The Making of Ruby: The Double Cross

David Gosselin
ATI Programming Lead
Outline

• Ruby Demo
• Goals
• Planning
• Making of from Rhino FX’s perspective
• ATI Art Notes
• Shaders
  – Depth of Field
  – Hair
  – Skin
  – Gem
  – ATI Logo
Ruby Demo

The Making of Ruby: The Double Cross
Demo Goals

• Showcase the Radeon X800
• Focus on Game Developers
• Focus on Game Players
Showcase the RADEON X800

- Over twice the performance of 9800XT
  - More Shaders
  - More image space/full screen effects
  - More off-screen effects

- Long shaders up to 512 instructions
  - More complex shaders
  - Fewer passes
  - Less shader rework

- Normal MapCompression (3DC)
  - Bigger/more Normal maps
Focus on Game Developers

• Better Characters
  – Skin
  – Hair
  – Motion Capture

• More Photo-Realism
  – Depth of Field
  – Glows and Light Blooms
Focus on Game Players

• Look like a next generation game ... not a tech demo

• Establish an emotional connection with the audience

• Realistic Characters

• Entertaining
Planning: Limits we gave Rhino

• Polygon Budget
  – Ruby: 75,000
  – Optico: 50,000
  – Ninja: 25,000
  – Environment: 100,000
  – Props: 50,000

• Lighting Limits
  – 3 Dynamic lights per shot (1 shadow casting)
  – Lightmaps used for set
More limits

• Animation Limits
  – 35 total blend shapes
  – 5 simultaneous blend shapes
  – 4 weighted bones per vertex
  – Number of on-screen characters limited to 4 at once
Making of from Rhino’s Perspective
Shaders

- Depth of Field
- Hair
- Skin
- Gem
- ATI Logo
Depth of Field

• Fundamental aspect of photorealistic rendering
  – all photos contain some depth of field

• An important part of the creative and storytelling process
  – allows the viewers eye’s to be directed to an area of the scene
What is Depth of Field?

• Computer Graphics traditionally uses the pinhole camera model
  – This results in perfectly sharp images

• Real Cameras use lenses with variable aperture sizes
  – This causes depth-of-field
  – Out-of-focus objects appear blurry
Depth Of Field
Depth Of Field
Depth of Field: Implementation

- Render normalized camera depth into destination alpha
- Compute a down-sampled pre-blurred image
- Create composite image
  - Compute “blurriness” from destination alpha
  - Use blurriness to compute kernel size
  - LERP between sharp image and pre-blurred image at each sample based on blurriness
  - Fade samples based on relative blurriness to prevent “leaking”
Populating Destination Alpha

- The post-processing shader needs blurriness and relative depth of each pixel
- We pass the camera distance of three planes to scene shaders:
  - Focal plane: Points on this plane are in focus
  - Near plane: Everything closer than this is fully blurred
  - Far plane: Everything beyond the far plane is fully blurred
- Each object’s pixel shader renders depth and blurriness information into destination alpha
Destination Alpha Example

3m focal plane

6m focal plane

12m focal plane

This is where the focal plane intersects with the floor
Pre-blurring the Image

MSAA image from back buffer
( Destination alpha contains blurriness)

1/16th Size (box filter)
3x3 Gaussian Blur
Pre-Blurred Image
Mapping Depth to Blurriness

- Map a point’s camera depth to [-1, 1] range as shown in pink graph
  - This gives us relative depth
- Scale and bias relative depth into [0, 1] range before writing to destination alpha
  - Saves us from writing blurriness and depth into two separate channels
- To get blurriness, just take the absolute value
Circle Of Confusion Filter Kernel

- Stochastic sampling
- Poisson distribution
- Samples stored as 2D offsets from center

- Center Sample
- Outer Samples

Small Blur

Large Blur
Filter Kernel For Circle Of Confusion

- Vary kernel size based on the “blurriness”
- Sample all taps from original and pre-blurred image
  - Blend between them based on tap blurriness
Reduction Of “Leaking”

- Conventional post-processing blur techniques cause “leaking” of sharp foreground objects onto blurry backgrounds
- Depth compare the samples and discard ones that can contribute to background “leaking”
  - Test each sample against center sample
  - If sample is “sharper” then weight sample by its blurriness
Leaking Example

Background Leaking

With Leak Prevention
Depth of Field Demo
DOF Wrap Up

- Allowed more creative control
- Focused viewers attention
- More photo-realistic
- Blend multiple samples from sharp and pre-blurred image
Hair Rendering

- Realistic hair is a key part of creating believable characters
- Hair Rendering is hard
  - There is a lot of it – around 150K strands on a human head
  - Many Different Hair Styles
  - Hair reacts to light in a complex manner
  - Hair is semi-transparent
- ~25% of the total render time of “Final Fantasy - The Spirits Within” was spent on the main character’s hair
Real-Time Hair Rendering

• Our approach also use the Kajiya-Kay shader model.

• Along with some observations derived from Marshner (Siggraph 2003):
  – Colored hair has 2 highlights
  – Primary (specular) shifted towards hair tip.
  – Secondary (colored) shifted towards hair root.
  – Secondary highlight also sparkles

• Ambient Occlusion is used for self-shadowing
Hair Model Authoring

- Several layers of patches to approximate volumetric qualities of hair
- Ambient occlusion to approximate self-shadowing
  - Per vertex
- Why Polygons
  - Lower geometric complexity than line rendering
  - Makes depth sorting faster
  - Integrates well into our art pipeline
Hair Lighting: Kajiya-Kay Model

- Anisotropic strand lighting model
- Use hair strand tangent (T) instead of normal (N) in lighting equations
- Assumes hair normal to lie in plane spanned by T and view vector (V)
Hair Lighting: Marschner Model

- Based on measurements of hair scattering properties
- Observations
  - Primary specular highlight shifted towards hair tip
  - Secondary specular highlight
    - colored
    - shifted towards hair root
  - Sparkling appearance of secondary highlight
- Math is complex, we’re just trying to match these observations phenomenologically
Shifting Specular Highlights

• To shift the specular highlight along the length of the hair, we nudge the tangent along the direction of the normal.

• Assuming T is pointing from root to tip:
  – Positive nudge moves highlight towards root
  – Negative nudge moves highlight towards tip

• We also add a noise component to avoid too much uniformity
Approximate Depth Sorting

- Need to draw in back-to-front order for correct alpha-blending
- For a head with hair this is very similar to inside to outside
- Use static index buffer with inside to outside draw order, computed at preprocess time
  - Sort connected components (hair strand patches) instead of individual triangles
Hair Demo
Hair Wrap Up

• Polygonal Model for ease of integration

• Kajia-Kay lighting plus Marshner observations

• Worked for our constraints

• Check out “Practical Real-Time Hair Rendering and Shading” Thursday in GPU2 sketch session
Skin Rendering

• Skin Rendering is hard
  – Most lighting comes from sub-surface scattering
  – Pigment color mainly from epidermis
  – Pink/red color mainly from blood in dermis

• Traditional Lambertian lighting model is designed for ‘hard’ surfaces with no sub-surface scattering
  – so it doesn’t work well for skin
Texture Space Subsurface Scattering

- From **Realistic Human Face Rendering for “The Matrix Reloaded” @ SIGGRAPH 2003**:
  
  - From *Matrix: Reloaded* sketch
  
  - Our results:
  
  
  Current skin in Real Time

The Making of Ruby: The Double Cross
Basis for Our Approach

- SIGGRAPH 2003 sketch *Realistic Human Face Rendering for “The Matrix Reloaded”* by George Borshukov and J. P. Lewis

- Rendered a 2D light map

- Simulate subsurface diffusion in image domain (different for each color component)

- Used traditional ray tracing for areas where light can pass all the way through (e.g. ears)

- Also capture fine detail normal maps and albedo maps
Texture Space Lighting for Real Time

- Render diffuse lighting into an off-screen texture using texture coordinates as position
- Blur the off-screen diffuse lighting
- Read the texture back and add specular lighting in subsequent pass
- We only used bump map for the specular lighting pass
Basic Approach

Geometry

Light in Texture Space

Blur

Sample texture space light

Back Buffer

The Making of Ruby: The Double Cross
Texture Coordinates as Position

• Need to light as a 3D model but draw into texture

• By passing texture coordinates as “position” the rasterizer does the unwrap

• Compute light vectors based on 3D position and interpolate
Standard Lighting Model
Texture Space Lighting Model

The Making of Ruby: The Double Cross
Spatially Varying Blur

• Used to simulate the subsurface component of skin lighting

• Used a grow-able Poisson disc filter

• Read the kernel size from a texture

• Allows varying the subsurface effect
  – Higher for places like ears/nose
  – Lower for places like cheeks
Shadows

• Used shadow maps
  – Apply shadows during texture lighting
  – Get “free” blur
    • Soft shadows
    • Simulates subsurface interaction
    • Lower precision/size requirements
    • Reduces artifacts

• Only doing shadows from one key light
Shadow Maps

- Create projection matrix to generate map from the light’s point of view
- Used bounding sphere of head to ensure texture space is used efficiently
- Write depth from light into off-screen texture
- Test depth values in pixel shader
Specular

- Use bump map for specular lighting
- Per-pixel exponent
- Need to shadow specular
  - Hard to blur shadow map directly
  - Expensive to do yet another blur pass for shadows
  - Modulate specular from shadowing light by luminance of texture space light
  - Darkens specular in shadowed areas but preserves lighting in unshadowed areas
- Shadow only dims one light (2 other un-shadowed)
Skin Rendering Demo
Skin Wrap Up

- Texture based lighting gives the Illusion of sub-surface scattering
- Practical on today’s hardware
- Check out “Real-Time Skin Rendering on Graphics Hardware” in Thursday’s GPU2 Sketch session
Rendering A Diamond

• Diamonds in the real-world have complex lighting
  – Reflection – internal & external
  – Refraction – multiple bounces
  – Color Shifts
  – Bright Highlights
Rendering A Diamond

• Reflection can be simulated effectively with cubemaps.
• Refraction is a bit harder
  – A single bounce can be calculated correctly
  – Multiple bounces require raytracing
  – not feasible on current hardware in real-time
  – So we fake it
Basic Algorithm

- Draw back face refractions to the back buffer
- Additively blend on top of back face refractions:
  - Front Face Refractions
  - Front Face Reflections (Environment Cube Map)
  - Front Face Specular Lighting
- Draw sparkles based on Illumination
Faking Refractions

- Look up into a refraction cubemap
- Use multiple refraction vectors
  - Straight up refraction vector
  - Refraction with different IR, then reflected by a vector random to each face
    - To prevent sampling close to first refraction ray
- Use multiple normals (lerp between smooth and face for more variation)
- Can also add an “edge” map to give even more hard edges (more visual complexity)
Creating a Refraction Cubemap

- Rendered with Maya
- Camera inside of gem looking out
- Lighting environment approximated by an environment map
Refractions

Back Face Refractions + (Edge) = Into Back Buffer

Front Face Refractions + (Edge) = Additive Blend With Back Buffer
Combined Refractions
Reflections

• Reflection cube map lookup

• To fake dispersion:
  – “Rainbow” cubemap lookup
  – Modulate rainbow sample with reflection sample

• Lerp between modulated and original reflection sample to control dispersion strength

• Modulate with Fresnel term

• Add specular highlights
Cube Maps

Blurred Environment Map

Rainbow Map

The Making of Ruby: The Double Cross
Environment Lighting

LERP

Environment Map

Rainbow Map

Fresnel Term

The Making of Ruby: The Double Cross
Final Look

Refractions  Environment Lighting  Specular Lighting

The Making of Ruby: The Double Cross
Sparkles

- Placed at strategic points on geometry
- Sparkles move rigidly with gem
- Expanded based on their texture coords
  - Screen-aligned
- Faded in based on an off-screen texture luminance at center of sparkle
- Modulate with a noise value to make them flicker a little bit
Flare Geometry

- Only center matters
- "Cloud" works well
- No need to reside only on faces, inside gem works too

Flare Geometry

The Making of Ruby: The Double Cross
Conceptual Flare Process

For each flare
- Look up luminance of its center in off-screen
- If luminance is > threshold
  - Draw (in reality don’t kill)
Gem Wrap Up

• Used multiple cube-map lookups to simulate refractions

• Used various maps to add visual complexity

• Flares based on off-screen buffer

• In the end it’s the look that counts
• Procedurally generated
  – Less texture memory
  – “infinite” resolution
• Store lines/points as shader constants
  – Position
  – Vector to endpoint
  – Vector to endpoint / length^2
• Test inside/outside to color red/white
ATI Logo Segments

• Test point against each line(ellipse)/point
  – Using DOT/Saturate and a few arithmetic instructions
• Use min to determine if it should be red/white
• Use smoothstep to AA the result
Ruby Demo
The Making of Ruby: The Double Cross

**ATI**

**Lead Programmer**
David Gosselin

**Lead Artist**
Eli Turner

**Programming / Shader Programming**
Alex Vlachos
Christopher Oat
Jason L. Mitchell
Natasha Tatarchuk
Pedro Sander
Thorsten Scheuermann

**Additional Engine Programming**
Chris Brennan
John Isidoro

**Production**
Callan McInally
Stephen Smith

**RhinoFx**

**Director:** Harry Dorrington

**Senior Executive Producer/Partner:** Rick Waggonheim

**Senior Executive Producer:** Camille Geier

**Senior Producer:** Karen Bianca

**Story & Concepts:** Harry Dorrington & David Zung

**Storyboarding / Visualization:** David Zung & Ji Yoon

**Lead Animator:** Jeff Guerrero

**Technical Consultant & Animation:** David Barosin

**Animator/Modeler:** Dan Vislocky

**Lead Lighter & Project Lead:** Joe Burrascano

**Texture Artist / Lighter:** Chimin Yang

**Texture Artist / Lighter:** Ido Kalir

**Lighter:** Natalia Senko

**Shader / Texture Artist:** Dylan Maxwell

**Texture Artist:** Martin Boksar

**Modeler:** Paul Liaw

**Modeler:** John Velazquez

**Modeler:** Shin Kull

**3D Artist:** Michael Ware

**Shader / Texture Artist:** Dylan Maxwell

**Technical Animation / Dynamics:** Ji Yoon

**Technical Director:** Jesse Clemens

**Compositor/Graphics:** Guy Atzman

**Editor/Compositor:** Marc Steinberg

**Software Development:** Jim Callahan

**Systems Engineer:** Paul Tsung

**Photoshop Artists / Graphics:** Chris Green

**Mo Cap Production Manager:** Kristen Ames

**Motion Capture Company:** Perspective Studios

**Motion Capture Stunt Actor:** Declan Mulvey

**Motion Capture Stunt Actor:** Andre “Chyna” McCoy

**Motion Capture Stunt Actor:** Casey Eastlick

**Fight Coordinator:** Declan Mulvey

**Casting – Voice and Stunt:** Wendy Litwack Casting

**Voice Talent:** Marlyne Afflack

**Voice Talent:** Chris Phillips

**Music House:** Amber Music New York / LA

**Music Producer:** Kate Gibson

**Music:** Will Richter

**Music Sound Design:** Bill Chesley

**Voice Record / Mix:** Tonic New York

**Sound Engineer:** Jody Nazarro