Ruby: Dangerous Curves
“Effects Breakdown: How’d they do that?”

Dan Ginsburg
3D Application Research Group
Overview

> RhinoFX/ATI relationship
> Ruby shot-by-shot breakdown
  > Glows
  > Motion Blur
  > Reflections
  > Explosions
> Demo engine modifications - Lua scripting
  > Motivation
  > Integration
  > Usage
RhinoFX/ATI Relationship

- RhinoFX
  - Track record in creating “Hollywood” content for movie and TV
  - Created artwork within the budgets and requirements of real-time
  - Provided full pre-rendered cinematic

- ATI
  - Achieve the high visual quality of *Ruby: Dangerous Curves* in real-time
  - Developed custom Maya exporters and pre-processing tools
  - Custom shader development
  - Targeted demo to run on desktop and mobility hardware
Glows
Glows - Motivation

- Bloom caused by light scattering within the eye [Spencer95]

- Affects visibility of nearby objects
- Exaggerates the brightness of light sources
- Improves the perceived dynamic range
- For *Ruby: Dangerous Curves*, implemented as a post-process
Glows – Implementation Overview

> The glow amount is rendered to the alpha channel of framebuffer
> Artists can control the glow amount with either:
> A texture that specifies glow value
> A single glow value for the whole object
> Alpha blended objects with glow are drawn twice:
> First with alpha masked and blend factor in alpha
> Second with color masked and glow amount in alpha
> The framebuffer is copied to an offscreen surface
> Forces a multisample resolve so it can be accessed as a texture
> The offscreen surface is downscaled to a ¼ x ¼ surface
> The glow algorithm iteratively runs a bloom filter over the image
> The final glow image is additively blended with the framebuffer
Glows - Buffers

- Copy of framebuffer
- Downsample color buffer
- Ping/pong blur filter
- Glow Amount in alpha
Glows – Bloom Filter

> Iteratively ping pongs between two $\frac{1}{4} \times \frac{1}{4}$ resolution buffers
> Filter kernel expands on each iteration [Kawase03]

Ping Buffer
> After $n$ iterations, blend the final iteration with the framebuffer on a full-screen quad (in Ruby, $n = 6$)
Motion Blur
Motion Blur - Motivation

- Simulate the effect of the scene moving during camera exposure
Motion Blur – Implementation Overview

> Render tunnel to back buffer
> Copy back buffer to offscreen surface
> Draw all other objects to back buffer
> Draw tunnel again with depth test set to LEQUAL
  > Per-pixel, calculate a screen-space velocity vector using the current and previous frame’s transform
  > Take “n” samples from offscreen surface along the velocity vector
Motion Blur – Methodology Discussion

> Advantages:
  > Only requires two passes over blurred tunnel
  > Works well if only blurring background layer
  > Quality is easily tunable by number of samples

> Disadvantages:
  > Does not work for non-fullscreen objects
    > Non-fullscreen objects require extrusion in order to sample in areas where object is not drawn
Reflections
Reflections - Motivation

> **Ruby: The Doublecross** had only planar dynamic reflections
  > Used a combination of a UCP and 2D projective texture fetch
> Non planar reflections were done with static cube maps
  > The objects were not in a moving environment

> **Ruby: Dangerous Curves** required dynamic reflections for the Ruby’s helmet, bike, and glass
> Implemented using dynamic cube maps
Reflections – Dynamic Cube Maps

> Created lower cost version of tunnel geometry
  > The maps are smaller resolution and pre-blended with a lightmap
  > The tunnel proxy geometry is lower poly than the original tunnel
> Each frame, the six cube faces from Ruby’s helmet POV are rendered
  > Tunnel proxy and explosions are rendered into 128x128 cube map
> Relatively inexpensive: the cost is <1.5ms per frame
  > Good bang for the buck!
Reflections – Dynamic Cubemap Wireframe
Reflections – Final Frame
Explosions
Explosions - Wireframe
Explosion – Overview

> Rhino created explosions in Maya
> Animated textures on moving quads
> Per-bone visibility trick used to reduce draw calls:
  > Each quad is animated with its own bone in Maya
  > The engine places a visibility value per-bone in the VS constant store
  > The VS sets the position to 0 if the bone is invisible
> Glows are used to achieve the brightness of pre-rendered cinematic
Sushi Demo Engine – Draw Script Overview

> The draw script in Sushi is responsible for:

  > Initializing all scene buffers
  > Initiating rendering commands

> A simple example from the Radeon 9700 CarPaint demo:

```c
// Create color buffer for reflection map
CreateColorBuffer cReflection(512, 384, RGBA)
CreateZBuffer zReflection(512, 384, 24)

// Draw reflections in reflection texture
SetRenderTarget (0, cReflection, zReflection)
SetViewport (0.0, 0.0, 1.0, 1.0)
DrawReflections ("NormalMap")

// Render scene to backbuffer
SetRenderTarget (0, backBuffer, backBuffer)
SetViewport (0.0, 0.0, 1.0, 1.0)
SetProjectionBasedOnViewport ()
Clear Z(1.0)
DrawObjects ("Normal", Opaque, Receive, NotReceive)
DrawScene ("Normal", Opaque, Trans)
DrawObjects ("Normal", Trans, Receive, NotReceive)
```
Sushi Draw Script – Motivation for Change

> New rendering algorithms require increased flexibility

> The previous scripting system:
  > A simple parsed language compiled into a stream of commands
  > Inflexible and awkward to extend
  > Lacked language constructs such as flow control
  > Often required special-cased engine code

> In preparation for Ruby: Dangerous Curves, we decided to re-architect the scripting engine.
Sushi Draw Script – Requirements

> Portable – Windows, Mac, Linux, and 64-bit
> Fast – draw script executed once per frame
> Small – web distribution
> Extensible – extensive customization required

Lua fits all four requirements:

> Portable – implemented in ANSI C, source code available
> Fast – relatively high benchmark for interpreted language
> Small - <500K for lua/tolua
> Extensible – easy to bind to C/C++ code
Sushi Draw Script – Lua Integration

> Used tolua to bind to engine code

  > Bound all existing draw script commands as Lua functions
    > Ported all draw scripts from previous demos to Lua as a proof-of-concept

  > Exposed numerous engine classes to Lua:
    > Matrices, Vectors
    > Lights
    > Debug UI Elements

> Added ability to bind Lua script variables to HLSL shader constants

> The demo engine automatically calls user Lua Initialize() function at startup

> A separate .ini file specifies Lua render function to execute at runtime

  > Each drawing mode has its own setting
Sushi Draw Script – Pre-Lua Example

> Rendering glows:

```plaintext
// extract parts that should glow
SetRenderTarget (0, cBlurPongBuffer)
SetViewport (0, 0, 1, 1)
DrawQuad ("qGlow", "ExtractGlow")

// apply bloom filter
SetRenderTarget (0, cBlurPingBuffer)
SetViewport (0, 0, 1, 1)
DrawQuad ("qGlow", "Bloom0")
// make border pixels black
DrawOutlineAroundViewport (0.0, 0.0, 0.0)

SetRenderTarget (0, cBlurPongBuffer)
SetViewport (0, 0, 1, 1)
DrawQuad ("qGlow", "Bloom1")
DrawOutlineAroundViewport (0.0, 0.0, 0.0)

SetRenderTarget (0, cBlurPingBuffer)
SetViewport (0, 0, 1, 1)
DrawQuad ("qGlow", "Bloom2")
DrawOutlineAroundViewport (0.0, 0.0, 0.0)

// render glows
SetRenderTarget (0, backBuffer, backBuffer)
SetViewport (0, 0, 1, 1)
DrawQuad ("qGlow", "CompositeGlow")
```
Sushi Draw Script – Lua Example

> Rendering glows in *Ruby: Dangerous Curves:*

```lua
function RenderGlow ()
    local glowPingpong = {[0] = cBlurPingBuffer, [1] = cBlurPongBuffer}
    local glowShaderTag = {[0] = "GlowToPing", [1] = "GlowToPong"}
    local bufferSelect = 0

    // downsample and extract parts that should glow
    SetRenderTarget (0, cBlurPongBuffer)
    SetViewport (0, 0, 1, 1)
    DrawQuad ("qGlow", "DownSample")

    // apply bloom blur
    for i = 0, 5, 1 do
        bufferSelect = math.mod(i, 2)
        shaderParam.glowPass = i
        SetRenderTarget (0, glowPingpong[bufferSelect])

        SetViewport (0, 0, 1, 1)
        DrawQuad ("qGlow", glowShaderTag[bufferSelect])
    end
end
```
Sushi Draw Script – Another Lua Example

> Rendering dynamic cube map in *Ruby: Dangerous Curves:*

```lua
// Draw reflection cubemap
function DrawReflectionCubemap ()
    EnableOverrideCamera (TRUE)
    vHelmetPos = GetBoundingSphereCenter("Helmet")

    for i = 0, 5, 1 do
        SetCubemapRenderTarget (0, cReflCubemap, CubemapFacePosX+i)
        SetWorldSpaceCubemapCamera (vHelmetPos, CubemapFacePosX+i)

        // Just clear the z-buffer
        Clear (Z)

        DrawObjects("TunnelReflection", Opaque, Receive)
        DrawObjects("Explosion", Opaque, Trans, Receive, NotReceive)
    end

    DisableOverrideCamera ()
```
