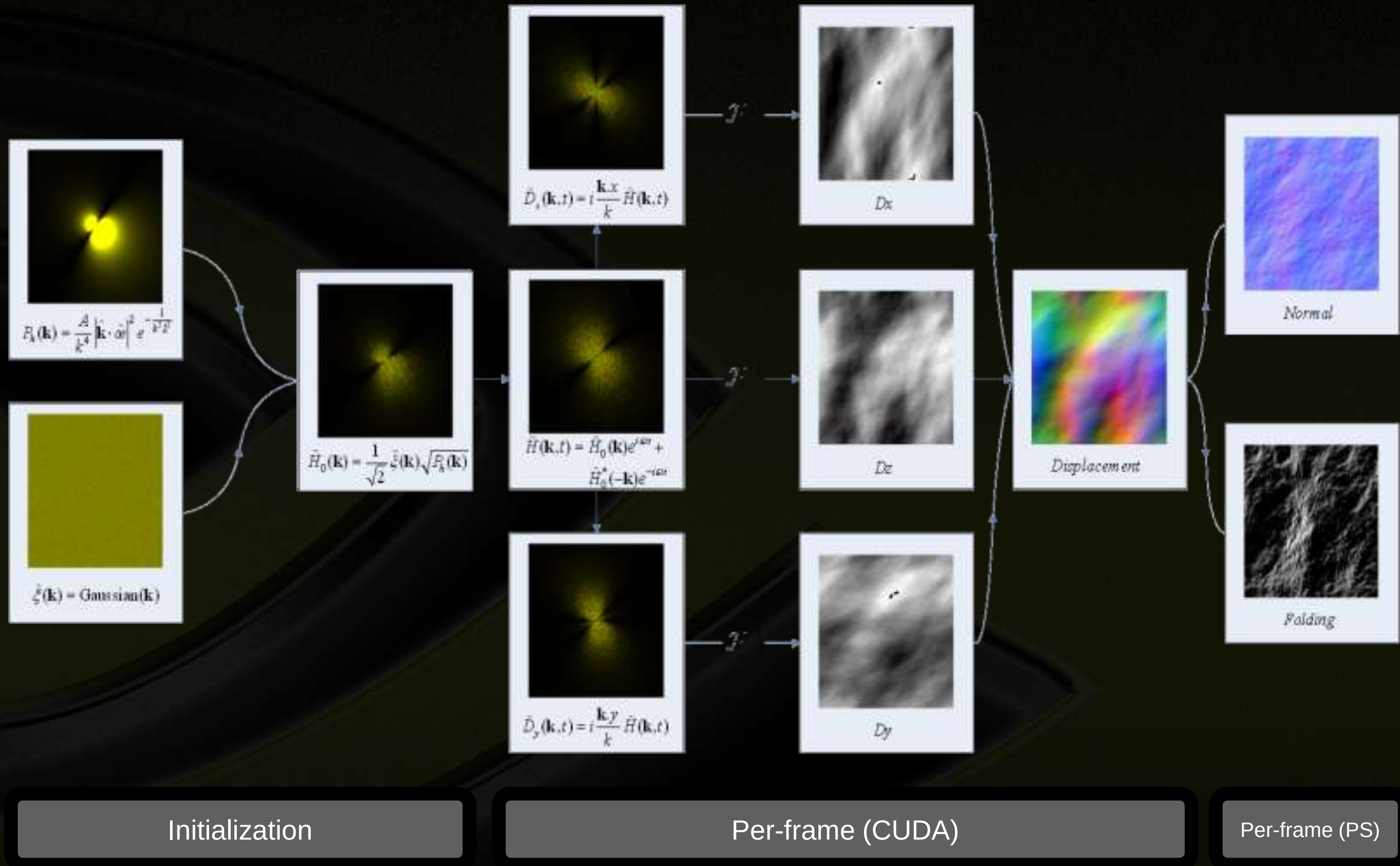
Y (choppy field)

$$\tilde{\mathbf{D}}_y(\mathbf{k},t) = i\frac{\mathbf{k}.y}{k}\tilde{H}(\mathbf{k},t)$$



- Perform inverse FFT on three spectrums
- Surface normal and other data are generated from displacement map

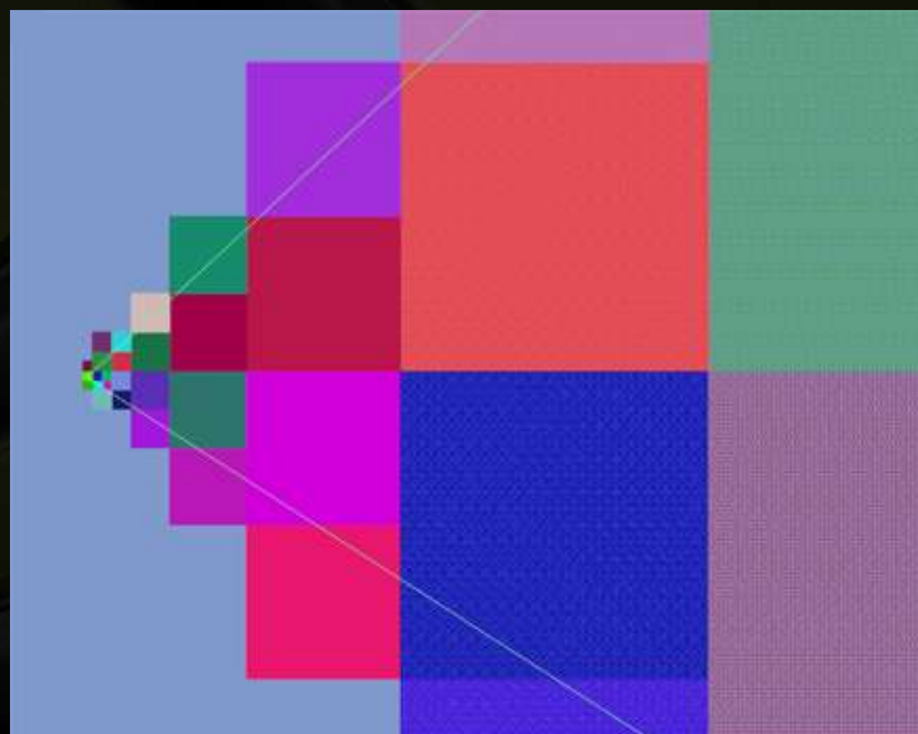
The Algorithm: The Full Simulation Chart



Rendering

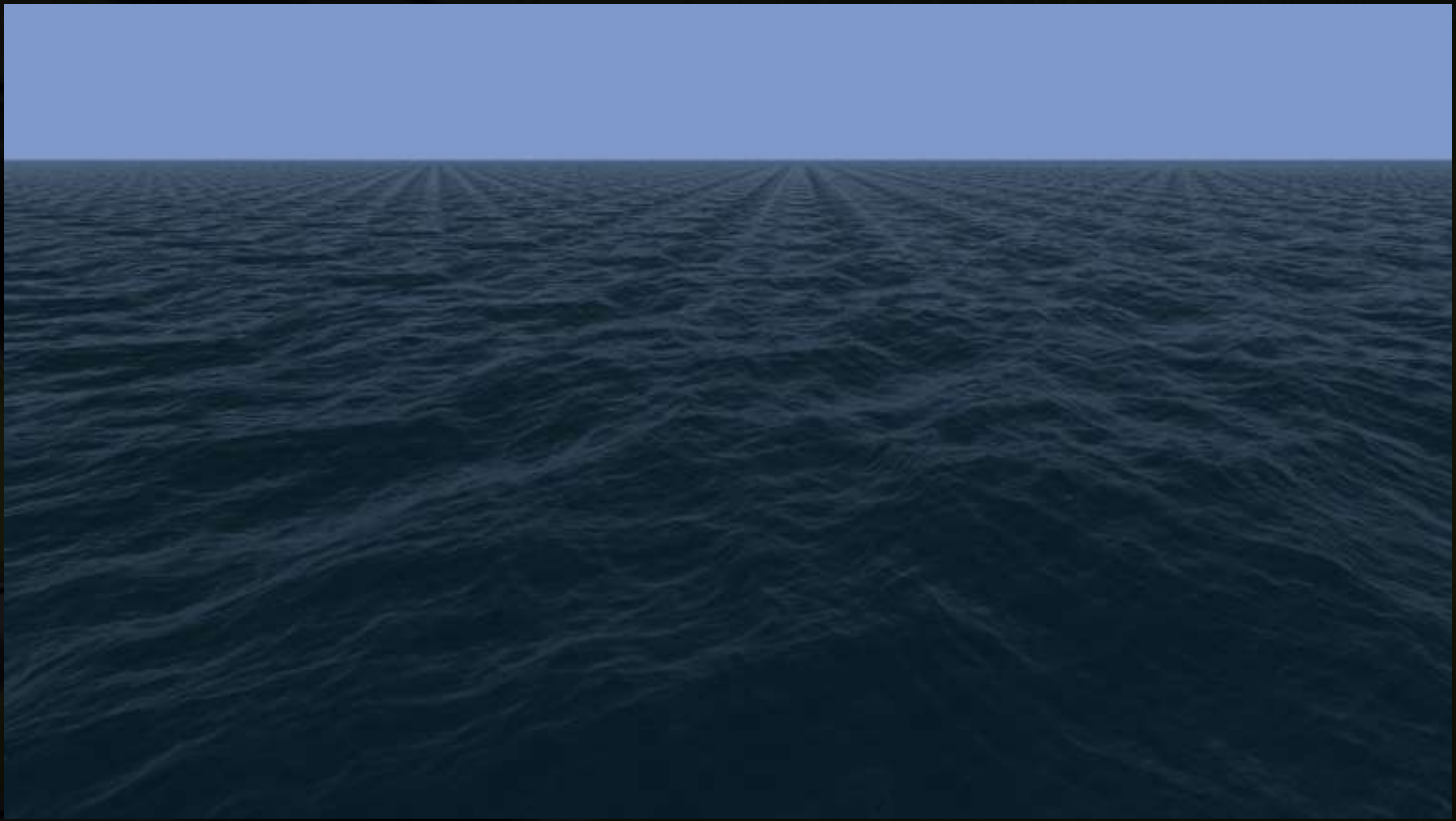
World Space Rendering

- We use world space rendering
- The mesh is created at half resolution of the displacement map
- Use quad-tree for frustum culling and mesh LOD



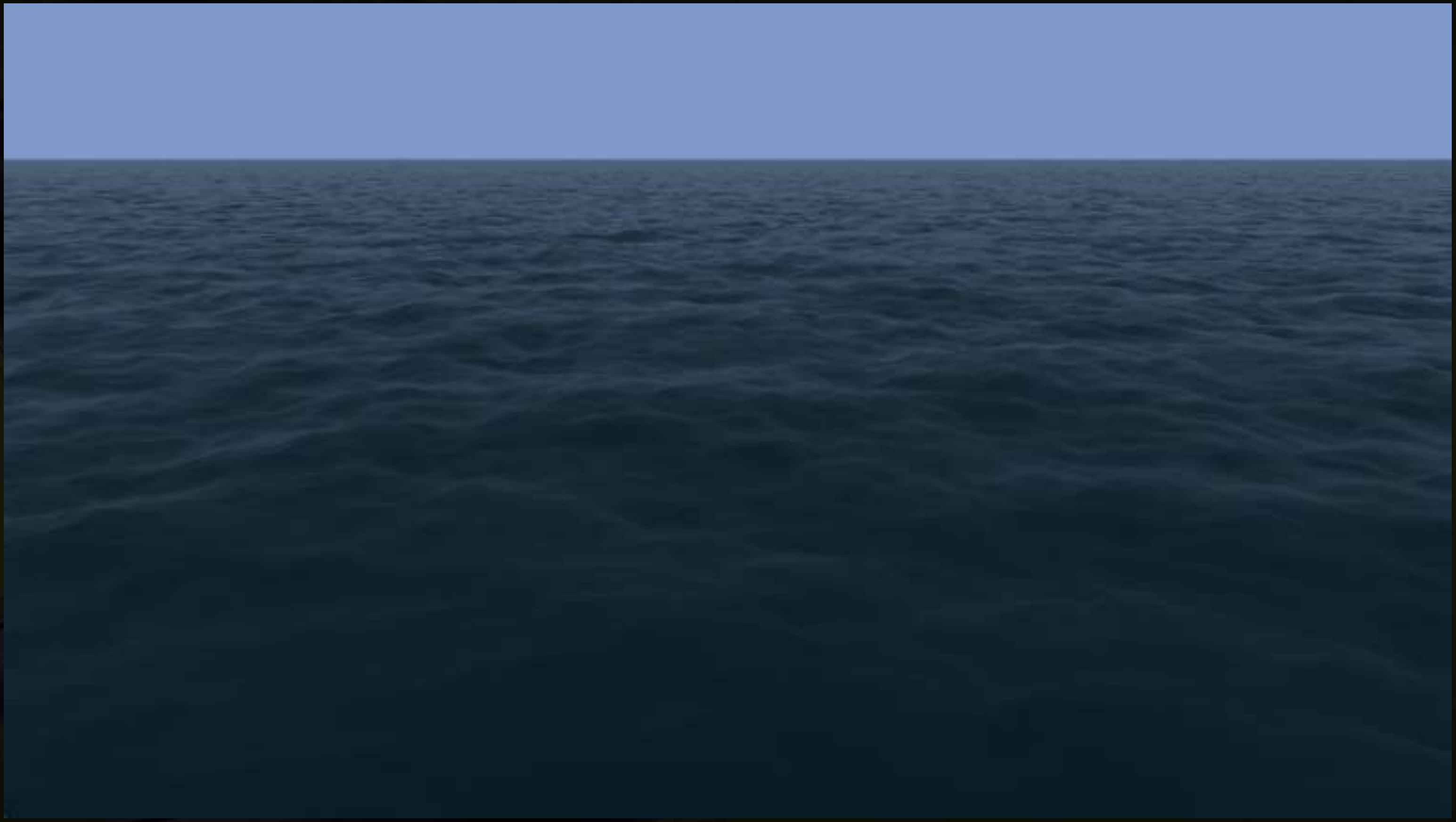
Tiling Artifact Removing (1)

- FFT produces a periodic pattern
  - Repeated pattern becomes distracting at distance
  - But looks okay close to the camera



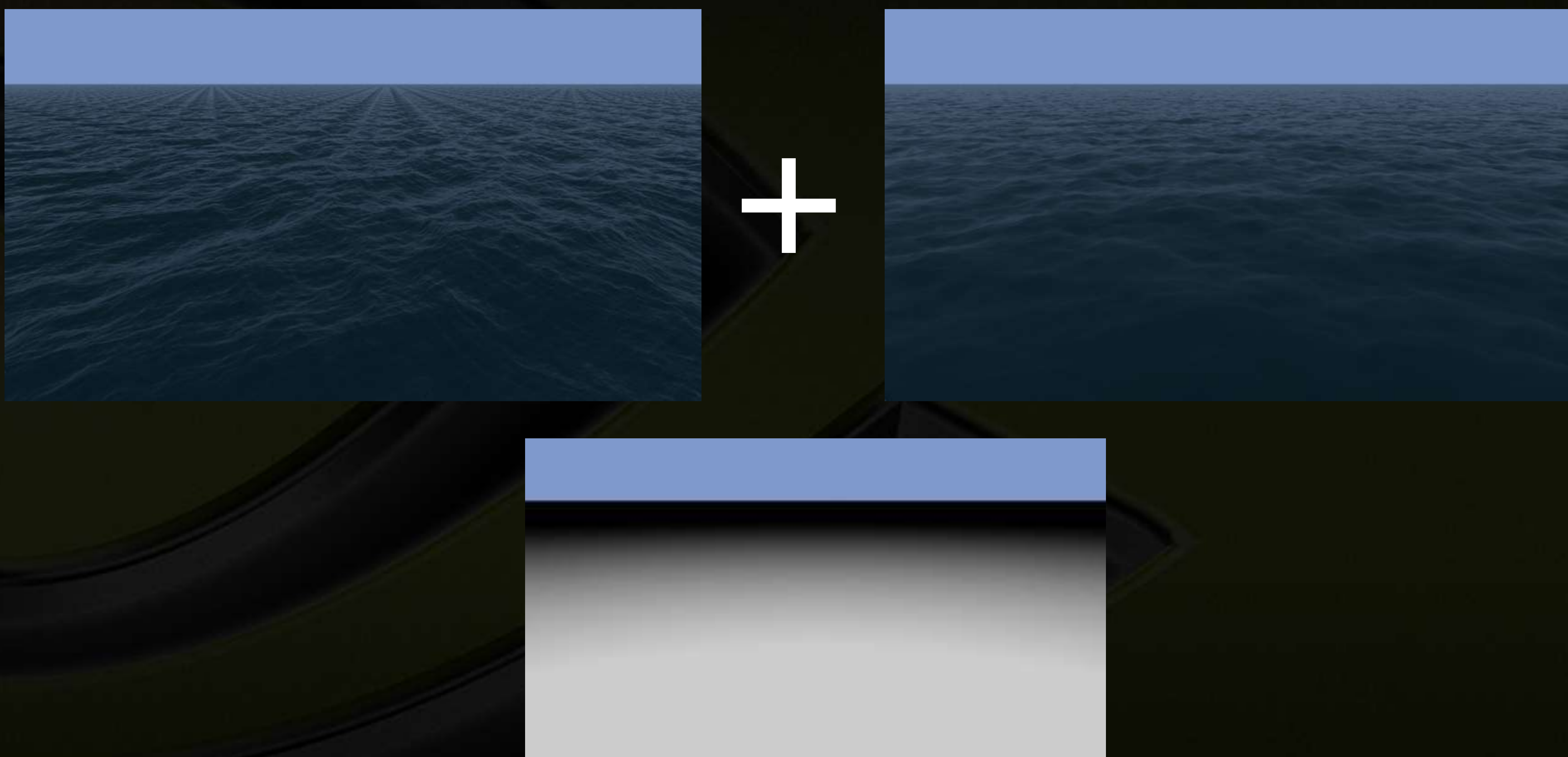
Tiling Artifact Removing (2)

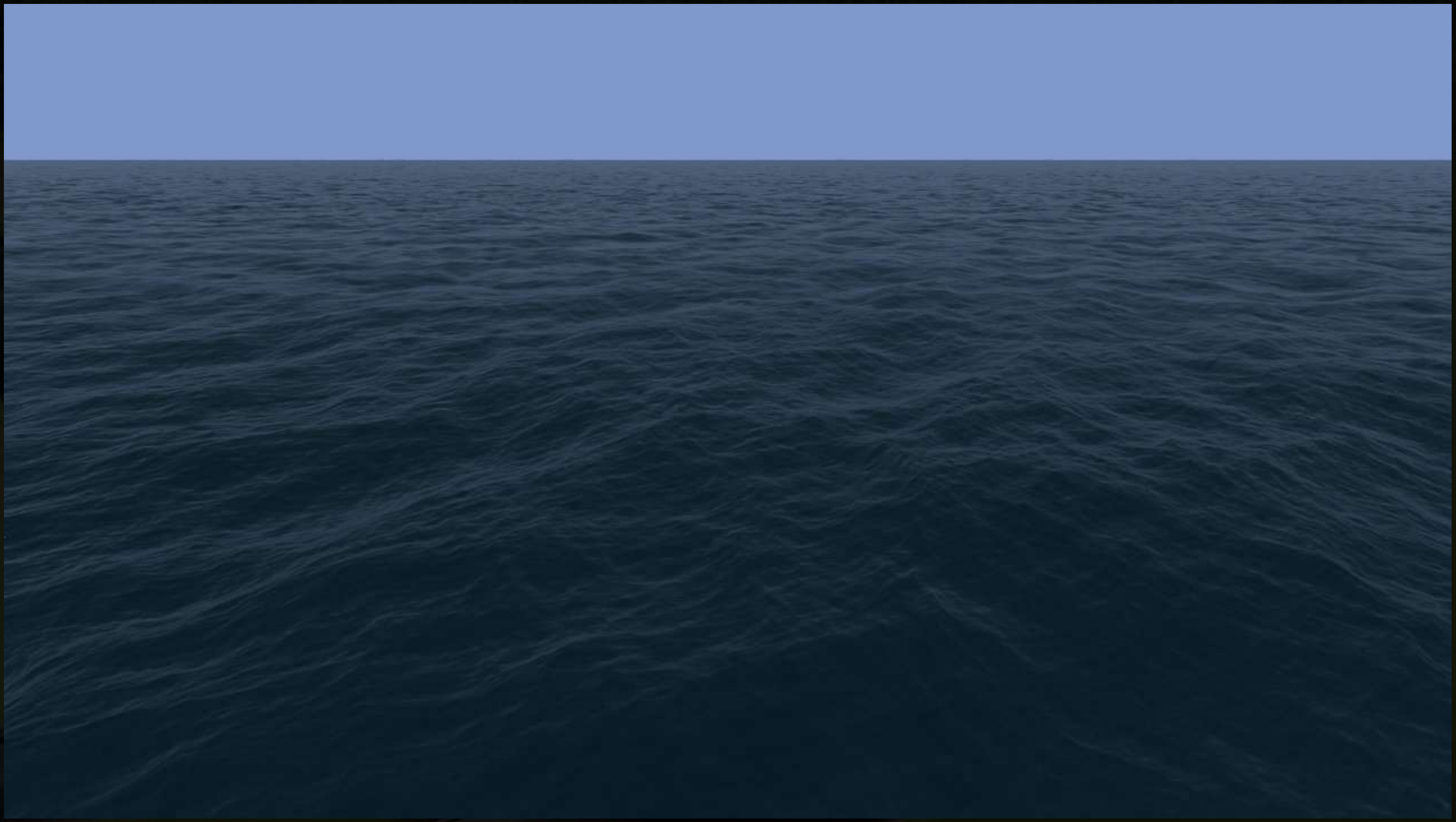
- **Perlin noise yields no tiling artifact**
  - **But lack of details close to camera**



Tiling Artifact Removing (3)

- **Solution: blend Perlin and FFT generated crests**





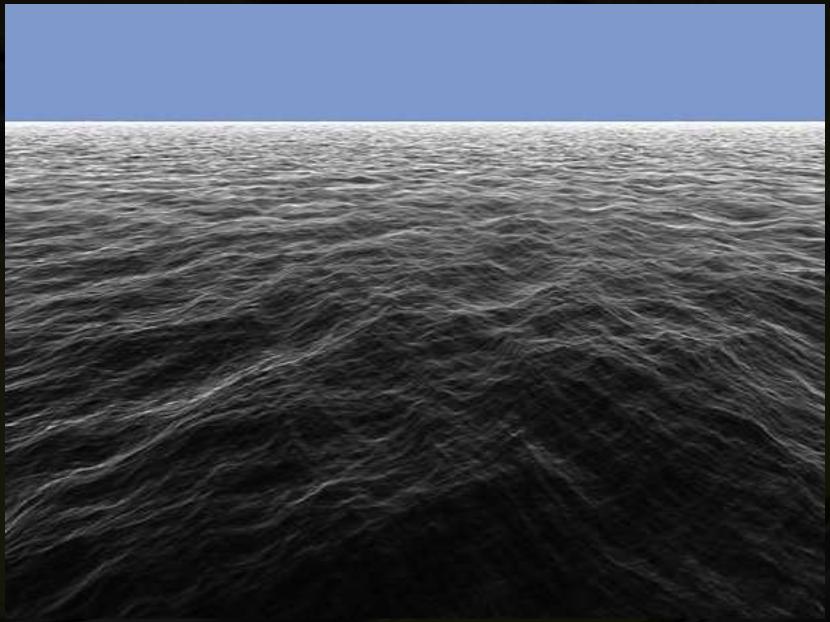
The result of blending FFT and Perlin noise
(simple rendering mode)

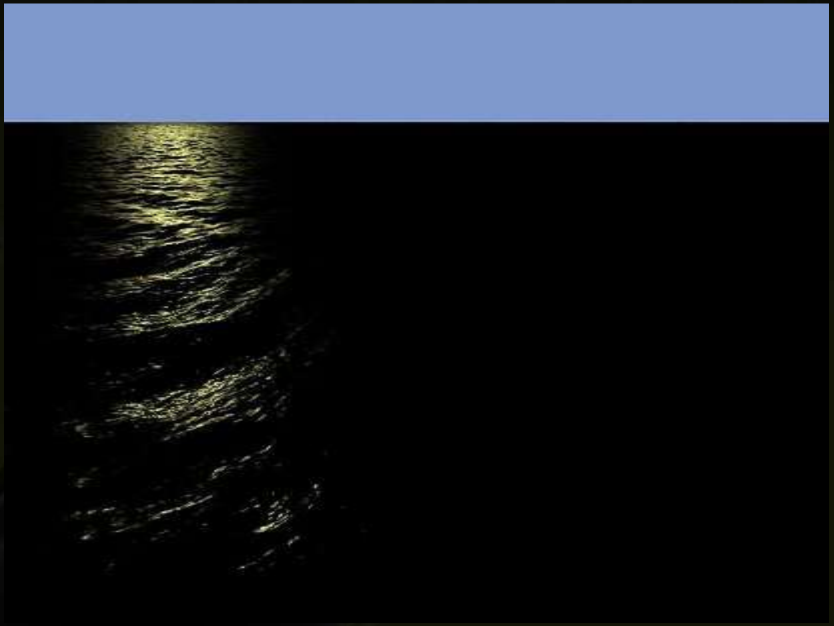
Ocean Shading (1)

- The demo only rendered for deep ocean water
  - Shallow water rendering is much more complicated
- Shading components
  - Water body color: using a constant color
  - Fresnel term for reflection: read from a pre-computed texture
  - Reflected color: using a small cubemap blend with a constant sky color
  - Vertical streak: computed from a modified specular term

Ocean Shading (2)

- **Fresnel term (left) and sun streak (right)**





CUDA C water – before & after

CUDA C Water – Video

References

- "Motivating Depth of Field using bokeh in games" http://beautifulpixels.blogspot.com/2008/11/motivating-depth-of-field-using-bokeh.html
- Joint Bilateral Upsampling, Kopf et al, SIGGRAPH 2007, http://johanneskopf.de/publications/jbu/index.html
- "Simulating Ocean Water", Tessendorf http://tessendorf.org/papers_files/coursenotes2004.pdf

Metro 2033: the game

- A combination of horror, survival, RPG and shooting
- Based on a novel by Dmitry Glukhovsky

Technology

- Developed by Oles Shishkovtsov
  - Lead architect of the STALKER engine
- Metro engine is based on new tech
- Packs a lot of innovation
  - Pervasive DX11 tessellation
  - Advanced post processing using DirectCompute

Depth of field

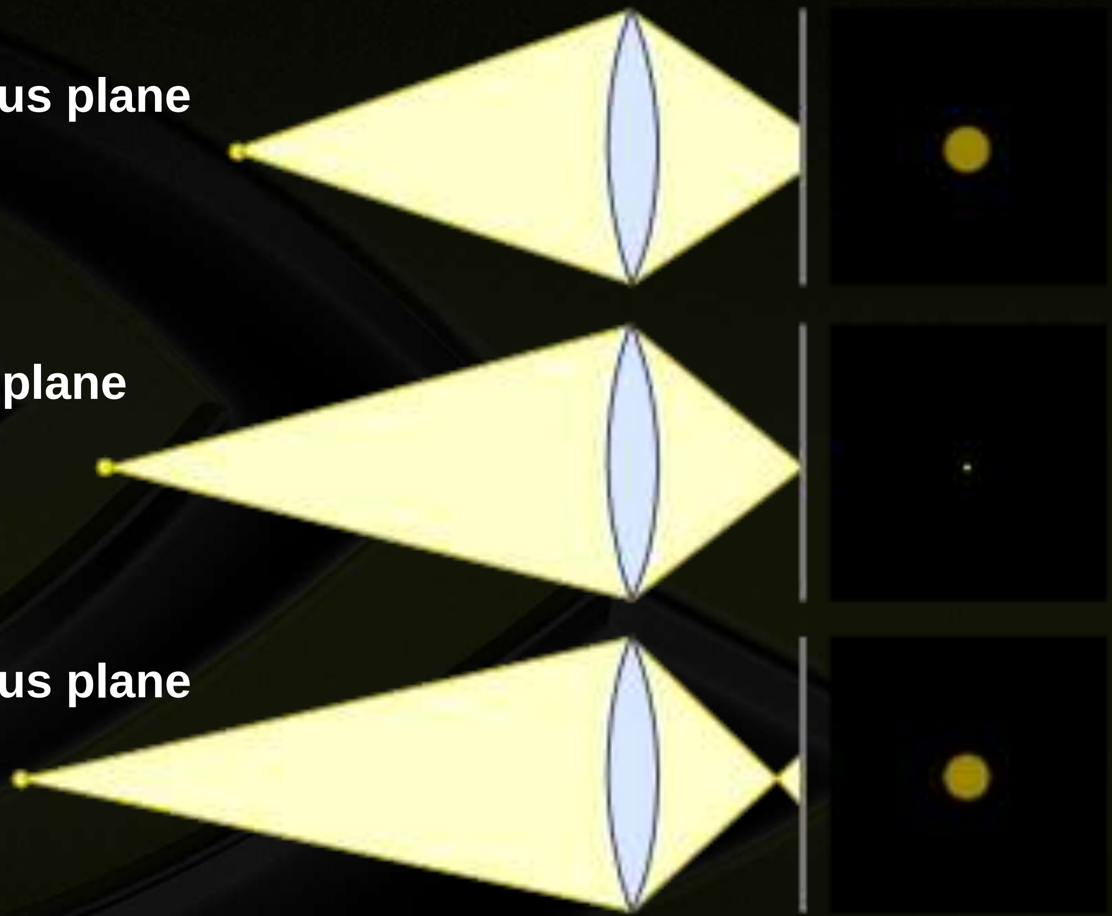
- Common effect in games these days
- Typically post-processing image from a pin-hole camera
- Wanted a more realistic, gritty look
  - Less filimic, so JC2-style Bokeh would not work as well
- Key challenge: Need to keep sharp in-focus objects and blurry backgrounds from bleeding into each other

Circle of Confusion (CoC)

Point within focus plane

Point on focus plane

Point beyond focus plane



Disk

Point

Disk

Depth of field effect

- **Post-processing input color layer by using depth layer to calculate CoC (circle of confusion)**

Bleeding artifacts

From *Metro 2033*, © THQ and 4A Games

Bleeding artifacts

From *Metro 2033*, © THQ and 4A Games

Diffusion DOF in Metro

From *Metro 2033*, © THQ and 4A Games

Diffusion DOF in Metro

From *Metro 2033*, © THQ and 4A Games

Diffusion DOF in Metro

From *Metro 2033*, © THQ and 4A Games

Diffusion-based DoF

- Introduced by Pixar Animation Studio back in 2006
  - See *Interactive DOF using Simulated Diffusion on a GPU*, Kass et al.
- Basic idea: DOF and heat diffusion analogy
  - Pixel color = Temperature sample
  - CoC = Thermal conductivity
  - Convert CoC into conductivity, and allow colors bleed like heat diffusion in a non-uniform media
- Challenges:
  - Blur kernel size varies across screen
  - Very large kernel size at distance

Benefits

- **No color bleeding**

From *Metro 2033*, © THQ and 4A Games

Benefits – detail view

From *Metro 2033*, © THQ and 4A Games

Benefits

- **Clear separation of sharp in-focus and blurred out-of-focus objects**

From *Metro 2033*, © THQ and 4A Games

Implementation

- **We cast DOF problem in terms of basic heat diffuse equation**

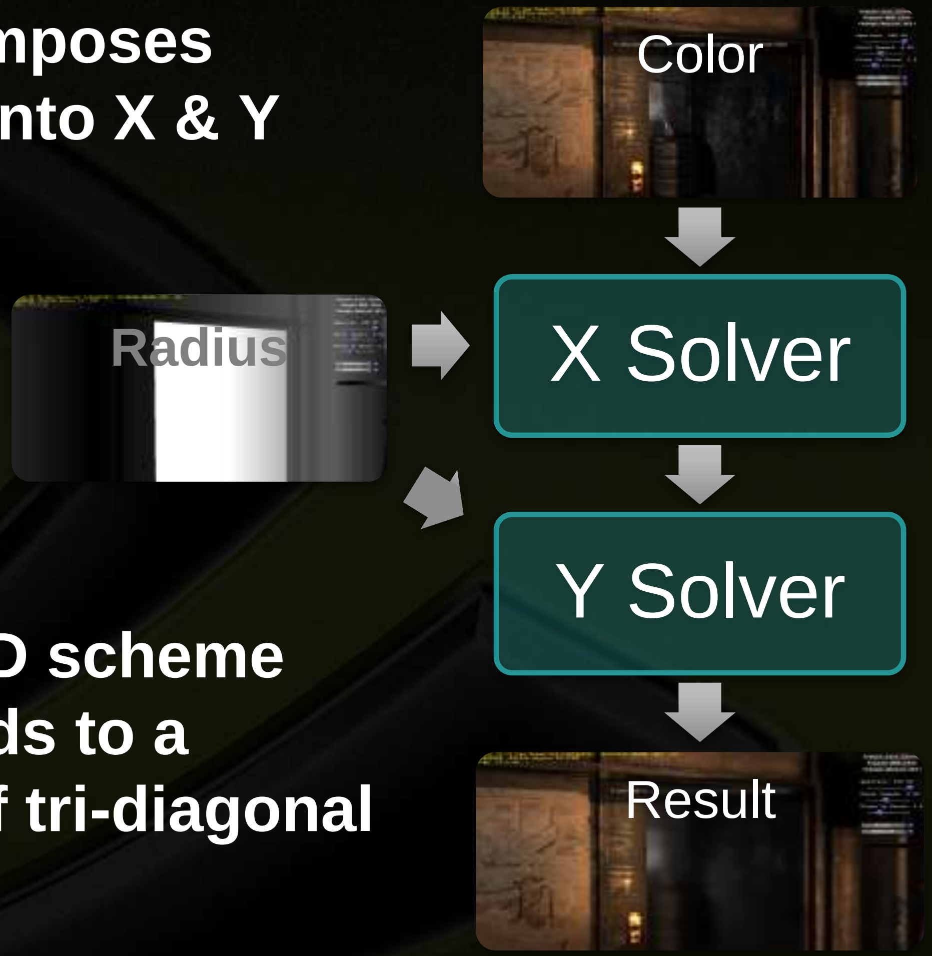
$$\frac{\partial u(x,y)}{\partial t} = \nabla \cdot \left(\beta(x,y)\nabla u(x,y)\right)$$

$u(x,y)$	Image color (temperature sample)
$\beta(x,y)$	Circle of confusion (heat conductivity)

- **Using Alternate Direction Implicit (ADI) numerical method**

Implementation

- ADI decomposes equation into X & Y directions
- Applies FD scheme which leads to a number of tri-diagonal systems



Solving tridiagonal systems

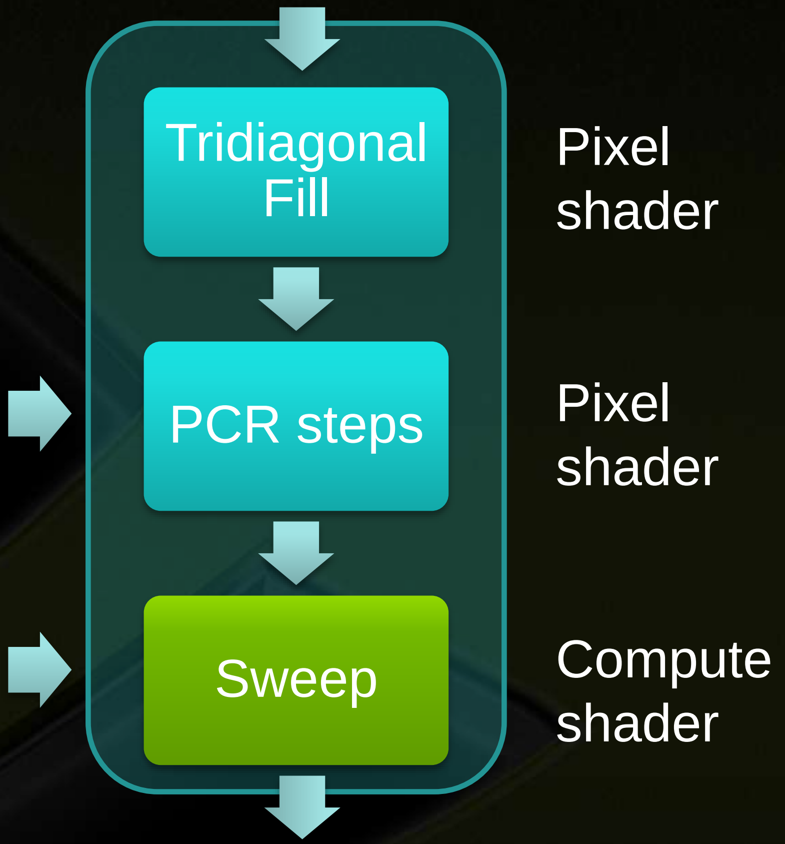
- A number of methods exist:
  - Cyclic reduction (CR)
  - Parallel cyclic reduction (PCR)
  - Simplified Gauss elimination (Sweep)
  - (see references for details)

- We use a new hybrid approach
  - PCR + Sweep

Tridiagonal solver in DX11

PCR steps = 3

Num systems	System size
Height	Width
Height*8	Width/8



Metro 2033 Depth of Field Video

References

- "Interactive depth of field using simulated diffusion on a GPU" Michael Kass, Aaron Lefohn, John Owens, Pixar Animation studios, Pixar technical memo #06-01
- "Tridiagonal solvers on the GPU and applications to fluid simulation" Nikolai Sakharnykh, GTC 2009
- "Fast tridiagonal solvers on the GPU" Yao Zhang, Jon Cohen, John D. Owens, PPoPP 2010

JX3 Online: Background

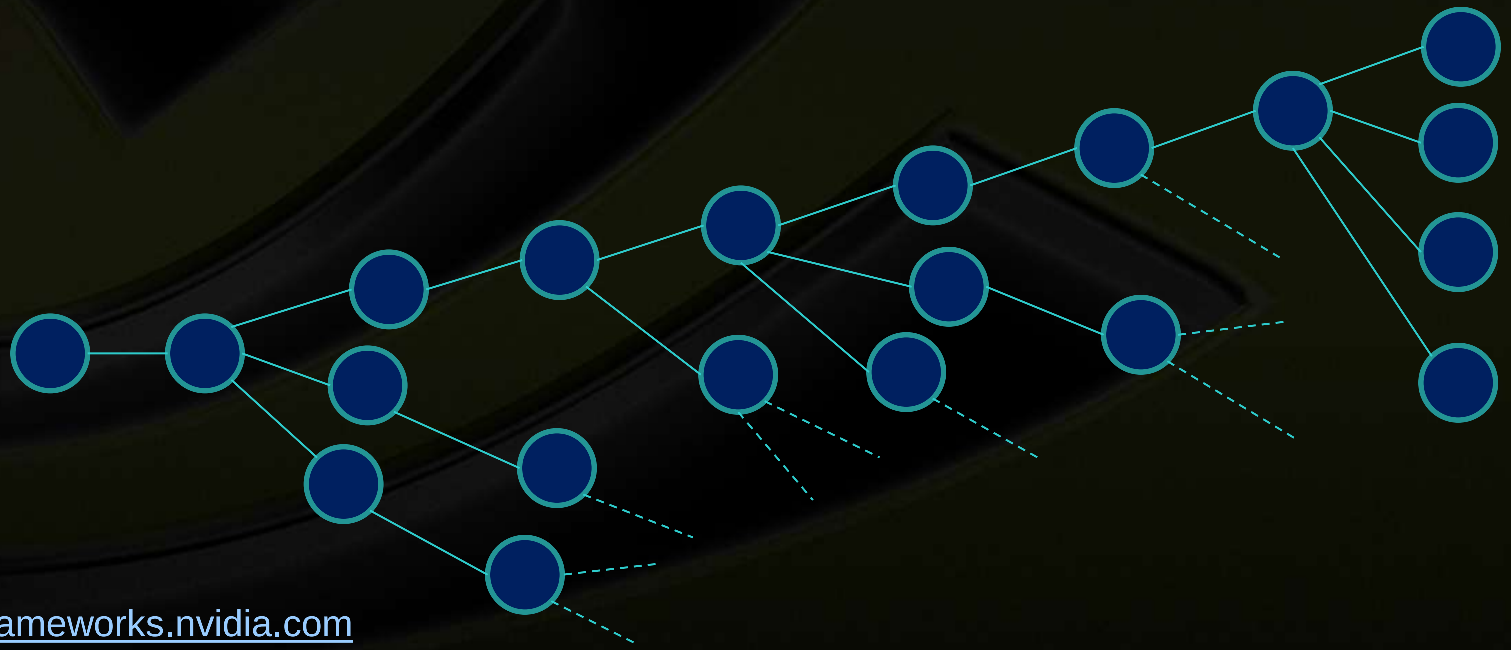
- Developer: Kingsoft Zhuhai Studio
- MMO RPG with Chinese Fantasy Setting

Character Animations in JX3

- Animation system in JX3
  - Each character: 90 ~ 120 bones, 3k ~ 5k triangles
  - 4 render passes: depth prepass, shadow, reflection & lighting
- Performance Issues
  - Original engine shows slowdown when featuring large number of onscreen characters
  - Both skeletal animation and skinning create large workload on CPU & GPU
- CUDA Animation
  - Offload skeletal animation from CPU to GPU
  - Single skinning pass for all rendering passes

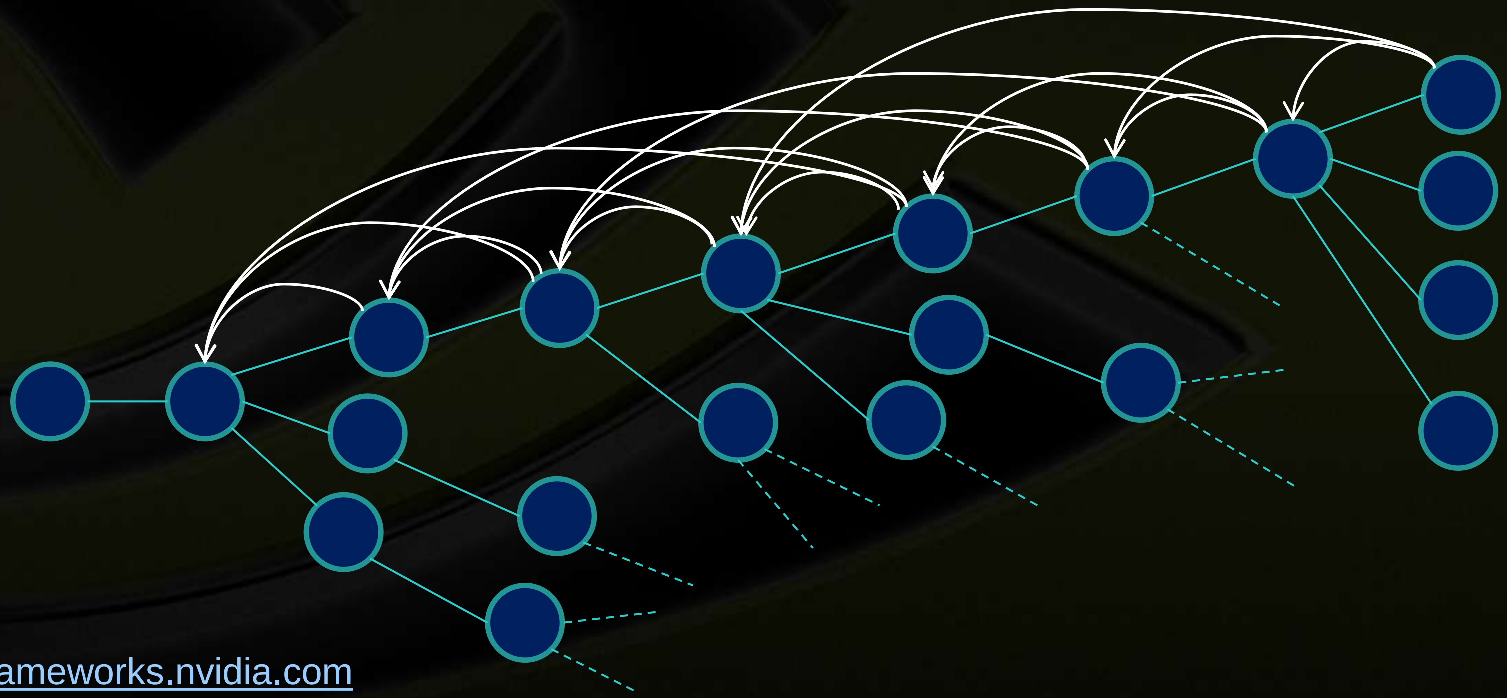
Skeletal Animation in JX3

- Each type of character maintains a skeletal tree
  - Depth: 12 ~ 15 levels
  - Width: 12 nodes at widest part (finger tips)
- Matrix update of skeletal tree
  - Original CPU code: top-down recursive updating



CUDA Skeletal Animation

- **Parallel updating of skeletal trees**
  - **CUDA code: bottom-up traverse**
  - **Each block handles a tree, each thread handles a bone (node in tree)**
  - **Node matrix math: $M'_L = M_L * M_{L-1} * M_{L-2} * M_{L-3} * \ldots M_0$ It's a prefix sum**



CUDA Skeletal Animation

- Reduce the overhead of branching
  - The topology of skeletal tree is static
  - The route between any node and the root is fixed
  - Store all node-to-root routes in a lookup table
- Reduce incoherent memory access
  - Place all intermediate matrices in shared memory, updating in-place

CUDA Skinning

- **Standard skinning processing**
  - **Similar to vertex shader skinning**
  - **Performed once per frame in CUDA**
  - **Data output to a large vertex buffer**
- **All render passes use the output of CUDA skinning**
  - **Depth prepass, shadow, reflection & lighting**
- **CUDA skinning enables draw call aggregation**
  - **Group similar draw calls into one (not possible in VS skinning due to per character bone matrices)**
  - **Draw calls number drops 80%**

CUDA Animation Performance

- **2x framerate boost for 200~300 onscreen characters**

Acknowledgements

- Many thanks to Calvin Lin, Iain Cantlay, Jon Jansen, Nikolai Sakharnykh, Callis Zhang

- Questions?

- cem@nvidia.com