

IG Custom Shaders and Effects

Rob Wyatt
Insomniac Games

Who needs custom rendering

- Artists
 - Materials which are not related to our standard shader model but need to be applied to regular assets. Anisotropic, Water, Fur etc.
- Programmers
 - Special rendering effects for all manner of things.
 - Typically need the effect applied to an individual instance, not the shader or class.

Types of custom rendering

- The engine provides support for the following:
 - Custom shaders
 - Procedural shaders
 - Render overrides
- All the custom rendering modes need some sort of programmer support, not always engine support.
- Its much easier than using something like Flex in which you have to manage your own geometry.

Custom Shaders

- Implements a standalone material that can be used on regular assets.
- Needs tools support for the editor dialog, enabling artists to set the properties.
- The shader data must be less than 128 bytes
- Ideally works on all asset types but in reality only a subset is implemented.
- Typically implemented by the engine.

Procedural Shaders

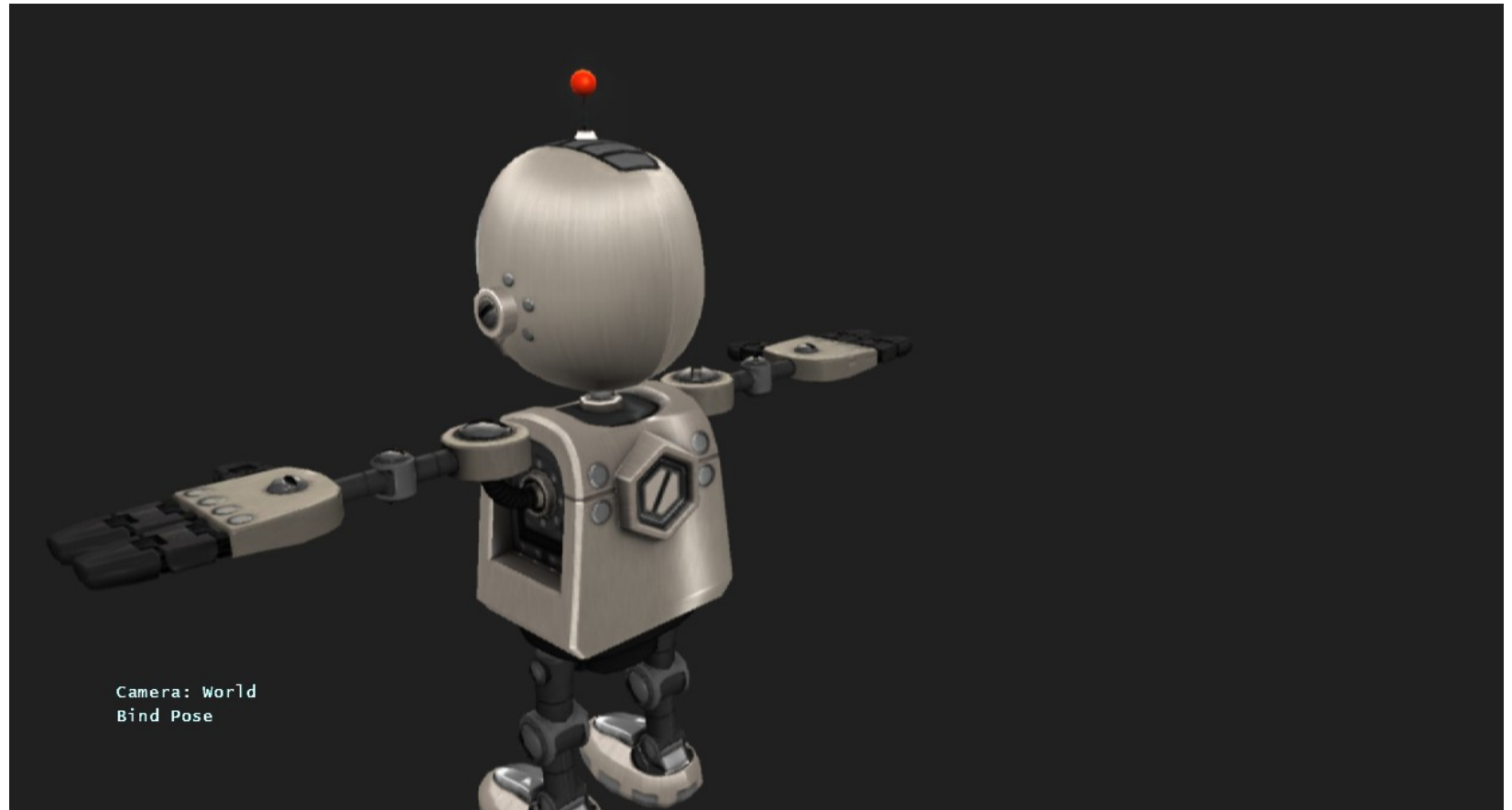
- Standard or custom shaders that have callbacks into engine/game code.
- The callback is more like a message and you get a reason code.
- Typical reason codes Init, Close and Update which is called per frame.
- Callbacks are normally game specific, live in game code and are implemented by gameplay programmers.
- The callback is free to modify the material in any way. It can animate shader values, change textures.
- More complex callbacks may render geometry or special effects to a texture and then use the result.

Render Overrides

- Only works for mobys.
- Per instance callback that can be set in the moby instance.
- The callback function is called instead of using the normal rendering code.
- The callback is called once for the entire moby.
- You can render the moby or you can render something completely different.
- Cloaking effect is implemented with this.
- Gameplay can implement their own callbacks.

Anisotropic

- Example of a full custom shader.



Anisotropic

- The following are examples of anisotropic surfaces:
 - Satin cloth
 - Christmas tree balls (the ornaments covered in threads)
 - Compact Discs
 - Brushed metal
 - Hair

Anisotropic

- Anisotropic surfaces have an inherent grain (locally, favors a particular surface tangent orientation)
- Distribution of surface normals along the surface scratches or fibers is different from the distribution across them
- A relatively distant observer sees the lighting result, but not the microstructure.

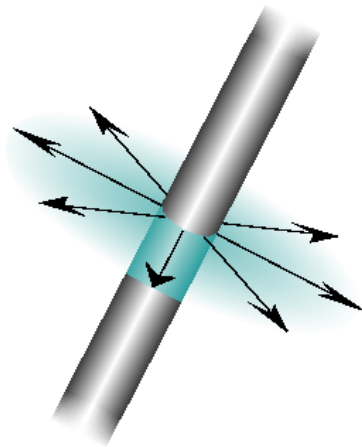
Anisotropic

- Traditional lighting models consider 3D surfaces as being locally flat
- The normal vector at a point on the surface is orthogonal to the flatness
- Implicitly assumes surface tangents are uniformly distributed (isotropic distribution)
- Effects such as bump mapping may perturb the surface normals, but substructure of the surface still modeled as locally flat

Anisotropic

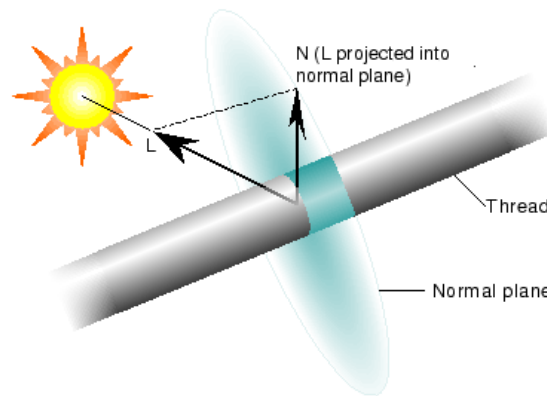
- From the art point of view the shader is pretty much the same as the basic shader.
- It has 3 textures, base map, normal map and grain map. It has a couple of tint colors and a specular power.
- The grain map sets the anisotropic direction. Its very difficult to create.
- Shader only supports mobys

- How would you light a thread or hair?
- There is no traditional surface normal!
- Logically, each point on the thread has an infinite “circle” of normals perpendicular to the tangent at the point.



Anisotropic

- The full solution is to integrate the full the contribution of all normals.
- Too expensive, Too hard and not possible in a fragment program
- Pick the most significant normal and use it.



Anisotropic

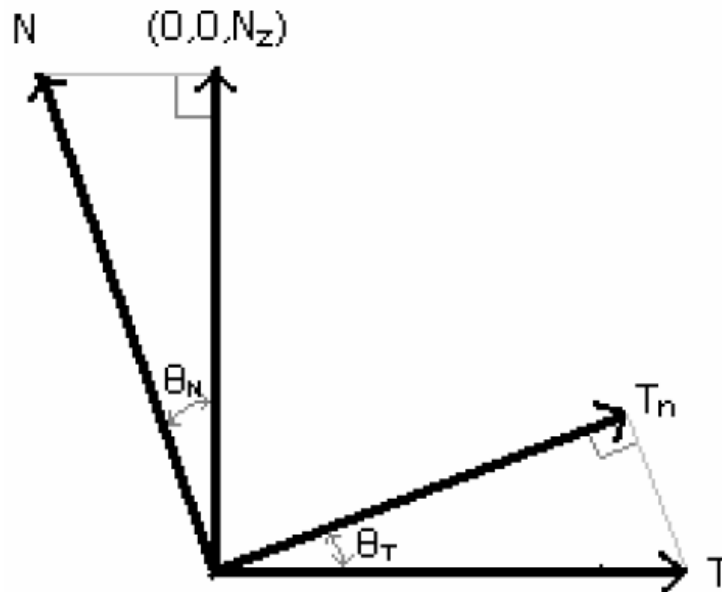
- Pick the normal that is co-planar with the light vector and view vector.
- This vector will maximize the lighting dot products (i.e., it's the most significant vector).
- If you work the math out on paper you don't need to calculate the most significant normal.
- You don't need the normal at all!

Anisotropic

- Diffuse = $\text{sqrt}(1 - (\langle L \cdot T \rangle)^2)$
- Specular = $\text{sqrt}(1 - (\langle L \cdot T \rangle)^2) \times \text{sqrt}(1 - (\langle V \cdot T \rangle)^2) - \langle L \cdot T \rangle \times \langle V \cdot T \rangle$
- L is light to thread direction
- V is eye to thread direction
- T is the tangent (grain) direction

Anisotropic

- Actually, We do need a normal.
- Our shader implements per pixel anisotropic and the grain direction changes with the normal map.



Anisotropic

- We need to keep the grain perpendicular to the per pixel normal.
- Graham-Schmidt orthonormalization does just this and its cheap.
 - $T_{\text{new}} = || T - (N \cdot T) N ||$
- Once we have computed the new grain we are done with the normal.

Anisotropic

- Implemented as a full custom shader.
- Artists can apply the shader like any other
- Artists control all the properties.
- There are not many controls:
 - Base map
 - Normal map
 - Grain map
 - Tint and specular power controls

Anisotropic

- The grain map is difficult to author.
- Its similar to a normal map but its stored as an XY component.
- Its basically just a direction and could be stored as an angle.
- Very difficult to author if you want swirls like brushed steel.
- The code is implemented as 2 vertex programs and a fragment program.

Anisotropic

- Problems with Anisotropic lighting.
- Its inherently 2 sided so you have to prevent the back side of surfaces from lighting.
- It needs a light direction making it very difficult to implement as a baked light – What is the light direction?
- Real time lights are much easier, they have a direction and a color.
- This is why we only support mobys, the have 2 directional lights.

Cloaked



Cloaked

- Implemented as a render override.
- Artists don't get any say in it, the shader is not available to them.
- Can be enabled per instance and different instances can use different settings.
- It's a very simple effect, no math!
- It's a little expensive for the GPU.
- Was used by the Pirates in RCF.

Cloaked

- Enabled by setting “m_user_layer_func” in the moby instance to the render callback function.
- “m_user_layer_data” points to a structure that contains the settings and controls.
- Available Control:
 - Scale and bias control the distortion
 - Tint color tints the cloaked effect
 - Bloom adjust allow the cloaked objects to bloom
- All the controls can be animated for more dynamic effects.

Cloaked

- When enabled the engine sends the entire moby to the sort layer.
- The callback is called by the core engine in sorted back to front order.
- The effect is treated as if its alpha.

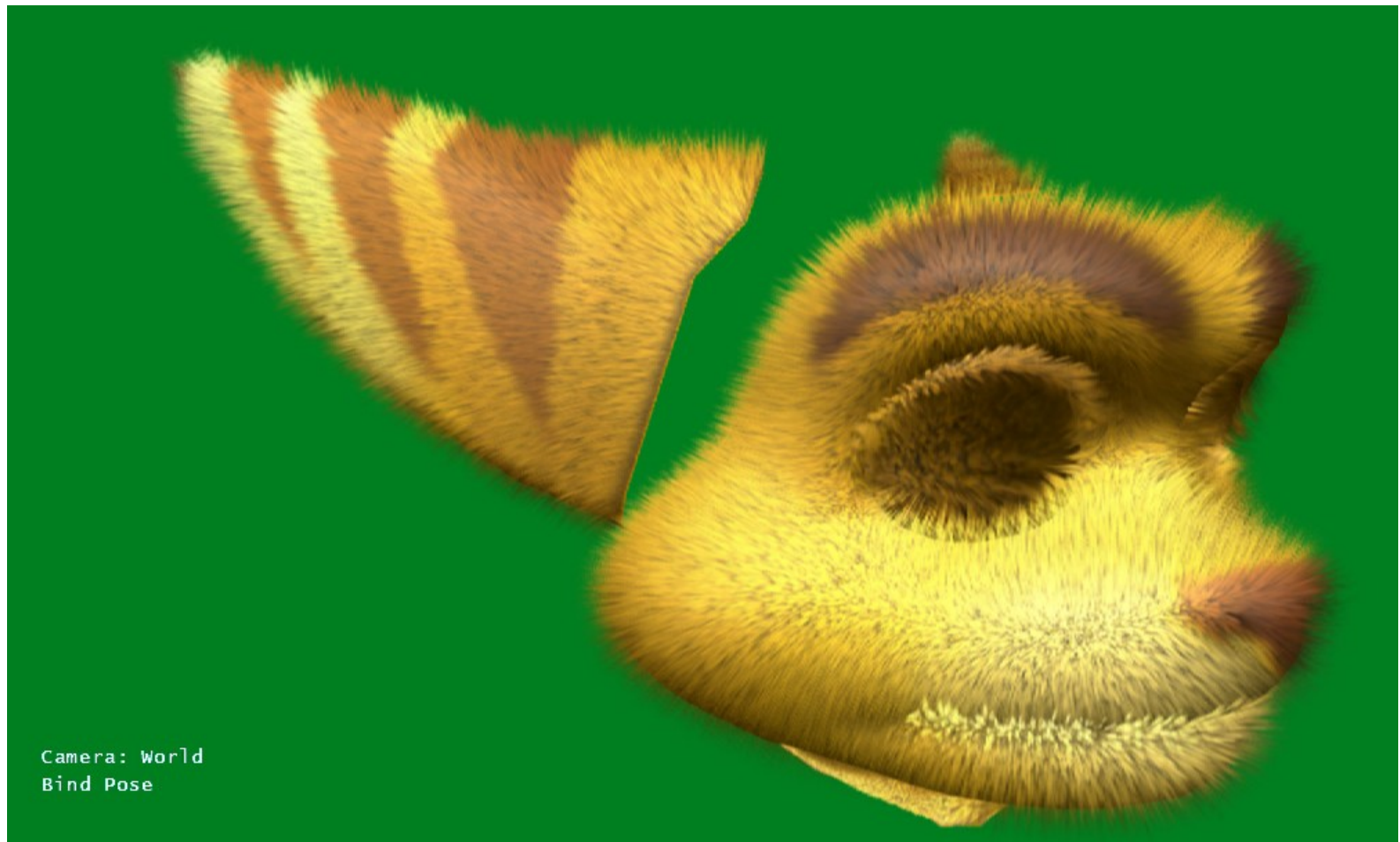
Cloaked

- The callback function grabs the current frame buffer as a texture.
 - Requires 4mb of memory in the effects heap.
- Renders the moby with depth only
- Renders the moby again using the grabbed framed buffer as a texture.
- The normal maps of the shaders used by the moby in normal rendering are used to offset the samples.
- Its all controlled by the control structure.

Cloaked

- One feature that is missing is the ability to mix the cloaked effect with the normal rendering.
- This would allow the solid objects to gradually cloak themselves rather than just switching on and off.
- It's not immediately obvious how this would be implemented.
- It's especially difficult if the movie has real time lighting or shadows.

Fur



Fur

- Implemented as an artist controlled custom shader.
- Only supports mobys.
- Has controls for length and density of the fur.
- Uses a base texture to color the hair.
- There is a built in texture that models the hair itself.

Fur

- It's a really simple technique.
- Its expensive for the GPU but can be done entirely on the GPU.
- Lots of render passes
 - Maybe skin and render the same mesh 16 or more times.
- Each pass is rendered in alpha so lots of bandwidth is used.

Fur

- Traditionally fur used fins and shells.
- The fins cause problems for a GPU implementation because they require finding the silhouette edge.
- Fins are typically done on the CPU but this has problems with skinning because the CPU never sees the skinned positions.
- This fur is rendered entirely with shells.
- Its known as “Lengyel’s concentric fur shell technique”.

Fur

- First a solid moby is rendered, this is the base or skin level.
- For each shell the moby is rerendered
 - The image of ratchet has 20 shells.
- Each shell is scaled per vertex in the direction of the normal.
- Each shell is rendered with a “Hair Slice” texture.

Fur



Fur

- The first slice texture is placed on the first shell, the second on the second shell etc etc
- The hair texture is noisy and tiles nicely.
- This allows so the base map UVs to be used for the slice UVs, regardless of what they are.
- We can control the global scale of the UVs, this controls the density and size of each strand of hair. At the lowest level it looks more like feathers.
- That's all it does for the basic fur.

Fur

- We do a little more work..
- Hair has anisotropic lighting and we know the direction of the hair.
- The hair direction can be changed via the normal map.
- We use the alpha channel in the basemap to mask where hair can go.
 - Very expensive because you are still paying the full cost of all the shells.
- We change the hair direction by slightly adjusting the UVs in a given direction.
- The UV adjustment is scaled by the shell distance so external shells move the UVs more than internal shells.

Fur

- Other people have used the GPU to apply physical properties to the fur. They render a texture which contains the offset of each hair strand.
- When making the offset texture you can account for wind, movement of the object, drag and inertia.
- You can do as much math as you like in the pixel shader, ultimately you are writing out a 2D vector.
- We tried this but it had some math which nobody seemed to understand.
- The fur was not used in RCF. Ratchet use a different technique that was half done by code and half by art.
- Ratchets fur was also a custom shader.