COMP 261: Data Structures and Algorithms

Luke O'Donnell

2020-11-03

Contents

Chapter 1

Graphs

A graph data structure consists of a finite (and possibly mutable) set of vertices (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges (also called links or lines), and for a directed graph are also known as arrows. Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

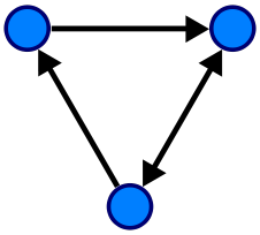

Figure 1.1: A directed graph with three vertices and three edges

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).
Graphs have many real world applications including:

- Locations with connections e.g. airports and flights, railways and train stations, intersections and roads
- Entities with relationships e.g. Social networks, Web pages

We will only consider directed graphs. Undirected graphs can be seen as a special case of a directed graph where the edge can be seen as a pair of directed edges e.g. (A, B) can be seen as (A $\rightarrow$ B) and (B $\rightarrow$ A). Complex graphs can have other properties such as loops (A, A) and parallel edges (multiple edges existing between the same pair of nodes.

Table 1.1: common graph operations

Operation	Description
adjacent(G, x, y)	tests whether there is an edge from the vertex x to the vertex y
neighbours(G, x)	lists all vertices y such that there is an edge from the vertex x to the ver
add_vertex(G, x)	adds the vertex x, if it is not there
remove_vertex(G, x)	removes the vertex x, if it is there
add_edge(G, x, y)	adds the edge from the vertex x to the vertex y, if it is not there
remove_edge(G, x, y)	removes the edge from the vertex x to the vertex y, if it is there
get_vertex_value(G, x)	returns the value associated with the vertex x
set_vertex_value(G, x, v)	sets the value associated with the vertex x to v
get_edge_value(G, x, y	returns the value associated with the edge (x, y)
set_edge_value(G, x, y, v)	sets the value associated with the edge (x, y) to v

What *data structure* should be used to represent a graph?

1.1 Representations

A proper data structure to represent a graph should support the following common graph operations efficiently.

There are two traditional data structures used for representing graphs in practice.

1. Adjacency Matrix
2. Adjacency List

1.1.1 Adjacency Matrix

An adjacency matrix is a square matrix used to represent a finite graph. The number of rows and columns in the matrix are equal to the number of nodes in the graph. The elements in the matrix indicate whether the pair of vertices are adjacent in the graph.

- $M_{i,j} = 1$ if there is an edge from node i to node j
- $M_{i,j} = 0$ otherwise

This form of an adjacency matrix (where elements are 0 or 1) cannot be used to represent a weighted graph i.e. edges with lengths.

- $M_{i,j} = w_{i,j}$ is the weight of the edge from node i to node j
- $M_{i,j} =$ `null` if there is no edge from node i to node j

However this and the previous form of the matrix cannot be used to represent graphs where there are parallel/multiple edges between same pair of vertices.

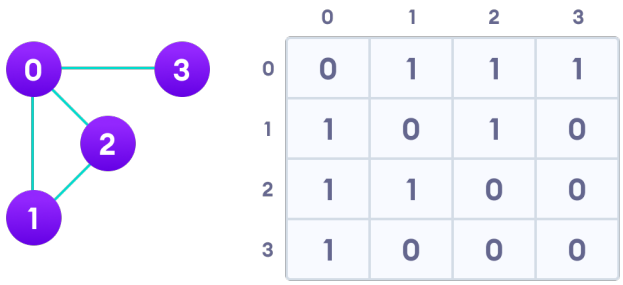

Figure 1.2: Adjancency matrix from a graph

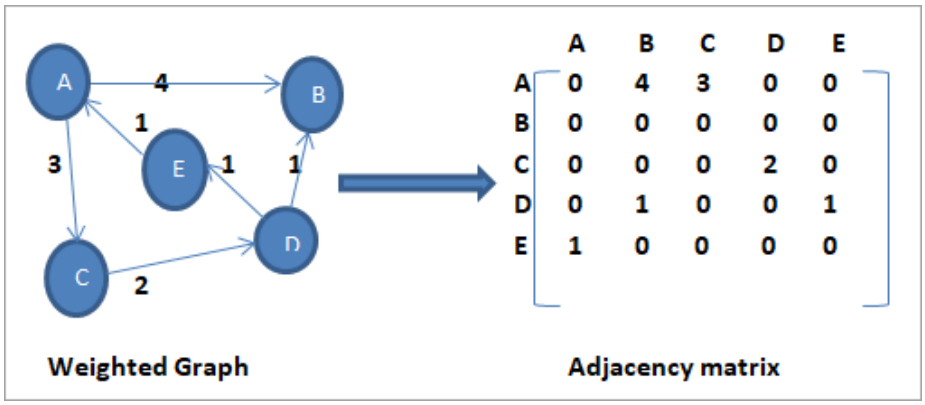

Figure 1.3: Adjancency matrix from a weighted graph

There is an additional implementation of a finite graph using an adjacency matrix that would allow it to represent complex finite graphs(has loops and parallel edges).

- $M_{i,j}$ is a set of edge objects[1]
- If the length of $M_{i,j}$ is 1, then there is a single edge between node i and node j
- If the length of $M_{i,j}$ is 0, then there is no edge connecting node i to node j
- If the length of $M_{i,j}$ is > 1 then there multiple parallel edges between node i and node j.

1.1.2 Adjacency List

1.1.3 Time Complexity

1.2 Displaying Graphs

1.2.1 Coordinate Systems

[1] Each instance of an Edge object should be unique, even in the cases when they are connected to the exact same nodes. This is an example of when you shouldn't overwrite the equals method.

Chapter 2

Trie

Here is a review of existing methods.

Chapter 3

Quad Tree

We describe our methods in this chapter.

Chapter 4

Path Finding

Some *significant* applications are demonstrated in this chapter.

4.1 Example one

4.2 Example two

Chapter 5

Articulation Points

Chapter 6

Minimum Spanning Trees

Chapter 7

Disjoint-Sets

7.1 Kruskal's Algorithm Revisited

The most time consuming step in Kruskal's algorithm is finding whether two node's are in the same tree (part of the forest) and if not merging the two tree's together.

7.1.1 Complexity

Two operations dominate the efficiency of Kruskal's algorithm

- **Find:** Determine whether two elements are in the same *tree* of the *forest*
- **Union:** Merge two trees into one

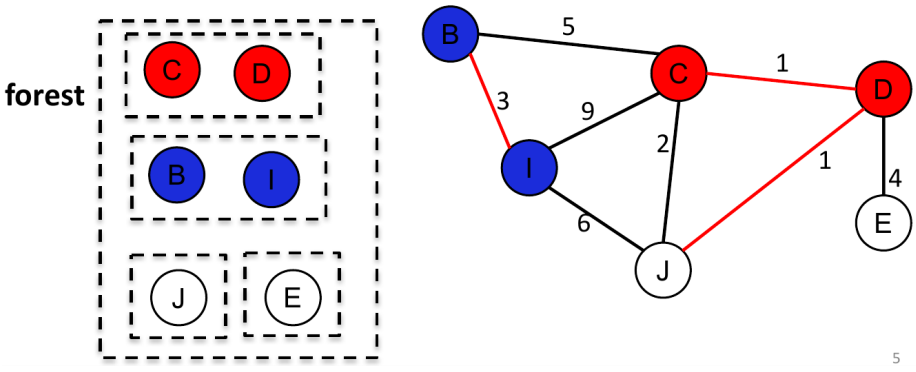

Figure 7.1: Finding the MST for a graph using Kruskal's algorithm

The cost of these two operations depend on the data structure used for the **forest**

7.1.1.1 Option 1: Set of Sets

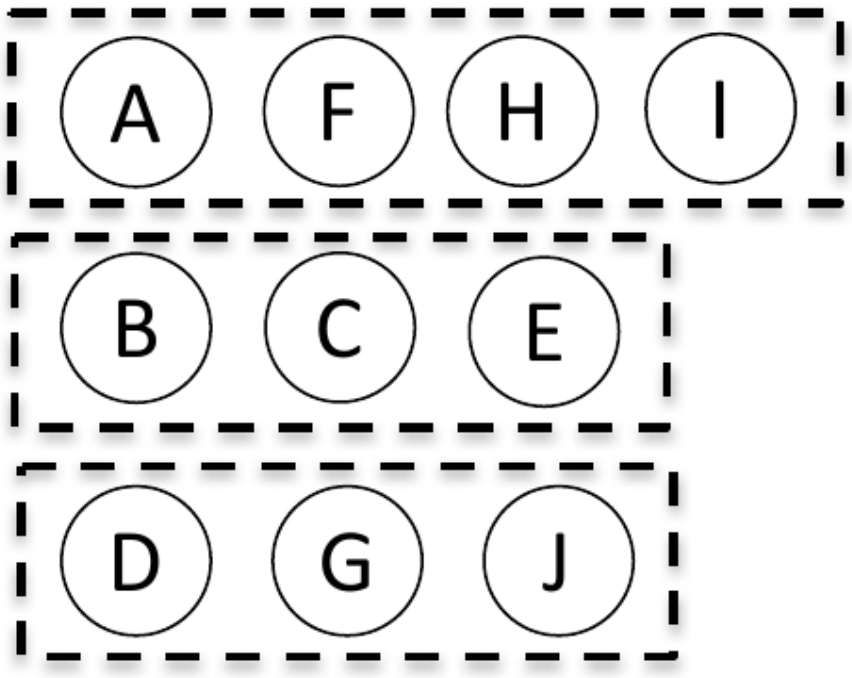

Figure 7.2: Representing the forest with a set of sets.

The cost of finding each node in the forest is *O(N)* as you must iterate through each element in the outer set to find the node you're looking for. Similarly cost of merging two trees is *O(N)* as you must add all elements from one set into the other set.

A set of sets e.g. `HashSet<HashSet<Node>>` would be a poor choice as the data structure of the forest.

7.1.1.2 Option 2: Add a Set Id to each Node

The cost of finding each node in the forest is *O(1)* as you only have to check if the two nodes have the same set Id. However, the cost of merging two trees is still *O(N)* as you must change the set Id of all the nodes in one set to the same Id in the other set.

Adding a identifier to each node is an improvement over a "set of sets" as the data structure of the forest. But, there is an even better data structure to use, the **Disjoint-Set**.

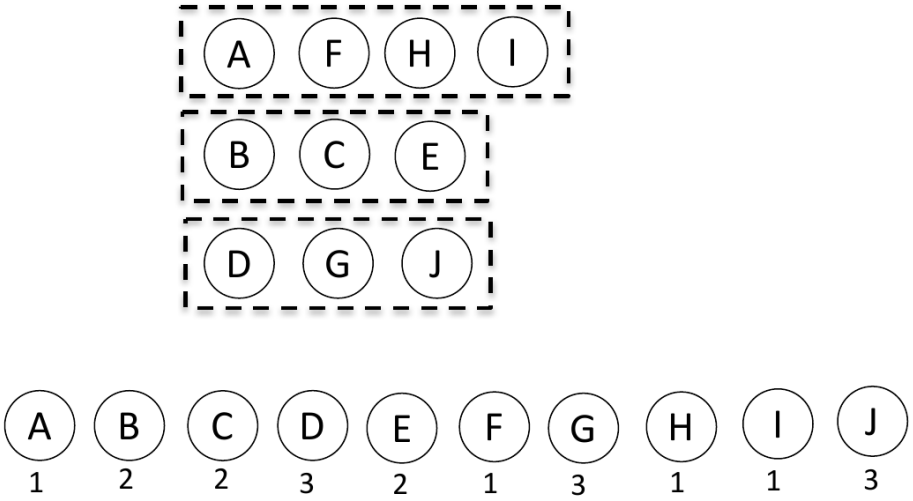

Figure 7.3: Adding a set id to each node

7.2 Disjoint-Set Data Structure

A disjoint-set, also called a union-find data structure, stores a collection of disjoint/non-overlapping sets. It provides efficient, near *O(1)* operations for finding a member in the set and merging sets.

The most common implementation of a disjoint-set data structure is called a *disjoint-set forest*

Under this representation a parent pointer tree is used to represent each element in the forest. A parent pointer tree can be thought of as an inverted tree; each node that isn't the root points to its parent node.

- Two nodes are in the same set if and only if the roots of the trees containing the nodes are equal.
- The forest is the set of the root node** of each parent pointer tree.

7.2.1 Operations

Disjoint-set data structures support three operations: Makinga new set containing a new element, finding the root of the set containing a given element, and merging two sets.

7.2.1.1 MakeSet

The `MakeSet` operation adds a new element. The element is placed into a new set containing only the new element, and the new set is added to the forest.

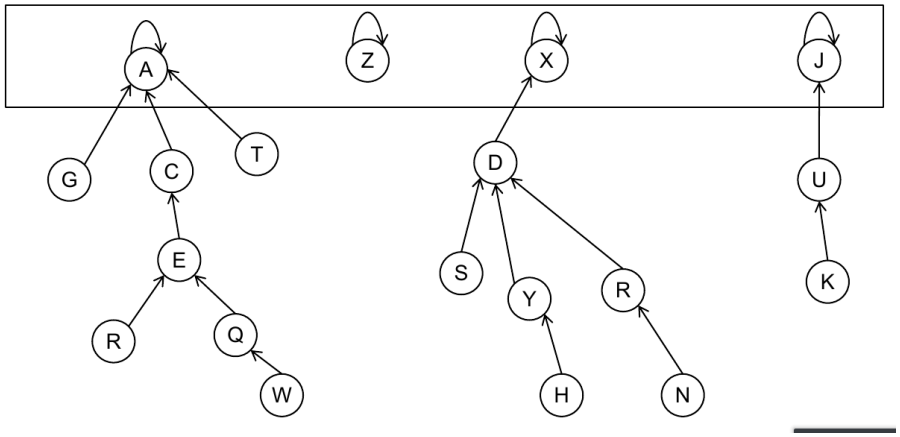

Figure 7.4: A representation of the disjoint-set forest. The forest is set containing the root nodes of each disjoint-set. Each disjoint set is parent-pointer tree

The `MakeSet` operation has **O(1)** complexity. In particular, initializing a disjoint-set forest with **n** nodes requires **O(n)** time.

Figure 7.5: Initializing a new disjoint-set with element x as the root

7.2.1.2 Find

The `Find` operation follows the chain of parent pointers from a specific node **x** until it reaches the root element. This root element represents the set to which **x** belongs and may be **x** itself. `Find` returns the root element it reaches.

The time in a `Find` operation is spent chasing parent pointers, so a flatter tree leads to faster `Find` operations.

Better performance, can be achieved by updating the parent pointers during a pass to point closer to the root. Because every element visited on the way to the root is part of the same set, this does not change the sets stored in the forest. But it makes future `Find` operations faster.

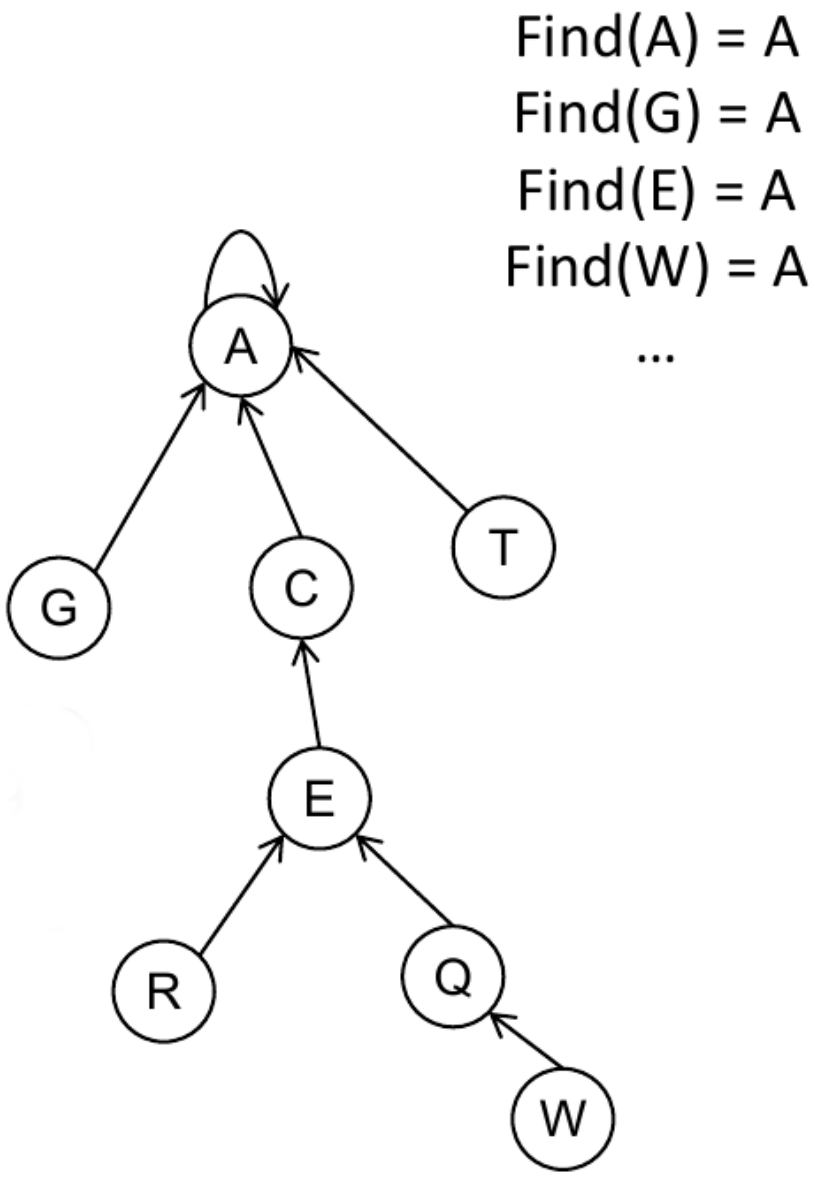

Figure 7.6: Find which tree several elements beloing to

There are several algorithms for Find that achieve the asymptotically optimal time complexity. One family of algorithms, known as path compression, makes every node between the query node and the root point to the root. Path compression can be implemented using a simple recursion as follows:

This implementation makes two passes, one up the tree and one back down.

7.2.1.3 Union

The operation `Union(x,y)` merges the two sets containing **x** and the set containing **y** together. `Union` first uses `Find` to determine the roots of the trees containing `x` and `y`:

- If the two roots are the same, there is nothing to do.
- Otherwise, the two trees must be merged. This is done by either setting the parent pointer of **x** to **y** or vice versa.

The choice of which node becomes the parent is consequential. If it is done carelessly, trees can become excessively tall, resulting in decreased performance of future `Find` and `Union` operations.

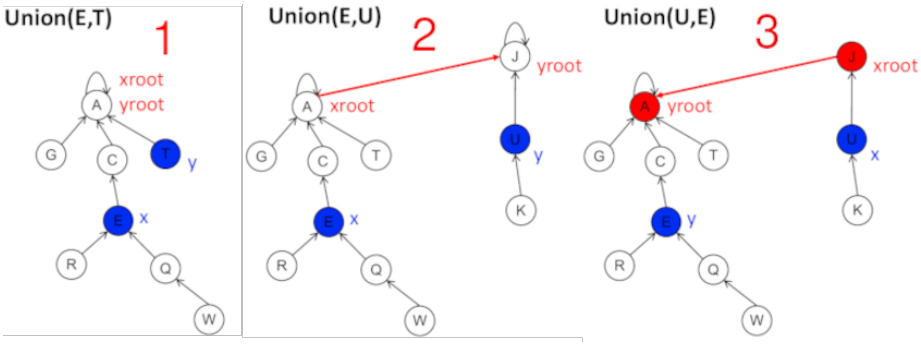

Figure 7.7: Merge sets

In order to ensure the optimal performance, shallower trees should be merged into deeper trees. This requires additional information to be stored in the disjoint-set, the **depth**.

Storing the depth requires, modifying the `MakeSet` operation

The `Union` operation can now make use of the **depth** to merge the shallower tree into the deeper tree.

Using a disjoint-set as the data structure for the forest gives near **O(1)** performance for determining whether two elements belong to the same set and merging two sets into one. This massively improves the performance of Kruskal's algorithm for MST.

Chapter 8

3D Rendering

8.1 3D Graphics

3D graphics have many applications, including in movies, games and animations. Rendering is the act of displaying 3D objects on a 2D screen.

Figure 8.1: Examples of the applications of 3D graphics

When rendering a 3D object several properties of the object must be considered:

- Shape
- Surface properties
- Material/Mass

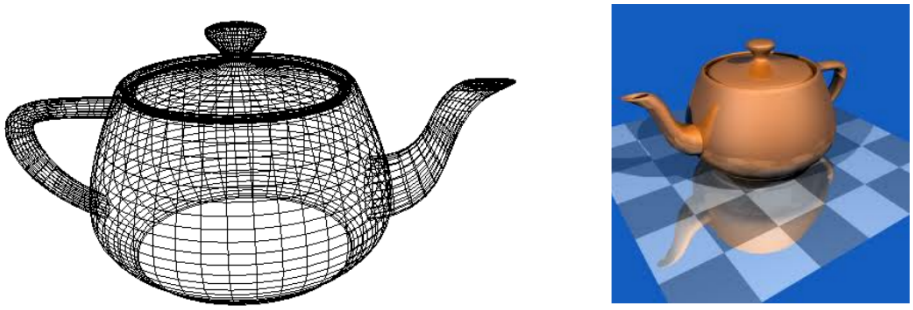

Figure 8.2: 3D object rendered on a 2D screen

- Movement/Animation
- Light sources

Only the visible surface's of a 3D object need to be drawn; any hidden parts can be ignored. This raises two questions:

- How can the visible surfaces be identified
- How can the visible surfaces be rendered on a computer screen.

Pixels are an elementary unit for representing 2D objects, however they are not performant enough to be used for representing the surfaces of 3D objects. An alternative rendering unit are polygons. Polygons can use triangles, squares and many other shapes to approximate the surfaces of 3D objects.

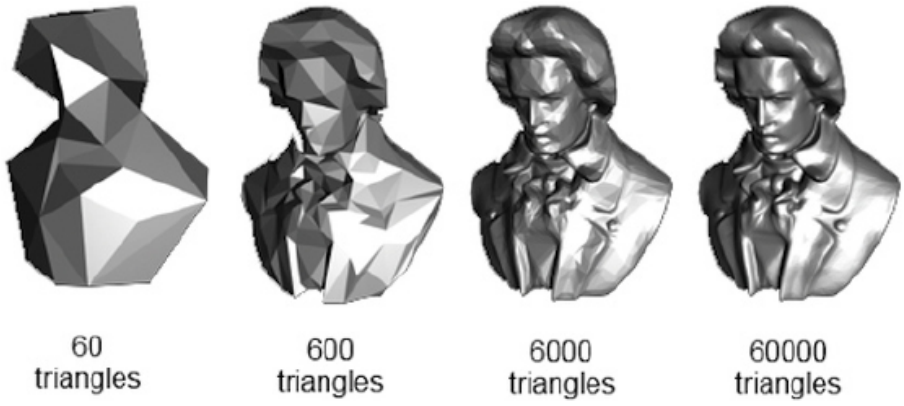

Figure 8.3: Polygons being used to approximate the surface of a 3D surface. The higher the resolution, the better the image

8.2 Polygon

A polygon consists of a list of vertices. Each vertex is a point in 3D space. The locations of each of the vertices is represented with a 3D coordinate (x, y, z)

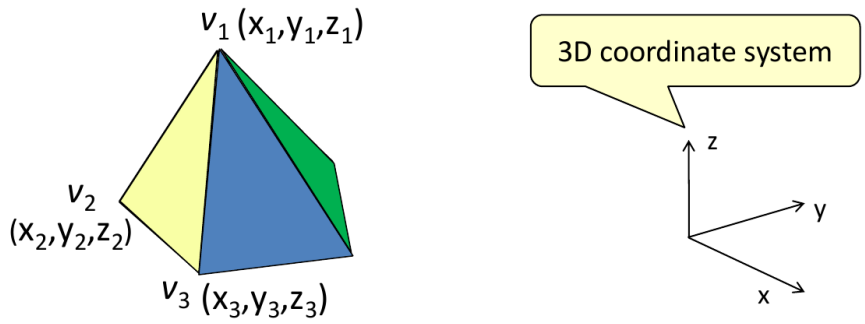

Figure 8.4: Polygons being used to approximate the surface of a 3D surface. The higher the resolution, the better the image

The locations of each vertex is represented with a 3D coordinate (x, y, z).

8.2.1 3D coordinate system

The 3D coordinate system used for rendering 3D objects, uses the right-hand rule to understand the orientation of the three axes in 3D space. For right-handed coordinates the right thumb points along the Z axis in the positive direction and the curl of the fingers represents a motion from the first or X axis to the second or Y axis. When viewed from the top or Z axis the system is counter-clockwise.

For left-handed coordinates the left thumb points along the Z axis in the positive direction and the culrled fingers of the left hand represent a motion from the first or X axis to the second or Y axis. When viewed from the top or Z axis the system is clockwise.

Under either rule the axes are ordered x, y and z

8.2.2 3D transformations

Like their 2D counter-parts transformation of 3D polygons i.e. translation, scaling and rotation requires changing the coordinate of each vertex in the polygon using linear algebra

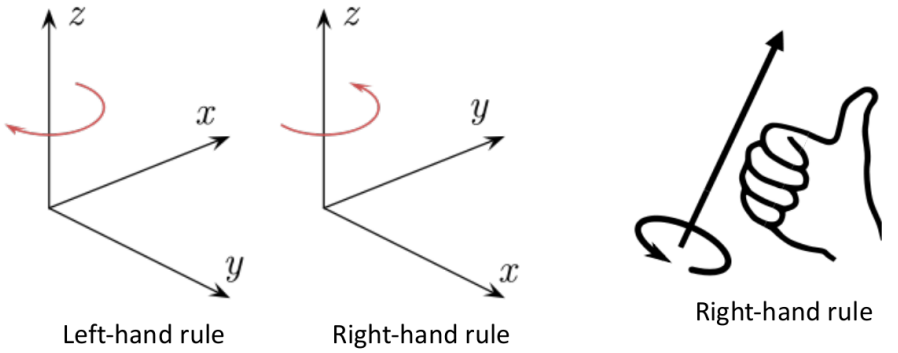

Figure 8.5: Left-handed coordinates on the left, right-handed coordinates on the right.

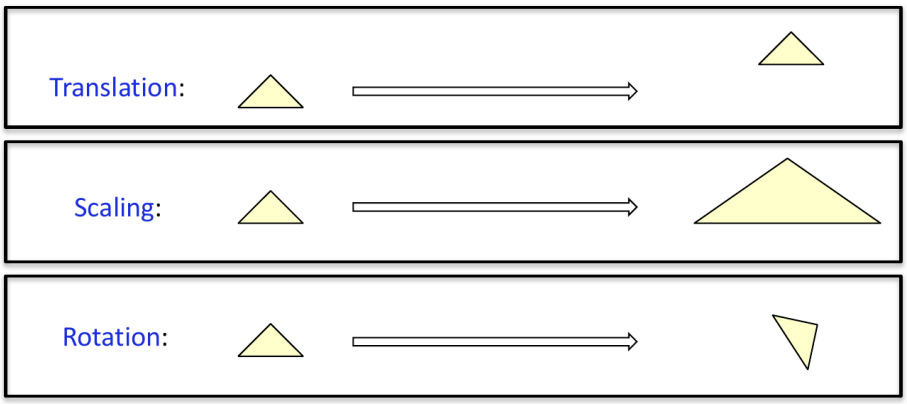

Figure 8.6: Translation, scaling and rotation

8.3 Linear Algebra: Basics

There are two basic structures to be considered in linear algebra, are the vector and matrix:

- A vector is a n dimensional list of numbers
- A matrix is an m-row-by-n-column array of numbers

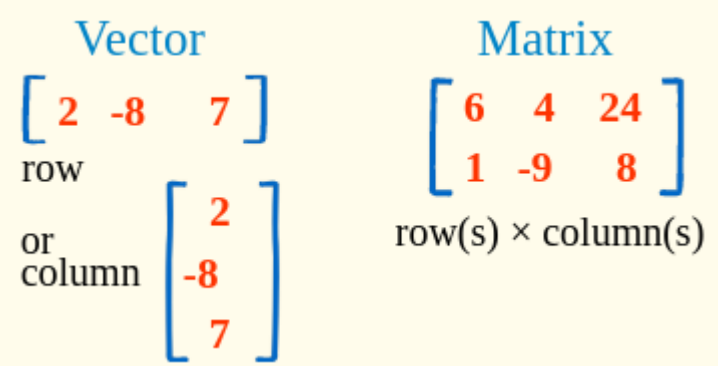

Figure 8.7: Scalars, Vectors and Matrices

A n-dim vector can be generalised as a n-row-1-column matrix. So the rules of linear algebra that apply to matrices also apply to vectors.

8.3.1 Matrix Addition

In order to add two matrices, both matrices must have the same dimensions (same number of rows and columns). If this condition is met, then matrix addition consists of adding the element in one matrix to the corresponding element in the other matrix.

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

Figure 8.8: Add the elements in one matrix to the corresponding elements in the other matrix

8.3.2 Matrix Multiplication

The condition for multiplying two matrices, is that the number of columns in matrix 1, must be the same as the number of rows in matrix 2.

- Matrix 1: m-row x n-col
- Matrix 2: n-row x k-col

To calculate element (i, j) of the resulting matrix, multiply row i in matrix 1 by column j in matrix 2.

$$\vec{a}_i^{\langle row \rangle} = (a_{i1}, \ldots, a_{in}),\ \vec{b}_j^{\langle col \rangle} = (b_{1j}, \ldots, b_{nj})^T$$

Figure 8.9: Get row i from matrix 1 and column j from matrix 2

And take the inner product of two vectors with the same dimension.

$$\vec{a}_i^{\langle row \rangle} \cdot \vec{b}_j^{\langle col \rangle} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj}$$

Figure 8.10: Calculate the inner product of row i and column j

Repeat this for every row in matrix 1 and column in matrix 2. The resulting matrix will have m-rows and k-columns.

- (m-row-n-col) x n-row-k-col → (m-row-k-col).

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} = \begin{bmatrix} \vec{a}_1^{\langle row \rangle} \cdot \vec{b}_1^{\langle col \rangle} & \cdots & \vec{a}_1^{\langle row \rangle} \cdot \vec{b}_k^{\langle col \rangle} \\ \vdots & \ddots & \vdots \\ \vec{a}_m^{\langle row \rangle} \cdot \vec{b}_1^{\langle col \rangle} & \cdots & \vec{a}_m^{\langle row \rangle} \cdot \vec{b}_k^{\langle col \rangle} \end{bmatrix}$$

Figure 8.11: Matrix multiplication

Matrix multiplication and addition are both associate (as with scalars), this means you could multiply the matrices of multiple transformations to form one resultant matrix that can be directly applied on a point.

This is one reason GPUs are optimized for fast matrix multiplications. In computer graphics, we need to apply lots of transformations to out #D object to display it on a 2D monitor. Those transforms are compiled down into one matrix which is applied to all the points in the 3D world.

8.4 3D Transformations

8.4.1 Translation

Translating a 3D vector or matrix is simply an application of matrix addition. To translate point (x, y, z) by (&Deltax, &Deltay, &Deltaz)

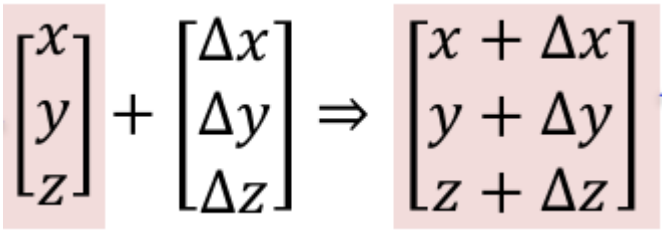

Figure 8.12: 3D Translation

8.4.2 Scaling

Scaling a 3D matrix is an application of matrix multiplication. To scale (x, y, z) by scaling factors (Sx, Sy, Sz), multiply the point vector by the scale matrix, S.

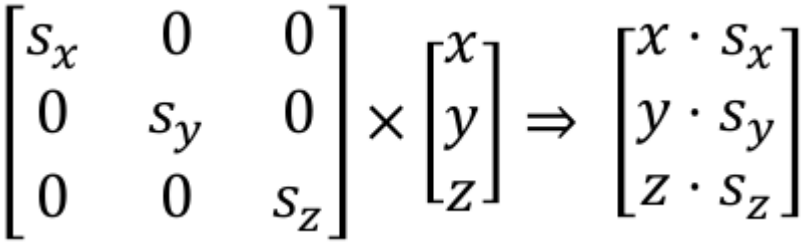

Figure 8.13: 3D Scaling

Prior to scaling the 3D point must be translated so that its center lies on the origin, and then appy the reverse translation after scaling.

8.4.3 Rotation

Rotation is a complicated scenario for 3D transforms. Here, you need an axis around which you rotate the object. Before generalizing the rotation for any axis, let's do it around the x, y, and z-axes. After doing it with one axis, the other two will become fairly easy.

- x-axis: Here imagine the y-z plane is your screen monitor, x-axis rotate anti-clockwise about the angle &theta,keeping x fixed.
- y-axis: Here imagine the x-z plane is your screen monitor, x-axis rotate anti-clockwise about the angle &theta,keeping y fixed.
- z-axis: Here imagine the x-y plane is your screen monitor, x-axis rotate anti-clockwise about the angle &theta,keeping z fixed.

The below three figures summarise rotation about the three axes.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} 1x + 0y + 0z \\ 0x + \cos\theta \cdot y - \sin\theta \cdot z \\ 0x + \sin\theta \cdot y + \cos\theta \cdot z \end{bmatrix}$$

Figure 8.14: 3D Rotation in the y-z plane

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} \cos\theta \cdot x - \sin\theta \cdot z \\ y \\ \sin\theta \cdot x + \cos\theta \cdot z \end{bmatrix}$$

Figure 8.15: 3D Rotation in the x-z plane

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \end{bmatrix}$$

Figure 8.16: 3D Rotation in the x-y plane

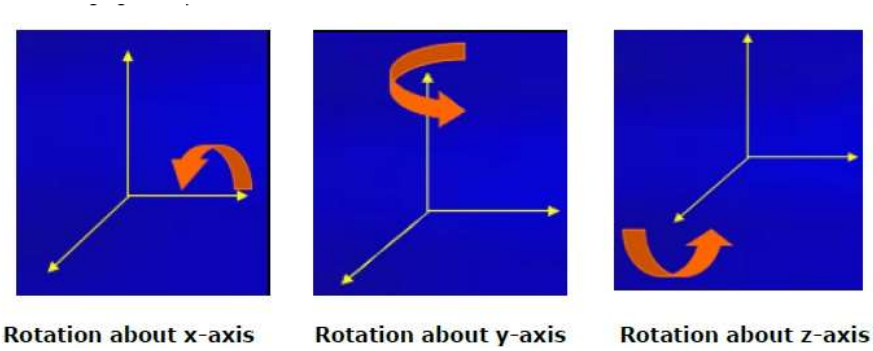

Figure 8.17: 3D Rotation summarised

8.5 Unified Transformation Operator

Each of translation, scaling and rotation involved a different matrix operation.

- Translation: add a vector
- Scaling: multiply by a matrix on the left
- Rotation: multiply by a matrix on the left

Having a unified transformation operator would significantly simplify 3D transformation. This would give a single method/function for all transformation scenario's. To achieve this, translation needs to be redefined as a matrix multiplication (on the left).

In order to achieve this the point (x, y, z), needs an extra dimension.

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{bmatrix}$$

Figure 8.18: Translation of a 3D point by matrix mulitplication. The translation matrix has 1s all along the diagonal the &Deltas are down the final column and all other entries are 0.

The other two transformations can also be redefined with the additional dimension.

- Scaling
- Rotation

8.5.1 Pseudocode

8.6 Composite Transformations

So far we've seen how to do a single transformation (translation, scaling, rotation) using unified the unified transformation operator, but what if we apply multiple transformations?

- translation + scaling or translation + rotation

This type of operation can be achieved with composite matrix multiplication

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \cdot s_x \\ y \cdot s_y \\ z \cdot s_z \\ 1 \end{bmatrix}$$

Figure 8.19: Scaling matrix. The top-left corner is the original 3x3 matrix with 1 as the final diagonal element and all other entries are zero

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z(\theta)$$
$$= \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 8.20: Rotation matrices. The top-left cornet is the original 3x3 matrix with 1 as the final diagonal element and all other entries are zero

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{bmatrix}$$

Figure 8.21: Rotation transformation about the x-axis

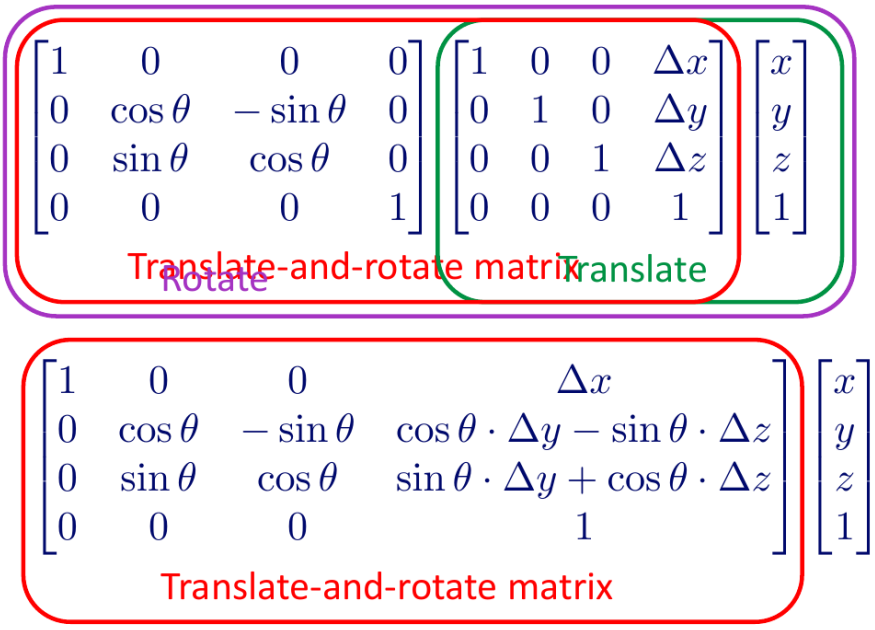

Figure 8.22: Translation and rotation as composite matrix multiplication

Matrix transformations should be applied from right to left i.e. first transformation to the right most matrix and last to the left most matrix

The prioritisation of transformation operations is:

1. Translation
2. Scaling
3. Rotation

8.7 Drawing Polygons

Only the visible surfaces of a polygon should be drawn. How, can you find out which surfaces are visible?

- Use 3D coordinate system + cross product

This method assumes the viewer looks along the z-axis and the polygons vertices are ordered anti-clockwise from the perspective of the viewer.

8.7.1 Cross Product

The cross product is an operation applied to two vectors of the same dimension. It returns a new vector that is perpendicular to the two input vectors

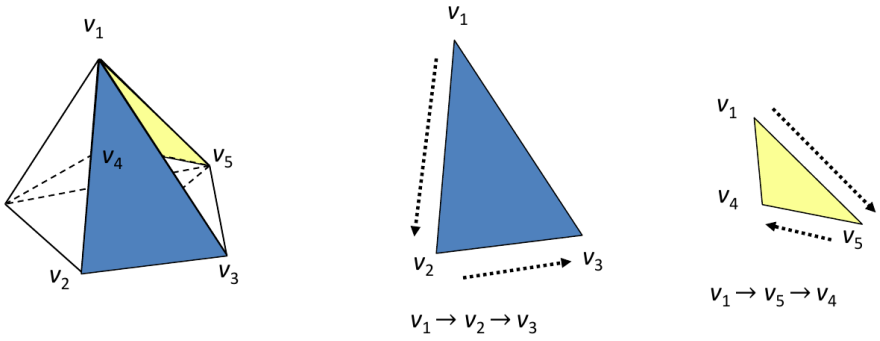

Figure 8.23: The z-axis faces the viewer (blue plane), the vertices are ordered anti-clockwise

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{bmatrix}$$

Figure 8.24: Cross product

The orientation of the three vectors follow the right-hand rule

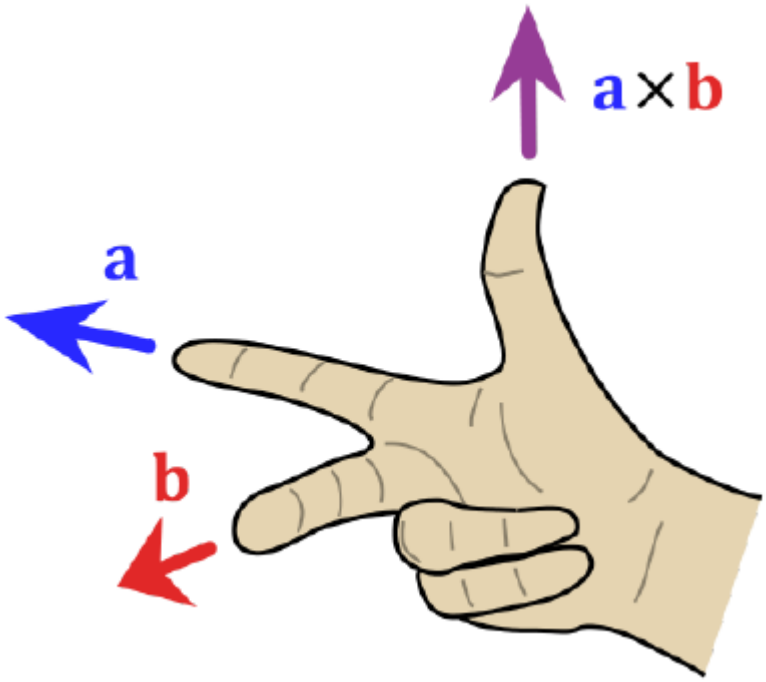

Figure 8.25: Right-hand rule orientation of the three vectors

And the cross product can be used to determine which direction the polygon is facing.

- the vertices v1, v2, v3
- the cross product vector (v1 - v2) x (v3 - v2) faces the viewer

8.7.2 Determining Visible/Invisible Polygons

The cross product (v1 - v2) x (v3 - v2) is called the normal of the polygon. A normal is a line that is perpendicular to the surface of the polygon and the direction of the normal is used to determine whether the polygon surface is visible.

Assuming that the polygon is being viewed along the z-axis:

- A polygon is visible if its normal has negative z-coordinate values
- A polygon is invisible if its normal has non-negative z-coordinate values

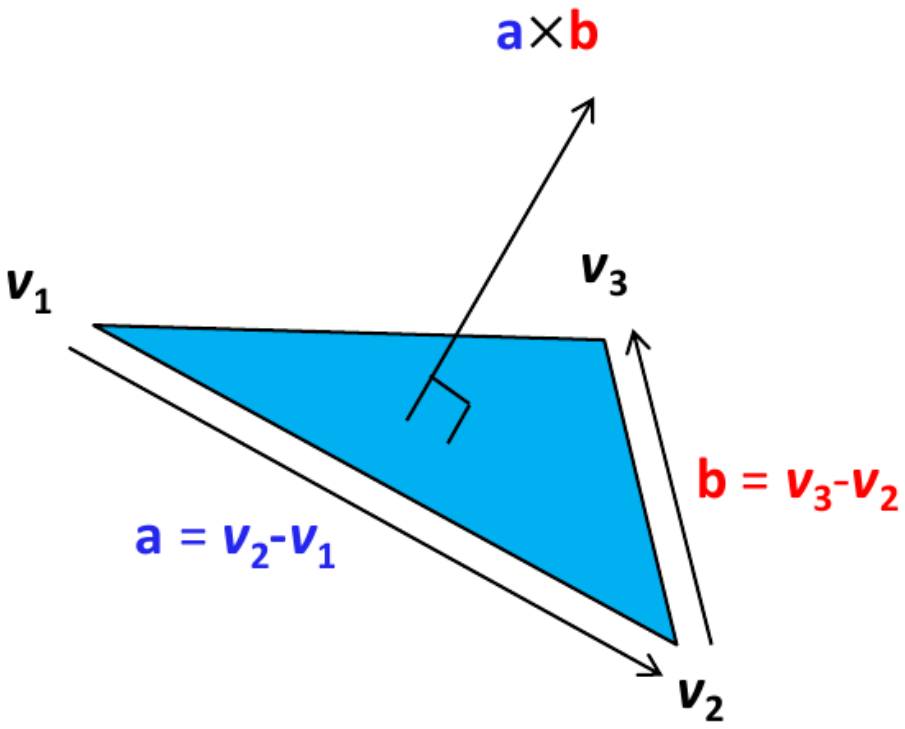

Figure 8.26: Orientation of the cross product relative to the three vertices

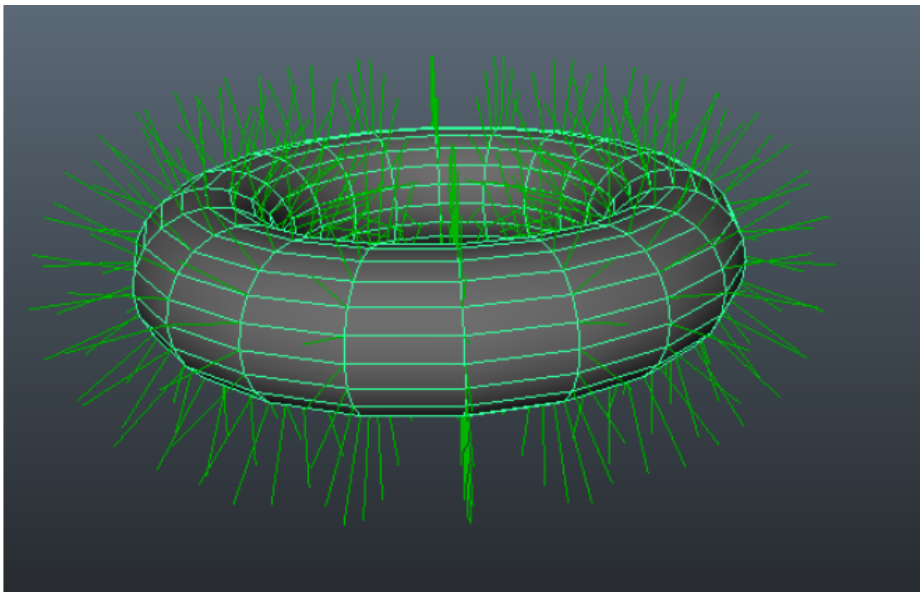

Figure 8.27: The green lines are the normals of the polygon

8.8 Shading

Shading refers to the light that is reflected off of the surfaces of objects. In 3D graphics shading is the process of altering the colour of the surfaces of visible polygons, dependent on

- Direction and colour of light sources
- Reflectance
- Matte/Shiny surface
- Colour, texture of the surface

A simple method to calculate the shading is:

- Assume a matte, uniform reflectance for red, green and blue colours
- Assume some ambient light in the range (0,1]. Ambient light is omnidirectional and all surfaces equally
- Assume an incident light source intensity (0, 1], and its direction
- Diffuse reflection depends on the direction of the incident light source
  - incident light = incident light intensity * reflectance * cos(&theta)
- light = ambient light + incident light

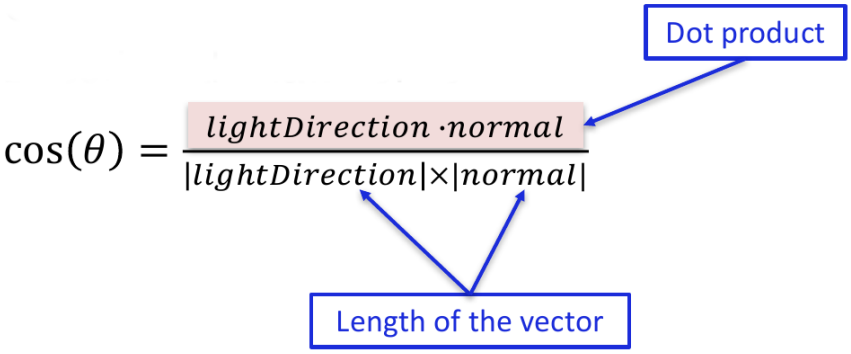

Figure 8.28: The calculation of cos theta is based on the law of cosines

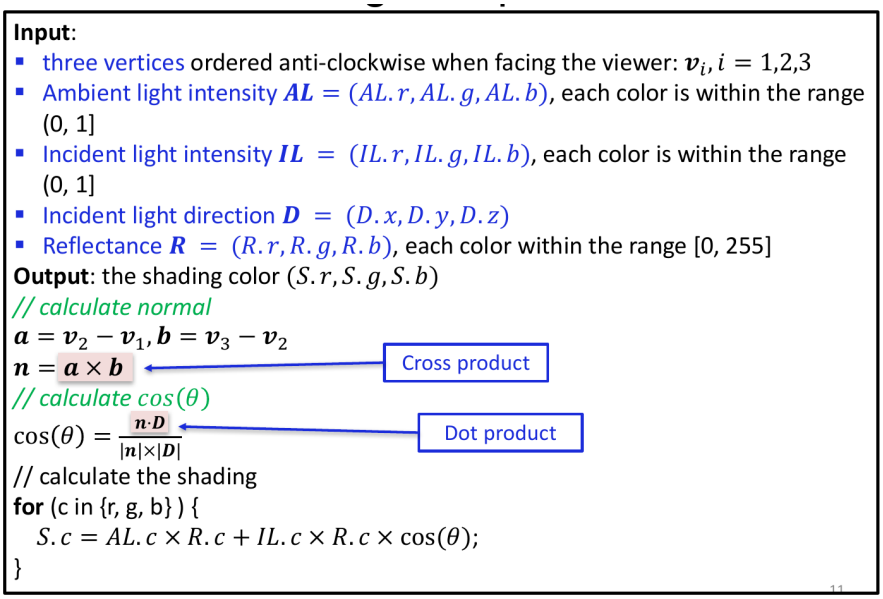

Figure 8.29: Algorithm to calculate the shading of 3D polygons

8.8.1 Algorithm

8.9 Advanced Shading

8.9.1 Flat shading

Here the light reflected from the polygon surface is span style=‘color: lightsea-green;’>uniform, based on the assumption that the polygons surface is flat. The color is computed from the polygon’s surface normal and is used for the whole polygon so it only needs to be calculated once, and makes the corners look sharp and contrast

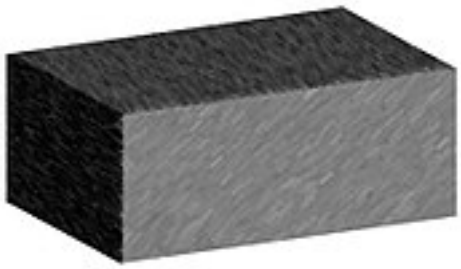

Figure 8.30: Flat shading of a cuboid

8.9.2 Smooth shading

In contrast to flat shading where the colours change discontinuously at polygon borders, with smooth shading the color changes from pixel to pixel (across the surface), resulting in a smooth color transition between two adjacent polygons and the surface approximates a curved surface

this situation the reflected light can be interpolated from the polygon vertices by using “vertex normals” (average of the surface at the vertex): Usually pixel values are first calculated in the vertices and interpolation is used to calculate the values of pixels between the polygon vertices. Types of smooth shading include Gourard shading and Phong shading

Gourard shading

1. Determine the normal at each polygon vertex and calculate the light intensity from the vertex normal
2. Interpolate the light intensities over the surface of the polygon.

Phong shading

Phong shading is similar to Gouraud shading, except that instead of interpolating the light intensities the normals are interpolated between the vertices and the lighting is evaluated per-pixel

1. Determine the normal at each vertex of the polygon
2. Interpolate the normal at each pixel
3. Calculate the light intensity from the computed normals across the surface of the polygon

8.10 Polygon Rendering

To Render a visible polygons you need to compute its shading colour and render the polygon with its shading colour. Polygon rendering involves drawing a 3D polygon on a 2D screen.

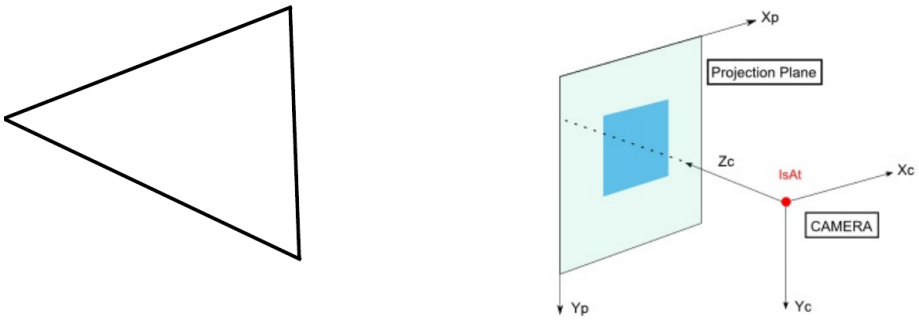

Figure 8.31: Rendering a 3D shape on a 2D screen

To do this, set the z-axis of the polygon as the viewing direction, meaning the display screen is the x-y plane of the polygon. Polygons are rendered line-by-line.

It is reasonably straight forward to obtain the ymin and ymax.

- ymin = v3.y
- ymin = v2.y

How can you get xmin(y) and xmin(y)? For any y-value, the xmin(y) and xmin(y) are on the edges of the polygon, assuming the edges are scanned in an anti-clockwise direction:

- When you are scanning downwards (in the y direction) - xmin(y)
- When you are scanning upwards (in the y direction) - xmax(y)

8.10.1 Linear Interpolation

Given the two end-nodes of an edge (x1, y1) and (x2, y2), what is the x value of the given y value along the edge?

- Along the two edges, y changes from y1 to y2 and x changes from x1 to x2

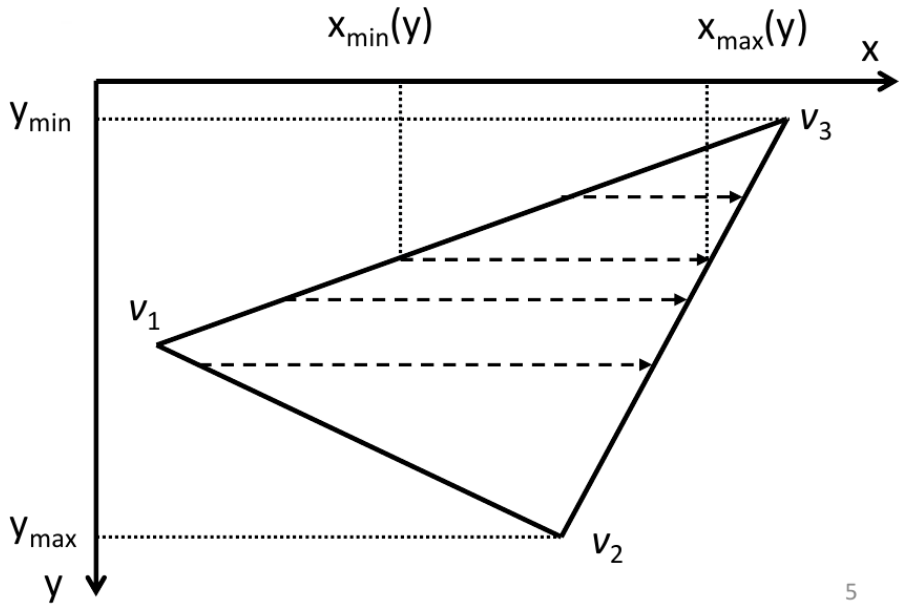

Figure 8.32: Diagramatic Rendering a 3D shape on a 2D screen

- For each unit change of y, x will change x2 - x1 / y2 - y1 (slope)
- x(y) = x1 + slope * (y - y1)

This is repeated to compute the x values for all y values along the edge and the results stored in a list.

Repeating this and scanning the three edges of the polygon will get the two lists xmin(y) and xmax(y)

8.10.2 Edge List

Scan the three edges in the polygon, and update a 2-column `EdgeList` object. This object will store the xmin and xmax for a given y-coordinate.

- If scanning upwards (along the y-axis), then update the xmin(y) column
- If scanning downwards (along the y-axis), then update the xmax(y) column
- Use linear interpolation to calculate the x value along each edge

8.10.3 Multiple polygons

What happens if two polygons intersect i.e. one polygon is in front of the other? In this case only the pixels of the closer polygon need to be rendered (lower z-coordinate), to do this the z-coordinate for each pixel needs to be worked out.

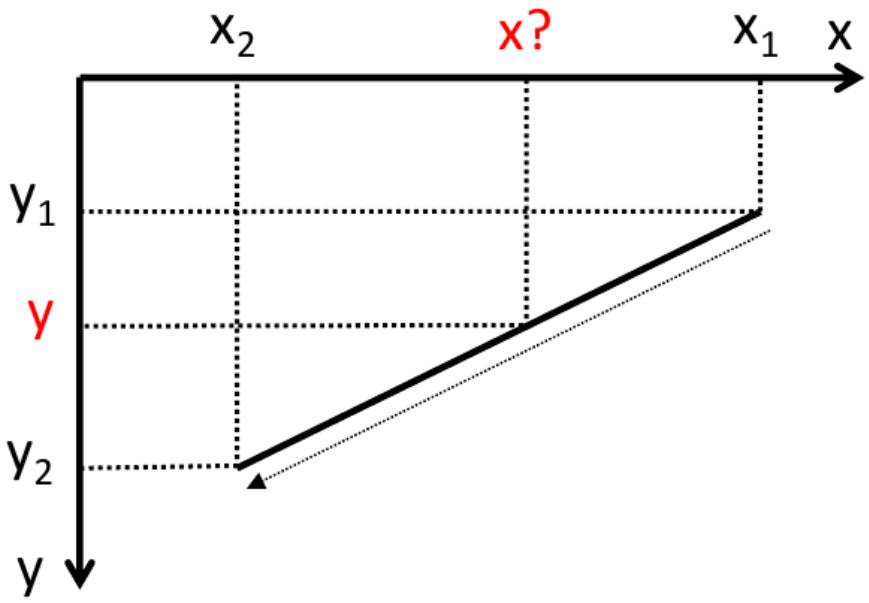

Figure 8.33: Using linear interpolation to compute the x coordinate from two edges and a given y value

The `EdgeList` object must now store additional information, to retrieve the z-coordinate for a given x and y.

- xmin(y) for the polygon edge
- xmax(y) for the polygon edge
- z[x, y] for every pixel

If a pixel occurs on multiple polygons only render the polygon where it has the smallest z value.

8.10.4 Render with Edge List and Z-Buffer

Compute the `EdgeList` for the x and z coordinates for the vertices on the polygon edges

- Compute the z value of a pixel inside the polygon using another linear interpolation

8.10.4.1 Pseudocode

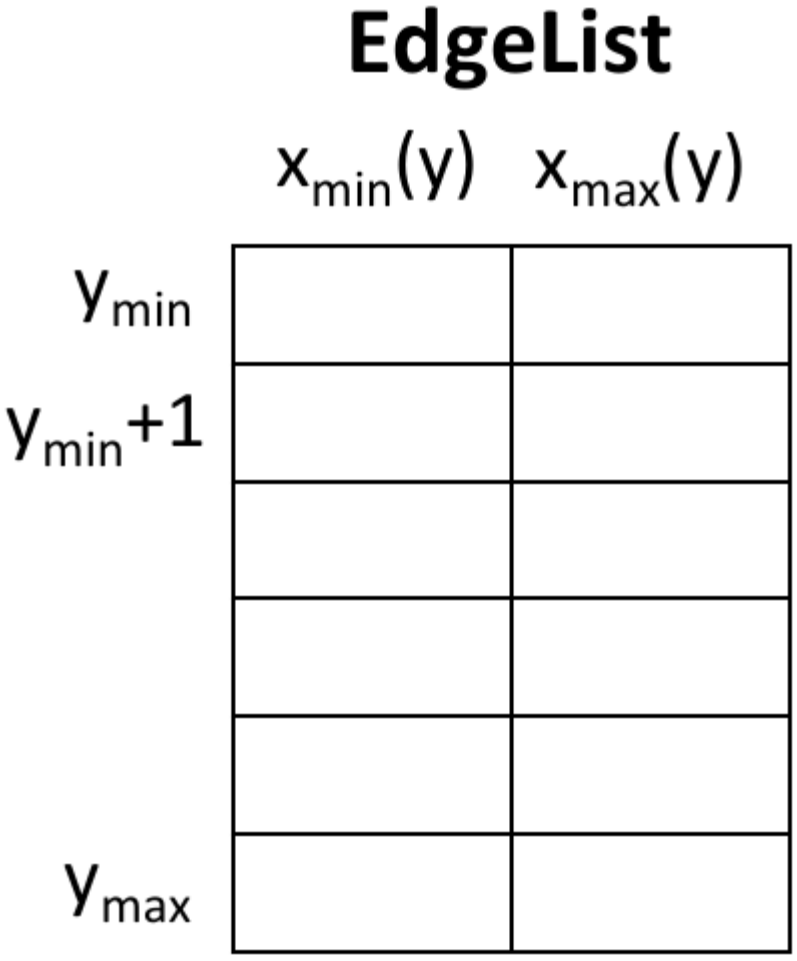

Figure 8.34: Structure of the EdgeList object which holds the min and max x-values for a given y value

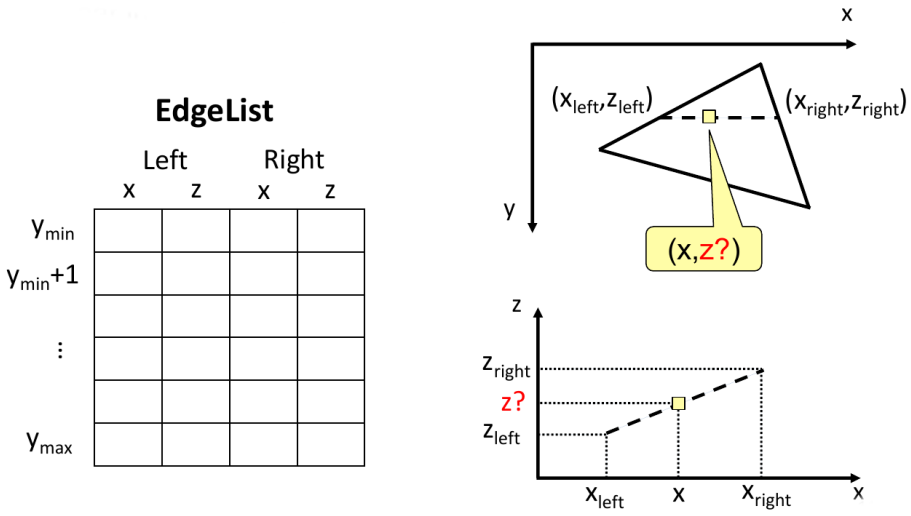

Figure 8.35: Interpolate the z coordinate for each pixel using linear interpolation

Chapter 9

Parsing

9.1 Structured Text

What makes the following texts structured?

- SQL schema query

```
DELETE FROM DomesticStudents2018
WHERE grade = 'E'
```

- Java statement

```
while (A[i] != x) {
  k++;
}
```

- XML documents

```
<html><head><title>My Web Page</title></head>
<body><p> Thanks for viewing </p></body></html>
```

Text is structured if it can be described using a grammar. A grammar consists of a set of rules:

- The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax
- However, a grammar does not describe the meaning of the strings or what can be done with them in whatever context - only their form.

9.2 Defining a Grammar

A grammar is a finite description of a possible infinite set of acceptable sentences. One of the simplest existing grammars are regular expressions (regex).