**Objectives**

- Introduce types of curves and surfaces
  - Explicit
  - Implicit
  - Parametric
  - Strengths and weaknesses
- Discuss Modeling and Approximations
  - Conditions
  - Stability

**Escaping Flatland**

- Until now we have worked with flat entities such as lines and flat polygons
  - Fit well with graphics hardware
  - Mathematically simple
- But the world is not composed of flat entities
  - Need curves and curved surfaces
  - May only have need at the application level
  - Implementation can render them approximately with flat primitives

**Modeling with Curves**

- Inverse with curves and surfaces
- Need curves and curved surfaces
- May only have need at the application level
- Implementation can render them approximately with flat primitives

**What Makes a Good Representation?**

- There are many ways to represent curves and surfaces
- Want a representation that is
  - Stable
  - Smooth
  - Easy to evaluate
  - Must we interpolate or can we just come close to data?
  - Do we need derivatives?

**Explicit Representation**

- Most familiar form of curve in 2D
  \( y = f(x) \)
- Cannot represent all curves
  - Vertical lines
  - Circles
- Extension to 3D
  - \( z = f(x,y) \) defines a surface
Implicit Representation

- Two dimensional curve(s)
  \[ g(x,y) = 0 \]
- Much more robust
  - All lines \( ax + by + c = 0 \)
  - Circles \( x^2 + y^2 = r^2 \)
- Three dimensions \( g(x,y,z) = 0 \) defines a surface
  - Intersect two surfaces to get a curve
- In general, we cannot exactly solve for points that satisfy the equation

Algebraic Surface

\[ \sum_{i,j,k} a_{ijk} x^i y^j z^k \]

- Quadratic surface \( 2 \geq i + j + k \)
- At most 10 terms
- Can solve intersection with a ray by reducing problem to solving quadratic equation

Parametric Curves

- Separate equation for each spatial variable
  \[ x = x(u) \]
  \[ y = y(u) \]
  \[ z = z(u) \]
- For \( u_{\text{max}} \geq u \geq u_{\text{min}} \) we trace out a curve in two or three dimensions
- Want functions which are easy to evaluate
  - Computation of normals
- Want functions which are easy to differentiate
  - Connecting pieces (segments)
- Want functions which are smooth

Selecting Functions

- Usually we can select “good” functions
  - not unique for a given spatial curve
  - Approximate or interpolate known data
  - Want functions which are easy to evaluate
    - Want functions which are easy to differentiate
      - Computation of normals
      - Connecting pieces (segments)
- Want functions which are smooth

Parametric Lines

We can normalize \( u \) to be over the interval \((0,1)\)
- Line connecting two points \( p_0 \) and \( p_t \)
  \[ p(u) = (1-u)p_0 + up_t \]
  \[ p(0) = p_0 \]
  \[ p(1) = p_t \]
- Ray from \( p_0 \) in the direction \( d \)
  \[ p(u) = p_0 + ud \]
  \[ p(0) = p_0 \]
  \[ p(1) = p_0 + d \]

Curve Segments

- After normalizing \( u \), each curve is written
  \[ p(u) = [x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0 \]
- In classical numerical methods, we design a single global curve
- In computer graphics and CAD, it is better to design small connected curve segments

Parametric Polynomial Curves

- \( x(u) = \sum_{i=0}^{N} c_{xi}u^i \)
- \( y(u) = \sum_{j=0}^{M} c_{yj}u^j \)
- \( z(u) = \sum_{k=0}^{K} c_{zk}u^k \)

- If \( N=M=K \), we need to determine \( 3(N+1) \) coefficients
- Equivalently, we need \( 3(N+1) \) independent conditions
- Noting that the curves for \( x, y \), and \( z \) are independent, we can define each independently in an identical manner
- We will use the form where \( p \) can be any of \( x, y, \) or \( z \)

\[
\begin{align*}
  x(u) &= \sum_{i=0}^{N} c_{xi}u^i \\
y(u) &= \sum_{j=0}^{M} c_{yj}u^j \\
z(u) &= \sum_{k=0}^{K} c_{zk}u^k
\end{align*}
\]

Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
- Must worry about continuity at join points including continuity of derivatives

Cubic Parametric Polynomials

- \( N=M=L=3 \), gives balance between ease of evaluation and flexibility in design
- Four coefficients to determine for each of \( x, y \), and \( z \)
- Seek four independent conditions for various values of \( u \) resulting in 4 equations in 4 unknowns for each of \( x, y \), and \( z \)
- Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data

\[
\begin{align*}
p(u) &= \sum_{i=0}^{3} c_{pi}u^i
\end{align*}
\]

Cubic Polynomial Surfaces

- \( p(u,v) = [x(u,v), y(u,v), z(u,v)]^T \)
- \( p(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{ij}u^i v^j \)
- \( p \) is any of \( x, y \), or \( z \)
- Need 48 coefficients (3 independent sets of 16) to determine a surface patch

Objectives

- Introduce the types of curves
  - Interpolating
  - Hermite
  - Bezier
  - B-spline
- Analyze their performance
Matrix-Vector Form

\[ p(u) = \sum_{k=0}^{3} c_k u^k \]

Define \( c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \) \( u = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} \)

then \( p(u) = u^T c = c^T u \)

Interpolating Curve

Given four data (control) points \( p_0, p_1, p_2, p_3 \)
determine cubic \( p(u) \) which passes through them

Must find \( c_0, c_1, c_2, c_3 \)

Special case of the Lagrange polynomial interpolation

Interpolating Multiple Segments

Use \( p = [p_0, p_1, p_2, p_3]^T \)

Get continuity at join points but not continuity of derivatives

Parametric and Geometric Continuity

- We can require the derivatives of \( x, y, \) and \( z \) to each be continuous at join points (parametric continuity)
- Alternately, we can only require that the tangents of the resulting curve be continuous (geometry continuity)
- The latter gives more flexibility since we need to satisfy only two conditions rather than three at each join point

Parametric Continuity

- **Continuity** (recall from the calculus):
  - Two curves are \( C^i \) continuous at a point \( p \) iff the \( i \)-th derivatives of the curves are equal at \( p \)

Other Types of Curves and Surfaces

- How can we get around the limitations of the interpolating form
  - Lack of smoothness
  - Discontinuous derivatives at join points
- We have four conditions (for cubics) that we can apply to each segment
  - Use them other than for interpolation
  - Need only come close to the data
Hermite Form

Use two interpolating conditions and two derivative conditions per segment.
Ensures continuity and first derivative continuity between segments.

Example

- Here the p and q have the same tangents at the ends of the segment but different derivatives.
- Generate different Hermite curves.
- This technique is used in drawing applications.

Objectives

- Introduce Bezier curves.
- Derive the required matrices.
- Introduce the B-spline and compare it to the standard cubic Bezier.

Bezier’s Idea

- In graphics and CAD, we do not usually have derivative data.
- Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives in the Hermite form.

Computing Derivatives

\[ p'(0) = \frac{p_1 - p_0}{1/3} \]
\[ p'(1) = \frac{p_2 - p_1}{1/3} \]

Slope: \[ p'(0) \] located at \( u=1/3 \)
Slope: \[ p'(1) \] located at \( u=2/3 \)
Equations

Interpolating conditions are the same

\[ p(0) = p_0 = c_0 \]
\[ p(1) = p_3 = c_0 + c_1 + c_2 + c_3 \]

Approximating derivative conditions

\[ p'(0) = 3(p_1 - p_0) = c_1 \]
\[ p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3 \]

Solve four linear equations for \( c = M_B p \)

Beziers Matrix

\[
\mathbf{M}_B = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-3 & 3 & 0 & 0 \\
3 & -6 & 3 & 0 \\
-1 & 3 & -3 & 1 \\
\end{bmatrix}
\]

\[ p(u) = u^T \mathbf{M}_B p = \mathbf{b}(u)^T p \]

Blending Functions

\[
\mathbf{b}(u) = \begin{bmatrix}
(1-u)^3 \\
3u(1-u)^2 \\
3u^2(1-u) \\
u^3
\end{bmatrix}
\]

Note that all zeros are at 0 and 1 which forces the functions to be smooth over (0,1)

Cubic Bezier Curve

\[ p(u) = (1-u)^3p_0 + 3u(1-u)^2p_1 + 3u^2(1-u)p_2 + u^3p_3 \]

Bernstein Polynomials

- The blending functions are a special case of the Bernstein polynomials
  \[ b_{kd}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k} \]
- These polynomials give the blending polynomials for any degree Bezier form
  - All zeros at 0 and 1
  - For any degree they all sum to 1
  - They are all between 0 and 1 inside (0,1)

General Form of Bezier Curve

\[ \vec{p}(u) = \sum_{i=0}^{k} \vec{p}_{i+1} \binom{k}{i} (1-u)^{k-i} u^i \]
Convex Hull Property

- The properties of the Bernstein polynomials ensure that all Bezier curves lie within the convex hull of their control points.
- Hence, even though we do not interpolate all the data, we cannot be too far away.

Analysis

- Although the Bezier form is much better than the interpolating form, its derivatives are not continuous at join points.
- Can we do better?
  - Go to higher order Bezier
    - More work
      - Derivative continuity still only approximate
        - Supported by OpenGL
      - Apply different conditions
        - Tricky without letting order increase

B-Splines

- Basis splines: use the data at \( p_i = [p_{i-2}, p_{i-1}, p_i, p_{i+1}]^T \) to define curve only between \( p_{i-1} \) and \( p_{i+1} \).
- Allows us to apply more continuity conditions to each segment.
- For cubics, we can have continuity of function, first and second derivatives at join points.
- Cost is 3 times as much work for curves (Add one new point each time rather than three)
- For surfaces, we do 9 times as much work.

Splines and Basis

- If we examine the cubic B-spline from the perspective of each control (data) point, each interior point contributes (through the blending functions) to four segments.
- We can rewrite \( p(u) \) in terms of the data points as
  \[
  p(u) = \sum_{i=0}^{m-1} B_i(u) p_i
  \]
  defining the basis functions \( \{B_i(u)\} \)

B-splines: Cox-deBoor Recursion

- Cox-deBoor Algorithm: defines the blending functions for spline curves (not limited to deg 3)
  - curves are weighted avgs of lower degree curves
- Let \( B_{i,k}(t) \) denote the \( i \)-th blending function for a B-spline of degree \( d \), then:

  \[
  B_{i,k}(t) = \begin{cases} 
  1, & \text{if } t_{i+k} \leq t < t_{i+k+1} \\
  0, & \text{otherwise} \\
  \end{cases} 
  \]

  \[
  B_{i,k}(t) = \frac{t - t_{i+k}}{t_{i+k+1} - t_{i+k}} B_{i+k,i+1}(t) + \frac{t_{i+k+1} - t}{t_{i+k+1} - t_{i+k}} B_{i+k+1,i+1}(t) 
  \]

B-spline Blending Functions

- \( B_{i,0}(t) \): a step function that is 1 in the interval
- \( B_{i,0}(t) \): spans two intervals and is a piecewise linear function that goes from 0 to 1 (and back)
- \( B_{i,3}(t) \): spans three intervals and is a piecewise quadratic that grows from 0 to 1/4, then up to 3/4 in the middle of the second interval, back to 1/4, and back to 0
- \( B_{i,5}(t) \): a cubic that spans four intervals growing from 0 to 1/6 to 2/3, then back to 1/6 and to 0
**B-spline Blending Functions for 2nd Degree Splines**

- Note: can’t define a polynomial with these properties (both 0 and non-zero for ranges)
- Idea: subdivide the parameter space into intervals and build a piecewise polynomial
  - Each interval gets different polynomial function

**Generalizing Splines**

- We can extend to splines of any degree
- Data and conditions do not have to be given at equally spaced values (the knots)
  - Nonuniform and uniform splines
  - Can have repeated knots
  - Can force spline to interpolate points
- Cox-deBoor recursion gives method of evaluation

**NURBS**

- Nonuniform Rational B-Spline curves and surfaces add a fourth variable w to x,y,z
  - Can interpret as weight to give more importance to some control data
  - Can also interpret as moving to homogeneous coordinate
- Requires a perspective division
  - NURBS act correctly for perspective viewing
  - Quadrics are a special case of NURBS

**Objectives**

- Introduce methods to draw curves
  - Approximate with lines
  - Subdivision
- Derive the recursive method for evaluation of Bezier curves
- Learn how to convert all polynomial data to data for Bezier polynomials

**Evaluating Polynomials**

- Simplest method to render a polynomial curve is to evaluate the polynomial at many points and form an approximating polyline
- For surfaces we can form an approximating mesh of triangles or quadrilaterals
- Use Horner’s method to evaluate polynomials
  \[ p(u)=c_0+u(c_1+uc_2+u(c_3+uc_4)) \]
  - 3 multiplications/evaluation for cubic
Basic case, with two points:

- Plotting a curve via repeated linear interpolation
  - Given \( \{p_0, p_1, \ldots \} \)
  - a sequence of control points
  - Simple case: Mapping a parameter \( u \) to the line \( p_0, p_1 \)

\[
p(u) = (1 - u)p_0 + up_1 \quad \text{for } 0 \leq u \leq 1
\]

The de Casteljau Algorithm

• The complete solution from the algorithm for three iterations:

\[
\begin{align*}
p_0(u) &= (1 - u)p_0 + up_1 \\
p_1(u) &= (1 - u)p_1 + up_2 \\
p(u) &= (1 - u)p_0(u) + up_1(u)
\end{align*}
\]

The de Casteljau Algorithm

• Input: \( p_0, p_1, p_2, \ldots, p_n \in \mathbb{R}^3, t \in \mathbb{R} \)
• Iteratively set:

\[
p_{i,r}(t) = (1 - t)p_{i,r-1}(t) + tp_{i+1,r-1}(t)
\]

Then \( p_{0,n}(t) \) is the point with parameter value \( t \) on the Bézier curve defined by the \( p_i \)’s

De Casteljau: Arc Segment Animation

De Casteljau: Cubic Curve Animation
Cubic Bezier Curve

- Multiplying it all out gives

\[ p(u) = (1-u)^3p_0 + 3u(1-u)^2p_1 + 3u^2(1-u)p_2 + u^3p_3 \]

Issues with 3D “mesh” formats

- Easy to acquire
- Easy to render
- Harder to model with
- Error prone
  - split faces, holes, gaps, etc

BRep Data Structure

- Vertex structure
  - X,Y,Z point
  - Pointers to n coincident edges
- Face structure
  - Pointers to m edges

BRep Data Structures

- Winged-Edge Data Structure (Weiler)
- Vertex
  - n edges
- Edge
  - 2 vertices
  - 2 faces
- Face
  - m edges

Biparametric Surfaces

- Biparametric surfaces
  - A generalization of parametric curves
  - 2 parameters: s, t (or u, v)
  - Two parametric functions
Parametric Surfaces

- Surfaces require 2 parameters
  \[ x = x(u, v) \]
  \[ y = y(u, v) \]
  \[ z = z(u, v) \]
  \[ p(u, v) = [x(u, v), y(u, v), z(u, v)]^T \]

- Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation

Parametric Planes

- Point-vector form
  \[ p(u, v) = p_0 + uq + vr \]

- Three-point form
  \[ q = p_1 - p_0 \]
  \[ r = p_2 - p_0 \]

Parametric Sphere

- \[ x(u, v) = r \cos \theta \sin \phi \]
- \[ y(u, v) = r \sin \theta \sin \phi \]
- \[ z(u, v) = r \cos \phi \]

- \[ 360 \geq \theta \geq 0 \]
- \[ 180 \geq \phi \geq 0 \]

- \( \theta \) constant: circles of constant longitude
- \( \phi \) constant: circles of constant latitude

- Differentiate to show \[ n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} \]

Normals

- We can differentiate with respect to \( u \) and \( v \) to obtain the normal at any point \( p \)

\[ \frac{\partial p(u, v)}{\partial u} = \begin{bmatrix} \frac{\partial p_1(u, v)}{\partial u} \\ \frac{\partial p_2(u, v)}{\partial u} \\ \frac{\partial p_3(u, v)}{\partial u} \end{bmatrix} \]

\[ \frac{\partial p(u, v)}{\partial v} = \begin{bmatrix} \frac{\partial p_1(u, v)}{\partial v} \\ \frac{\partial p_2(u, v)}{\partial v} \\ \frac{\partial p_3(u, v)}{\partial v} \end{bmatrix} \]

Bicubic Surfaces

- Recall the 2D curve: \[ Q(s) = G \cdot M \cdot S \]
  - \( G \): Geometry Matrix
  - \( M \): Basis Matrix
  - \( S \): Polynomial Terms \[ [s^3 \ s^2 \ s^1 \ s^0] \]

- For 3D, we allow the points in \( G \) to vary in 3D along \( t \) as well:

\[ Q(s, t) = \begin{bmatrix} G_1(t) & G_2(t) & G_3(t) & G_4(t) \end{bmatrix} \cdot M \cdot S \]
Observations About Bicubic Surfaces

- For a fixed $t_1$, $Q(s, t_1)$ is a curve.
- Gradually incrementing $t_1$ to $t_2$, we get a new curve.
- The combination of these curves is a surface.
- $G_i(t)$ are 3D curves.

Bicubic Surfaces

- Each $G_i(t)$ is $G_i(t) = G_i \cdot M \cdot T$, where
  $$G_i = \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}$$
- Transposing $G_i(t)$, we get
  $$G_i(t) = T^T \cdot M^T \cdot G_i^T = T^T \cdot M^T \cdot \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}^T$$

Bicubic Surfaces

- Substituting $G_i(t)$ into $Q(s, t)$, we get $Q(s, t)$.
- The $g_{i1}$, etc. are the control points for the Bicubic surface patch:
  $$Q(s, t) = T^T \cdot M^T \cdot \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ g_{12} & g_{22} & g_{32} & g_{42} \\ g_{13} & g_{23} & g_{33} & g_{43} \\ g_{14} & g_{24} & g_{34} & g_{44} \end{bmatrix} \cdot M \cdot S$$

Bézier Patches

- Bézier Surfaces (similar definition)
  $$x(s, t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S$$
  $$y(s, t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S$$
  $$z(s, t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S$$

Blending Functions

- Blending Functions
  $$p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) p_{ij}$$
  Each $b_i(u)b_j(v)$ is a blending function.
  Shows that we can build and analyze surfaces from our knowledge of curves.
  A point on the patch is a weighted sum of the control points.
Beziers Patches

Using same data array \( P = [p_{ij}] \) as with interpolating form

\[
\tilde{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)\tilde{p}_{ij} = u^3 M_1 P M_0^T v
\]

Patch lies in convex hull

---

Bezier Blending Functions

\[
b(u) = \begin{bmatrix}
(1 - u)^3 \\
3u(1 - u)^2 \\
3u^2(1 - u) \\
u^3
\end{bmatrix}
\]

Note that all zeros are at 0 and 1 which forces the functions to be smooth over \((0,1)\)

---

Bézier Patches

- Expanding the summation

\[
\tilde{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)\tilde{p}_{ij} = b_0(u)b_0(v)\tilde{p}_{00} + b_0(u)b_1(v)\tilde{p}_{01} + b_0(u)b_2(v)\tilde{p}_{02} + b_0(u)b_3(v)\tilde{p}_{03} + b_0(u)b_0(v)\tilde{p}_{10} + \text{etc.}
\]

---

Features of Bicubic Bezier Patch

- Interpolates 4 corner control points
- 4 edges are Bezier curves
- Lies within convex hull of control points
- Normal at 4 corners from nearby CPs

---

Beziers Surfaces

- \( C^0 \) and \( G^0 \) continuity can be achieved between two patches by setting the 4 boundary control points to be equal
- \( G^1 \) continuity achieved when cross-wise CPs are co-linear
Bézier Surfaces: Example
Utah Teapot

• Utah Teapot modeled with 306 3D control points that define 32 Bézier patches with $G^1$ continuity.

Faceting

• Double loop that increments through the $u$ and $v$ parameters
  - Values between 0 and 1
• For each $(u,v)$ pair calculate 3D point on patch. Keep track of linear index.
• This produces a 2-D array of 3D points on the patch and their indices to the linear array.
• Define triangles that tessellate the patch.

Defining the Triangles

```c
// This assumes that the vertices are in a 2D array, verts(i,j)
// num_u & num_v are the number of points in u and v directions
for i = 0 to (num_u − 2)
  for j = 0 to (num_v -2)
    triangle0 = (verts[i,j], verts[i+1,j], verts[i+1,j+1])
    triangle1 = (verts[i,j], verts[i+1,j+1], verts[i,j+1])
```

Normals

• For rendering we need the normals if we want to shade
  - Can compute from parametric equations
    $$
n = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}
$$
  - Can approximate by averaging triangle normals.
Utah Teapot

- Most famous data set in computer graphics
- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches

Bezier Surface: Example

- Increased facet resolution
- Rendered

Drawing Parametric Surfaces

- Usually done “patch by patch”
- Two choices
  - Draw/render directly from the parametric description
  - Approximate the surface with a polygon mesh, then draw/render the mesh

Patch to Polygon Conversion

Two methods:
- **Object Space Conversion**
  - Techniques
    - Iterative evaluation
    - Uniform subdivision
    - Non-uniform subdivision
  - Resolution: depends on object space
- **Image Space Conversion**
  - Resolution: depends on pixels and screen

Object Space Conversion: Uniform Subdivision

Basic Procedure
- Cut parameter space into equal parts
- Find new points on the surface
- Recurse/Repeat “until done”
- Split squares into triangles
- Render triangles

Object Space Conversion: Non-Uniform Subdivision

- Basic idea
  - More facets in areas of high curvature
  - Use change in normals to surface to assess curvature
    - More derivatives
  - Break patch into sub-patches based on curvature changes
**Image Space Conversion**

- Idea: control subdivision based on screen criteria
  - Minimum pixel area
    - Stop when patch is basically one pixel
  - Screen flatness
    - Stop when patch converges to a polygon
  - Screen flatness of silhouette edges
    - Stop when edge is straight or size of pixel

**How do I know if I’ve found a silhouette edge?**

- If the viewing ray is tangent to the surface at the point it hits the surface!
  \[ N(X) \cdot L = 0 \]
  - Where \( N \) is the normal at the point where \( L \), the line of sight, hits the surface

**Silhouette Determination**

\[ N \cdot L = 0 \]

Brenner & Hughes, Brown U.  
Kowalski, et al.

**Suggestions for HW7**

- Write a function that takes control points and a \((u,v)\) pair and returns a 3D point on patch
- Use formula or de Casteljau Algorithm to compute point
- Compute an array of 3D points that lie on the patch with a double loop that increments through \( u \) and \( v \), from 0 to 1
  - Iterate over integers!
  - This would be an \( n \times m \) array, where \( n \) is the number of points in the \( u \) direction and \( m \) is the number in the \( v \) direction

**Suggestions for HW7**

- Use a double loop to iterate through \( i \) & \( j = 0 \to n-2 \) & \( 0 \to m-2 \)
- For each \((i, j)\) pair you define two triangles. The first has vertices \([(i, j), [i+1, j], [i, j+1]]\). The 2nd triangle is defined with vertices \([(i+1, j), [i+1, j+1], [i, j+1]]\).
- Now you have a mesh defined like an SMF model. Modify your HW5 code to render it.

**Suggestions for HW7**

- Implement the interface that allows the user to change \( n \) and \( m \) (the resolution of the mesh)
- When these values are changed by the user, you'll need to regenerate the mesh
- Flip normals that face away from the eye point, so both sides of the mesh are shaded
  - Or take \( \text{abs}(L \cdot N) \)