Métodos para Rendering de Superfície

SCC0250 - Computação Gráfica

Profa. Maria Cristina F. Oliveira

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

20 de novembro de 2023

Sumário

Sumário

Introdução

- A partir do modelo de iluminação calculado nos vértices, a **cor dos pixels** é determinada usando um método de **rendering de superfície**

- Há duas abordagens clássicas de rendering de superfícies
  - **Ray tracing**: o modelo de iluminação é calculado nas superfícies associadas a cada pixel projetado (realismo)
  - **Scanline**: modelo de iluminação é calculado nos vértices de ada polígono (associados a algum pixel) e os valores são interpolados para obter a iluminação nos demais pixels (tempo real)

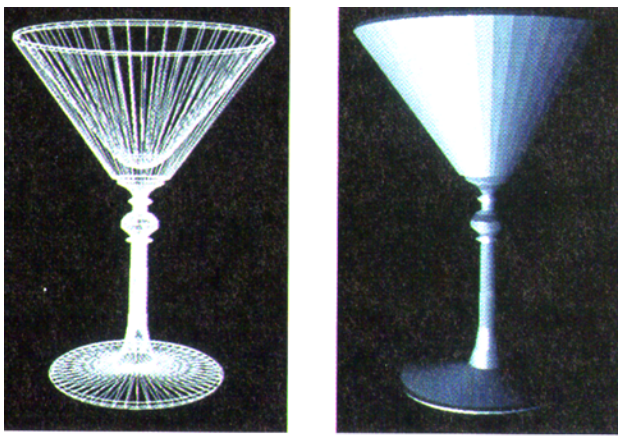
Métodos de Rendering de Superfície

- A maioria das APIs gráficas padrão adota o algoritmo *scanline*
  - Hardware gráfico atual otimizado para esse tipo de processamento
  - O modelo de iluminação é aplicado nos vértices dos polígonos para determinar a iluminação dos pixels correspondentes
  - As intensidades são interpoladas para obter a iluminação nos demais pixels dos polígonos

Rendering de Superfície: *Flat shading*

- O método mais simples de renderizar um polígono é usar a mesma cor em todos os seus pixels
- **Flat surface rendering** – ou rendering com intensidade constante

- O modelo de iluminação é computado uma única vez para determinar as intensidades das 3 componentes RGB em um ponto
  - Pode ser em um dos vértices, ou no centro de massa do polígono
  - Todos os demais pixels recebem essa mesma cor



(a) (b)

Rendering de Superfície: *Flat Shading*

- O **flat surface rendering** pode apresentar bons resultados se
  - O polígono é uma face de um poliedro e **não** uma seção de uma **superfície curva**
  - Todas as **fontes de luz** estão muito **distantes** do polígono, de modo que $\mathbf{N} \cdot \mathbf{L}$ e a função de **atenuação** são **constantes**
  - A **posição de observação** está **distante** o suficiente do polígono de modo que $\mathbf{V} \cdot \mathbf{R}$ (ou $\mathbf{N} \cdot \mathbf{H}$) é constante

$$I \quad = k_a \; I_a + \sum_{l=1}^{n} I_l [k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})^{n_s}]$$

- Mesmo se alguma dessas condições for falsa, ainda é possível obter uma boa aproximação visual se o modelo poligonal tem resolução alta (polígonos são pequenos)

Rendering de Superfície: *Gouraud shading*

- O esquema de **Gouraud surface rendering** consiste em calcular o modelo de iluminação nos vértices (i.e., nos pixels correspondentes) e interpolar linearmente as intensidades para obter as intensidades nos demais pixels do polígono

- O efeito é de suavizar a aparência dos modelos poligonais que aproximam superfícies curvas, amenizando as transições de intensidades entre polígonos adjacentes
  - Elimina as descontinuidades de intensidade características do **flat shading**

Rendering de Superfície: *Gouraud shading*

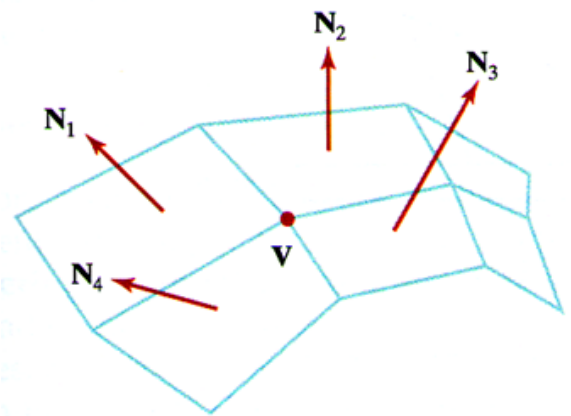
- Processa cada polígono de uma superfície com o seguinte procedimento
  1. Determina o vetor unitário normal médio em cada vértice do polígono
  2. Aplica o modelo de iluminação em cada vértice para obter as intensidades (R,G,B)
  3. Interpola linearmente as intensidades nos vértices sobre a área projetada do polígono

Rendering de Superfície: *Gouraud shading*

- O vetor normal médio em um vértice é dado pela média dos vetores normais de todos os polígonos que compartilham esse vértice (adjacentes a ele)

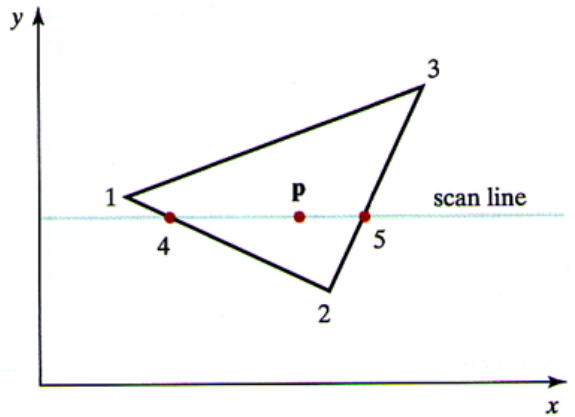
$$\mathbf{N}_V = \frac{\sum_{k=1}^{n} \mathbf{N}_k}{|\sum_{k=1}^{n} \mathbf{N}_k|}$$

- O modelo de iluminação é computado, utilizando esse vetor normal, para obter as intensidades em cada vértice



Rendering de Superfície: *Gouraud shading*

- Os valores de intensidade obtidos nos vértices são interpolados para obter as intensidades nos pixels ao longo das *scanlines* que intersectam a área projetada do polígono

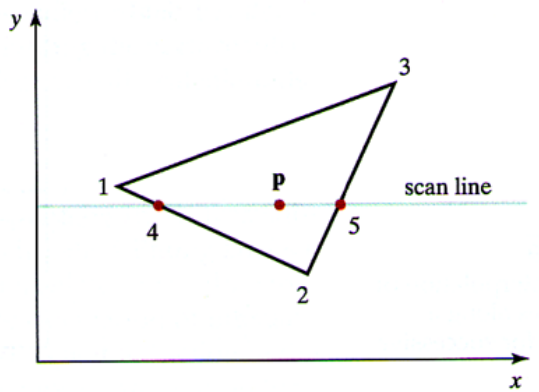


- As intensidades nos pontos em que as *scanlines* interceptam as arestas dos polígonos são obtidas interpolando linearmente as intensidades dos vértices extremos das arestas (as quais foram computadas aplicando o modelo de ilunminação)

Rendering de Superfície: *Gouraud shading*

- Por exemplo, a intensidade no ponto 4 pode ser calculada considerando somente o deslocamento vertical da *scanline* em relação aos vértices da aresta (pontos 1 e 2)

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2$$

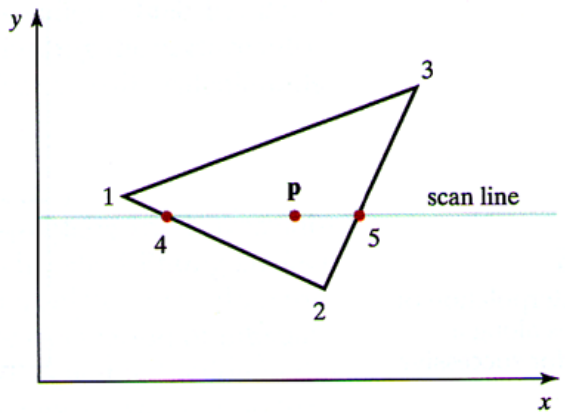


- A intensidade no ponto relativo ao vértice 5 pode ser obtida da mesma maneira, interpolando verticalmente as intensidades computadas nos pontos relativos aos vértices 2 e 3

Rendering de Superfície: *Gouraud shading*

- As intensidades em qualquer ponto/pixel $\mathbf{p}$ ao longo da *scanline* podem ser obtidas interpolando na horizontal as intensidades computadas nos pontos 4 e 5

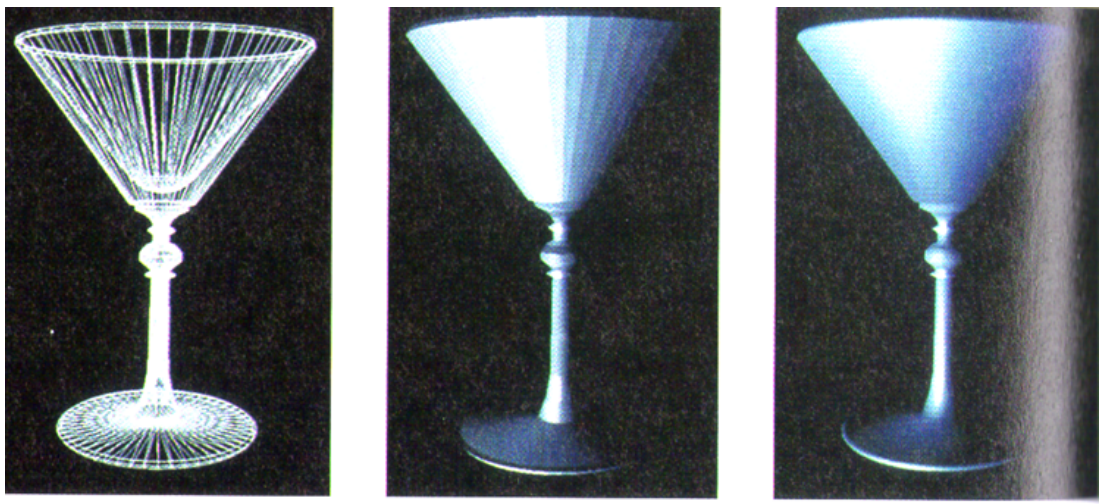
$$I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5$$



- Esse método é conhecido como **interpolação bilinear** e é executado separadamente para cada um dos 3 componentes RGB

Rendering de Superfície: *Gouraud shading*

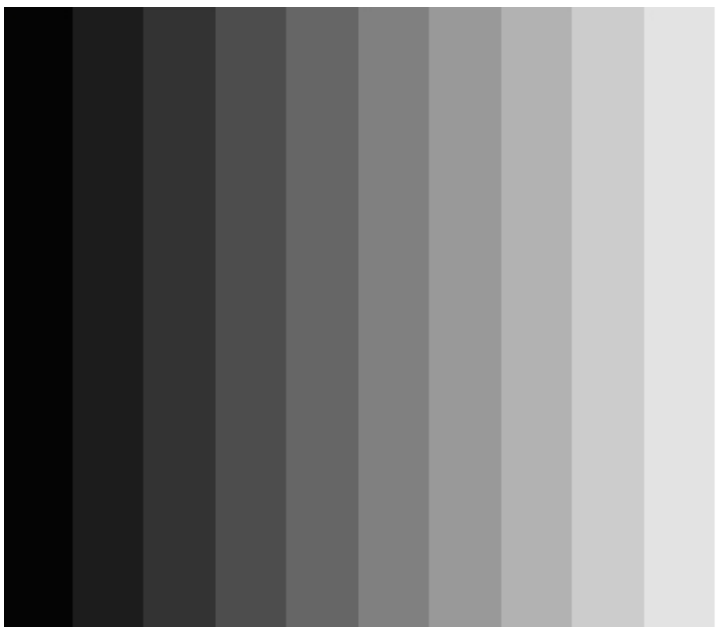
- Essa interpolação das intensidades elimina descontinuidades nas faces poligonais, mas tem alguns **problemas**
  - Brilhos na superfície podem apresentar formatos estranhos
  - Intensidades claras ou escuras podem parecer "riscadas" (**mach bands**)
  - Na figuira, (a) wireframe; (b) *flat shading*; (c) *Gouraud shading*



(a) (b) (c)

Rendering de Superfície: *Gouraud shading*

- Efeito de **mach bands** consiste em faixas claras ou escuras que são percebidas nas fronteiras entre duas regiões de diferentes gradientes de luz



Rendering de Superfície: *Phong shading*

- Um método mais preciso de interpolação é conhecido como **Phong shading**, ou *Phong surface rendering*

- Ao invés de interpolar os valores das intensidades computados nos vértices, interpola os vetores s normais nos vértices para obter as normais em cada ponto/pixel do polígono
- Aplica o modelo de iluminação em cada ponto/pixel usando o vetor normal obtido
  - Cálculos mais precisos de intensidades
  - Efeitos de brilho nas superfícies mais realísticos
  - Redução do efeito *mach-band*

- Computacionalmente mais caro do que o *Gouraud shading*

Rendering de Superfície: *Phong shading*

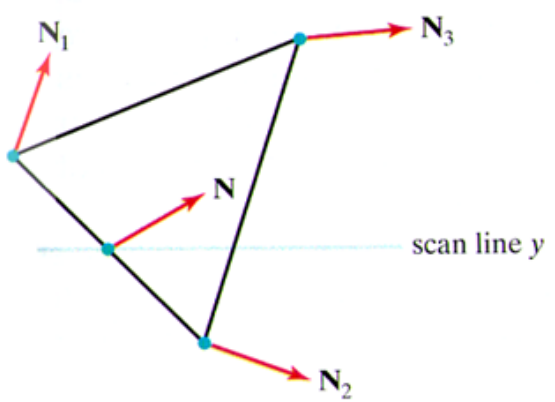
- Cada polígono é processado conforme o seguinte procedimento
  1. Determina o vetor unitário normal médio em cada vértice do polígono
  2. Interpola linearmente as normais dos vértices sobre a área projetada do polígono
  3. Aplica o modelo de iluminação nas posições ao longo da *scanline* para calcular as intensidades nos pixels, usando as normais interpoladas

Rendering de Superfície: *Phong shading*

- O procedimento de interpolação das normais é semelhante ao da interpolação das intensidades do método de Gouraud

- Por exemplo, o vetor normal $\mathbf{N}$ ao longo de uma aresta é obtido interpolando verticalmente os valores dos vetores normais nos vértices extremos da aresta:

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2}\mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2}\mathbf{N}_2$$

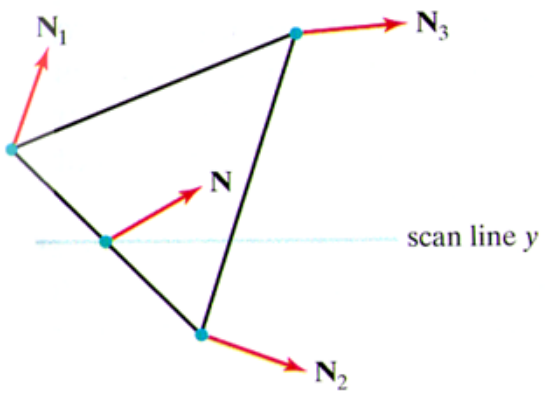


Rendering de Superfície: *Phong shading*

- O procedimento de interpolação das normais é semelhante ao da interpolação das intensidades do método de Gouraud

- Em seguida, as normais em cada ponto/pixel p ao longo da *scanline* são computadas interpolando horizontalmente os valores das normais computadas nos pontos de intersecção da scanline com as arestas

$$\mathbf{N}_p = \frac{x_5 - x_p}{x_5 - x_4}\mathbf{N}_4 + \frac{x_p - x_4}{x_5 - x_4}\mathbf{N}_5$$



Sumário

Funções de Rendering de Superfície

- OPENGL implementa dois métodos de rendering da superfície: (1) intensidade constante (*flat*); (2) e o modelo de Gouraud (*smooth shading*)
  - Não embute suporte para o modelo de Phong

- Para definir o método de rendering usamos

```
1 gl.glShadeModel(rendering_method);
```

- Em que *rendering_method* pode ser *GL.GL_FLAT* e *GL.GL_SMOOTH*

Funções de Rendering de Superfície

- Para definir o vetor normal usamos

```
1 gl.glNormal3*(Nx, Ny, Nz);
```

- O sufixo depende do tipo de parâmetro b, s, i, f e d, ou com a adição de v caso o parâmetro seja um vetor
  - Valores *byte*, *short* e *integer* são convertidos para valores de ponto flutuante no intervalo $[-1.0, 1.0]$

- O vetor normal é um valor de estado da OpenGL, com valor *default* igual a $(0.0, 0.0, 1.0)$

Funções de Rendering de Superfície

- Para rendering de intensidade constante define-se um único vetor normal para cada polígono

```
1 gl.glNormal3fv(normal_vector);
2
3 gl.glBegin(GL.GL_TRIANGLES);
4   gl.glVertex3fv(vertex1);
5   gl.glVertex3fv(vertex2);
6   gl.glVertex3fv(vertex3);
7 gl.glEnd();
```

Funções de Rendering de Superfície

- Para o rendering de Gouraud, define-se um vetor normal em cada vértice

```
1  gl.glBegin(GL_TRIANGLES);
2    gl.glNormal3fv(normal_vector1);
3    gl.glVertex3fv(vertex1);
4
5    gl.glNormal3fv(normal_vector2);
6    gl.glVertex3fv(vertex2);
7
8    gl.glNormal3fv(normal_vector3);
9    gl.glVertex3fv(vertex3);
10 gl.glEnd();
```

Funções de Rendering de Superfície

- Apesar dos vetores normais não precisarem ser especificados com tamanho unitário, fazer isso reduz o custo computacional

- É possível solicitar à OpenGL normalizar qualquer vetor normal que não seja unitário, com a função

```
1 gl.glEnable(GL.GL_NORMALIZE);
```

- Esse comando renormaliza todas as normais às superfícies, incluindo as que foram modificadas por transformações geométricas de escala e cisalhamento

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2
3      /** calcula o produto vetorial. */
4      private Point3D crossproduct(Point3D a, Point3D b) {
5          Point3D prod = new Point3D(a.y * b.z - a.z * b.y,
6                  a.z * b.x - a.x * b.z,
7                  a.x * b.y - b.x * a.y);
8          return prod;
9      }
10
11     /** normaliza um vetor de entrada. */
12     private Point3D normalize(Point3D p) {
13         float norm = (float) Math.sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
14         return new Point3D(p.x / norm, p.y / norm, p.z / norm);
15     }
16
17     /** Representa uma Clula com posio 3D e uma normal. */
18     class Cell {
19         public Point3D point;
20         public Point3D normal;
21     }
22
23     /** Representa um ponto ou vetor 3D (x,y,z). */
24     class Point3D {
25         public Point3D(float x, float y, float z) {
26             this.x = x; this.y = y; this.z = z;
27         }
28         public float x;
29         public float y;
30         public float z;
31     }
32 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3
4      /** cria uma matriz formando uma gaussiana. As primeiras e ltimas
5          linhas e colunas so criadas apenas como forma para facilitar
6          o clculo das normais em cada ponto. Essas no devem ser usadas
7          para o desenho da gaussiana. */
8      private Cell[][] createMatrix(int size) {
9          Cell[][] matrix = new Cell[size + 2][size + 2];
10
11         for (int i = 0; i < size + 2; i++) {
12             for (int j = 0; j < size + 2; j++) {
13                 matrix[i][j] = new Cell();
14                 float x = (4.0f * j) / ((float) size + 2) - 2.0f;
15                 float z = (4.0f * i) / ((float) size + 2) - 2.0f;
16                 float y = (float) (Math.exp(-x * x) * Math.exp(-z * z));
17                 matrix[i][j].point = new Point3D(x, y, z);
18             }
19         }
20
21         return matrix;
22     }
23 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3
4      /** calcula as normais em cada ponto. As primeiras e ltimas linhas
5          e colunas so ignoradas. Seja V1 o ponto para o qual ser calculada
6          a normal, V2 o ponto na mesma linha e prxima coluna e V3 o ponto
7          na prxima linha e mesma coluna. A normal  computada como o
8          produto vetorial do vetores de V1 para V2 e de V3 para V1. */
9      private void calculateNormal(Cell[][] matrix) {
10         for (int i = 1; i < matrix.length - 1; i++) {
11             for (int j = 1; j < matrix[0].length - 1; j++) {
12                 Point3D v1 = matrix[i][j].point;
13                 Point3D v2 = matrix[i][j + 1].point;
14                 Point3D v3 = matrix[i + 1][j].point;
15                 Point3D vec1 = new Point3D(v2.x-v1.x, v2.y-v1.y, v2.z-v1.z);
16                 Point3D vec2 = new Point3D(v1.x-v3.x, v1.y-v3.y, v1.z-v3.z);
17                 matrix[i][j].normal = normalize(crossproduct(vec1, vec2));
18             }
19         }
20     }
21 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3
4      public static void main(String[] args) {
5          //acelera o rendering
6          GLCapabilities caps = new GLCapabilities();
7          caps.setDoubleBuffered(true);
8          caps.setHardwareAccelerated(true);
9
10         //cria o painel e adiciona um ouvinte GLEventListener
11         GLCanvas canvas = new GLCanvas(caps);
12         canvas.addGLEventListener(new Rendering());
13
14         //cria uma janela e adiciona o painel
15         JFrame frame = new JFrame("Aplicao JOGL Simples");
16         frame.getContentPane().add(canvas);
17         frame.setSize(800, 800);
18         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19
20         //inicializa o sistema e chama display() a 60 fps
21         Animator animator = new FPSAnimator(canvas, 60);
22         frame.setLocationRelativeTo(null);
23         frame.setVisible(true);
24         animator.start();
25     }
26
27     ...
28 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3
4      public void init(GLAutoDrawable drawable) {
5          GL gl = drawable.getGL();
6          GLU glu = new GLU();
7
8          gl.glClearColor(1.0f, 1.0f, 1.0f, 1.0f); //define a cor de fundo
9          gl.glEnable(GL.GL_DEPTH_TEST); //habilita o teste de profundidade
10
11         gl.glMatrixMode(GL.GL_MODELVIEW); //define que a matrix  a model view
12         gl.glLoadIdentity(); //carrega a matrix de identidade
13         glu.gluLookAt(1.5, 0.75, 1.0, //posio da cmera
14                 0.0, 0.0, 0.0, //para onde a cmera aponta
15                 0.0, 1.0, 0.0); //vetor view-up
16
17         gl.glMatrixMode(GL.GL_PROJECTION); //define que a matrix  a de projeo
18         gl.glLoadIdentity(); //carrega a matrix de identidade
19         gl.glOrtho(-2.5, 2.5, -2.5, 2.5, -4.5, 4.5);
20
21         lighting(drawable); //defininido os parmetros de iluminao
22     }
23 }
```

Rendering de Superfície

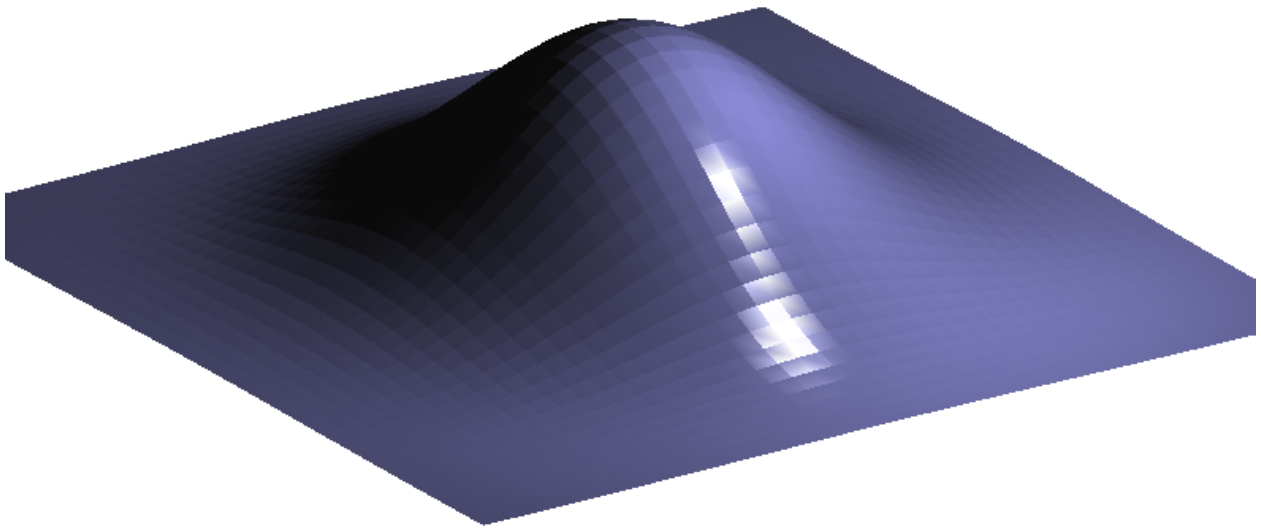
```
1  public class Rendering implements GLEventListener {
2      ...
3
4      private void lighting(GLAutoDrawable drawable) {
5          GL gl = drawable.getGL();
6
7          //define a posio e parmetros da luz 0
8          float position[] = {1.0f, 1.5f, -0.5f, 1.0f};
9          float white[] = {1.0f, 1.0f, 1.0f, 1.0f};
10         float black[] = {0.0f, 0.0f, 0.0f, 1.0f};
11         gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, position, 0);
12         gl.glLightfv(GL.GL_LIGHT0, GL.GL_AMBIENT, black, 0);
13         gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE, white, 0);
14         gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPECULAR, white, 0);
15
16         //ativa a iluminao
17         gl.glEnable(GL.GL_LIGHTING);
18         gl.glEnable(GL.GL_LIGHT0);
19     }
20 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3
4    public void display(GLAutoDrawable drawable) {
5      GL gl = drawable.getGL();
6      gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
7
8      //define as propriedades de reflexo da superfcie
9      float diffuse[] = {0.5f, 0.5f, 0.8f, 1.0f};
10     float specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
11     float shininess = 100.0f;
12     gl.glMaterialfv(GL.GL_FRONT, GL.GL_DIFFUSE, diffuse, 0);
13     gl.glMaterialfv(GL.GL_FRONT, GL.GL_SPECULAR, specular, 0);
14     gl.glMaterialf(GL.GL_FRONT, GL.GL_SHININESS, shininess);
15
16     //cria matrix com as clulas para desenho
17     Cell[][] m = createMatrix(40);
18     calculateNormal(m);
19
20     //desenha a superfcie
21     for (int i = 1; i < m.length - 1; i++) {
22       for (int j = 1; j < m.length - 1; j++) {
23         gl.glNormal3f(m[i][j].n.x, m[i][j].n.y, m[i][j].n.z);
24
25         gl.glBegin(GL.GL_TRIANGLE_STRIP);
26           gl.glVertex3f(m[i+1][j].p.x, m[i+1][j].p.y, m[i+1][j].p.z);
27           gl.glVertex3f(m[i+1][j+1].p.x, m[i+1][j+1].p.y, m[i+1][j+1].p.z);
28           gl.glVertex3f(m[i][j].p.x, m[i][j].p.y, m[i][j].p.z);
29           gl.glVertex3f(m[i][j+1].p.x, m[i][j+1].p.y, m[i][j+1].p.z);
30         gl.glEnd();
31       }
32     }
33
34     gl.glFlush(); //fora o desenho das primitivas
35   }
36 }
```

Rendering de Superfície

- Um vetor normal por face: tonalização da face é constante



Rendering de Superfície

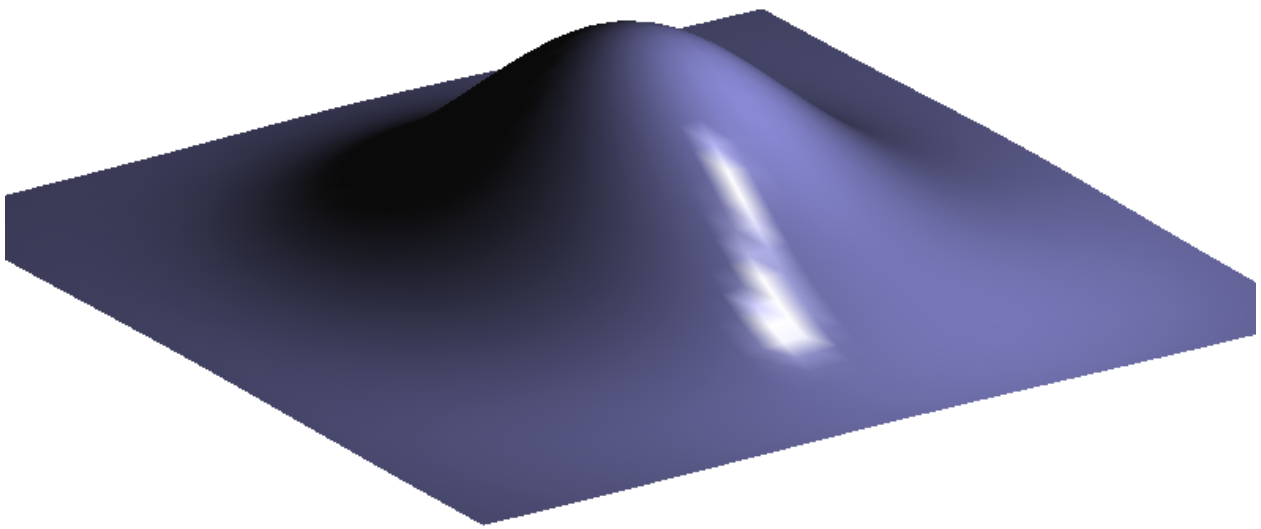
```
1  public class Rendering implements GLEventListener {
2    ...
3    public void display(GLAutoDrawable drawable) {
4      ...
5      for (int i = 1; i < m.length - 1; i++) {
6        for (int j = 1; j < m.length - 1; j++) {
7          gl.glBegin(GL.GL_TRIANGLE_STRIP);
8          if (m[i+1][j].n != null) {
9            gl.glNormal3f(m[i+1][j].n.x, m[i+1][j].n.y, m[i+1][j].n.z);
10         }
11         gl.glVertex3f(m[i+1][j].p.x, m[i+1][j].p.y, m[i+1][j].p.z);
12
13         if (m[i+1][j+1].n != null) {
14           gl.glNormal3f(m[i+1][j+1].n.x, m[i+1][j+1].n.y, m[i+1][j+1].n.z);
15         }
16         gl.glVertex3f(m[i+1][j+1].p.x, m[i+1][j+1].p.y, m[i+1][j+1].p.z);
17
18         if (m[i][j].n != null) {
19           gl.glNormal3f(m[i][j].n.x, m[i][j].n.y, m[i][j].n.z);
20         }
21         gl.glVertex3f(m[i][j].p.x, m[i][j].p.y, m[i][j].p.z);
22
23         if (m[i][j+1].n != null) {
24           gl.glNormal3f(m[i][j+1].n.x, m[i][j+1].n.y, m[i][j+1].n.z);
25         }
26         gl.glVertex3f(m[i][j+1].p.x, m[i][j+1].p.y, m[i][j+1].p.z);
27         gl.glEnd();
28       }
29     }
30     ...
31   }
32 }
```

Rendering de Superfície

```
1  public class Rendering implements GLEventListener {
2      ...
3      private void lighting(GLAutoDrawable drawable) {
4          GL gl = drawable.getGL();
5
6          //define a posio e parmetros da luz 0
7          float position[] = {1.0f, 1.5f, -0.5f, 1.0f};
8          float white[] = {1.0f, 1.0f, 1.0f, 1.0f};
9          float black[] = {0.0f, 0.0f, 0.0f, 1.0f};
10
11         gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, position, 0);
12         gl.glLightfv(GL.GL_LIGHT0, GL.GL_AMBIENT, black, 0);
13         gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE, white, 0);
14         gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPECULAR, white, 0);
15
16         //ativa a iluminao
17         gl.glEnable(GL.GL_LIGHTING);
18         gl.glEnable(GL.GL_LIGHT0);
19
20         //ativa rendering 'smooth'
21         gl.glShadeModel(GL.GL_SMOOTH); //ou GL.GL_FLAT para 'flat';
22     }
23 }
```

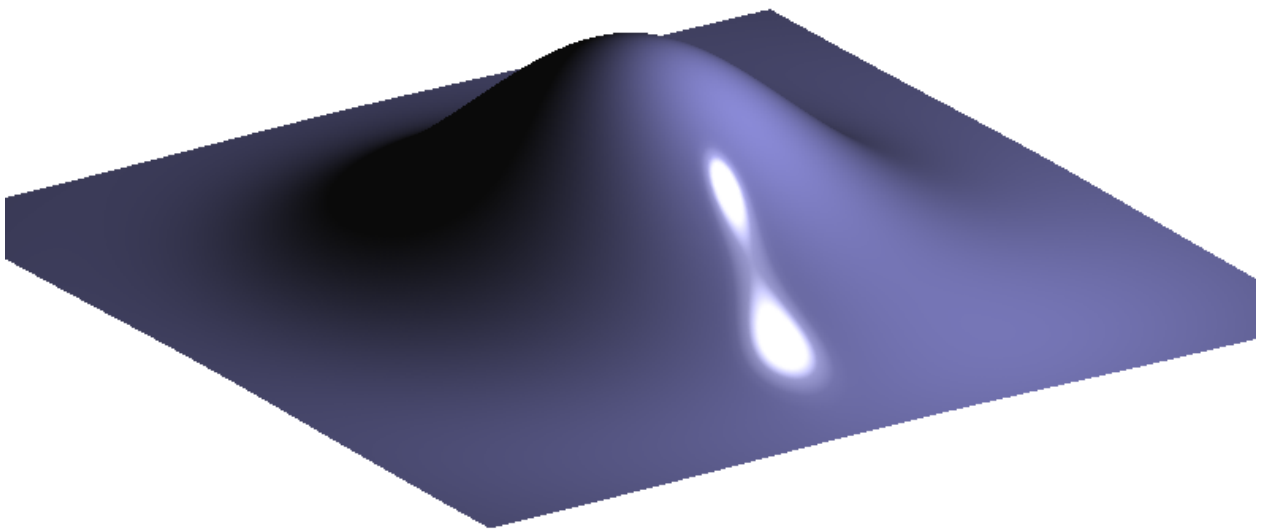
Rendering de Superfície

- Um vetor normal por vértice: a tonalização da face é contínua
  - Geometria poligonal: matriz com 40×40 faces: a qualidade visual da reflexão especular é ruim



Rendering de Superfície

- Melhorando a qualidade da reflexão especular aumentando a resolução do modelo: matriz com 400×400 faces
  - Mas a opção mais razoável não pé aumenta a resolução do modelo. O mellhor e usar GLSL e implementar *Phong shading*



Sumário

OpenGL Shading Language

- **OpenGL Shading Language (GLSL)** é uma linguagem de alto nível que permite aos desenvolvedores maior controle sobre o pipeline gráfico, sem precisar se preocupar com aspectos específicos de hardware
  - Incluída na versão 2.0 de OpenGL
  - Similar a linguagem C, suportando laços, comandos de decisão, etc.

- Alguns benefícios são
  - Compatibilidade entre diferentes plataformas, incluindo linux, Mac OS e Windows
  - Possibilidade de escrever *shaders* que podem ser usados em sistemas gráficos que suportam GLSL

OpenGL Shading Language

- GLSL shaders são conjuntos de strings que são passados para a placa gráfica para compilação diretamente de dentro de uma aplicação OpenGL
  - Podem ser criados *"on the fly"* de dentro de uma aplicação ou lidos a partir de arquivos texto

- Cada placa gráfica inclui um compilador GLSL em seu *driver* – código pode ser otimizado para utilizar as particularidades da placa

OpenGL Shading Language

- Um exemplo bem simples de GLSL que transforma um vértice da mesma forma do pipeline original

```
1  void main(void)
2  {
3    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
4  }
```

- E colore os fragmentos de verde

```
1  void main(void)
2  {
3    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
4  }
```

Vertex Shader (Phong Shading)

- Usando a GLSL é possível implementar o modelo de *rendering* de Phong
  - É preciso criar um **vertex shader** para transferir os dados dos vértices para o **fragment shader**
  - A posição do vértice e o vetor normal ao vértice são passados para o *fragment shader*, e são automaticamente interpolados

```
1  /* vertex shader for per-fragment Phong shading */
2
3  varying vec3 normal;
4  varying vec4 position;
5
6  void main(void)
7  {
8     normal = gl_NormalMatrix * gl_Normal;
9     position = gl_ModelViewMatrix * gl_Vertex;
10    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; //ou = ftransform();
11 }
```

- deve-se usar *gl_NormalMatrix* no lugar da *gl_ModelViewMatrix* para garantir que os vetores normais não sejam afetados por escalas não uniformes

Fragment Shader (Phong Shading)

```
1  /* fragment shader for per-fragment Phong shading */
2
3  varying vec3 normal;
4  varying vec4 position;
5
6  void main()
7  {
8    vec3 normv = normalize(normal); //normal
9    vec3 lightv = normalize(gl_LightSource[0].position.xyz - position.xyz); //luz
10   vec3 viewv = normalize(-position.xyz); //direo de viso
11   vec3 halfv = normalize(lightv + viewv); //vetor mdio
12
13   //calculando a componente ambiente
14   vec4 ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
15
16   //calculando a componente difusa
17   float diff = max(0.0, dot(lightv, normv));
18   vec4 diffuse = diff * gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
19
20   //calculando a componente especular
21   float spec = max(0.0, pow(dot(halfv, normv), gl_FrontMaterial.shininess));
22   vec4 specular = spec * gl_FrontMaterial.specular * gl_LightSource[0].specular;
23
24   //calculando a cor final
25   gl_FragColor = ambient + diffuse + specular;
26 }
```

OpenGL Shading Language

Figura: Diferença de fazer o *shading* considerando os vértices (à esquerda) e considerando os fragmentos (à direita).

- Para maiores informações consulte `http://www.opengl.org/documentation/glsl/`

Sumário

Usando Shaders

```
1  public class RenderingSmoothGLSL implements GLEventListener {
2    ...
3
4    public void init(GLAutoDrawable drawable) {
5      //ativa modo debug da jogl
6      drawable.setGL(new DebugGL(drawable.getGL()));
7
8      GL gl = drawable.getGL();
9      GLU glu = new GLU();
10
11     gl.glClearColor(1.0f, 1.0f, 1.0f, 1.0f); //define a cor de fundo
12     gl.glEnable(GL.GL_DEPTH_TEST); //habilita o teste de profundidade
13
14     gl.glMatrixMode(GL.GL_MODELVIEW); //define que a matrix  a model view
15     gl.glLoadIdentity(); //carrega a matrix de identidade
16     glu.gluLookAt(1.5, 0.75, 1.0, //posio da cmera
17             0.0, 0.0, 0.0, //para onde a cmera aponta
18             0.0, 1.0, 0.0); //vetor view-up
19
20     gl.glMatrixMode(GL.GL_PROJECTION); //define que a matrix  a de projeo
21     gl.glLoadIdentity(); //carrega a matrix de identidade
22     gl.glOrtho(-2.5, 2.5, -2.5, 2.5, -4.5, 4.5);
23
24     lighting(drawable); //defininido os parmetros de iluminao
25     shader(drawable); //ativando o shader
26   }
27 }
```

Carregando Shaders

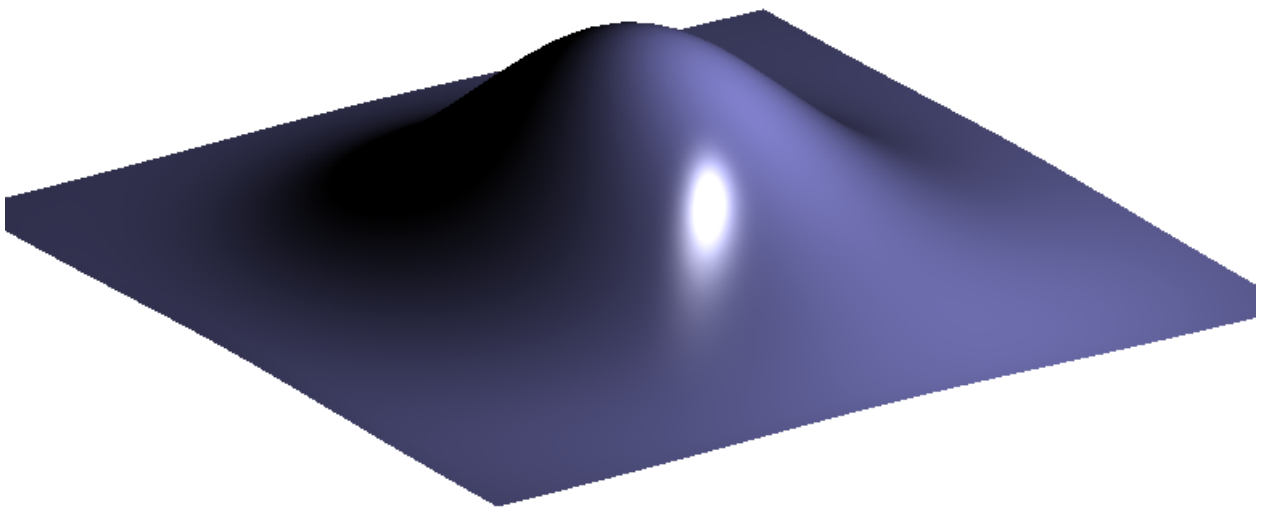
```
1  public class RenderingSmoothGLSL implements GLEventListener {
2    ...
3
4    private void shader(GLAutoDrawable drawable) {
5      GL gl = drawable.getGL();
6
7      //cria o fragment e vertex shaders
8      int v = gl.glCreateShader(GL.GL_VERTEX_SHADER);
9      int f = gl.glCreateShader(GL.GL_FRAGMENT_SHADER);
10
11     //carrega e compila o vertex shader
12     String[] vsrc = readShader("/shaders/phong_vertex_shader.txt");
13     gl.glShaderSource(v, 1, vsrc, null, 0);
14     gl.glCompileShader(v);
15
16     //carrega e compila o fragment shader
17     String[] fsrc = readShader("/shaders/phong_fragment_shader.txt");
18     gl.glShaderSource(f, 1, fsrc, null, 0);
19     gl.glCompileShader(f);
20
21     //anexa, valida e usa os shaders
22     int shaderprogram = gl.glCreateProgram();
23     gl.glAttachShader(shaderprogram, v);
24     gl.glAttachShader(shaderprogram, f);
25     gl.glLinkProgram(shaderprogram);
26     gl.glValidateProgram(shaderprogram);
27     gl.glUseProgram(shaderprogram);
28   }
29 }
```

Lendo Arquivo de Shading

```
1  public class RenderingSmoothGLSL implements GLEventListener {
2    ...
3
4    /** L um arquivo de shader no formato texto. */
5    private String[] readShader(String shadername) {
6      try {
7        InputStream isv = getClass().getResourceAsStream(shadername);
8        BufferedReader brv = new BufferedReader(new InputStreamReader(isv));
9        StringBuilder buffer = new StringBuilder();
10       String line;
11       while ((line = brv.readLine()) != null) {
12         buffer.append(line).append("\r\n");
13       }
14       brv.close();
15       isv.close();
16       return new String[]{buffer.toString()};
17     } catch (IOException ex) {
18       Logger.getLogger(getClass().getName()).log(Level.SEVERE, null, ex);
19     }
20
21     return null;
22   }
23 }
```

Phong Shader GLSL

- Melhorando a reflexão especular usando GLSL



Shaders

- Exemplo de um *shader* simples: a cor do pixel é dada pelo posição de um vértice

```
1  /* vertex shader */
2  varying vec3 vertex_color;
3
4  void main(void)
5  {
6    gl_Position = ftransform();
7    vertex_color = gl_Vertex.xyz; //cor ser a posio do vrtice
8  }
```

```
1  /* fragment shader */
2  varying vec3 vertex_color;
3
4  void main(void)
5  {
6    gl_FragColor = vec4(vertex_color, 1.0);
7  }
```

Resultado

- Isso também mostra que **vertex_color** é interpolado quando passado do *vertex shader* para o *fragment shader*