LONDON UK

30 AUG TO 1 SEPT

GDCE >05

Richer Worlds for Next Gen Games: Data Amplification Techniques Survey

Natalya Tatarchuk
3D Application Research Group
ATI Research, Inc.

Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Games and Current Hardware

- Games currently are CPU-limited
  - Tuned until they are not GPU-limited
  - CPU performance increases have slowed down (clock speed hit brick walls)
  - Multicore is used in limited ways

Trends in Games Today

- Market demands larger, more complex game worlds
  - Details, details, details
- Existing games already see increase in game data
  - Data storage progression: Floppies → CDs → DVDs
  - HALO2.0 - 4.2GB
  - HD-DVD/Blueray → 20GB

[David Blythe, MS Meltdown 2005]

Motivation

- GPU performance increased tremendously over the years
  - Parallel architecture allows consumption of ever increasing amounts of data and fast processing
  - Major architectural changes happen roughly every 2 years
  - 2x speed increase also every 2 years
  - Current games tend to be
    - Arithmetic limited (shader limited)
    - Memory limited
- GPUs can consume ever increasing amounts of data, producing high fidelity images
  - GPU-based geometry generation is on the horizon
  - Next gen consoles

Why Not Just Author A Ton of Assets?

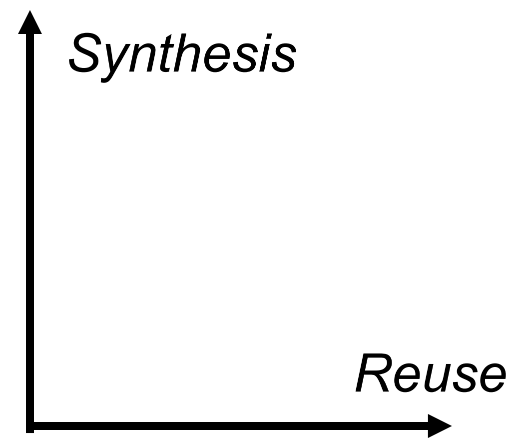
- Rising development cost
  - Content creation is the bottleneck
    - $10M content budget
  - Art Pipeline is not scaling
- Cost of authoring these datasets is increasing – skyrocketing game development costs
- Mass storage (Hard Drive and DVD) and volatile storage (RAM) are slow and small compared to the ability of the GPU to consume data
- *Amplify* the data
  - Get the most out of what we build
  - Get the most out of what we have loaded in memory at any given time

Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Database Amplification

- Alvy Ray Smith coined this term in his 1984 SIGGRAPH Paper *Plants, Fractals and Formal Languages*
- The idea is to exploit computational resources to generate extreme complexity from concise descriptions



- Two axes of data amplification

Data Amplification

- **Database amplification** – Create complex images from small datasets
- If you can generate it, an artist doesn't have to build it
  - Consoles and PCs have limited memory but monstrous GPU power
  - Network bandwidth is limited. Would be nice to "grow" data from seeds sent across the wire. Games could use this for foliage placement
  - Has LOD opportunities built right in
- **Emergence** – Complex appearance from components with simple rules

Data Amplification Techniques Survey

- Provide you with ideas on how to generate your data
- How to create large amounts of data
  - Procedural data generation ideas
- So you've generated a ton of data, now what?
  - Geometry instancing
  - LOD / data streaming
  - Data simplification

Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Procedural Data Generation

- Demoscene
- Textures
  - Compositing signals and noise
  - Hybrid textures
  - Flow-based video synthesis
- Plants
- Geometry synthesis
  - Particle systems
  - Ocean water

Procedural Data Generation

- Demoscene
- Textures
  - Compositing signals and noise
  - Fluid dynamics with Navier-Stokes equations
- Geometry synthesis
  - Particle systems
  - Ocean water

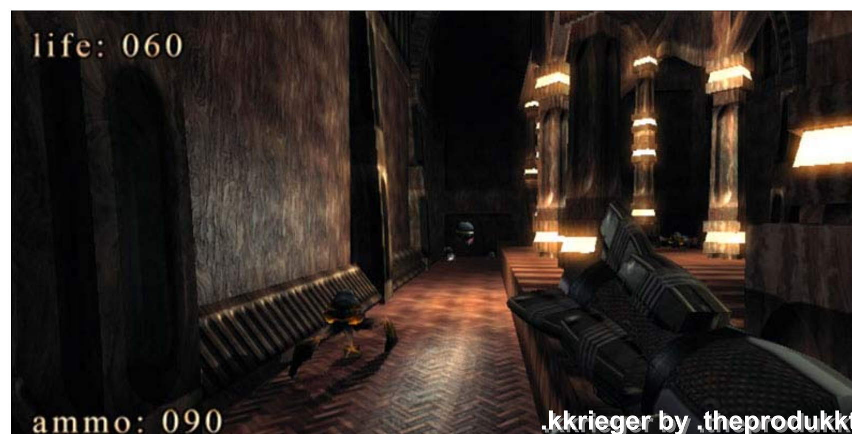
Procedural Environments

- 2D games have done this for years
- The demoscene does a ton of this
- Some games are now extending to 3D:
  - Author time
  - Run time

Demoscene

- To fit into 64kb or 96kb, the demoscene guys are doing a lot of procedural generation

- The 96kb game winner at *Breakpoint 2004* (.kkrieger by .theprodukkt) uses a lot of procedural generation and they have even posted their tools online:
  www.theprodukkt.com



Demo

.kkrieger by .theprodukkt

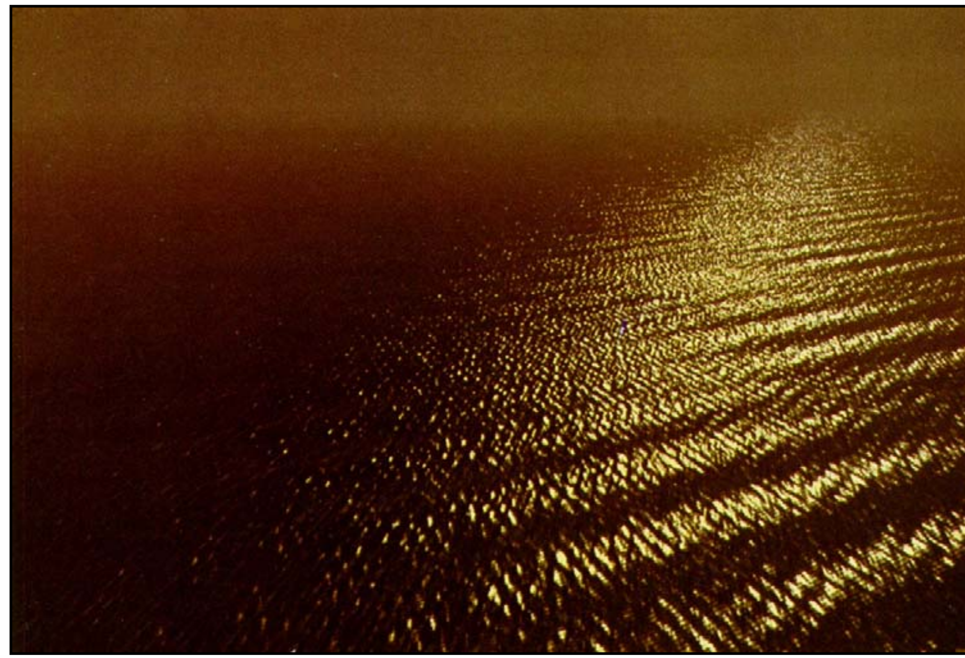
Procedural Textures

- Combine signals at different frequencies to make more stuff
- Examples
  - Clouds
  - Hybrid procedural and authored approaches
  - Wang tiles
  - Flow-based synthesis
  - Fourier-domain water synthesis

From [Dube05]

An Image Synthesizer

- [Perlin85]
- Uses noise to synthesize a variety of natural phenomena
- This same paper also introduces the ideas of volume noise and high level language pixel shaders



Synthesize New Data From Video

- Real footage of flowing media (water, fire, clouds) can rarely be matched by average synthesized material
- Use existing footage to generate new, desired sequences to fit your game purpose
- Key idea: *approximate* the video of natural phenomena by ***continuous motion of particles*** along well-defined motion lines

Flow-Based Video Synthesis

- Introduced in "*Flow-based Video Synthesis and Editing*", K. S. Bhat *et al,* Siggraph 2004
- Noted that natural phenomena such as waterfalls and streams have **time-varying appearance** but roughly ***stationary*** temporal dynamics
  - *Example:* velocity at a single fixed point on a waterfall is roughly constant over time
- Describe the phenomena in terms of particles moving through flow lines
  - Start lifetime when they enter the image
  - Exit when they become invisible or leave the image

Modeling Phenomena Through Particle Motion

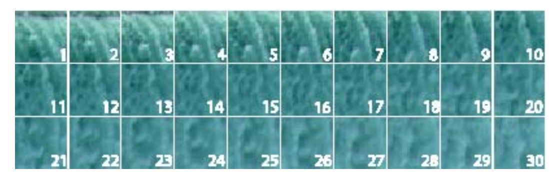
- Each particle has associated texture (a patch of pixels)
  - Changes as the particle moves along the flow line!
- User defines a set of flow lines in the video
- The system extracts the particles and their corresponding textures based on the given flow line set
  - Determines blending ("feathering") weights
- When the particle positions along the flow lines in each new image is determined, blend the particle textures to produce the rendered results

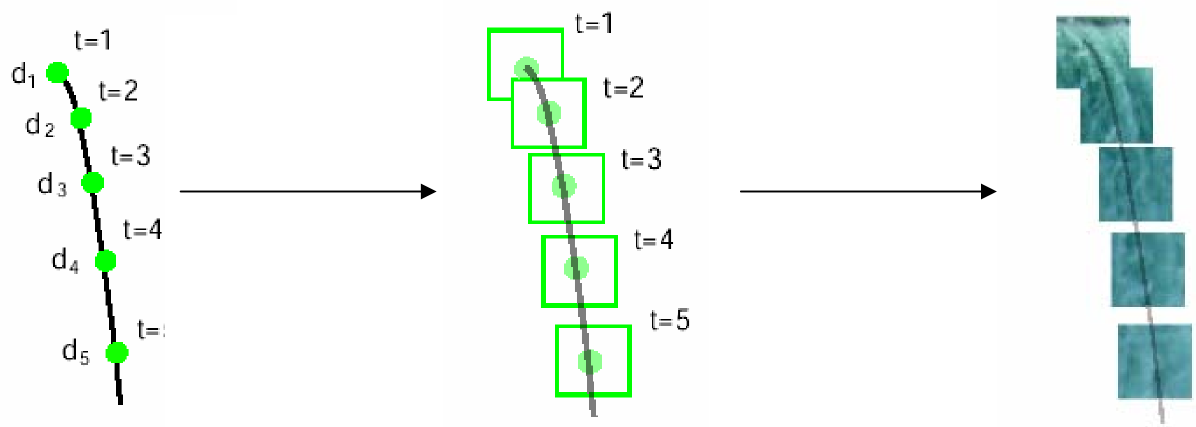
Example: Waterfall

1. Input video

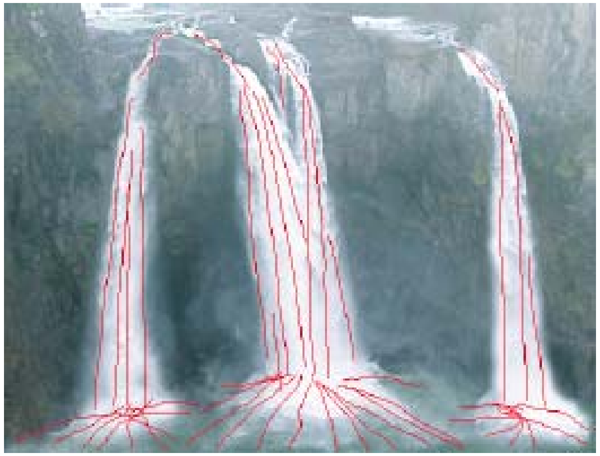
2. Define flow lines

3. Extract particles for each flow line:





Synthesize New Video Sequences

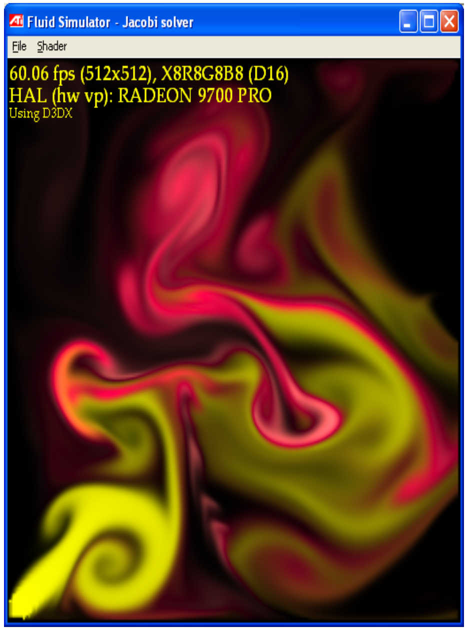


Insert new flow lines

New synthesized video

Fluid Dynamics With Navier-Stokes Equations

- It is now possible to do 2D fluid simulations on GPUs
  - "*Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering*", ATI Technical Report, P. V. Sander, N. Tatarchuk, J. L. Mitchell
  - "*Fast Fluid Dynamics Simulation on the GPU*", GPU Gems, M. Harris, UNC
- Can be useful for generating decorative smoke wisps
- On next gen consoles, can be used for gameplay / interaction
- Perform full physics computation directly on the GPU



Demo

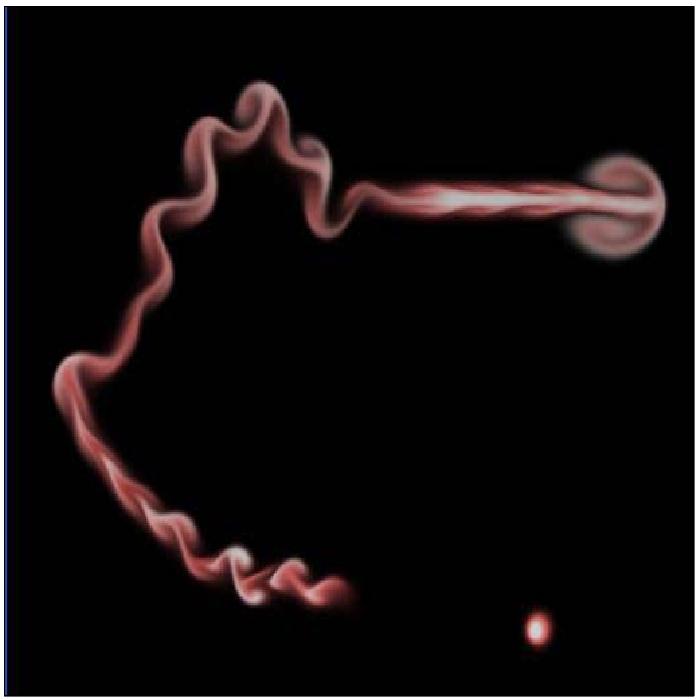
Fluid Flow

Fluid Flow

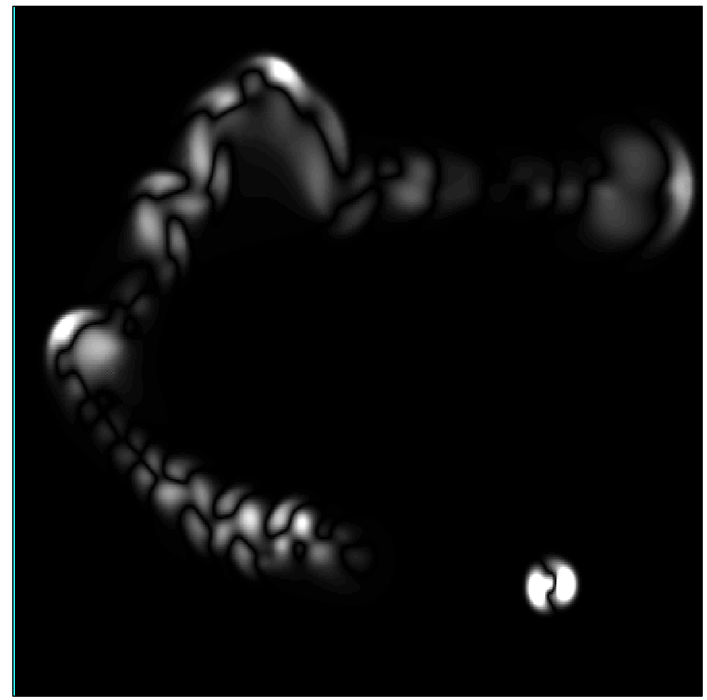
- We can use Early-Z to cull computation in some cases, since the z buffer is not needed for traditional hidden surface removal
- Fluid flow can be sparse, making it a candidate for Early-Z optimizations
- Can reduce computation in areas of low pressure to achieve faster / better simulation results

Flow Density and Pressure

Density



Pressure

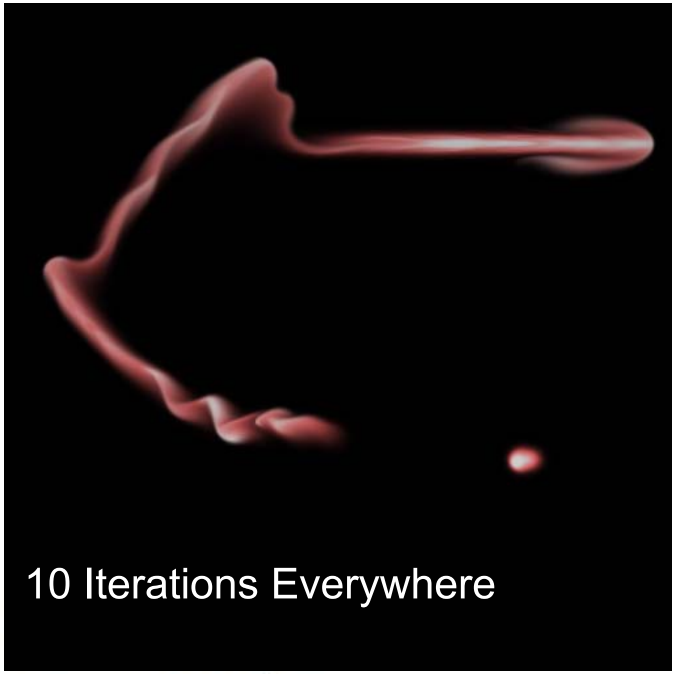


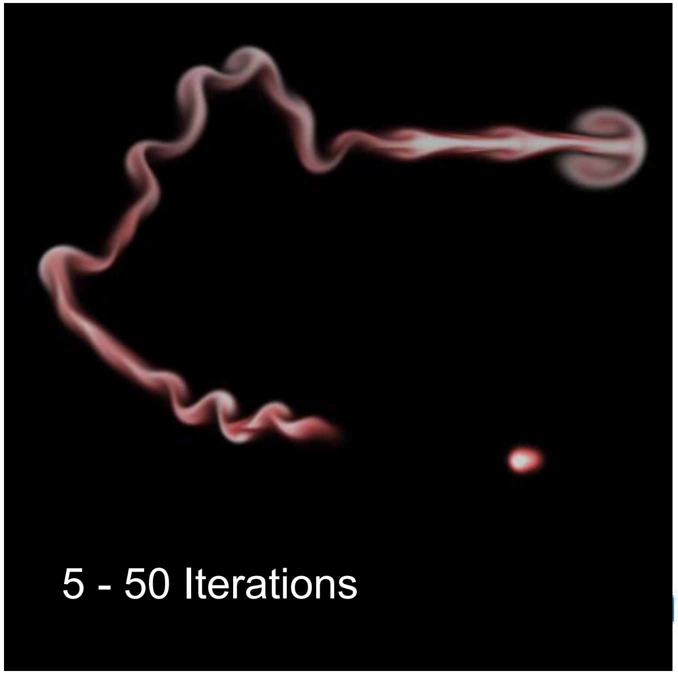
Iterations Vary With Pressure

- Assume that low-pressure regions need fewer computation iterations for convergence
- Set z buffer according to pressure buffer
- Draw 30 full screen quads in projection step
  - Vary the z from quad to quad so that the number of iterations varies with pressure
- Early-Z complete culls pixel shader instances
- Up to 3x performance improvement

Qualitative Improvement

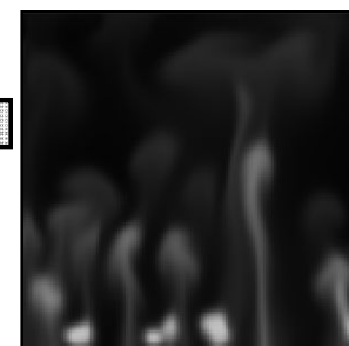
- You can alternatively look at this as a qualitative improvement
- Better simulation quality for a given frame rate





Integration into Scene

- Obviously, this doesn't have to be physically accurate, just plausible
- Once you have the implementation and the GPU cycles to burn, you can drop this sort of thing in anywhere



Demo

The Crowd demo

Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Geometry Amplification

- It's easy to play games with textures using pixel shaders, but how do we amplify our geometry?
  - ✓*Synthesis*: Make more!
  - ✓*Instancing*: Reuse the data in interesting ways which hide the replication

Geometry Synthesis

- Textures are easy to generate using pixel shaders as image processing kernels, but we want to process geometry too
- For certain 1:1 or many:1 operations, GPU-based geometry processing and generation is real
  - Really it has been around a while, but the APIs are in the way
- Want to synthesize data on demand rather than store a lot of it
  - This includes geometry!

On-demand Synthesis of Water

- Storing lots of precomputed water animation takes up lots of memory
  - Would be nice if it could be generated on demand
- Computing water animation via realistic simulation in real-time is expensive
  - It just has to be plausible
- Simply scrolling noise can look OK, but we want to do better
  - We've done scrolling noise in the past, but we can do better

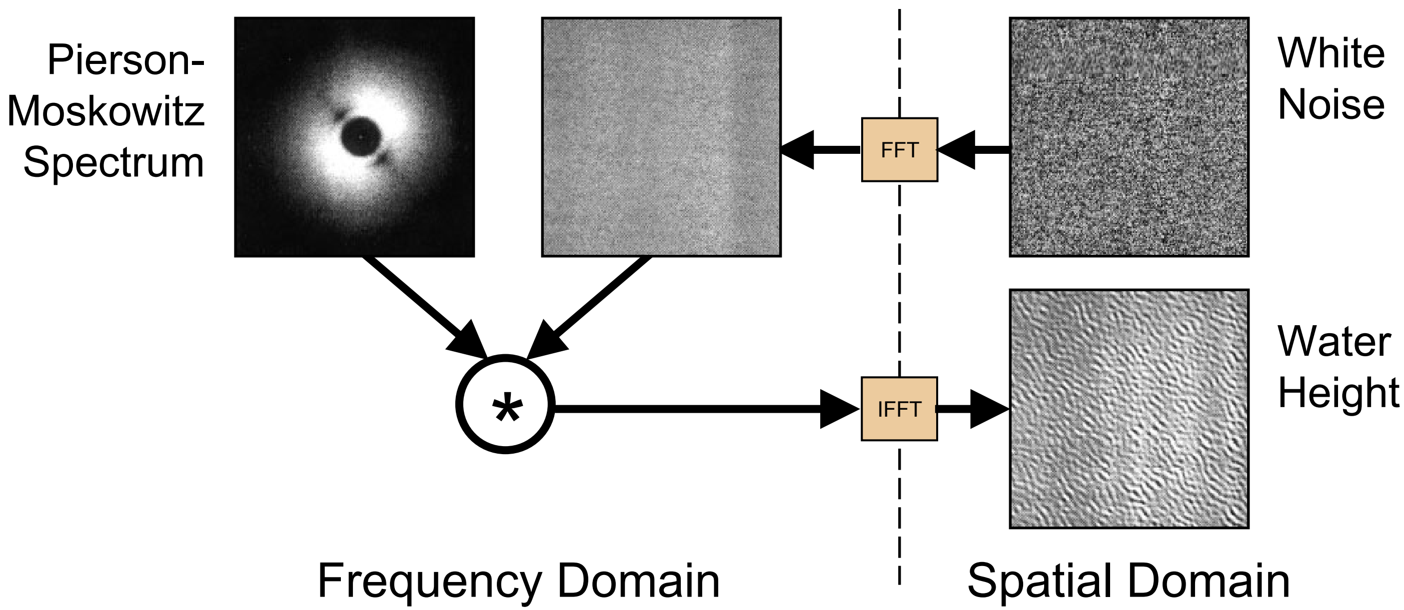
Two Classes of Approach

- Spatial domain
  - Compute superposition of a finite set of waveforms directly
  - Can be sinusoids or trochoids or something more arbitrary
- Fourier domain
  - Synthesize and animate spectrum of ocean water
  - Take IFFT to get height and normal maps

Fourier Synthesis of Ocean Scenes

- [Mastin87]
- Transformed white noise to the Fourier domain and then filtered it using a spectrum which resembles ocean water
  - Used the Pierson-Moskowitz spectrum which was derived from real ocean wave measurements
  - Relates wind speed to spectrum of sea
- Inverse FFT of the filtered result produces a tileable height map which resembles ocean waves
- Can portray wave motion by manipulating the phase

Fourier Synthesis of Ocean Scenes: The Process

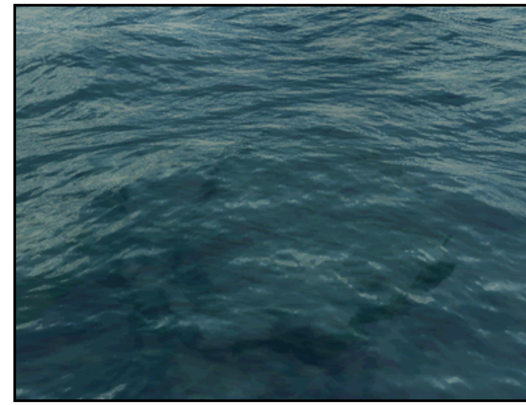


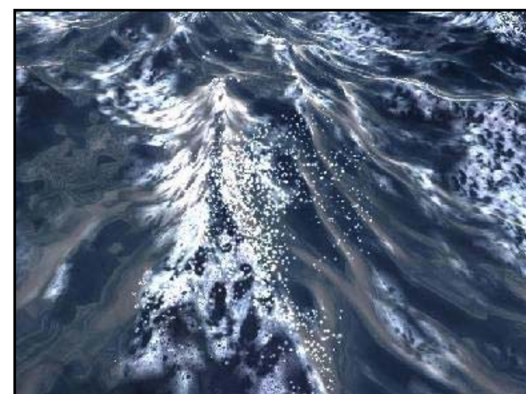
Simulating Ocean Water

- [Tessendorf99]
- Did water for *Waterworld*, *Titanic* and many others
- Works with sums of sinusoids but starts in Fourier domain
- Can evaluate at any time *t* without having to evaluate other times
- Uses the Phillips Spectrum and describes how to tune it to get desired looks
  - Roughness of the sea as a function of wind speed
  - Directional dependence to simulate waves approaching shore

Deep-Water Animation and Rendering

- [Jensen01]
- Adopted many techniques from Tessendorf, all in real time
- Used low frequencies to displace geometry and high frequencies in a normal map
- First attempt at Fourier synthesis of ocean water in real time, but IFFT was done on the CPU
- Also played with all sorts of other things like foam, spray, caustics and godrays





FFT on the GPU

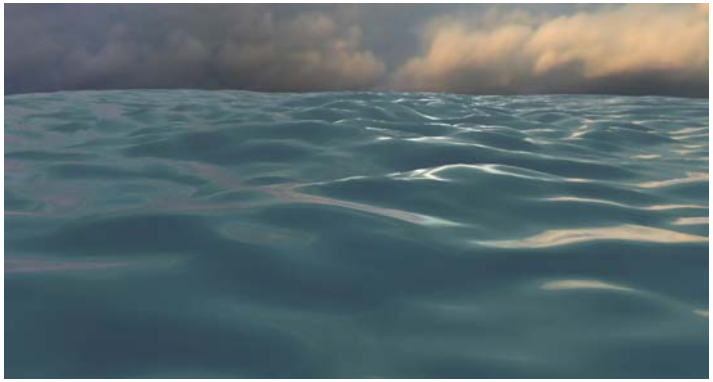
- A couple of different GPU-based FFT implementations have been developed in the last few years
  - Evan Hart developed a GPU-based FFT implementation of Cooley and Tukey's "Decimation in Time" algorithm ([Cooley65])
    - Published in the image processing chapter in ShaderX^2 [Mitchell03]
  - [Moreland03] also published a paper on doing the FFT on a GPU
- Multipass algorithm, but performs efficiently on the GPU
- Allows us to move the entire computation onto the GPU

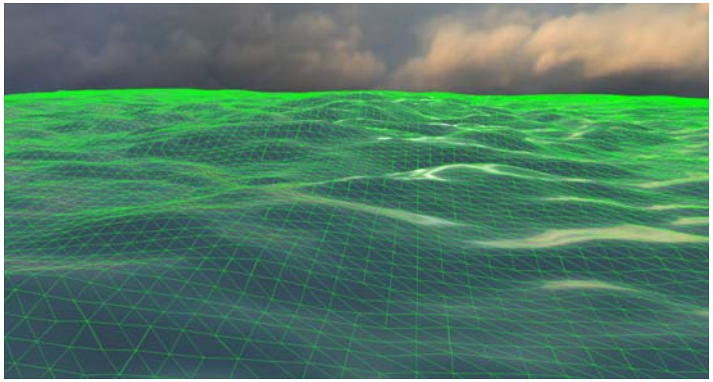
Migrate It All to The GPU

1. Load initial frequency data to static textures
2. For each frame
   a. Generate Fourier spectrum at time *t*
   b. Do IFFT to transform to spatial domain height field
   c. Filter height field to generate normal map
   d. Cast height field to vertex buffer to use as displacement stream
   e. Render mesh tiles using displacement stream and normal map to shade

Synthesized Water

- Apply synthesized height field to vertices and displace vertically
- Filter to create a normal map for shading
- *"Real-Time Synthesis and Rendering of Ocean Water"*, Jason L. Mitchell, ATI Technical Report, 2005





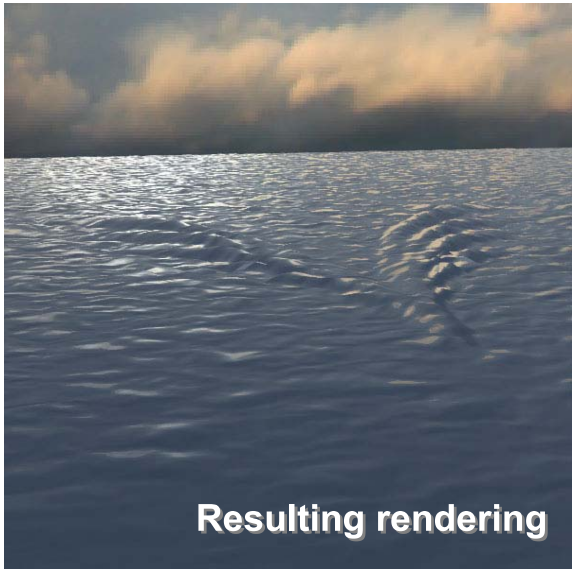
Render-To-Vertex Buffer

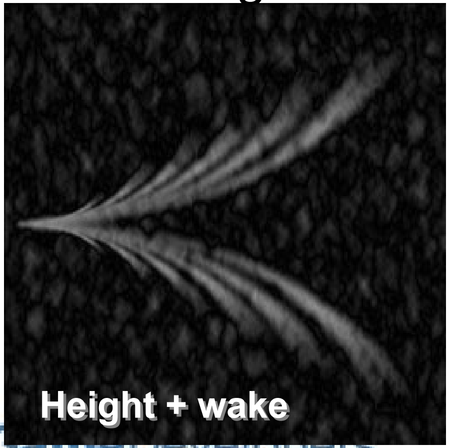
- Render vertex data into data buffer as a texture
- This buffer is directly bound as a vertex buffer afterwards
- Then go on using it as regular vertex data
- Dynamic generation of geometry and data!

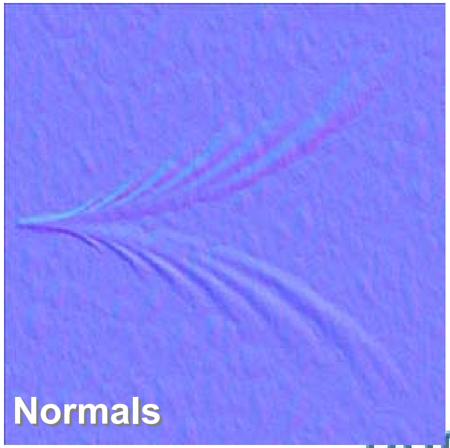
- Soon to be released, access to the beta driver through your ATI contact

Additional Waveforms

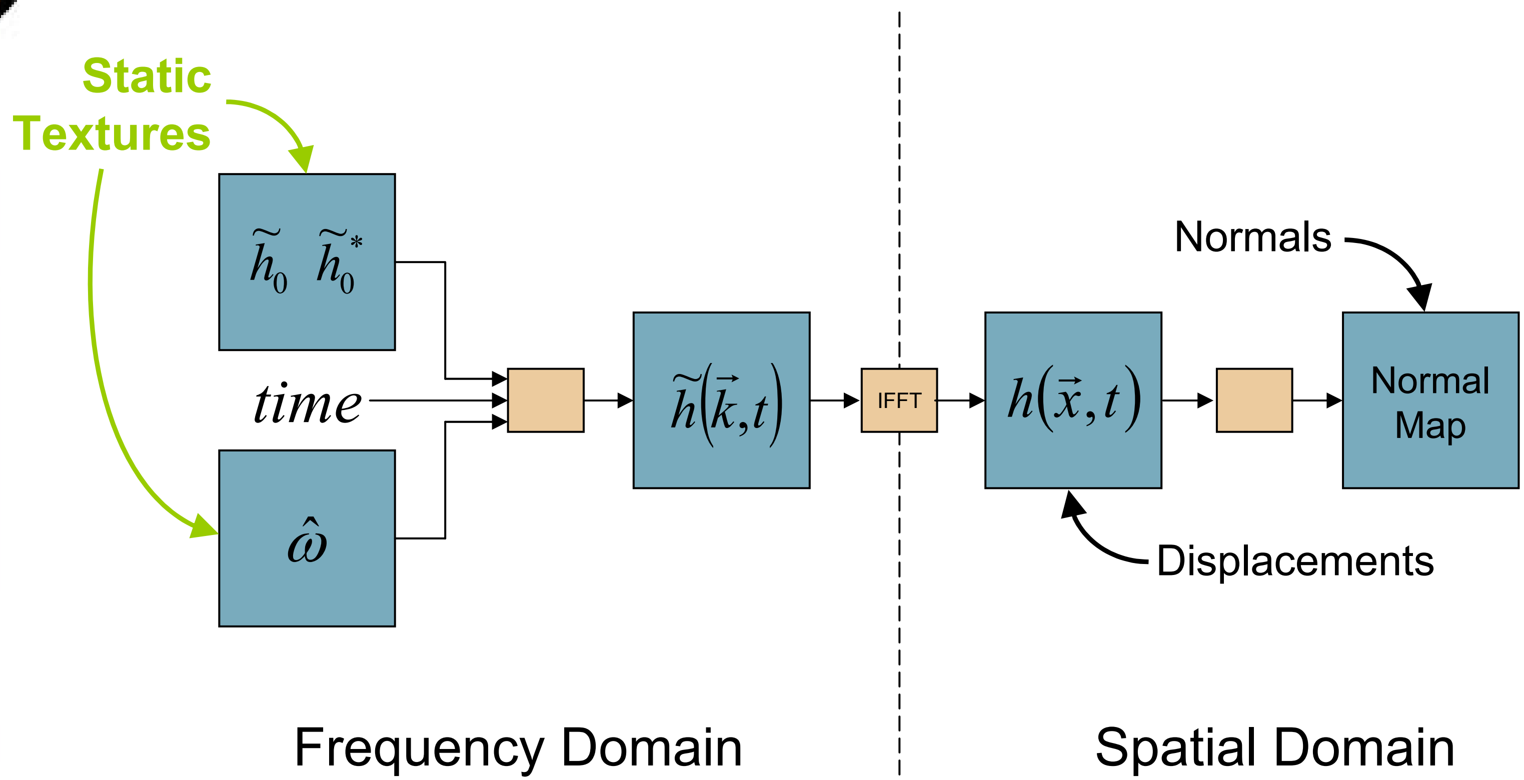
- Easy to composite wake, eddies, simulation etc
- Precomputed waveforms or real-time simulation like the Navier-Stokes simulation demonstrated earlier
- Then filter to get normals for shading



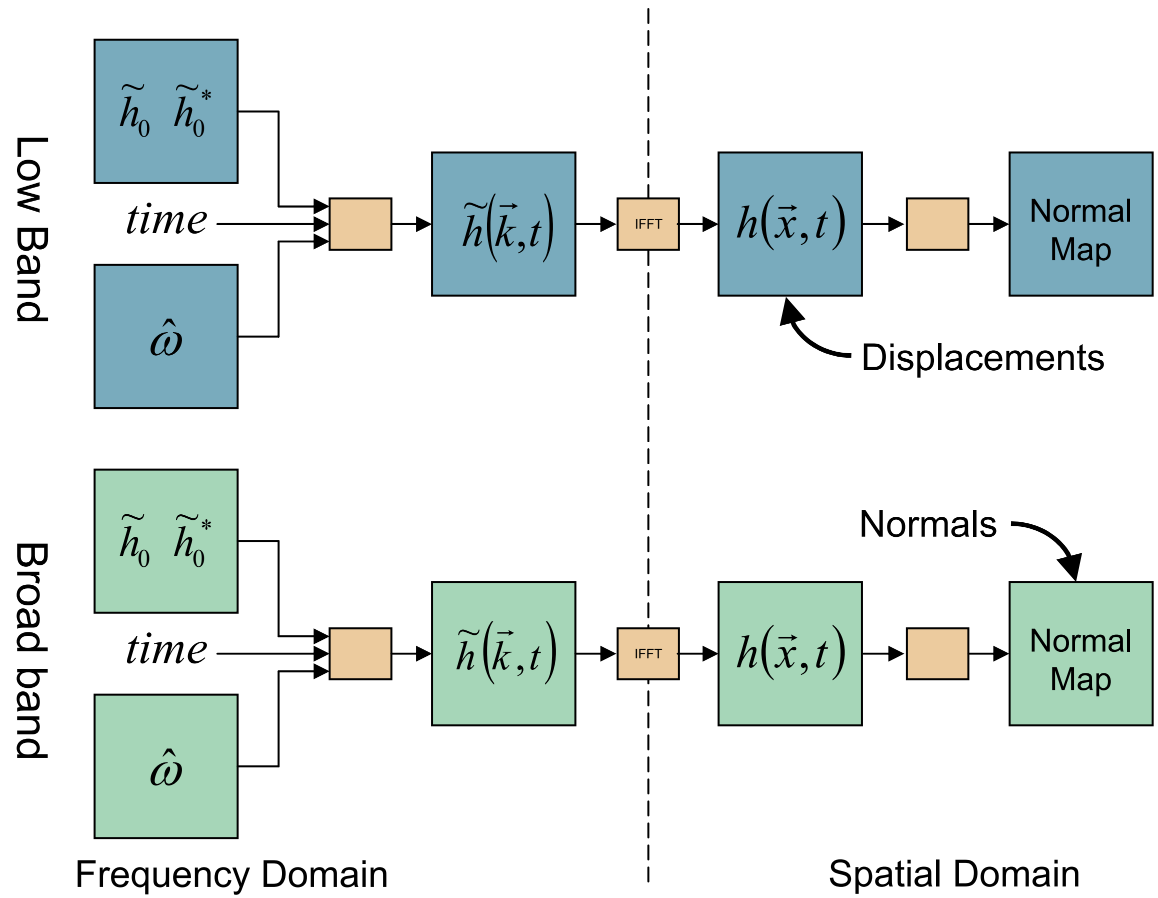




Single-band approach



Dual-band approach



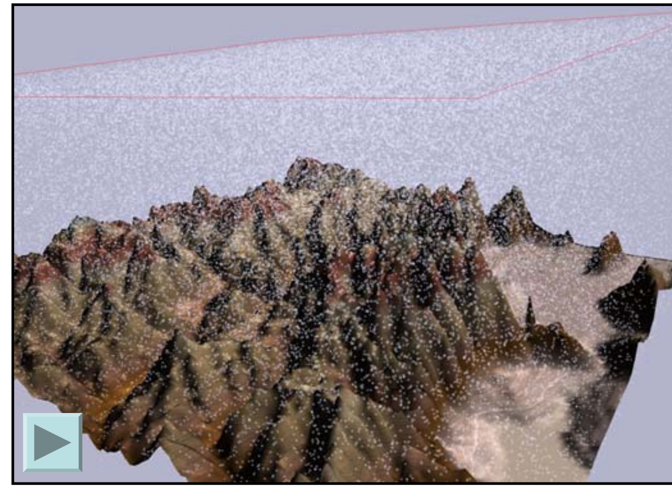
Interaction

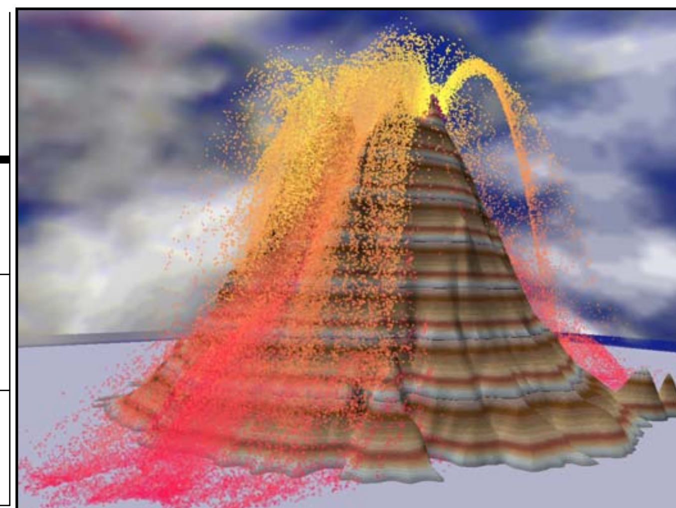
- If the GPU does the amplification, what does this do to our interactions with the world, which are simulated on the CPU?
  – Multi-resolution synthesis (low resolution on CPU for gross collision interaction & high resolution on GPU for rendering)

Particle Systems

- In [Kipfer04], the authors use the GPU to implement a particle engine they call Uberflow
- Particle-Particle and Particle-Scene collisions
- Can sort back to front
- Measure the following perf in frames per second:

	No collisions	Collisions with height field	Particle-Particle collisions	Sorting, but no collisions	CPU sorting, no collisions
256^2	640	155	133	39	7
512^2	320	96	31	8	2
1024^2	120	42	7	1.4	0.4





Instancing and Variation

- Want to use a single source model at multiple physical locations in an environment
- The best way to handle groups of similar things
  - Foliage, crowds
- Ideally done with one API call and no data replication
  - Direct3D has recently added an instancing capability
- Use shaders to generate uniqueness across instances with spatially varying parameters

Instancing in Practice

- New API in Direct3D
  - Store per-instance data in a separate data stream
  - Draw multiple instances in one shot
- Example from *Far Cry*, for a representative forest scene:
  - Approximately 24 kinds of vegetation
  - 4 types of grass (45 to 120 polygons per instance)
  - 12 types of bushes (100 to 224 polygons per instance)
  - 8 types of trees (500 to 1600 polygons per instance)
  - Instancing on some scenes is very efficient. Number of **draw-calls** (including other non-instanced objects) is **reduced from 2500 to 430**

FarCry Geometry Instancing – Results

- Depending on the amount of vegetation, rendering speed increases up to 40% (when heavily draw call limited)
- Allows them to increase sprite distance ratio, a nice visual improvement with only a moderate rendering speed hit

Slides courtesy of Carsten Wenzel, CryTek, GDC 2005


Scene from FarCry drawn normally
Slides courtesy of Carsten Wenzel, CryTek,
GDC 2005

FarCry Batches visualized – Vegetation objects tinted the same way get submitted in ***one*** draw call!

Slides courtesy of Carsten Wenzel, CryTek, GDC 2005

Another Example: Hundreds of Instanced Characters

Drawing A Crowd: The Goal

- Draw over a thousand animated characters on screen simultaneously
  - We draw ~1400
  - With shadows (a special trick)
- Individualized look
  - Colors / textures / decals
- Efficient use of API – can't simply have over a thousand draw calls!
- See "*Drawing A Crowd*" ShaderX3 article by D. Gosselin *et al* for code samples and details

Drawing a Crowd: Optimizing Draw Calls

- Reduce the number of draw calls
  - Several characters per draw call
- Pack a number of instances of character vertex data into a single vertex buffer
  - Skinning is done on GPU: Pack multiple transforms into constant store for each draw calls
  - Therefore we can draw several unique instances with its own unique animation in a single draw call!
- Limitation: constant store size
  - Depends on the number of bones for skinning
  - We use 20 → with some tweaks → draw a group of 4 characters with one draw call
- 250-300 draw calls for a crowd of 1400 characters!

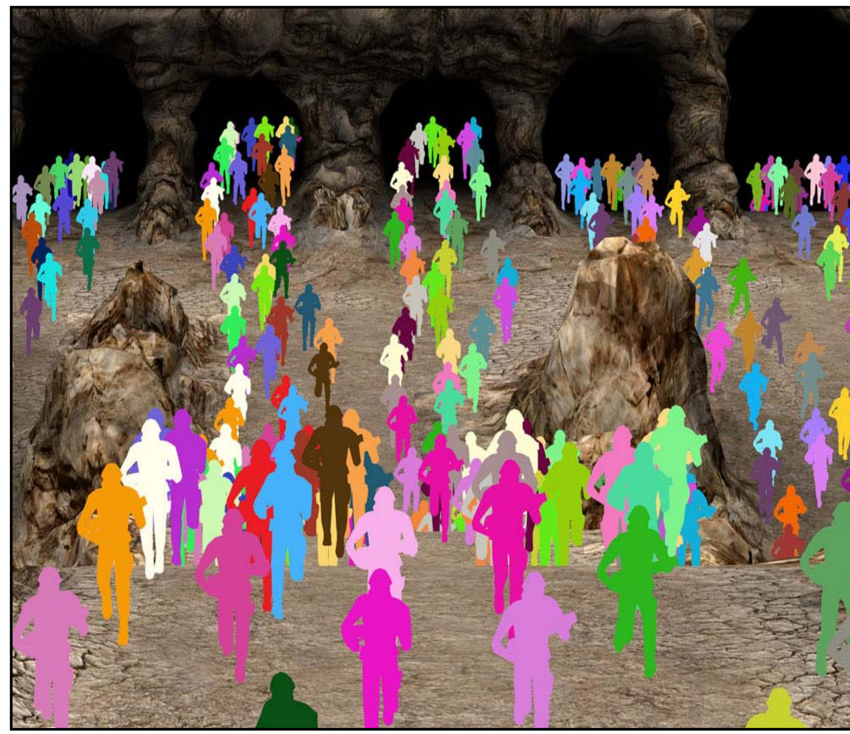
Unique Seeds for Instanced Shading

- Simply instancing characters is not enough – we don’t want a clone army
  - The characters should have individualized look
- We determine the ID of the character drawn in a given draw call using the number of bones per character
- Vertex shader selects from a set of random numbers based on character ID
- In the pixel shader, we look up into a color tint texture with the given random number and use this color to tint each character
  - Also can apply randomized decals

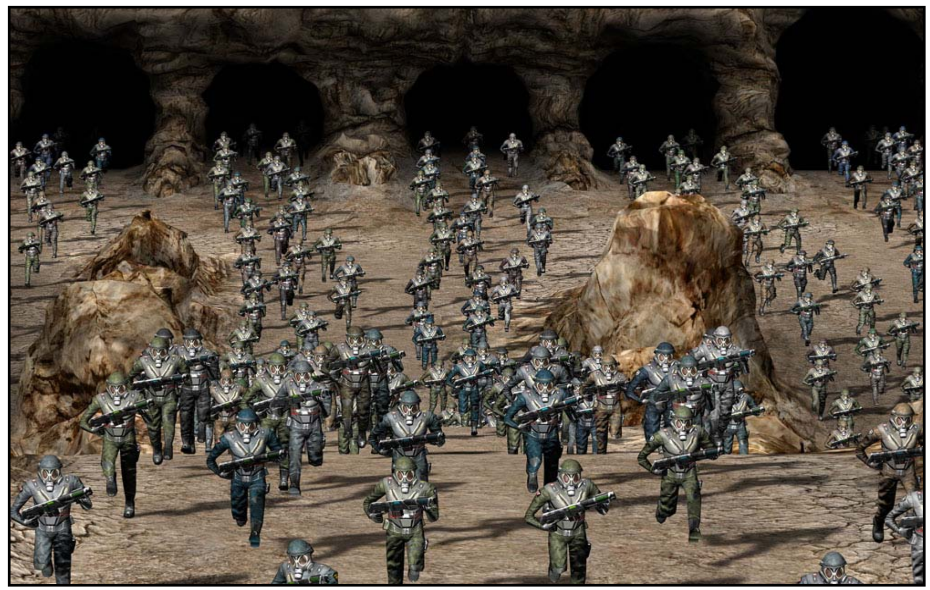
Example: Unique Seeds for Individual Look

Character tint texture

Random IDs assigned to each character for further tinting



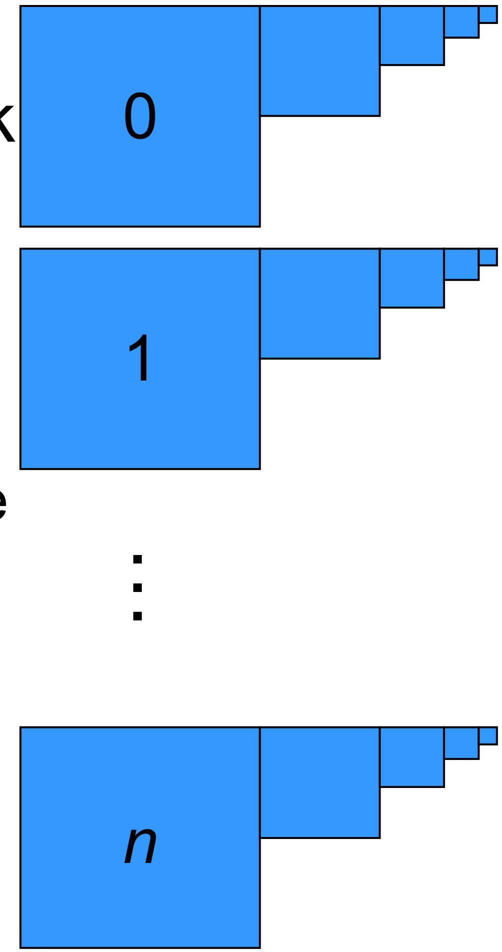
Demo



The Crowd Demo

Texture Arrays for Instancing

- Useful for providing unique look to instanced objects
  - Index into an array of textures based upon some “state” stored with each instance (like color seeds on previous slide)
  - Same “state” can be used to drive flow control as well
  - Like being able to change texture handles mid draw call
- DirectX 10 feature
- XBox 360 feature



Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

You've Got Lots of Data.. Now What?

- Once you have huge datasets (Gigs and gigs of art assets) there is the problem of getting it to the GPU for rendering
- Data streaming is one solution
  - Various custom approaches exist
  - Many utilize LOD
- Ideally, we want a smooth LOD with data streaming
  - Transparent to the player

Use Progressive Buffers!

- A preprocessing method and a rendering system for geometry and texture
- View-dependent dynamic LOD
- Suitable for variety of hardware
- Easily handles gigabytes of vertex data with textures

How Do Progressive Buffers Help You?

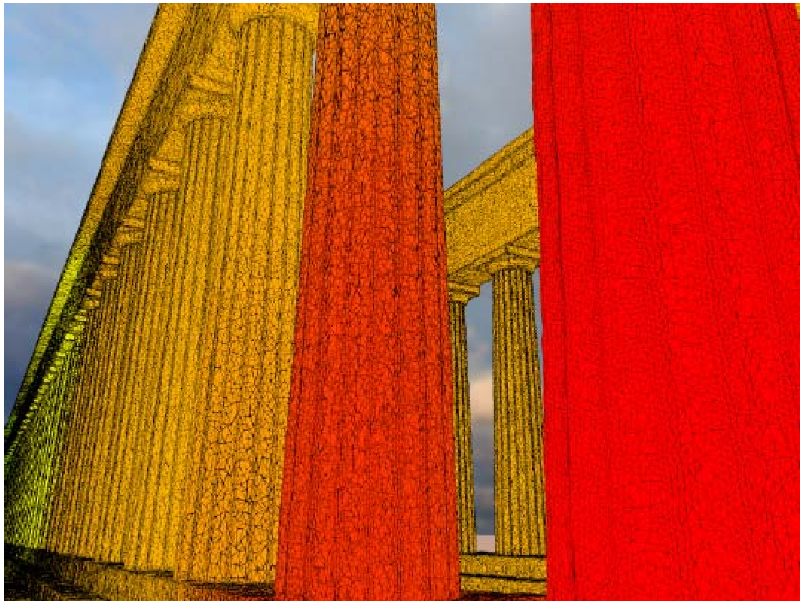
- New rendering method geomorphing the geometry using per-vertex weights (in the vertex shader)
  - Prevents LOD pops and boundary cracks
  - Uses static GPU-friendly vertex and index buffers
- Hierarchical method to render far-from-the-viewer geometry
  - Reduces the number of draw calls
- Scheduling algorithm to load the data as required on demand from disk into video memory
  - Load levels while players are finishing the current one

Progressive Buffers

- "*Progressive Buffers: View-dependent Geometry and Texture LOD Rendering*", Pedro V. Sander, Jason L. Mitchell, *Symposium on Geometry Processing*, 2005
- A data structure and system for rendering of a large polygonal model:
  - Out-of-core
  - Texture / normal-mapping support
  - Smooth transitions between levels of detail *(no popping)*

Slide2.avi

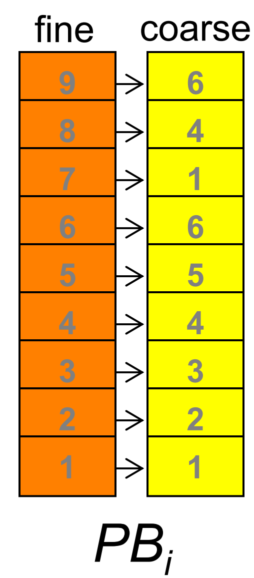
Demo



Out-of-Core Rendering with Progressive Buffers

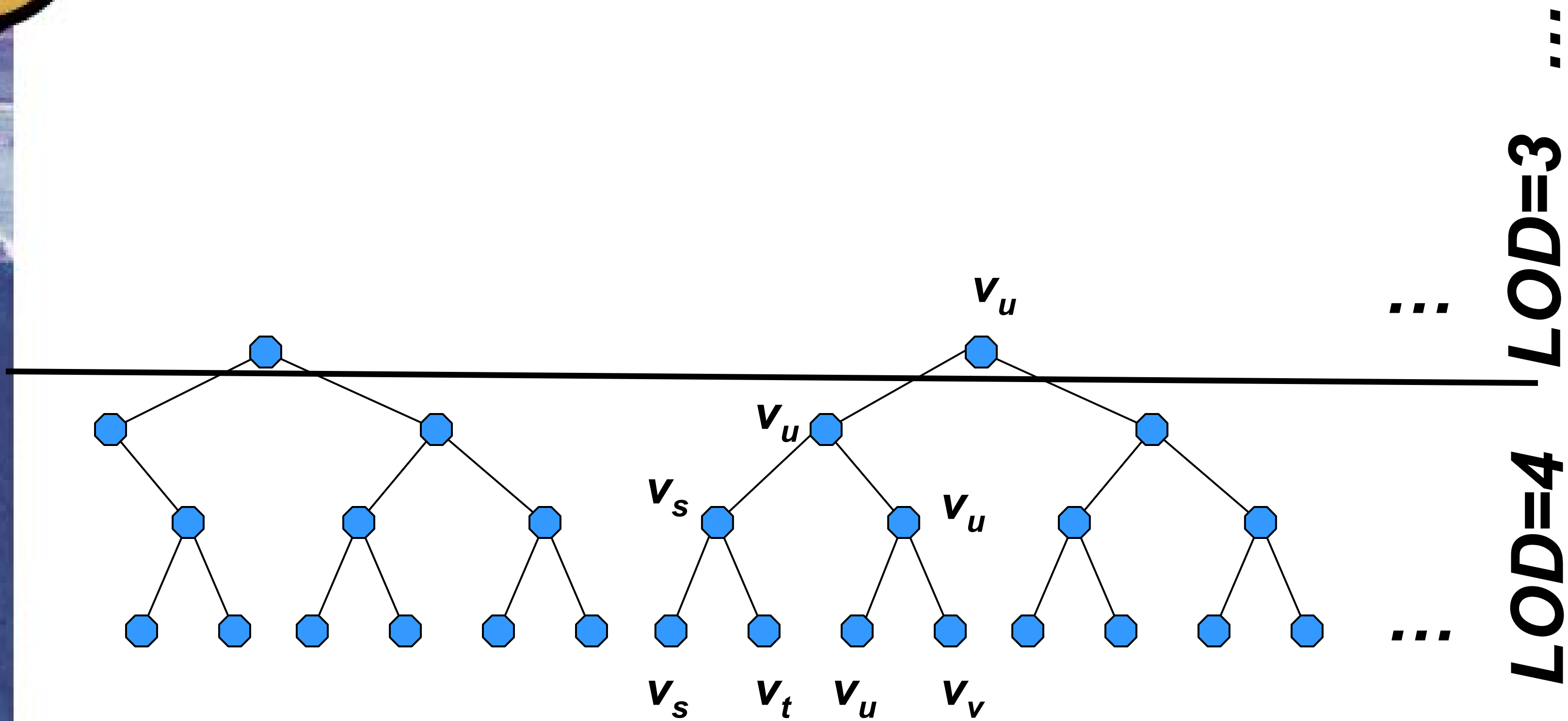
The Progressive Buffer

- Represent the mesh's LODs with several static buffers
- Each static buffer will contain an index buffer and two vertex buffers
  - Fine vertex buffer
    Representing the vertices in the current LOD
  - Coarse vertex buffer
    Vertex-aligned with the fine buffer such that each vertex corresponds to the "parent" vertex of the fine buffer in the next coarser LOD
- Combination of these buffers for all LOD is the ***"progressive" buffer***



(Note: requires ***vertex duplication****)*

The Progressive Buffer



Vertex parents for LOD = 4: $v_s, v_t, v_v \rightarrow v_u$

Progressive Buffer Construction

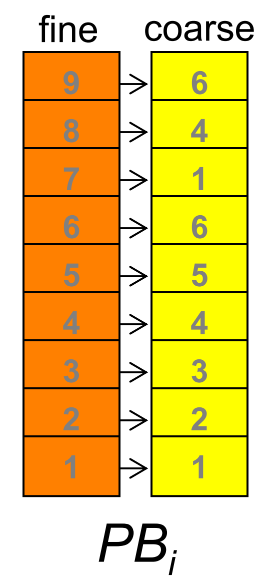
Preprocess (mostly based on previous methods):

- Split model into clusters
- Parametrize clusters and sample textures
- Create multiple (e.g., five) static vertex/index buffers for different LODs, each having ¼ of the vertices of its parent
  - Simplify each chart at time from one LOD down to the next, and simplify the boundary vertices to its neighbor
  - Simplify respecting boundary constraints and preventing texture flips
    [Cohen 98, Sander 01]
- Perform vertex cache optimization for each of these buffers [DX9; Hoppe 99]

Rendering the Progressive Buffer

Runtime:

- A static buffer is streamed to vertex shader
  *(LOD determined based on **cluster's center** distance to camera)*
- Vertex shader smoothly blends position, normal and UVs.
  *(blending weight based on **vertex** distance to camera)*

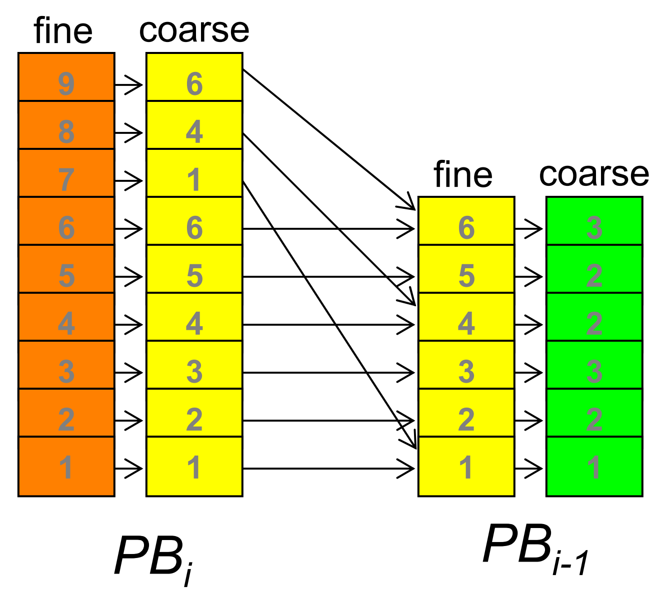


Continuous LOD Control

- Texture-mapping
  Allows for lower geometric level of detail without loss in quality (e.g., flat regions can be textured).
- Geomorphing
  A lower number of rendered triangles causes undesired popping when changing level of detail. Geomorphing provides a smoother transition.
- Summary:
  - Complex models
  - Wide range of graphics hardware
  - No need for tiny pixel-sized triangles

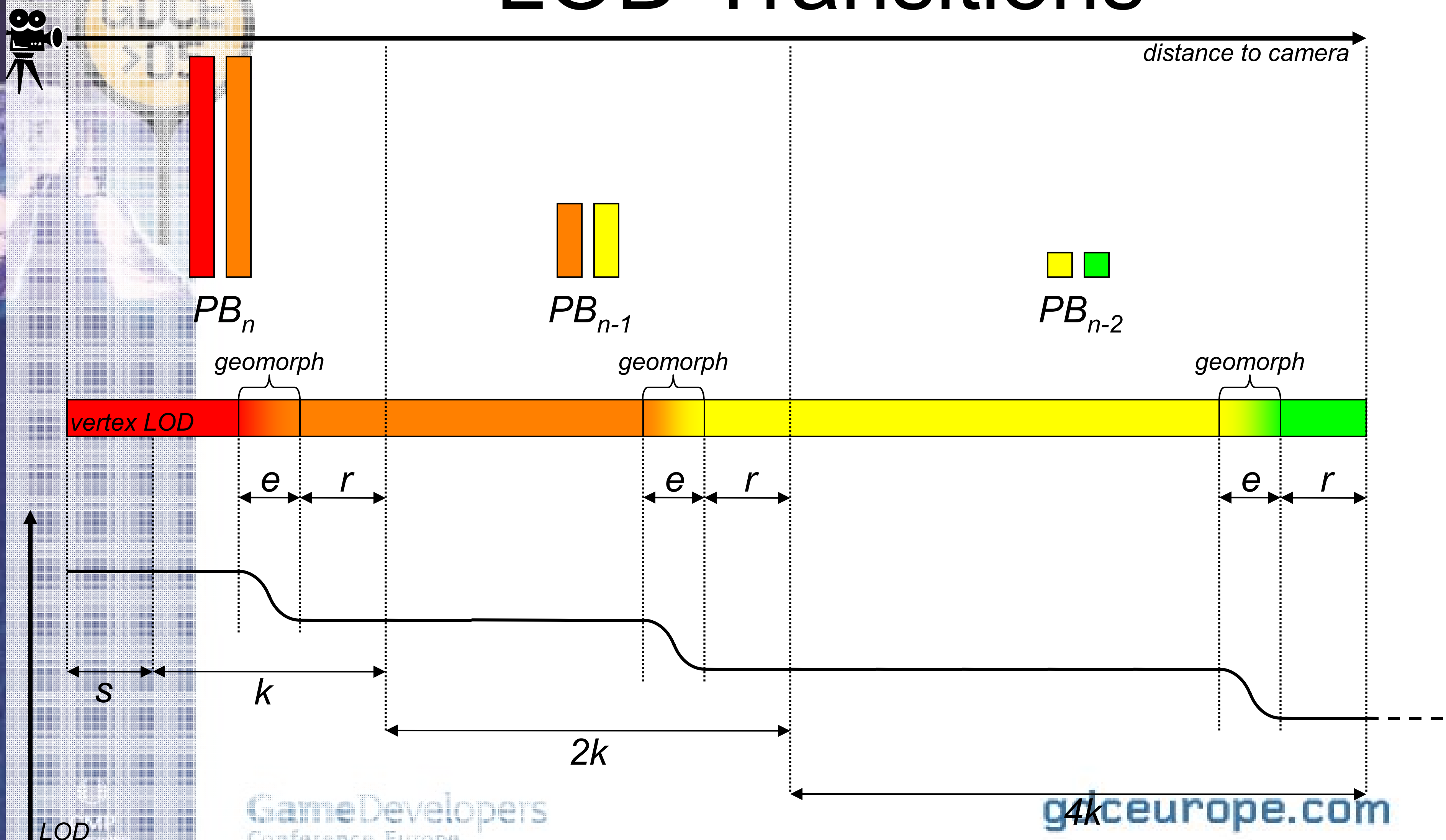
Buffer Geomorphing

- Decrease level of detail:
  - Geomorph
    PB_i orange → yellow
  - Switch buffer
    PB_i → PB_{i-1}
  - Geomorph
    PB_{i-1} yellow → green

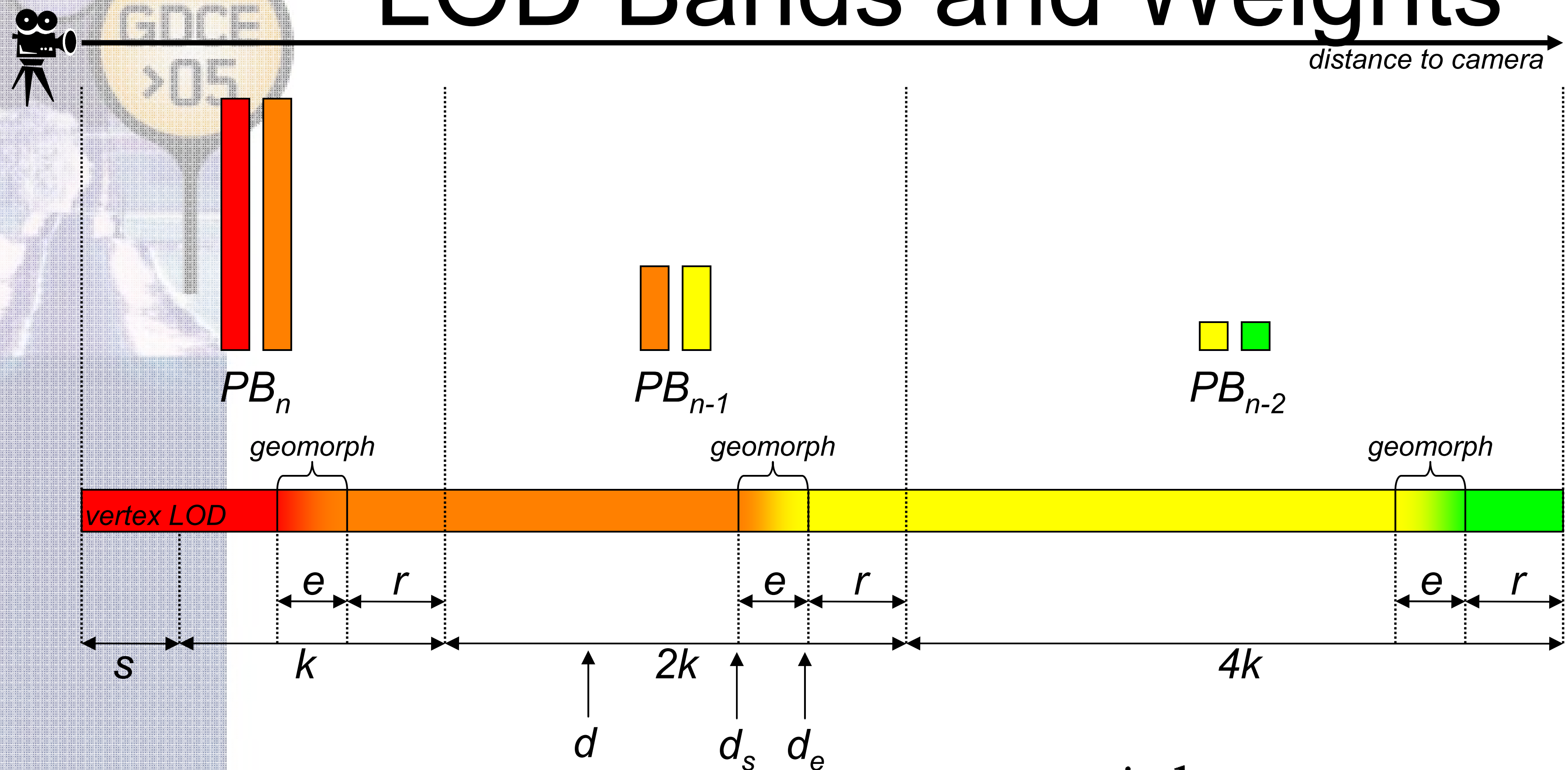


- Increase level of detail by reversing the order of operations

LOD Transitions



LOD Bands and Weights



$$i = \mathrm{floor}\left(\log_2\left(\frac{d-s}{k}+1\right)\right)$$

$$d_e = (2^{i+1}-1)k + s - r$$
$$d_s = d_e - e$$

Texture LOD

- Analogous to vertex LOD
- Each LOD also has texture
- Each coarser LOD has ¼ of the # of vertices and ¼ of the # of texels of the previous LOD
- Essentially, we drop the highest mip level when coarsening, and add a mip level when refining
- Textures are blended just like vertices:
  – Vertex geomorph weight passed down to pixel shader
  – Pixel shader performs two fetches (one per LOD)
  – Pixel shader blends resulting colors according to the interpolated weight

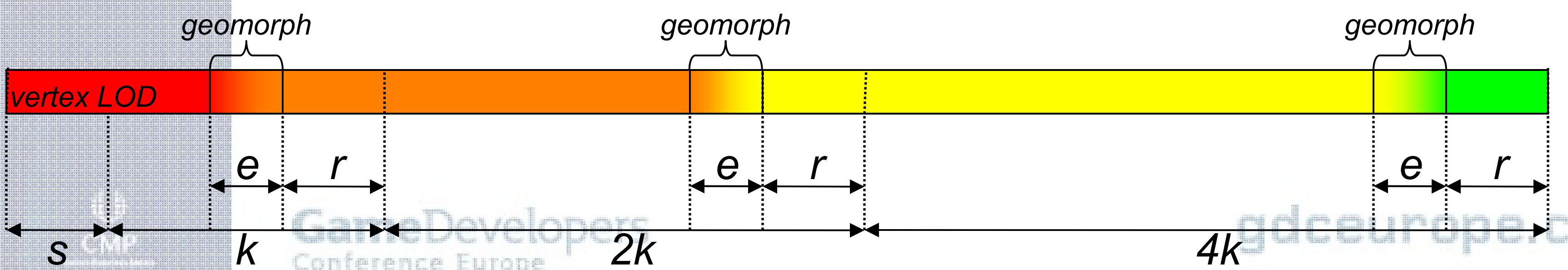
Slide14.avi

Limitations of Data Structure

- Vertex buffer size is doubled
  (but only small subset of data resides in video memory)
- Clusters should be about the same size
  (a large cluster would limit minimum LOD band size)
- Larger number of draw calls than purely hierarchical algorithms
  (cannot switch textures within same draw call;
  coarse level hierarchy partly addresses this)
- Texture stretching due to straight boundaries

Automatic LOD Control

- Bounds:
  - System memory
  - Video memory
  - Framerate (less stable)
  - Maximum band size
- Values of *k* and *s* slowly adjusted accordingly to remain within the above bounds



Memory Management

- Separate thread loads data, and based on distance to viewer sets priorities as follows:

Priority	**System memory**	**Video memory***	**Sample bounds**
3 (active)	Yes	Yes	100MB
2 (almost active)	Yes	Yes	20MB
1 (needed soon)	Yes	No	50MB
0 (not needed)	No	No	Full dataset

*Priority (with LRU as tie-breaker) used for determining what is loaded on video memory

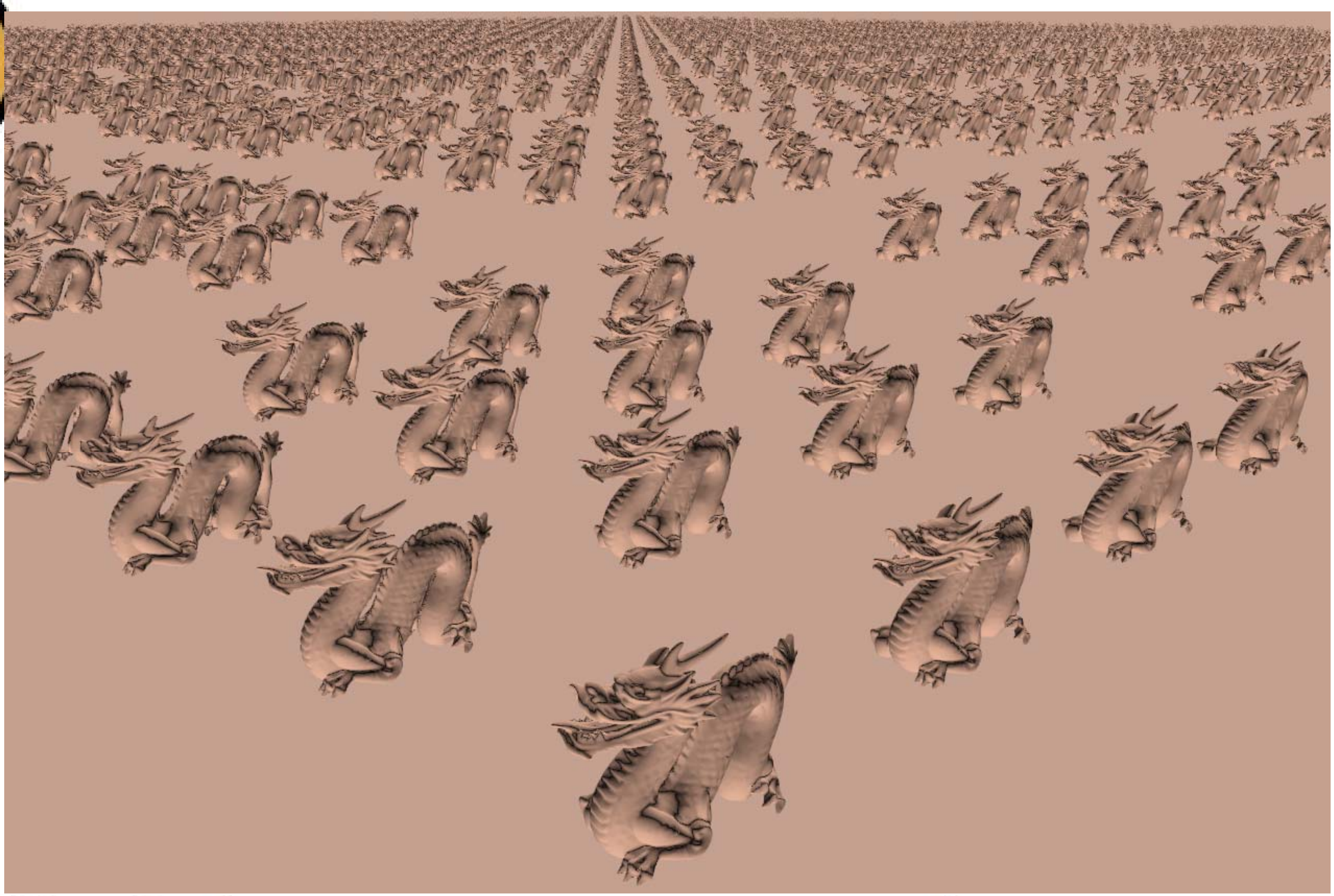
Computing Continuous LOD

- We compute continuous LOD of each buffer.
- Taking the integer part, we get the static buffer, and assign it priority 3: $i = \mathrm{floor}\left(\log_2\left(\frac{d-s}{k}+1\right)\right)$
- If the continuous LOD is within a specified threshold of another static buffer's LOD, we set that buffer's priority accordingly:

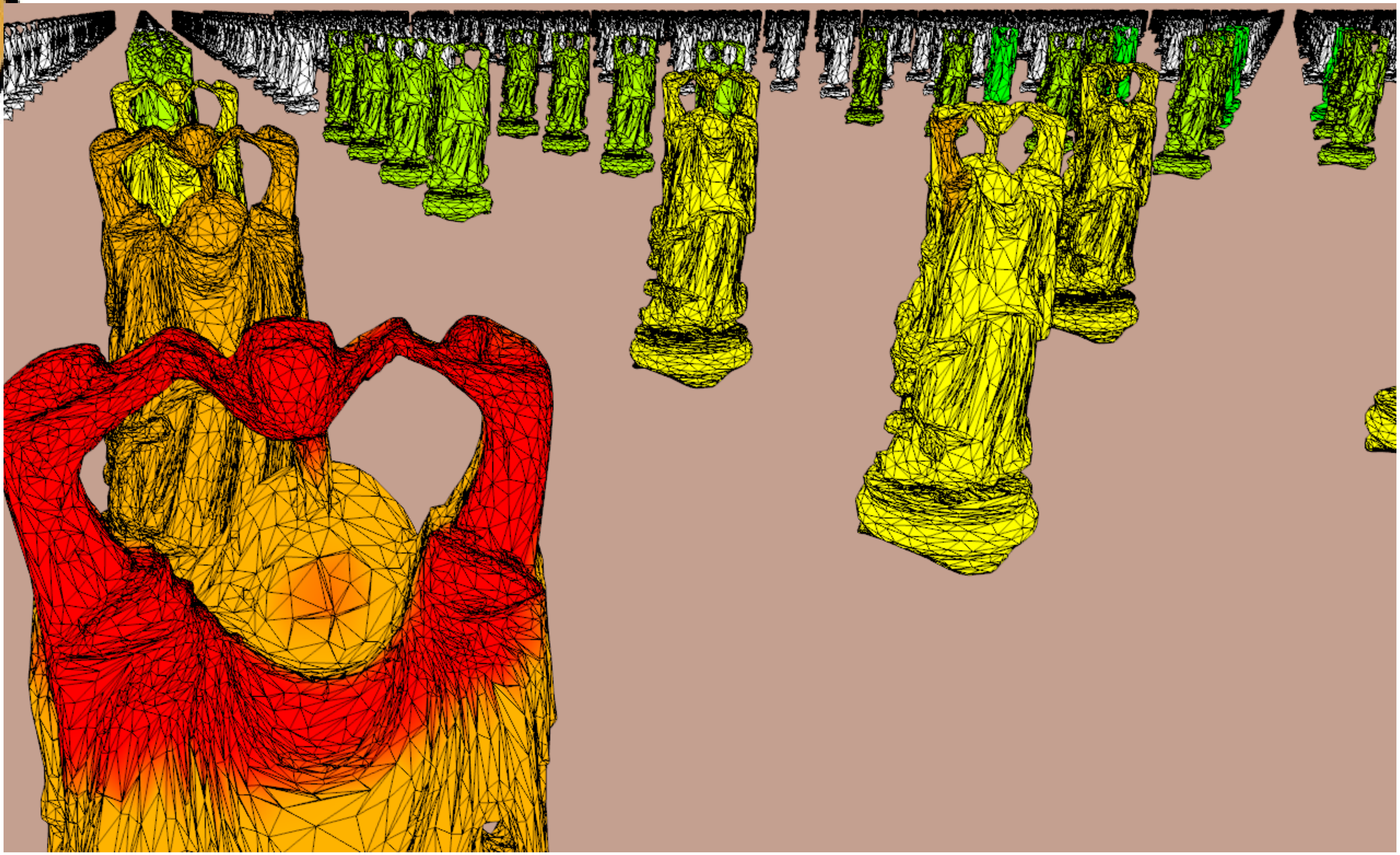
Threshold	Priority	Target	Example
e_{video}	2	Video memory	0.75
e_{system}	1	System memory	1.00

Instancing Example

1600 dragons, 240M polygons



Instancing Example



Overview

- Defining the problem and motivation
- Data Amplification
- Procedural data generation
- Geometry amplification
- Data streaming
- Data simplification
- Conclusion

Objective

- We want to render very detailed surfaces
- Don't want to pay the price of millions of triangles
  - Vertex transform cost
  - Memory footprint
- Want to render those detailed surfaces accurately
  - Preserve depth at all angles
  - Dynamic lighting
  - Self occlusion resulting in correct shadowing

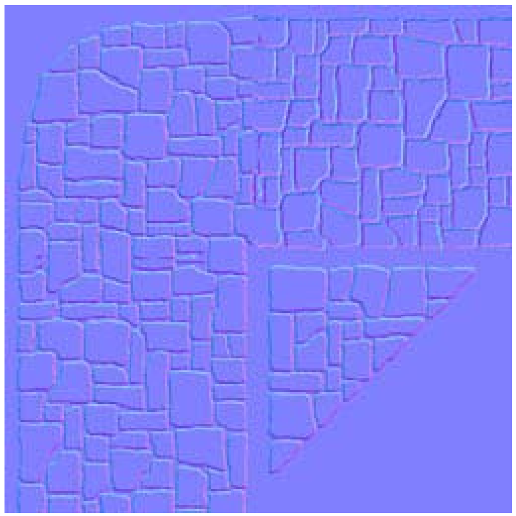
Parallax Occlusion Mapping

- Per-pixel ray tracing at its core
- Correctly handles complicated viewing phenomena and surface details
  - Displays motion parallax
  - Renders complex geometric surfaces such as displaced text / sharp objects
  - Uses occlusion mapping to determine visibility for surface features (resulting in correct self-shadowing)
  - Uses flexible lighting model

Contributions

- Increased precision of height field – ray intersections per-pixel
- Dynamic real-time lighting of surfaces with soft shadows due to self-occlusion under varying light conditions
- Directable level-of-detail control system with smooth transitions between levels
- Motion parallax simulation with perspective-correct depth

Encoding Displacement Information



Tangent-space normal map

Displacement valu

All computations are done in tangent space, and thus can be applied to arbitrary surfaces

Implementation: Per-Vertex

- Compute the viewing direction, the light direction in tangent space
- May compute the parallax offset vector (as an optimization)
  - Interpolated by the rasterizer

Implementation: Per-Pixel

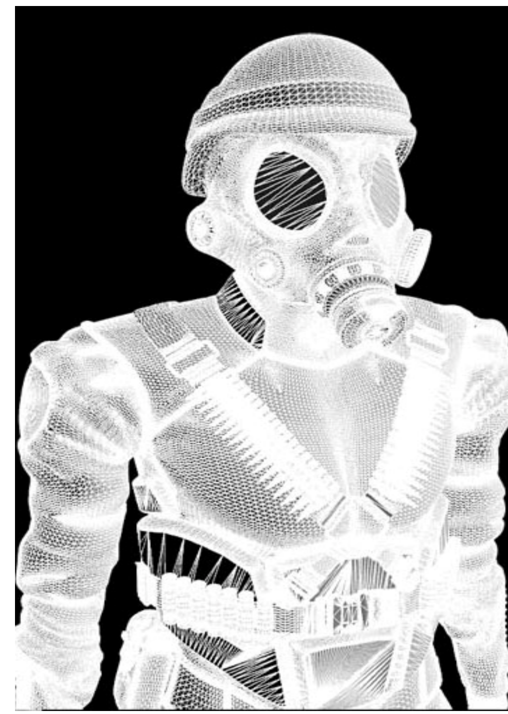
- Ray-cast the view ray along the parallax offset vector
- Ray – height field profile intersection as a texture offset
  - Yields the correct displaced point visible from the given view angle
- Light ray – height profile intersection for occlusion computation to determine the visibility coefficient
- Shading
  - Using any attributes
  - Any lighting model

Adaptive Level-of-Detail System

- Compute the current mip map level
- For furthest LOD levels, render using normal mapping (threshold level)
- As the surface approaches the viewer, increase the sampling rate as a function of the current mip map level
- In transition region between the threshold LOD level, blend between the normal mapping and the full parallax occlusion mapping

Parallax Occlusion Mapping vs. Actual Geometry

An 1,100 polygon object rendered with parallax occlusion mapping (wireframe)



A 1.5 million polygon object rendered with diffuse lighting (wireframe)

Parallax Occlusion Mapping vs. Actual Geometry

-1100 polygons with parallax occlusion mapping (8 to 50 samples used)
- **Memory**: 79K vertex buffer
  6K index buffer
  13Mb texture (3Dc)
  (2048 x 2048 maps)

Total: **< 14 Mb**

Frame Rate:
- ***255 fps*** on ATI Radeon hardware
- ***235 fps*** with skinning

- 1,500,000 polygons with diffuse lighting
- **Memory**: 31Mb vertex buffer
  14Mb index buffer

Total: **45 Mb**

Frame Rate:
- ***32 fps*** on ATI Radeon hardware

Demo

Parallax Occlusion Mapping

Parallax Occlusion Mapping: Summary

- Powerful technique for rendering complex surface details in real time
- Produces excellent lighting results
- Has modest texture memory footprint
  - Comparable to normal mapping
- Efficiently uses existing pixel pipelines for highly interactive rendering
- Supports dynamic rendering of height fields and animated objects

Conclusion

- Data amplification can be done now!
  - Current hardware is powerful enough
  - Next gen consoles offer many new options
- Generate data on the file
  - Procedural data amplification
  - Geometry synthesis
- Use individualized geometry instancing to draw many objects
  - Crowds of characters
  - Forests
  - Herds of animals – you'll come up with many examples!
- Data streaming can be made to work efficiently with GPUs
  - Progressive buffers technique make it possible
- Simplify geometry and use other algorithms (like parallax occlusion mapping) to render complex scenes

Thank You

- Jason L. Mitchell, Valve
- Zoe Brawley, Relic Entertainment
- Pedro V. Sander, ATI Research
- Dan Roeger, Daniel Szecket, Abe Wiley and Eli Turner – our artists
- ATI 3D Applications Research Group and the demo team

Questions?

- devrel@ati.com

References

- [Barrett04] Sean Barrett, “Hybrid Procedural Textures,” *Game Developer Magazine*, October 2004
- [Bhat04] Kiran S. Bhat, Steven M. Seitz, Jessica K. Hodgins and Pradeep K. Khosla, “Flow-based Video Synthesis and Editing,” SIGGRAPH 2004.
- [Cohen03] Michael F. Cohen, Jonathan Shade, Stefan Hiller and Oliver Deussen, “Wang Tiles for Image and Texture Generation,” SIGGRAPH 2003, July, 2003
- [Cooley65] James W. Cooley and John W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series.” Math. Comput. 19, 297-301, 1965.
- [Dube05] Jean-François Dubé, “Realistic Cloud Rendering on Modern GPUs” in *Game Programming Gems 5*, Charles River Media 2005
- [Ebert03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin and Steven Worley, *Texturing and Modeling: A Procedural Approach*, Morgan Kaufmann 2003.
- [Jensen01] Lasse Staff Jensen and Robert Goliáš, “Deep-Water Animation and Rendering,” Game Developers Conference Europe, 2001. http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm

References (cont.)

- [Kipfer04] Peter Kipfer, Mark Segal and Rüdiger Westermann , “UberFlow: A GPU-Based Particle Engine,” Graphics Hardware 2004
- [Mastin87] Gary A. Mastin, Peter A. Watterger, and John F. Mareda, “Fourier Synthesis of Ocean Scenes,” *IEEE Computer Graphics and Applications*, March 1987, p. 16-23.
- [Mitchell03] Jason L. Mitchell, Marwan Y. Ansari and Evan Hart, “Advanced Image Processing with DirectX 9 Pixel Shaders” in *ShaderX 2 - Shader Tips and Tricks*, Wolfgang Engel editor, Wordware, Sept. 2003.
- [Moreland03] Kenneth Moreland and Edward Angel, “The FFT on a GPU,” *SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings*, pp. 112–119, July 2003.
- [Perlin85] Ken Perlin, “An Image Synthesizer,” SIGGRAPH 1985.
- [Tessendorf99] Jerry Tessendorf, “Simulating Ocean Water,” *Simulating Nature: Realistic and Interactive Techniques* Course Notes, SIGGRAPH 1999