VU Rendering SS 2012

Unit 9: Renderman

Overview

1. Pixar RenderMan / REYES
- Highly complex software system used for a large portion of today's industrial CG work

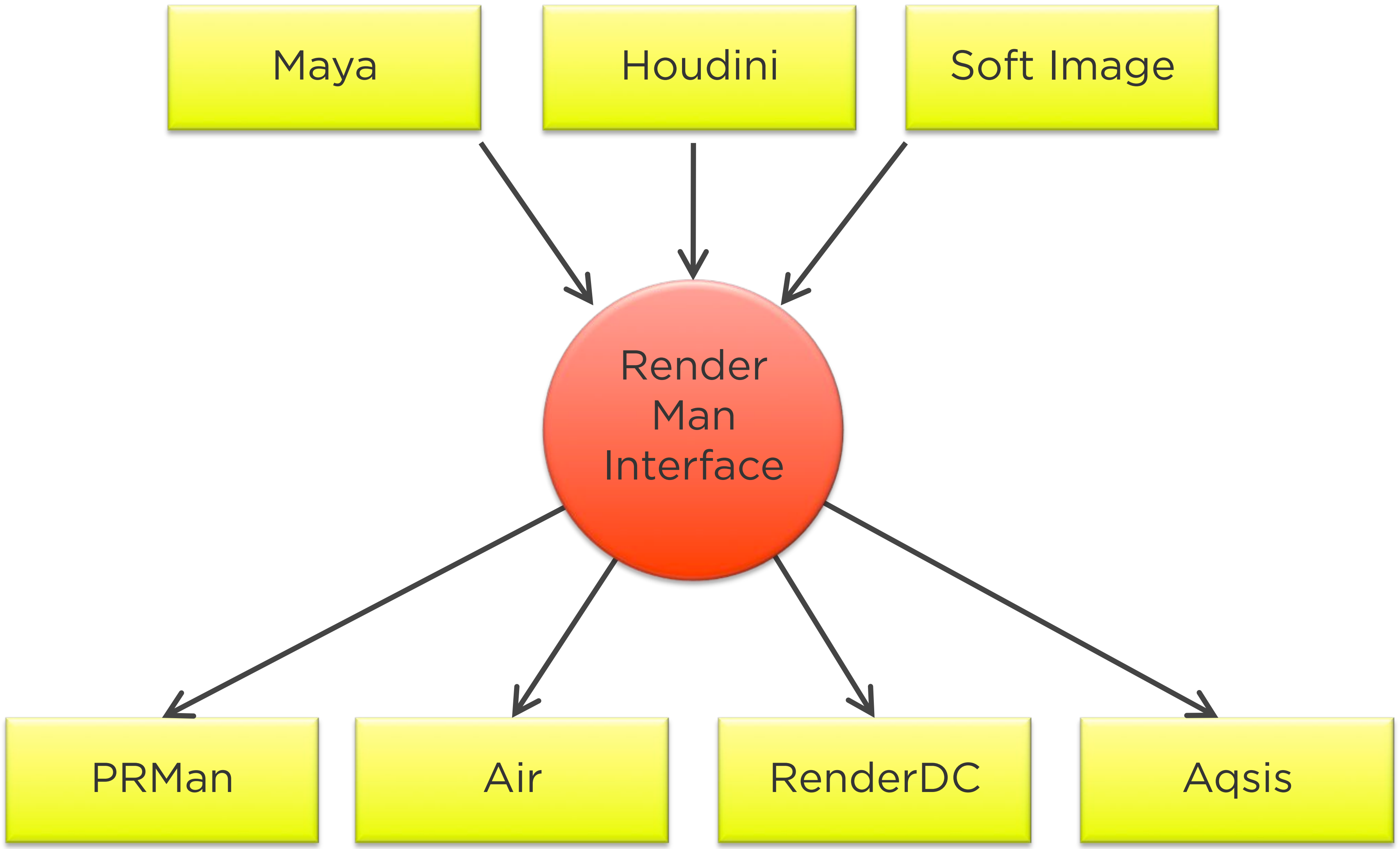
2. Software shaders
- Technology behind complex object appearance with simple basic geometry

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)

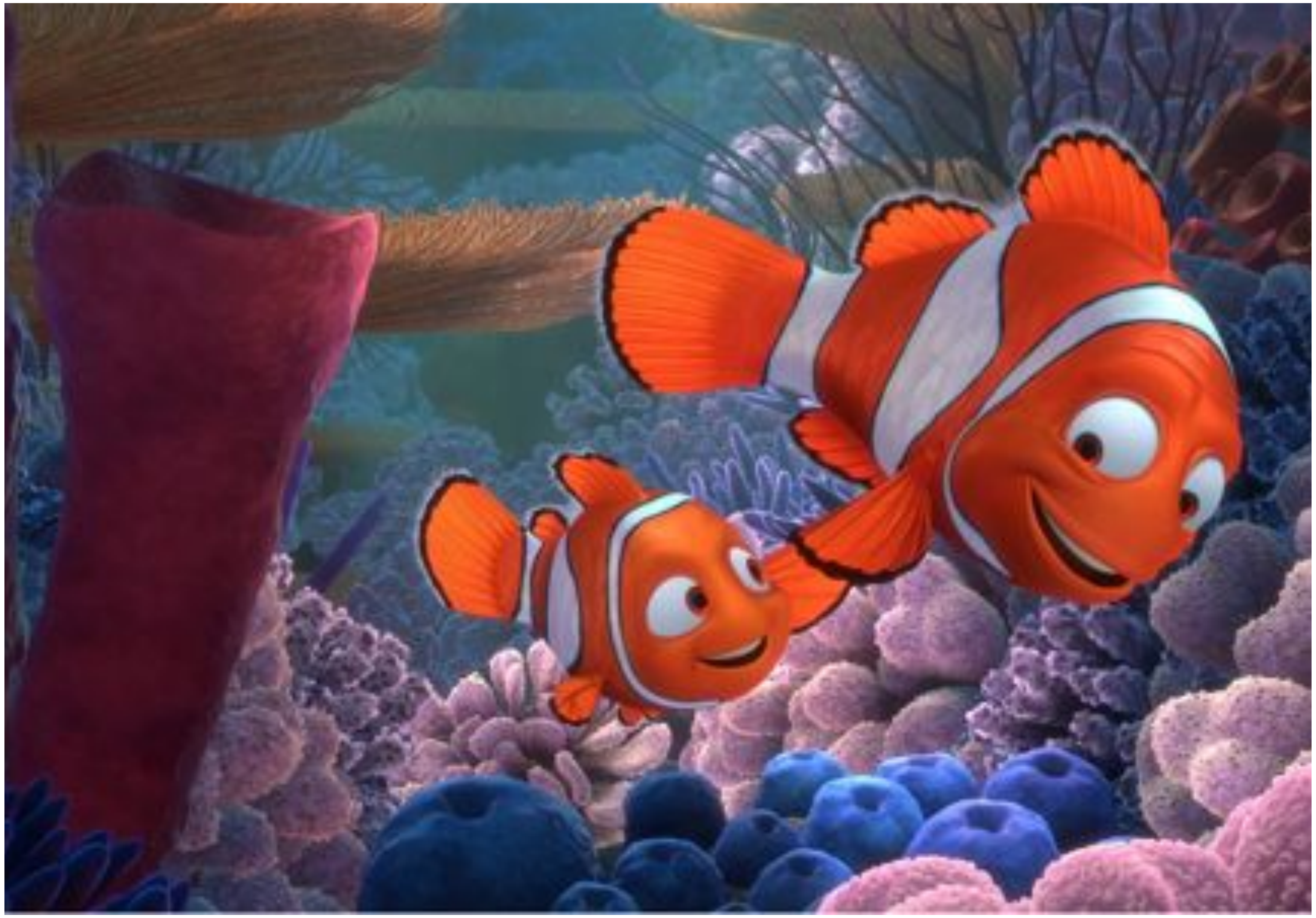
What is Renderman?



RenderMan – The Product (PRMan)

- Pixar Photorealistic RenderMan (PRMan)
- Evolved gradually since 1982 / 84 from the Lucasfilm Renderer
- Basically a sophisticated scanline renderer (can be bypassed)
- Currently at release 16.0
  - Displacements
  - Camera controls
  - Particles
  - Indirect illumination / GI
  - Hair & fur optimizations
  - SSS
  - Parallel network rendering
  - On demand raytracing

PRMan Features - Displacements

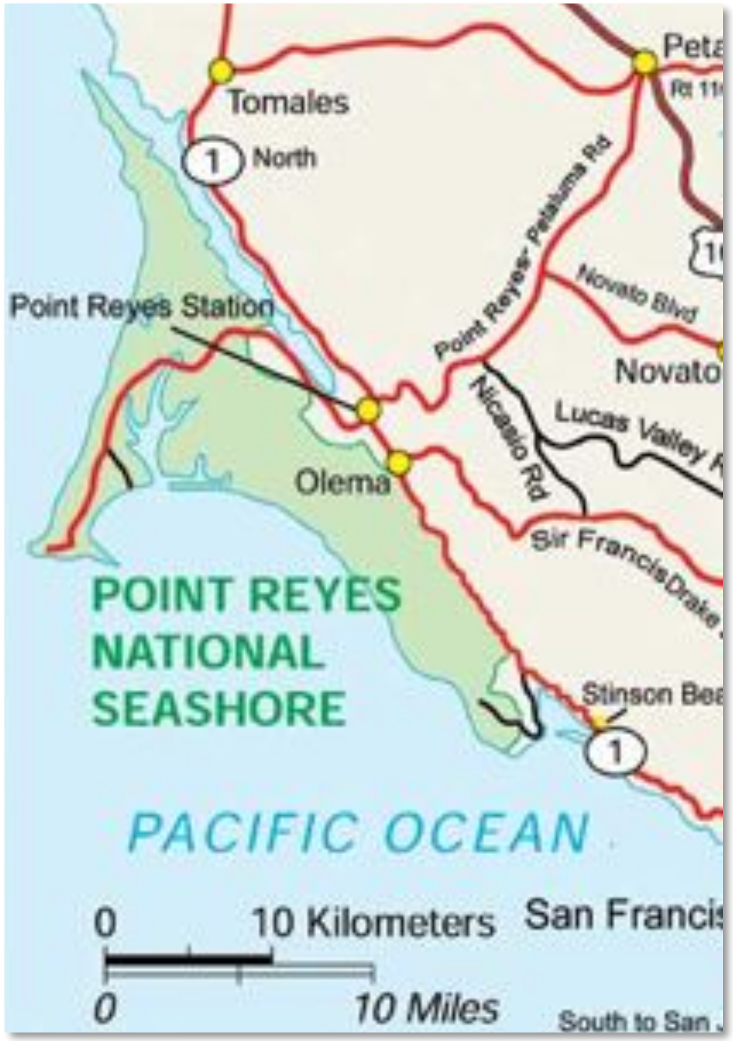


PRMan Features - Hair

PRMan Features – Motion Blur

REYES

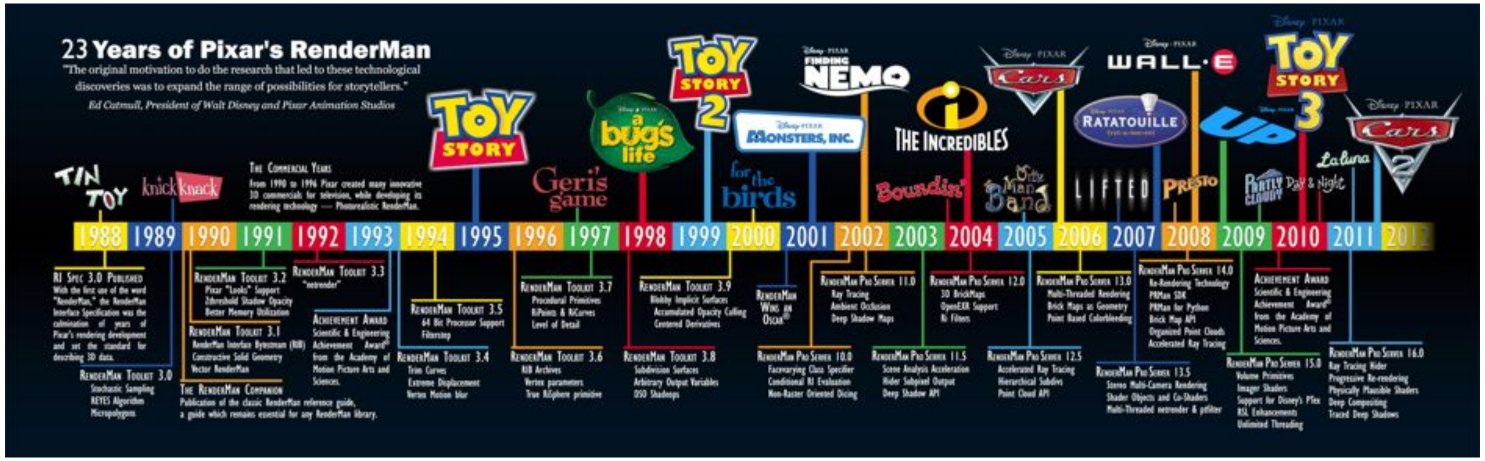
- "Render everything you ever saw"
- REYES = software architecture
- PRMan implements the REYES architecture



REYES Assumptions and Goals 1

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

REYES Assumptions and Goals 2



23 Years of Pixar's RenderMan

"The original motivation to do the research that led to these technological discoveries was to expand the range of possibilities for storytellers."
Ed Catmull, President of Walt Disney and Pixar Animation Studios

THE COMMERCIAL YEARS
From 1990 to 1996 Pixar created many innovative 3D commercials for television, while developing its rendering technology — Photorealistic RenderMan.

RI SPEC 3.0 PUBLISHED
With the first use of the word "RenderMan," the RenderMan Interface Specification was the culmination of years of Pixar's rendering development and set the standard for describing 3D data.

RENDERMAN TOOLKIT 3.0
Stochastic Sampling
REYES Algorithm
Micropolygons

RENDERMAN TOOLKIT 3.2
Pixar "Looks" Support
Zthreshold Shadow Opacity
Better Memory Utilization

RENDERMAN TOOLKIT 3.1
RenderMan Interface Bytestream (RIB)
Constructive Solid Geometry
Vector RenderMan

THE RENDERMAN COMPANION
Publication of the classic RenderMan reference guide, a guide which remains essential for any RenderMan library.

RENDERMAN TOOLKIT 3.3
"netrender"

ACHIEVEMENT AWARD
Scientific & Engineering Achievement Award® from the Academy of Motion Picture Arts and Sciences.

RENDERMAN TOOLKIT 3.5
64 Bit Processor Support
Filterstep

RENDERMAN TOOLKIT 3.4
Trim Curves
Extreme Displacement
Vertex Motion blur

RENDERMAN TOOLKIT 3.7
Procedural Primitives
RiPoints & RiCurves
Level of Detail

RENDERMAN TOOLKIT 3.6
RIB Archives
Vertex parameters
True RiSphere primitive

RENDERMAN TOOLKIT 3.9
Blobby Implicit Surfaces
Accumulated Opacity Culling
Centered Derivatives

RENDERMAN TOOLKIT 3.8
Subdivision Surfaces
Arbitrary Output Variables
DSO Shadeops

RENDERMAN WINS AN OSCAR®

RENDERMAN PRO SERVER 10.0
Facevarying Class Specifier
Conditional RI Evaluation
Non-Raster Oriented Dicing

RENDERMAN PRO SERVER 11.0
Ray Tracing
Ambient Occlusion
Deep Shadow Maps

RENDERMAN PRO SERVER 11.5
Scene Analysis Acceleration
Hider Subpixel Output
Deep Shadow API

RENDERMAN PRO SERVER 12.0
3D BrickMaps
OpenEXR Support
Ri Filters

RENDERMAN PRO SERVER 12.5
Accelerated Ray Tracing
Hierarchical Subdivs
Point Cloud API

RENDERMAN PRO SERVER 13.0
Multi-Threaded Rendering
Brick Maps as Geometry
Point Based Colorbleeding

RENDERMAN PRO SERVER 13.5
Stereo Multi-Camera Rendering
Shader Objects and Co-Shaders
Multi-Threaded netrender & ptfilter

RENDERMAN PRO SERVER 14.0
Re-Rendering Technology
PRMan SDK
PRMan for Python
Brick Map API
Organized Point Clouds
Accelerated Ray Tracing

RENDERMAN PRO SERVER 15.0
Volume Primitives
Imager Shaders
Support for Disney's PTex
RSL Enhancements
Unlimited Threading

ACHIEVEMENT AWARD
Scientific & Engineering Achievement Award® from the Academy of Motion Picture Arts and Sciences.

RENDERMAN PRO SERVER 16.0
Ray Tracing Hider
Progressive Re-rendering
Physically Plausible Shaders
Deep Compositing
Traced Deep Shadows

- 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

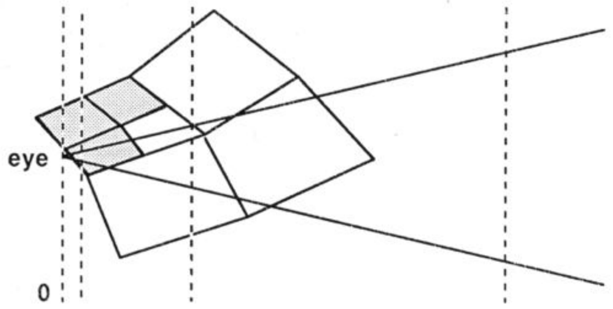
$$\begin{tikzcd}
& \text{MODEL} \arrow[d] & \\
& \text{read model} \arrow[d] & \\
& \text{bound} \arrow[d] & \\
& \text{on screen?} \arrow[r, "N"] \arrow[d, "Y"] & \text{cull} \\
\text{split} \arrow[uur, bend left] & \text{diceable?} \arrow[l, "N"'] \arrow[d, "Y"] & \\
\text{TEXTURES} \arrow[r] & \text{shade} \arrow[d] & \\
& \text{sample} \arrow[d] & \\
\text{BACK DOOR} \arrow[r] & \text{visibility} \arrow[d] & \\
& \text{filter} \arrow[d] & \\
& \text{PICTURE} &
\end{tikzcd}$$

- Natural coordinates
- Vectorisation
- Common representation: Micropolygons
- Locality
- Linearity
- Large models
- Backdoor
- Efficient texture maps

Geometric Primitive Routines

$$\begin{tikzcd}[row sep=small] \text{MODEL} \arrow[d] & \\ \text{read model} \arrow[d] & \\ \text{bound} \arrow[d] & \\ \text{on screen?} \arrow[r, "N"] & \text{cull} \end{tikzcd}$$

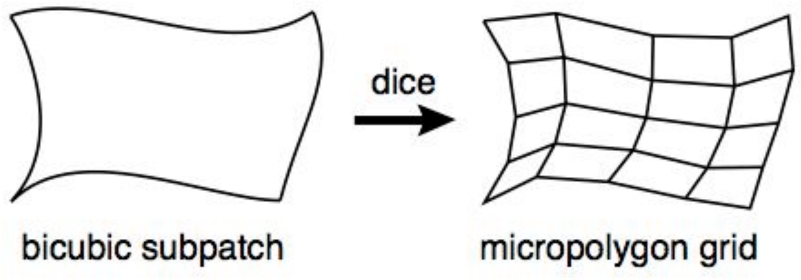
- Bound
  - Computes bounding box information
- Culling
  - Primitives which do not intersect the visible region are discarded without being diced or split
  - Each part is tested again



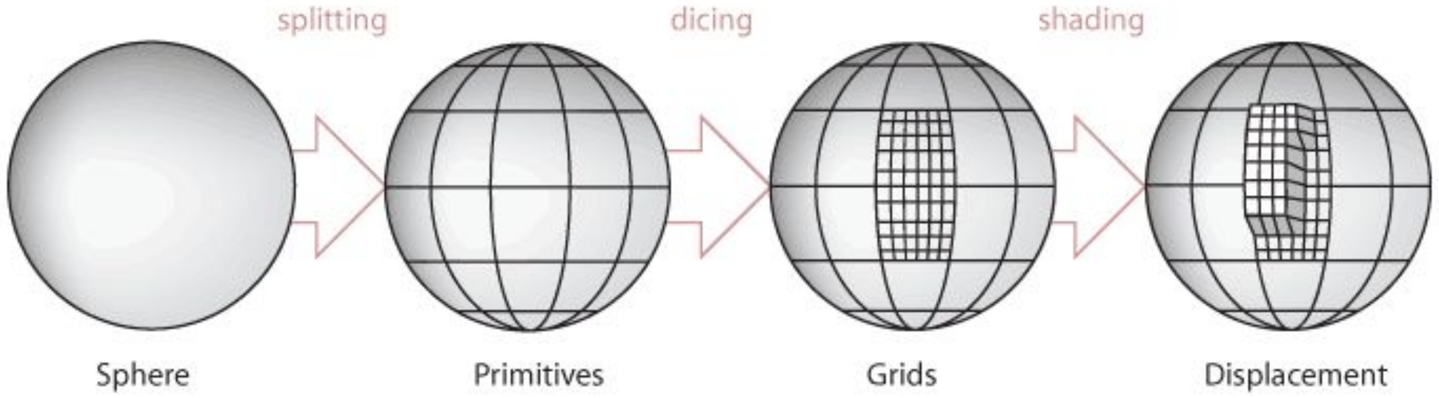
Geometric Primitive Routines

$$\begin{tikzcd}
& \text{MODEL} \arrow[d] & \\
& \text{read model} \arrow[d] & \\
& \text{bound} \arrow[d] & \\
& \text{on screen?} \arrow[r, "N"] \arrow[d, "Y"] & \text{cull} \\
\text{split} \arrow[uur] & \text{diceable?} \arrow[l, "N"'] &
\end{tikzcd}$$

- Diceable test
  - Examines micropolygon size & number
- Split
  - Subdivision into other geometric primitives
- Dice
  - Perform the actual split into micropolygons



Micropolygons

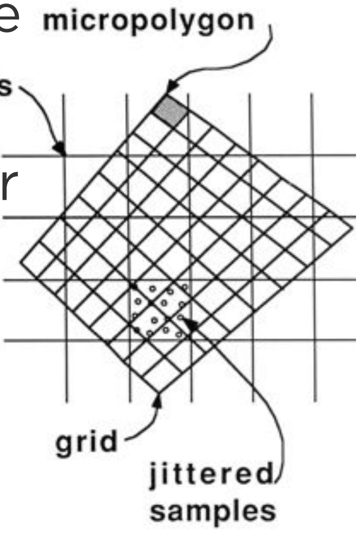


- All primitives are diced into micropolygons
- The entire shading process operates on this single monochrome type of primitive
- Roughly half a pixel across
- MP generation operates in eye space
- Subdivision is always done in the primitive's (u,v) space, never in screen space

Hiding

$$\begin{tikzcd}
 & \text{MODEL} \arrow[d] & \\
 & \text{read model} \arrow[d] & \\
 & \text{bound} \arrow[d] & \\
 & \text{on screen?} \arrow[r, "N"] \arrow[d, "Y"] & \text{cull} \\
\text{split} \arrow[uur, bend left] & \text{diceable?} \arrow[l, "N"'] \arrow[d, "Y"] & \\
\text{TEXTURES} \arrow[r] & \text{shade} \arrow[d] & \\
 & \text{sample} \arrow[d] & \\
\text{BACK DOOR} \arrow[r] & \text{visibility} \arrow[d] & \\
 & \text{filter} \arrow[d] & \\
 & \text{PICTURE} &
\end{tikzcd}$$

- 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

```
Initialize 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-yon z range, cull it.
        If the primitive spans the ε 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-yon 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 at 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 unread portion of the model file.
Filter the visible sample hits to produce pixels.
Output the pixels.
```

REYES Advantages

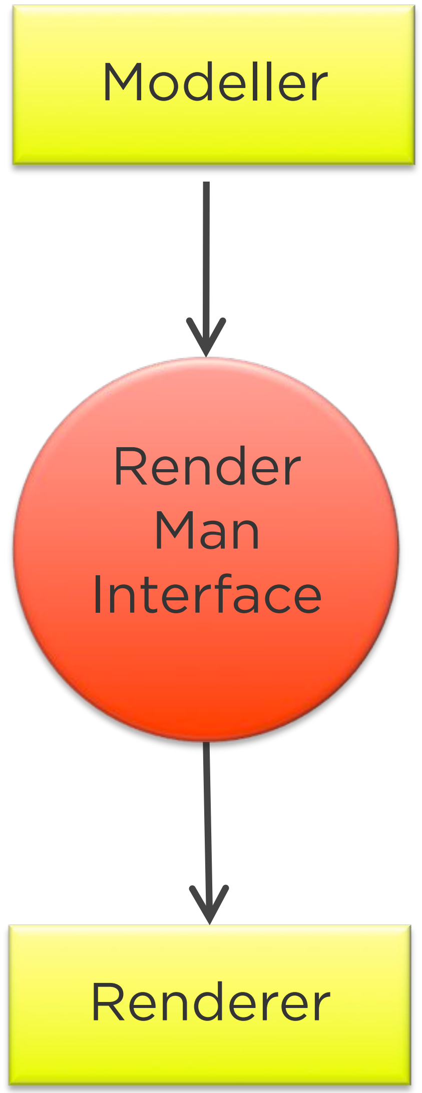
- Can handle arbitrary number of primitives
  - Basically a batch renderer for individual primitives
- No inversions – projections of pixels onto textures
- Computations can easily be vectorised (e.g. shading)
- No clipping calculations
- Frequently no texture filtering is needed

REYES Disadvantages

- No natural way to dice polygons
- Shading before sampling causes problems for motion blur
- Dicing is difficult for some types of primitives like e.g. blobs
- No coherency is exploited for large uniform objects – everything is diced into micropolygons
- No GI information of any kind is computed

RenderMan Interface

- Interface between rendering and modelling
- Powerful set of primitive surface types
  - Quadric surfaces
  - Polygons
  - Parametric surfaces
- Hierarchical modeling, geometry
- Constructive solid geometry
- Camera model (orthographic, perspective)
- Generalized shading model



Using the RM Interface

- Two basic options exist:
  - Use of RM function calls from a high-level language (e.g. C) implementation of the RM API
  - Feeding archived RM function calls from a RenderMan Interface Bytestream (RIB) to a compliant renderer (hand generated, output from modelling program)
- The actual renderers are usually non-interactive
- Separate preview renderers are used during the design phase

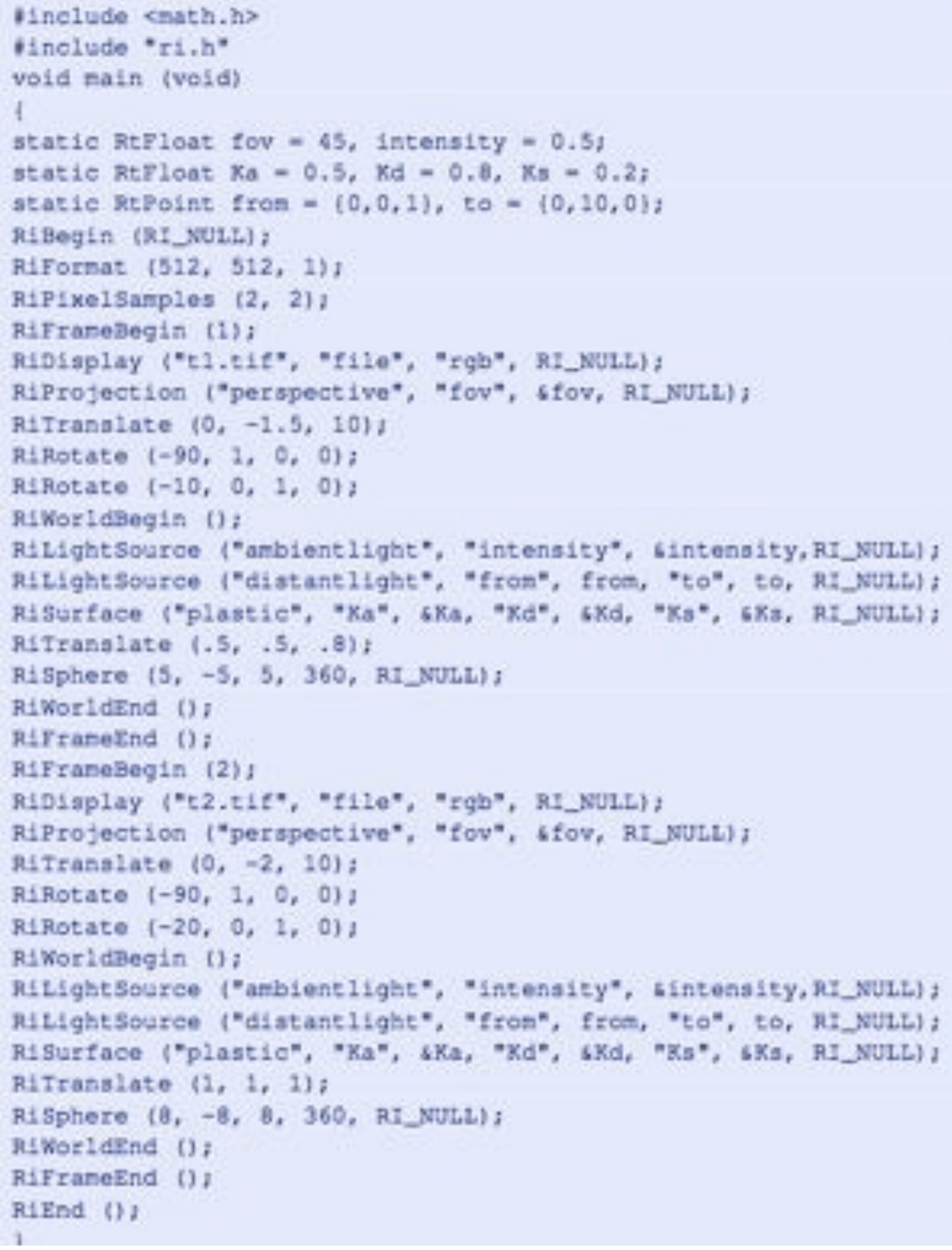
RM Program Structure

- Consistent naming of API calls (Ri…)
- All function calls bracketed between one pair of RiBegin and RiEnd
- One global graphics state is maintained within this bracket
- All API calls modify this state
- API calls are frequently varargs, and have to be terminated with RI_NULL
- Most calls deal with surface properties → shading language

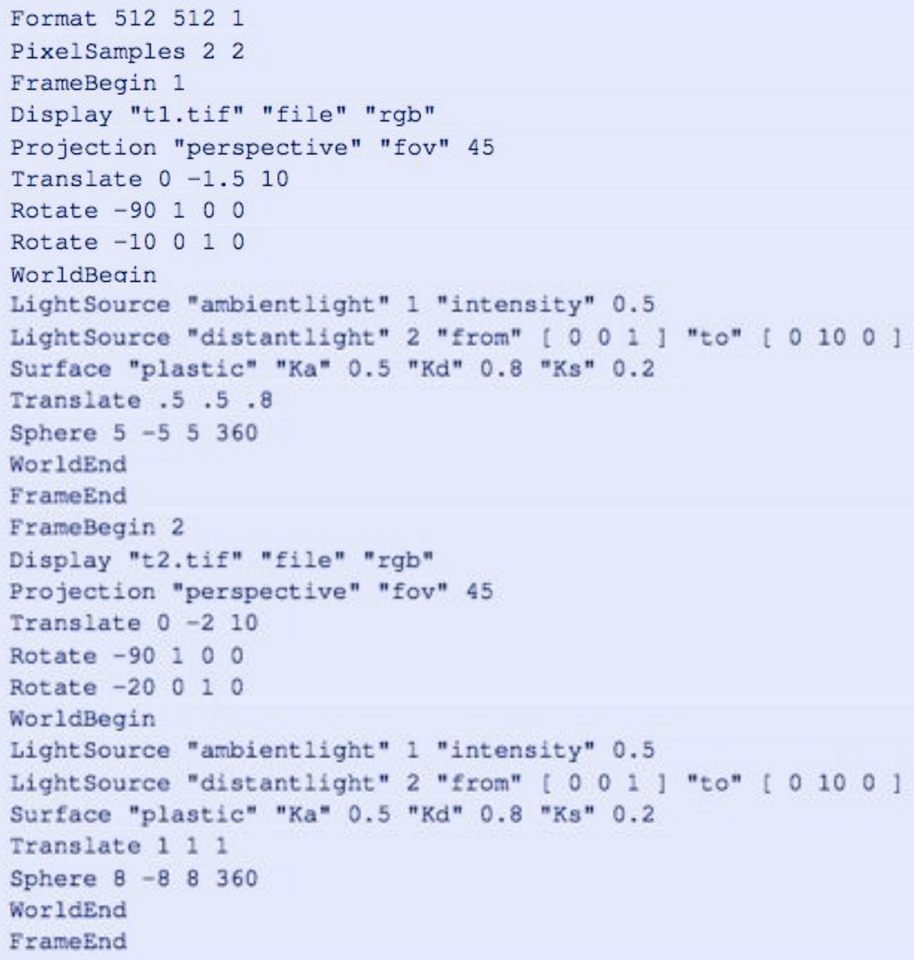
RIB File

- Sequence of requests to the renderer
- No loops, branches, …
- Hierarchical attributes/transformations
- Geometry, lights and materials are specified inside a WorldBegin/WorldEnd block
- Normally RIB file contains just one frame's worth of data

API Calls vs. RIB



```
#include <math.h>
#include "ri.h"
void main (void)
{
static RtFloat fov = 45, intensity = 0.5;
static RtFloat Ka = 0.5, Kd = 0.8, Ks = 0.2;
static RtPoint from = {0,0,1}, to = {0,10,0};
RiBegin (RI_NULL);
RiFormat (512, 512, 1);
RiPixelSamples (2, 2);
RiFrameBegin (1);
RiDisplay ("t1.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -1.5, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-10, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (.5, .5, .8);
RiSphere (5, -5, 5, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiFrameBegin (2);
RiDisplay ("t2.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -2, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-20, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (1, 1, 1);
RiSphere (8, -8, 8, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiEnd ();
}
```

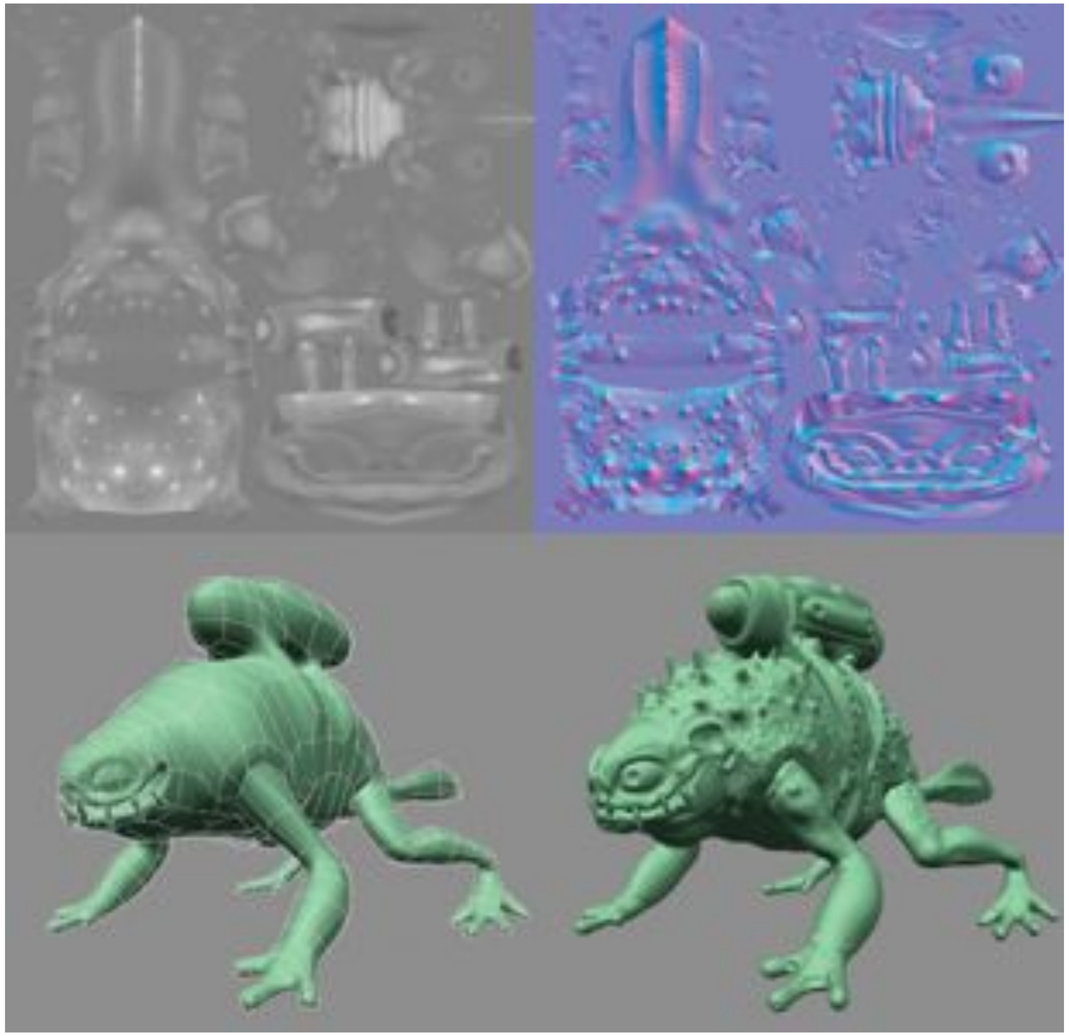


```
Format 512 512 1
PixelSamples 2 2
FrameBegin 1
Display "t1.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -1.5 10
Rotate -90 1 0 0
Rotate -10 0 1 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate .5 .5 .8
Sphere 5 -5 5 360
WorldEnd
FrameEnd
FrameBegin 2
Display "t2.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -2 10
Rotate -90 1 0 0
Rotate -20 0 1 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate 1 1 1
Sphere 8 -8 8 360
WorldEnd
FrameEnd
```

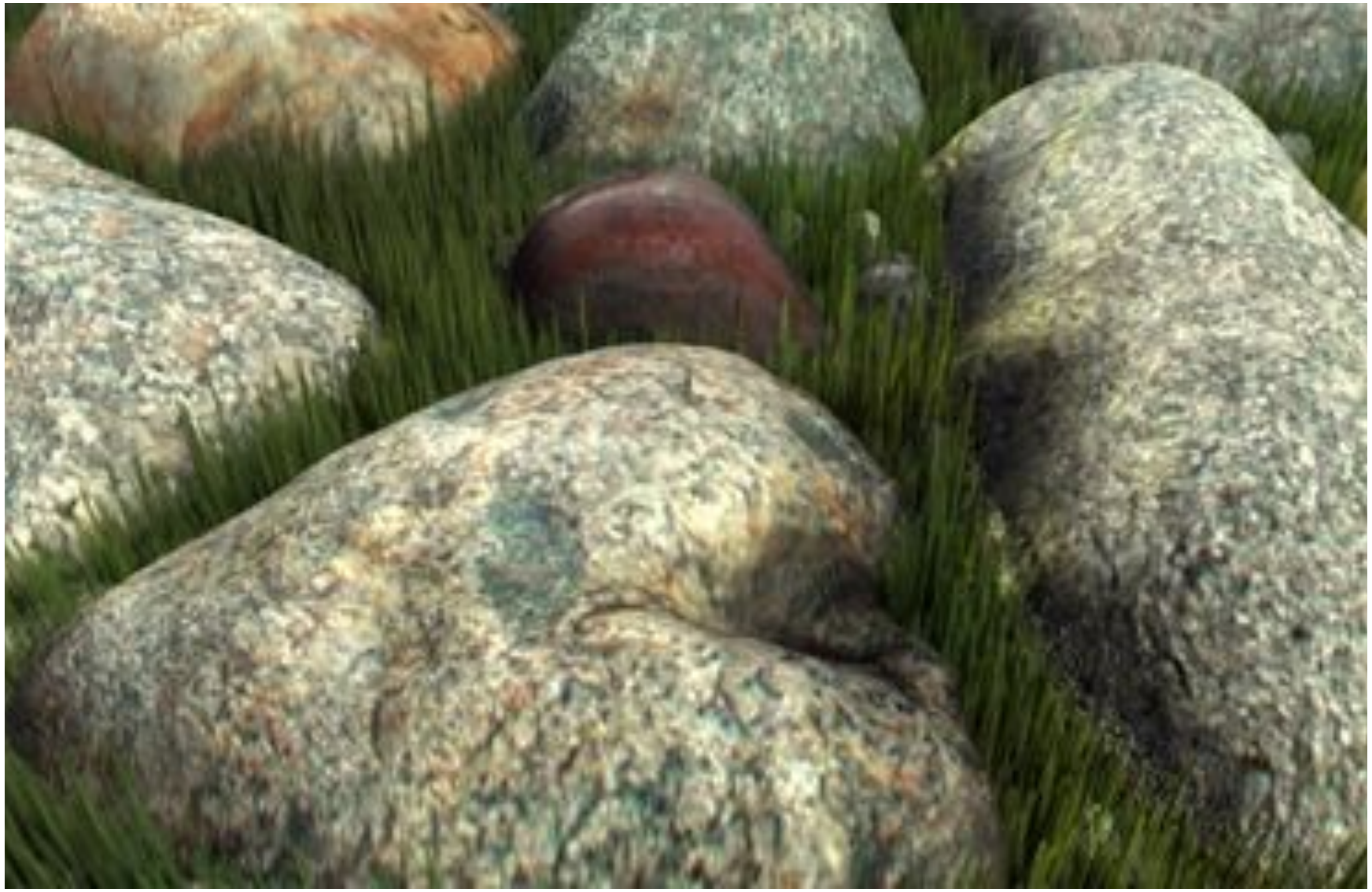
Shape vs. Shading

- 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



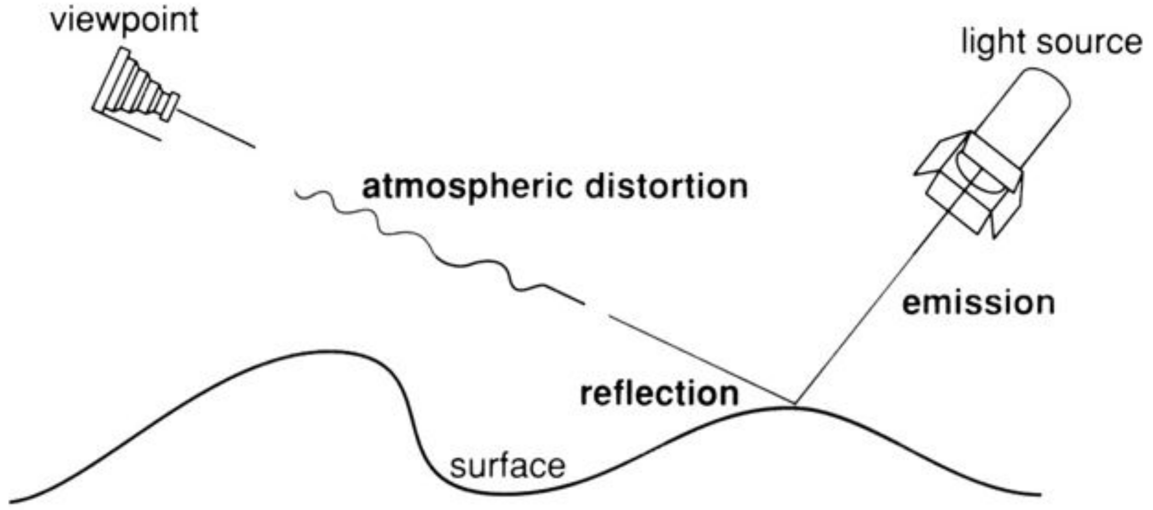
Shape vs. Shading Example



Only two square polygons, Transformed by shading

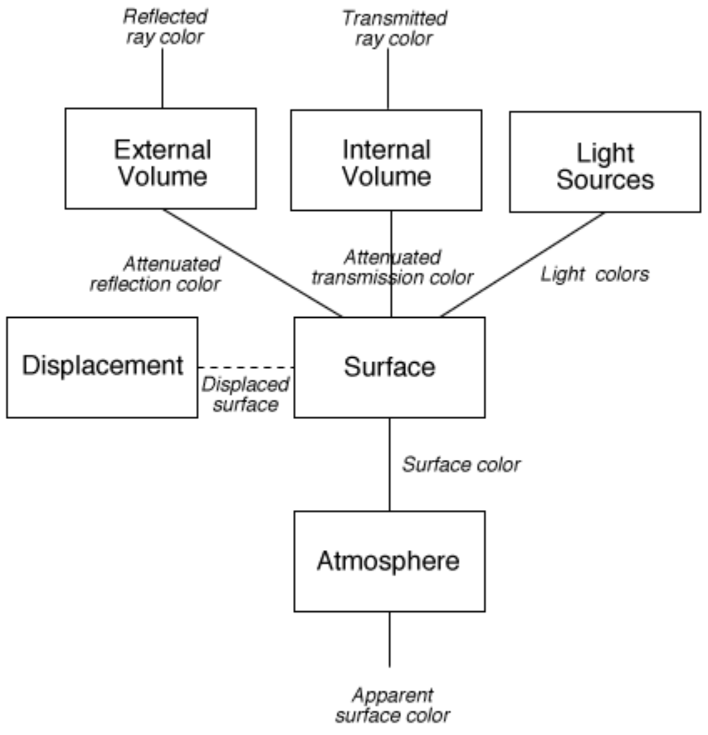
Shading Pipeline

- 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



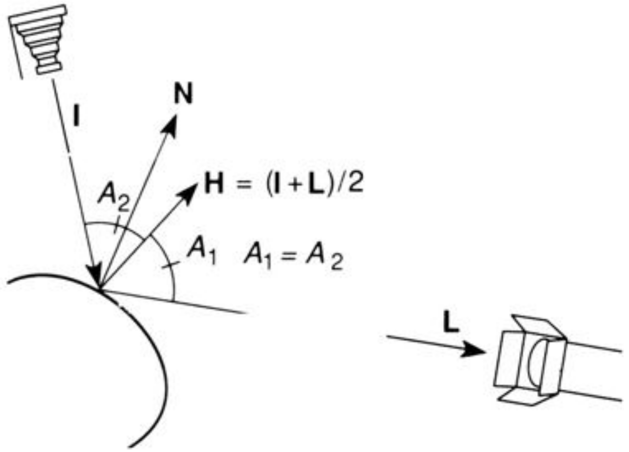
Shader Evaluation Pipeline

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

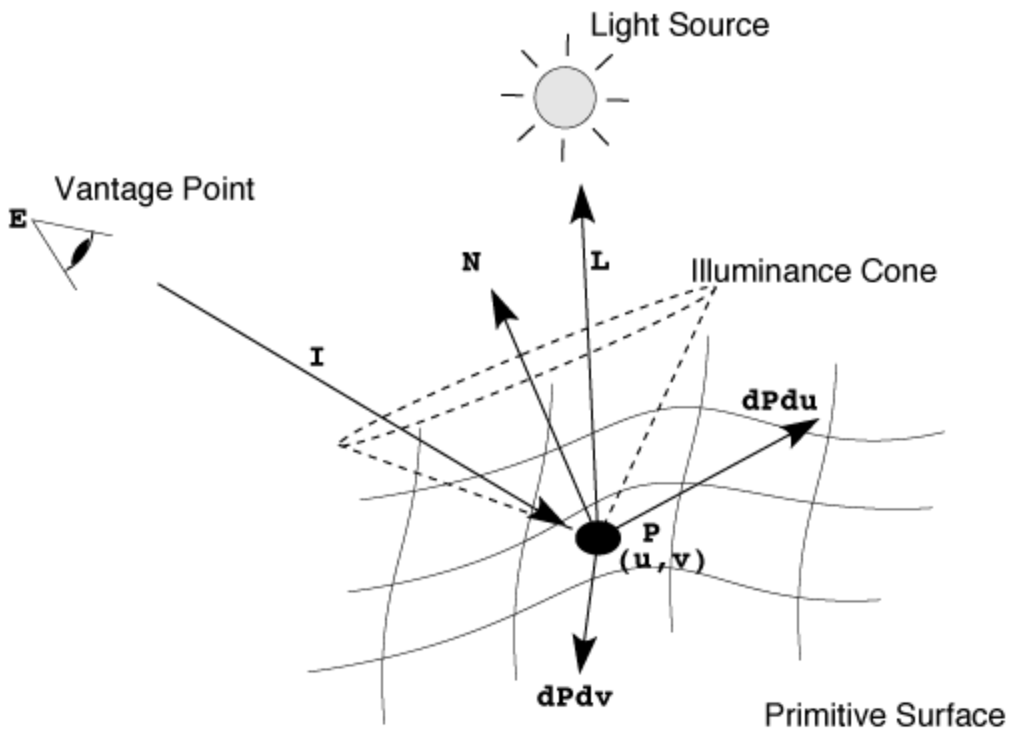
Surface Shader

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



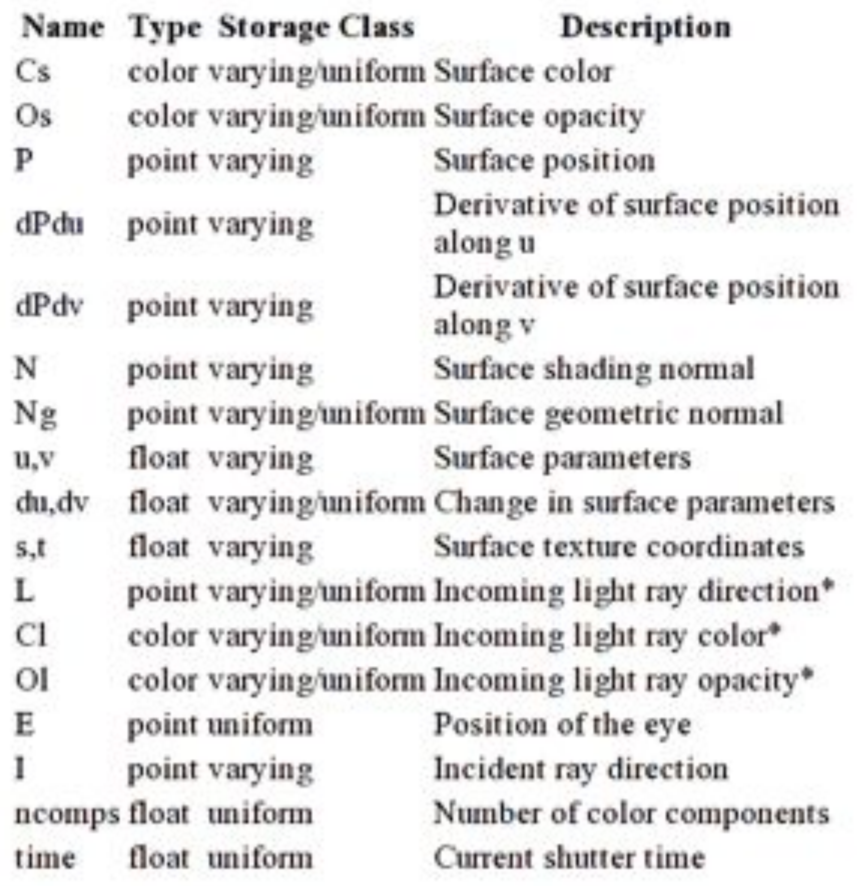
Surface Shader Environment

- Expected results: incident ray colour and opacity



Surface Shader

Surface Shader Vars

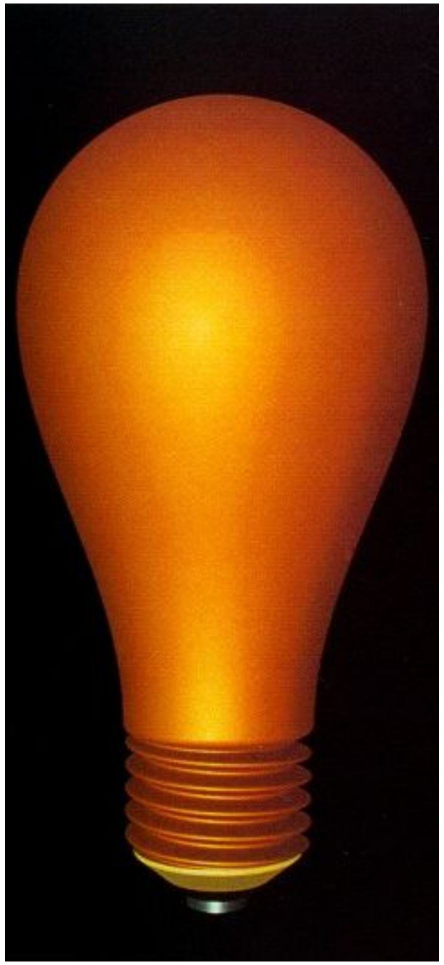


Name	Type	Storage Class	Description
Cs	color	varying/uniform	Surface color
Os	color	varying/uniform	Surface opacity
P	point	varying	Surface position
dPdu	point	varying	Derivative of surface position along u
dPdv	point	varying	Derivative of surface position along v
N	point	varying	Surface shading normal
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*
Ol	color	varying/uniform	Incoming light ray opacity*
E	point	uniform	Position of the eye
I	point	varying	Incident ray direction
ncomps	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

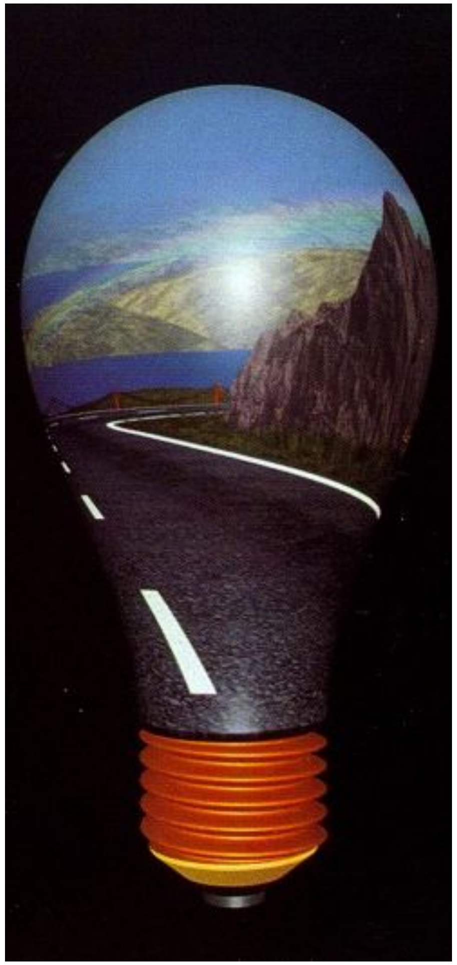
Example: Surface Shader #1

```
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));
}
```

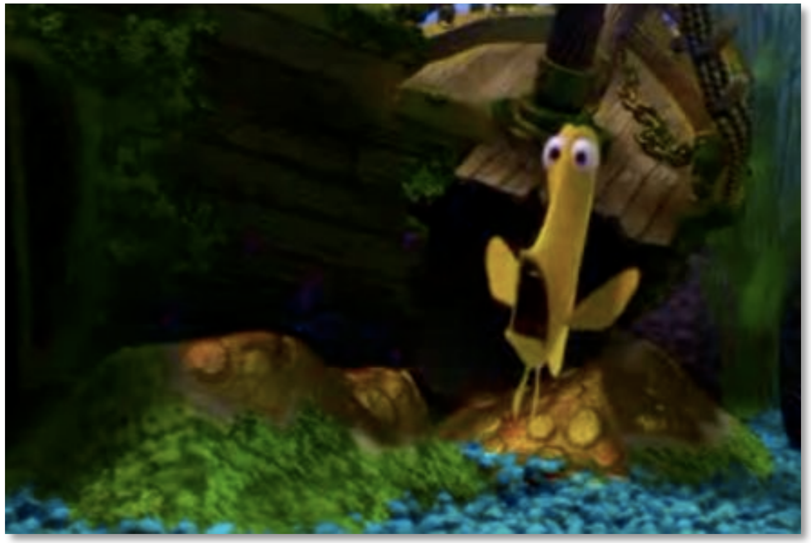


Example: Surface Shader #2

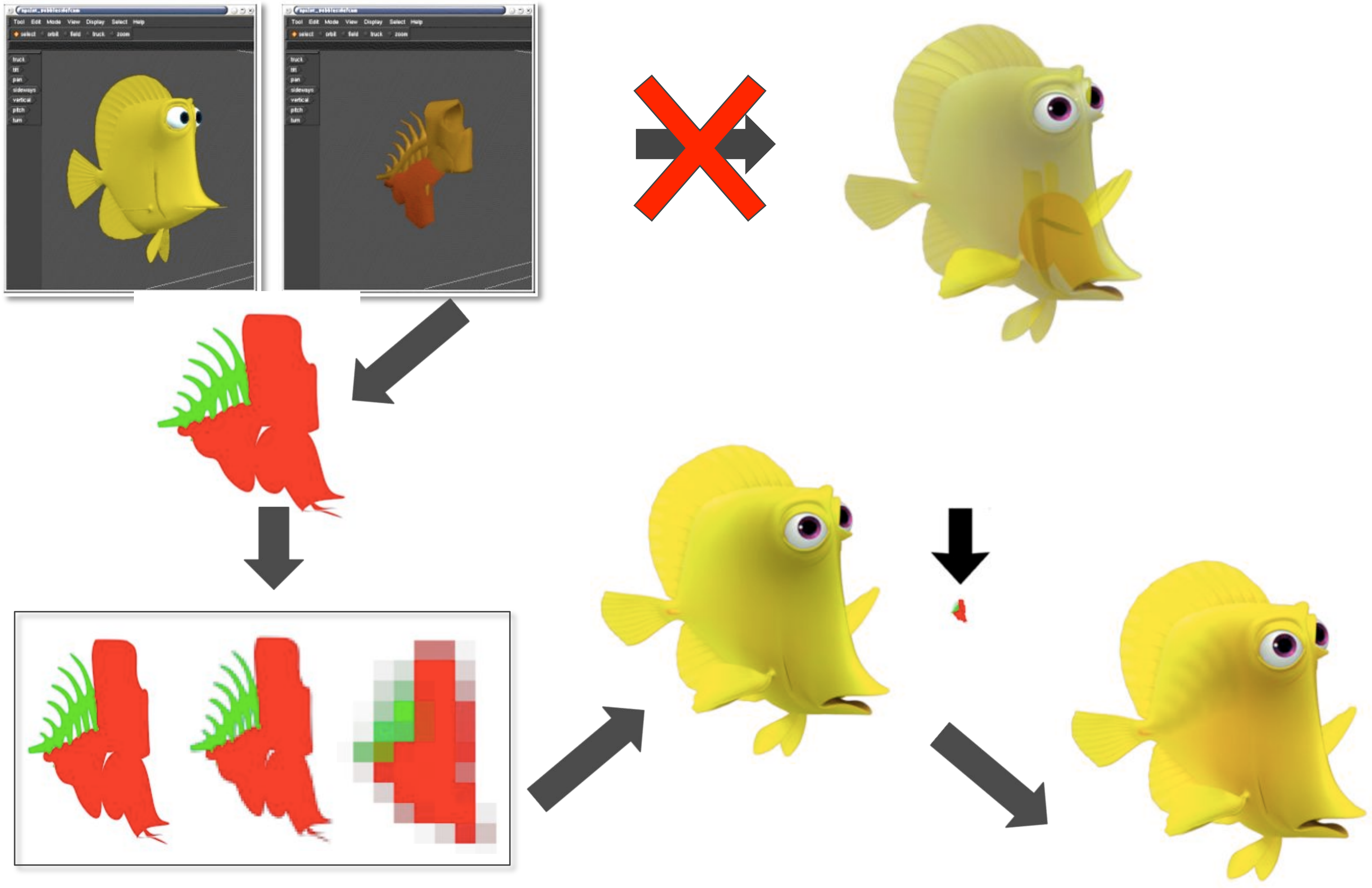
```
surface txtplastic(
    float Ka = 1;
    float Kd = .5;
    float Ks = .5;
    float roughness = .1;
    color specularcolor = 1;
    string mapname = “”)
{
    point Nf = faceforward(N,I);

    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*

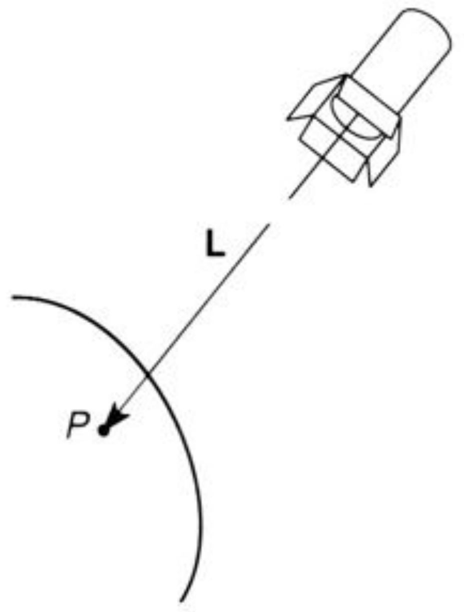


Example: Shading Fish Guts

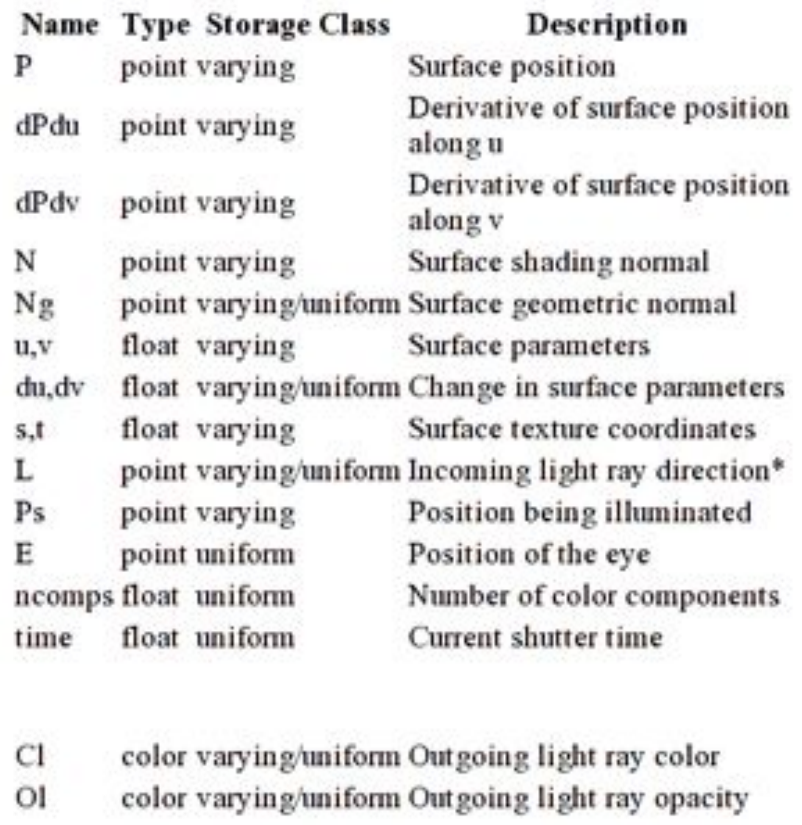


Light Source Shader

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



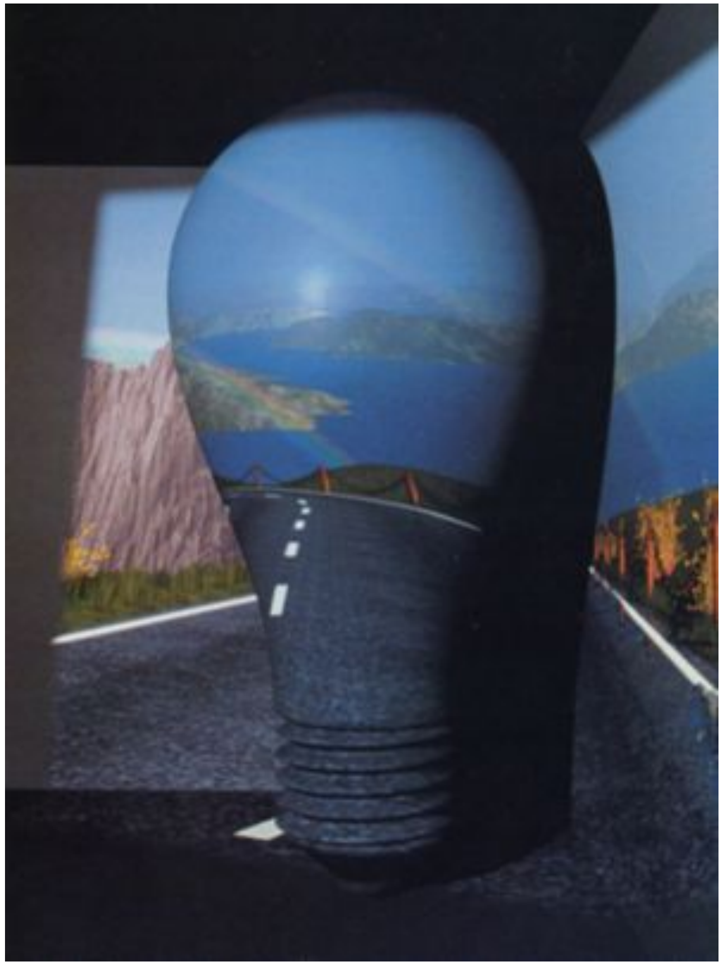
Light Source Shader Vars



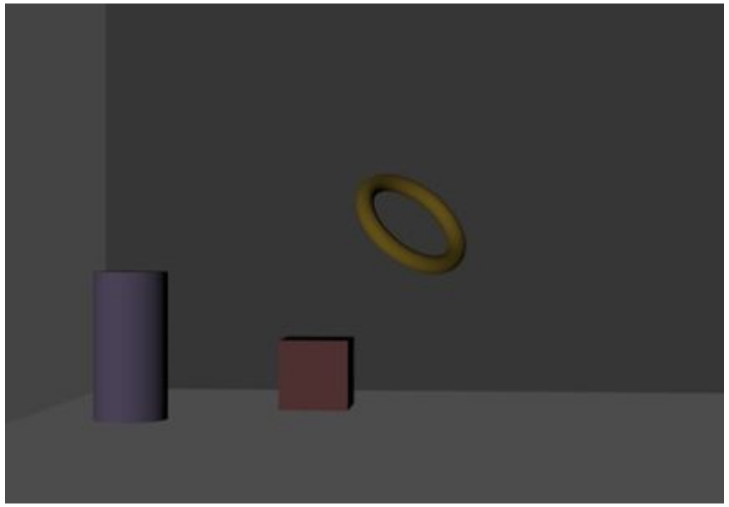
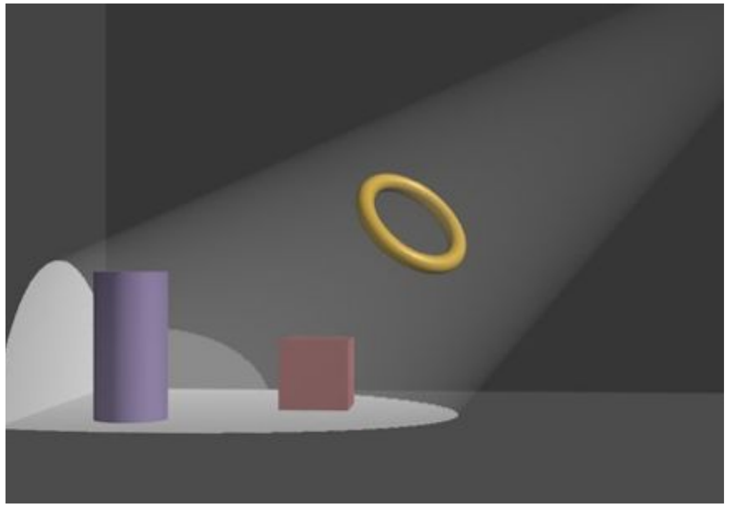
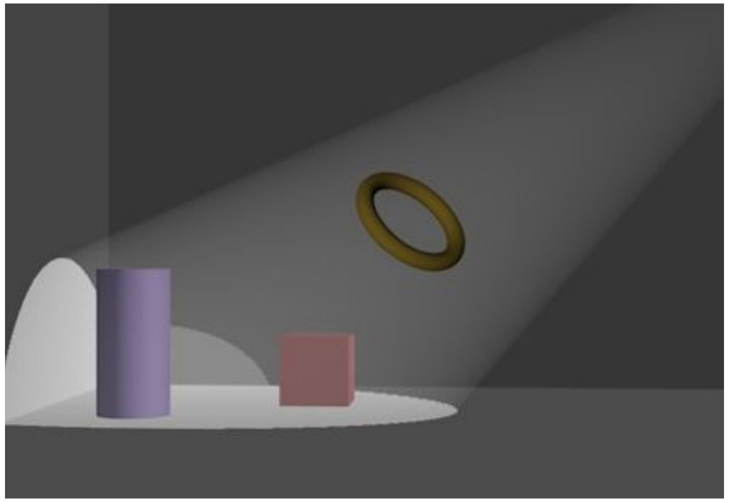
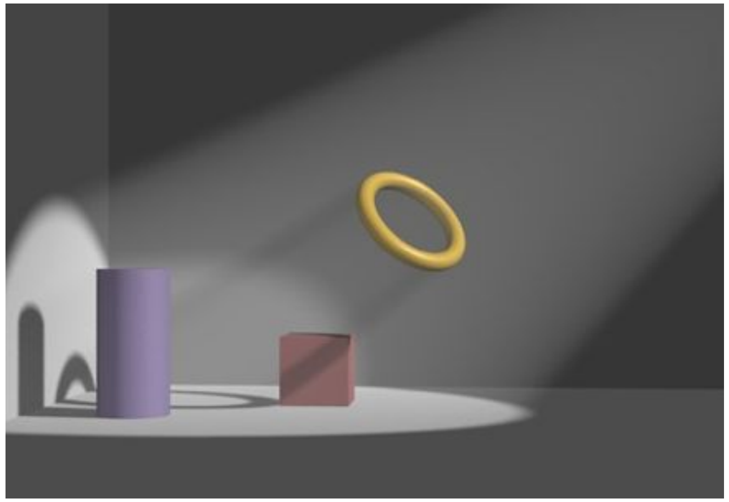
Name	Type	Storage Class	Description
P	point	varying	Surface position
dPdu	point	varying	Derivative of surface position along u
dPdv	point	varying	Derivative of surface position along v
N	point	varying	Surface shading normal
Ng	point	varying/uniform	Surface geometric normal
u,v	float	varying	Surface parameters
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
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
Cl	color	varying/uniform	Outgoing light ray color
Ol	color	varying/uniform	Outgoing light ray opacity

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

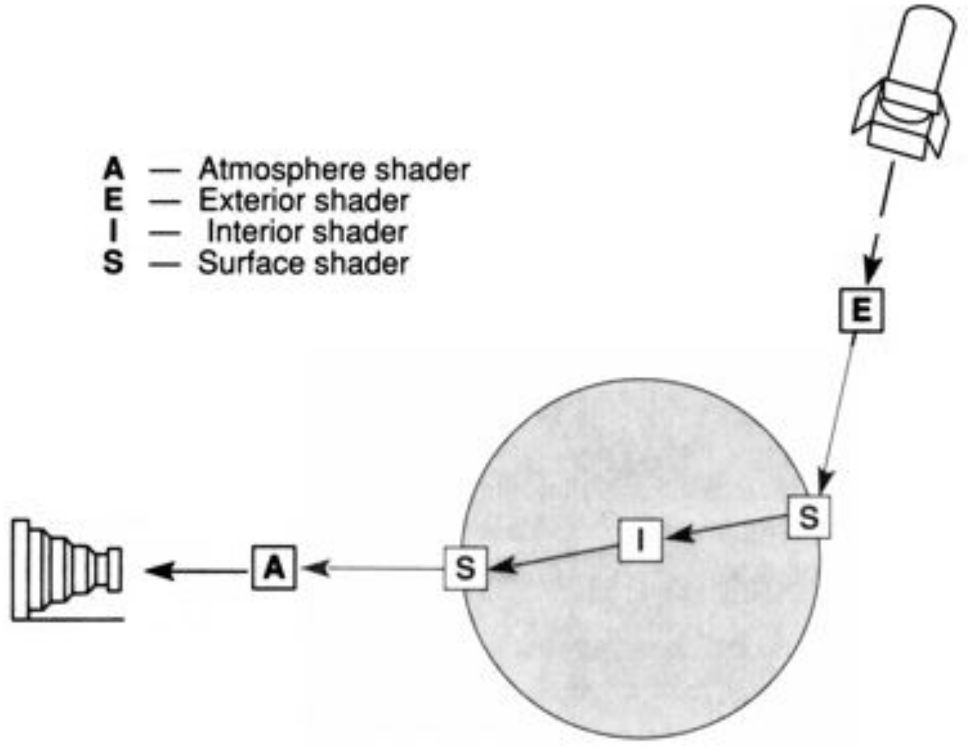
Example: Slideprojector



Flexible Lightsource Examples

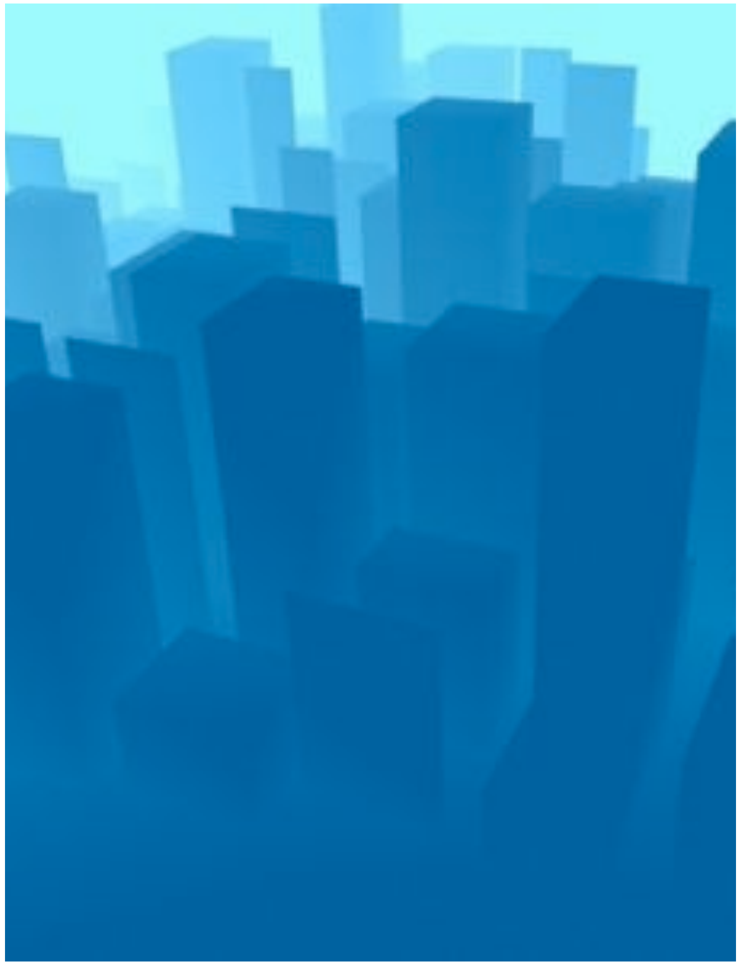
Volume (Atmosphere) Shader



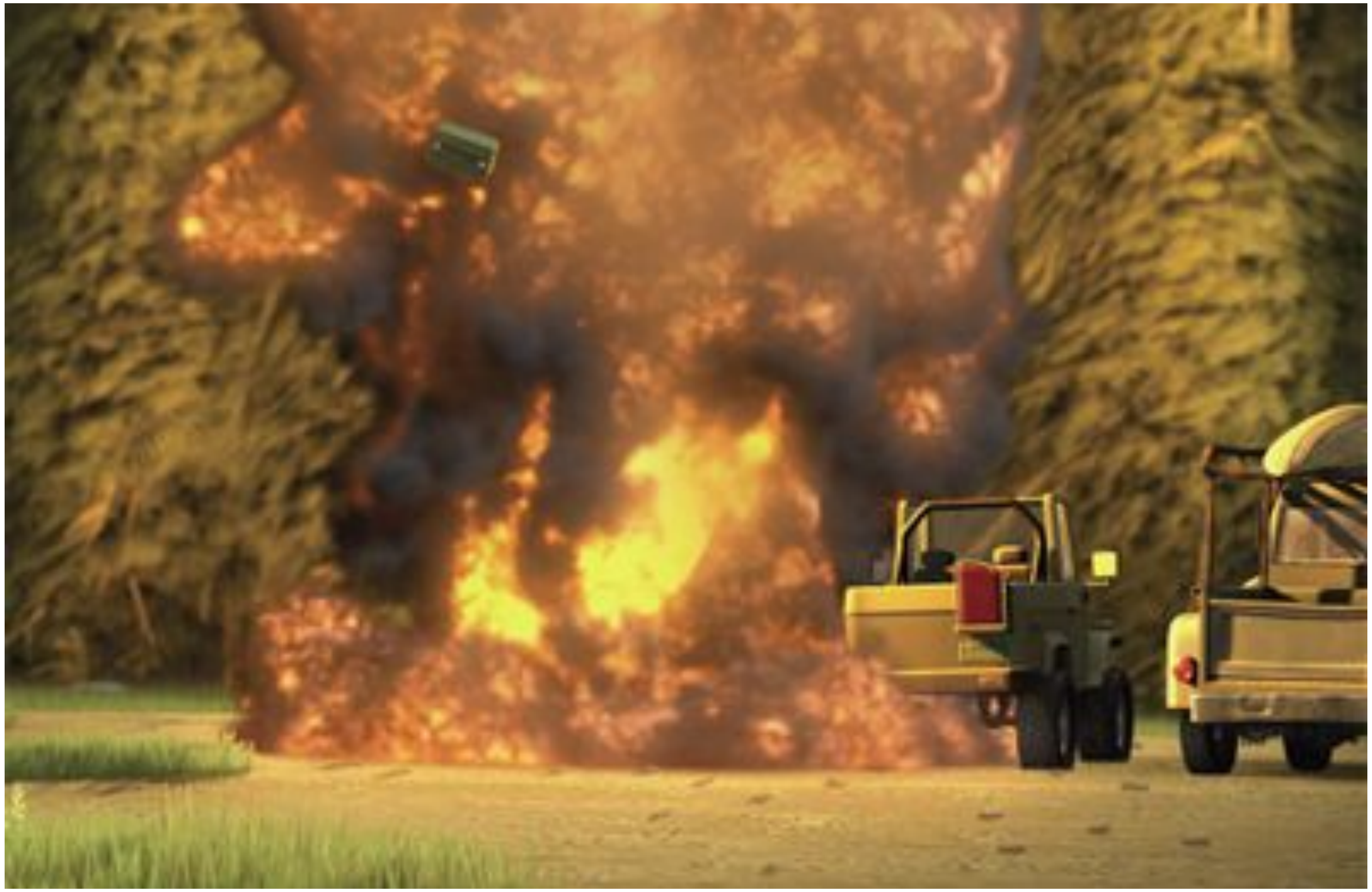
- Generalizes the idea of atmosphere affecting light passing through space between a surface and the eye
- (Not really supported until very recently)

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);
       }

    Ci = inten*mix(Ci,c,mixf);
    Oi = mix( Oi, color (1,1,1), d );
}// underwater()
```



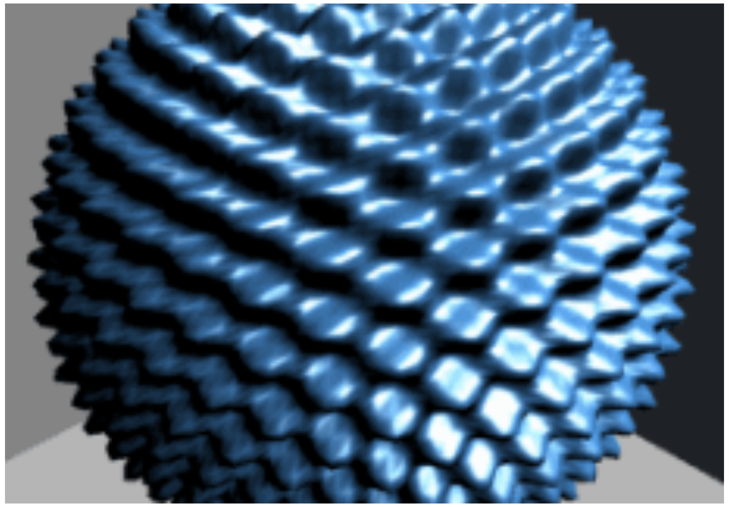
Volumetric Shader in *Open Season*

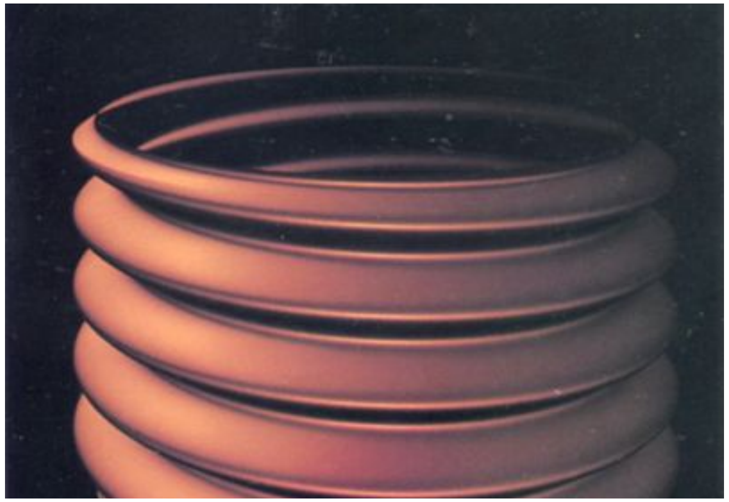


© 2006 Sony Pictures Imageworks

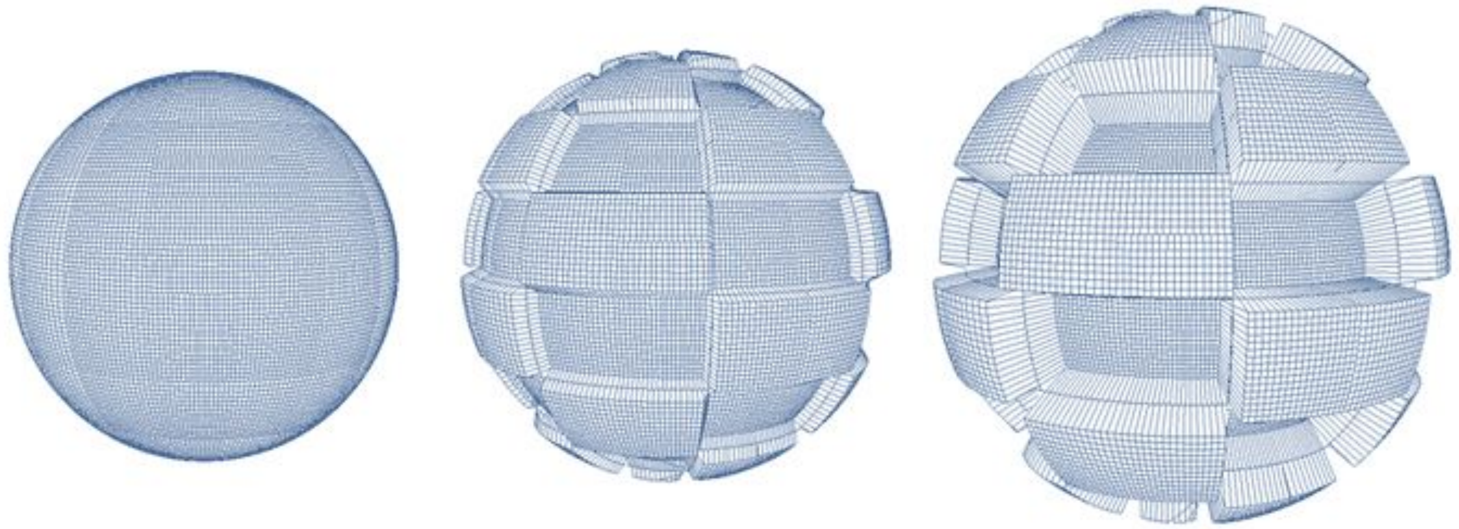
Displacement Shaders

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

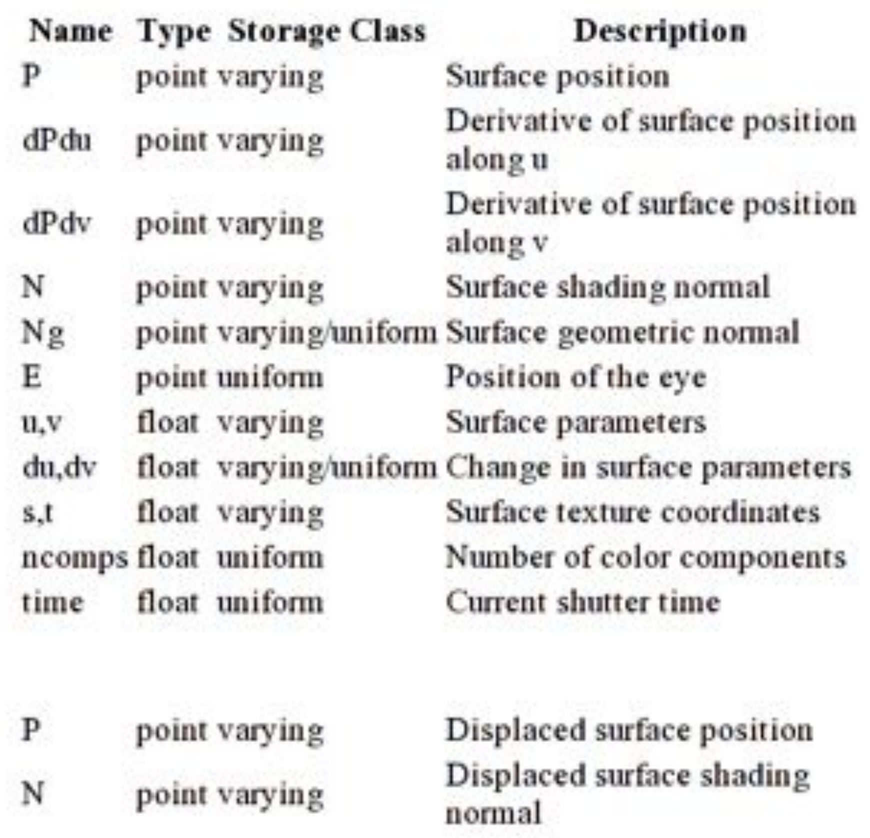




Displacement Shader Geometry



Displacement Vars

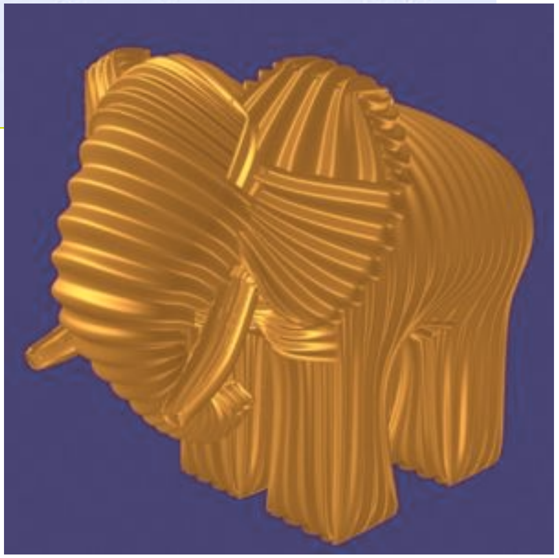


Name	Type	Storage Class	Description
P	point	varying	Surface position
dPdu	point	varying	Derivative of surface position along u
dPdv	point	varying	Derivative of surface position along v
N	point	varying	Surface shading normal
Ng	point	varying/uniform	Surface geometric normal
E	point	uniform	Position of the eye
u,v	float	varying	Surface parameters
du,dv	float	varying/uniform	Change in surface parameters
s,t	float	varying	Surface texture coordinates
ncomps	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

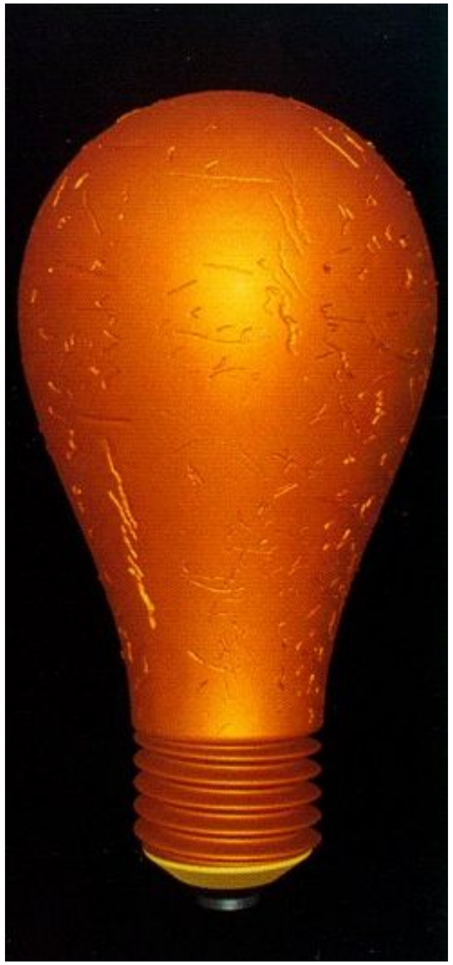
Displacement Shader Example #1

```
displacement sinewaves(float freq=1.0, ampl=1.0, sphase=0, tphase=0, paramdir=0)
{
  // displace along normal, using sin(s) or sin(t) or both
  if(0==paramdir)
    {
      P += ampl*sin(sphase+s*freq*2*PI)*normalize(N);
    }
  else if (1==paramdir)
    {
      P += ampl*sin(tphase+t*freq*2*PI)*normalize(N);
    }
  else
  {
      P += ampl*sin(sphase+s*freq*2*PI)*sin(tphase+t*freq*2*PI)*normalize(N);
  }
  N = calculatenormal(P);
}// sinewaves
```



Displacement Shader Example #2

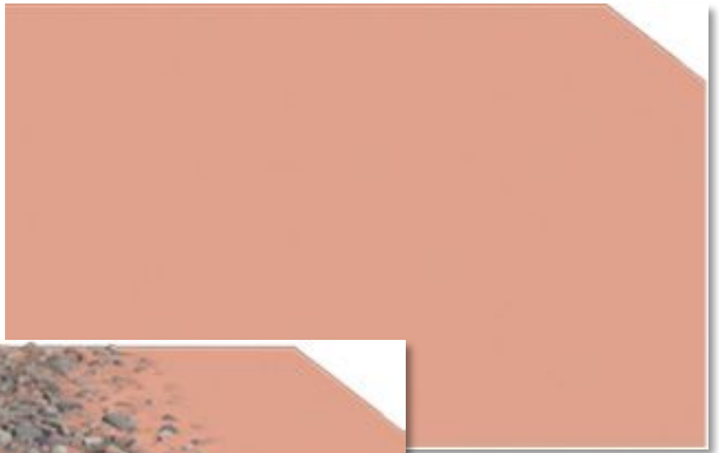
```
Displacement pits(
    float Km = 0.03;
    string mapname = "")
{
    float magnitude;

    if (marks!= "")
        magnitude = float texture(marks);
    else
        magnitude = 0;

    P += -Km * magnitude * normalize(N);
    N = calculatenormal(P);
}
```

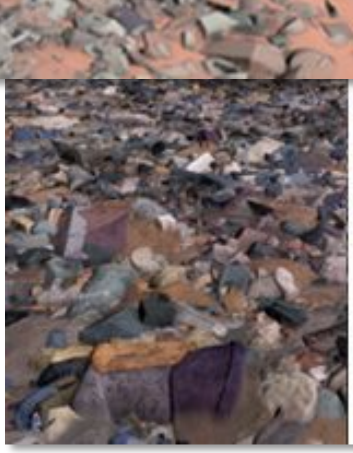


River Raindrops in *Ratatouille*

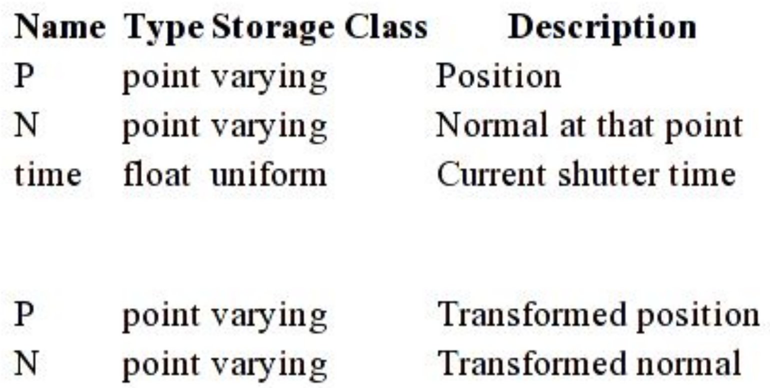
© Disney / Pixar.

WALL-E Trash Shader





Transformation Shaders

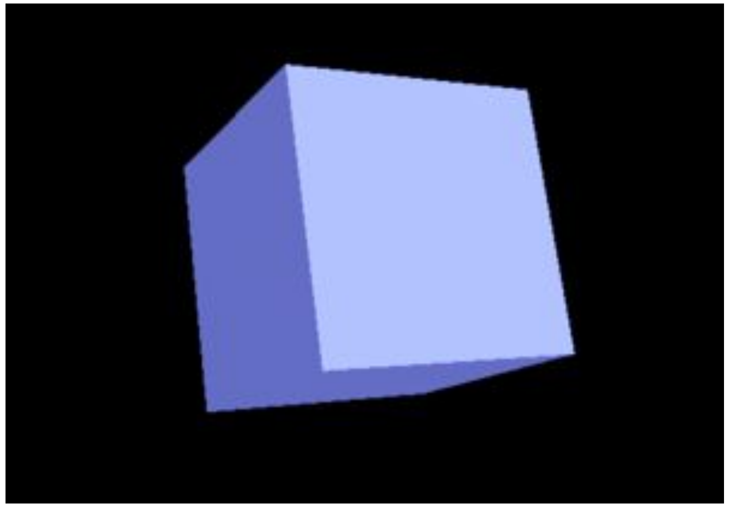


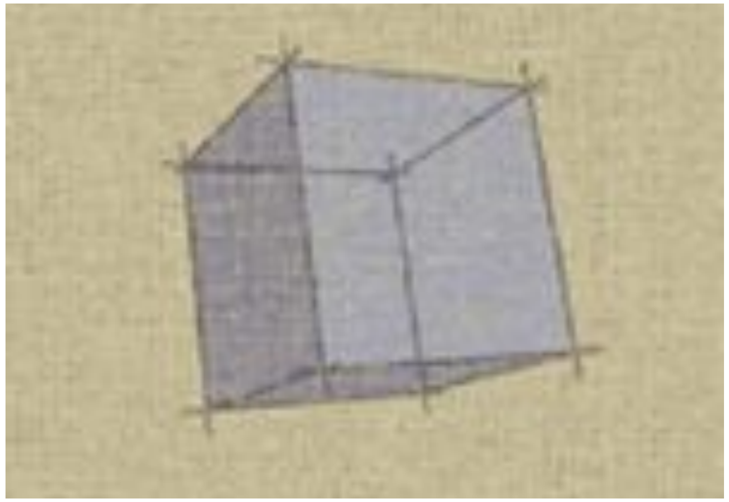
Name	Type	Storage Class	Description
P	point	varying	Position
N	point	varying	Normal at that point
time	float	uniform	Current shutter time
P	point	varying	Transformed position
N	point	varying	Transformed normal

- 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
- (not supported?)

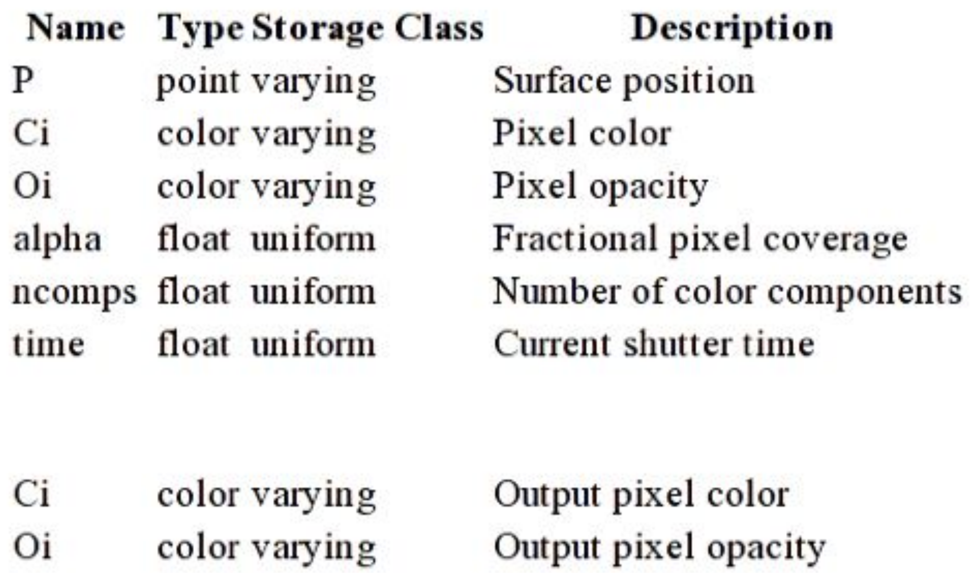
Imager Shaders

- 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

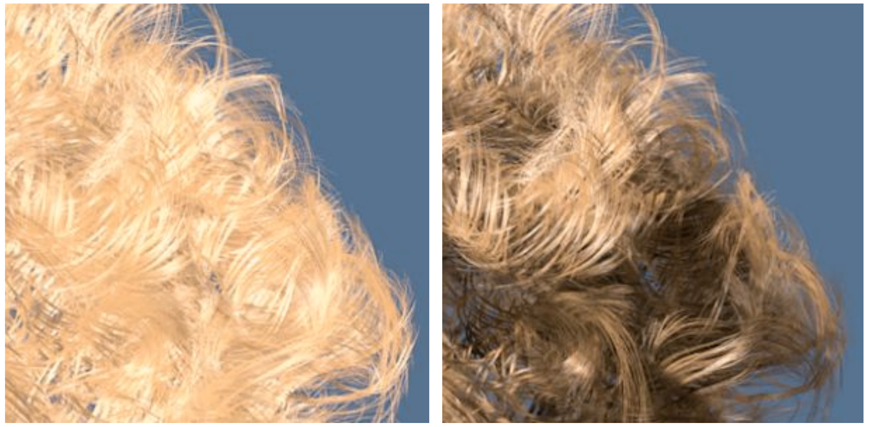
– Only colour information and z-Buffer data are provided

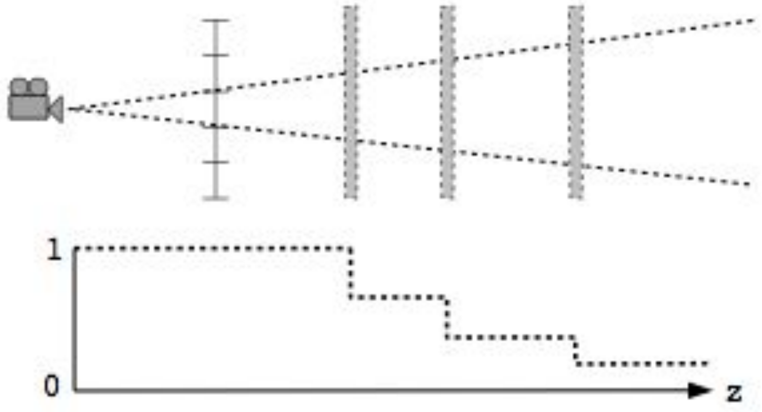
Shadows: A Sore Point

- 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 fratcional 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





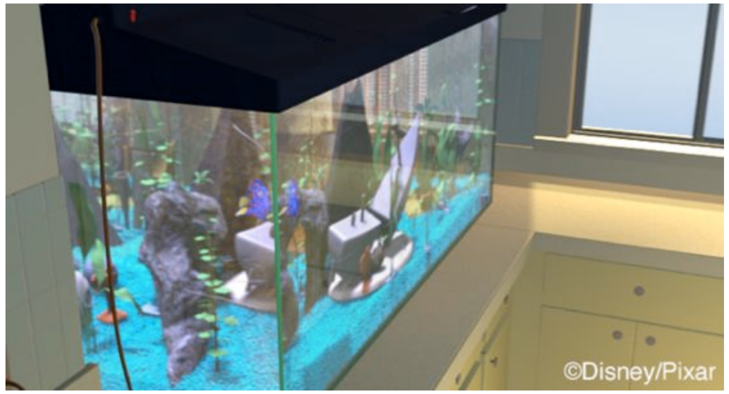
DSM: King Kong

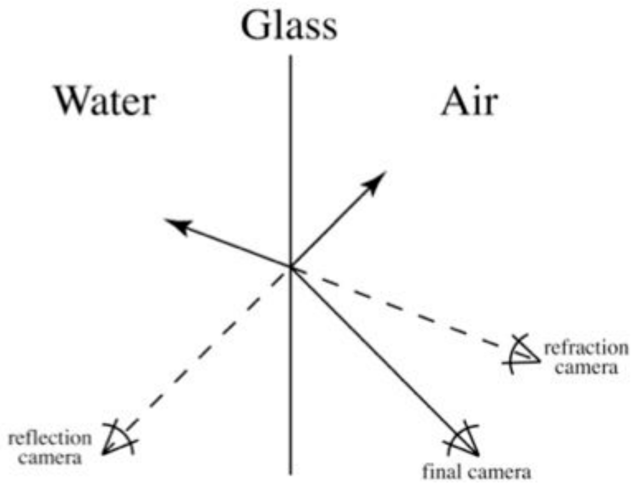
Reflections: Also A Sore Point

- 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

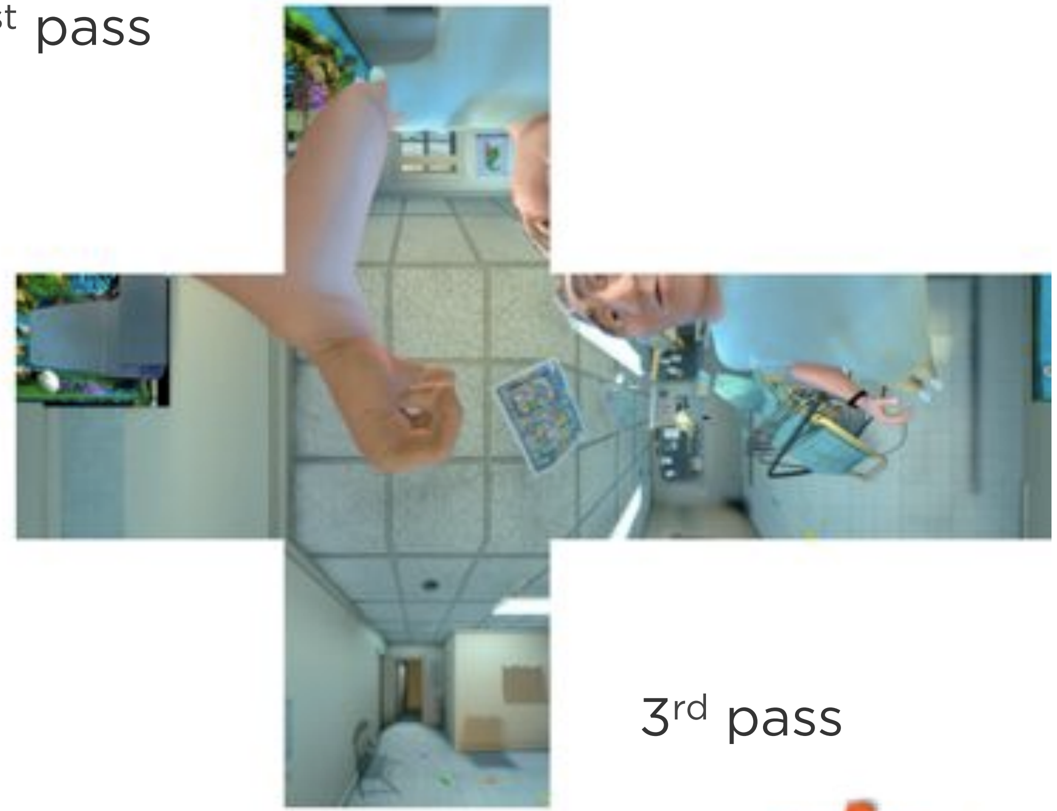
- Multi-pass rendering
- Reflection and refraction images are imput for tank shader
- Each wall has unique pair of reflection and refraction camera
- Texture is projected





Example: Reflection Map

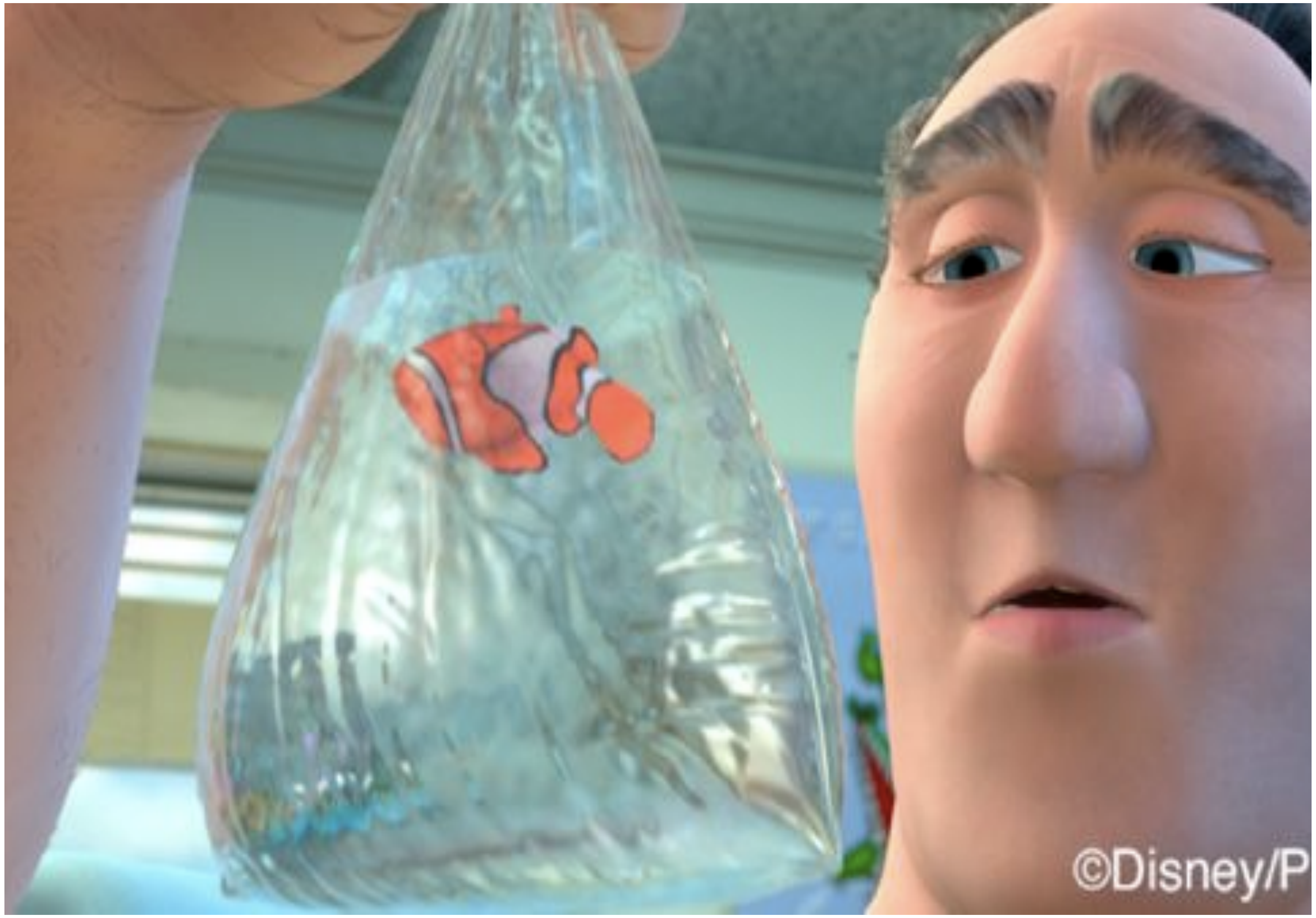
1st pass



2nd pass

3rd 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 capabilites
- BMRT
  - Raytracer, has disappeared after legal action was taken against author
- Pixie
  - Open-source RenderMan
- 3Delight
- Realtime techniques
  - Ongoing research topic

The End
Thank you for your attention!