EECS 351 : Introduction to Computer Graphics

Points, Vectors, Vertices & Coordinate Systems

Jack Tumblin, ver. 1.8

Task 0: Write your first WebGL program of your own using Chap 1,2 'WebGL Programming Guide'. Search for interesting 2D/3D geometric patterns and tilings;

> Google for: tessellations, Celtic knots, Penrose Tiles and other tilings—hyperbolic tiles, Escher tiles; Spirograph patterns, Lissajous patterns, Harmonographs, fractal curves (e.g. Peano Curves), planetary gears, Jacquard looms, optical illusions, snowflake-like crystals, mathematical knitting, computed origami, Martin Gardner, Andrew Glassner, Erik Demaine, etc. Post your favorites on the discussion board; share.

Computer Drawing

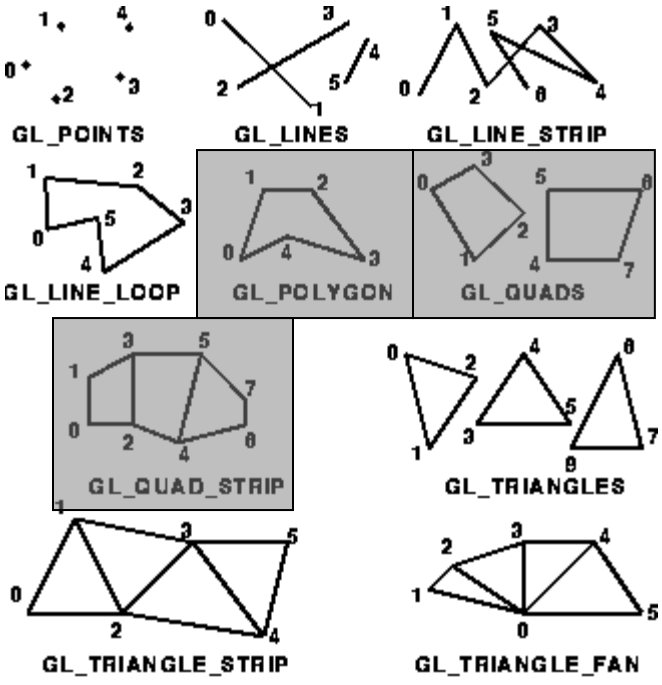
How would you make pictures with a computer-controlled pencil? ANS: Probably by a 'connect-the-dots' method--as WebGL does. Compute positions of points; draw straight lines between them. Curves? draw very short connected lines with slowly-varying orientation. Solids? fill in areas between lines.

Shape-Describing Terminology used for Computer Drawings

JARGON: 'vertex', plural: 'vertices' (not 'vertexes'!). The endpoint of one (or more) line segments.
JARGON: 'edge': a line segment that connects a pair of specified vertices.
JARGON: 'polygon': a closed sequence of edges, all contained within one plane (includes triangles)
JARGON: 'rasterize': to 'fill in' an on-screen area bounded by edges.
JARGON: 'mesh': a set of polygons with shared edges that describe one individual shape.
JARGON: 'model' a collection of vertices, edges, polygons, meshes, surface textures, etc. that describe a shape or scene. Verb form: 'to model' – to create a model for computer graphics use.

WebGL Drawing Primitives:

OpenGL 2.1 and earlier supports 10 basic drawing primitives, shown here, but OpenGL 3.0 and beyond has 'deprecated' three of them: GL_POLYGON, GL_QUAD, GL_QUAD_STRIP (shown in gray; these primitives just won't work in WebGL code). Avoid using deprecated primitives in any new code, as some vendors may not render them correctly on future graphics hardware.



WebGL draws each of these primitives from transformed lists of 'vertices' supplied by your program. While the drawing can get quite sophisticated (multiple texture-mapped materials, with transparency, lighting, bump maps, etc) OpenGL always uses the same 'connect-the-dots' strategy you might use for a computerized pencil:

To draw lines and edges, OpenGL/WebGL linearly interpolates between transformed vertex positions on-screen (and other vertex attributes: color, texture address, surface normals, etc.). To draw surfaces, WebGL extends this same linear interpolation to two or more dimensions. While programmable shading (OpenGL 2.1 & beyond) permits you to implement some other drawing methods, linear interpolation's powerful advantages still dominate most OpenGL/WebGL drawing processes.

LERP: == Linear Interpolation, or 'How to trace along an edge':

How can we get a computer to draw something by the so-called 'connect the dots' method?
Lets define an 'edge' as the line-segment that connects 2 specific points in space. To draw an 'edge' from point P0 to point P1, we need to somehow trace along the path between them. To do that, let's define a parameter 't', a parameter that varies between 0 and 1 (0<=t<=1) as we draw the edge. Then each and every point along the edge will have its own unique value for 't', and we can say that any point on the edge has coordinates P(t).
We'll use 'normalized' t values between zero and one: P(t) = P0 when t=0, and P(t) = P1 when t=1:

Then our parametric equation of an edge is:
P(t) = P0 + t*(P1-P0), or
P(t) = P0 + t*V using vector **V = (P1-P0).**

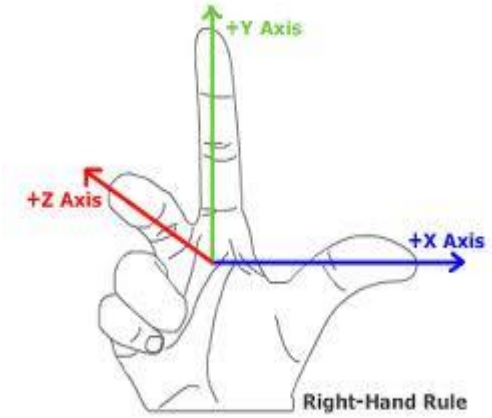
JARGON: This simple method is known as Linear Interpolation, or LERP.

You'll sometimes hear graphics people talk of 'LERPing' a pair of points, edges, vectors, colors, poses of animated characters, or even 3D meshes of connected vertices.

--Puzzle: what is P(t) when t>1 or <0? Can you prove that **P(t) = P1 – (1-t)*V** as well?
--Puzzle: how could you construct a P(t) as a weighted sum 4 points and two parameters(s,t)? Could you do it for just 3 points? Could you do it for 5 or more points? How?
--Puzzle: Could you combine two 'LERP' operations to assign a unique parameter pair for every point in a convex polygon made of 4 vertices? Could you extend the method to a polygon with just 3 vertices (a triangle)?
--Puzzle: Use Google to look up the term 'Barycentric coordinates': investigate and think about how you might use it to solve programming problems in graphics. Can you construct 'Barycentric Coordinates' using just LERPs? If not, what else do you need?
--Puzzle: Orderly extensions of the LERP idea also lets P(t) trace out curves and surfaces as you vary s and t. See DeCastlejau construction, Bezier Curves (look for interactive demos on the web).

Coordinate Systems Review

We need exact, always-consistent definitions of coordinate systems to draw OpenGL/WebGL primitives exactly where we want them. For drawing or for any other computer graphics operations, we will always use a 'right-handed coordinate system' to specify exact locations and to make any measurements. A scant few early picture-making systems (e.g. PHIGS) mixed left-handed and right-handed coordinate systems, but died out (or were killed off by aggravated users?). In OpenGL, DirectX, this course and most graphics books, ALL coordinate systems are ALWAYS **right handed**.



Recall that a Cartesian coordinate system is actually a construction, an assemblage of points and vectors that label all points in an n-dimensional space with a unique n-tuple of real numbers.

To build a Cartesian coordinate system that labels every point in 3D space with its own unique 3-number 'coordinate value', we choose one fixed point '**O**' as the origin point, we choose 3 mutually perpendicular directions outward from that point (usually named x,y,z respectively), and then define a unit-length vector for each of those directions (vectors usually named **i,j,k** respectively) emerging from the origin **O**. This trio of mutually-perpendicular vectors can specify just two possible geometric arrangements (or 'ordering' or 'chirality') for the coordinate system.

To identify a `right handed' coordinate system, flex your thumb, index, and middle finger of your right hand to mutually-perpendicular directions, as shown. If you align your thumb to point in the +x-axis (**i** vector) direction and your index finger to point in the +y-axis (**j** vector) direction, then your middle finger points in the +z-axis (**k** vector) direction.
Equivalently, if you point your right-hand thumb along the +z axis (**k** vector) direction, your fingers will curl naturally across the x-axis (**i** vector) and then the y-axis (**j** vector) just 90 degrees away. To convert a left-handed coordinate system to a right-handed system, just reverse the direction of the z axis.

Cartesian coordinate systems in 3D assign unique scalar coordinates (x,y,z) to each and every point in 3D space. By a simple construction, you can find the location of any individual 3D point from its coordinates alone; multiply each of the 3 unit-length 'basis' vectors (**i,j,k**) by its corresponding scalar (x,y,z) coordinate, and add these scaled vectors to the origin point:

$$\mathbf{P}(x,y,z) = \mathbf{O} + x*\mathbf{i} + y*\mathbf{j} + z*\mathbf{k}$$

If we move the origin point or change the directions of the 3 basis vectors that define the coordinate system, then of course, the coordinate scalars (x,y,z) assigned to any fixed point '**P**' must change as well. Conversely, if we leave the origin and the axis-defining vectors (**i,j,k**) fixed and unchanged but change the (x,y,z) values, then **P** must 'move' to a different location in 3D space.

To find the coordinates of any given 3D point **P**, imagine an infinite plane perpendicular to the **i** vector that defines the x axis. If we slide that plane along the x axis (but keep it perpendicular to **i**) we can find one special position where the plane passes through point **P**. Then the point where the plane meets the x axis will have the same x-coordinate as point **P**. We can find the y and z coordinates of **P** by repeating the process with planes perpendicular to the y and z axes respectively. (We can also find these same coordinates by computing 'dot products', explained later).

For nearly all picture-making by mathematical plotting and drawing, we just define points on a rectangular portion of a 'drawing plane' or 'display plane' or 'image plane' where z=0, and draw lines between those points, usually by LERPs. Don't call each one of them a 'point' however—the proper term in computer graphics (and geometry) is a single '**vertex**' or multiple '**vertices**' (***never*** "vertexes"), because a vertex is a *group of attributes* that describe a single point. OpenGL lets you specify and use as many or as few attributes as you wish, including: position (x,y,z,…), color (R,G,B, etc), surface normal vector, texture coordinates, and even user-defined attributes (OpenGL 3.0 and beyond). One **vertex** describes attributes for just one location, and OpenGL accesses vertex attributes as it renders (draws) drawing primitives on-screen, built up from the information held in lists of vertices.

Encapsulation:

A surprisingly large portion of all computer graphics work consists of encapsulation; unifying and hiding lots of the messy details of drawing, and distributing them among many SIMD processors (single-instruction, multiple-data; all work on the same instruction at the same time). A good graphics system finds clean, simple, easy-to-control and easy-to-adjust methods for all desirable graphical

operations. As very nearly ***everything*** in graphics is done by linear algebra (e.g. multiply, add, and weighted sums), very nearly ***everything*** is expressed as a 4-element vector or 4x4 matrix operation. '***Everything***'==equations of lines, planes, and many curved surfaces; the way a ray of light travels through space, how much light a surface absorbs, transmits, or reflects as a function of incoming angle, outgoing angle, position, wavelength, and time; how humans estimate 'color' from spectral power distribution of visible light wavelengths, the way we store images as strokes, dots or pixels; the way we measure rigid movements, find collisions, simulate fluids, etc. etc.

Now that we've defined coordinate systems, we'll use them to describe the vertices, vectors and matrices we use for shape description:

3D Points (Vertices) and 3D Vectors (Edge directions)

For reasons we'll learn later, computer graphics methods routinely describe 3D points and 3D vectors using a 4-tuple—an array of 4 elements with coordinate values x,y,z,w.
What's 'w'? for now, just set it to 1; Why? You'll understand when you're older (a few weeks older).
OpenGL, this course, and most graphics books write this 4-tuple as a **column (**a matrix of 4 rows and 1 column) and *never* write it as a **row** (a matrix of 1 row with 4 columns).

(CAUTION: Microsoft's proprietary DirectX is trivially different: see 'ASIDE' below).
Note that we can write a 4-element column compactly as $[x\ y\ z\ w]^T$, and this column of numbers describes either a geometrical **point** or a geometrical **vector**. What's the difference?

'Point' == a position in space. $[xpos\ ypos\ zpos\ 1]^T$ == a location with no direction or magnitude.
'Vector'==direction and magnitude in space. $[xdir\ ydir\ zdir\ 0]^T$ == a direction and magnitude with no intrinsic location. A vector describes displacement; a *difference* between 2 points.

If given two 3D points $\mathbf{P_0}$, $\mathbf{P_1}$, we can always define a vector **E** by their difference: $\mathbf{E = P_1 - P_0}$;
what happens to 'w' in a vector? **Answer: as w=1 for points, then we must have w=0 for vectors.**

The difference between a point and a vector may seem a trivial, fussy and subtle distinction, because every geometrical point in 3D space can specify a corresponding vector from the origin to that point. Yet in computer graphics, the point and the vector are NOT interchangeable: the 3D point has a specific position, but the vector does not. Mixing them can cause subtle bugs!
Is an edge the same as a vector? Not quite—it is a link between 2 vertices: its position DOES matter.

THUS we define a (geometrical) 'vector' as the result of *subtraction* of 2 points; it has direction and length, but no position. What happens when we apply addition, multiplication, and division, averaging, weighted sums (or "LERPing"), and other math operators to points and vectors? Which combinations of operators and operands offer sensible tools for 3D shapes, and which ones have no geometric meaning?

--(Point,Point): Binary: ~~Add?~~ Subtract(makes a vector), Average, weighted sum,
~~multiply?~~ ~~divide?~~
Unary: ~~Scale?~~ ~~Sign-change~~?(BAD! Results vary with choice of origin point)
--(Point,Vector) Binary: Add, Subtract (point minus vector makes a point),
~~average~~, ~~weighted sum~~, ~~multiply~~, ~~divide~~?
--(Vector,Vector)? Binary: Add, Subtract (makes a vector), Average, Weighted sum,
multiply (two ways—see below), ~~divide~~?
Unary: Scale? Sign-change?