

VO Rendering SS 2010

Unit 9: Pixar RenderMan

Sources:

Overview

- Pixar RenderMan / REYES
 - ◆ Highly complex software system used for a large portion of today's industrial CG work
- Software shaders
 - ◆ Technology behind complex object appearance with simple basic geometry

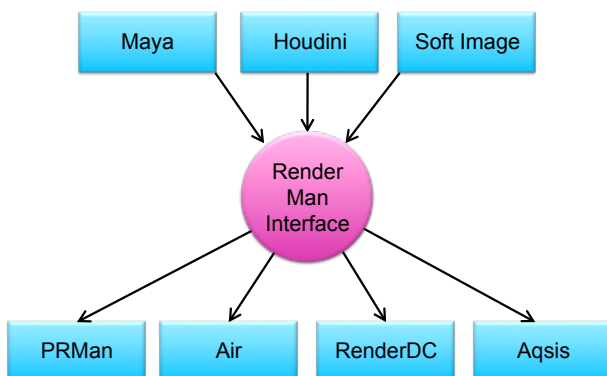
State of the Art in Graphics



RenderMan Naming Confusion

- RenderMan can be either one of three things:
 - ◆ **RenderMan SL**: The 3D scene description language defined by Pixar Inc.
 - ◆ **RenderMan Interface**: The interface between modelling and rendering
 - ◆ **PRMan**: The RenderMan-compliant hybrid scanline renderer sold by Pixar (REYES)
- For a long time, Pixar PRMan was the only Rman-compliant system
- BMRT was the first alternative (now defunct)

What is Renderman?



RenderMan – The Product (PRMan)

- Pixar Photorealistic RenderMan (PRMan)
- Evolved gradually since 1982 / 84 from the Lucasfilm Renderer
- Basically a sophisticated scanline renderer
- Currently at release 14.0
 - ◆ Indirect illumination / GI
 - ◆ Hair & fur optimizations
 - ◆ Parallel network rendering
 - ◆ On demand raytracing

PRMan Features - Hair



PRMan Features – Motion Blur



REYES



- “Render everything you ever saw”
- REYES = software architecture
- PRMan implements the REYES architecture



REYES Assumptions and Goals 1



- High possible model complexity
 - ◆ Esp. data amplification primitives, such as fractals, procedural models etc.
- Diverse types of primitives
 - ◆ Complexity of real scenes comes from surface appearance as much as from geometry
 - ◆ Programmable shaders a necessity
- Shading complexity



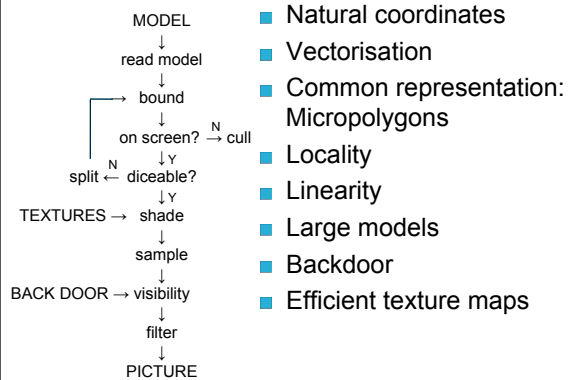
REYES Assumptions and Goals 2



- Minimal ray tracing
 - ◆ Approximation of non-local effects through other means (e.g. shadow maps)
- Speed
 - ◆ Important for animations: 2h movie in 1 year - < 3 min per frame!
- Image quality
 - ◆ Anti-aliasing and proper pixel filtering is considered to be essential



REYES Algorithm: Design Principles



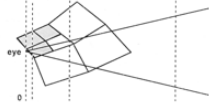
Geometric Primitive Routines



- ```

MODEL
 ↓
read model
 ↓
bound
 ↓
on screen? → N cull

```
- **Bound**
    - ◆ Computes bounding box information
  - **Culling**
    - ◆ Primitives which do not intersect the visible region are discarded without being diced or split
    - ◆ Each part is tested again

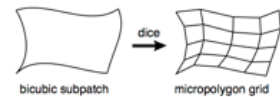


## Geometric Primitive Routines



- ```

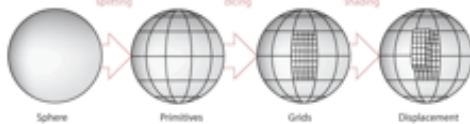
MODEL
  ↓
read model
  ↓
bound
  ↓
on screen? → N cull
  ↓
split ← N diceable?
    
```
- **Diceable test**
 - ◆ Examines micropolygon size & number
 - **Split**
 - ◆ Subdivision into other geometric primitives
 - **Dice**
 - ◆ Perform the actual split into micropolygons



Micropolygons



- All primitives are diced into micropolygons
- The entire shading process operates on this single monochrome type of primitive
- Roughly half a pixel across
- MP generation operates in eye space
- Subdivision is always done in the primitive's (u,v) space, never in screen space



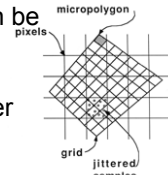
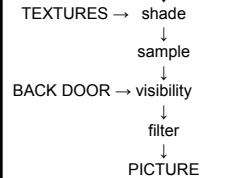
Hiding



- ```

MODEL
 ↓
read model
 ↓
bound
 ↓
on screen? → N cull
 ↓
split ← N diceable?

```
- **Backdoor feature** intended to incorporate raytracing
  - **Sampling**
    - ◆ Micropolygons are the Nyquist limit for their pixels
    - ◆ Jittered samples trade aliasing artifacts for noise
    - ◆ Sampling can be influenced
  - **Reconstruction functions** are user choice: RiFilter



## Complete Algorithm



```

Initiate the z buffer.
For each geometric primitive in the model,
 Read the primitive from the model file.
 If the primitive can be bounded,
 Bound the primitive in eye space.
 If the primitive is completely outside of the hither-far z range, cull it.
 If the primitive spans the z plane and can be split,
 Mark the primitive undiceable.
 Else
 Convert the bounds to screen space.
 If the bounds are completely outside the viewing frustum, cull the primitive.
 If the primitive can be diced,
 Dice the primitive into a grid of micropolygons.
 Compute normals and tangent vectors for the micropolygons in the grid.
 Shade the micropolygons in the grid.
 Break the grid into micropolygons.
 For each micropolygon,
 Bound the micropolygon in eye space.
 If the micropolygon is outside the hither-far range, cull it.
 Convert the micropolygon to screen space.
 Bound the micropolygon in screen space.
 For each sample point inside the screen space bound,
 If the sample point is inside the micropolygon,
 Calculate the z of the micropolygon at the sample point by interpolation.
 If the z of the sample point is less than the z in the buffer,
 Replace the sample in the buffer with this sample.
 Else
 Split the primitive into other geometric primitives.
 Put the new primitives at the head of the unsorted portion of the model file.
 Filter the visible sample hits to produce pixels.
 Output the pixels.

```



## REYES Advantages



- Can handle arbitrary number of primitives
  - ◆ Basically a batch renderer for individual primitives
- No inversions – projections of pixels onto textures
- Computations can easily be vectorised (e.g. shading)
- No clipping calculations
- Frequently no texture filtering is needed





## Shape vs. Shading



- Shape
  - ◆ Geometric configuration of objects
- Shading
  - ◆ Calculates the appearance of an object in a scene under a set of light sources
  - ◆ Result defined by
    - Colors of the surface and the light source(s)
    - Position and orientation of the surface relative to the light
    - Roughness of the surface



## Shape vs. Shading



plain geometry



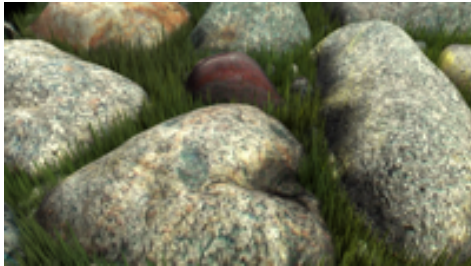
vs. shaded geometry



## Shape vs. Shading Example



- Only two square polygons
- Transformed by shading



## Shading Pipeline



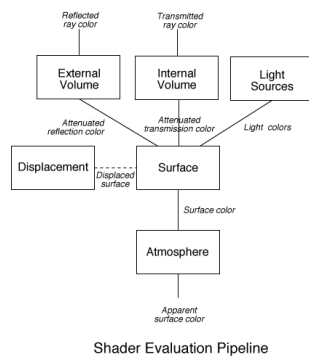
- Three major types of shaders
  - ◆ Emission at the light source
  - ◆ Interaction of the light with the surface
  - ◆ Atmospheric effects between the surface and the viewpoint



## Types of Shaders



- RenderMan Interface supports
  - ◆ Light source shaders
  - ◆ Surface shaders
  - ◆ Volume shaders
  - ◆ Displacement shaders
  - ◆ Imager shaders
- Each shader type has a specific set of variables and result types



## Shader Language Var Types



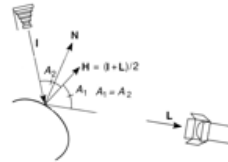
- Floats
- Colours
  - ◆ Multiple colour models can be used, RGB default
- Points
- Strings
- Uniform vs. Varying variables
  - ◆ Uniform vars are constant everywhere over the area under consideration



## Surface Shader



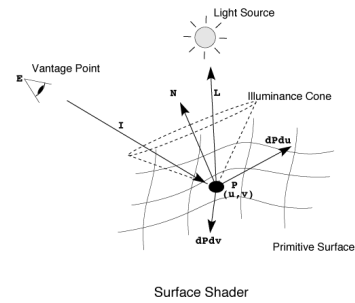
- Determines the colour of light reflecting from a point on a surface in a particular direction
- Does not have to be physically plausible!



## Surface Shader Environment



- Expected results: incident ray colour and opacity



Surface Shader



## Surface Shader Vars



| Name   | Type  | Storage Class   | Description                            |
|--------|-------|-----------------|----------------------------------------|
| Cs     | color | varying/uniform | Surface color                          |
| Os     | color | varying/uniform | Surface opacity                        |
| P      | point | varying         | Surface position                       |
| dPdu   | point | varying         | Derivative of surface position along u |
| dPdv   | point | varying         | Derivative of surface position along v |
| N      | point | varying         | Surface shading normal                 |
| Ng     | point | varying/uniform | Surface geometric normal               |
| u,v    | float | varying         | Surface parameters                     |
| du, dv | float | varying/uniform | Change in surface parameters           |
| s,t    | float | varying         | Surface texture coordinates            |
| L      | point | varying/uniform | Incoming light ray direction*          |
| Cl     | color | varying/uniform | Incoming light ray color*              |
| OI     | color | varying/uniform | Incoming light ray opacity*            |
| E      | point | uniform         | Position of the eye                    |
| I      | point | varying         | Incident ray direction                 |
| acomps | float | uniform         | Number of color components             |
| time   | float | uniform         | Current shutter time                   |

- Some of these input variables can be modified, e.g. the shading normal
- Most – such as the geometric normal – are fixed



## Example: Surface Shader #1



```
surface metal (
 float Ka = 1,
 float Ks = 1,
 roughness = .25)
{
 point Nf = faceforward(normalize(N),I);
 OI = Os;
 Ci = Os * Cs * (Ka * ambient() +
 Ks * specular(Nf, -I, roughness));
}
```



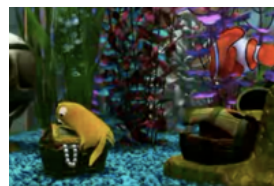
## Example: Surface Shader #2



```
surface txtplastic(
 float Ka = 1;
 float Kd = .5;
 float Ks = .5;
 float roughness = .1;
 color specularcolor = 1;
 string mapname = "")
{
 point Nf = faceforward(N,I);
 if (mapname != "")
 Ci = color texture(mapname);
 else
 Ci = Cs;
 OI = Os;
 Ci = Os * (Ci * (Ka * ambient() + Kd * diffuse(Nf)) +
 specularcolor * Ks * specular(Nf, -I, roughness));
}
```



## Bubbles from Finding Nemo



### Example: Shading Fish Guts

### Light Source Shader

- Calculates the intensity and color of light sent by the light source to a point on a surface

### Light Source Shader Vars

| Name  | Type                  | Storage Class | Description                            |
|-------|-----------------------|---------------|----------------------------------------|
| P     | point varying         |               | Surface position                       |
| dPdu  | point varying         |               | Derivative of surface position along u |
| dPdv  | point varying         |               | Derivative of surface position along v |
| N     | point varying         |               | Surface shading normal                 |
| Ng    | point varying/uniform |               | Surface geometric normal               |
| u,v   | float varying         |               | Surface parameters                     |
| du,dv | float varying/uniform |               | Change in surface parameters           |
| s,t   | float varying         |               | Surface texture coordinates            |
| L     | point varying/uniform |               | Incoming light ray direction*          |
| Ps    | point varying         |               | Position being illuminated             |
| E     | point uniform         |               | Position of the eye                    |
| ncmps | float uniform         |               | Number of color components             |
| time  | float uniform         |               | Current shutter time                   |
| Cl    | color varying/uniform |               | Outgoing light ray color               |
| OI    | color varying/uniform |               | Outgoing light ray opacity             |

- Very similar to surface variable set
- Describes lightsource, not the surface being illuminated!

### Example: Slideprojector

### Flexible Lightsource Examples

### Volume (Atmosphere) Shader

- Generalizes the idea of atmosphere affecting light passing through space between a surface and the eye

- (Not really supported until very recently)

## Volume Shader Example



```

volume underwater{
 float mindist=0, maxdist= 1;
 color fg=1, bg=1;
 float inten=1, gam=1, mixf=0.5;
}
{
 color c;
 float d;

 d = length(I);
 if(d<mindist)
 c = fg;
 else if(d==maxdist)
 c = bg;
 else
 d = (d-mindist)/(maxdist-mindist);
 d = pow(d,gam);
 c = mix(fg,bg,d);

 Cl = inten*mix(Cl,c,mixf);
 Qi = mix(Qi, color(1,1,1), d);
}
// underwater()

```



## Volumetric Shader in *Open Season*



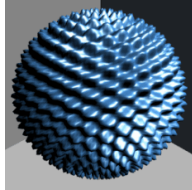
© 2006 Sony Pictures Imageworks



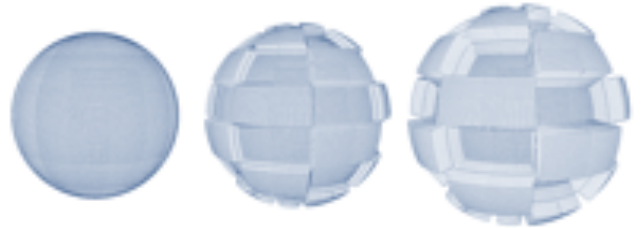
## Displacement Shaders



- Distort the geometry of the basic object
- Difference to bump maps (which would be a surface shader): the silhouette is correct!
- Costly to evaluate



## Displacement Shader Geometry



## Displacement Vars



| Name  | Type  | Storage Class   | Description                            |
|-------|-------|-----------------|----------------------------------------|
| P     | point | varying         | Surface position                       |
| dPdu  | point | varying         | Derivative of surface position along u |
| dPdv  | point | varying         | Derivative of surface position along v |
| N     | point | varying         | Surface shading normal                 |
| Ng    | point | varying/uniform | Surface geometric normal               |
| E     | point | uniform         | Position of the eye                    |
| u,v   | float | varying         | Surface parameters                     |
| du,dv | float | varying/uniform | Change in surface parameters           |
| st    | float | varying         | Surface texture coordinates            |
| ncmps | float | uniform         | Number of color components             |
| time  | float | uniform         | Current shutter time                   |
| P     | point | varying         | Displaced surface position             |
| N     | point | varying         | Displaced surface shading normal       |

- Full gamut of surface environment variables is accessible



## Displacement Shader Example #1



```

displacement sinwaves(float ffreq=1.5, amp=1.5, sphase=0, tphase=0, parandir=0)
{
 // displace along normal, using sin(x) or sin(t) or both
 if(!parandir)
 {
 P += amp*cos(sphase+u*ffreq)*PI)*normalize(N);
 }
 else if (!tphase)
 {
 P += amp*cos(tphase+u*ffreq)*PI)*normalize(N);
 }
 else
 {
 P += amp*cos(sphase+u*ffreq)*PI)*sin(tphase+u*ffreq)*PI)*normalize(N);
 }
 N = calculateNormal(P);
}
// sinwaves

```



## Displacement Shader Example #2



```

Displacement pits(
 float Xm = 0.03;
 string mapname = "");
{
 float magnitude;
 if (marks != "")
 magnitude = float texture(marks);
 else
 magnitude = 0;
 P += -Xm * magnitude * normalize(N);
 N = calculatenormal(P);
}

```



## River Raindrops in *Ratatouille*



© Disney / Pixar.



## WALL-E Trash Shader



## Transformation Shaders



- Similar to displacement shaders in that they modify object geometry resp. point coordinates
- Difference: used at a different, earlier stage of the rendering pipeline
- Used to transform entire objects
- Restricted variable set

| Name | Type  | Storage Class | Description          |
|------|-------|---------------|----------------------|
| P    | point | varying       | Position             |
| N    | point | varying       | Normal at that point |
| time | float | uniform       | Current shutter time |

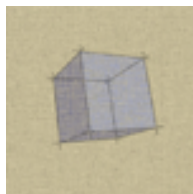
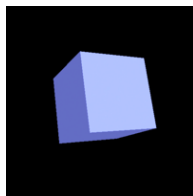
|   |       |         |                      |
|---|-------|---------|----------------------|
| P | point | varying | Transformed position |
| N | point | varying | Transformed normal   |



## Imager Shaders



- Used to transform already computed colours to something else
- Applications: e.g. cartoonish distortions of realistic renderings



## Imager Variables



| Name   | Type  | Storage Class | Description                |
|--------|-------|---------------|----------------------------|
| P      | point | varying       | Surface position           |
| Ci     | color | varying       | Pixel color                |
| Oi     | color | varying       | Pixel opacity              |
| alpha  | float | uniform       | Fractional pixel coverage  |
| ncomps | float | uniform       | Number of color components |
| time   | float | uniform       | Current shutter time       |
| Ci     | color | varying       | Output pixel color         |
| Oi     | color | varying       | Output pixel opacity       |

- Only colour information and z-Buffer data are provided



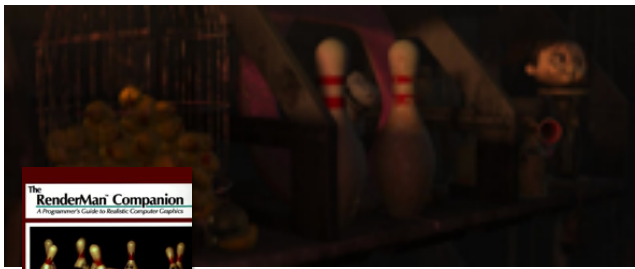
## Combined Example: Bowling Pin



## Bowling Pins



## Easter Egg: Bowling Pins in WALL · E



## Shadows: A Sore Point



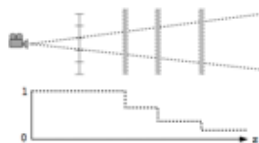
- Automatic shadow generation not classic RM
- Shadow maps (i.e. depth images from the perspective of the light) have to be prepared for each lightsource (!)
- Surface shaders have to use this information
  - ◆ The simple shaders in the previous examples are not capable of exhibiting shadows!
- Raytracing and GI options in newer RM versions somewhat obsolete this



## Deep Shadow Maps



- DSM store representation of the fractional visibility through a pixel at all possible depths
- Transmittance function describes light falloff
- Stored as an array of floating-point pairs
- Can handle volumetric effects and semi-transparent surfaces



## Reflections: Also A Sore Point



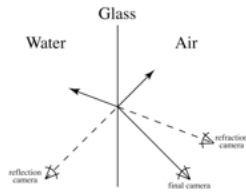
- Similar to depth images for the lights, reflections have to be pre-computed
- Reflection maps have artefacts: no multiple inter-reflections
- Fast
- (Sort of obsoleted by raytracing on demand)



## Reflection/Refraction Multipass Rendering



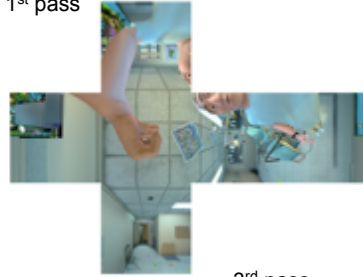
- Multi-pass rendering
- Reflection and refraction images are input for tank shader
- Each wall has unique pair of reflection and refraction camera
- Texture is projected



## Example: Reflection Map



1<sup>st</sup> pass



2<sup>nd</sup> pass



3<sup>rd</sup> pass



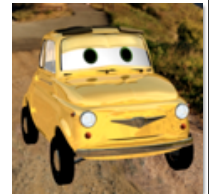
## Final Rendering



## Ray Tracing On Demand



- Scanline: fast, can handle complex scenes; shadows and reflections are problematic
- Ray tracing can not deal with complex scenes
- Ray differentials
- First-level rays originate from REYES shading points



## Luigi – CG and RL :-)



Copyright Disney/Pixar 2006



## Implementations



- Original Pixar renderer (REYES)
  - ◆ Micropolygon-based hybrid with raytracing capabilities
- BMRT
  - ◆ Raytracer, has disappeared after legal action was taken against author
- Pixie
  - ◆ Open-source RenderMan
- Realtime techniques
  - ◆ Ongoing research topic



The End  
Thank you for your attention!

