Computational Methods for Radiance

Render “the full variety offered by the direct observation of objects.” (Computationally).
Methods for Plenoptic 1.0
Computing with Radiance

- Goal: Render “the full variety offered by the direct observation of objects.”

- Computational tasks:
  - Capture radiance (sample, digitize)
  - Interpreting a digital plenoptic image as radiance
  - Rendering radiance to image
  - Algorithms for transforming radiance
  - Real-time interactive implementation using GPUs
Radiance Representation (Plenoptic 1.0)

- Sensor image represents sampled radiance
  - Position is sampled by microlenses “as pixels”
Radiance Representation (Plenoptic 1.0)

- Sensor image represents sampled radiance
  - Position is sampled by microlenses
  - Direction is sampled by sensor pixels

\[ I(q) = \frac{d}{f} r(0, \frac{q}{f}) \]
Radiance Representation (Plenoptic 1.0)

Captured by Sensor

Interpreted as Radiance

Conventional plenoptic camera

Main lens image plane

Main (Objective) Lens

Microlens (Lenslet) Array

f

angle

space
Radiance Representation (Plenoptic 1.0)
Radiance Representation (Plenoptic 1.0)

- Plenoptic image is a “flat” 2D array of 2D arrays
- 4D array
- “Position major”
Radiance Representation (Plenoptic 1.0)
Radiance Representation (Plenoptic 1.0)

- Plenoptic image is a “flat” 2D array of 2D arrays
- 4D array
- “Direction major”
Creating Radiance Array (Position Major)

- Given 2D “flat” captured by radiance camera
- Create 4D array
- Sample the same directional pixel from every position
- Convention $r[i, j, m, n]$
  - Follow $r(q,p)$
  - $i,j$ are position
  - $m,n$ are direction
Creating Radiance Array (Position Major)

- Given 2D position major “flat” from radiance camera
- Create 4D array
- If 2D position major “flat” is regular

```python
(jnds,inds) = mgrid[0:height:nump,0:width:nump]
for j in range(0,nump):
    for i in range(0,nump):
        radiance[:,:,j,i] = image[jnds+j,inds+i]
```

- Python, matlab very similar
- Samples the same directional pixel from every position
Camera Arrays

- The most popular lightfield camera is simply an array of conventional cameras, like the Stanford array.

- Alternatively, an array of lenses/prisms with a common sensor, like the Adobe array.
Creating Radiance Array (Direction Major)

- Given 2D “flat” captured by radiance camera
- Create 4D array
- Sample the same positional pixel from every direction
- Convention $r[i, j, m, n]$
  - Follow $r(q,p)$
  - $i,j$ are position
  - $m,n$ are direction
Creating Radiance Array (Direction Major)

- Given 2D direction major “flat” from radiance camera
- Create 4D array
- If 2D direction major “flat” is regular

\[
(jnds, inds) = \text{mgrid}[0:\text{height}:\text{numq},0:\text{width}:\text{numq}]
\]

```python
for j in range(0, numq):
    for i in range(0, numq):
        radiance[j,l,:,:] = image[jnds+j,inds+i]
```

- Python, matlab very similar
- Samples the same positional pixel from every direction
Aside: Dimensionality

- How large of a sensor do we need to capture radiance?
- Memory, computation requirements?
- What is a reasonable size for a rendered image?
Image Rendering

- A traditional image (a picture) is formed by integrating rays from every direction at each pixel.

\[ I(q) = \int r(q, p) \, dp \]
Integration is averaging over directions at each position.

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[i, j, m, n] \]
Image Rendering

- Integration is averaging over directions at each position

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[i, j, m, n] \]

- Corresponding python code:

```python
for j in range(0, nump):
    for i in range(0, nump):
        rendered[:, :] += radiance[:, :, j, i]

rendered /= (nump * nump)
```
Rendering the Wide Variety

- Averaging recovers traditional picture
- Wide variety can also be rendered
  - Different aperture
  - Different viewpoint
  - Different focus
  - Different depth of field
  - Stereo
  - High dynamic range
  - Super resolution
  - ...
Changing Aperture Computationally

- What does the sensor capture with smaller aperture?
- What is this in terms of phase space?
Different Aperture

- A smaller aperture is a smaller set of directions
Different Apertures

- A smaller aperture is a smaller set of directions

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[m, n, i, j] \]

- Corresponding python code:

```python
for j in range(alpha, nump-alpha):
    for i in range(alpha, nump-alpha):
        rendered[:, :] += radiance[::, ::, j, i]

rendered /= (nump * nump)
```
What does the sensor capture with different viewpoints?
What is this in terms of phase space?
Pinhole Rendering (single viewpoint)

- Only render from one pixel from each microimage

\[ \text{rendered}[::,::] = \text{radiance}[::,::,j,i] \]
Different Viewpoints

- Different viewpoint is different direction
- Render different directions (or sets of directions)
Different Viewpoints

- Different viewpoint is different direction
- Render different directions (or sets of directions)
Different Viewpoints

- Different viewpoint is different direction
- Render different directions (or sets of directions)
Example [Ren Ng]
Example [Ren Ng]
Example [Ren Ng]
Refocusing

- When we refocus a camera, we change the distance from the lens to the sensor.
- Same object is no longer in focus.
Computational Refocusing

- What does the sensor capture with different focal planes?
- What is this in terms of phase space?
Computational Refocusing

- We capture radiance $r_1$. How can we compute $r_2$?
- Apply translation transform of the radiance.
Algorithm: Computational Refocusing

- Apply shearing transformation: \( r'(q, p) = r(q - tp, p) \)

Then render the new image:

\[
I'(q) = \int r'(q, p) dp
\]
Algorithm: Refocusing

(yind, xind, wind, vind) = mgrid[0:m,0:n,0:r,0:s]

shear_y = y + t*wind / r
shear_x = x + t*vind / s

rad_p = interpolate(rad, [shear_y, shear_x, wind, vind])

- Evaluate the original radiance at a new set of coordinates (interpolation)
- Render as before
Computational Refocusing (Ren Ng)
Computational Refocusing (Ren Ng)
Fourier Slice Refocusing

Ng 2005
Efficient Computational Refocusing

- Refocusing in the spatial domain requires $O(N^4)$ operations for each refocused image.
- An alternative approach (invented by Ren Ng) requires $O(N^4 \log(N))$ for initial setup but then for each rendered image we need only $O(N^2 \log(N))$.

- Insight: Refocus in the frequency domain.
- The frequency domain representation of the rendering integral is the DC directional component (slice).
The Fourier transform of a rendered image:

\[ \hat{I}(\omega_q) = \int I(q)e^{i\omega_q \cdot q} dq \]

Recall that

\[ I(q) = \int r(q, p) dp \]

Thus we have

\[ \hat{I}(\omega_q) = \int \int r(q, p)e^{i\omega_q \cdot q} dpdq \]

\[ = R(\omega_q, 0) \]

In other words, the transform of the rendered image is the DC directional component of \( R(\omega) \).
Translation in the Frequency Domain

- Recall \( R'(\omega) = R(A^T \omega) \)
- In the case of translation

\[
R'(\omega_q, \omega_p) = R(\omega_q, \omega_p - t\omega_q)
\]

- But we are interested in the case \( \omega_p = 0 \)

- I.e., \( I'(\omega_q) = R'(\omega_q, 0) = R(\omega_q, -t\omega_q) \)

- The refocused image is just a **slice** (with slope t)
Algorithm: Fourier Slice Refocusing

- Take FFT of radiance: \( R[i, j, m, n] = FFT(r[i, j, m, n]) \)
- Interpolate to get a slice: \( R[i, j, m, n] \rightarrow \hat{I}[i, j] \)
- Take inverse FFT: \( I'[i, j] = IFFT(\hat{I}[i, j]) \)
Algorithm: Fourier Slice Refocusing

radiancefft = fftn(radiance)

(yind, xind) = mgrid[0:m, 0:n]

vind = t*yind / m
uind = t*xind / n

slice = interpolate(radiancefft, [yind, xind, vind, uind])

rendered = ifft2(slice)
Fourier Slice Refocusing (Ren Ng)
Fourier Slice Refocusing (Ren Ng)
Fourier Slice Refocusing (Ren Ng)
Fourier Slice Refocusing (Ren Ng)
Methods for Plenoptic 2.0
Radiance Representation (Plenoptic 2.0)

- Sensor image samples the radiance
  - Each microlens image samples in position and direction
Radiance Representation (Plenoptic 2.0)

Captured by Sensor

Interpreted as Radiance
Radiance Representation (Plenoptic 2.0)
Radiance Representation (Plenoptic 2.0)

- Plenoptic 2.0 image is a “flat” 2D array of 2D arrays
- Radiance is 4D array
- “Direction major” (approximately, in the sense of tilting)
Creating Radiance Array (Direction Major)

- Given 2D “flat” radiance captured by plenoptic camera
- Create 4D array
- Sample the same positional pixel from every direction
- Convention \( r[i, j, m, n] \)
  - Follow \( r(q, p) \)
  - \( i, j \) are position
  - \( m, n \) are direction
Creating Radiance Array (Direction Major)

- Given 2D direction major “flat” from radiance camera
- Create 4D array
- If 2D direction major “flat” is regular

```
(jnds,inds) = mgrid[0:height:numq,0:width:numq]
for j in range(0,numq):
    for i in range(0,numq):
        radiance[j,l,:,:] = image[jnds+j,inds+i]
```

- Python, matlab very similar
- Samples the same positional pixel from every direction
Rendering Image from 2.0 Data

\[ I[i,j] = \frac{1}{N^2} \sum_{m,n} r[i,j,m,n] \]
Rendering One View from 2.0 Data

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[i, j, m, n] \]
Plenoptic 2.0 Rendering

- Captured Radiance
- Microlens Image
- Patch
- Rendered Image

Full Resolution Rendering
Plenoptic 2.0 Rendering Example
Plenoptic 2.0 Rendering Example
Rendering One View from 2.0 Data

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[i, j, m, n] \]
Integrating Multiple Views

\[ I[i, j] = \frac{1}{N^2} \sum_{m,n} r[i, j, m, n] \]
Rendering the Wide Variety

- Averaging recovers traditional picture
- Wide variety can also be rendered
  - Different aperture
  - Different viewpoint
  - Different focus
  - Different depth of field
  - Stereo
  - High dynamic range
  - Super resolution
  - …
Plenoptic 2.0 Rendering Parallax

Captured Radiance

Microlens Image

Patch

Rendered Image

Full Resolution Rendering
Plenoptic 2.0 Rendering Parallax
Plenoptic 2.0 Rendering Parallax

Captured Radiance

Microlens Image

Patch

Rendered Image

Full Resolution Rendering
Plenoptic 2.0 Rendering Parallax
Plenoptic 2.0 Rendering Parallax
Plenoptic 2.0 Rendering Parallax
Computational Refocusing

- What does the sensor capture with different focal planes?
- What is this in terms of phase space?
Computational Refocusing

- We capture radiance $r_1$. How can we compute $r_2$?
- Apply translation transform of the radiance.
Algorithm: Computational Refocusing

- Apply shearing transformation: \( r'(q, p) = r(q - tp, p) \)

- Then render the new image: 
  \[
  I'(q) = \int r'(q, p) \, dp
  \]
Plenoptic 2.0 Refocusing
Plenoptic 2.0 Refocusing
Plenoptic 2.0 Refocusing
Plenoptic 2.0 Refocusing
Focused Plenoptic Refocusing Principle

- Rendering for two different focal planes
- Comments?
Focused Plenoptic Refocusing Principle

- Rendering for two different focal planes
- Resolution depends on focal plane!
Efficient Implementation with GPU

Real-Time Radiance Rendering and Transforms
Graphics Processing Units

- Radiance processing is computationally expensive
- CPU clock speeds stalled at 3.0GHz
- Nvidia GTX 295:
  - 1.8 Tflop
  - $500
GPU Programming

- Basic alternatives for programming GPU: General purpose (CUDA) or graphics-based (GLSL)
- Open GL Shader Language (GLSL) a natural fit
Rendering with GPU using Open GL

- Read in plenoptic radiance image
- Create 2D texture object for radiance
- Serialize image data to Open GL compatible format
- Define the texture to OpenGL

```python
image = Image.open("lightfield.png")
str_image = image.tostring("raw", "RGBX", 0, -1)

glActiveTexture(GL_TEXTURE0)
lfTexture = glGenTextures(1)
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, lfTexture)
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 3,
             image.size[0], image.size[1], 0,
             GL_RGBA, GL_UNSIGNED_BYTE, str_image)
```
Rendering

- Scene consists of a single (rectangular) polygon to which the radiance texture is mapped
- We provide a fragment shader to render the radiance according to

\[ I[i, j] = \frac{1}{N^2} \sum_{m, n} r[i, j, m, n] \]
GLSL Implementation of Rendering

- Patch
- Microlens Image
- Captured Radiance
- Rendered Image
- Full Resolution Rendering
GLSL Implementation of Rendering

Texture (Captured Radiance)

Polygon

gl_TexCoord[0].st

Fragment Processor

Per Fragment Operations

gl_FragColor

Rendered Image
GLSL Implementation of Rendering

- Given a fragment coordinate
- Find value from texture
GLSL Implementation of Rendering

- Given output pixel coordinate $\text{gl}_\text{TexCoord}[0].st$
- Find relevant microimage $p = \left\lfloor \frac{x}{\mu} \right\rfloor$
- Find offset within $q = \left( x - \left\lfloor \frac{x}{\mu} \right\rfloor \mu \right) \frac{M}{\mu} = \left( \frac{x}{\mu} - p \right) M$
- Center $q' = q + \frac{\mu - M}{2} = \left( \frac{x}{\mu} - p \right) M + \frac{\mu - M}{2}$
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect flat;
uniform float mu;
uniform float M;

void main()
{
    vec2 p = floor(gl_TexCoord[0].st / mu);
    vec2 qp = (gl_TexCoord[0].st / blockSize - p) * M + 0.5*(mu - M);
    vec2 fx = p * mu + qp;

    gl_FragColor = texture2DRect(flat, fx);
}
GLSL Implementation (Live Demo)

- Rendering
- Parallax
- Refocusing
- Stereo