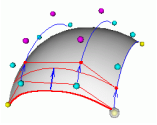
Advanced Texture-Mapping
Curves and Curved Surfaces





CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
July 26, 2005

Pre-Lecture Business

- loadtexture example
- midterm handed back, code posted
- (still) get going on pp3!
- more on texturing
- review quiz

Texture Modes

Useless parameter... always GL_TEXTURE_ENV

```
glTexEnvi(GL_TEXTURE_ENV,
      GL_TEXTURE_ENV_MODE,
      mode);
```

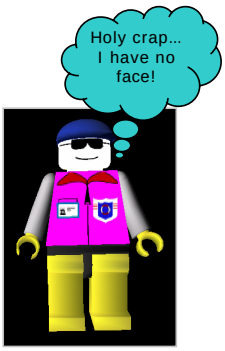
Tell GL what to do with my textures

Tell GL I'm changing the texture mode...

untextured | GL_DECAL | GL_MODULATE

Texture Modes

What happened to lego man's face?

What does a texture map for a complex model "look like"?


Review quiz

- What is *shading*?
- What is *lighting*?
- What is the difference between *smooth shading* and *flat shading*?
- Why do we interpolate texture coordinates in 3D instead of in 2D?

Outline for today

- Advanced texture mapping
- Texture coordinate generation
- Curves and curved surfaces
- The OpenGL pipeline revisited

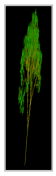
Advanced Texture Mapping

- Billboarding (easiest)
- Bump Mapping (in between)
- Environment Mapping (hardest)

- Some form of any of these would be great extra credit for your pp's…

Billboarding

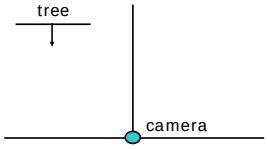
- Sometimes I can really get away with letting entire objects be 2D
  - Objects that are far-away
  - Objects that look the same from everywhere, like particles of dust…
- An easy way to render 2D objects that look nice is to just use a textured quad…



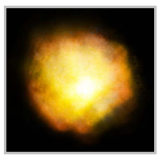
Billboarding

- So a good trick is to make sure "billboard" objects *always* face the camera
- Assume we know the camera is at (cx,cy,cz) and my tree is at (tx,ty,tz)
- Assume we know how to render a quad facing +z
- **How can I do this in OpenGL?**

When does this approach break down, even if I'm still far away? I.e., when will my tree not look right?



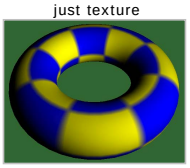
Billboarding Examples



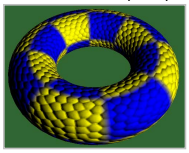
Bump mapping (aka normal mapping)

- Instead of modulating the *color* at each pixel, what if we modulated the *normal* at each pixel?
- Lots more detail without more vertices…

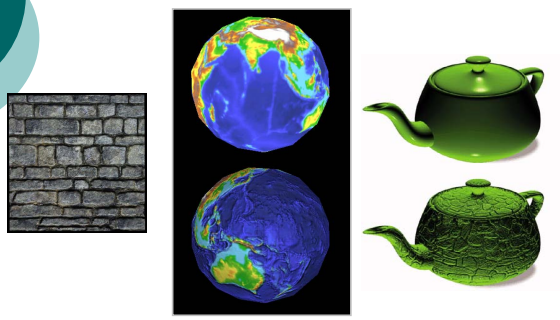
just texture



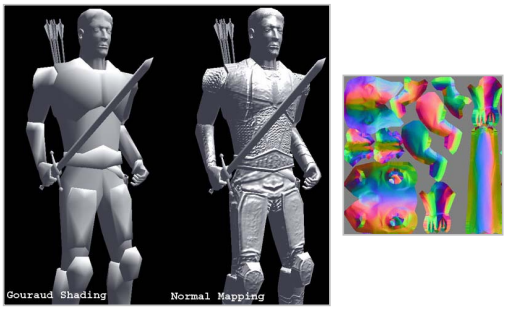
texture + bump map



Bump mapping examples



What does a bump map “look like”?



Bump mapping in OpenGL (overview)

- Store all the surface normals in one texture
- Store a vector from each vertex to our light in the other texture
- Using an *OpenGL extension*, tell OpenGL to dot these two textures, i.e. perform the *diffuse lighting computation* at each pixel

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
    GL_COMBINE_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
    GL_DOT3_RGB_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,
    GL_PREVIOUS_ARB);
```

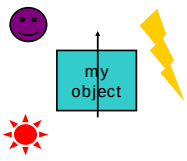
Why does bump mapping come with a big performance penalty?

Environment Mapping

- “Shiny” objects should reflect light from the objects around them and act like mini-mirrors
- OpenGL lighting can’t do this
  - In fact OpenGL lighting ignores all other objects in the scene when it lights each vertex…
- But we can approximate this with textures…

Environment Mapping: Building a Texture

- Imagine putting my OpenGL camera at the position of an object and taking a panoramic picture of the world, then storing the result in a texture…
  - This really means “rendering a few times with the camera pointed in different directions”

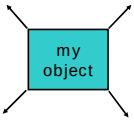


1D environment map of a 2D scene

2D environment map of a 3D scene

Environment Mapping: Generating Texture Coordinates

- Now imagine that I assign texture coordinates to my object by just subtracting the object center from each vertex position…
- I.e., each texture coordinate tells me “which way this vertex looks”



texture coordinates

Environment Mapping: The Result

- If I get the mapping right, this will let me paste “a picture of the environment” on my object

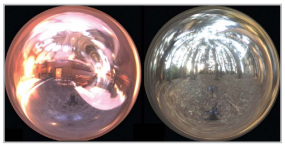
Environment Mapping: Limitations

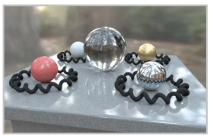
Why is it hard to do this in real-time?

- In practice, static environment maps are often used and look pretty good…

Environment Mapping: Relighting

- Can also capture *real* panoramic pictures of the world and use them to “re-light” virtual objects
- Often done by photographing a mirrored ball





Outline for today

- Advanced texture mapping
- Texture coordinate generation
- Curves and curved surfaces
- The OpenGL pipeline revisited

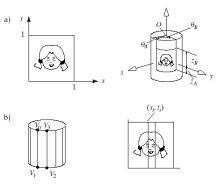
Texture Coordinate Generation

- Where did all these texture coordinates come from?
- Generally part of the *modeling* process: textures are built to line up with the texcoords on one specific object

- What tools are available to modelers to assign texture coordinates to an object?

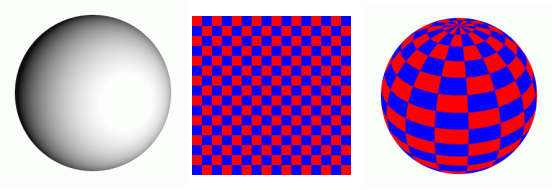
Texture Coordinate Generation

- In general, mapping a flat surface to a 3-D object is difficult
- Some objects – like cylinders – have a natural mapping, because you can wrap a flat sheet around them
  - I.e. if I gave you a piece of wrapping paper and told you to paste it onto a cylinder it would be pretty straightforward



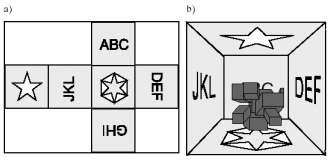
Texture Coordinate Generation

- Some objects – like spheres – will always cause distortion when wrapped with a texture
  - I.e. if I gave you a piece of wrapping paper and told you to paste it onto a bowling ball, you would run into problems



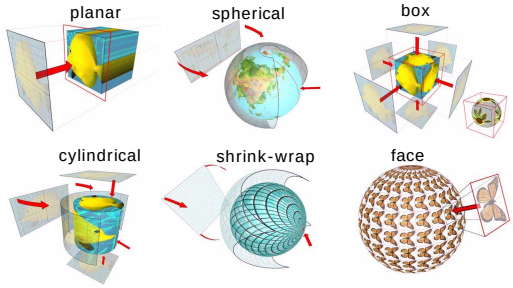
Texture Coordinate Generation

- In general, modeling programs know how to generate textures in small patches that fit well



Texture Coordinate Generation

- Modeling programs usually offer various tools for generating texture coordinates:



TexGen in OpenGL

- Usually your texture coordinates come from a model file or are generated explicitly
- But OpenGL can also generate texture coordinates on-the-fly…

```
glEnable(GL_TEXTURE_GEN_T);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE,
    GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE,
    GL_SPHERE_MAP);
```

- …now whenever I send vertices to OpenGL, it will generate sphere map coordinates for me.

When might I want to do this (something we talked about earlier today)?

Projective Textures

- A real strength of using OpenGL's texture generation is that it can generate planar texture coordinates on a plane that faces the eye or faces some point…
- This lets us do *projective texturing*:

Outline for today

- Advanced texture mapping
- Texture coordinate generation
- Curves and curved surfaces
- The OpenGL pipeline revisited

Curves and Curved Surfaces

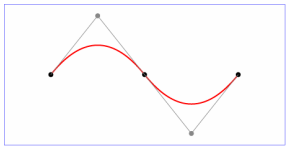
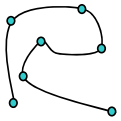
- Bezier Curves
- B-Splines
- Bezier Surfaces

Why do we need more curves?

- We talked about ruled surfaces, surfaces of revolution, quadrics, etc.
- Most real curved objects can't be built from these simple primitives

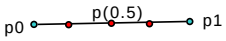

Control Points

- The most common approach to defining curves is to specify a set of control points: "my curve should go somewhere near these points"
- A mathematical function or algorithm can automatically generate the curve based on the control points

Take One: Linear Interpolation

- Let's generate a line segment between p0 and p1, parameterized by *t* on the interval [0,1]
- $p(t) = (1-t)p_0 + (t)p_1$

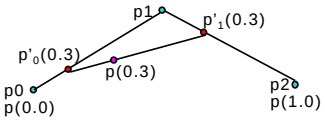


- We could use this to interpolate lots of points if we wanted to…

Why do we need fancier methods?

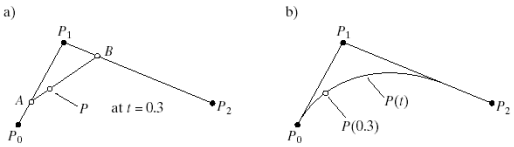
de Casteljau's Algorithm

- I want to generate a curve controlled by three points.
- We'll parameterize on t = [0,1]
- To find p(t), interpolate between p0 and p1, and interpolate between p1 and p2
- Then interpolate between those two points to find p(t)



Bezier Curves

- If we do this for lots of t values, we get a smooth curve:



- This is called a *Bezier* curve, because Monsieur Bezier developed the algebraic form of the same curve that de Casteljau generated

Bezier Curves: Closed Form

- What's the closed-form expression for p(t)?
- For our curve with only three points:

$p'_0(t) = (1-t)p_0 + (t)p_1$
$p'_1(t) = (1-t)p_1 + (t)p_2$

$p(t) = (1-t)*p'_0(t) + t*p'_1(t)$

$p(t) = (1-t)^2 * p_0 + 2t(1-t) * p_1 + t^2 * p_2$

- This curve is of degree 2 (a parabola)
- We *could* do the same math for any degree (any number of control points)

Bezier Curves: Bernstein Form

- Fortunately someone else already did that math for us…
- For a Bezier curve with L control points p_0, p_1, p_2, … p_L, the Bezier curve is:

$$p(t) = \sum_{k=0}^{L} p_k B_k^{\,L}(t)$$

- $B_k^L(t)$ is a *Bernstein polynomial*:

$$B_k^{\,L}(t) = \binom{L}{k}(1-t)^{L-k} * t^k$$

B-polynomials are *blending* functions

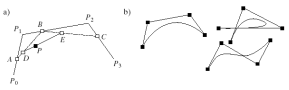
- The closed form expression again:

$$p(t) = \sum_{k=0}^{L} p_k B_k^{\,L}(t)$$

- The Bezier curve is a *blend* of the control points
- Bernstein polynomials control how much weight each control point gets; we call them *blending functions*
- There are lots of different blending functions out there… how did the Bernstein functions get to be so popular?

Nice properties of Bezier curves

- Endpoint interpolation:
  - A Bezier curve always passes through the first and last control points
- Transformation invariance:
  - I can transform the control points and the curve will transform "correctly"
- Convex hull preservation:
  - Bezier curves stay within the convex hull of the control points
- Smoothness at endpoints:
  - The slope of the curve at an endpoint is the same as the slope of the last "control segment"
- Cubic Bezier functions are particularly popular…



Bezier curves in OpenGL [bezcurve.cpp]

- OpenGL *evaluators* take t values and control points and generate vertices for you:

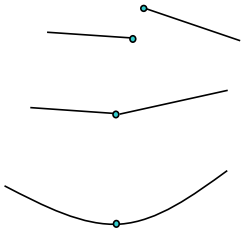
```
// GL, when I tell you to, evaluate a Bezier curve
// with order N and these control points…
glMap1f(GL_MAP1_VERTEX_3, 0, 1, 3, N,
    controlpoints);

// GL, please generate a vertex for the value t, using
// the Bezier curve I told you about previously
glEvalCoord1f(t);

// GL, please generate 30 points on my curve from
// t = 0 to t = 1 and draw them
glMapGrid1f(30, 0.0, 1.0);
glEvalMesh1(GL_LINE_STRIP, 0, 30);
```

Less-than-nice properties of Bezier curves

- Global control:
  - Moving any point has an effect on the entire curve
- Complexity:
  - Typical curves may have hundreds of control points, and evaluating high-degree Bezier curves is impractical
- **How can we get all the nice properties of Bezier curves with *local* control and low complexity if I have lots of control points?**

An aside: curve continuity

- A curve or surface is said to be C^n continuous at a point *t* if its n^{th} derivative at that point is continuous



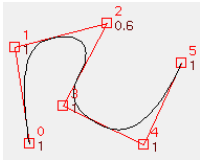
not c^0 continuous

c^0 continuous
not c^1 continuous

c^1 continuous
hard to tell about
higher orders…

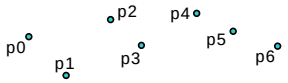
Cubic B-splines

- Piecewise approximations of cubic polynomial functions with C^0, C^1, and C^2 continuity
- Translation: stitch together a bunch of cubics without weird artifacts at the joints



Cubic B-splines: The Big Picture

- Segments of the curve are influenced by *four* control points



- First segment: cubic curve using p0, p1, p2, p3
- Next segment: cubic curve using p1, p2, p3, p4
- etc…

Cubic B-splines: The Math [bspline.cpp]

- Represent each segment as a standard cubic:

$$x(t) = a_3t^3 + a_2t^2 + a_1t + a_0$$
$$y(t) = b_3t^3 + b_2t^2 + b_1t + b_0$$

- What are the coefficients? They should represent the control points somehow…
- After much derivation, we would get a common formula that gives nice cubics:

$$a_3 = (-x_{i-1} + 3x_i - 3x_{i+1} + x_{i+2}) / 6$$
$$a_2 = (x_{i-1} - 2x_i + x_{i+1}) / 2$$
$$a_1 = (-x_{i-1} + x_{i+1}) / 2$$
$$a_0 = (x_{i-1} + 4x_i + x_{i+1}) / 6$$

- $x_{i-1} \rightarrow x_{i+2}$ are the four points that control this segment

Cubic B-splines: Nice Properties

- Local Control
  - Moving or adding a control point doesn't affect the whole curve
- Low degree
  - Cubics are easy to compute and are used to describe many complex curves
- All the nice properties of Bezier curves

B-splines: Limitations

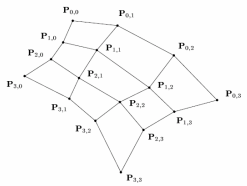
- Not guaranteed to pass through any of the control points (even the first and the last)
- Still can't express all shapes (e.g. circles)
- We've only talked about *uniform* B splines
  - If there are 5 control cubics, each cubic determines 1/5 of the overall curve
- We've only talked about *non-rational* B-splines
  - If we put these in Bernstein form, our blending functions would look a lot like the Bezier blending functions
- We can fix the above problems with
  - *non-uniform* B-splines: arbitrary influence regions for each point
  - *rational* B-splines: blending functions are *ratios* of polynomials
- The state of the art in OpenGL curves is NURBS (an important buzzword): non-uniform rational B-splines
- See your textbook for more information…

Review Quiz
(candy for correctness and brevity)

- What is *texture mapping*?
- What is *billboarding*?
- What is *bump mapping*?
- Why not use bump mapping all the time?
- What is *environment mapping*?
- Why not use environment mapping all the time?
- Why do we usually define curves with control points instead of with lots of vertices?
- What's a *Bezier curve*?
- What's a *B-spline*?
- What advantages do B-splines have over *Bezier curves*?

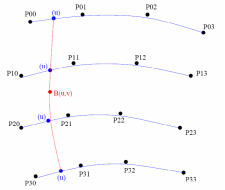
Going from curves to surfaces

- Everything we've learned in 1D scales nicely to 2D (from lines to surfaces)…
- A *Bezier patch* is the 2D cousin of a Bezier curve
- For a Bezier patch, we specify a *grid* of control points that we want the surface to "look like"



Going from curves to surfaces

- To find a point p(s,t) on a Bezier patch:
  - Generate the points p(s) for each of the four curves along one axis
  - Use the four resulting points to as a new Bezier curve, and generate the point p(t) along that curve



If I just knew how to evaluate p(u,v), how could I render a Bezier patch in OpenGL?

Going from curves to surfaces

- This was our Bezier curve: $p(t) = \sum_{k=0}^{L} p_k B_k^{\,L}(t)$
- Now let the control points themselves be functions of another variable:

$$p(s,t) = \sum_{k=0}^{L} p_k(s) B_k^{\,L}(t)$$

- Specifically, let those functions be Bezier curves:

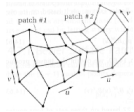
$$p_k(s) = \sum_{j=0}^{M} p_{j,k} B_j^{\,M}(s)$$

- This gives us the function for a Bezier patch:

$$p(s,t) = \sum_{j=0}^{M} \sum_{k=0}^{L} p_{j,k} B_j^{\,M}(s) B_k^{\,L}(t)$$

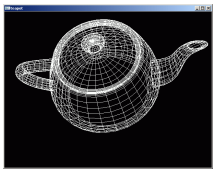
Connecting Bezier patches

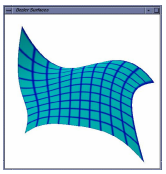
- Just as we did with curves, we'll often build complex surfaces by piecing together Bezier patches or B-spline patches

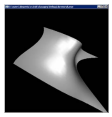


- Continuity is not guaranteed with Bezier patches… it's often up to the designer or the modeling software to make sure that corresponding points have the same positions and derivatives

Bezier and B-spline surfaces: Examples







Bezier surfaces in OpenGL [bezmesh.cpp]

- OpenGL *evaluators* take u and v values and control points and generate vertices for you:

```
// GL, when I tell you to, evaluate a Bezier surface
// with order N and these control points…
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4,
    controlpoints);

// GL, please generate a vertex for the value t, using
// the Bezier curve I told you about previously
glEvalCoord2f(s,t);

// GL, please generate 400 points on my surface from
// u = 0 to u = 1 and v = 0 to v = 1 and draw them
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
glEvalMesh2(GL_FILL, 0, 20, 0, 20);
```

Outline for today

- Advanced texture mapping
- Texture coordinate generation
- Curves and curved surfaces
- The OpenGL pipeline revisited

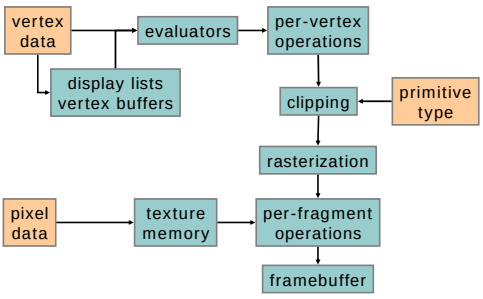
Terminology: fragment

- A fragment is an like a baby pixel that might or might not grow up to be a pixel (appear in the framebuffer)
- Fragments are the output of the rasterizer

Why might a fragment generated by the rasterizer not appear in the framebuffer?

What data does OpenGL store with each fragment?

The OpenGL Pipeline Revisited



Next Time

- Selection
- Transparency