2D Geometry for Computer Graphics

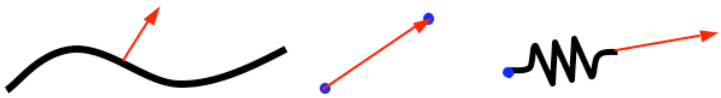
Jarek Rossignac

1 - Linear Algebra

We first discuss linear geometry in the plane.

1.1 Vectors and operators

A **vector** may represent a direction (for instance a curve tangent), a displacement between two points (for instance the correction between the actual and predicted location of a point), or a force (for instance the attraction exerted by a spring).

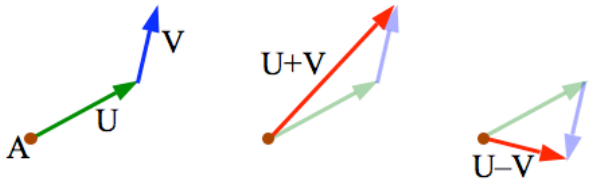


A vector has a length and a direction. The **length** of a vector V will be denoted V.norm and is often called its **magnitude** or **norm**. Note that a vector may have norm zero. We say then that it is a **null vector**.

The **direction** of vector V will be denoted V.direction. Given a direction D, we will denote the **opposite direction** by –D. We say that two vectors are **parallel** if they have either the same or the opposite direction. Consider two parallel vectors. If they have the same direction, we say that they have the **same orientation.** Otherwise, we say that they have **opposite orientation**.

Let s be a positive real number (scalar). The product sV denotes the vector V **scaled** by s, which has the same direction as V, but has as norm the product of s by V.norm. For example, 2V is a vector of the same direction as V, but twice longer. The **opposite** vector, denoted –V, has the same norm as V, but opposite direction. –V is obtained by rotating V by 180 degrees. Hence, –sV is a vector with norm s(V.norm) and opposite direction to V.

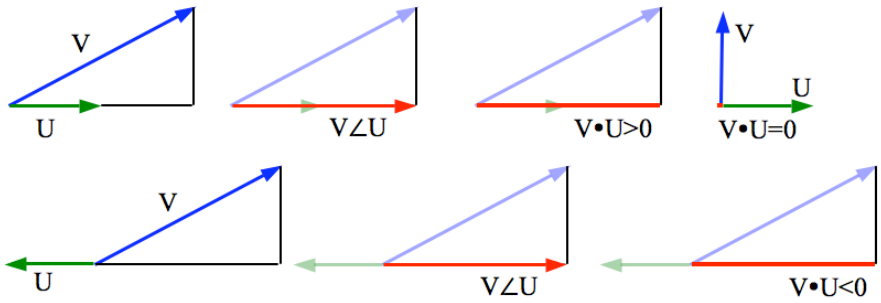
Note that a vector is defied by its direction and norm independently of its origin. Hence, we may draw it anywhere. The **sum** U+V of vectors U and V is best defined by thinking of the vectors as displacements. Start at some arbitrary point A. Move by U and then by V. U+V is the total displacement. The **difference** U–V is the same as U+(–V).



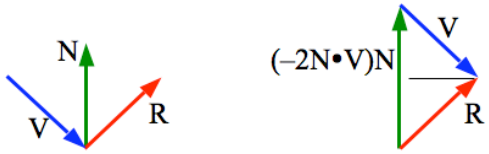
A vector whose norm equals 1 (measured in an agreed-upon unit) is called a **unit vector**. To simplify what follows, we will **represent a direction** D by a unit vector having D as direction. We will also use the word **direction** as a substitute for unit vector. Hence, V.direction of a non-null vector V is the scaling (1/V.norm)V of V by the inverse of its norm.

We will now review the most important operator on vectors: the dot-product. Let us first assume that U is a direction. The **projection** of V on U will be denoted V∠U and is a vector parallel to U. Its magnitude is the length

of the orthogonal projection of V onto U. The **dot-product** V•U of V with a U is a scalar whose magnitude is the norm of V∠U. It is essential to note that V•U=0 when U and V are **perpendicular** (or when at least one of them is null). Assume that U and V are not perpendicular. V•U is positive when V∠U has the same orientation as U and negative otherwise.



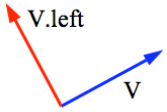
To illustrate the power of the dot product, consider the problem of computing the direction R, where an incoming ray of direction V is reflected after bouncing off a surface S at a point P where the surface is smooth and has normal direction N. The formula R=V–2(N•V)N is based on the observation that R–V= –2V∠N.



So far, we have defined the projection and dot-product between a vector V and a direction U. We will now generalize these to the case when U is an arbitrary vector, and not necessarily a direction. The projection of V onto U is U.norm(V∠U.direction). The dot-product between V and U is U.norm(V•U.direction). In other words, we use the direction of U, compute the projection or dot-product, and then scale them by the norm of U.

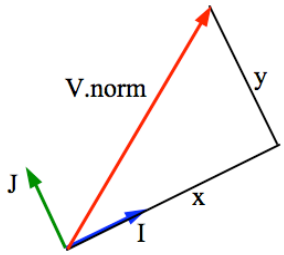
It is useful to note that V•U=(V.norm)(U.norm)cos(a), where a is the angle between U and V. From the symmetry of this formula, we conclude that the dot-product is commutative, i.e., V•U=U•V. Because cos(a+π) = –cos(a), we also conclude that –(U)•V=(–U)•V. Furthermore, the dot-product **distributes** over vector addition, hence (V+U)•W=V•W+U•W.

It is convenient to define the **orthogonal** vector, V.left, of vector V. V.left is obtained by rotating V by 90 degrees counterclockwise. Note that V and V.left have the same norm and that V•V.left=0.



1.2 Basis, components, and implementation

Consider two directions, I and J, such that J=I.left. They define a **basis** [I,J]. The **components** <x,y> of vector V in the basis [I,J] are defined as x=V•I and y=V•J. Note that, given a basis [I,J] and two components <x,y>, the vector V may be expressed as the **weighted vector sum** V=xI+yJ. Note that V.norm is the diagonal of a right triangle, hence, by **Pythagoras's theorem**, $x^2+y^2=V.norm^2$.



Consider two bases: $[I_1,J_1]$ and $[I_2,J_2]$. Let $<x_1,y_1>$ be the components of a vector V in $[I_1,J_1]$. If we wish to **change the basis**, we must compute the components $<x_2,y_2>$ of V in $[I_2,J_2]$. We do this by first computing $V=x_1I_1+y_1J_1$ and then $x_2=V\bullet I_2$ and $y_2=V\bullet J_2$.

To **implement** scaling, addition, subtraction, projection, and dot-product, we must decide on a **representation** for vectors. Usually, one represents vectors by their components in a **global basis** that is implicit, i.e., not specified. In two dimensions, it is customary to assume that the first basis vector (the X-direction) is horizontal pointing towards the right of the page and that the second basis vector (the Y-direction) is therefore vertical. Mathematicians often assume that it is pointing up, while some graphics packages assume that it points down. We call this a **Cartesian representation** of the vector.

Let <V.x,V.y> be the components of V. Note that if we do not specify a basis, we assume that the components are computed with respect to the global basis. Similarly, let <U.x,U.y> be the components of U. We implement the **operators** discussed above as follows:

$sU = < sU.x , sU.y >,$
$–U = < –U.x, –U.y>$
$U+V = < U.x+V.x , U.y+V.y>,$
$U–V = < U.x–V.x , U.y–V.y>$
$U.left = < –U.y , U.x >$
$V\bullet U = U.xV.x + U.yV.y$
$V.norm = \sqrt{(V\bullet V)}$
Assuming $U.norm\neq 0$: $U.direction = (1/U.norm)U$, $V\angle U = (V\bullet U/U.norm^2)U$

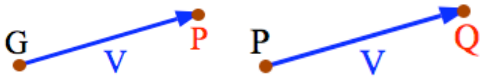
Assume that $U.norm\neq 0$ and $V.norm\neq 0$. U and V are **perpendicular** when $V\bullet U=0$ (i.e., when $U.xV.x + U.yV.y=0$) and **parallel** when $V\bullet U.orthognal=0$ (i.e., when $U.yV.x = U.xV.y$).

1.3 Points and coordinate systems

Points in two-dimensions are associated with locations in the plane. If we assume the existence of an implicit (unspecified) global **origin** G, to each point P of the plane corresponds a vector from G to P. We use the combination GP to denote that vector. Note that GP=P–G and that GP= –PG.

Consequently, we may use the components <x,y> of GP to represent P. However, we must not confuse points and vectors, which have different semantics and different operators. Consequently, when referring to a point P, we will call x and y its **coordinates** and will denote them using parentheses (x,y), rather than brackets.

Let (P.x,P.y) be the coordinates of point P and (Q.x,Q.y) be the coordinates of point Q. The **vector** PQ=Q–P has components <Q.x–P.x,Q.y–P.y>.



One **should not add points** because the results is dependent of the choice of G (which is implicit and hence should not impact the result of geometric calculations). However, for simplicity of notation combinations of points where the weights add up to one are often used. The most common one is the **midpoint**, mid(P,Q) of P and Q. It has coordinates ((P.x+Q.x)/2,(P.y+Q.y)/2). It should be expressed formally as P+PQ/2 using the operators discussed above, which formulate a vector (PQ) as the displacement between two points, then scale it by ½, then add it to a point P. Yet, for simplicity of notation, mid(P,Q) is often written as (P+Q)/2. Such a short cut is acceptable, provided that it has an equivalent **proper formulation**. More generally, a **weighted combination** of points P_i with weights w_i, where $\sum w_i =1$, is often written as $\sum w_iP_i$. This is acceptable, because there is an equivalent proper formulation: $P_0+\sum w_jP_0P_j$, with $j\neq 0$

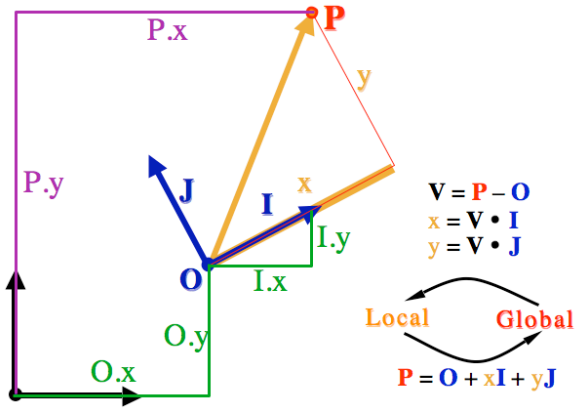
1.4 Rigid body transformations

Consider a point P represented by its coordinates (x,y) with respect to a **local coordinate system** in which the point was specified by the user as part of a shape. Now, we want to place the shape in a scene or to animate its motion through a scene. To do so, we will use compositions of two **rigid body transformations**: rotations and translations. The **translation** $T_V(P)$ of P by vector V is P+V. The **rotation** $R_a(P)$ of point P by angle a conter-clockwise around the origin yields the point (x cos(a) – y sin(a) , x sin(a) + y cos (a)). We can compose the effects of several transformations. For example, $T_V(R_a(P))$ will first rotate by a, then translate by V. Similarly, the

transformation $T_U(R_b(T_V(R_a(P))$ will first rotate by a, then translate by V, then rotate by b, and finally translate by U. Note that two consecutive translations commute and can be merged: $T_U(T_V(P))=T_V(T_U(P))=T_{U+V}(P)$. Similarly, consecutive rotations (in 2D) commute and can be merged: $R_b(R_a(P))=R_a(R_b(P))=R_{a+b}(P)$. However, rotations and translations do not commute: $T_V(R_a(P))\neq R_a(T_V(P))$.

In order to provide a fixed size (canonical) representation of the composition of an arbitrary number of rotations and translations, we can use a **coordinate system** [I,J,O] which defines the origin O and basis directions [I,J]. Given the coordinates (x,y) of point P in [I,J,O], we compute the point P=(P.x,P.y) in the global coordinate system of the scene as P=O+xI+yJ.

Occasionally, we may want to perform the **inverse transformation** and compute the coordinates (x,y) of some point P defined by its global coordinates (P.x,P.y). For example, the user may have selected a point on an instance of the object in the scene. To do so, we use the change of basis discussed above: x=OP•I, y=OP•J.



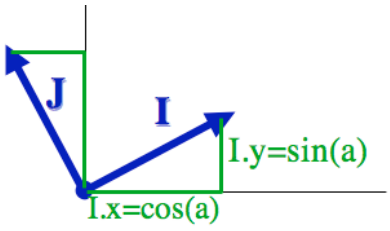
Consider now that we have a vector or direction V defined by their components <x,y> in the local coordinate system where the object was designed. How can we compute their components in the global coordinate system? We simply **ignore the translation part** (origin) and perform a change of basis, V=xI+yJ, as explained above. Remember that **vectors are not affected by translations**.

1.5 Homogeneous matrices

To use matrix multiplications for composing rotations and translations, we represent a transformation that defines a local coordinate system [I,J,O] by a 3×3 **homogeneous matrix** [**I.h J.h O.h**] where the .h operator maps a point (x,y) to its homogeneous counterpart (the homogeneous three-dimensional vector <x,y,1> obtained by **adding a 1** as third coordinate) and maps a vector <x,y> to its homogeneous counterpart (the homogeneous three dimensional vector <x,y,0> obtained by adding a zero as third coordinate). Notice the difference: points are padded with a 1 (so that we take translations into account) and vectors with a 0 (so that we do not take translations into account).

To transform a point in local coordinates (x,y), we perform the matrix-vector multiplication and compute <P.x,P.y,1> as [**I.h J.h O.h**](x,y,1) = x**I.h** + y**J.h** + **O.h**.

$$\begin{pmatrix} P.x \\ P.y \\ 1 \end{pmatrix} = \begin{pmatrix} I.x & J.x & O.x \\ I.y & J.y & O.y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$



Note that the top left 2×2 portion of that homogeneous matrix is the basis [I J], which represents a rotation R_a, where I.x = J.y = cos(a) and I.y = –J.x = sin(a). The top two entries of the right most column represent the translation vector. Hence, given any 3×3 matrix representing such a transformation or the composition of an arbitrary number of transformations, one can compute the **final rotation angle** a=atan2(I.y,I.x) and the **final translation vector** V=<O.x,O.y>. The function atan2(y,x) returns the angle $a\in[-\pi,\pi]$ between vector <x,y> and the x-axis. Note that this transformation corresponds to $T_V(R_a(P))$, where the rotation is performed first. Inversely, given angle a and translation vector V, we can compute the coefficients of the matrix.

We can similarly **transform a vector** V=<V.x,V.y> by multiplying its homogeneous formulation <V.x,V.y,0> by the homogeneous matrix. Note that the padded zero will multiply—and hence cancel—the translation effect.

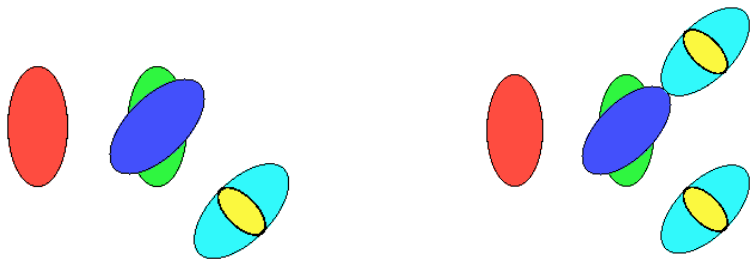
Also, note that the **inverse** of R_a is R_{-a}, which is obtained from R_a by changing the signs of the I.y and J.x coefficients, since sin(–a) = –sin(a) and cos(–a)=cos(a). Finally, the **inverse** of $T_V(R_a(P))$ is $R_{-a}(T_{-V}(P))$.

1.6 Transformations in graphics libraries

Graphics libraries provide support for these transformations. However, they must be called in **reverse order**. For example, to render the transformed point $T_U(R_b(T_V(R_a(P))$, we would issue the sequence of commands: `translate(U.x,U.y); rotate(b); translate(V.x,V.y); rotate(a); render(P);` To develop an intuitive understanding of this approach, consider that the calls to translate and rotate transform the global coordinate system with respect to which all subsequent transformation and rendering operations will be performed. Hence, in the sequence `{translate(U.x,U.y); rotate(b);}`the rotation is performed around the new origin, U. For instance, one may draw a tree

Graphics libraries also provide a **scaling** transformation, which is not a rigid body motion transformation. Specifically, `{scale(a,b); render(P);}` will render point (aP.x,bP.y). Scaling does not preserve distances, nor vector norms. Furthermore, when a≠b, the scaling operation will not preserve angles. Hence, one should be careful when transforming normal directions and other vectors by a scaling transformation.

Let `paint()` render an ellipse with height 100 and width 50. The sequence `{fill(red); paint(); translate(100,0); fill(green); paint(); rotate(PI/4); fill(blue); paint(); translate(100,0); fill(cyan); paint(); scale(1.0,0.25); fill(yellow); paint(); }` would produce the image below on the left, while the sequence `{fill(red); paint(); translate(100,0); fill(green); paint(); rotate(PI/4); fill(blue); paint(); pushMatrix(); translate(100,0); fill(cyan); paint(); scale(1.0,0.25); fill(yellow); paint(); popMatrix(); translate(0, -100); fill(cyan); paint(); scale(1.0,0.25); fill(yellow); paint(); }` would produce the image below on the right. Note that the command `pushMatrix()` pushes the current coordinate system (the one used to paint the blue ellipse) on the stack and that `popMatrix()` restores it. The global x-axis goes right. The y-axis goes down.



1.7 Lines, intersections, and half-spaces

A **line** may be defined **parametrically** as LineParametric(S,T) by a point S that it contains and by its tangent direction T. Given a scalar value t of the parameter, the corresponding point on the line is defined as P(t)=S+tT. Note that the absolute value |t| of t denotes the distance between S and P(t). Given two points A and B, the line passing through them may be defined as LineParametric(A,T), where T=AB.direction.

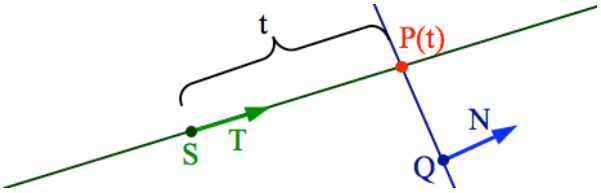
A **line** may also be defined **implicitly** as LineImplicit(Q,N) using a point Q that it contains and a normal direction N. Point P is on LineImplicit(Q,N) if it satisfies the implicit equation QP•N=0, which expresses the fact that vector QP is orthogonal to N and hence parallel to the line. Given two points A and B, the line passing through them may be defined implicitly as LineImplicit(A,N), where N=AB.direction.left, which is a short for (AB.direction).left.

Note that one can easily **convert** between parametric and implicit representations. In particular, the implicit representation of LineParametric(S,T) is LineImplicit(S,T.left) and the parametric representation of LineImplicit(S,N) is LineParametric(S,–N.left).

Hence, given a line L, we will use L.normal to obtain its normal direction, L.tangent to obtain it tangent direction, and L.point to obtain a point on it. The method L.Contains(P) returns true if P is **contained** in L. The method L.rightContains(P) returns true if P is contained in L.right.

Given a point P and a line L=LineParametric(S,T), we compute the **parameter t of point P with respect to line L** as t=SP•T. Note that P does not need to be in L.

To compute the **intersection** of two lines, it is easiest to express one in parametric form and the other one in implicit form. For example, to compute the intersection of LineParametric(S,T) expressed in parametric form with LineImplicit(Q,N) expressed in implicit form, one may substitute S+tT for P(t) in QP(t)•N=0 and solve for t, which yields (S+tT–Q)•N=0. Rearranging the terms and noticing that S–Q is QS, we obtain (QS+tT)•N=0. Distributing the dot-product over the vector addition yields QS•N+tT•N=0 and, if the lines are not parallel, t= –QS•N/T•N or equivalently t=SQ•N/T•N. When the lines are parallel, their intersection is a line when S lies in LineImplicit(Q,N) and the empty set otherwise.



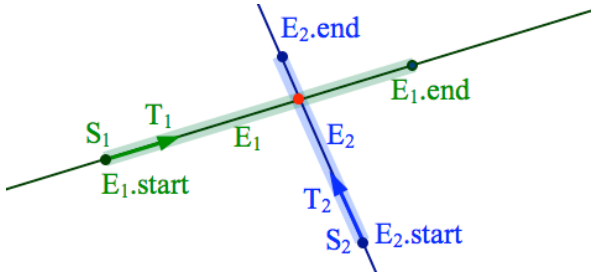
A line L=LineImplicit(Q,N) partitions the plane into three sets. The line itself is the set of points P satisfying equation QP•N=0. The **interior**, denoted L.interior, is the set of points satisfying equation QP•N<0. The **exterior**, denoted L.exterior, is the set of points satisfying equation QP•N>0. Note that these three sets are pairwise disjoint. L.interior and L.exterior are called **half-spaces**. They are **open** because they do not contain their boundary L. Sometimes, a **closed** half-space is desired. For example, the closed interior is the union of L with L.interior and may be denoted as L.interior.closure. Note that choice of terminology implies that the normal N points towards the exterior. Similarly, assume that you travel in the direction T along L=LineParametric(S,T). If we define the parametric form of L as LineImplicit(S,T.left), then L.interior is on your **right**, denoted L.right. Notice that the choice of T (or equivalently N) defines an orientation of a line and hence its interior (or equivalently its right) half-space. Imagine for example driving along the road (line L) that bounds a park on one side and a school on the other. Which one is on your right depends on the direction of travel. Once you have chosen a travel direction (T), assume that the normal (N) is T.left. The interior half-space is on your right.

1.8 Line segments, edges, and intersections

An **edge** E, from point A and point B will be denoted Edge(A,B). It is an oriented line segment. It is the set of points E(t)=A+tAB with t∈[0,1]. By rewriting tAB as t(B–A) and then distributing and collecting terms, one obtains an equivalent weighted sum formulation: E(t)=(1–t)A+tB. Let E.start denote the **starting point** A and E.end the **ending point** B of edge E. Edges may correspond to the sides of a polygonal region and may be oriented (clockwise) so that the interior of the polygon is on their right.

Let E.line be LineParametric(E.start,(E.end–E.start).direction), which contains E and inherits its orientation. Let E.right denote the **right half-space** E.line.right. A point P is in E.right, written E.rightContains(P), when (P–E.start)•(E.end–E.start).left<0.

Consider two edges: E_1 and E_2. How can we check whether they intersect? One approach is to compute the intersection point P of the two lines that each a different edge and then to check whether the parameter of P with respect to each line is comprised between zero and the length of the corresponding edge. Instead, we check whether each line splits the edge. We say that E_1 and E_2 **intersect** when (E_1.rightContains(E_2.start)≠E_1.rightContains(E_2.end)) && (E_2.rightContains(E_1.start)≠E_2.rightContains(E_1.end)).



1.9 Point-containment in triangles and polygons

Consider triangle T=Tri(A,B,C) with vertices A, B, and C. For simplicity, we assume that the three points are not collinear. The **triangle is oriented clockwise** if Edge(A,B).rightContains(C). If it is not, we will change its orientation be swapping B and C. A point P is in T (oriented clockwise) when Edge(A,B).rightContains(P)&& Edge(B,C).rightContains(P)&& Edge(C,A).rightContains(P).

1.10 Disks and circles

The **circle** of center C and radius r will be denoted Circle(C,r). Point P is **on the circle** when $CP^2=r^2$, where CP^2 stands for CP•CP. The **disk** of center C and radius r will be denoted Disk(C,r). Point P is **inside** the disk when $CP^2<r^2$.

To compute the **intersection** of LineParametric(S,T) with Circle(C,r), we replace P in $CP^2=r^2$ by S+tT and solve the resulting quadratic equation for t. Replacing P by S+tT in CP=P–C yields P–S–tT, which can be written SP–tT. Distributing • over – in (SP–tT)•(SP–tT) yields $(SP•SP)–2(SP•T)t+(T•T)t^2$. Given that T is a direction, T•T=1. With these changes, the equation defining the t-values of intersection points is: $t^2–2(SP•T)t+(SP^2–r^2)=0$. When two separate real roots, t_1 and t_2, exist, assuming $t_1<t_2$, the intersection of the line with Disk(C,r) is the line segment described parametrically by S+tT when $t∈]t_1,t_2[$.

Two disks, Disk(C_1,r_1) and Disk(C_2,r_2) **interfere** (i.e., have a non-empty intersection) when $(C_1C_2)^2<(r_1+r_2)^2$.

Resources

The Virginia tech Math Emporium has interactive practice tools for learning about vectors and matrices in the Math 1114 module http://www.emporium.vt.edu/math1114/index.html and on Vector Geometry in the Math 1224 module accessible from at http://www.emporium.vt.edu/math1224/index.html.

Practice

Exercises

1) Let V=<3,4>. Compute –V, 2V, V.norm, V.direction, V.left.
2) Are vectors <3,4> and <–9,–25> parallel?
3) Let U=<1,–2> and V=<3,4>. Compute U+V, U–V, U•V, V•U, V∠U, and U∠V. Which of these operators are commutative?
4) A ball arrives at velocity V=<3,–4>. What it its velocity U after an elastic collision with the ground (no gravity)?
5) Compute a coordinate system (O_1,I_1,J_1) that would be the result of a translation by <3,4> followed by a rotation of 90 degrees counterclockwise. Compute a second coordinate system (O_2,I_2,J_2) that would be the result of a rotation of 90 degrees counterclockwise followed by translation by <3,4>.
6) Provide two expressions for the center of mass of a triangle with vertices A, B, and C: one as a weighted combination of points and one using proper operators on points and vectors.
7) Let (x_1,y_1) be the coordinates of a point in the system (O_1,I_1,J_1). Let (x_2,y_2) be the coordinates of a point in the system (O_2,I_2,J_2). Provide closed-form conversion expression for computing (x_2,y_2) in terms of all the other variables.
8) Let U and V be two non-null vectors. Provide simple equations for testing whether U and V are parallel and whether U and V are orthogonal in terms of their coordinates <V.x,V.y> and <U.x,U.y>.

9) Write a test that will determine whether edge (A,B) intersects edge (C,D).

10) Given an edge E=Edge(A,B) and a point C, provide the pseudocode for computing point D that is the point of E closest to C. Then, compute the parameter t for D in a parametric representation of E.

11) Write the homogeneous matrix form representing a translation by <100,0> followed by a rotation of 90 degrees. What is the total translation of that transformation? What is its inverse matrix? Use the inverse matrix to identify its inverse translation and rotation parameters.

12) Use an example to show that $T_V(R_a(P)) \neq R_a(T_V(P))$.

13) Given a viewpoint E, a point P, and a screen between point A and point B, provide a test establishing whether the ray from A to P intersects the screen. Then, provide the construction for such an intersection point P'.

14) Provide a formulation for computing the description of a line that passes through point P and is parallel to a line that passes through points A and B.

15) Write the test for establishing whether a point P is on LineParametric(S,T) and if not, whether it is on the left of that line.

16) Write the details of a procedure that will test whether edge(A,B) intersects Circle(C,r).

Projects

1) Implement classes for points and vectors and provide methods for the operators discussed above.

2) Implement a program that tracks the position C of the cursor as you move the mouse and will compute and draw (as a small red disk) the closest point D to C on a polyloop P.

3) Compute the intersection points between two polyloops P and Q and render them (as small red disks) as one of the polygons is moved or edited by the user.

4) Compute the points of collision of a disk with a polyloop. Assume that the disk moves with constant velocitie between collisions and that the collisions are elastic. Compute the new velocity after each collision. Add a user interface for specifying the initial velocity of the a disk.

Problems

1) Compute the cost (measured as the number of arithmetic operations used) in the point-in-triangle test discussed above. Assume that you will be doing this test many times for the same triangle, (A,B,C), but for different candidate points P. Modify or improve the approach to reduce the expected cost when half of the points would be in the triangle.

2) You are looking on an architectural drawing. There is a short wall connecting points A and B. There is a long wall connecting points C and D. There is a light source at E. Provide the detailed geometric construction for computing the shadow of wall (A,B) onto wall (C,D). Provide test for detecting: (1) whether the shadow is empty, (2) whether it is complete, and (3) whether is covers (C,D).

3) Consider a family of rays from point E. Provide a geometric expression for testing whether a ray of the family intersects Circle(C,r). Assume that you will be shooting a large number of such rays. Provide the most efficient intersection test (you may assume that E, C, and r are fixed and may pre-compute some sub-expressions).