

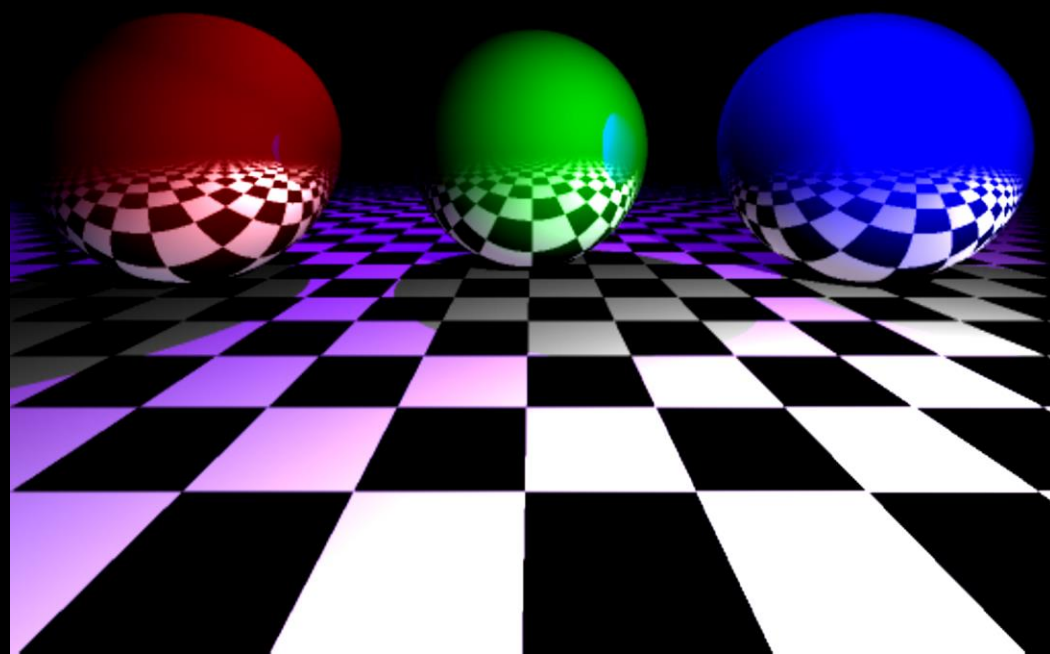
INFOGR – Computer Graphics

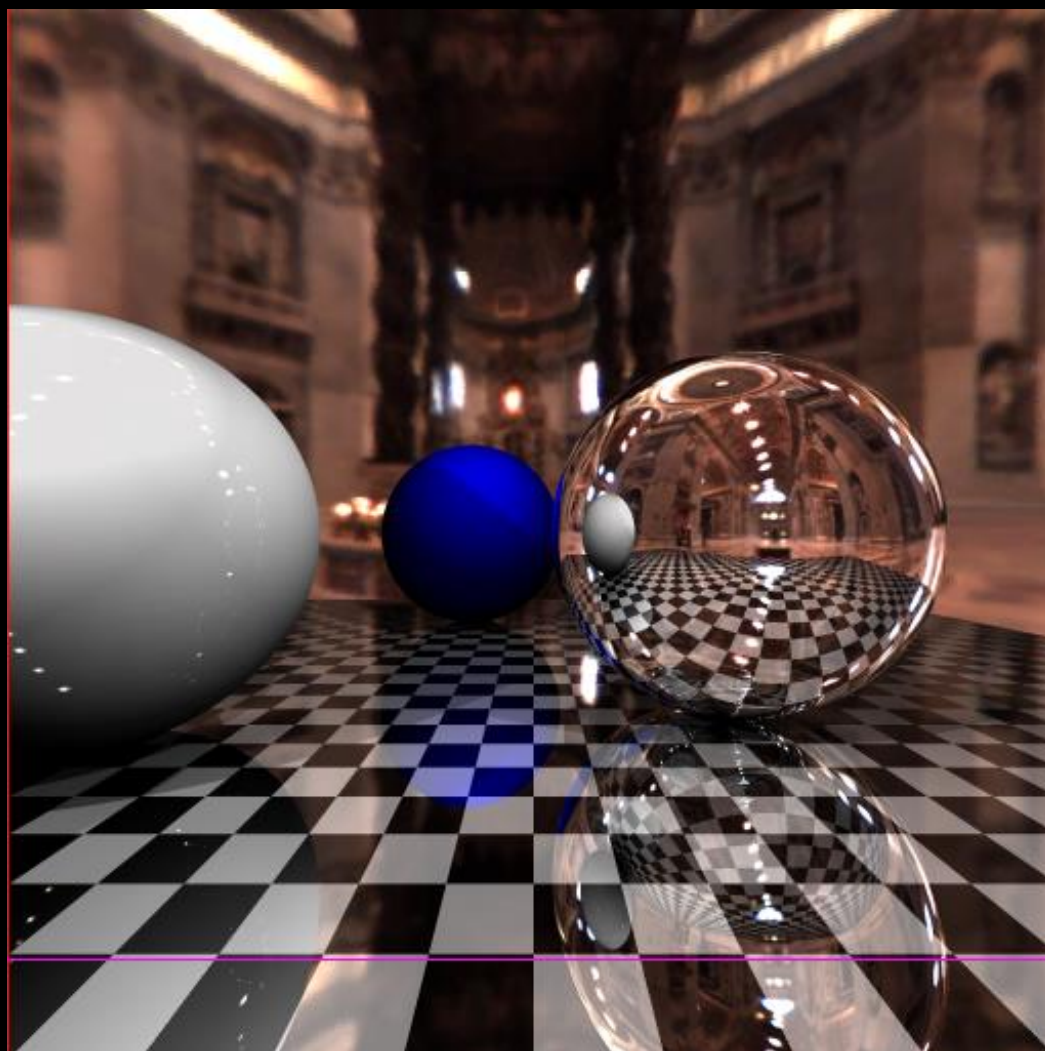
Jacco Bikker & Debabrata Panja - April-July 2017

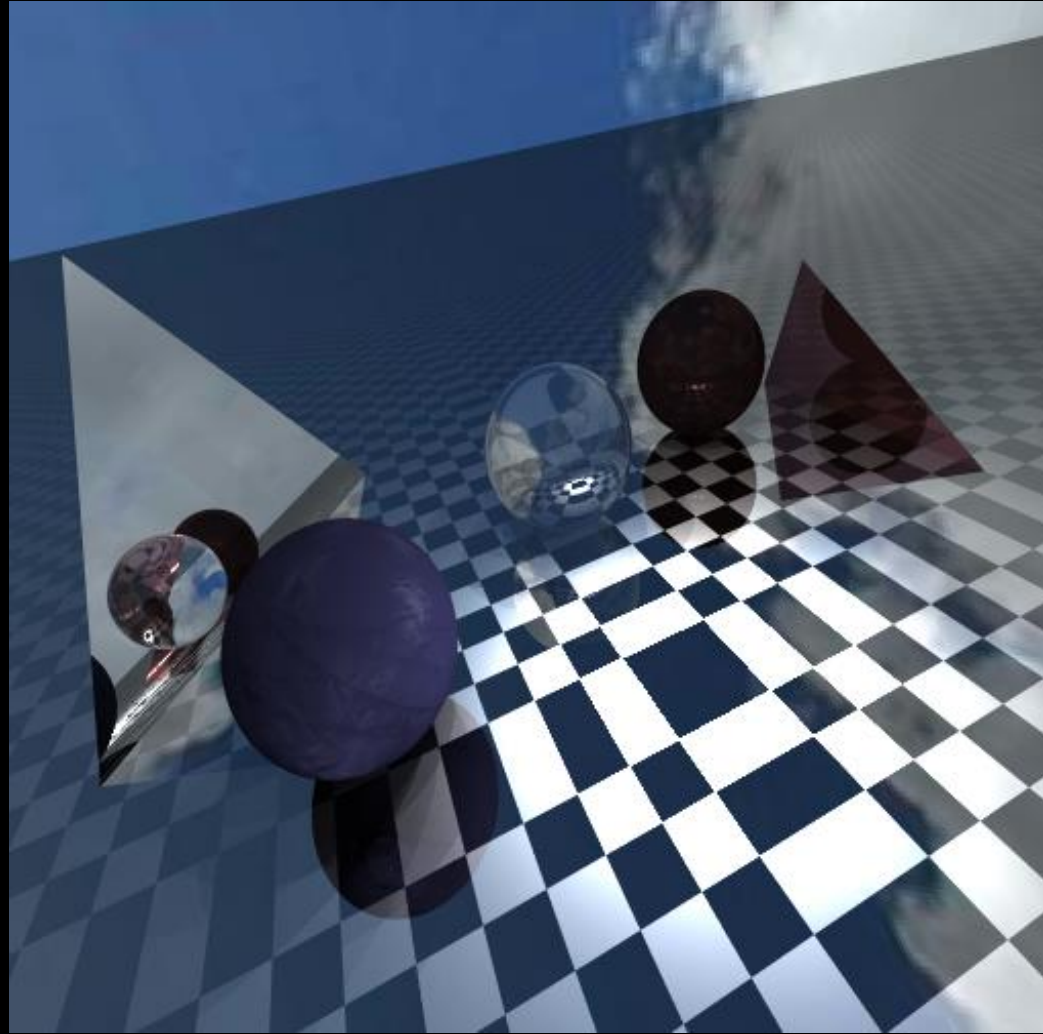
Lecture 10: “Shaders”

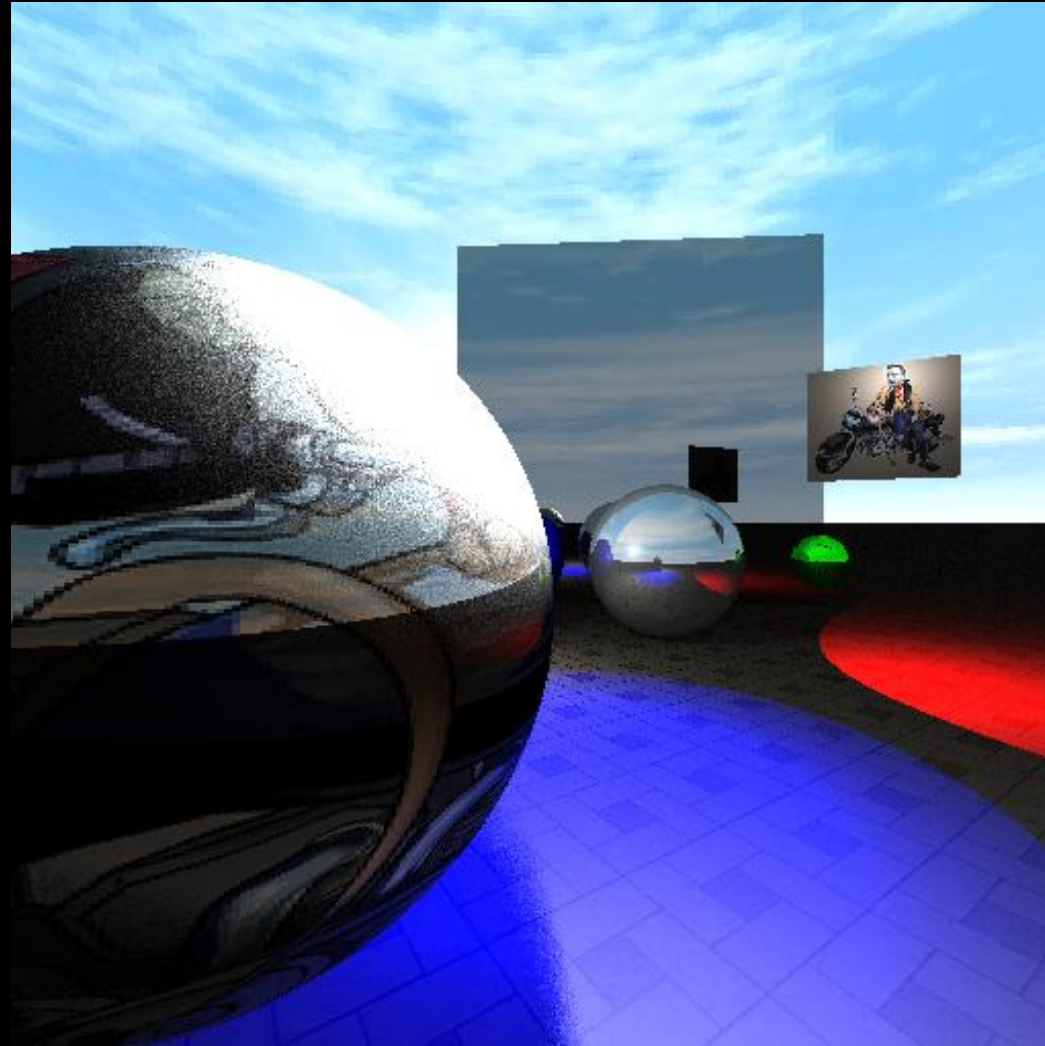
Welcome!

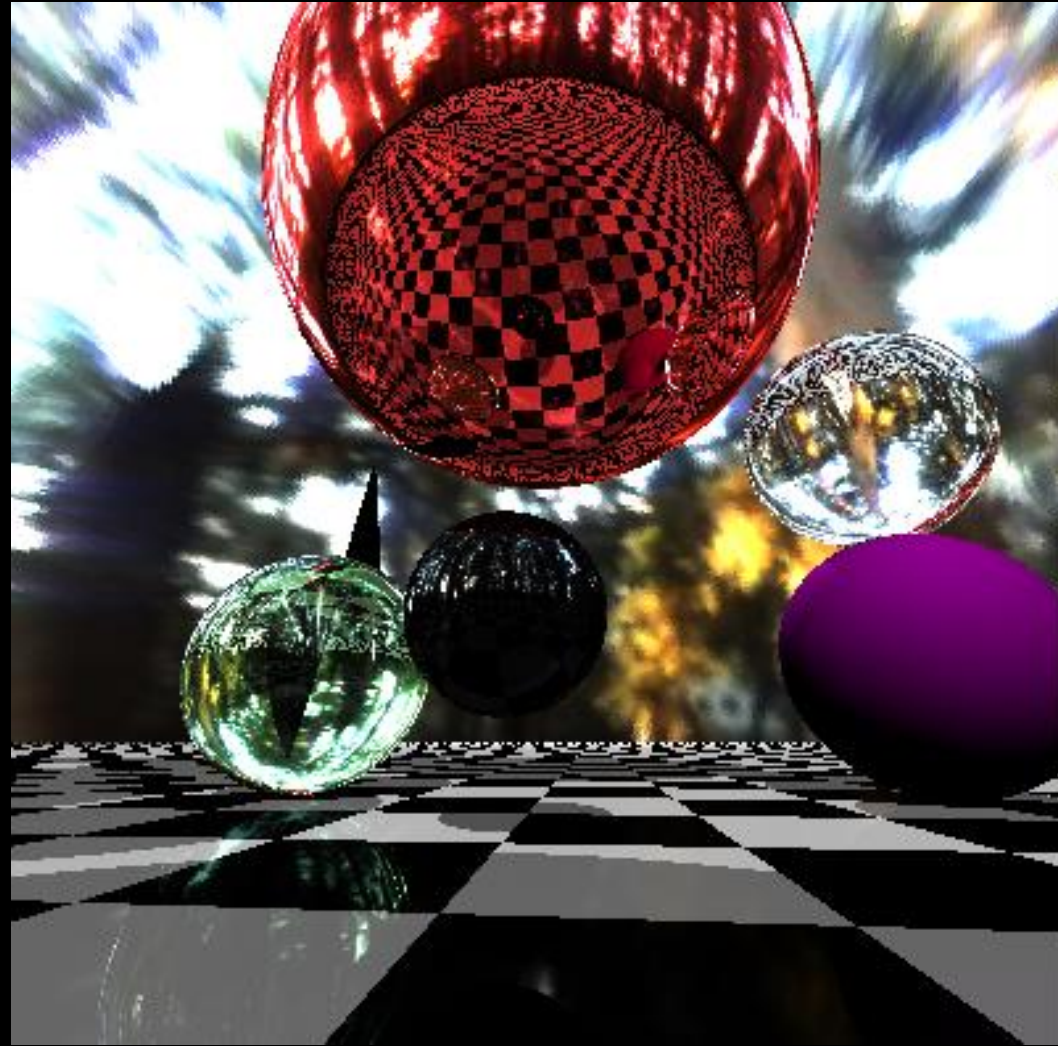




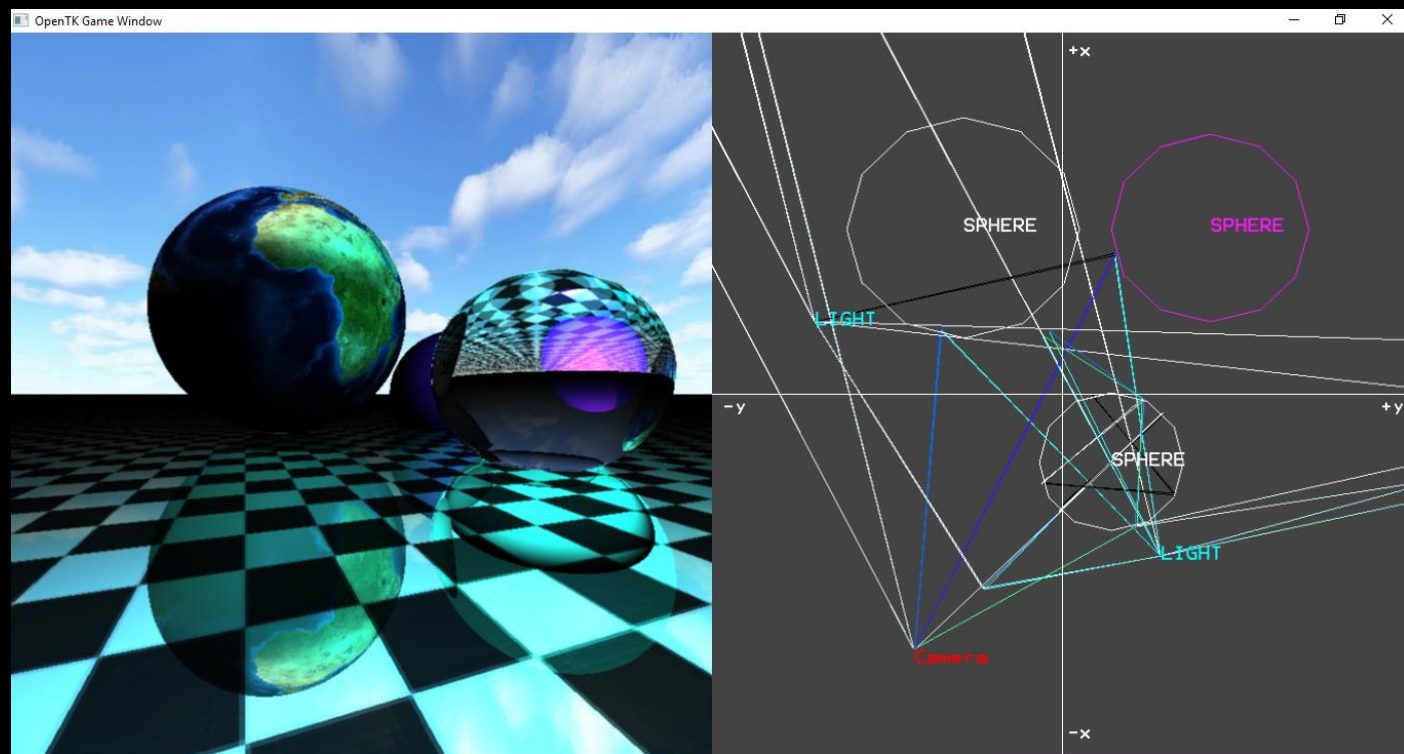


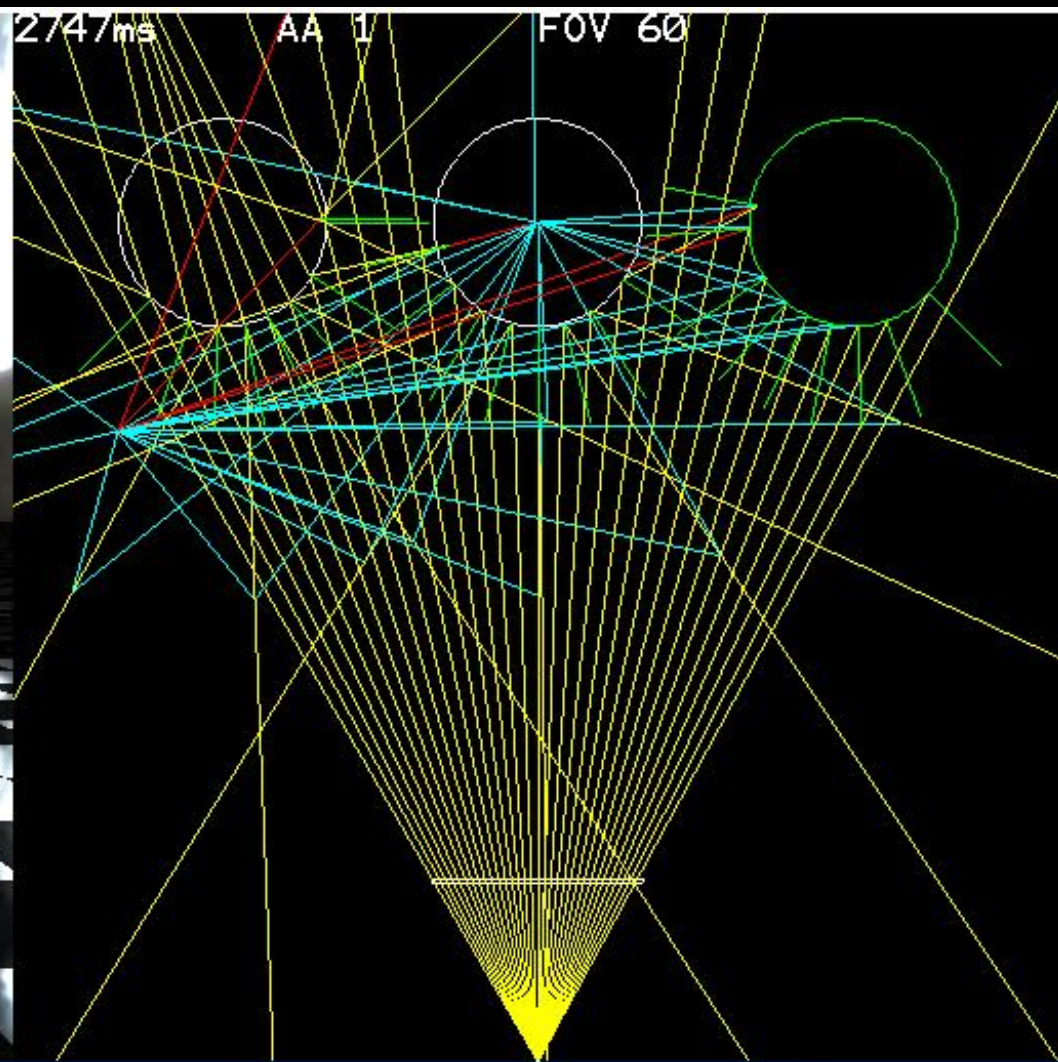
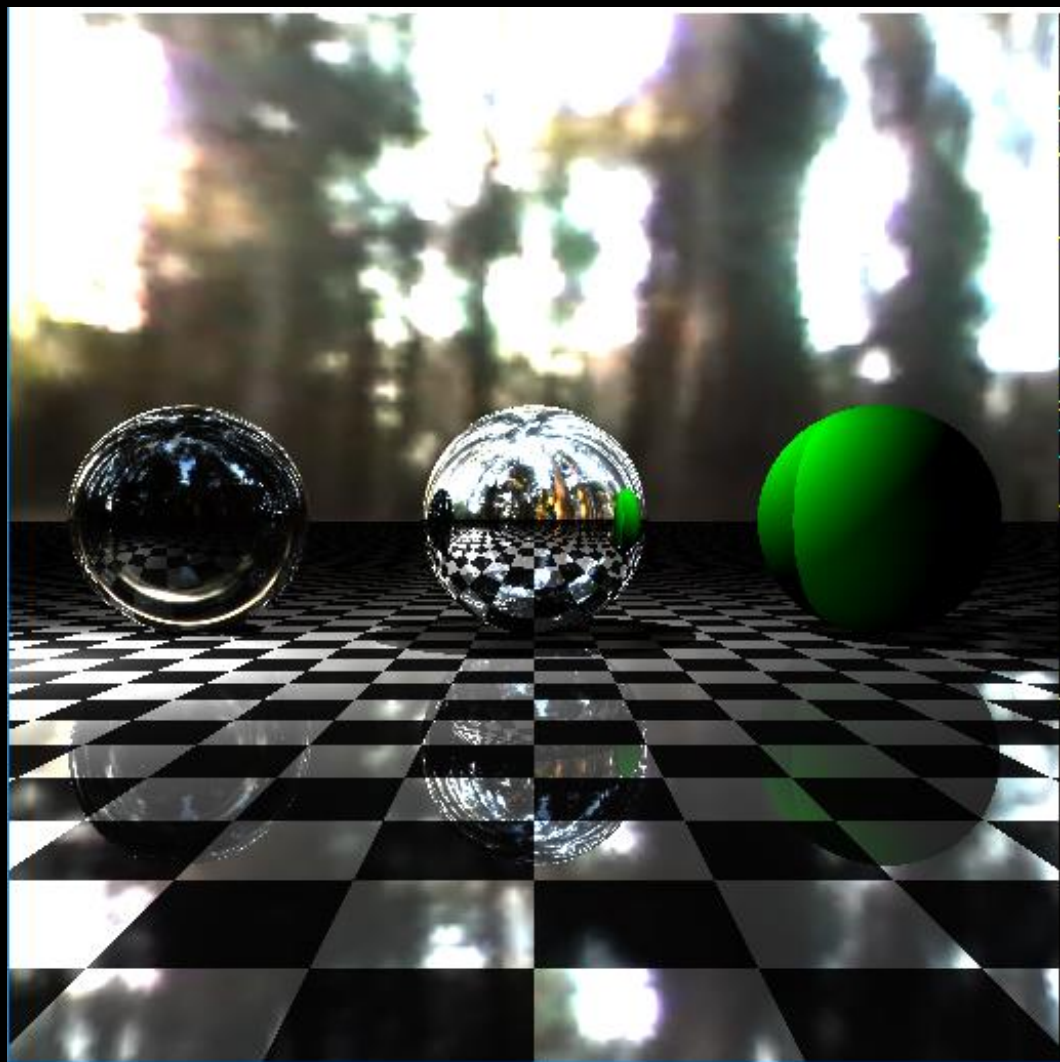


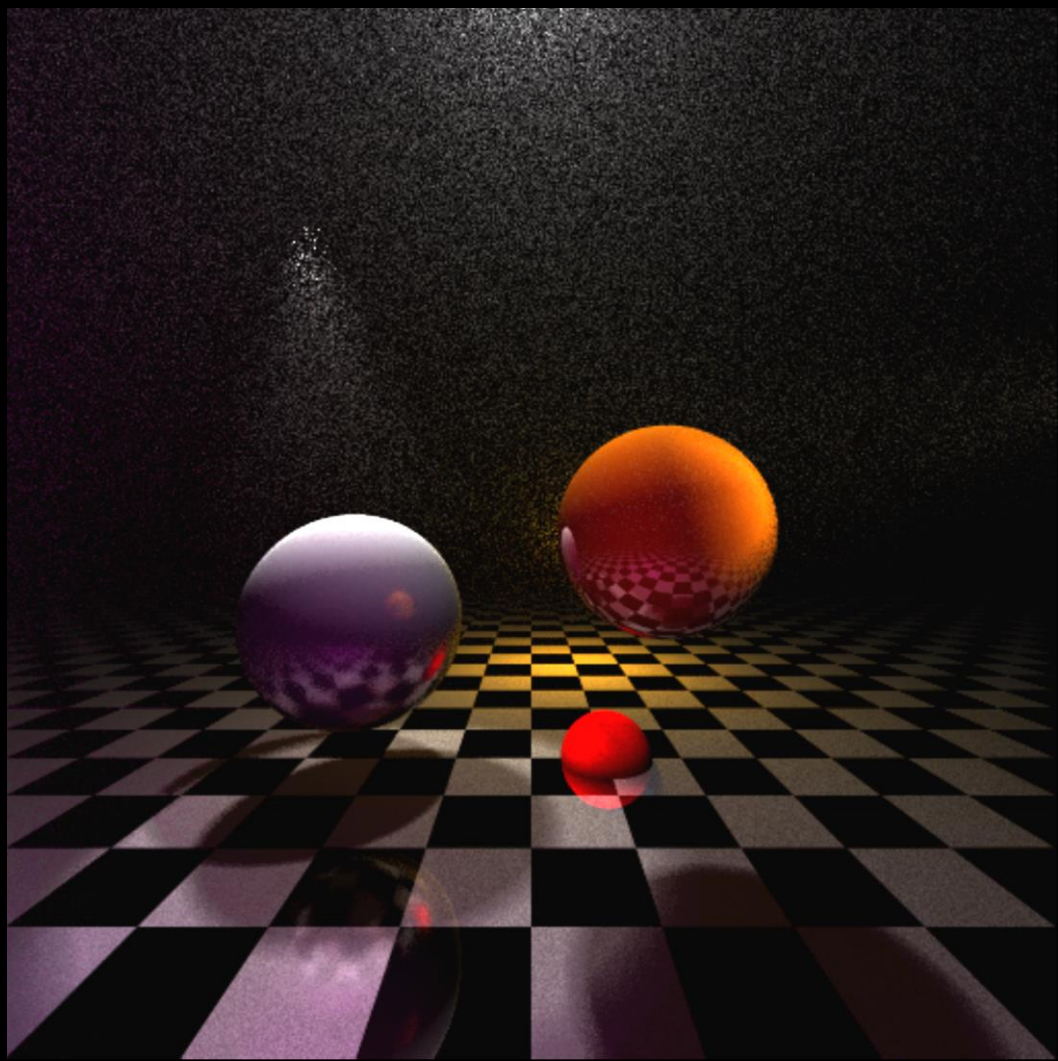


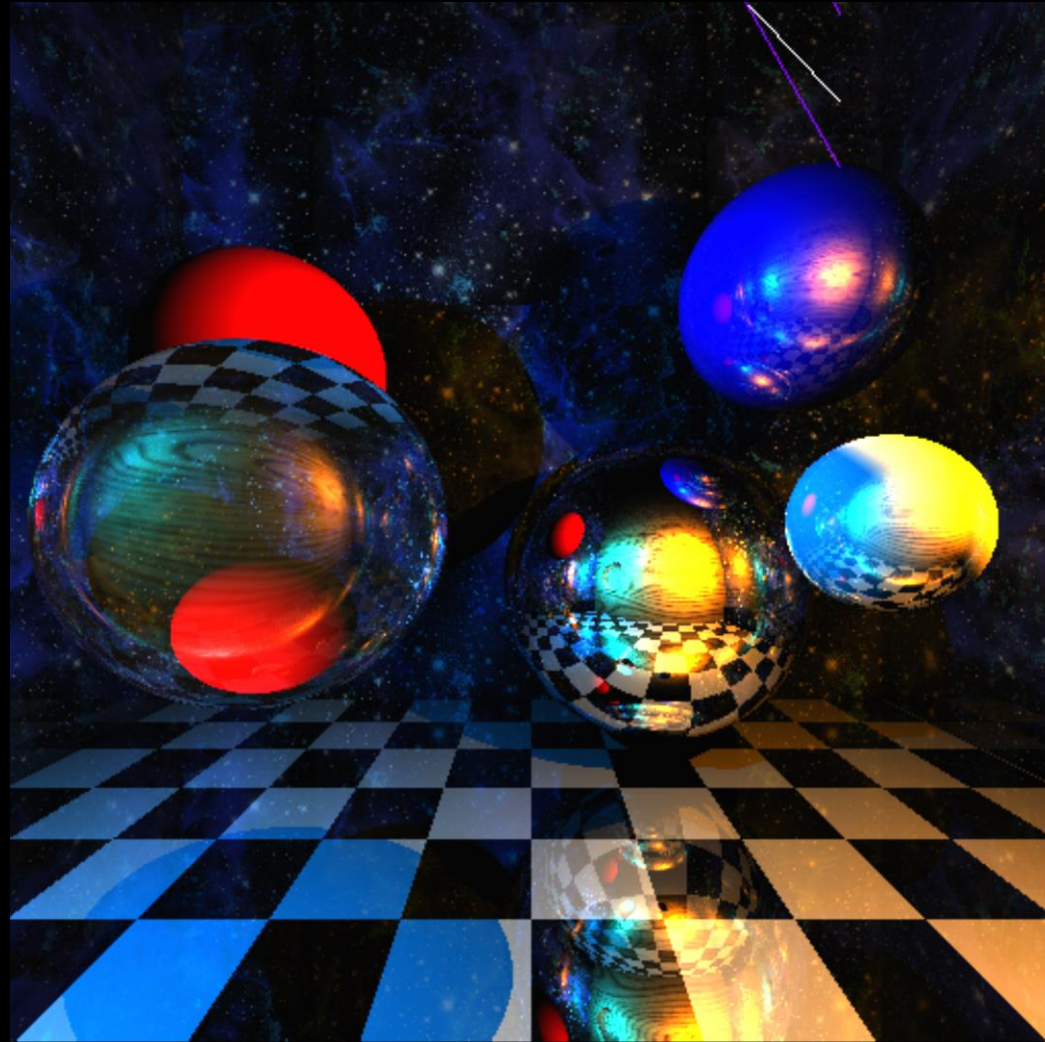


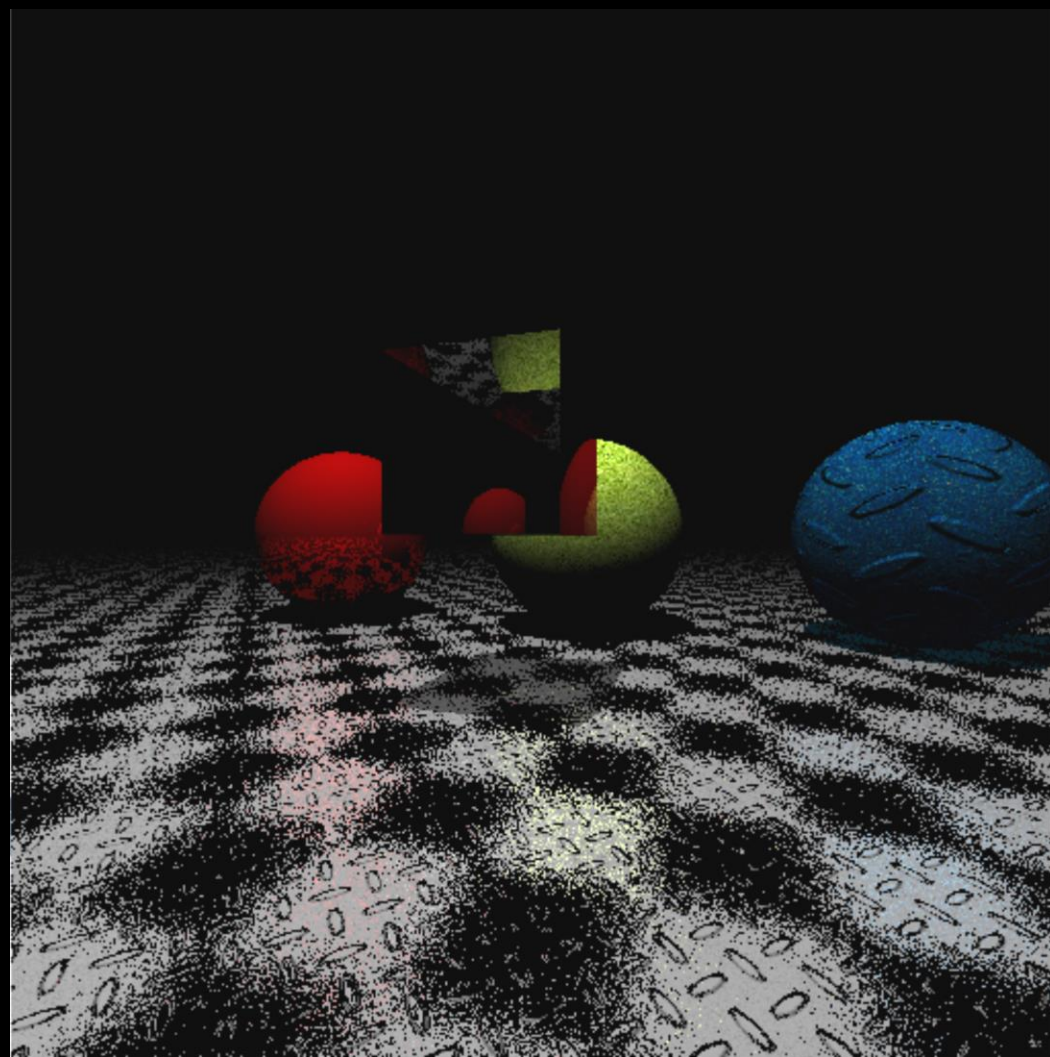














INFOGR2016/17

Today's Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur



Diffuse

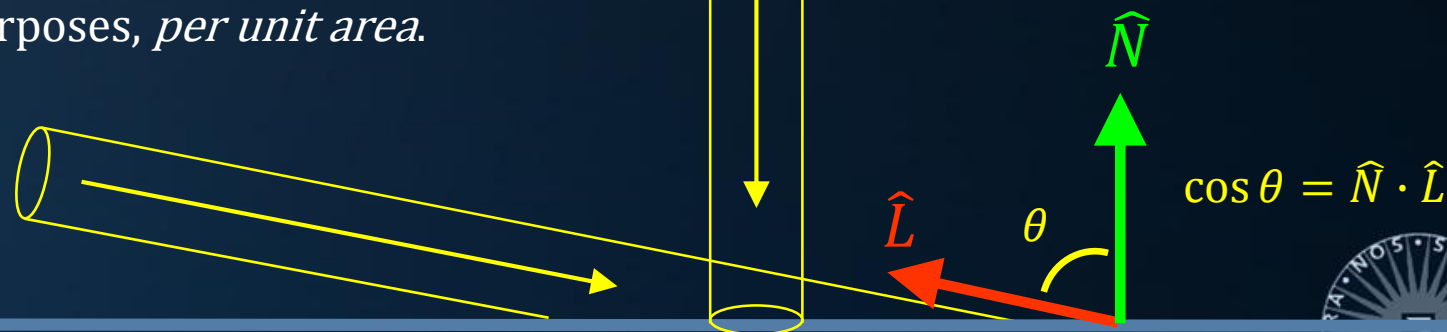
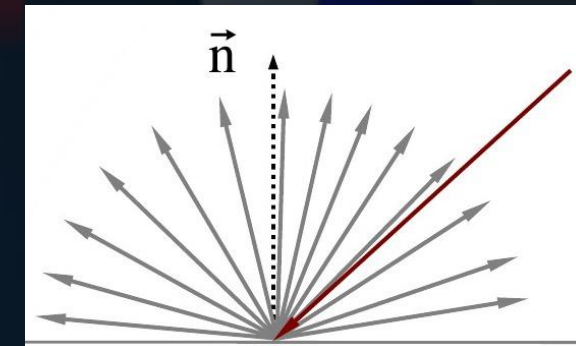
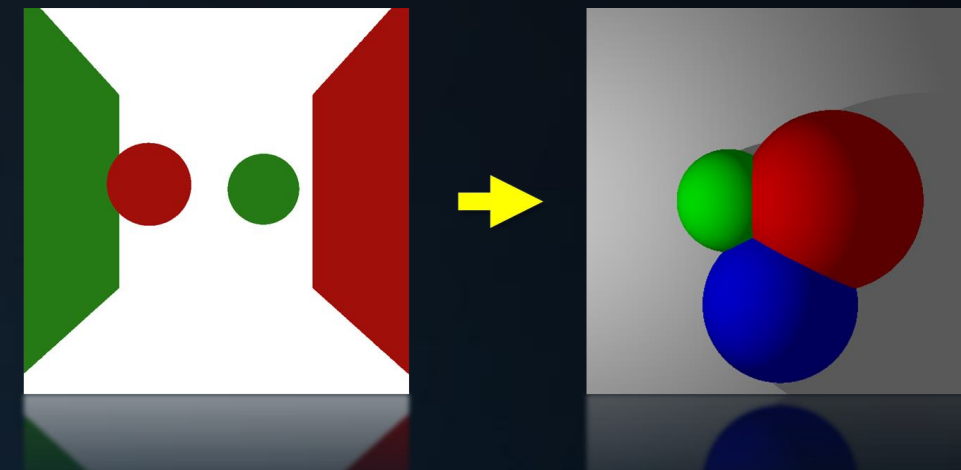
Basics of Shading

A diffuse material scatters incoming light equally in all directions.

Two aspects to this:

1. Diffuse materials are view-independent.
2. We need to know how much light is arriving.

Arriving: *irradiance*, expressed in Joules per second per m^2 . Or, for our purposes, *per unit area*.



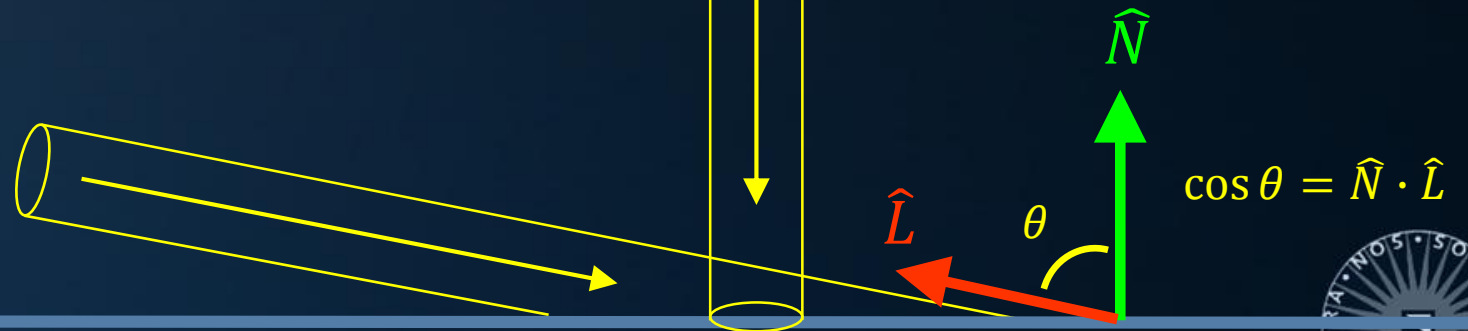
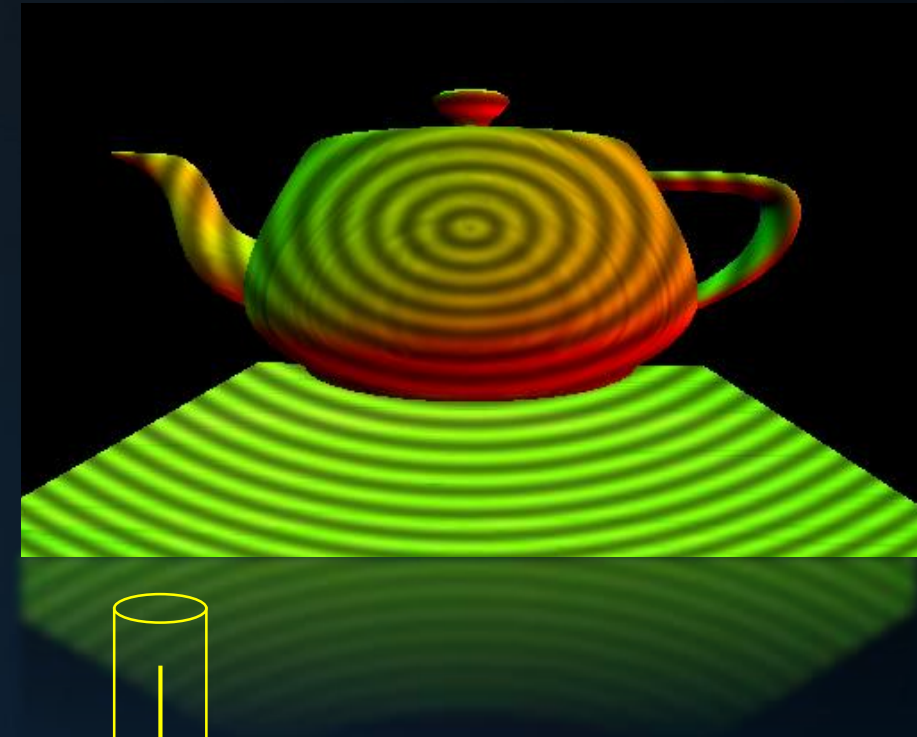
Diffuse

Basics of Shading

So, in the fragment shader:

- Calculate L : $L = \text{lightPos} - \text{intersectionPoint}$
- Use N : already passed via vertex shader
- Apply attenuation, scale by material color
- Multiply by light color
- Emit fragment color.

But wait...



Diffuse

Basics of Shading

So, in the fragment shader:

- Calculate L : $L = \text{lightPos} - \text{intersectionPoint}$
- Use N : already passed via vertex shader
- Apply attenuation, scale by material color
- Multiply by light color
- Emit fragment color.

But wait...

We have significant problems:

1. How do we specify a light position
2. In *which space* do we operate?

Ehm...

That one?

```
#version 330

// shader input
in vec2 vUV;
in vec3 vNormal;
in vec3 vPosition;

// shader output
out vec4 normal;
out vec2 uv;
uniform mat4 transform;

// vertex shader
void main()
{
    // transform vertex using supplied matrix
    gl_Position = transform * vec4( vPosition, 1.0 );

    // forward normal and uv coordinate
    normal = transform * vec4( vNormal, 0.0f );
    uv = vUV;
}
```

model space

model space
to
screen space

Diffuse

Spaces

Default matrix in game.cs:

```
Matrix4 transform = Matrix4.CreateFromAxisAngle( new Vector3( 0, 1, 0 ), a );
transform *= Matrix4.CreateTranslation( 0, -4, -15 );
transform *= Matrix4.CreatePerspectiveFieldOfView( 1.2f, 1.3f, .1f, 1000 );
```

This produces:

- a teapot that spins around it's pivot
- a camera located at (0, 4, 15) *or* the teapot spins at (0, -4, 15), camera is at (0, 0, 0).

The last line adds perspective.

➔ We need a *‘base system’* in which we can define a light position: *world space*.



Diffuse

Spaces

Getting model space coordinates to world space:

```
Matrix4 transform = Matrix4.CreateFromAxisAngle( new Vector3( 0, 1, 0 ), a );
Matrix4 toWorld = transform;
transform *= Matrix4.CreateTranslation( 0, -4, -15 );
transform *= Matrix4.CreatePerspectiveFieldOfView( 1.2f, 1.3f, .1f, 1000 );
```

We need some additional changes now:

```
public void Render(
    Shader shader,
    Matrix4 transform,
    Matrix4 toWorld,
    Texture texture
```

...



Diffuse

Changes

The vertex shader now takes two matrices:

```
// transforms
uniform mat4 transform; // full transform: model space to screen space
uniform mat4 toWorld;   // model space to world space
```

...and uses them:

```
gl_Position = transform * vec4( vPosition, 1.0 );
worldPos = toWorld * vec4( vPosition, 1.0f );
normal = toWorld * vec4( vNormal, 0.0f );
```



Diffuse

Changes

The shader class needs to know about the two matrices:

```
public int uniform_mview;
public int uniform_2wrld;
```

```
...
```

```
uniform_mview = GL.GetUniformLocation( programID, "transform" );
uniform_2wrld = GL.GetUniformLocation( programID, "toWorld" );
```

And the mesh class needs to pass both to the shader:

```
// pass transforms to vertex shader
GL.UniformMatrix4( shader.uniform_mview, false, ref transform );
GL.UniformMatrix4( shader.uniform_2wrld, false, ref toWorld );
```



Diffuse

Changes

The new fragment shader, complete:

```
#version 330

in vec2 uv;           // interpolated texture coordi
in vec4 normal;        // interpolated normal, world s
in vec4 worldPos;      // world space position of frag
uniform sampler2D pixels; // texture sampler

out vec4 outputColor; // shader output

uniform vec3 lightPos; // light position in world space

void main()           // fragment shader
{
    vec3 L = lightPos - worldPos.xyz;
    float dist = L.length();
    L = normalize( L );
    vec3 lightColor = vec3( 10, 10, 8 );
    vec3 materialColor = texture( pixels, uv ).xyz;
    float attenuation = 1.0f / (dist * dist);
    outputColor = vec4( materialColor * max( 0.0f, dot( L, normal.xyz ) ) *
        attenuation * lightColor, 1 );
}
```

In game.cs, Init():

```
// set the light
int lightID = GL.GetUniformLocation(
    shader.programID,
    "lightPos"
);
GL.UseProgram( shader.programID );
GL.Uniform3(
    lightID,
    0.0f, 10.0f, 0.0f
);
```



```

k (depth < MAXDEPTH) {
    // Sample point on sphere
    float r = inside / (1 + sqrt(0.5 * (1 - cosPhi)));
    int nt = nt / nc, ndd = nnt / nnc;
    double u = rand(), v = rand();
    double cos2t = 1.0f - nnt * u;
    double t = sqrt(cos2t);
    double D, N };
}

// Evaluate diffuse reflection
float E * diffuse;
// ...
= true;

// ...
refl + refr)) && (depth < MAXDEPTH)) {
    // ...
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
// estimation - doing it properly, classical Monte Carlo
if;
radiance = SampleLight( &rand, I, &l, &align, &pdf );
e.x + radiance.y + radiance.z ) > 0) && (cosThetaOut > 0) {
    w = true;
    brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    st3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance);

random walk - done properly, closely following Monte Carlo
(survive)

};
st3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
tion = true;

```



Today's Agenda:

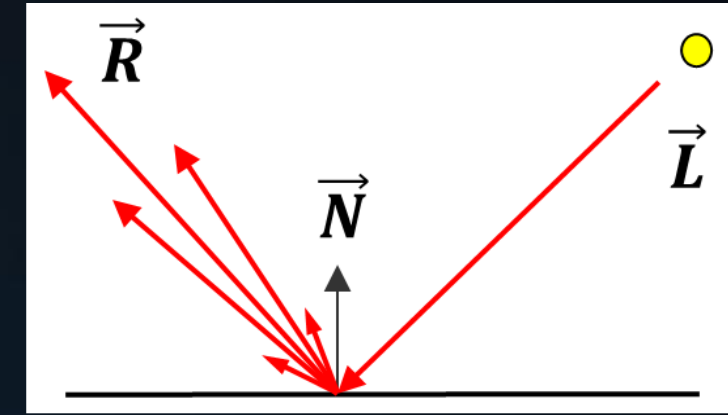
- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur



Phong

Glossy Materials

A glossy material reflects, but the reflection is somewhat fuzzy:



Using a ray tracer we achieve this effect by sending multiple rays in directions close to the reflection vector $\hat{R}$.



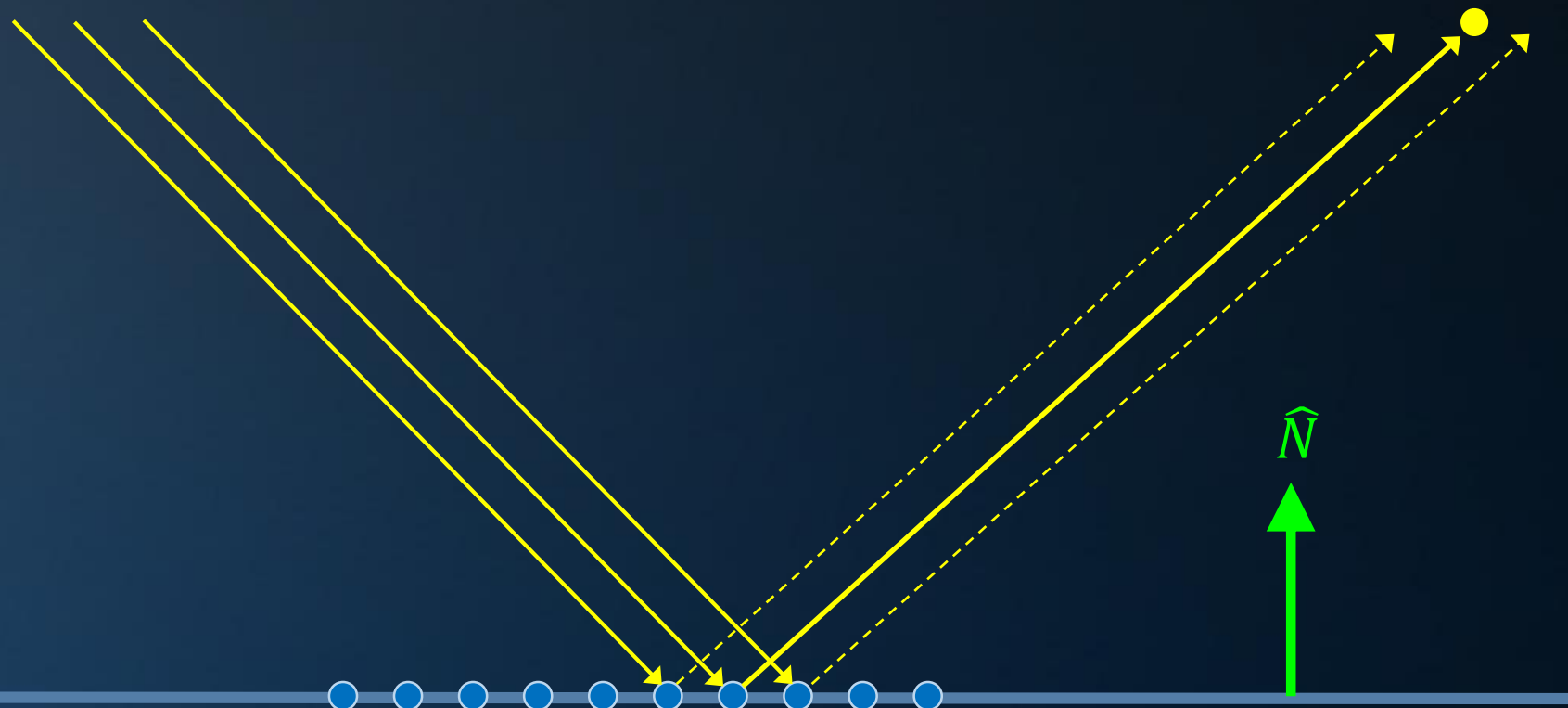
Phong

Glossy Materials

```

ices
    & (depth < MAXDEPTH)
    {
        // Inside / Outside
        nt = nt / nc; nct = nc / nt;
        // Russian roulette
        r = 1.0f - nnt * nnt;
        // D, N
        D = D * N;
        // a = nt - nc, b = nt * nc
        // Tr = 1 - (R0 + (1 - R0) * r)
        // R = (D * nnt - N * (1 - R0))
        // E * diffuse;
        // = true;
        // refl + refr) && (depth < MAXDEPTH)
        // D, N
        // refl * E * diffuse;
        // = true;
        // MAXDEPTH
        // survive = SurvivalProbability( diffuse );
        // estimation - doing it properly, closely following
        // if;
        // radiance = SampleLight( &rand, I, &lt, &light );
        // e.x + radiance.y + radiance.z > 0) && (depth < MAXDEPTH)
        // v = true;
        // at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        // at3 factor = diffuse * INVPI;
        // at weight = Mis2( directPdf, brdfPdf );
        // at cosThetaOut = dot( N, L );
        // E * ((weight * cosThetaOut) / directPdf) * (radiance
        // random walk - done properly, closely following
        // (survive)
        // at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &rand, &pdf );
        // survive;
        // pdf;
        // n = E * brdf * (dot( N, R ) / pdf);
        // ion = true;
    }

```



Glossy Materials

- $\hat{L}$ be a vector from the fragment position b to the light;
- $\hat{R}$ be the vector $\hat{V}_b$ reflected in the plane with normal $\hat{N}$

if $\hat{R} = \hat{L}$ then $\hat{R} \cdot \hat{L} = 1$

if $\hat{R} \approx \hat{L}$ then $\hat{R} \cdot \hat{L} \rightarrow 1$

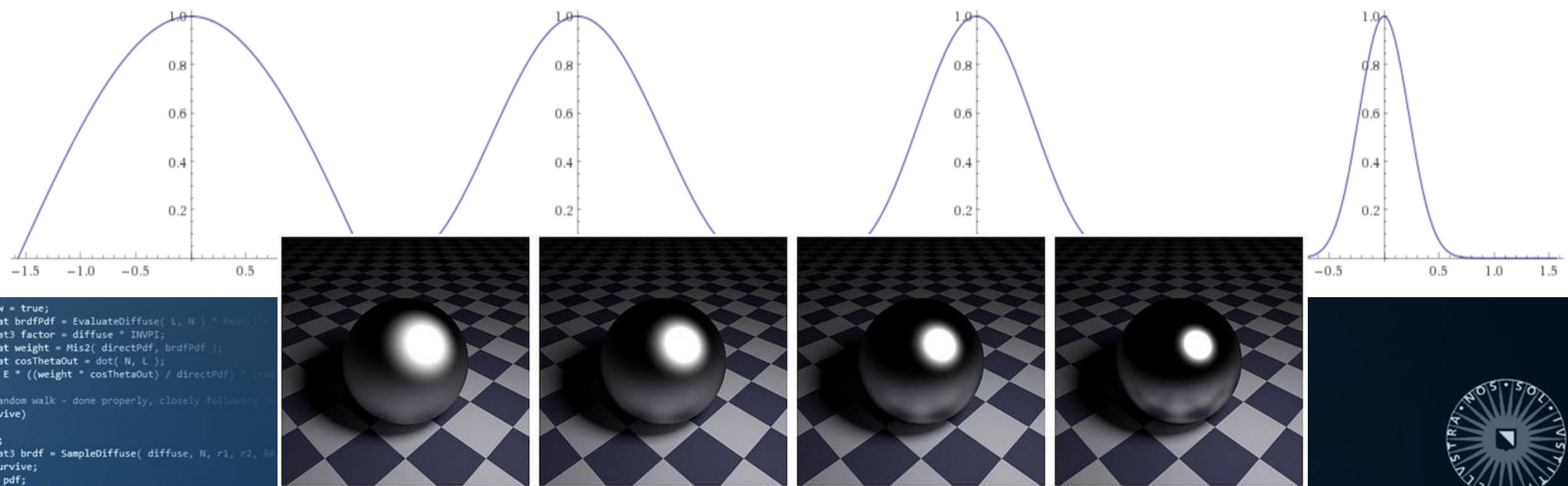


Phong

Glossy Materials

“Locations near b receive almost as much light as b .”

But how much? $(\hat{L} \cdot \hat{V})^\alpha$



The Full Phong

The Phong material model is a combination of:

- 'Specular' illumination: $(\vec{L} \cdot \vec{R})^\alpha$, times the 'specular color' of the light, times the 'specular color' of the material;
- Diffuse illumination: $(\vec{N} \cdot \vec{L})$, times the 'diffuse color' of the light, times the 'diffuse color' of the material;
- An 'ambient color'.



Phong



Today's Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur



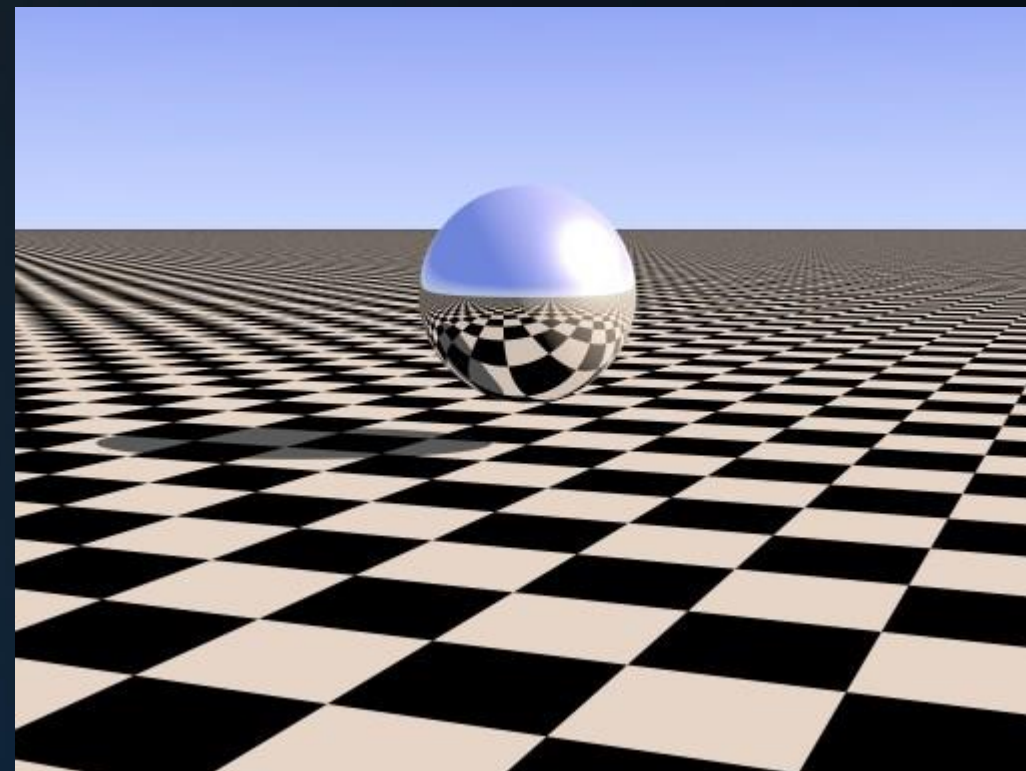
Mirrors

Reflections

Reflections in a ray tracer are easy:

$$\vec{R} = \vec{L} - 2(\vec{L} \cdot \vec{N})\vec{N}$$

But what about rasterizers?





Mirrors



```

ices
& (depth < MAXDEPTH)
{
    nt = inside / nc; nct = nt / nc; nctd = nt / nc;
    cos2t = 1.0f - nct; nct = sqrt(cos2t);
    D, N );
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (RB + (1 - RB) * nct);
Tr) R = (D * nnt - N * (1 - nnt));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following walls
if;
radiance = SampleLight( &rand, I, &t, &light );
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
{
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance);
}

random walk - done properly, closely following walls
survive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Mirrors

Planar Reflections

We can fake reflections in a rasterizer by duplicating the scene:

The mirror is not really there; it's just a hole through which we see a copy of the scene.

```

ics
& (depth < MAXDEPTH)
{
    t = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
}

at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) * t);
Tr) R = (D * nnt - N * (a *
)

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
-refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &t, &light);
e.x + radiance.y + radiance.z) > 0) && (depth <
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
andom walk - done properly, closely following well
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Mirrors

```

    // Ray-sphere intersection
    float t;
    if (depth < MAXDEPTH)
    {
        // Inside the sphere
        t = inside / 1.5;
        nt = nt / nc, ndd = ndd * ndd;
        cos2t = 1.0f - nnt * ndd;
        if (cos2t < 0) cos2t = 0;
        float D, N;
        // ...
    }

    // Ray-sphere intersection
    float a = nt - nc, b = nt * nc;
    float Tr = 1 - (R0 + (1 - R0) * t);
    float R = (D * nnt - N * ndd);

    // ...
    E * diffuse;
    = true;

    // ...
    refl + refr)) && (depth < MAXDEPTH))
    {
        D, N;
        refl * E * diffuse;
        = true;

        MAXDEPTH)

    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, close
    if;
    radiance = SampleLight( &rand, I, &L, &
    e.x + radiance.y + radiance.z) > 0) &&

    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) *
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPd

    random walk - done properly, closely following the
    (survive)

    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



Mirrors

Environment Mapping

Reflections on complex surfaces are faked using an environment map.

This is done exactly as in a ray tracer:

- at the fragment position, we have $\hat{V}$ and $\hat{N}$;
- based on these we calculate the reflected vector $\hat{R}$;
- we use $\hat{R}$ to look up a value in the skydome texture.

Limitations:

- we will not reflect anything but the skydome;
- the reflection is static.



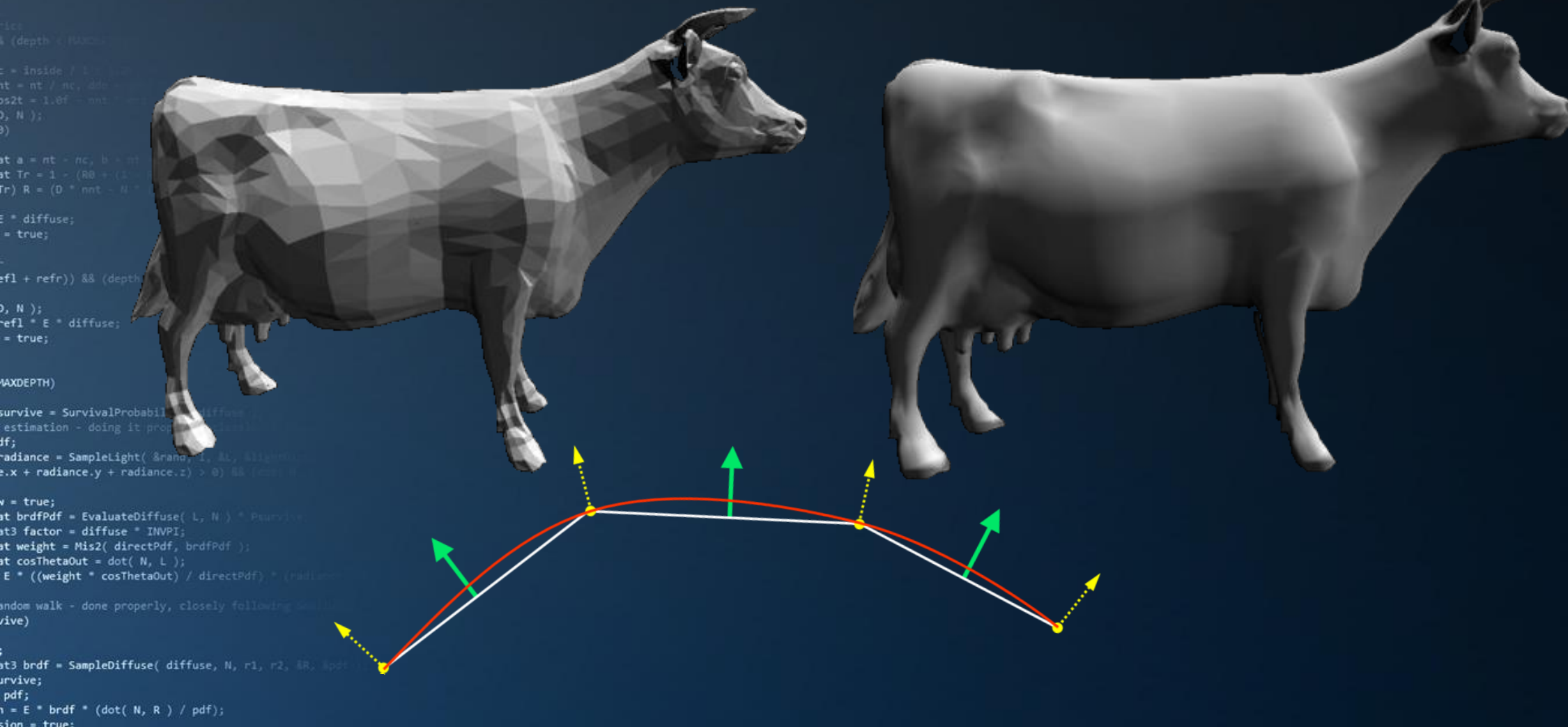
Today's Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur



Bumps

Recap: Normal Interpolation



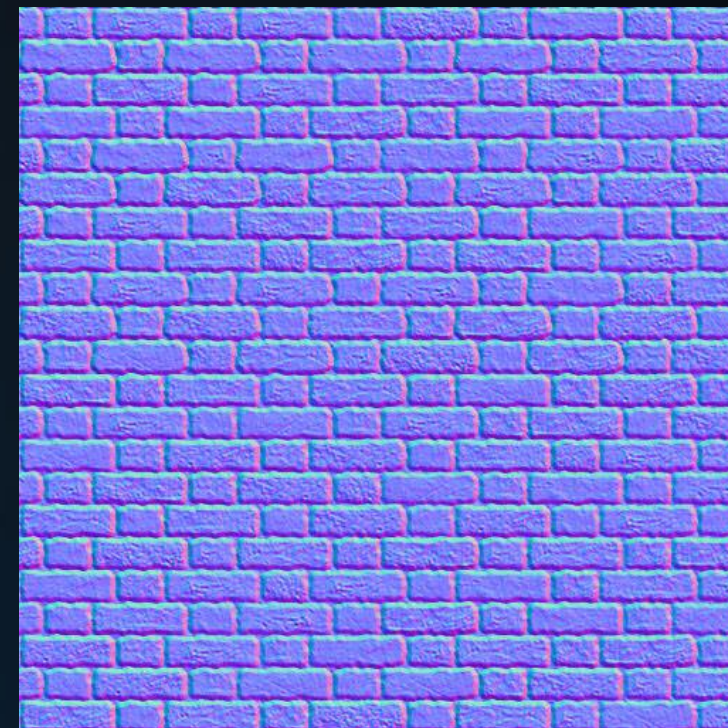
Bumps

Normal Maps

A normal map is similar to a texture:

- we use textures to lookup a color at a particular position on a mesh;
- we use normal to lookup a normal at a particular position.

Normal maps generally store normals in *tangent space*.



Bumps

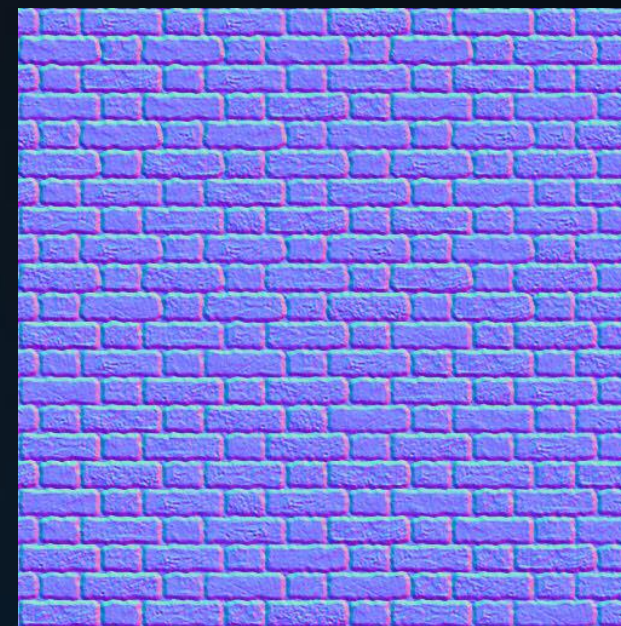
Normal Map Data

A normal map stores 2 or 3 components per texel.

For 3 components: $x, y, z \in [0..255]$.

To get this to the correct range:

- Cast to float
- Divide by 128 $\rightarrow x, y, z \in [0..2]$
- Subtract 1 $\rightarrow x, y, z \in [-1..1]$



Bumps

Normal Map Data

A normal map stores 2 or 3 components per texel.

For 2 components: $x, y \in [0..255]$.

To reconstruct the normal, we first apply the same conversion as we did for three components. Then:

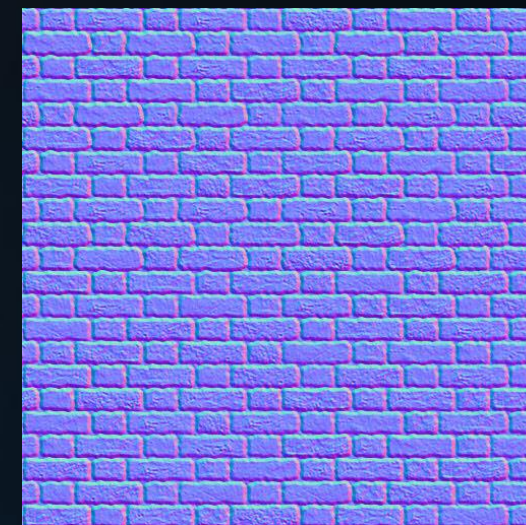
$$\sqrt{x^2 + y^2 + z^2} = 1$$

$$x^2 + y^2 + z^2 = 1$$

$$z^2 = 1 - (x^2 + y^2)$$

$$z = \pm\sqrt{1 - (x^2 + y^2)}$$

$$z = \sqrt{1 - (x^2 + y^2)}$$



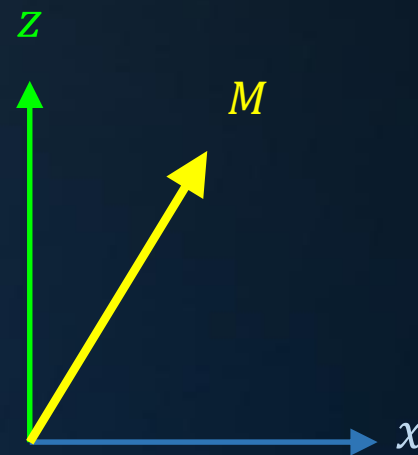
Bumps

Tangent Space

Things are easy when $x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ and $z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$:

$$N = M_x x + M_y y + M_z z$$

$$= M_x \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + M_y \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + M_z \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix}$$



Bumps

Tangent Space

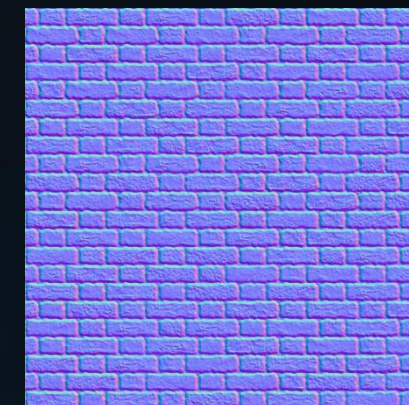
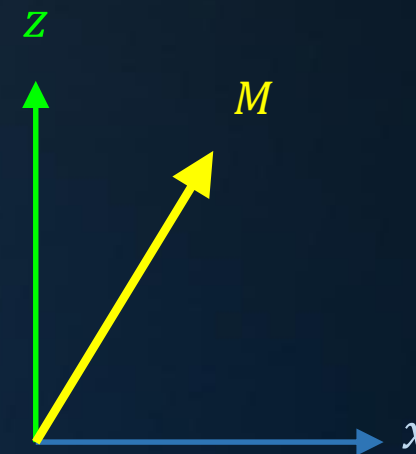
Things are still easy for arbitrary x, y and z :

$$N = M_x x + M_y y + M_z z$$

Obtaining this x, y and z:

- z = (interpolated) surface normal
- x is perpendicular to z
- y is perpendicular to z and x

See Crytek for details*.



<http://docs.cryengine.com/display/SDKDOC4/Tangent+Space+Normal+Mapping>

<https://fenix.tecnico.ulisboa.pt/downloadFile/845043405449073/Tangent%20Space%20Calculation.pdf>



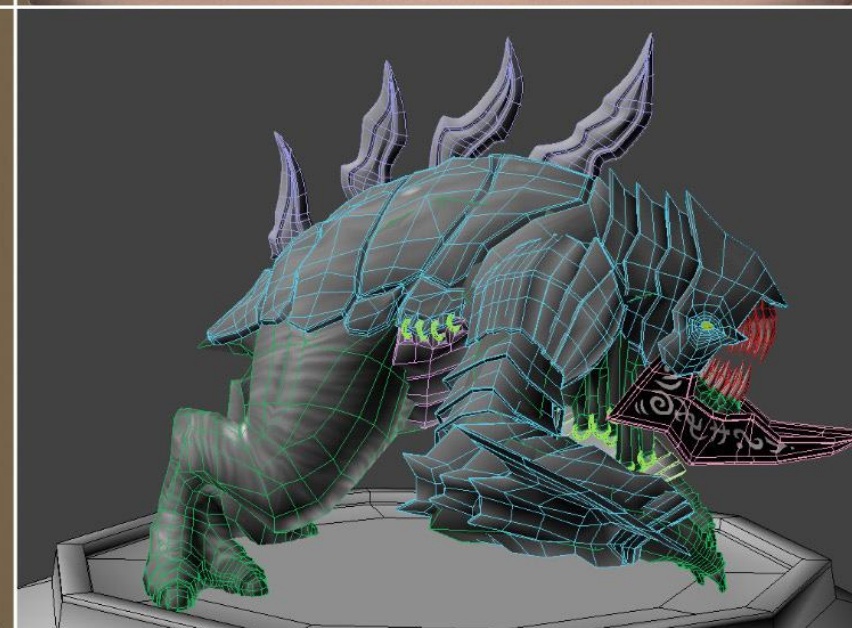
Bumps



```
...ics
& (depth < MAXDEPTH)
{
    // Inside / outside
    int nt = nt / nc;
    float cos2t = 1.0f - nt;
    float r = sqrt(cos2t);
    float D, N;
    ...
    float a = nt - nc, b = nt - nc;
    float Tr = 1 - (R0 + (1 - R0) * r);
    float R = (D * nt - N * (1 - Tr));
    ...
    E * diffuse;
    ...
    = true;
    ...
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N;
        refl * E * diffuse;
        = true;
        ...
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely
    if;
    radiance = SampleLight( &rand, I, &t, &light );
    e.x + radiance.y + radiance.z) > 0) && (rand < 1)
    {
        v = true;
        float brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        float3 factor = diffuse * INVPI;
        float weight = Mix2( directPdf, brdfPdf );
        float cosThetaOut = dot( N, L );
        float E = ((weight * cosThetaOut) / directPdf) * (rand < 1);
        ...
    }
    random walk - done properly, closely following
    (survive)
    {
        float3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        n = true;
    }
}
```



Bumps



Today's Agenda:

- Recap: Diffuse Materials
- The Phong Shading Model
- Environment Mapping
- Normal Mapping
- Rendering Short Fur



Fur

Fur, Grass etc.



```
bool survive = true;
float3 brdfPdf = EvaluateDiffuse( L, N );
float3 factor = diffuse * INVPI;
float weight = Mix2( directPdf, brdfPdf );
float cosThetaOut = dot( N, L );
float E = ((weight * cosThetaOut) / directPdf) * (radiance * 0.5);
// random walk - done properly, closely following path
// survive
};
float3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
float3 survive;
float pdf;
float3 n = E * brdf * (dot( N, R ) / pdf);
float3 survive = true;
```



Fur

Fur, Grass etc.



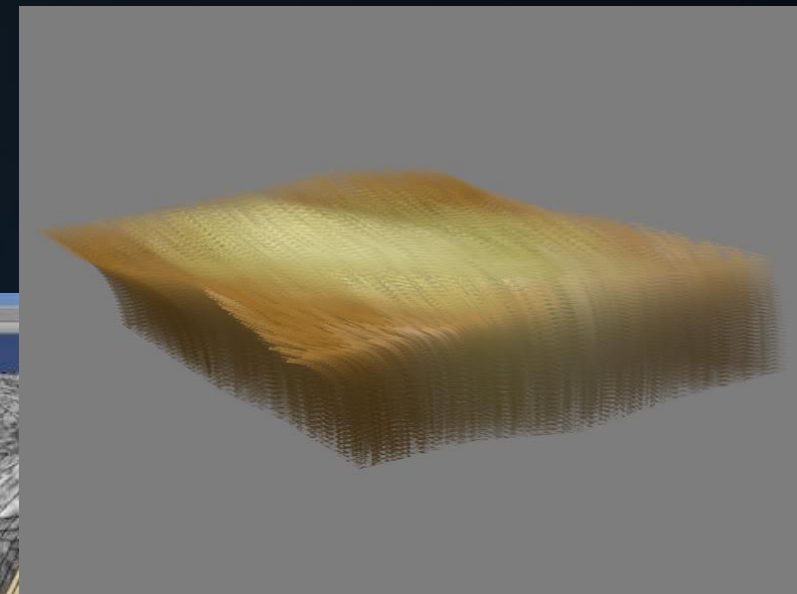
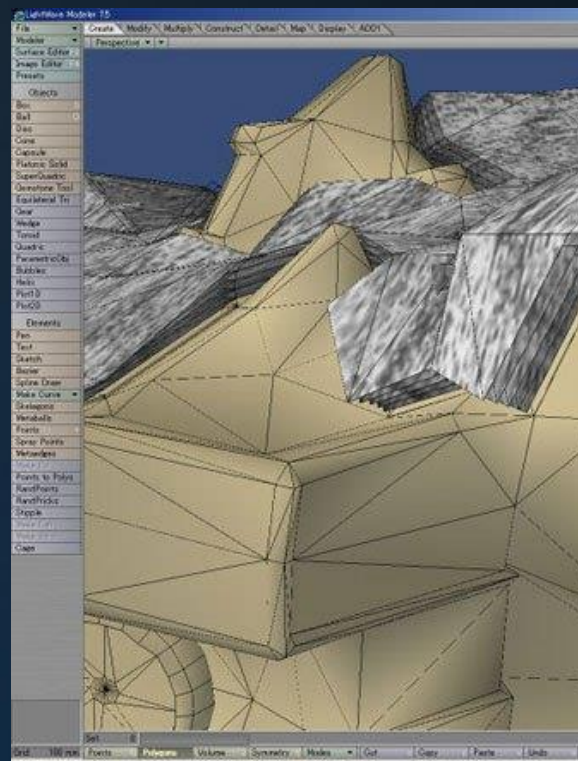
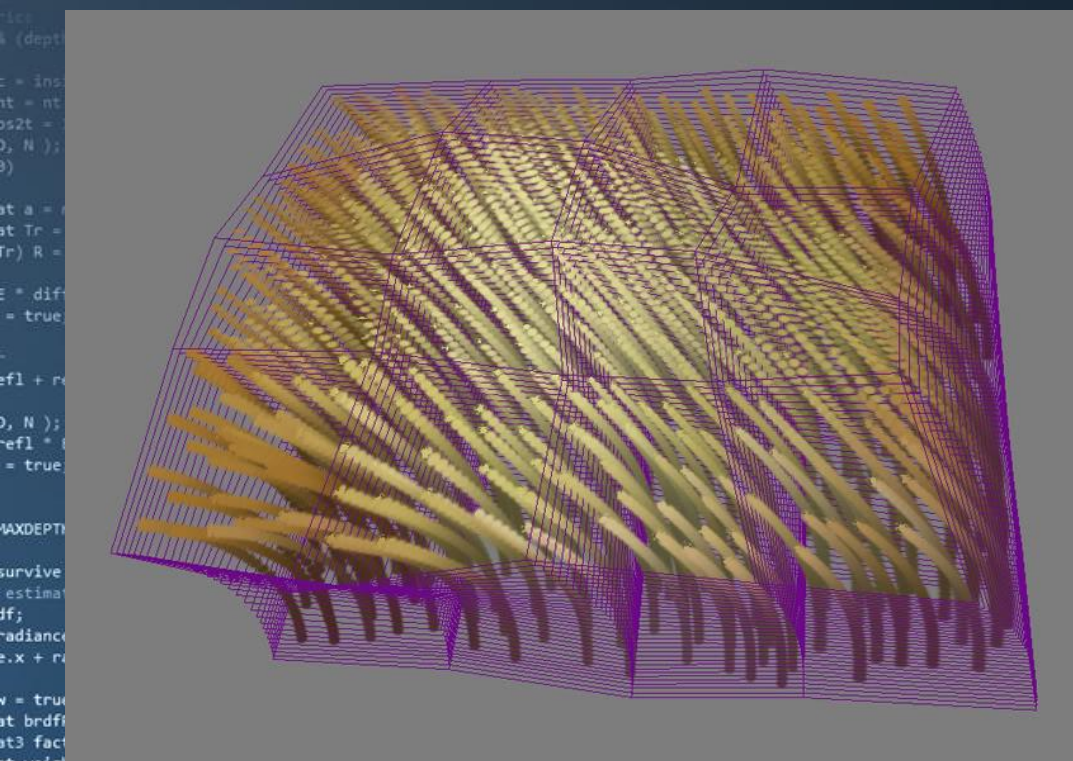
```
if (v == true;
    at brdfPdf = EvaluateDiffuse( L, N, ) * PI * r1 * r2;
    at3 factor = diffuse * INVPI;
    at weight = Mix2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radius *
random walk - done properly, closely following walls, etc.
survive)
```

```
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

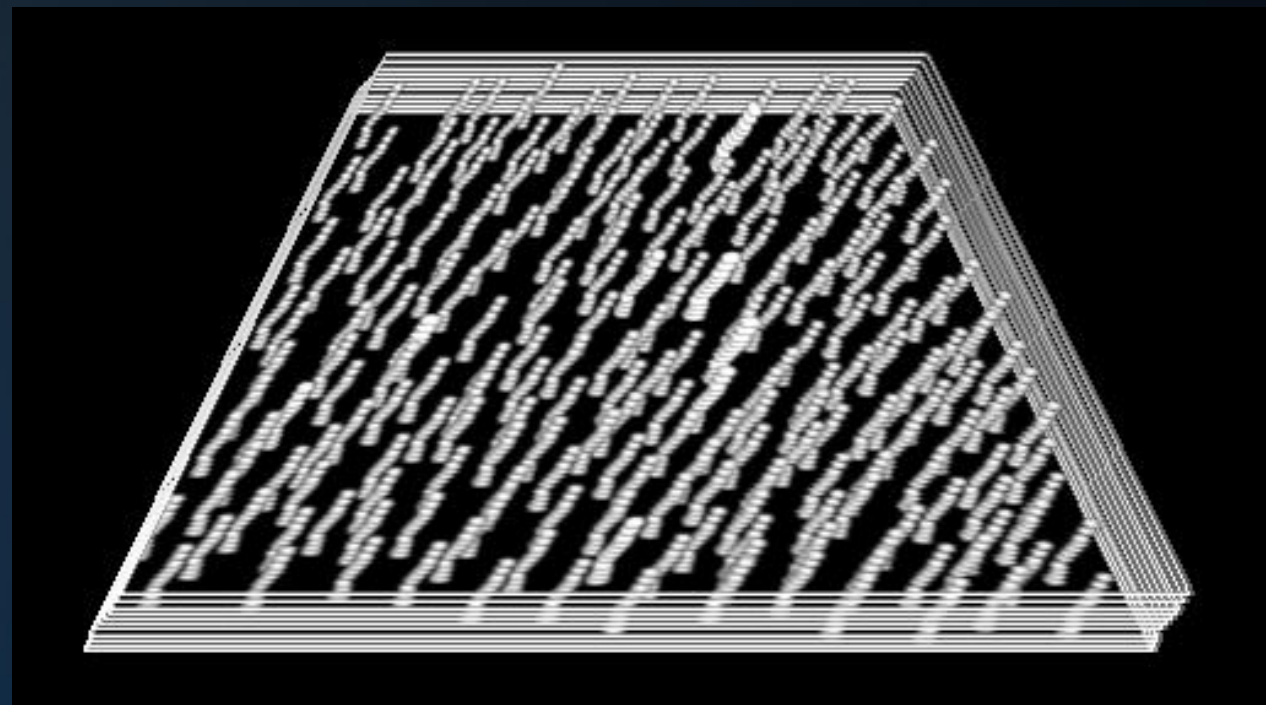


Fur

Fur, Grass etc.



Fur, Grass etc.



Fur

Fur, Grass etc.

```
#version 330
```

```
...
```

```
// shell offset
```

```
uniform float shellOffset;
```

```
// vertex shader
```

```
void main()
```

```
{
```

```
    gl_Position = transform * vec4( vPosition + shellOffset * vNormal, 1.0 );
```

```
    worldPos = toWorld * vec4( vPosition + shellOffset * vNormal, 1.0f );
```

```
    normal = toWorld * vec4( vNormal, 0.0f );
```

```
    uv = vUV;
```

```
}
```



Fur

Fur, Grass etc.

In game.cs:

```
for( int i = 0; i < 30; i++ )
{
    GL.UseProgram( shader.programID );
    GL.Uniform1( offsetID, (float)i * 0.02f );
    mesh.Render( shader, prevMat[19 - i / 4], prevWld[19 - i / 4], fur );
}
```



```

    c = inside / l * i;
    nt = nt / nc; ddn = dot( n, -c);
    cos2t = 1.0f - nt * nt; r = sqrt(cos2t);
    D, N));
    )
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * idot( n, n ));

E * diffuse;
= true;

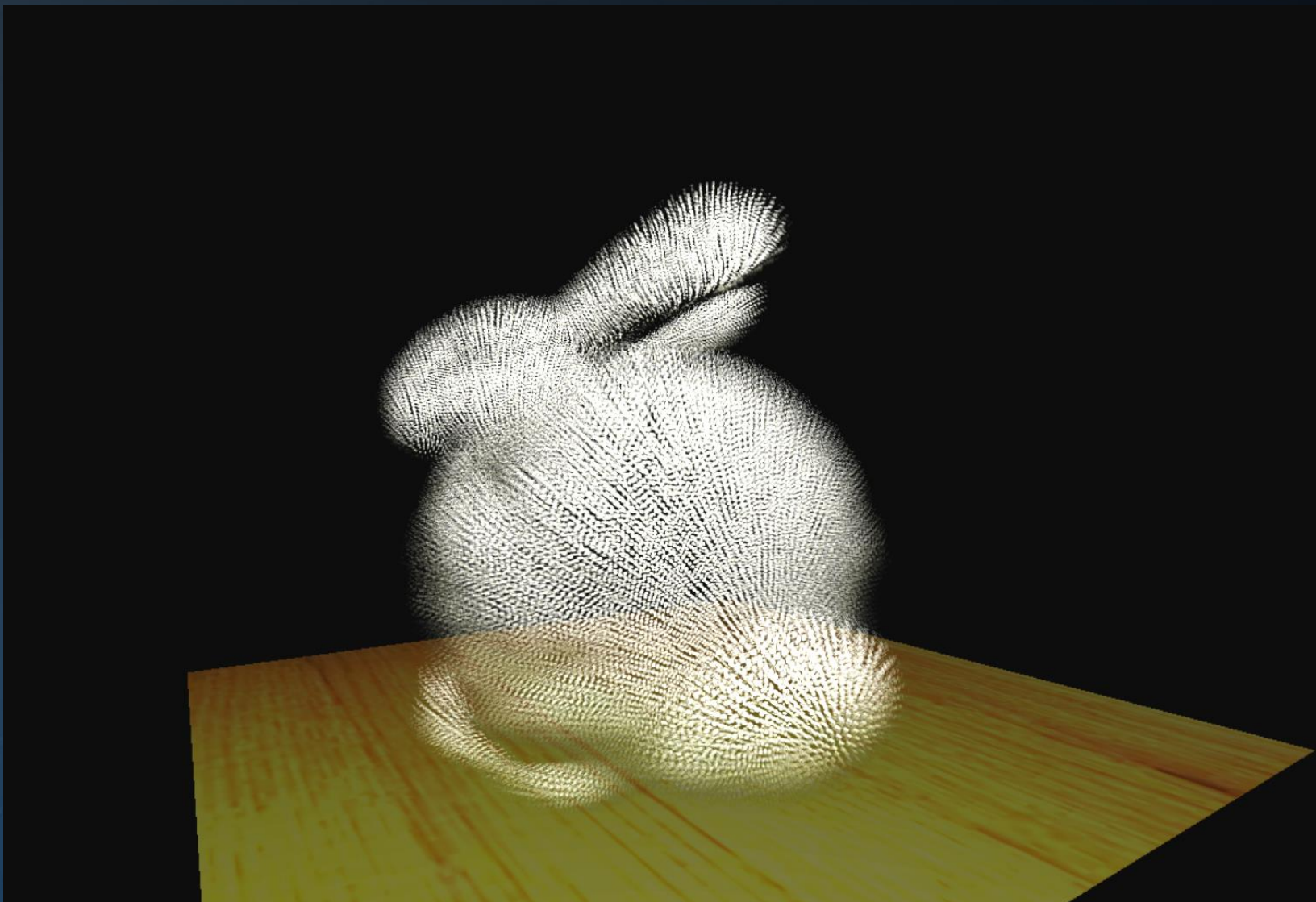
-
efl + refr)) && (depth < MAXDEPTH)) {
D, N );
-refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, clearly
if;
radianc = SampleLight( &brand, I, &l, &light
.e.x + radianc.y + radianc.z) > 0) && (mode
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Pearre
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
random walk - done properly, closely followi
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
ision = true;

```



INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2017

END OF lecture 10: “Shaders”

Next lecture: “Visibility”

