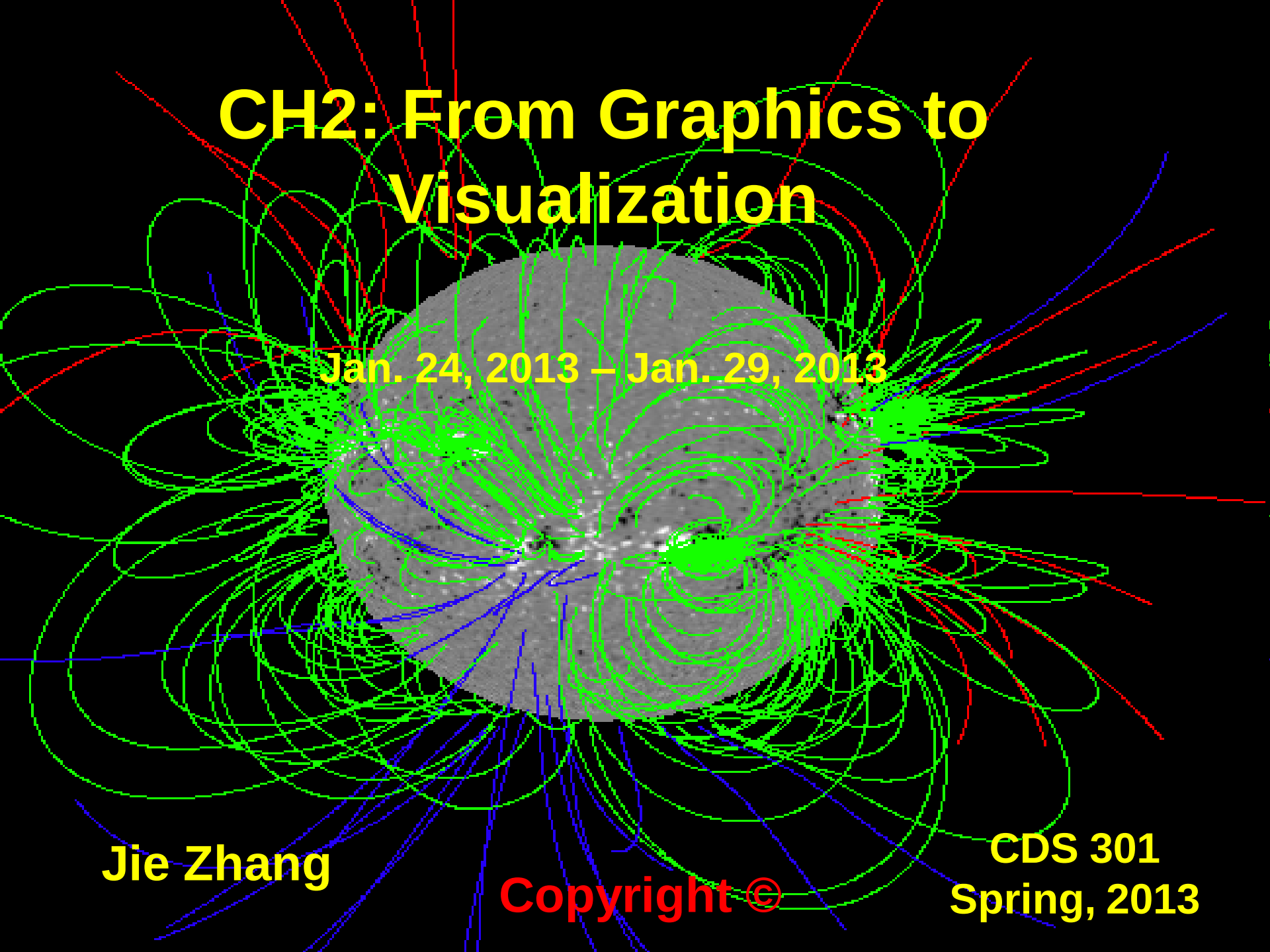
CH2: From Graphics to Visualization



Jie Zhang

CDS 301
Spring, 2013

Outline

- Example: make a 2-D Gaussian plot
- Data sampling
- Graphic Rendering
  - Light Rendering Model
- Texture Mapping
- Transparency and Blending
- Visualization Pipeline

2-D Gaussian Function

$$f(x, y) = e^{-(x^2+y^2)}$$

where

$$X = [-1,1], \quad Y = [-1,1]$$

$$D = [-1,1] \times [-1,1]$$

Compact Domain, Continuous Data

$$S \in R^3$$

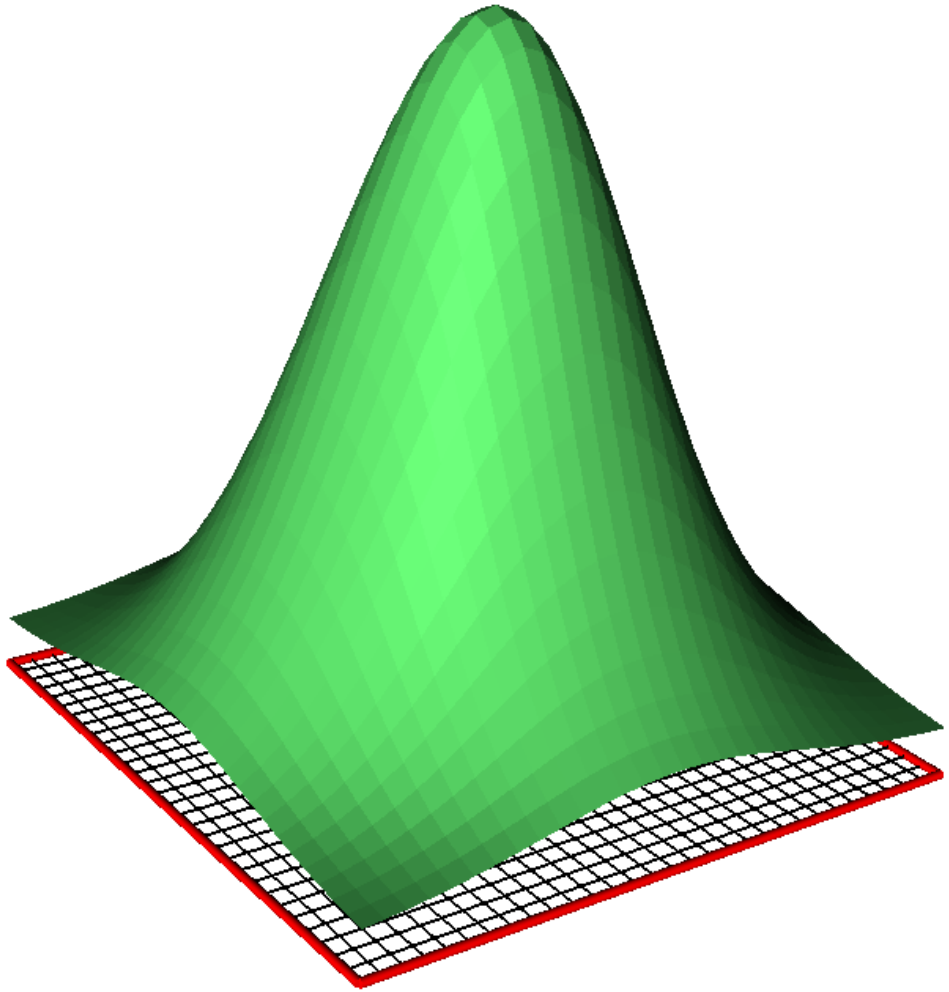
A Continuous 3D Surface in 3-D Space

2-D Gaussian Function

Sampling:

- **Nx=30**
- **Ny=30**
- **Uniform grid in the domain space**

- **Mapping the 3-D surface to**
  - **A set of four-vertex polygons, or quadrilateral**



2-D Gaussian Function

Quadrilateral **(quad)** in 3-D space:

V1: (x, y, f(x,y))
V2: (x+dx, y, f(x+dx,y)
V3: (x+dx, y+dy, f(x+dx,y+dy))
V4: (x, y+dy, f(x,y+dy))

These graphic primitives are rendered fast by computers

A **quad** has uniform illumination or lighting on its surface, so called **flat shading**

2-D Gaussian Function

Implementation:

```
float X_min, X_max;
float Y_min, Y_max;
int     N_x,N_y;
float  dx = (X_max-X_min)/N_x;
float  dy = (Y_max-Y_min)/N_y;
float  f(float,float);

for (float x=X_min;x<=X_max-dy;x+=dx)
  for (float y=Y_min;y<=Y_max-dy;y+=dy)
   {
      Quad, q;
       q.addPoint (x,y, f(x,y));
       q.addPoint (x+dx,y, f(x+dx,y));
      q.addPoint (x+dx,y+dy, f(x+dx,y+dy));
      q.addPoint (x,y+dy, f(x,y+dy));
   }
```

Rendering: the basics

•**Computer graphic rendering** generate a computer image of 3D scene, for a given dataset

•Ingredients:
- •A 3D scene
- •A set of Lights
- •A viewpoint

•How to determine the illumination of the primitive shapes in the 3-D scene?

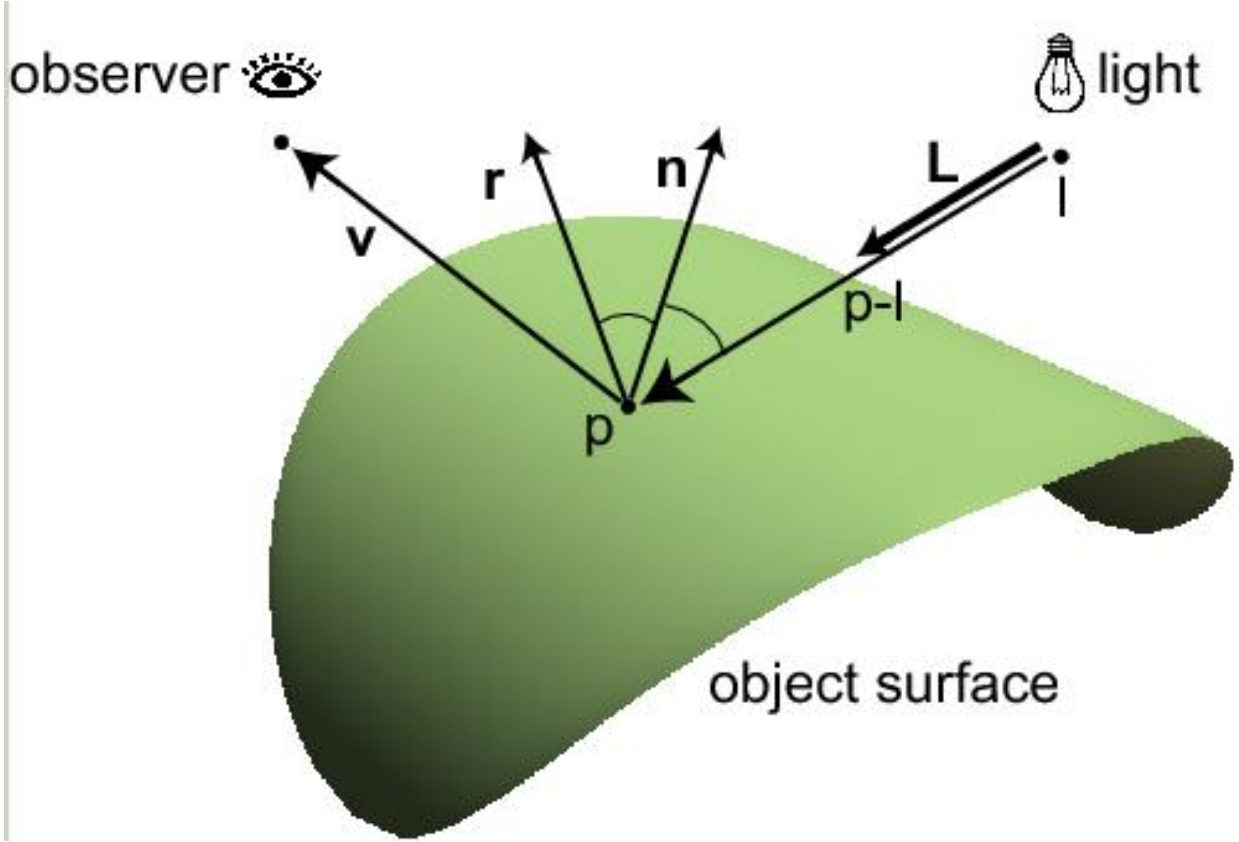
Rendering Equations

- Rendering methods differ in approximating lighting effects
  - Global illumination: ray tracing, accurate but computationally expensive
  - Local illumination: relate the illumination of a given scene point directly to the light set, not to any other scene points

- Phong Local Lighting Model (OpenGL)
  - Assuming that the scene consists of opaque objects in void space, illuminated by a point-like light source

Phong Local Lighting Model

$$I(p,\mathbf{v},\mathbf{L}) = I_l(C_{amb} + C_{diff}\max(-\mathbf{L}\cdot\mathbf{n},0) + C_{spec}\max(-\mathbf{r}\cdot\mathbf{V},0)^{\alpha})$$



Calculate Normal Direction

Using gradient:

$$\vec{\mathrm{n}} = (-\frac{\partial \mathrm{f}}{\partial \mathrm{x}}, -\frac{\partial \mathrm{f}}{\partial \mathrm{y}}, 1)$$

Using normal averaging:

$$\vec{\mathrm{n}}_{\mathrm{i}} = \frac{\sum_{p_j} \vec{n}(p_j)}{N}$$

Calculate reflection Direction

$$\vec{r} = \vec{L} - 2(\vec{L} \cdot \vec{n})\vec{n}$$

Phong Local Lighting Model

$$I(p,\mathbf{v},\mathbf{L}) = I_l(C_{amb} + C_{diff}\max(-\mathbf{L}\cdot\mathbf{n},0) + C_{spec}\max(-\mathbf{r}\cdot\mathbf{V},0)^{\alpha})$$

I(p,**v,L**): intensity of the scene point
I_l: Intensity of the light
p: location of the scene point
V: direction vector from p to the viewpoint
L: direction vector from the light to p
n: surface normal at p
r: direction of the reflecting light
α: specular power α

Three coefficients:

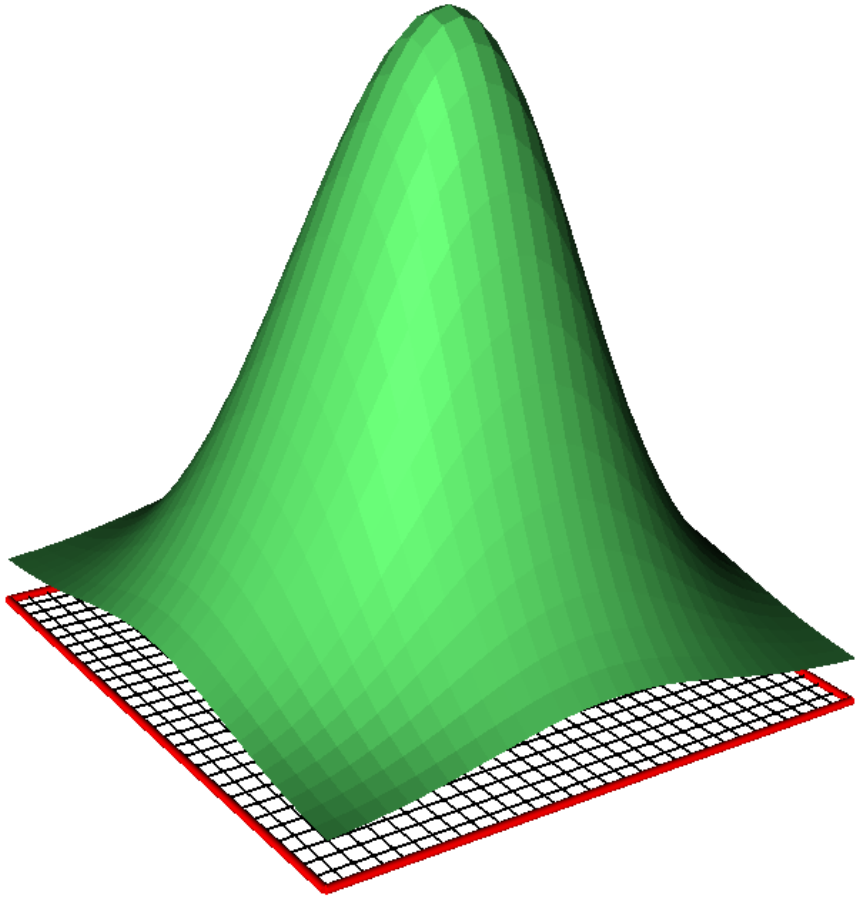
1. **C_{amb}: ambient lighting, overall effect of other objects, assuming to be constant**
2. **C_{diff}: diffuse lighting, scattering of the surface, equal in all direction; plastic surface**
3. **C_{spec}: specular lighting, mirror-like surface**

Primitive Shape Shading

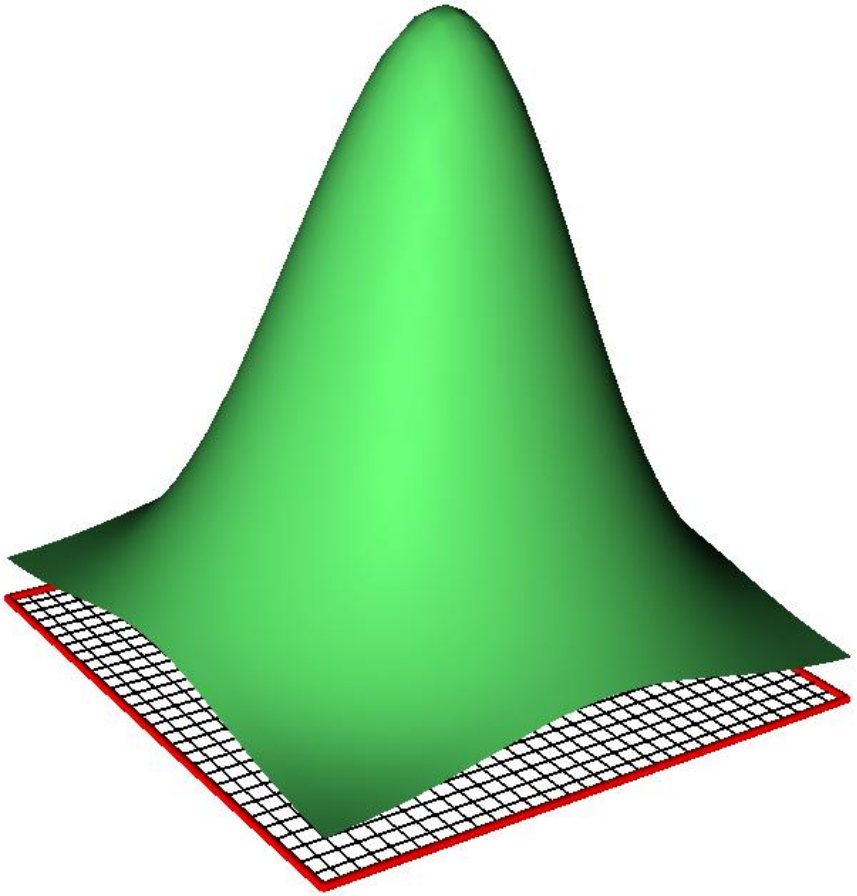
- **Flat shading:**
  - **Given a polygon surface (e.g., quadrilateral), flat shading applies the lightening model once for the complete polygon, e.g., for the center point**
  - **The whole polygon surface has the same light intensity**
  - **The least expensive**
  - **But visual artifact of the "faceted" surface**

- **Gouraud shading (or smooth shading):**
  - **Apply lightening model at every vertex of the polygon**
  - **The intensity between the vertices are calculated using interpolation, thus yielding smooth variation**

Primitive Shape Shading



Flat Shading



Gouraud Shading

Primitive Shape Shading

Implementation:

```
glShadingMode(GL_FLAT);

glColor3f(r,g,b)  ;set material color

glEnable(GL_LIGHTENING);

glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0,GL_POSITION,p);

glLightfv(GL_LightT0,GL_AMBIENT,I);
```

Jan. 29, 2013

Review

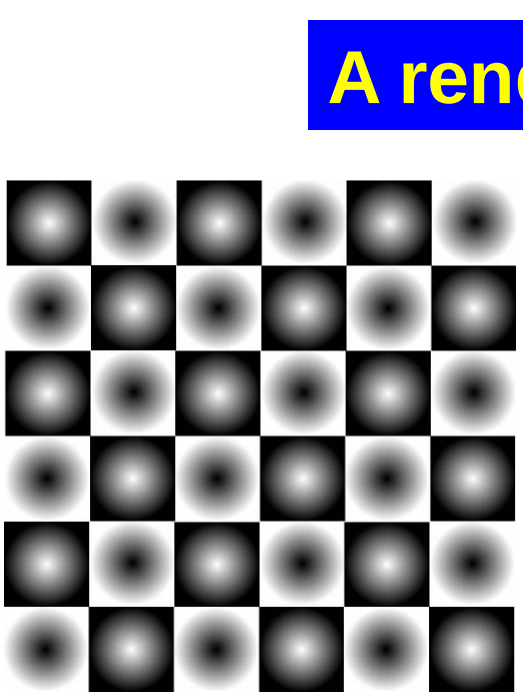
- Example: make a 2-D Gaussian plot
- Sampling
- Mapping
- Graphic Rendering
  - Light Rendering Model

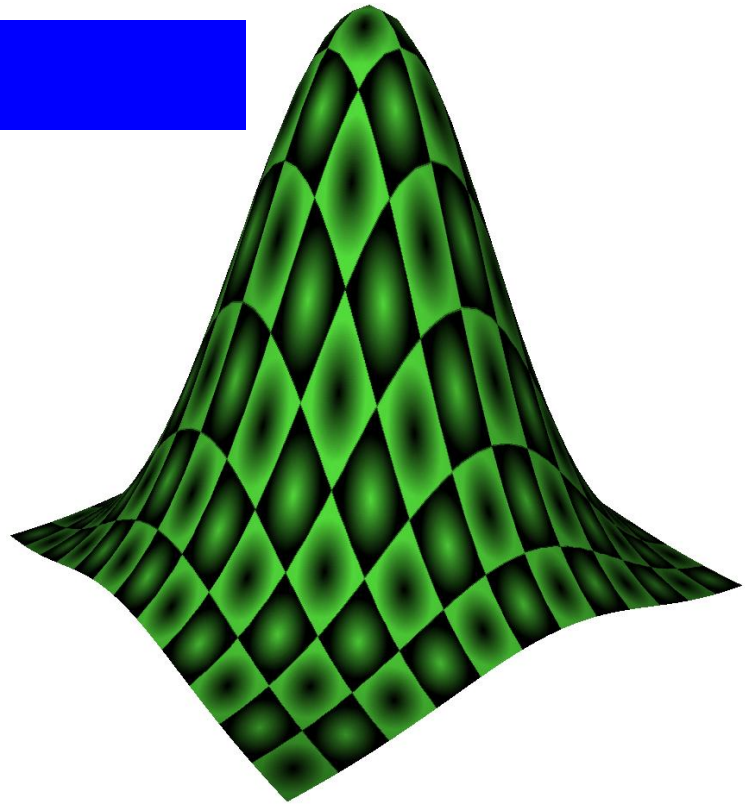
Then, finish up CH2
And move to CH3

Texture Mapping

- Effectively simulate a wide range of appearance on the surface of a rendered object
- Improve the realism of visualization
- Method: pre-define a texture image, mapping the polygon vertex coordinates to texture coordinates, and then combine the texture color with the polygon color

A rendering trick





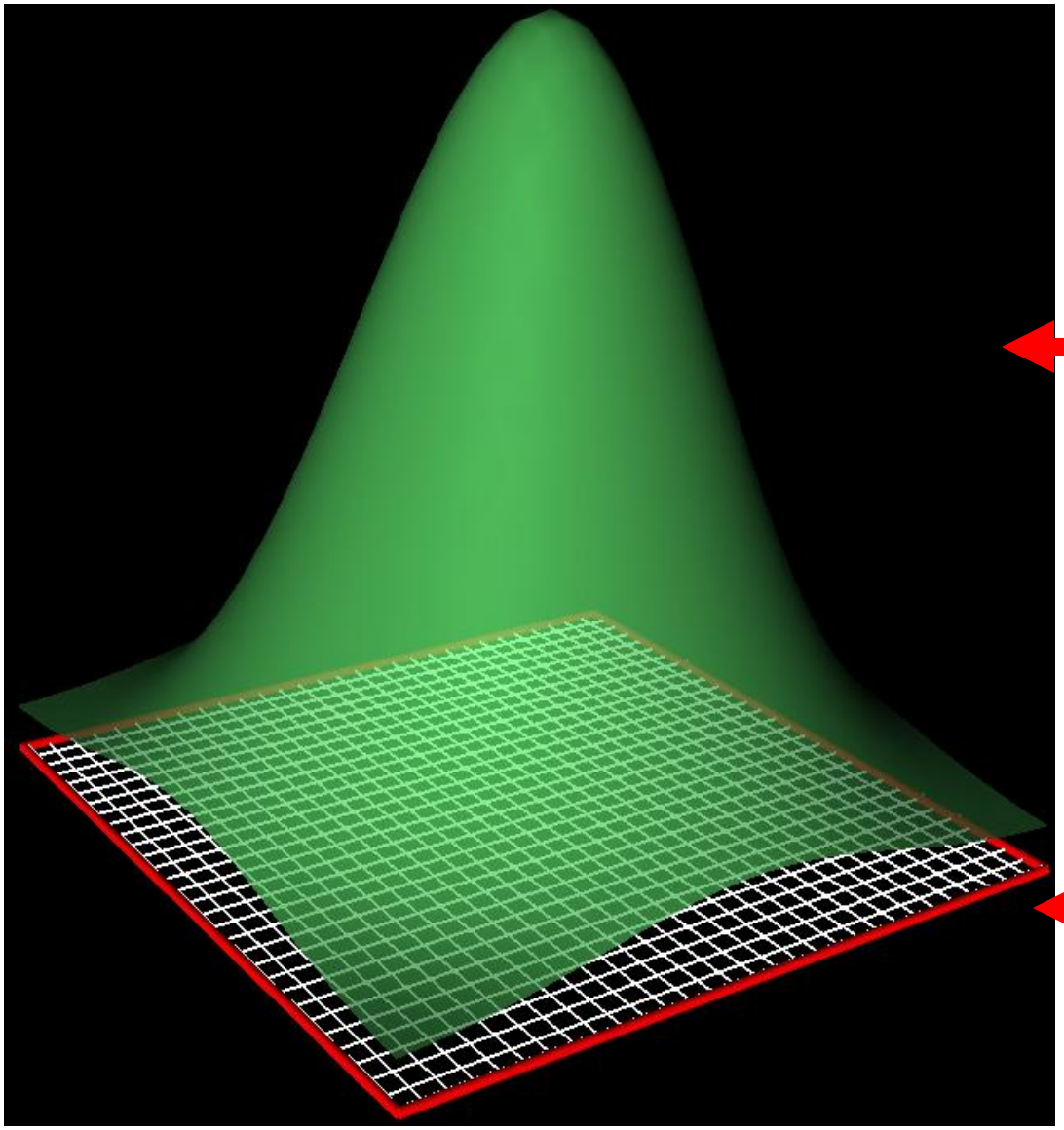
Transparency and Blending

- Rendering translucent (or half-transparent) shape
- See multiple shapes at the same time
  - e.g., the Gaussian surface and the underlying gridded domain

- Blending method (rendering the transparent effect)
  - Source: the shape is being drawn
  - Destination: the frame buffer (currently displayed image)
  - Source + Destination

Another rendering trick

Transparency and Blending



The Gaussian Surface

The Grid

Transparency and Blending

$$dst' = sf * source + df * destination$$

- **The blending output is a weighted combination of the source and the destination**
  - **sf: source weight factor, [0,1]**
    - **Also called alpha component**
  - **df: destination weight factor, [0,1]**
  - **sf and df: also called blending factors**

```
glBlendFunc(GL_SOURCE_ALPHA, GL_ONE);
glColor4f(r,g,b,sf)
```

Visualization Pipeline

Processes

Input/ Output

$$f(x, y) = e^{-(x^2+y^2)}$$

Continuous data

Data Acquisition

float data[N_x,N_y]

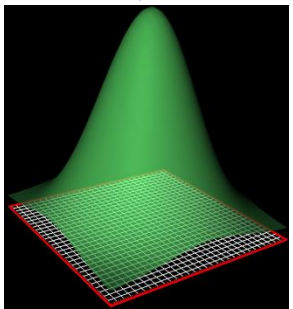
Discrete **dataset**

Data Mapping

Class Quad

Geometric object:
The fixed 3-D scene

Rendering



Displayed image

End