What You’ll See in This Chapter

This chapter is about geometric primitives in general and in specific. It is divided into seven sections.

• Section 9.1 is on representing geometric primitives.
• Section 9.2 is on lines and rays.
• Section 9.3 is on spheres and circles.
• Section 9.4 is on bounding boxes.
• Section 9.5 is on planes.
• Section 9.6 is on triangles.
• Section 9.7 is on polygons.

Chapter 9: Geometric Primitives

Fletcher Dunn
Valve Software

Ian Parberry
University of North Texas

What’s a Geometric Primitive

A geometric primitive is a simple geometric object, among which we will include the:

• Ray
• Line
• Plane
• Sphere
• Polygon

Section 9.1: Representation Techniques

Representation Forms

There are three basic ways to represent a geometric primitive

• Implicit form
• Parametric form
• “Straightforward” form

Different forms for different uses.
Implicit Form

- In *implicit form* we supply a Boolean function \( f(x,y,z) \) that is true for all points of the primitive and false for all other points.
- For example, the equation for the surface of a unit sphere centered at the origin:
  \[ x^2+y^2+z^2 = 1 \]
  is true for all points on the surface of a unit sphere centered at the origin.
- Implicit form is useful for things like point inclusion tests.

Parametric Form

- Express \( x, y, z \) as functions of another parameter or parameters) that vary between certain limits.
- For example:
  \[
  x(t) = \cos 2\pi t \\
  y(t) = \sin 2\pi t
  \]
  As \( t \) varies through \( 0 \leq t \leq 1 \), the point \((x(t), y(t))\) describes a unit circle.

Fluid Shapes

- Certain methods for representing fluid and organic shapes work using implicit representation.
- A density function is defined for a given point in space by considering the proximity of nearby fluid points. A point \( p \) is considered to be inside the fluid if the density at \( p \) exceeds some nonzero threshold.
- The *marching cubes algorithm* is a classic technique for converting an arbitrary implicit form into a surface description (such as a polygon mesh).

Univariate and Bivariate

- It is convenient to use a *normalized* parameter in range \( 0 \leq t \leq 1 \), although we may allow \( t \) to assume any range of values we wish.
- Another common choice is \( 0 \leq t \leq n \), where \( n \) is the “length” of the primitive.
- When our functions are in terms of one parameter, we say that the functions are *univariate*. Univariate functions trace out a 1D shape: a curve.
- We also use more than one parameter. A *bivariate* function accepts two parameters, such as \( s \) and \( t \). Bivariate functions trace out a surface rather than a line.

“Straightforward” Form

- Nonmathematical; no equations.
- Easy for humans to understand.
- Example: “A unit sphere centered at the origin.”

Degrees of Freedom

- *Degrees of freedom*: the minimum number of “pieces of information” which are required to describe the entity unambiguously.
- For each geometric primitive, some representation forms use more numbers than others.
- Usually due to a redundancy that could be eliminated by assuming the appropriate constraint, such as a vector having unit length.
Section 9.2: Lines and Rays

Classical definitions:
- A line extends infinitely in two directions.
- A line segment is a finite portion of a line that has two endpoints.
- A ray is half of a line that has an origin and extends infinitely in one direction.

Computer graphics definition:
- A ray is a directed line segment.

The Importance of Being Ray
- A ray will have an origin and an endpoint.
- A ray defines a position, a finite length, and (unless it has zero length) a direction.
- A ray also defines a line and a line segment (containing it).
- Rays are important in computational geometry and computer graphics.

Two Points Representation of Rays
Give the two points that are the ray origin and the ray endpoint: \( \mathbf{p}_{\text{org}} \) and \( \mathbf{p}_{\text{end}} \).

Parametric Representation of Rays
Three equations in \( t \):
\[
\begin{align*}
x(t) &= x_0 + t \Delta x \\
y(t) &= y_0 + t \Delta y \\
z(t) &= z_0 + t \Delta z
\end{align*}
\]
The parameter \( t \) is restricted to \( 0 \leq t \leq 1 \).

Vector Notation
Alternatively, use vector notation:
\[
\mathbf{p}(t) = \mathbf{p}_0 + t \mathbf{d}
\]
Where:
\[
\begin{align*}
\mathbf{p}(t) &= [x(t) \ y(t) \ z(t)] \\
\mathbf{p}_0 &= [x_0 \ y_0 \ z_0] \\
\mathbf{d} &= [\Delta x \ \Delta y \ \Delta z]
\end{align*}
\]
Vector Notation

\( \mathbf{p}(0) = \mathbf{p}_0 \) is the origin point. 
\( \mathbf{p}(1) = \mathbf{p}_0 + \mathbf{d} \) is the end point. 
\( \mathbf{d} \) is the ray’s length and direction.

\[
\mathbf{p}_0 \\
\mathbf{d} \\
t = 1 \\
t = 0
\]

Variant

- Let \( \mathbf{d} \) be a unit vector. 
- Vary \( t \) in the range \([0, \ell]\), where \( \ell \) is the length of the ray. 
- \( \mathbf{p}(0) = \mathbf{p}_0 \) is the origin point. 
- \( \mathbf{p}(\ell) = \mathbf{p}_0 + \ell \mathbf{d} \) is the end point. 
- \( \mathbf{d} \) is the ray’s direction.

Lines in 2D

Implicit representation of a line:

\[ ax + by = d \]

Some people prefer the longer:

\[ ax + by + d = 0 \]

Vector notation: let \( \mathbf{n} = [a \ b] \), \( \mathbf{p} = [x \ y] \) and use dot product:

\[ \mathbf{p} \cdot \mathbf{n} = d \]

Special Case

- Make \( \mathbf{n} \) a unit vector (scale \( d \) appropriately.) 
- Then \( d \) is the signed distance from the origin to the line measured perpendicular to the line (parallel to \( \mathbf{n} \)). 
- By signed distance, we mean that \( d \) is positive if the line is on the side of the origin that the normal points towards. 
- As \( d \) increases, the line moves in the direction of \( \mathbf{n} \).

Variation

- Give a point \( \mathbf{q} \) that is on the line, rather than the distance to the origin. 
- Any point will do. 
- The direction of the line is described using a normal to the line \( \mathbf{n} \), as before.
Slope-Intercept Form

\[ y = mx + b \]

\( m \) is the slope of the line, expressed as a ratio of \textit{rise} over \textit{run}:

- For every \textit{rise} unit that we move up,
- we move \textit{run} units to the right.

\( b \) is the \textit{y-intercept}

- Substituting \( x = 0 \) shows that the line crosses the \textit{y}-axis at \( y = b \).

Caveats

- The slope of a horizontal line is zero.
- A vertical line has infinite slope and cannot be represented in slope-intercept form since the implicit form of a vertical line is \( x = k \).

Another Form

- Another way to specify a line is to give a normal vector \( \mathbf{n} \) and the perpendicular distance \( d \) from the line to the origin.
- The normal describes the direction of the line, and the distance describes its location.

Yet Another Form

- One final way to define a line is as the perpendicular bisector of two points \( q \) and \( r \).
- In fact, this is one of the earliest definitions of a line: the set of all points equidistant from two given points.
Some Line Conversions 1

To convert a ray defined using two points to parametric form:

\[ p_0 = p_{\text{org}} \]
\[ d = p_{\text{end}} - p_{\text{org}} \]

The opposite conversion, from parametric form to two-points form:

\[ p_{\text{org}} = p_0 \]
\[ p_{\text{end}} = p_0 + d \]

Some Line Conversions 2

Given a parametric ray, we can compute the implicit line that contains this ray:

\[ a = d_y \]
\[ b = -d_x \]
\[ d = p_{\text{org}}d_y - p_{\text{org}}d_x \]

Some Line Conversions 3

To convert a line expressed implicitly to slope-intercept form: (Note that since \( b \) is the traditional variable to use in both the slope-intercept and implicit forms, so we added color to distinguish between the two \( b \)'s.) A red \( b \) means the \( b \) in the implicit form, which is different from the black \( b \) from the slope-intercept form.

\[ m = -a/b \]
\[ b = d/b \]

Some Line Conversions 4

Converting a line expressed implicitly to “normal and distance” form:

\[ n = \left[ \begin{array}{c} a \\ b \end{array} \right] / \sqrt{a^2 + b^2} \]

\[ \text{distance} = d / \sqrt{a^2 + b^2} \]

Some Line Conversions 5

Converting a normal and a point on line to normal and distance form (assuming \( n \) is a unit vector):

\[ n = n \]
\[ \text{distance} = n \cdot q \]

Some Line Conversions 6

Perpendicular bisector to implicit form:

\[ a = q_y - r_y \]
\[ b = r_x - q_x \]
\[ d = \frac{q + r}{2} \cdot \left[ \begin{array}{c} a \\ b \end{array} \right] \]
\[ = \frac{q + r}{2} \cdot \left[ \begin{array}{c} q_y - r_y \\ r_x - q_x \end{array} \right] \]
\[ = \frac{(q_x + r_x)(q_y - r_y) + (q_y + r_y)(r_x - q_x)}{2} \]
\[ = \frac{(q_xq_y - q_xr_y + r_xq_y - r_xr_y) + (q_yr_x - q_yq_x + r_yr_x - r_yq_x)}{2} \]
\[ = r_xq_y - q_xr_y \]
Section 9.3: Spheres and Circles

Chapter 9 Notes 3D Math Primer for Graphics & Game Dev

Circles and Spheres

- A sphere is the set of all points that are a given distance from a given point.
- The distance from the center of the sphere to a point is known as the radius of the sphere.
- The straightforward representation of a sphere is its center $c$ and radius $r$. (see next slide)
- A circle is a 2D sphere, of course. Or a sphere is a 3D circle, depending on your perspective.

Spheres in Collision Detection

- A bounding sphere is often used in collision detection for fast rejection because the equations for intersection with a sphere are simple.
- Rotating a sphere does not change its shape, and so a bounding sphere can be used for an object regardless of the orientation of the object.

Implicit Representation

The implicit form of a sphere with center $c$ and radius $r$ is the set of points $p$ such that:
$$|| p - c || = r.$$  
For collision detection, $p$ is inside the sphere if:
$$|| p - c || \leq r.$$  
Expanding this, if $p = [x\ y\ z]$
$$(x - c_x)^2 + (y - c_y)^2 = r^2 \quad (2D\ circle)$$
$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2 \quad (3D\ sphere)$$

Some Measurements

For both circles and spheres, we compute the diameter $D$ (distance from one point to a point on the exact opposite side), and circumference $C$ (distance all the way around the circle) from the radius $r$ as follows:
$$D = 2r$$
$$C = 2\pi r = \pi D$$
The area of a circle is:
$$A = \pi r^2$$
The surface area $S$ and volume $V$ of a sphere are:
$$S = 4\pi r^2$$
$$V = \frac{4}{3}\pi r^3$$
Section 9.4: Bounding Boxes

Types of Bounding Box

- Like spheres, bounding boxes are also used in collision detection.
- AABB: axially aligned bounding box.
  - sides aligned with world axes
- OABB: object aligned bounding box.
  - sides aligned with object axes
- Axially aligned bounding boxes are simpler to create and use.

Points In AABB

The points \((x, y, z)\) inside an AABB satisfy:

\[
\begin{align*}
  x_{\text{min}} &\leq x \leq x_{\text{max}} \\
  y_{\text{min}} &\leq y \leq y_{\text{max}} \\
  z_{\text{min}} &\leq z \leq z_{\text{max}}
\end{align*}
\]

Two special corner points are:

\[
\begin{align*}
  \mathbf{p}_{\text{min}} &= [x_{\text{min}}, y_{\text{min}}, z_{\text{min}}] \\
  \mathbf{p}_{\text{max}} &= [x_{\text{max}}, y_{\text{max}}, z_{\text{max}}]
\end{align*}
\]

The center point \(\mathbf{c}\) is given by:

\[
\mathbf{c} = (\mathbf{p}_{\text{min}} + \mathbf{p}_{\text{max}})/2.
\]

AABB Size & Radius

The size vector \(\mathbf{s}\) is the vector from \(\mathbf{p}_{\text{min}}\) to \(\mathbf{p}_{\text{max}}\) and contains the width, height, and length of the box:

\[
\mathbf{s} = \mathbf{p}_{\text{max}} - \mathbf{p}_{\text{min}}
\]

The radius vector \(\mathbf{r}\) is the vector from the center to \(\mathbf{p}_{\text{max}}\):

\[
\mathbf{r} = \mathbf{p}_{\text{max}} - \mathbf{c} = \mathbf{s}/2.
\]

Defining an AABB

- To unambiguously define an AABB requires only two of the five vectors \(\mathbf{p}_{\text{min}}, \mathbf{p}_{\text{max}}, \mathbf{c}, \mathbf{s}, \) and \(\mathbf{r}\).
- Other than the pair \(\mathbf{s}, \mathbf{r}\), any pair may be used.
- Some representation forms are more useful in particular situations than others.
- A good idea to represent a bounding box using \(\mathbf{p}_{\text{min}}, \mathbf{p}_{\text{max}}\), since in practice these are needed far more frequently that \(\mathbf{c}, \mathbf{s}, \) and \(\mathbf{r}\).
- Of course, computing any of these three from \(\mathbf{p}_{\text{min}}, \mathbf{p}_{\text{max}}\) is very fast.
AABB Declaration

class AABB3 {
  public:
    Vector3 min;
    Vector3 max;
};

Chapter 9 Notes 3D Math Primer for Graphics & Game Dev

An Empty AABB

We first reset the minimum and maximum values to infinity, or what is effectively bigger than any number we will encounter in practice.

```cpp
void AABB3::empty() {
  const float kBigNumber = 1e37f;
  min.x = min.y = min.z = kBigNumber;
  max.x = max.y = max.z = -kBigNumber;
}
```

Adding a Point to an AABB

Then we can add a single point into the AABB by expanding the AABB if necessary to contain it:

```cpp
void AABB3::add(const Vector3 &p) {
  if (p.x < min.x) min.x = p.x;
  if (p.x > max.x) max.x = p.x;
  if (p.y < min.y) min.y = p.y;
  if (p.y > max.y) max.y = p.y;
  if (p.z < min.z) min.z = p.z;
  if (p.z > max.z) max.z = p.z;
}
```

Chapter 9 Notes 3D Math Primer for Graphics & Game Dev

Adding a List of Points

```cpp
// Our list of points
const int n;
Vector3 list[n];

// First, empty the box
AABB3 box;
box.empty();

// Add each point into the box
for (int i = 0; i < n; ++i)
  box.add(list[i]);
```

Chapter 9 Notes 3D Math Primer for Graphics & Game Dev

AABBs vs Spheres

• Computing the optimal AABB for a set of points is easy and takes linear time.
• Computing the optimal bounding sphere is a much more difficult problem.
• For many objects that arise in practice, AABBs usually provide a “tighter” bounding volume, and thus better trivial rejection.

Chapter 9 Notes 3D Math Primer for Graphics & Game Dev

Which is Best?

• Of course, for some objects, the bounding sphere is better.
• In the worst case, AABB volume will be just under twice the sphere volume.
• However, when a sphere is bad, it can be really bad.
Transforming an AABB

- When you transform an object, its AABB changes.
- Can recompute a new AABB from the transformed object. This is slow.
- Faster to transform the AABB itself.
- But the transformed AABB may not be an AABB.
- So, transform the AABB, and compute a new AABB from the transformed box.
- There are some small but significant optimizations for computing the new AABB.

Downside to Transforming the AABB

Transforming an AABB may give you a larger AABB than recomputing the AABB from the object.

Some Code

The class Matrix4x3 is a 4 x 3 transform matrix, which can represent any affine transform (a 4 x 4 matrix where the rightmost column is assumed to be [0, 0, 0, 1]^T).

```cpp
void AABB3::getTransformedBox(const AABB3 &box, const Matrix4x3 &m) {
    // Start with the last row of the matrix, which is the translation portion, i.e., the location of the origin after transformation.
    min = max = getTranslation(m);
    // Examine each of the 3 matrix elements and compute the new AABB
    if (m.m11 > 0.0f) {
        min.x = m.m11 * box.min.x; max.x = m.m11 * box.max.x;
    } else {
        min.x = m.m11 * box.max.x; max.x = m.m11 * box.min.x;
    }
    if (m.m21 > 0.0f) {
        min.y = m.m21 * box.min.x; max.y = m.m21 * box.max.x;
    } else {
        min.y = m.m21 * box.max.x; max.y = m.m21 * box.min.x;
    }
    if (m.m31 > 0.0f) {
        min.z = m.m31 * box.min.x; max.z = m.m31 * box.max.x;
    } else {
        min.z = m.m31 * box.max.x; max.z = m.m31 * box.min.x;
    }
}
```

The Rest of the Code

The rest of the code does the same for m12 through m33 by changing the m11 in the previous slide:

```cpp
if (m.m12 > 0.0f) {
    min.x = m.m12 * box.min.x; max.x = m.m12 * box.max.x;
} else {
    min.x = m.m12 * box.max.x; max.x = m.m12 * box.min.x;
}
if (m.m22 > 0.0f) {
    min.y = m.m22 * box.min.x; max.y = m.m22 * box.max.x;
} else {
    min.y = m.m22 * box.max.x; max.y = m.m22 * box.min.x;
}
if (m.m32 > 0.0f) {
    min.z = m.m32 * box.min.x; max.z = m.m32 * box.max.x;
} else {
    min.z = m.m32 * box.max.x; max.z = m.m32 * box.min.x;
}
```

Planes

- A plane in 3D is the set of points equidistant from two points.
- A plane is perfectly flat, has no thickness, and extends infinitely.
- Planes are very common tools in video games.
- The implicit form of a plane is given by all points p = (x, y, z) that satisfy the plane equation:

  \[ ax + by + cz = d \]

  \[ \mathbf{p} \cdot \mathbf{n} = d \]

  where \( \mathbf{n} = [a, b, c] \).
Chapter 9 Notes 3D Math Primer for Graphics & Game Dev 62

Front and Back

- We often consider a plane to have a front and a back side.
- The front of the plane is the side that n points away from.
- This is a particularly important concept in computer graphics, because we can skip drawing the so-called back-facing triangles that face away from the camera. More in the next chapter.

Plane from 3 Points

- Another way we can define a plane is to give three noncollinear points that are in the plane.
- Collinear points (points in a straight line) won't do because there would be an infinite number of planes that contain that line, and there would be no way of telling which one we mean.
- Every set of three noncollinear points defines a unique plane passing through them.

n and d for a Plane Passing Thru 3 Pts

- Let's compute n and d for a plane that passes through three noncollinear points $p_1, p_2, p_3$.
- Which way will n point? The standard way to do this in a left-handed coordinate system is to assume that $p_2, p_3$ are listed in clockwise order when viewed from the front side of the plane.
- In a right-handed coordinate system, we usually assume the points are listed in counter-clockwise order so that the equations are the same.

Using Cross Product

- Construct two vectors $e_1$ and $e_2$ using the clockwise ordering of the points.
- The cross product of these two vectors yields the perpendicular vector n, but this vector is not necessarily of unit length. We can normalize it if we need to.
- The notation “e” stands for “edge” vector, since these equations commonly arise in computer graphics when computing the plane equation for a triangle.

$$e_1 = p_2 - p_1$$
$$e_2 = p_3 - p_2$$
$$n = e_1 \times e_2$$

Please forgive the arcane numbering system for $e_1$ and $e_2$. All will become clear if you look at the numbering in the above equations. (At least, as Slartibartfast said to Arthur Dent, closer than it is now.)

Best Fit Plane for More Than 3 Points

- Occasionally we may wish to compute the plane equation for a set of more than three points.
- The most common example of such a set of points is the vertices of a polygon. In this case, the vertices are assumed to be enumerated in a clockwise fashion around the polygon.
- One naive solution is to arbitrarily select three consecutive points and compute the plane equation from those three points. However,
  - The three points we chose may be collinear, or nearly collinear, which is almost as bad because it is numerically inaccurate
  - Or perhaps the polygon is concave and the three points we have chosen are a point of concavity and therefore form a counterclockwise turn (which would result in a normal that points the wrong direction).
  - Or the vertices of the polygon may not be coplanar, which can happen due to numeric imprecision or the method used to generate the polygons.
- What we really want is a way to compute the best fit plane.

Best Fit Plane Normal

Given $n$ points $p_1, p_2, \ldots, p_n$ where $p_i = [x_i, y_i, z_i]$ for $1 \leq i \leq n$, the best fit perpendicular vector is given by $n = [n_x, n_y, n_z]$, where (taking $p_{n+1} = p_1$):

$$n_x = \sum_{i=1}^{n} (z_i + z_{i+1})(y_i - y_{i+1})$$
$$n_y = \sum_{i=1}^{n} (x_i + x_{i+1})(z_i - z_{i+1})$$
$$n_z = \sum_{i=1}^{n} (y_i + y_{i+1})(x_i - x_{i+1})$$
Best Fit Plane Distance

The best-fit $d$ value is the average of the $d$ values for each point:

$$d = \frac{1}{n} \sum_{i=1}^{n} (p_i \cdot n) = \frac{1}{n} \left( \sum_{i=1}^{n} p_i \right) \cdot n$$

Distance from Point to Plane

- Imagine a plane and a point $q$ that is not in the plane.
- There exists a point $p$ that lies in the plane and is the closest point in the plane to $q$.
- Clearly, the vector from $p$ to $q$ is perpendicular to the plane, and thus is of the form $an$ for some scalar $a$.

Distance $a$ from Point to Plane

- If we assume the plane normal $n$ is a unit vector, then the distance from $p$ to $q$ (and thus the distance from $q$ to the plane) is simply $a$.
- This distance will be negative when $q$ is on the back side of the plane.
- Compute $a$ as follows:

$$p + an = q$$

$$(p + an) \cdot n = q \cdot n$$

$$p \cdot n + (an) \cdot n = q \cdot n$$

$$d + a = q \cdot n$$

$$a = q \cdot n - d$$

Triangles

- Triangles are fundamental to modeling and graphics.
- The surface of a complex 3D object, such as a car or a human body, is approximated with many triangles.
- Such a group of connected triangles forms a triangle mesh, which we will see in Chapter 10.
- But before we learn how to manipulate many triangles, we must first learn how to manipulate one triangle.
Basic Properties of a Triangle

• A triangle is defined by listing three vertices.
• The order that these points are listed is significant. In a left-handed coordinate system we list the points in clockwise order when viewed from the front side of the triangle.
• We will refer to the three vertices as \( v_1, v_2, v_3 \).
• A triangle lies in a plane, and the equation of this plane (the normal \( \mathbf{n} \) and distance to origin \( d \)) is important in a number of applications.

The Parts of Triangle

![Diagram of a triangle with edges and vertices labeled]

Edge Length

• Let \( l_i \) denote the length of \( e_i \) for \( i = 1, 2, 3 \).
• Notice that \( e_i \) is opposite \( v_i \), the vertex with the corresponding index.
• Then,

\[
\begin{align*}
  e_1 &= v_3 - v_2 \\
  e_2 &= v_1 - v_3 \\
  e_3 &= v_2 - v_1 \\
  l_1 &= ||e_1|| \\
  l_2 &= ||e_2|| \\
  l_3 &= ||e_3||
\end{align*}
\]

Laws of Sines and Cosines

Law of Sines

\[
\frac{\sin \theta_1}{l_1} = \frac{\sin \theta_2}{l_2} = \frac{\sin \theta_3}{l_3}
\]

Law of Cosines

\[
\begin{align*}
  l_1^2 &= l_2^2 + l_3^2 - 2l_2l_3 \cos \theta_1 \\
  l_2^2 &= l_1^2 + l_3^2 - 2l_1l_3 \cos \theta_2 \\
  l_3^2 &= l_1^2 + l_2^2 - 2l_1l_2 \cos \theta_3
\end{align*}
\]

Area of a Triangle

The area of a triangle of base \( b \) and perpendicular height (also called altitude) \( h \) is given by

\[
A = \frac{bh}{2}.
\]

We covered this in Chapter 2 (recall the diagram to the right?)

Heron’s Formula

• If the altitude is not known, then Heron’s formula can be used, which requires only the lengths of the three sides.
• Let \( s \) equal one half the perimeter (also known as the semiperimeter).
• Then the area \( A \) is given by:

\[
A = \sqrt{s(s-l_1)(s-l_2)(s-l_3)}.
\]
Area from Point Coordinates 1

- The area of a triangle can be computed directly from the Cartesian coordinates of the vertices.
- Let's tackle this problem in 2D.
- Compute, for each of the three edges of the triangle, the signed area of the trapezoid bounded above by the edge and below by the x-axis.
- By signed area, we mean that the area is positive if the edge points from left to right, and negative otherwise.
- There will always be at least one positive edge and at least one negative edge. A vertical edge has zero area.

Area from Point Coordinates 2

The formulas for the areas under each edge are:

\[ A(e_1) = \frac{(y_3 + y_2)(x_3 - x_2)}{2}, \]
\[ A(e_2) = \frac{(y_1 + y_3)(x_1 - x_3)}{2}, \]
\[ A(e_3) = \frac{(y_2 + y_1)(x_2 - x_1)}{2}. \]

Area from Point Coordinates 3

By summing the signed areas of the three trapezoids, we arrive at the area of the triangle.

\[ A = A(e_1) + A(e_2) + A(e_3) \]
\[ = \frac{(y_3 + y_2)(x_3 - x_2) + (y_1 + y_3)(x_1 - x_3) + (y_2 + y_1)(x_2 - x_1)}{2} \]
\[ = \frac{\left( y_3x_3 - y_2x_2 + y_2x_3 - y_3x_3 + y_1x_1 - y_3x_1 + y_3x_1 - y_2x_3 + y_2x_2 - y_1x_2 + y_1x_1 - y_2x_1 + y_2x_1 - y_3x_1 \right)}{2} \]
\[ = \frac{y_1(x_2 - x_3) + y_2(x_3 - x_1) + y_3(x_1 - x_2)}{2}. \]

Area from Point Coordinates 4

We can actually simplify this just a bit further. The idea is to realize that we can translate the triangle without affecting the area. So we will shift the triangle vertically by subtracting \( y_2 \) from each of the \( y \) coordinates.

\[ A = \frac{y_2(x_2 - x_3) + y_3(x_3 - x_1) + y_3(x_1 - x_2)}{2} \]
\[ = \frac{(y_1 - y_3)(x_2 - x_3) + (y_2 - y_1)(x_3 - x_1) + (y_3 - y_2)(x_1 - x_2)}{2} \]
\[ = \frac{(y_1 - y_3)(x_2 - x_3) + (y_2 - y_1)(x_3 - x_1)}{2}. \]

Area from Cross Product

- Recall from Chapter 2 that the magnitude of the cross product of two vectors \( \mathbf{a} \) and \( \mathbf{b} \) is equal to the area of the parallelogram formed on two sides by \( \mathbf{a} \) and \( \mathbf{b} \).
- Since the area of a triangle is half the area of the enclosing parallelogram, we have a simple way to calculate the area of the triangle. Given two edge vectors from the triangle, \( \mathbf{e}_1 \) and \( \mathbf{e}_2 \), the area of the triangle is given by:
  \[ A = ||\mathbf{e}_1 \times \mathbf{e}_2||/2. \]
- This is in fact the same as the method on the previous slide.

Barycentric Space

- Even though we certainly use triangles in 3D, the surface of a triangle lies in a plane and is inherently a 2D object.
- Moving around on the surface of a triangle that is arbitrarily oriented in 3D is somewhat awkward.
- It would be nice to have a coordinate space that is related to the surface of the triangle.
- Barycentric space is just such a coordinate space.
- Many practical problems that arise when making video games, such as interpolation and intersection, can be solved using barycentric coordinates.
Barycentric Coordinates

- Any point in the plane of the triangle can be expressed as weighted average of the vertices. 
- These weights are known as barycentric coordinates. 
- Barycentric coordinates \((b_1, b_2, b_3)\) correspond to the point \(b_1\mathbf{v}_1 + b_2\mathbf{v}_2 + b_3\mathbf{v}_3\). 
- Of course this is simply a linear combination of some vectors. 
- We've already talked about how Cartesian coordinates can also be interpreted as a linear combination of the basis vectors, but the subtle distinction between barycentric coordinates and ordinary Cartesian coordinates is that in the former the sum of the coordinates is restricted to be unity. 

Observations

- The triangle vertices have a trivial form in barycentric space: \((1, 0, 0)\) corresponds to \(\mathbf{v}_1\), \((0, 1, 0)\) to \(\mathbf{v}_2\), and \((0, 0, 1)\) to \(\mathbf{v}_3\). 
- All points on the side opposite a vertex will have a zero for the barycentric coordinate corresponding to that vertex. For example, \(b_i = 0\) for all points on the line containing \(\mathbf{v}_i\) (which is opposite \(\mathbf{v}_i\)). 
- Any point in the plane can be described using barycentric coordinates, not just the points inside the triangle. 
  - The barycentric coordinates of a point inside the triangle are between 0 and 1. 
  - The barycentric coordinates of a point outside the triangle will have at least one negative value. 

Redundancy of Representation

- The nature of barycentric space is not exactly the same as that of Cartesian space. 
- This is born out by the fact that barycentric space is 2D, but there are three coordinates. 
- Since the sum of the coordinates is one, barycentric space really only has two degrees of freedom; there is one degree of redundancy. 
- In other words, we could completely describe a point in barycentric space using only two of the coordinates and compute the third from the other two. 

Applications of Barycentric Coordinates 1

- In graphics, it is common for parameters to be edited (or computed) per vertex, such as texture coordinates, surface normals, colors, lighting values, etc. 
- We often then need to determine the interpolated value of one of those parameter at an arbitrary location within the triangle. 
- This is easily done using barycentric coordinates. We first determine the barycentric coordinates of the interior point in question, and then take the weighted average of the color, texture coordinates, etc. in the vertices using the barycentric weights.
Applications of Barycentric Coordinates 2

• Another important example is intersection testing. One simple way to perform ray-triangle testing is to determine the point where the ray intersects the infinite plane containing the triangle, and then to decide if this point lies within the triangle.
• An easy way to make this decision is to calculate the barycentric coordinates of the point, using the techniques described below.
• The point is inside the triangle if all of the barycentric coordinates lie in the 0 to 1 range.

Cartesian Coords from Barycentric

• Easy, we’ve done this already. Recall that the barycentric coordinates \((b_1, b_2, b_3)\) correspond to the point \(b_1\mathbf{v}_1 + b_2\mathbf{v}_2 + b_3\mathbf{v}_3\).
• Now let’s see how to determine barycentric coordinates from Cartesian coordinates.
• Start with the 2D case because it’s easier.

Simultaneous Equations

This gives us 3 equations in 3 unknowns:

\[
\begin{align*}
px &= b_1x_1 + b_2x_2 + b_3x_3 \\
y &= b_1y_1 + b_2y_2 + b_3y_3 \\
b_1 + b_2 + b_3 &= 1
\end{align*}
\]

Solving the system of equations yields:

\[
\begin{align*}
b_1 &= \frac{(y - y_3)(x_2 - x_3) + (y_2 - y_3)(x_3 - x_1)}{(x_2 - x_3)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)} \\
b_2 &= \frac{(y - y_1)(x_3 - x_2) + (y_1 - y_3)(x_2 - x_1)}{(x_2 - x_3)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)} \\
b_3 &= \frac{(y - y_2)(x_1 - x_2) + (y_2 - y_1)(x_1 - x_3)}{(x_2 - x_3)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)}
\end{align*}
\]

Area Relationships

• Notice that the denominator is the same in each of \(b_1\), \(b_2\), and \(b_3\).
• It is equal to the twice the area of the triangle.
• What’s more, for each barycentric coordinate \(b_j\), the numerator is equal to twice the area of the sub-triangle \(T_j\).
2D Barycentric Coordinates

• In other words, \( b_i = A(T_i)/A(T) \) for \( i = 1, 2, 3 \).
• Note that this interpretation applies even if \( p \) is outside the triangle, since our equation for computing area yields a negative result if the vertices are enumerated in a counterclockwise order.
• If the three vertices of the triangle are collinear, then the area in the denominator will be zero, and thus the barycentric coordinates cannot be computed.

3D Barycentric Coords

• Computing barycentric coordinates for an arbitrary point \( p \) in 3D is more complicated than in 2D.
• We cannot solve a system of equations as we did before, since we have three unknowns and four equations (one equation for each coordinate of \( p \), plus the normalization constraint on the barycentric coordinates).
• Another complication is that \( p \) may not lie in the plane that contains the triangle, in which case the barycentric coordinates are undefined.
• For now, let’s assume that \( p \) lies in the plane containing the triangle.

Project 3D Onto 2D

• One trick that works is to turn the 3D problem into a 2D one, simply by discarding one of \( x, y, \) or \( z \).
• This has the effect of projecting the triangle onto one of the three cardinal planes.
• Intuitively, this works because the projected areas are proportional to the original areas.
• But which coordinate should we discard? We can’t just always discard the same one, since the projected points will be collinear if the triangle is perpendicular to the projection plane. If our triangle is nearly perpendicular to the plane of projection, we will have problems with floating point accuracy.

Maximize the Area

• A solution to this dilemma is chose the plane of projection so as to maximize the area of the projected triangle.
• This can be done by examining the plane normal; whichever coordinate has the largest absolute value is the coordinate that we will discard.
• Eg, if the normal is \([0.267, -0.802, 0.535]\) then we would discard the \( y \) values of the vertices and \( p \), projecting onto the \( xz \) plane.
• Here’s some code...

Computing Barycentric Coords

```cpp
bool computeBarycentricCoords3d(
    const Vector3 v[3], // vertices of the triangle
    const Vector3 &p, // point that we wish to compute coords for
    float b[3]) // barycentric coords returned here
{
    // First, compute two clockwise edge vectors
    Vector3 d1 = v[2] - v[1];
    Vector3 d2 = v[2] - v[0];
    // Compute surface normal using cross product. In many cases
    // this step could be skipped, since we would have the surface
    // normal precomputed. We do not need to normalize it, although
    // if a precomputed normal was normalized, it would be ok.
    Vector3 n = crossProduct(d1, d2);
    // Locate dominant axis of normal, and select plane of projection
    float u1 = v[0].y - v[2].y;
    float u2 = v[1].y - v[2].y;
    float u3 = p.y - v[0].y;
    float u4 = p.y - v[2].y;
    float u5 = v[0].z - v[2].z;
    float u6 = v[1].z - v[2].z;
    float u7 = p.z - v[0].z;
    float u8 = p.z - v[2].z;
    if (fabs(u5) >= fabs(u3) && fabs(u5) >= fabs(u7)) {
        // Discard x, project onto yz plane
        v1 = v[0].y - v[2].y;
        v2 = v[1].y - v[2].y;
        v3 = p.y - v[0].y;
        v4 = p.y - v[2].y;
        v1 = v[0].z - v[2].z;
        v2 = v[1].z - v[2].z;
        v3 = p.z - v[0].z;
        v4 = p.z - v[2].z;
    } else { // Discard y, project onto xz plane
        // Code...
    }
}
```
Case of Projecting onto \( xz \) Plane

```c
} else if (fabs(n.y) >= fabs(n.z)) {
    // Discard \( y \), project onto \( xz \) plane
    u1 = v[0].z - v[2].z;
    u2 = v[1].z - v[2].z;
    u3 = p.z - v[0].z;
    u4 = p.z - v[2].z;
    v1 = v[0].x - v[2].x;
    v2 = v[1].x - v[2].x;
    v3 = p.x - v[0].x;
    v4 = p.x - v[2].x;
}
```

Case of Projecting onto \( xy \) Plane

```c
} else {
    // Discard \( z \), project onto \( xy \) plane
    u1 = v[0].x - v[2].x;
    u2 = v[1].x - v[2].x;
    u3 = p.x - v[0].x;
    u4 = p.x - v[2].x;
    v1 = v[0].y - v[2].y;
    v2 = v[1].y - v[2].y;
    v3 = p.y - v[0].y;
    v4 = p.y - v[2].y;
}
```

Finishing Up

```c
// Compute denominator, check for invalid
float denom = v1*x2 - v2*x1;
if (denom == 0.0f) {
    // Bogus triangle - probably triangle has zero area
    return false;
}

// Compute barycentric coordinates
float oneOverDenom = 1.0f / denom;
b[0] = (v4*u2 - v2*u4) * oneOverDenom;
b[1] = (v1*u3 - v3*u1) * oneOverDenom;
b[2] = 1.0f - b[0] - b[1];

// OK
return true;
```

Using Cross Product

- Another technique for computing barycentric coordinates in 3D is based on the method for computing the area of a 2D triangle using the cross product.
- Recall that given two edge vectors \( \mathbf{e}_1 \) and \( \mathbf{e}_2 \) of a triangle, we can compute the area of the triangle as \( \frac{1}{2} | \mathbf{e}_1 \times \mathbf{e}_2 | \).
- Once we have the area of the entire triangle and the areas of the three sub-triangles, we can compute the barycentric coordinates.
- There is one slight problem: the magnitude of the cross product is by definition always positive, so the barycentric coordinates given to us by this method are always positive.
- This is fine for points inside the triangle, but not so good for points outside the triangle, since these points must always have at least one negative barycentric coordinate.

Using Dot Product

- However, they may point in opposite directions.
- Recall from Chapter 2 that the dot product of two vectors is equal to the product of their magnitudes times the cosine of the angle between them.
- Since we know that \( \mathbf{n} \) is a unit vector, and the vectors are either pointing in the exact same or the exact opposite direction, we have
  \[
  \mathbf{c} \cdot \mathbf{n} = |\mathbf{c}| |\mathbf{n}| \cos \theta
  = |\mathbf{c}| |\mathbf{n}| (1, \pm 1)
  = \pm |\mathbf{c}|.
  \]
- Dividing this result by two, we have a way to compute the signed area of a triangle in 3D.

A Work-Around

- What we really need is a way to calculate the length of the cross product vector that would yield a negative value if the vertices were enumerated in the wrong order.
- There is a simple way to do this using the dot product.
- Let \( \mathbf{c} \) be the cross product of two edge vectors of a triangle. Remember that the magnitude of \( \mathbf{c} \) will equal twice the area of the triangle.
- Assume we have a normal \( \mathbf{n} \) of unit length. Now, \( \mathbf{n} \) and \( \mathbf{c} \) are parallel, since they are both perpendicular to the plane containing the triangle.
Vectors from \( v_i \) to \( p \)

- Armed with this trick, we can now apply the observation from earlier, that each barycentric coordinate \( b_i \) is proportional to the area of the subtriangle \( T_i \).
- As you can see in the diagram at right, each vertex has a vector from \( v_i \) to \( p \) named \( d_i \).

Vector Equations

Summarizing the equations for the vectors:
\[
\begin{align*}
e_1 &= v_3 - v_2 \\
e_2 &= v_1 - v_3 \\
e_3 &= v_2 - v_1 \\
d_1 &= p - v_1 \\
d_2 &= p - v_2 \\
d_3 &= p - v_3
\end{align*}
\]

Areas of Triangles

We also need a surface normal:
\[
\hat{n} = \frac{e_1 \times e_2}{|e_1 \times e_2|}
\]

The areas of the entire triangle \( T \) and the three subtriangles are:
\[
\begin{align*}
A(T) &= ((e_1 \times e_2) \cdot \hat{n})/2 \\
A(T_1) &= ((e_1 \times d_3) \cdot \hat{n})/2 \\
A(T_2) &= ((e_2 \times d_1) \cdot \hat{n})/2 \\
A(T_3) &= ((e_3 \times d_2) \cdot \hat{n})/2
\end{align*}
\]

Compute Barycentric Coords

The barycentric coordinates \( b_i \) are given by \( A(T_i)/A(T) \), as shown below:
\[
\begin{align*}
b_1 &= A(T_1)/A(T) = \frac{(e_1 \times d_3) \cdot \hat{n}}{(e_1 \times e_2) \cdot \hat{n}} \\
b_2 &= A(T_2)/A(T) = \frac{(e_2 \times d_1) \cdot \hat{n}}{(e_1 \times e_2) \cdot \hat{n}} \\
b_3 &= A(T_3)/A(T) = \frac{(e_3 \times d_2) \cdot \hat{n}}{(e_1 \times e_2) \cdot \hat{n}}
\end{align*}
\]

A Last Comment on the Method

- Notice that \( \hat{n} \) is used in all of the numerators and all of the denominators, and so it is not actually necessary that it be normalized.
- This technique for computing barycentric coordinates involves more scalar math operations than the method of projection into 2D.
- However, it is branchless and offers considerable opportunity for optimization by a vector coprocessor, thus it may be faster on a superscalar processor with a vector coprocessor.

Special Points

Three points on a triangle have special significance:
- Center of gravity
- Incenter
- Circumcenter

For each point, we will discuss its geometric significance and construction, and give its barycentric coordinates.
Center of Gravity

- The center of gravity, also known as the centroid, is the point on which the triangle would balance perfectly.
- It is the intersection of the medians.
- A median is a line from a vertex to the midpoint of the opposite side.

Computing the Center of Gravity

The center of gravity is the geometric average of the three vertices:

$$c_{Grav} = \frac{v_1 + v_2 + v_3}{3}$$

Its barycentric coordinates are:

$$\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right).$$

Incenter

- The incenter is the point in the triangle equidistant from all sides.
- It is the center of a circle inscribed in the triangle.
- It is the intersection of the angle bisectors.

Computing the Incenter

Suppose the triangle has sides of length $l_1, l_2, l_3$, and that $p = l_1 + l_2 + l_3$ is the perimeter. The incenter $c_{In}$ is given by:

$$c_{In} = \frac{l_1 v_1 + l_2 v_2 + l_3 v_3}{p}.$$

The barycentric coordinates of the incenter are

$$\left(\frac{l_1}{p}, \frac{l_2}{p}, \frac{l_3}{p}\right).$$

More on the Incenter

- The radius of the inscribed circle can be computed by dividing the area of the triangle by its perimeter $r_{In} = A/p$.
- The inscribed circle also solves the problem of finding a circle tangent to three lines.

The Circumcenter

- The circumcenter is the point in the triangle that is equidistant from the vertices.
- It is the center of the circle that circumscribes the triangle.
- It is constructed as the intersection of the perpendicular bisectors of the sides.
Computing the Circumcenter

Intermediate values:
\[ d_1 = -e_2 \cdot e_3 \quad c_1 = d_2 d_3 \]
\[ d_2 = -e_3 \cdot e_1 \quad c_2 = d_3 d_1 \]
\[ d_3 = -e_1 \cdot e_2 \quad c_3 = d_1 d_2 \]
\[ c = c_1 + c_2 + c_3 \]

Barycentric coordinates of the circumcenter are
\[ \left( \frac{c_2 + c_3}{2c}, \frac{c_3 + c_1}{2c}, \frac{c_1 + c_2}{2c} \right) \]

More on the Circumcenter

The circumcenter is given by:
\[ \mathbf{c}_{\text{Circ}} = \frac{(c_2 + c_3) \mathbf{v}_1 + (c_3 + c_1) \mathbf{v}_2 + (c_1 + c_2) \mathbf{v}_3}{2c} \]

The circumradius is:
\[ r_{\text{Circ}} = \frac{\sqrt{(d_1 + d_2)(d_2 + d_3)(d_3 + d_1)}}{2} c \]

The circumradius and circumcenter solve the problem of finding a circle passing through 3 points.

Section 9.7: Polygons

• A polygon is a flat object made up of vertices and edges.
• A simple polygon does not have any holes whereas a complex polygon may have holes.
• A simple polygon can be described by listing the vertices in order around the polygon. (Clockwise from the front in our left-handed world.)

Complex to Simple

• We can turn any complex polygon into a simple one by adding pairs of seam edges, as shown here.
• As the close-up on the right shows, we add two edges.
• The edges are actually coincident, although in the close-up they have been separated so you could see them.
• When ordered around the polygon, the two seam edges point in opposite directions.

Self-intersecting Polygons

• The edges of most simple polygons do not intersect each another. If the edges do intersect, the polygon is called self-intersecting.
• Most people usually find it easier to arrange things so that self-intersecting polygons are either avoided, or simply rejected.
• In most situations this is not a huge burden on the user.
Convex and Concave Polygons

• Non-self-intersecting simple polygons may be further classified as either convex or concave.
• Giving a precise definition for convex is actually somewhat tricky, since there are many sticky degenerate cases.
• For most polygons the following commonly used definitions are equivalent, although some degenerate polygons may be classified as convex according to one definition and concave according to another.

Rules for the Practicing Programmer

• “If my code that is only supposed to work for convex polygons can deal with it, then it’s convex.”
• “If my algorithm that tests for convexity decides it’s convex, then it’s convex.”

Convexity Testing

• How can we know if a polygon is convex or concave?
• One method is to examine the sum of the angles at the vertices. Consider a convex polygon with \( n \) vertices. The sum of interior angles in a convex polygon is \((n - 2)180^\circ\).
• There are two different ways to show this:

Three Definitions

1. Intuitively, a convex polygon doesn’t have any dents. A concave polygon has at least one vertex that is a dent, called a point of concavity.
2. In a convex polygon, the line between any two points in the polygon is completely contained within the polygon. In a concave polygon, we can find a pair of points in the polygon for which the line between the points is partially outside the polygon.
3. As we move around the perimeter of a convex polygon, at each vertex we will turn in the same direction. In a concave polygon, we will make some left-hand turns, and some right-hand turns. We will turn the opposite direction at the point(s) of concavity. (Note that this applies to non-self-intersecting polygons only.)

Some Easy to Agree on Examples

Convex polygons

Concave polygons

First Method

• First, let \( \theta_i \) measure the interior angle at vertex \( i \).
• Since the polygon is convex, \( \theta_i \leq 180^\circ \).
• The amount of “turn” that occurs at vertex \( i \) will be \( 180^\circ - \theta_i \).
• A closed polygon will of course “turn” one complete revolution, or 360°. Therefore,
Second Method

- As we will show a bit later, any convex polygon with \( n \) vertices can be triangulated into \( n - 2 \) triangles.
- From classical geometry, the sum of the interior angles of a triangle is 180°.
- Therefore sum of the interior angles of all of the triangles of a triangulated polygon is \((n - 2)180°\), and this sum must also be equal to the sum of the interior angles of the polygon itself.
- This appears to be a bit of a red herring because the sum of the interior angles is \((n - 2)180°\) for both concave and convex polygons.

Dot Product to the Rescue

- The trick is that for a convex polygon, the interior angle is not larger than the exterior angle.
- So, if we take the sum of the smaller angle (interior or exterior) at each vertex, then the sum will be \((n - 2)180°\), for convex polygons, and less than that for concave polygons.
- How do we measure the smaller angle? Luckily, we have a tool that does just that – the dot product. In Chapter 2 we learned how to compute the angle between two vectors using the dot product. The angle returned using this method always measured the shortest arc.

```cpp
bool isPolygonConvex(int n, Vector3 v[], int angleSum = 0.0f) {
    // Initialize sum to 0 radians
    for (int i = 0; i < n; ++i) {
        // Get edge vectors. We have to be careful on
        // the first and last vertices. Also, note that
        // this could be optimized considerably
        Vector3 e1 = v[i] - v[(i + 1) % n];
        e1.normalize();
        Vector3 e2 = v[(i + 1) % n] - v[i];
        e2.normalize();
        // Compute smaller angle using "safe" function that protects
        // against range errors which could be caused by numerical imprecision
        float dot = e1.dot(e2);
        // Sum it up
        angleSum += std::acos(dot);
    }
    // Figure out how much the sum of the angles should be, assuming
    // we are convex. Remember that pi/2 rad = 90 degrees
    float convexAngleSum = (float)(n - 2) * pi;
    // Now, check if the sum of the angles is less than it should be.
    // Then we're concave. We give a slight tolerance for
    // numerical imprecision
    if (angleSum < convexAngleSum - 0.0001f) {
        // We're concave
        return false;
    } else { // We're convex, within tolerance
        return true;
    }
}
```
Points of Concavity and Cross Product

• Another method for determining convexity is to search for vertices that are points of concavity. If none are found, then the polygon is convex.
• The basic idea is that each vertex should turn in the same direction. Any vertex that turns in the opposite direction is a point of concavity.
• We can determine which way a vertex turns using the cross product on the edge vectors. Recall from Chapter 2 that in a left handed coordinate system, the cross product will point towards you if the vectors form a clockwise turn.

Computing the Polygon Normal

• What does towards you mean in this case?
• We'll view the polygon from the front, as determined by the polygon normal. If this normal is not available to us initially, we have to exercise some care in computing it.
• Use the technique we saw earlier in this lecture for computing the best fit normal from a set of points.

Computing Points of Concavity from Polygon Normal

• Once we have a polygon normal, we compute a vertex normal at each vertex by taking the cross product of the adjacent clockwise edge vectors.
• We take the dot product of the polygon normal with the normal computed at that vertex, to determine if they point in opposite directions.
• If so (the dot product is negative), then we have located a point of concavity.

Triangulating Polygons

• Any polygon can be divided into triangles.
• Thus all of the operations and calculations for triangles can be piecewise applied to polygons.
• Triangulating complex, self-intersecting, or even simple concave polygons is no trivial task and is slightly out of the scope of our book.
• Luckily, triangulating simple convex polygons is a trivial matter. One obvious triangulation technique is to pick one vertex (say, the first one) and construct a triangle fan around this vertex.

Avoiding Sliver Triangles

• Fanning tends to create many long, thin sliver triangles, which can be troublesome in some situations, such as computing a surface normal.
• Certain consumer hardware can run into precision problems when clipping very long edges to the view frustum.
• Smarter techniques exist that attempt to minimize this problem. One idea is to triangulate as follows:
  – Consider that we can divide a polygon into two pieces with a diagonal between two vertices.
  – When this occurs, the two interior angles at the vertices of the diagonal are each divided into two new interior angles. Thus a total of four new interior angles are created.
  – To subdivide a polygon, select the diagonal that maximizes the smallest of these four new interior angles. Divide the polygon in two using this diagonal.
  – Recursively apply the procedure to each half, until only triangles remain.
• This algorithm results in a triangulation with fewer slivers.
That concludes Chapter 9. Next, Chapter 10:
Mathematical Topics from 3D Graphics