Lecture 14: Illumination
SageMath Code
CS 410
October 10, 2017
Start Full & Work Backwards
Scene Specification

cam = { 'eye' : vector(RR, 3, (25, 25, 60)),
         'look' : vector(RR, 3, (25, 25, 10)),
         'up' : vector(RR, 3, (0, 1, 0)),
         'bnds' : { 'l' : -1, 'r' : 1, 'b' : -1, 't' : 1},
         'nefa' : { 'near' : -10, 'far' : -60},
         'resl' : { 'width' : 64, 'height' : 64} }
globe = { 'c' : vector(RR, 3, (25, 25, 10)), 'r' : 4.0 }  
matl = { 'ka' : vector(RR, 3, (0.2, 0.2, 0.2)),
         'kd' : vector(RR, 3, (0.7, 0.7, 0.7)),
         'ks' : vector(RR, 3, (0.5, 0.5, 0.5)) }
ambient = vector(RR, 3, (0.2, 0.2, 0.2))
lights = [{ 'p' : vector(RR, 3, ( 5, 32, 30)),
            'e' : vector(RR, 3, (0.5, 1.0, 0.5))},
           { 'p' : vector(RR, 3, (45, 32, 30)),
            'e' : vector(RR, 3, (1.0, 0.5, 0.5))}]
Code Implements fast algorithm

Note intermediate variable we have discussed previously ‘disc’.

Note that this function returns a list and the first element indicates whether an intersection exists.
Ray Sphere RGB

• This code does all three
  – Ambient, diffuse and specular

```python
def raySphereRGB(ray, sph):
    hitp = raySphereTest(ray, sph)
    if hitp[0] :
        ptos = hitp[2]
        snrm = ptos - sph['c'] ; snrm = snrm / snrm.norm()
        color = ambient.pairwise_product(mat1['ka'])
        for lt in lights :
            ptL = lt['p']
            emL = lt['e']
            toL = ptL - ptos; toL = toL / toL.norm()
            if (snrm.dot_product(toL) > 0.0) :
                color += mat1['kd'].pairwise_product(emL) * snrm.dot_product(toL)
                toC = ray['L'] - ptos; toC = toC / toC.norm()
                spR = (2 * snrm.dot_product(toL) * snrm) - toL
                color += mat1['ks'].pairwise_product(emL) * toC.dot_product(spR)**16
                # print(ray['L'], toC.dot_product(spR)**4)
            return [True, color]
    else :
        return [False]
```
First, pass ray and sphere to intersection code and if the response returns true, then compute and return illumination, otherwise return false.

```python
def raySphereRGB(ray, sph):
    hitp = raySphereTest(ray, sph)
    if hitp[0]:
        ptos = hitp[2]
        snrm = ptos - sph['c'] ; snrm = snrm / snrm.norm()
        color = ambient.pairwise_product(mat1['ka'])
        for lt in lights:
            ptL = lt['p']
            emL = lt['e']
            toL = ptL - ptos; toL = toL / toL.norm()
            if (snrm.dot_product(toL) > 0.0):
                color += mat1['kd'].pairwise_product(emL) * snrm.dot_product(toL)
                toC = ray['L'] - ptos; toC = toC / toC.norm()
                spR = (2 * snrm.dot_product(toL) * snrm) - toL
                color += mat1['ks'].pairwise_product(emL) * toC.dot_product(spR)**16
            #print(ray['L'], toC.dot_product(spR)**4)
        return [True, color]
    else:
        return [False]
```
Retrieve the 3D position of the point of intersection on the sphere and then use that position to compute the surface normal. Recall that it is particularly easy to compute sphere surface normals.

```python
def raySphereRGB(ray, sph):
    hip = raySphereTest(ray, sph)
    if hip[0]:
        ptos = hip[2]
        snrm = ptos - sph['c'] ; snrm = snrm / snrm.norm()
        color = ambient.pairwise_product(mat1['ka'])
        for lt in lights:
            ptL = lt['p']
            emL = lt['e']
            toL = ptL - ptos ; toL = toL / toL.norm()
            if (snrm.dot_product(toL) > 0.0):
                color += mat1['kd'].pairwise_product(emL) * snrm.dot_product(toL)
                toC = ray['L'] - ptos ; toC = toC / toC.norm()
                spR = (2 * snrm.dot_product(toL) * snrm) - toL
                color += mat1['ks'].pairwise_product(emL) * toC.dot_product(spR)**16
            # print(ray['L'], toC.dot_product(spR)**4)
        return [True, color]
    else:
        return [False]
```
Create the color starting with the ambient illumination computation. That is the pairwise product of the material property 'ka' and strength of the ambient light source. Recall 'ambient' is a vector and hence the ability to do a pairwise product in one method call.
The next step iterates over all light sources.

‘toL’ is a unit length vector pointing toward the light source.

If and only if dot product of ‘toL’ and ‘snrm’ is positive should illumination be computed (think through why this is true).

The illumination is a combination of diffuse and specular (next slides).

```python
def raytrace(sph, mat1, lights):
    toL = sph['display'] - sph['center']
    snrm = ptos - sph['center']
    snrm = snrm / snrm.norm()
    color = ambient.pairwise_product(mat1['ka'])
    for lt in lights:
        pL = lt['p']
        eL = lt['e']
        toL = pL - ptos; toL = toL / toL.norm()
        if (snrm.dot_product(toL) > 0.0):
            color += mat1['kd'].pairwise_product(eL) * snrm.dot_product(toL)
            toC = ray['L'] - ptos; toC = toC / toC.norm()
            spR = (2 * snrm.dot_product(toL) * snrm) - toL
            color += mat1['ks'].pairwise_product(eL) * toC.dot_product(spR)**16

    return [True, color]
else:
    return [False]
```
Recall: Diffuse Reflection

- Illumination/area drops by $\theta$

\[
\begin{align*}
  i_r &= b_r \cos \theta \\
  i_g &= b_g \cos \theta \\
  i_b &= b_b \cos \theta
\end{align*}
\]

- Where light $L$ emits light with brightness $B$:

\[
B = \begin{bmatrix}
  b_r \\
  b_g \\
  b_b
\end{bmatrix}
\]

- And as SageMath code

```python
color += matl['kd'].pairwise_product(emL) * snrm.dot_product(toL)
```
Last but not least, specular illumination.

Superficial observation, three lines of code compared with one.

Substantive observation, there are three steps

1. Determine vector indicating direction to Camera, called ‘toC’
2. Determine vector that is reflection ray, called ‘spR’
3. Modulate specular illumination by Phong Shading formula

```python
def render_object(*args, **kwargs):
    i = 0
    for lt in lights:
        ptL = lt['p']
        emL = lt['e']
        toL = ptL - pts; toL = toL / toL.norm()
        if (snrm.dot_product(toL) > 0.0):
            color += mat1['kd'].pairwise_product(emL) * snrm.dot_product(toL)
            toC = ray['L'] - pts; toC = toC / toC.norm()
            spR = (2 * snrm.dot_product(toL) * snrm) - toL
            color += mat1['ks'].pairwise_product(emL) * toC.dot_product(spR)**16
            # print(ray['L'], toC.dot_product(spR)**4)
    return [True, color]
else:
    return [False]
```
Review: Reflection and Viewer

- $V$ is the unit vector from the surface point to the camera:
  
  $$V = (C - S)/|C - S|$$

- Calculating $R$ is more involved
Review: Reflection Vector

• Let NL be the projection of L onto N
  – Assume N is a unit vector
    \[ NL = (L \cdot N)N \]
• T is defined as
  \[ T = NL - L = (L \cdot N)N - L \]
• R is \( L + 2T \)
  \[
  R = L + 2\left( (L \cdot N)N - L \right) \\
  = 2\left( L \cdot N \right)N - L
  
\]
Review: Phong Shading

\[ I = k_s B (\cos \Phi)^\alpha \]

- \( \Phi \) is the angle between the viewing ray and the angle of reflection.
- \( \alpha \) is the so-called Phong constant, expressing how “shiny” an object is
  - Mirrors: \( \alpha = 200 \)
  - Dull objects: \( \alpha = 5 \ldots 50 \)
Review: Changing $\alpha$

- Consider which has the larger alpha?
Phong 16 vs 200
Back to Ray Tracing

• Put it all together and generate an image

```python
img = Image('RGB', (width, height), 'black')
pix = img.pixels()
for i in range(width):
    for j in range(height):
        res = raySphereRGB(pixelRay(i,(height - j -1)), globe)
        if res[0]:
            pix[i,j] = tuple(map(lambda x: ZZ(max(0,min(255,round(255.0 * x)))), res[1]))
img
```

• Note the code is simple at this level of abstraction – this is as it should be.