



Secrets of the NVIDIA Demo Team

GPU Jackpot
Las Vegas 2004



Introduction

Eugene d'Eon
NVIDIA Demo Team



Outline

- Our Demo Engine
- Dawn
- Vulcan
- Nalu
- Timburry
- ClearSailing



Disclaimers

- We love pushing the boundaries back
- We throw a lot of resources at a single scene
 - Not everything we get away with is necessarily applicable to today's games
- No ogres, fairies, mermaids or were-wolfs were harmed in the making of our demos. (but a few artists were...)



Our Engine - NVDemo

- Custom scene description file format (editable ascii, and binary versions)
- Maya Exporter
 - we could support other packages
- OpenGL and DirectX layers
 - Have used strictly OpenGL for the last 3 year
- Custom Real-time Adaptive subdivision surface engine
- Cg



Models

- Polygons (and lines) only
- Custom real-time subdivision surface tessellation engine within NVDemo
 - Subdivides only visible geometry
 - Varies complexity based on how close the camera is
 - Catmull-Clark or Doo-Sabin
 - Works with skinning and Blend-Shapes
- Create normal maps using Melody
- Our artists use Photoshop, DeepPaint 3D, ZBrush



Animation

- Simple transform animation
- Blend shapes
 - CPU and GPU animation paths
- Skinning
 - CPU and GPU skinning selectable per mesh
- Mesh Keyframes
 - Good for small meshes with arbitrary rigs within any animation package (Ogre)
- We match Maya's curve animation exactly
- Procedural animation



Physics

- Custom physics solutions engineered as needed (Nalu's hair & fins, Pirate ship)
- Bake dynamics from Maya or another package
 - Fairly limited and non-interactive
- Novodex and Havok physics engines



Shading

- Extremely flexible shading environment
- Cg
 - Hand editable shaders
- Completely customizable lighting per object
 - Each shader can selectively reference lights in the scene, adding new lighting as desired
- Custom textures and procedural inputs
 - Color textures can be input to influence any piece of the lighting equation
- Sliders



Lighting

- We support an arbitrary number of point, directional and spot lights
 - Limited only by performance and number of constant shader inputs
- Multiple shadow map lights
- Ambient Occlusion
 - Pre-baked: XSI, Mentalray in Maya, Custom tools



Post Processing

- Specify arbitrary render passes within the NVDemo scene files
 - Render from different cameras with different geometry sets and store results to textures
 - Render subsequent passes which refer to those textures



Maya Exporter

- Automatic shader generation
 - Limited to single phong, lambert, blinn shading nodes
 - No shade trees supported yet
 - Procedural vertex shader generation for BlendShapes, Skinning and combinations thereof
- Shader “hijacking”
 - Use whatever shading you want within Maya
 - Every time you export, the correct Cg shader is put in it's place
- Skinning and Blendshapes defined in Maya

The Making of “Dawn”





Fairy: Intro

- Modeling, texturing and animation done in Maya
- Hair created in Simon Green's custom hair combing tool
- Ambient Occlusion (per-vertex data) from custom raytracing tool
- Motion capture performed by House of Moves
 - Mocap data refined in Maya to add hand, feet, and facial motion, plus tweaking



Animation details

- 50 faces
 - 30 emotion faces (angry, happy, sad...)
 - 20 modifiers (left eyebrow up, right smirk ...)
- Skinning also computed in a vertex shader on the GPU
- Ambient occlusion computed for each blend-shape



Skin shader inputs

- HDR Cubemap environment map
 - Multiple exposure shots in a forest with an IPIX camera
 - Convolution operation computed Diffuse and Specular Cubemaps
- Fragment Shader texture inputs:
 - Normalization Cubemap (Procedural, indexed by any vector)
 - Diffuse Lighting Cubemap (HDRShop, indexed by normal)
 - Specular Lighting Cubemap (HDRShop, indexed by reflection)
 - Hilight Lighting Cubemap (Indexed by world eye direction)
 - Colormap/Specular (Texcoord, rgb = color, a = “front” specular)
 - Bumpmap/Specular (Texcoord, rgb = bump, a = “side” specular)
 - BloodColorMap (Texcoord, rgb = blood color)
 - BloodTransmissionMap (Texcoord)



Ambient occlusion

- custom ray-tracing tool (baked once offline)
 - casts 2048 rays from each of the 180,000 vertices in Dawn's mesh in a 180 degree hemisphere
 - Store the fraction of rays the are un-occluded by Dawn's own geometry. (value from 0.0 to 1.0)
 - This occlusion value weights the cube-map lighting
 - A uniform distribution was less noisy than the stochastic method
 - Also computed for each blend-shape pose and the values were blended between the rest pose and the blend-shape values (on the GPU)
 - Helps a lot with facial features: nose, ears and mouth.
 - Subtle effect on the rest of the body: tend not to notice when it doesn't change with arm and leg animations.



Skin Shader

- Like anything, diddle the knobs until it's pretty...
- Our fairy shader ended up as:

`worldNormal = TangentToWorldMatrix * BumpMap`

`diffuseLight = DiffuseLightCube(worldNormal)`

`specularLight = SpecularLightCube(ComputeReflection(worldEyeDir, worldNormal))`

`passThruLight = HilightCube(worldEyeDir)`

`bloodAmount = dot (BloodTransmissionMap, BloodTransmissionTerms)`

`diffuseColor = lerp(ColorMap, BloodColorMap, bloodAmount)`

`specularColor = lerp(frontSpecularMap, sideSpecularMap, BloodTransmissionVector.z)`

`return (occlusion*(diffuseLight *diffuseColor + specularLight *specularColor + passThruLight))`

Fire in the “Vulcan” Demo





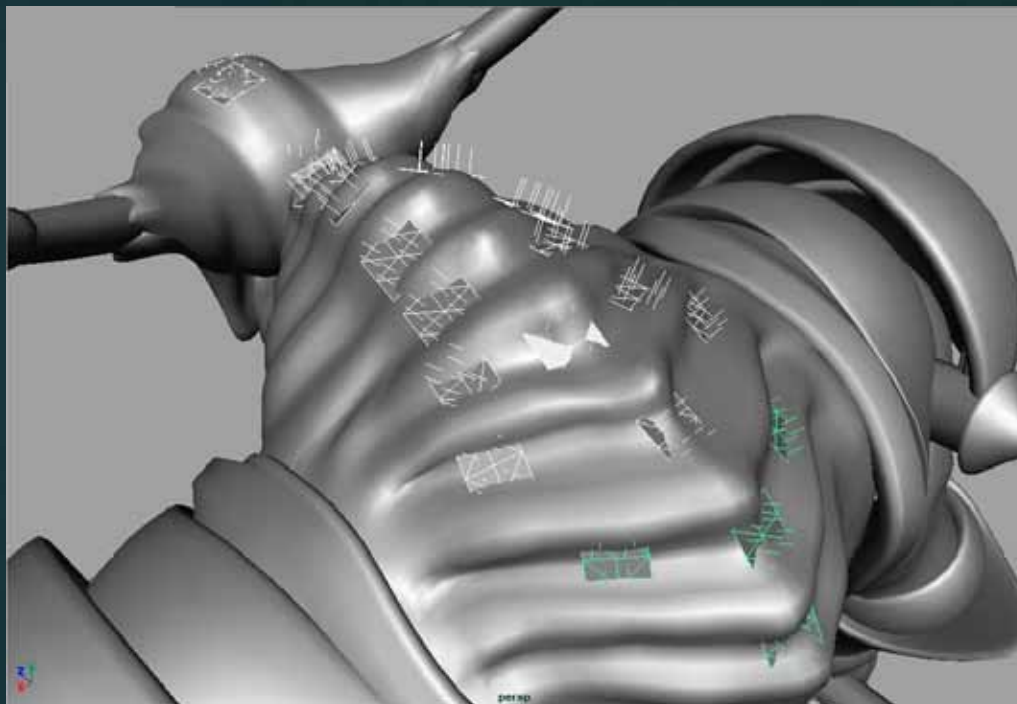
Creating realistic flames

- Considered two procedural methods
- Fully procedural:
 - required little memory
 - required thousands of particles
 - heavy load on the CPU and GPU
- 2D image warping
 - Warp a fixed flame image to create motion
 - hard to integrate with arbitrary free camera motion
 - could not integrate smoke effectively
- Neither procedural technique was ideal
 - Solution: video sprites



Flame motion

- The video sprites were bound to particles
- Emitters placed on the Vulcan's back and arms
 - could specify velocity and density of particles for each emitter



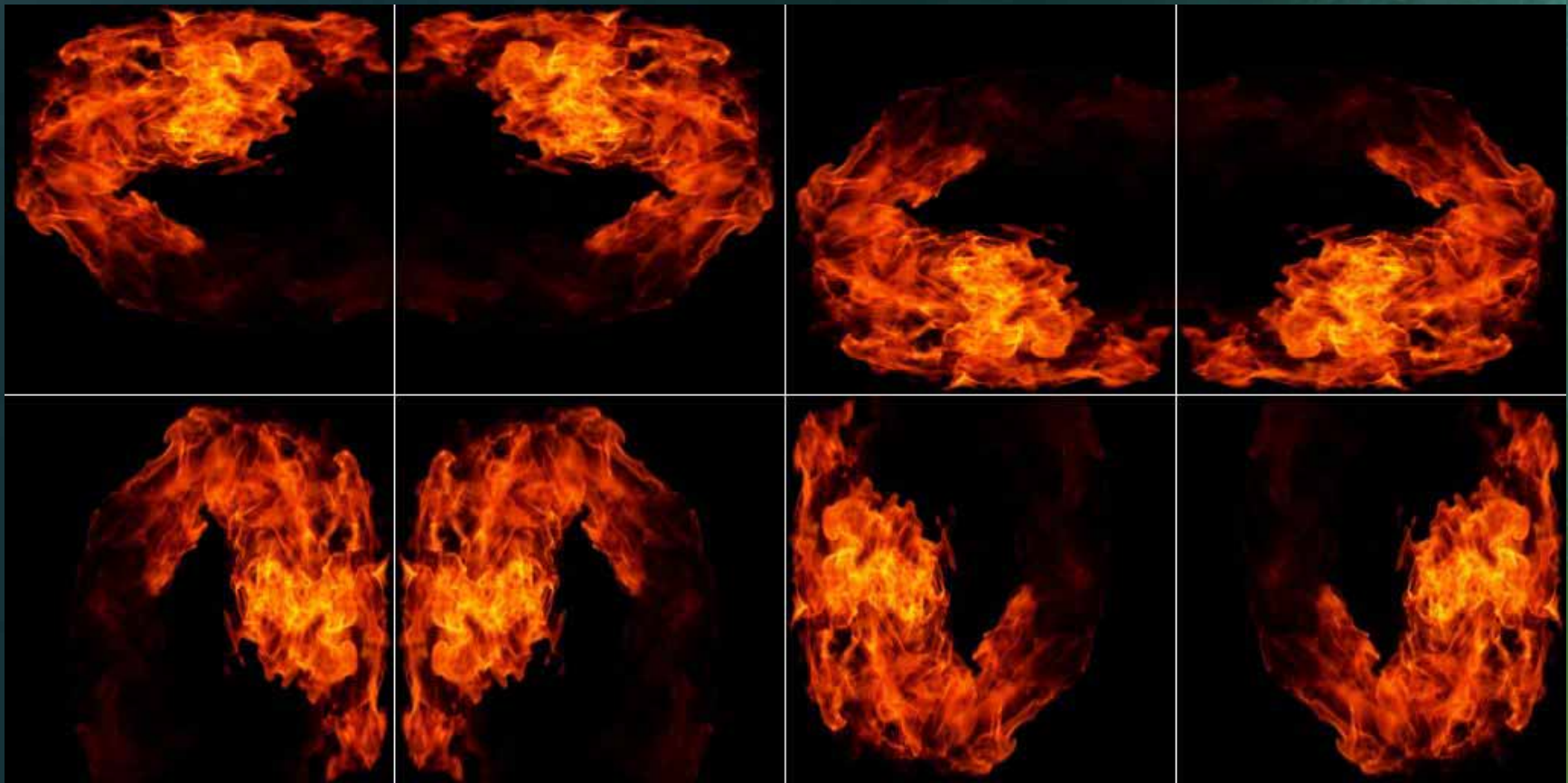
Smoke



- Particle system based – static sprites
- rolling motion on the x axis to give appearance of self-folding



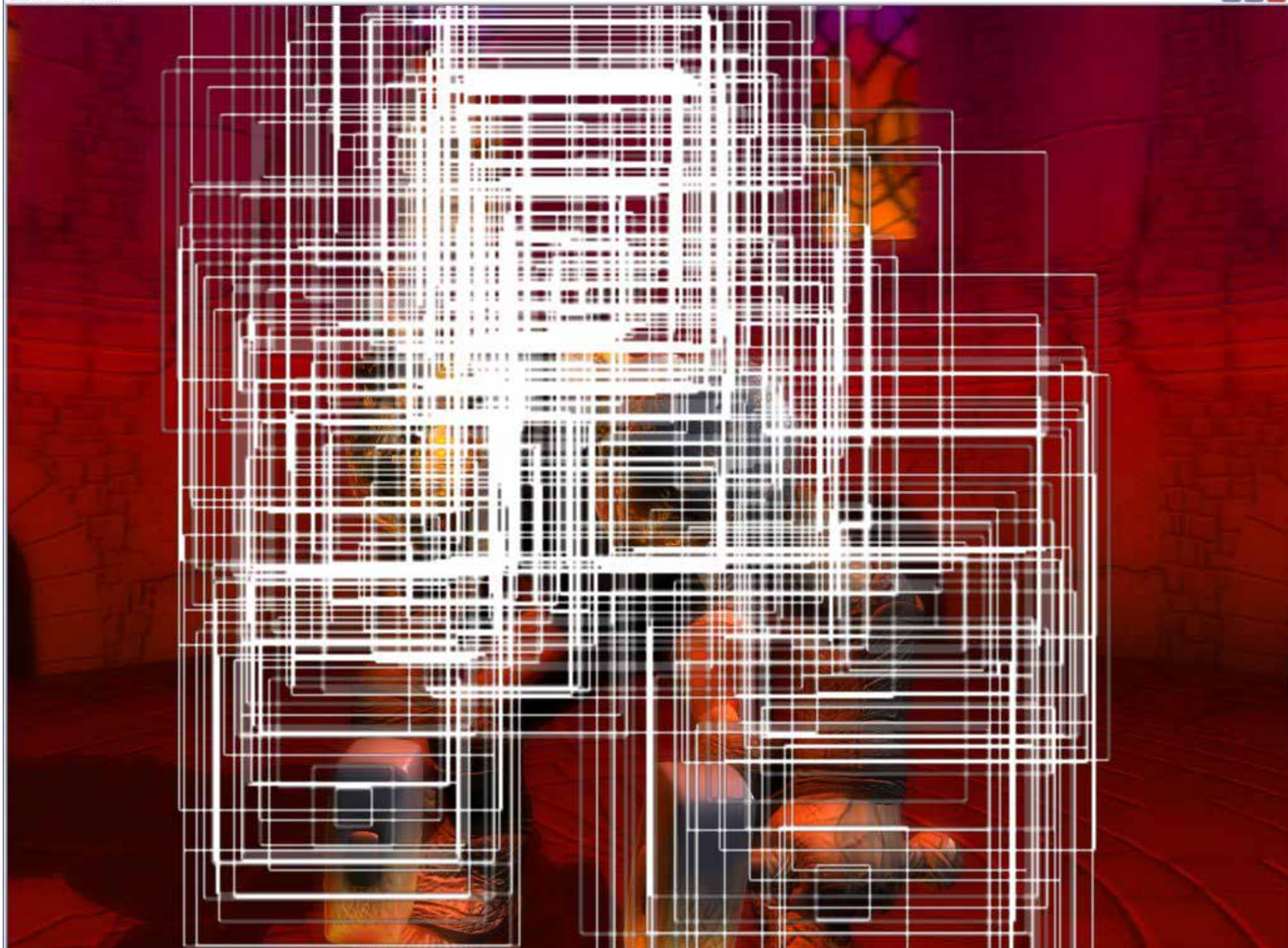
Flame Variety – the cheap way





Particle motion

- Required a fair bit of attention to get looking good
- All the detail is in the video. Used only a few hundred particles
- Bind particles to their emitters
 - if the emitter moves, the particle moves with it, but it's influence fades with time
- Too few particles to notice the effects of fluid dynamics





The Making of “Nalu”

Hubert Nguyen
Will Donnelly
NVIDIA Corporation

Long, Blonde Hair Rendering





Long, Blonde Hair Rendering

- **Long**

- Requires dynamic animation
 - Thus cannot bake lighting
- Requires lots of it
 - Thus shading has to be fast

- **Blonde**

- Three visible highlights, black only has one
- Shadows much more visible



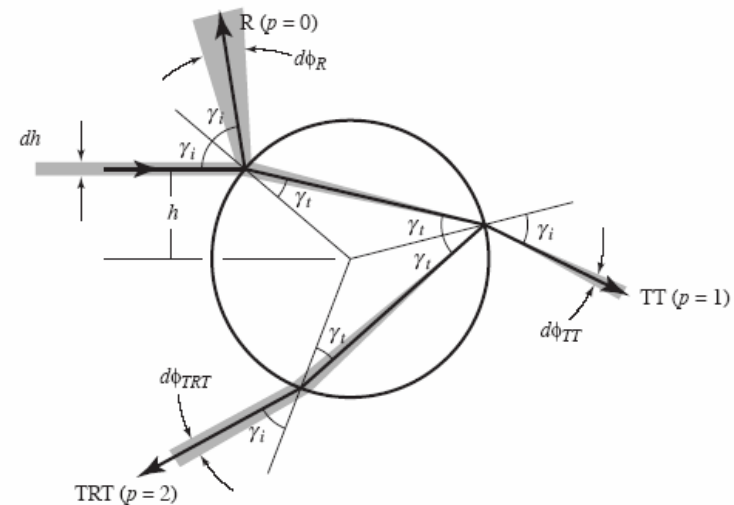
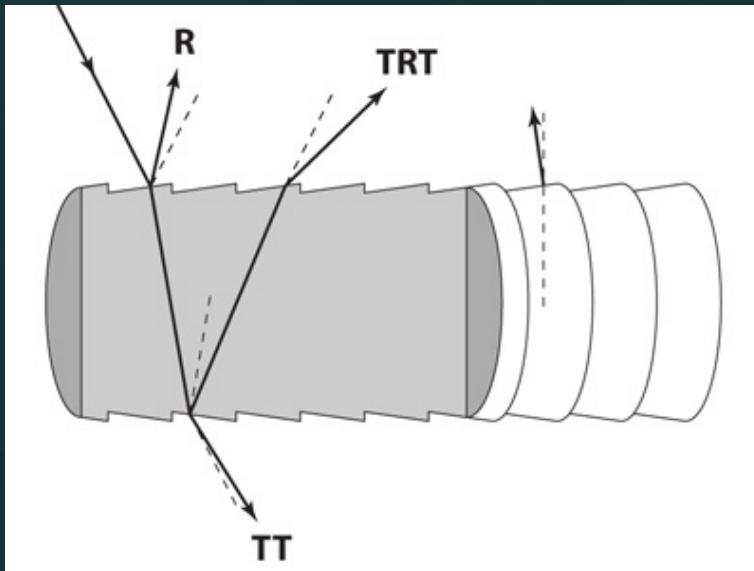
Acknowledgements

- “Light Scattering from Human Hair Fibers”
- By Steve Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan
- SIGGRAPH 2003

Paper Models three Distinct Highlights



- Consider only 3 most significant terms
 - R, TT, TRT



- Uses path notation
 - R is reflection
 - T is transmission

TT Highlight



- TT – strong forward scattering component
 - Important for underwater hair





Why Opacity Shadow Maps?

- **Opacity Shadow Maps vs Shadow Maps**

“What percentage of light is blocked from here?”

vs.

“Is the light blocked from here?”

- **Thus supports AA edges and volumetric rendering**

- **Regular shadow maps alias around edges**

- Hair is 100% edges!

Results from Kim & Neumann



No Shadows



15 slices



OSM Creation

- **Render hairs to 16 slices**
 - Original implementation : 16 render passes (RP)
- **Can use lower hair LOD**



... up to 16

In the demo...



Hair **WITH**~~OUT~~ Shadows



Hair **WITH** Shadows



Shafts of Light : “God Rays”



Shafts Are Based on a Radial Blur Effect



Radial Blur



“God Rays” in the Demo



Hair Geometry & Dynamics





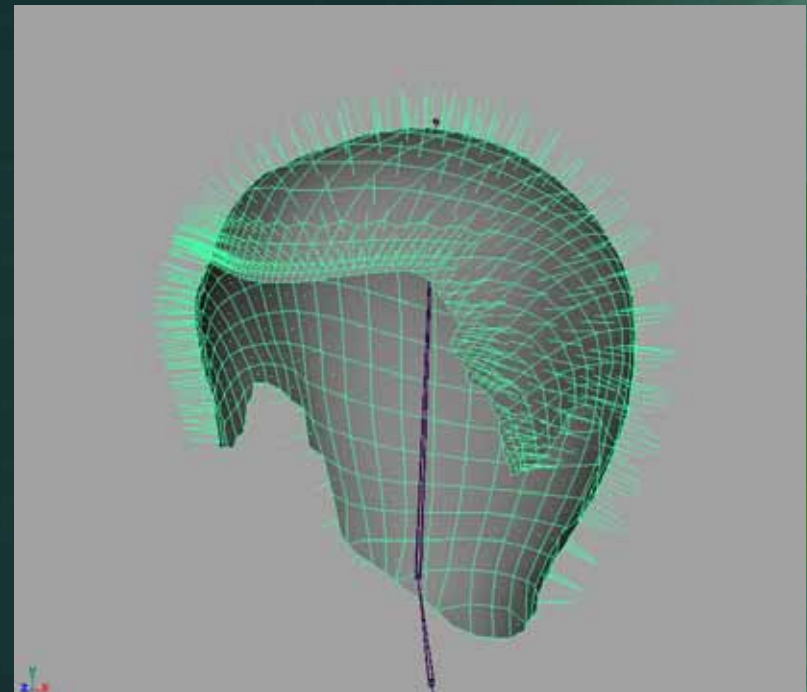
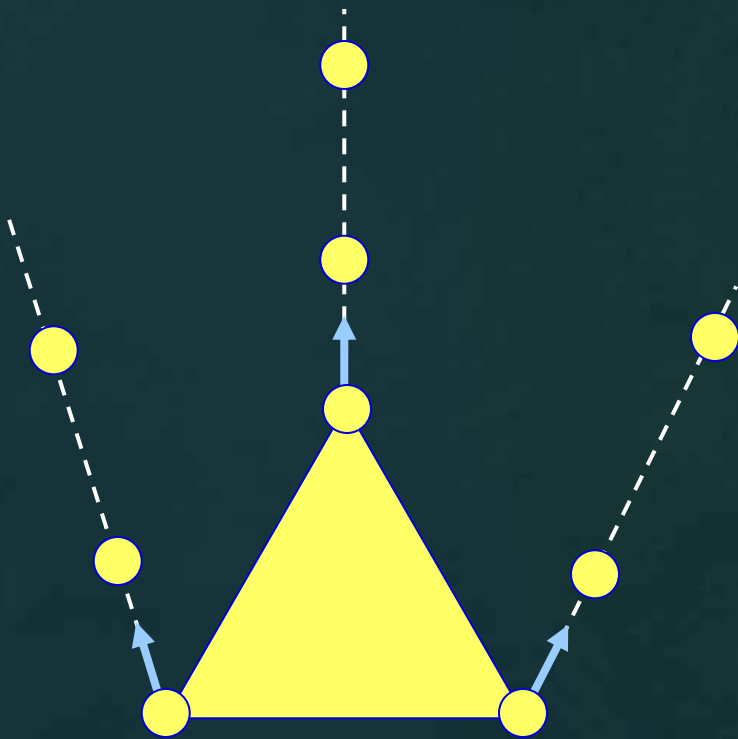
Hair Geometry: Overview

- 4095 individual hairs driven by 762 “control hairs”
- “Control hairs”
 - Set of hairs that is really driven by dynamics/collisions
 - Based on a particle system, where particles are connected by distance constraints.
 - Grown from a reference geometry
- “Fine hair” geometry is created by smoothing & interpolating the “control hair”.



Hair Geometry: Layout & Growth

- “Control hair” grows from a dedicated geometry



Hair Geometry: Control Hairs (left image)



- Physics/dynamics/collisions are performed on the control hairs

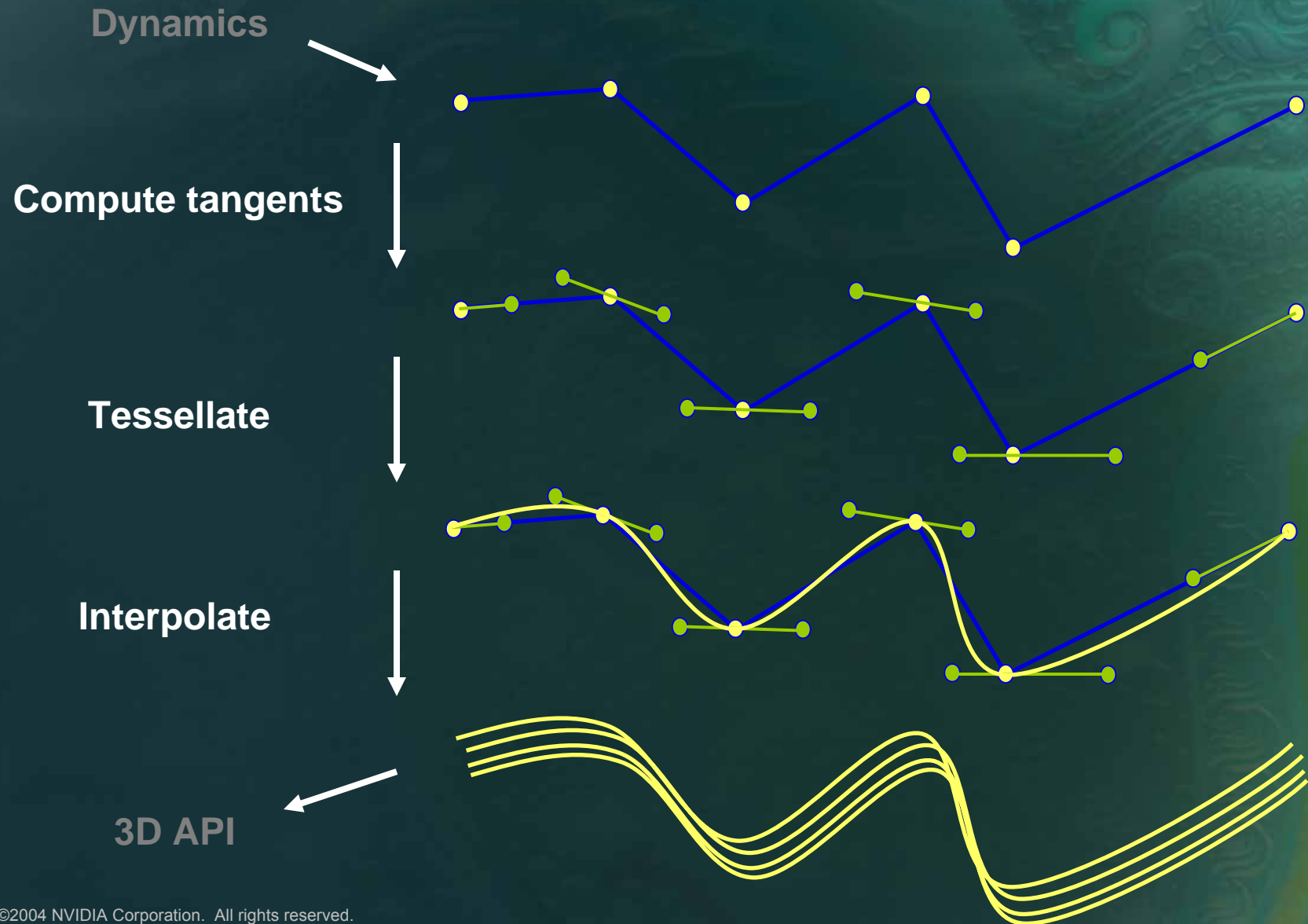
Control Hair



Fine Hair



Hair Geometry : Lifecycle (per frame)





Hair Dynamics

- Based on a particle system
- Uses the “Verlet” integration
 - previous frame position to compute velocity

$$\mathbf{x}' = 2\mathbf{x} - \mathbf{x}^* + \mathbf{a} \cdot \Delta t^2$$

$$\mathbf{x}^* = \mathbf{x}$$

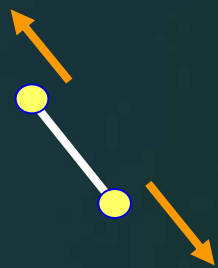
Reference: “**Advanced Character Physics**”

Thomas Jakobsen, IO Interactive, Denmark.

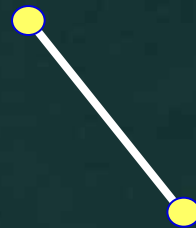


Hair Dynamics: Constraints

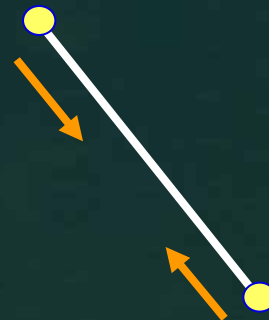
- Two constraint types:
 - **Infinite mass**: applied to the “hair root” particles. Allows the head to “pull” the hair.
 - **Distance constraints**: forces “control hair” segments length to stay constant



Too short,
expand



Desired
length



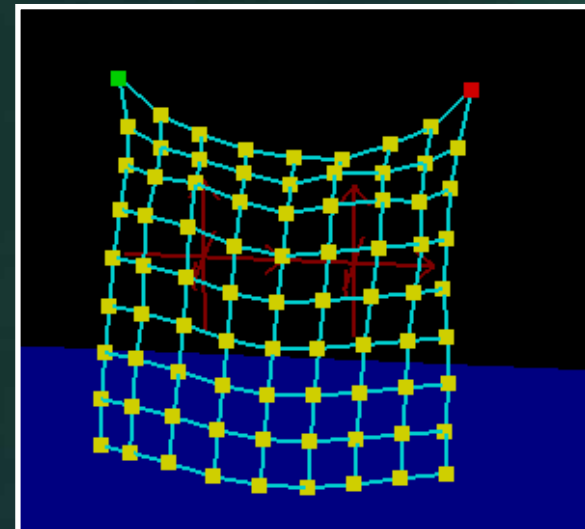
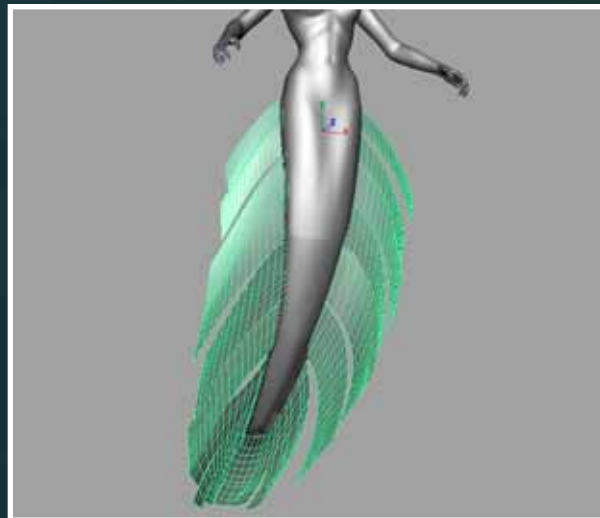
Too long,
contract

- If we apply those constraints iteratively, the particles will globally converge to the desired solution



Fins

- Fins are a cloth simulation.
- Any mesh can be turned into a cloth by using triangle edges as constraints

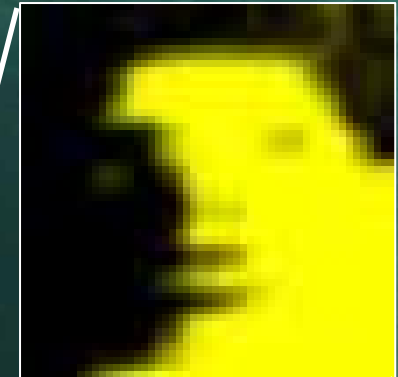


Soft Shadows



- Based on
“Texture Space Diffusion” see “GPU Gems”:
 - Do the regular shadow mapping computations
 - But render in **Texture Space**
 - Using the UV coordinates as Vertex Shader Output Position
 - Blur the Texture Space B&W shadow result
 - Use the blurred shadow result in place of shadow compare when rendering the character

Soft Shadows: Visualizations



UV Space rendering



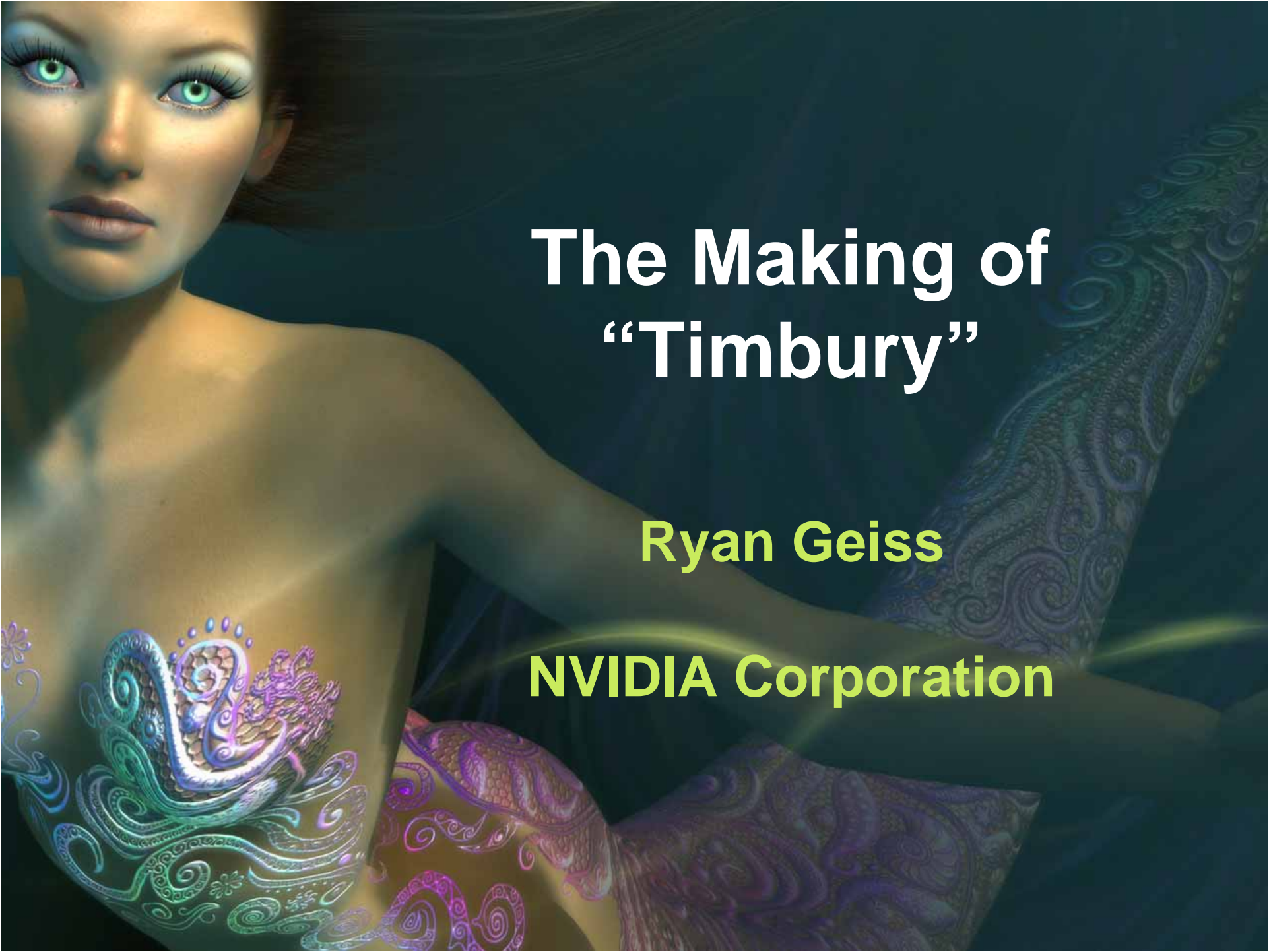
Soft Shadows: Challenges

- Unfold character in UV Space
- Visible Seams where UV are not continuous



Ottakes





The Making of “Timbury”

Ryan Geiss

NVIDIA Corporation



Full Name: Timburly Entonin Mudgett
Born: Cleveland, England, 1896
Profession: Entomologist
His deal: He's off to do "science."



Technologies & Effects Used

- High Dynamic Range (HDR) lighting
 - allows very high-precision lighting computations
 - Automatic Gain Control: camera responds to bright light
 - look at bright light -> normal objects become silhouettes
 - “regular” white tones (like a shirt) actually darken, but saturated whites (like the sun) *stay bright*
- Environmental lighting with fp16 cubemaps
- Post-processing “softening” of the image
- Animation: Skinning & Blendshapes
- Adaptive Subdivision Surfaces
- “Soft” (multi-tap) shadows
- Refractive Eyeglasses



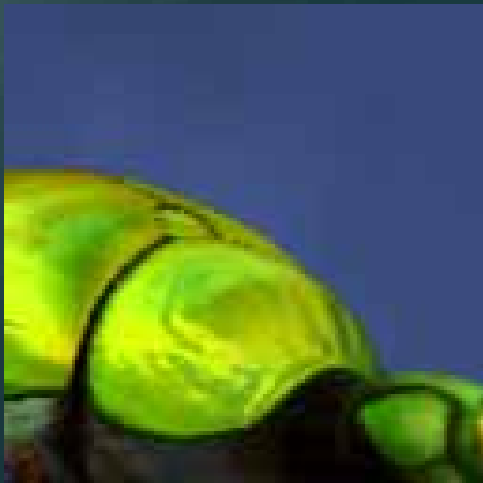
The Rendering of a Frame

- Render scene to a texture
- Analyze how bright it was
- Make a copy and blur it profusely
- Mix the crisp and blurred together (to soften the image);
- Darken it (based on light analysis);
- Draw final image to the screen.

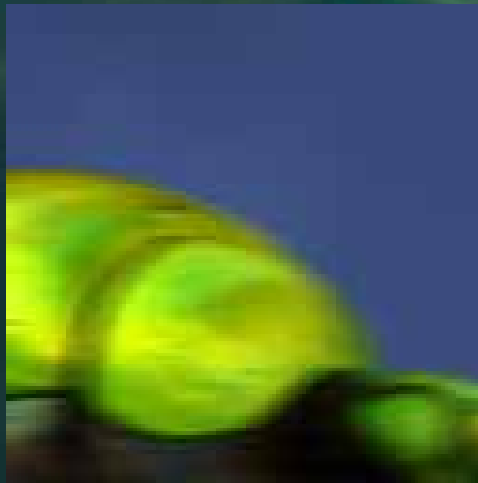


Rendering (More Detail)

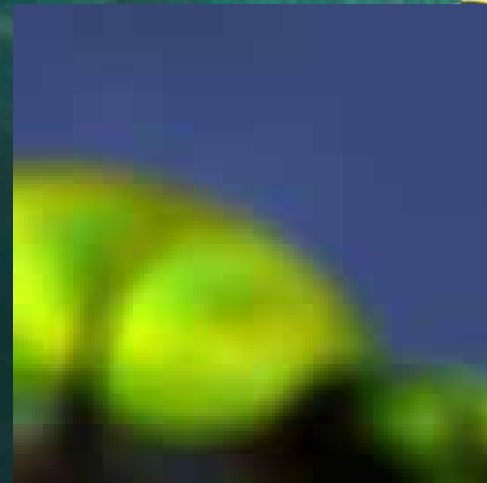
- First, render entire scene to a big fp16 texture.
- Render that to a 256x128 texture, sampling 3x3 source texels per (destination) pixel.
- Four more passes, all on 256x128 fp16 textures:
 - blur on x
 - blur on y
 - blur on x
 - blur on y
- (creates a nice near-gaussian blurred image) (but fast!)
- For final render pass, average this blurred image with the original crisp image for a nice, soft, cinematic look.



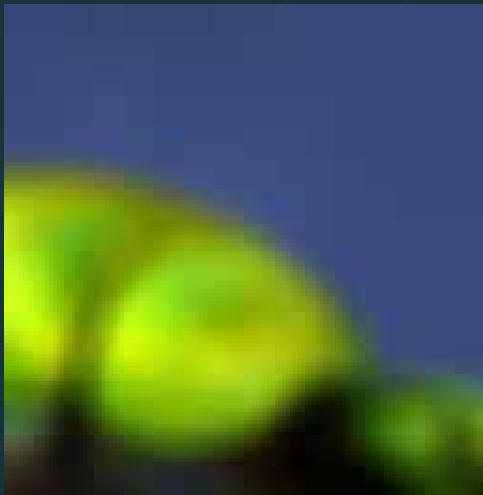
• original



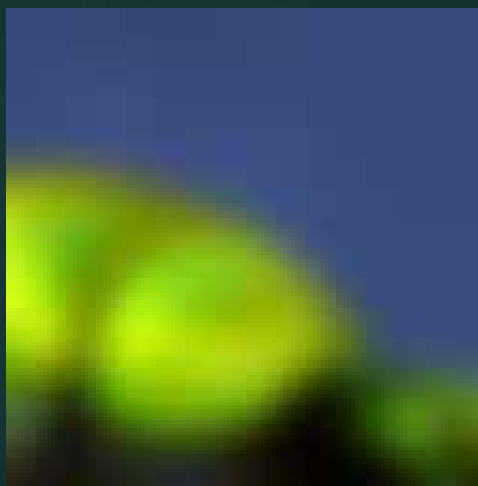
blur/x



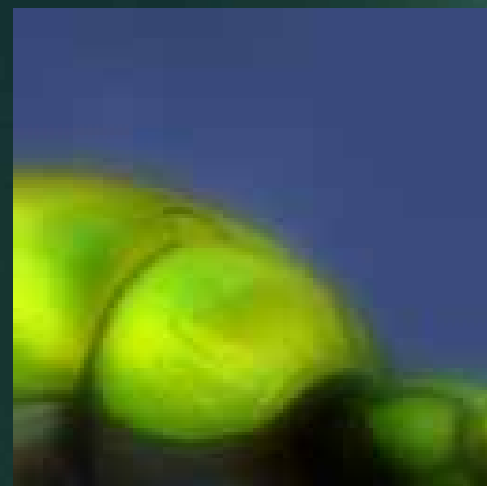
blur/y



• blur/x (2)



blur/y (2)



mixed



Automatic Gain Control (AGC)

- Mimics what a camcorder or human eye does in response to too much light: shrinks the aperture.
- The really bright part of our scene is the sun
 - about 40 times brighter than most colors in scene
- Looking around the scene, you can see the aperture open/close in response to how much light is coming in.
- Look at the sun → aperture shrinks, turning most objects into silhouettes.

AGC Example



regular lighting



looking into the sun

log-sum & tone mapping:



(+) good distrib. of luminances



(-) poor contrast

linear sum &
simple scaling:



(-) poor distrib. of luminances

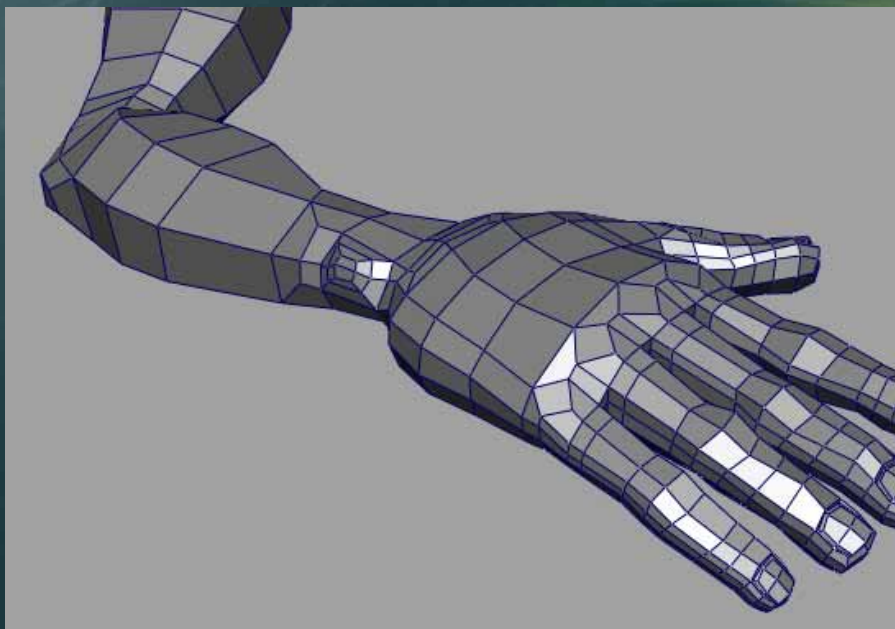


(+) good contrast

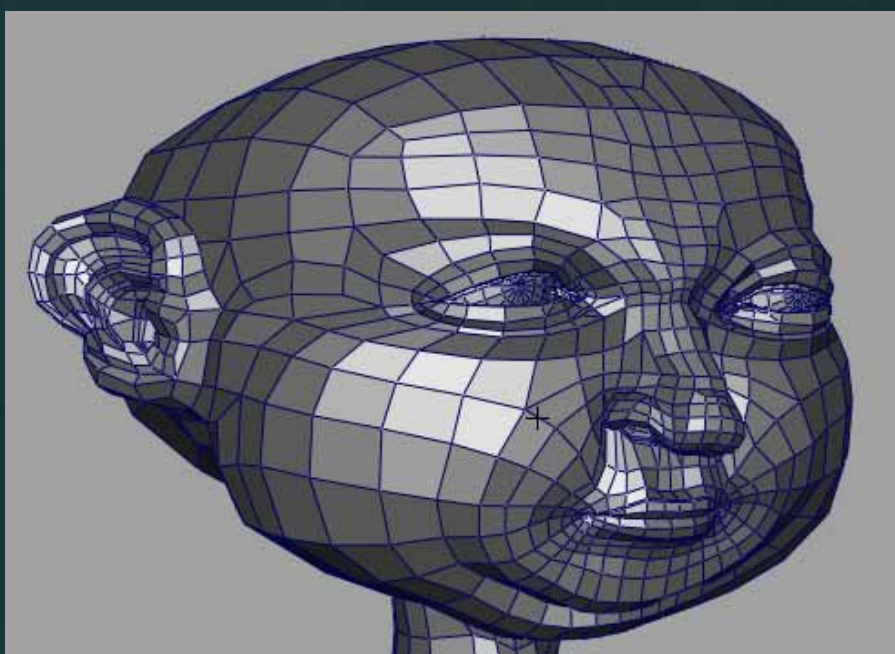


Adaptive Subdivision Surfaces

- Work by Michael Bunnell of Nvidia
- Goal: adaptively subdivide polygons to keep meshes looking good & drawing efficiently.
 - add polygons as you zoom in, as an elbow bends (increasing curvature), etc.
- Original mesh: prefer quads, but triangles work too
 - should be low-resolution, but still just high enough to describe essential features of the model (...see next slide).
- At startup, convert it entirely to Catmull-Clark patches (quads).
- **Goal:** each frame, adaptively subdivide polygons until the screen-space “error” is ≤ 1.0 pixels.



original control meshes for arm, hand



original control mesh for head

Adaptive Subdivision Surfaces (2)



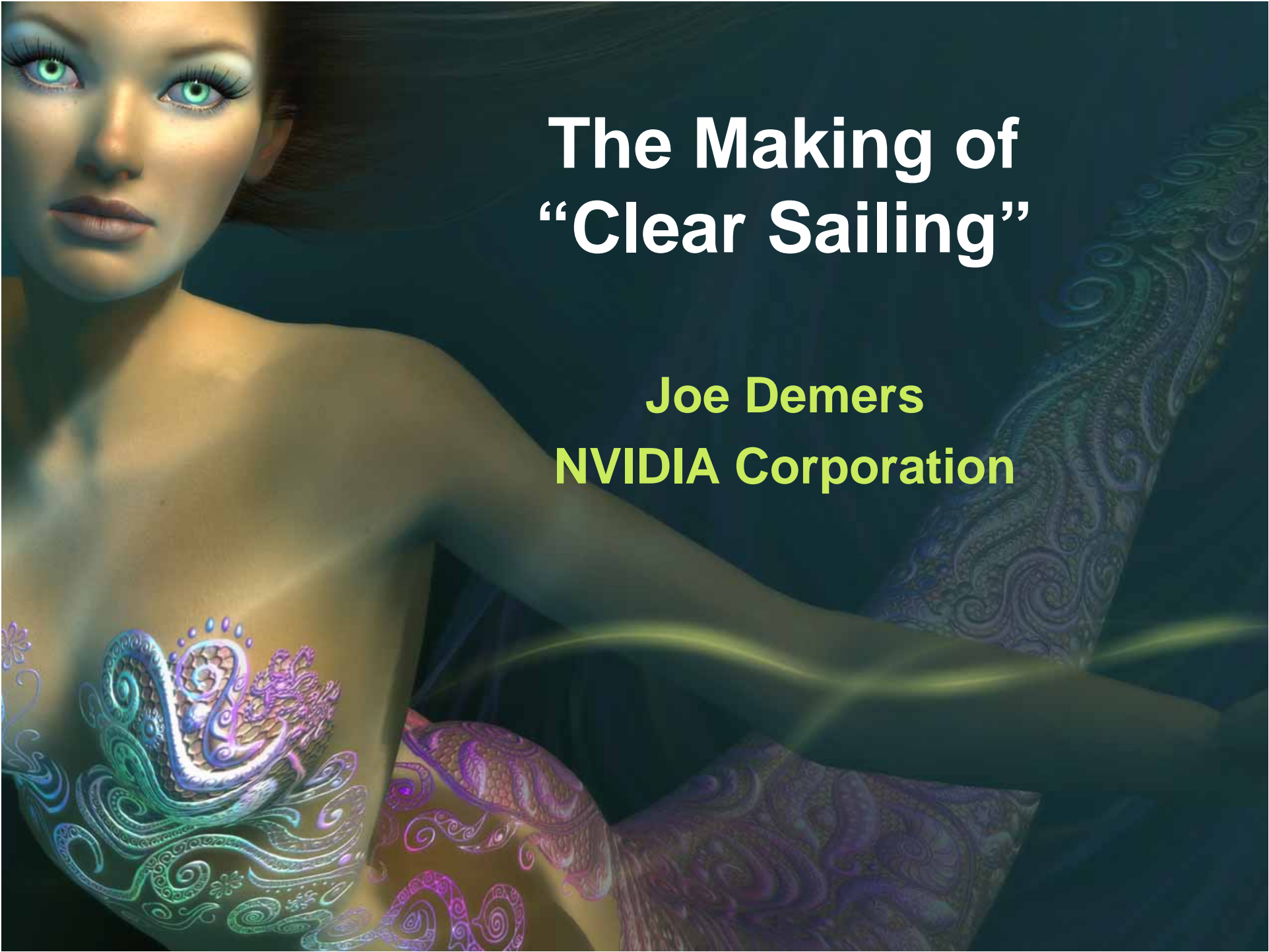
- The “error” is how far the edges are from the ideal, smoothed surface.
 - Specifically, the error is the distance from the center of each edge, to the center of the ideal curved edge, in **world space**.
 - Project that distance value into screen space – this is the error, in pixels.
- If two opposite sides of a quad have high error, tessellate along those edges. likewise for the other two edges.
 - For things like cylinders or arms, produces tessellation in the direction where it’s needed most.

Ottakes



Outtakes





The Making of “Clear Sailing”

Joe Demers
NVIDIA Corporation



Ocean Simulation and Rendering

- Our goal was to create a realistic looking ocean with choppy waves and a ship sailing on it
- This required us to:
 - simulate choppy ocean waves
 - tessellate the ocean surface
 - render ocean water (and foam)
 - calculate physics for the ship from the waves
 - render spray where the ship meets the waves

Ocean Simulation and Rendering





Ocean Simulation

- The two common models in high-end ocean simulation are the Gerstner wave model and FFT-based statistical models
- We chose the Gerstner wave model for its simplicity and non-periodicity
- The Gerstner wave model moves points on the surface of the ocean in circles parallel to the direction of travel of the wave, allowing for 'cusping' as the wave height increases
- We found 45 Gerstner waves on a 150x150 grid gave us the best quality/performance tradeoff for the Clear Sailing demo



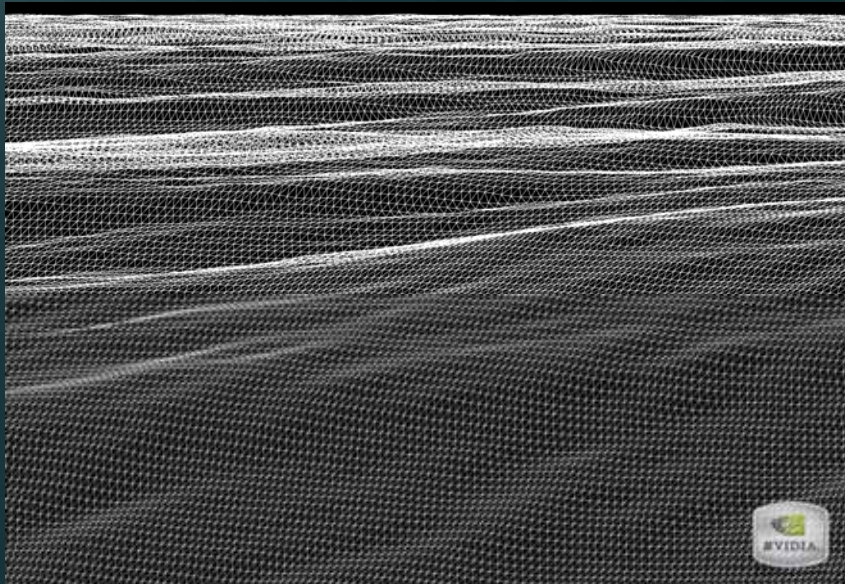
Ocean Tessellation

- Most previous techniques create regular grids of vertices in world space, and either tile the grid, or apply dense fog after the grid, or both
- We tessellated in eye space, mapping a regular grid to the intersection of the ocean plane and the camera viewport
- This allows us to only simulate and render geometry that is seen, and tessellate more finely in the foreground than the background

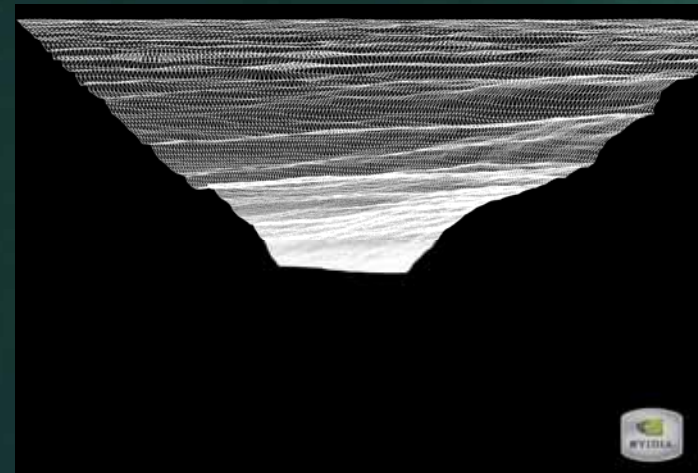
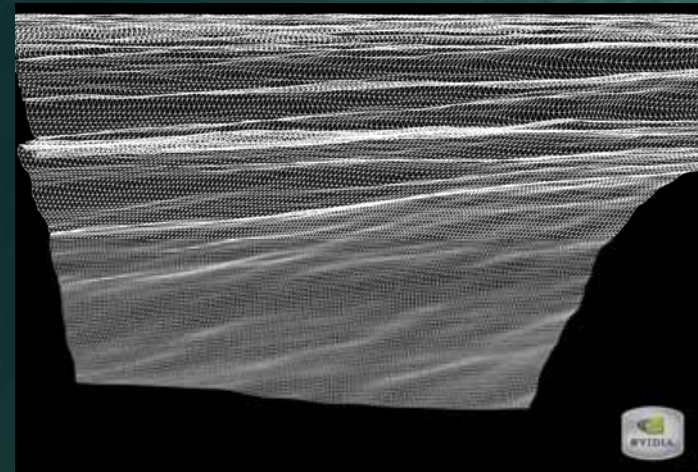


Ocean Tessellation

Screen-filling tessellation



Zoom out to see tessellation



Freezing the geometry and pulling the camera back allows us to see the actual geometry being drawn



Ocean Rendering

- Rendering deep water involves multiple sources of lighting
 - sunlight (directional light) reflected off the surface
 - sky light (cubemap texture light) reflected off the surface
 - scattered light (constant term) from below the water surface
- We can blend between the reflected and scattered terms using a simple fresnel function
- Unfortunately, fresnel exponents that look good along the surface tend to wash out the ocean when looking straight down, so a fairly low exponent works best when all viewing angles are possible



Reflecting the Ship

- The ship occludes reflected light in two ways
 - sunlight is occluded by using z-buffer shadows
 - skylight is occluded by ray casting into a 2d geometry imposter for the ship
- A little per-pixel noise breaks up the reflection and gives the illusion of higher frequency waves

Reflecting the Ship





Rendering Foam

- Foam is effectively a semi-transparent layer above the water surface
- Foam is generated (i.e. the foam layer is made opaque) where waves cusp, and also along the wake lines behind the ship
- We used render to texture with some additive blending to allow the foam to fade off over time

Rendering Foam





Ship Physics

- The ship is thrown about by the waves, but doesn't affect the water
- The ship moves up and down, and pitches and rolls, but doesn't slide along the ocean surface or turn left or right
- Above the water, the ship is affected by gravity and wind
- Below the water, the ship is affected by friction and a buoyancy proportional to the amount of water displaced
- The physics requires a lot of tuning/tweaking to get it to look right, but having tuning sliders also means you can have lots of fun
 - the ship moves like a toy boat if you increase the wave speed and decrease the scale that physics is computed at (effectively scaling the universe)



Ship Lighting

- The ship is lit from 4 light sources
 - direct sunlight, shadowed via z-buffer shadows
 - sky lighting, via diffuse and specular cubemaps
 - ambient light, baked global illumination from the sky
 - reflected light, a directional bluish light from below
- A diffuse color map gives the ship color
- Bump maps and specular maps give the ship more detail and give the appearance of more geometric complexity than there really is



Ropes & Sails

- The sails are two-sided, and softly glow when the sun is behind them
- It's very important for the facing-the-sun and away-from-sun shaders match when the normal is perpendicular to the sun, otherwise you'll see seams
- Ropes are drawn using lines, which change thickness depending upon distance from the viewer
- You can get a nice antialiasing effect by fading their transparency when the line thickness falls below 1 pixel
- Using alpha-to-coverage means you don't even need to sort!

Ropes & Sails





Splashes and Spray

- Splashes

- emit particles when cannonballs hit water
- render large particles with a texture of many small droplets

- Spray

- when the ribs (or keel) of the ship move down through the water surface, emit particles between those ribs
- faster motion generates more spray thrown further (higher)

Splashes and Spray





Smoke and Splinters

● Smoke

- when the ship's cannons fire, emit fairly large particles along the path of fire with animated textures (using a 3d texture)
- smoke particles start moving very quickly, but then dampen their motion almost immediately, and slowly grow and fade
- smoke particles "lit" by darkening lower half - cheesy, but looks good

● Splinters

- when cannonballs hit the ship, generate lots of little triangles
- move and tumble them with simple (but fast) verlet dynamics

Smoke and Splinters





Post Processing

- HDR-style glow
 - using a simple 8-bit single channel hdr effect
 - render overbrightness into the alpha channel
 - blur the alpha and add it back into the scene
- A little fog helps integrate the water, sky and boat together into the scene

Post Processing



Questions?

