CS559: Computer Graphics

Lecture 15: B-Spline, Lighting, and Shading

Li Zhang

Spring 2010
Beziers Curve Subdivision

• Why is subdivision useful?
  – Collision/intersection detection
    • Recursive search
  – Good for curve editing and approximation
Open Curve Approximation
Closed Curve Approximation
<table>
<thead>
<tr>
<th>Type</th>
<th>Interpolate control points</th>
<th>Has local control</th>
<th>C2 continuity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural cubics</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Hermite cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cardinal Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bezier Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Interpolate control points</td>
<td>Has local control</td>
<td>C2 continuity</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------</td>
<td>-------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Natural cubics</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Hermite cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cardinal Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bezier Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bspline Curves</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Bsplines

• Given p1, ... pn, define a curve that approximates the curve.

\[ f(t) = \sum_{i=1}^{n} b_i(t)p_i \]

If \( b_i(t) \) is very smooth, so will be \( f \).

If \( b_i(t) \) has local support, \( f \) will have local control.
Uniform Linear B-splines

\[ b_{i,2}(t) = b_{0,2}(t - i) \]

\[ b_{0,2}(u) = \begin{cases} 
  u & u \in [0,1) \\
  2 - u & u \in [1,2) \\
  0 & \text{otherwise}
\end{cases} \]

\[ f(t) = \sum_{i=1}^{n} b_i(t) p_i \]
How can we make the curve smooth?

- Convolution/filtering

\[ f(t) = \sum_{i=1}^{n} b_i(t)p_i \]

\[ f(t) \otimes \text{Box}(t) = \left( \sum_{i=1}^{n} b_i(t)p_i \right) \otimes \text{Box}(t) \]

\[ = \left( \sum_{i=1}^{n} \left( b_i(t) \otimes \text{Box}(t) \right)p_i \right) \]
Uniform Quadratic B-splines

\[ f(t) = \sum_{i=1}^{n} b_i(t)p_i \]
Uniform Cubic Bspline

\[ f(t) = \sum_{i=1}^{n} b_i(t)p_i \]
Uniform B-splines

• Why smoother?
  – Linear = box filter \( \otimes \) box filter
  – Quadric = linear \( \otimes \) box filter
  – Cubic = quadric \( \otimes \) box filter

• Sum = 1 property, translation invariant

• Local control

\[
f(t) = \sum_{i=1}^{n} b_i(t)p_i
\]

• \( C(k-2) \) continuity
<table>
<thead>
<tr>
<th>Curve Type</th>
<th>Interpolate control points</th>
<th>Has local control</th>
<th>C2 continuity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural cubics</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Hermite cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cardinal Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bezier Cubics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bspline Curves</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
So far...

• We’ve talked exclusively about geometry.
  – What is the shape of an object?
    • glBegin() ... glEnd()
  – How do I place it in a virtual 3D space?
    • glMatrixMode() ...
  – How to change viewpoints
    • gluLookAt()
  – How do I know which pixels it covers?
    • Rasterization
  – How do I know which of the pixels I should actually draw?
    • Z-buffer, BSP
So far

```c
glColor(...);
Apply_transforms();
Draw_objects();
```
Next...

• Once we know geometry, we have to ask one more important question:
  – To what value do I set each pixel?
• Answering this question is the job of the shading model.
• Other names:
  – Lighting model
  – Light reflection model
  – Local illumination model
  – Reflectance model
  – BRDF
An abundance of photons

• Properly determining the right color is really hard.
An abundance of photons

- Properly determining the right color is really hard.
An abundance of photons

• Properly determining the right color is really hard.
An abundance of photons

- Properly determining the right color is really hard.

Global Effect
Our problem

• We’re going to build up to an approximation of reality called the Phong illumination model.

• It has the following characteristics:
  – not physically based
  – gives a “first-order” approximation to physical light reflection
  – very fast
  – widely used

• In addition, we will assume local illumination, i.e., light goes: light source -> surface -> viewer.

• No interreflections, no shadows.
Setup...

- Given:
  - a point $P$ on a surface visible through pixel $p$
  - The normal $N$ at $P$
  - The lighting direction, $L$, and intensity, $I$, at $P$
  - The viewing direction, $V$, at $P$
  - The shading coefficients at $P$

- Compute the color, $I$, of pixel $p$.

- Assume that the direction vectors are normalized:
“Iteration zero”

• The simplest thing you can do is...
• Assign each polygon a single color:

\[ l = k_e \]

• where
  – \( l \) is the resulting intensity
  – \( k_e \) is the **emissivity** or intrinsic shade associated with the object

• This has some special-purpose uses, but not really good for drawing a scene.
“Iteration one”

• Let’s make the color at least dependent on the overall quantity of light available in the scene:

\[ I = k_e + k_a L_a \]

- \( k_a \) is the **ambient reflection coefficient**.
  • really the reflectance of ambient light
  • “ambient” light is assumed to be equal in all directions
- \( L_a \) is the **ambient light intensity**.

• Physically, what is “ambient” light?
Ambient Term

• Hack to simulate multiple bounces, scattering of light
• Assume light equally from all directions
Wavelength dependence

• Really, $k_e$, $k_a$, and $L_a$ are functions over all wavelengths $\lambda$.

• Ideally, we would do the calculation on these functions. For the ambient shading equation, we would start with:

$$I(\lambda) = k_a(\lambda)L_a(\lambda)$$

• then we would find good RGB values to represent the spectrum $I(\lambda)$.

• Traditionally, though, $k_a$ and $I_a$ are represented as RGB triples, and the computation is performed on each color channel separately:

$$I_R = k_{a,R}L_{a,R}$$
$$I_G = k_{a,G}L_{a,G}$$
$$I_B = k_{a,B}L_{a,B}$$
Diffuse reflection

\[ I = k_e + k_a L_a \]

• So far, objects are uniformly lit.
  – not the way things really appear
  – in reality, light sources are localized in position or direction

• **Diffuse**, or **Lambertian** reflection will allow reflected intensity to vary with the direction of the light.
Diffuse reflectors

- Diffuse reflection occurs from dull, matte surfaces, like latex paint, or chalk.
- These **diffuse** or **Lambertian** reflectors reradiate light equally in all directions.
Diffuse reflectors

• Diffuse reflection occurs from dull, matte surfaces, like latex paint, or chalk.
• These **diffuse** or **Lambertian** reflectors reradiate light equally in all directions.
• Picture a rough surface with lots of tiny **microfacets**.
Diffuse reflectors

- ...or picture a surface with little pigment particles embedded beneath the surface (neglect reflection at the surface for the moment):

- The microfacets and pigments distribute light rays in all directions.
- Embedded pigments are responsible for the coloration of diffusely reflected light in plastics and paints.
- Note: the figures above are intuitive, but not strictly (physically) correct.
Diffuse reflectors, cont.

- The reflected intensity from a diffuse surface does not depend on the direction of the viewer. The incoming light, though, does depend on the direction of the light source:
“Iteration two”

• The incoming energy is proportional to \( \cos \theta \), giving the diffuse reflection equations:

\[
I = k_e + k_a L_a + k_d L \cdot (L \cdot N)
\]

\[
= k_e + k_a L_a + k_d L \cdot \max(0, L \cdot N)
\]

• where:
  – \( k_d \) is the **diffuse reflection coefficient**
  – \( L_d \) is the intensity of the light source
  – \( N \) is the normal to the surface (unit vector)
  – \( L \) is the direction to the light source (unit vector)
Gouraud interpolation artifacts

• Gouraud interpolation has significant limitations.
  – If the polygonal approximation is too coarse, we can miss specular highlights.
Phong interpolation

• To get an even smoother result with fewer artifacts, we can perform **Phong interpolation**.

• Here’s how it works:
  1. Compute normals at the vertices.
  2. Interpolate normals and normalize.
  3. Shade using the interpolated normals.
Gouraud vs. Phong interpolation
How to compute vertex normals

A weighted average of normals of neighboring triangles

\[
\mathbf{n}_{\text{vertex}} = \frac{\sum_{\text{triangle}} \text{area}_{\text{triangle}} \mathbf{n}_{\text{triangle}}}{\sum_{\text{triangle}} \text{area}_{\text{triangle}}}
\]
How to compute vertex normals

A weighted average of normals of neighboring triangles

\[
\mathbf{n}_{\text{vertex}} = \frac{\sum_{\text{triangle}} \text{area}_{\text{triangle}} \mathbf{n}_{\text{triangle}}}{\sqrt{\sum_{\text{triangle}} \text{area}_{\text{triangle}}^2}}
\]