Tables describing 3D and 4D prisms

The tables on this page list essential relationships in 3D and 4D prisms and facilitate the process of drawing and painting with computer code. What they mean and how they are derived will be explained. It is possible to make a program that finds these connections on its own. But the shortest and easiest way is this quite generic approach of using tables (arrays) with constants in them.

3D corner rays
0, [1, 2, 4]
1, [0, 3, 5]
2, [0, 3, 6]
3, [1, 2, 7]
4, [0, 5, 6]
5, [1, 4, 7]
6, [2, 4, 7]
7, [3, 5, 6]

3D corner rings
0, [0, 1, 3, 2]
1, [4, 5, 7, 6]
2, [0, 1, 5, 4]
3, [2, 3, 7, 6]
4, [0, 2, 6, 4]
5, [1, 3, 7, 5]

4D corner rays
0, [1, 2, 4, 8]
1, [0, 3, 5, 9]
2, [0, 3, 6, 10]
3, [1, 2, 7, 11]
4, [0, 5, 6, 12]
5, [1, 4, 7, 13]
6, [2, 4, 7, 14]
7, [3, 5, 6, 15]
8, [0, 9, 10, 12]
9, [1, 8, 11, 13]
10, [2, 8, 11, 14]
11, [3, 9, 10, 15]
12, [4, 8, 13, 14]
13, [5, 9, 12, 15]
14, [6, 10, 12, 15]
15, [7, 11, 13, 14]

4D corner rings
0, [0, 1, 3, 2]
1, [4, 5, 7, 6]
2, [8, 9, 11, 10]
3, [12, 13, 15, 14]
4, [0, 1, 5, 4]
5, [2, 3, 7, 6]
6, [8, 9, 13, 12]
7, [10, 11, 15, 14]
8, [0, 2, 6, 4]
9, [1, 3, 7, 5]
10, [8, 10, 14, 12]
11, [9, 11, 15, 13]
12, [0, 1, 9, 8]
13, [2, 3, 11, 10]
14, [4, 5, 13, 12]
15, [6, 7, 15, 14]
16, [0, 2, 10, 8]
17, [1, 3, 11, 9]
18, [4, 6, 14, 12]
19, [5, 7, 15, 13]
20, [0, 4, 12, 8]
21, [1, 5, 13, 9]
22, [2, 6, 14, 10]
23, [3, 7, 15, 11]

4D face clusters
0, [0, 1, 4, 5, 8, 9]
1, [2, 3, 6, 7, 10, 11]
2, [0, 2, 12, 13, 16, 17]
3, [1, 3, 14, 15, 18, 19]
4, [4, 6, 12, 14, 20, 21]
5, [5, 7, 13, 15, 22, 23]
6, [8, 10, 16, 18, 20, 22]
7, [9, 11, 17, 19, 21, 23]

Legend: “Corner rays” are one-to-many relationships from corner to other corners. “Corner rings” are one-to-many relationships from face (surface) to corners, where sequence matters because the corners describe a ring. “Face clusters” are one-to-many relationships from room to faces (surfaces), only used in 4D where 3D rooms represent a level between 4D prism and 2D faces.

Table of n-dimensional prisms, from reduced prisms (point, line, rectangle) via normal rectangular prism to extended prisms (4D, 5D, 6D)

Dim	Points	Lines	Faces	Rooms	4D	5D	6D
0	1	0	0	0	0	0	0
1	2	1	0	0	0	0	0
2	4	4	1	0	0	0	0
3	8	12	6	1	0	0	0
4	16	32	24	8	1	0	0
5	32	80	80	40	10	1	0
6	64	192	240	160	60	12	1

Extrusion of .. into .: Point → Line segment → Rectangle → Prism → Prism 4D → Prism 5D → Prism 6D

Calculating the bordering elements of 1D - 4D prisms

Corners (points) **2^Dim**
Edges (lines) **2^Dim / 2 * Dim**
Surfaces (rectangles) **2^Dim / 8 * (Dim - 1) * Dim**
Surface rooms (prisms) **2^Dim / 16 * (0 * 1 + 1 * 2 + 2 * 3 + .. + (Dim - 2) * (Dim - 1))**

Each dimension includes one body of prism character (with straight lines and right angles).
Each body is extruded from the closest lower dimension.
The table shows how many points, lines, faces etc. you must include in a projection of such a body.
For instance a glass die (in 3D) must be drawn (in 2D) with 6 faces, 12 edges and 8 corners.
At least.. the table and formulas show what I found out and what the simulation builds on.

Rectangle like objects up to 5 dimensions

Perhaps I should use words like “vertex” and “vertices”, but I prefer “corner” and “corners”. The words “face” and “side” often mean the same (not always). A face is always a 2D surface.

Moving a point or a line or a rectangle in a direction that is not within the realm of the point, line or rectangle, creates a new dimension. Literally an action is transformed into an object. So a point creates a line segment, a line segment creates a rectangle, and a rectangle creates a rectangular prism.

The lower dimensions remain. The point that turned into a line segment, remains as a starting point and is duplicated at the end. These are the end points. The line segment that turned into a rectangle, remains as an edge and is duplicated on the other side as another egde. Not only the line segment undergoes this change. The end points of the line segment are moved up one dimension to line segments, and each point is duplicated as new end points. So I must add moved points (extruded points) to the duplicated line segments. And I must add moved line segments (extruded line segments) to duplicated rectangles. Here are the rules..

One dimension
A point is moved in one direction. The result is 1 line segment with 2 end points.

Two dimensions
A line segment is moved to the side at a right angle to itself. It results in
- 1 rectangle (face),
- 2 line segments (edges) from duplication,
- 2 line segments (edges) from extrusion, which is the movement of end points (corners),
- 4 end points (corners) from duplication.

The final result is 1 rectangle (face), 4 line segments (edges) and 4 end points (corners).

Three dimensions
A rectangle is moved at a right angle to itself. It results in
- 1 prism,
- 2 rectangles (faces) from duplication,
- 4 rectangles (faces) from the movement of line segments (edges),
- 8 line segments (edges) from duplication,
- 4 line segments (edges) from extrusion, which is the movement of end points (corners),
- 8 end points (corners) from duplication.

The final result is 1 prism, 6 rectangles (faces), 12 line segments (edges) and 8 end points (corners).

Four dimensions
A prism is moved at a right angle to itself (requires an out of this world axis). This results in
- 1 prism in 4D (which is very different from a prism in 3D),
- 2 prisms in 3D (I call them siderooms) from duplication,
- 6 prisms in 3D (siderooms) from the movement of rectangles (faces),
- 12 rectangles (faces) from duplication,
- 12 rectangles (faces) from the movement of line segments (edges),
- 24 line segments (edges) from duplication,
- 8 line segments (edges) from the movement of end points (corners),
- 16 end points (corners) from duplication.

The final result is 1 prism in 4D, 8 prisms in 3D (siderooms), 24 rectangles (faces), 32 line segments (edges) and 16 end points (corners).

Five dimensions
A prism in 4D is moved at a right angle to itself (requires another out of this world axis). It results in
- 1 prism in 5D,
- 2 prisms in 4D (siderooms in 4D) from duplication,
- 8 prisms in 4D (siderooms in 4D) from the movement of prisms in 3D (siderooms),
- 16 prisms in 3D (siderooms) from duplication,
- 24 prisms in 3D (siderooms) from the movement of rectangles (faces),
- 48 rectangles (faces) from duplication,
- 32 rectangles (faces) from the movement of line segments (edges),
- 64 line segments (edges) from duplication,
- 16 line segments (edges) from the movement of points (corners),

- 32 end points (corners) from duplication.

The final result is 1 prism in 5D, 10 prisms in 4D (siderooms in 4D), 40 prisms in 3D (siderooms), 80 rectangles (faces), 80 line segments (edges) and 32 end points (corners).

Formulas

There is a more direct way with the rules I found. The syntax I use, could be misinterpreted. Division and multiplication have same precedence and should be performed in sequence from left to right if there are no encapsulating parenthesis. Also the “hat” symbol means exponentiation.

Corners = 2^Dim
Edges = 2^Dim / 2 * Dim
Faces = 2^Dim / 8 * (Dim - 1) * Dim
Rooms = 2^Dim / 16 * (0 * 1 + 1 * 2 + 2 * 3 + .. + (Dim - 2) * (Dim - 1))

The last formula contains a series that could be converted to something simpler? I’m not a mathematician, so I really don’t know. Also be critical. This is what I found out. But when I visualize it in an app, it looks correct.

Corners for a line (end points) = 2^1 = 2
Corners in a rectangle = 2^2 = 4
Corners in a prism = 2^3 = 8
Corners in a prism in 4D = 2^4 = 16
Corners in a prism in 5D = 2^5 = 32

Edges in a rectangle = 2^2 / 2 * 2 = 4 / 2 * 2 = 2 * 2 = 4
Edges in a prism = 2^3 / 2 * 3 = 8 / 2 * 3 = 4 * 3 = 12
Edges in a prism 4D = 2^4 / 2 * 4 = 16 / 2 * 4 = 8 * 4 = 32
Edges in a prism 5D = 2^5 / 2 * 5 = 32 / 2 * 5 = 16 * 5 = 80

Faces in a prism = 2^3 / 8 * (3 - 1) * 3 = 8 / 8 * 2 * 3 = 1 * 2 * 3 = 6
Faces in a prism in 4D = 2^4 / 8 * (4 - 1) * 4 = 16 / 8 * 3 * 4 = 2 * 3 * 4 = 24
Faces in a prism in 5D = 2^5 / 8 * (5 - 1) * 5 = 32 / 8 * 4 * 5 = 4 * 4 * 5 = 80

Rooms in a prism in 4D = 2^4 / 16 * (0 * 1 + 1 * 2 + 2 * 3) = 16 / 16 * (0 + 2 + 6) = 1 * 8 = 8
Rooms in a prism in 5D = 2^5 / 16 * (0 * 1 + 1 * 2 + 2 * 3 + 3 * 4) = 32 / 16 * (0 + 2 + 6 + 12) = 2 * 20 = 40

Create a numbering system for corners

Imagine 3 axis where x and y is on a flat desk and z is up from the desk. Axis x points to the right and y points towards the backwall. Now I can place a prism or a cube so that all dimensions are positive. There will be a corner in origo (0, 0, 0), a corner on the x axis, a corner on the y axis and one on the z axis. I name these corners “origo”, “x”, “y” and “z”. There are 4 more corners where parallel and perpendicular edges meet. All of these are combinations of x-, y- and z-values. Two edges meet in the “xy” corner, two meet in “xz”, and two meet in “yz”. Only one corner is completely apart from the axis (plural) and the planes, “xyz”. It needs a coordinate different from zero for all the dimensions x, y and z.

Therefore I name the corners “origo”, “x”, “y”, “xy”, “z”, “xz”, “yz”, “xyz”. The sequence matters, where z is considered more significant, y less significant and x least significant, with only “origo” as a smaller value. In a table setup there are 3 columns “zyx” in that order where the number 1 indicates if x, y or z is used (has a value that differs from zero). If not, there is a 0 here. This way “origo” is assigned 000, and “xyz” is assigned 111. The corners are already sorted in the right order and can be translated to: “origo” = 000, “x” = 001,“y” = 010, “xy” = 011, “z” = 100, “xz” = 101, “yz” = 110, “xyz” = 111.

When a number has only 0 or 1 in its digits, it is called a binary number, which can be converted to an ordinary number. The first digit in a 3-digit binary number is the the four-position (how many fours?), then comes the two-position (how many twos?), then comes the one-position (how many ones?).

Now I can list the corners over again as: “origo” = 000 = 0, “x” = 001 = 1, “y” = 010 = 2, “xy” = 011 = 3, “z” = 100 = 4, “xz” = 101 = 5, “yz” = 110 = 6, “xyz” = 111 = 7. This works as an aid to find the edges because an edge is always parallel to one axis and perpendicular to the other axis. When a line is drawn at a right angle from an axis, the value on this axis won’t change. Therefore two of the values won’t change for an edge in a prism that is aligned with the axis. When the line is parallel to an axis, this value will change. Two bits in the binary number will therefore remain constant while only one bit changes. Example: In the 011 corner there are 3 edges that meet, and they come from the 001, 010 and 111 corners. This is because 001 is only one bit away from 011, meaning only the two-position digit has changed. Also 010 is one bit away from 011 since only the one-position digit has changed. And finally 111 is one bit away from 011 since only the four-position digit has changed. They are therefore all related or neighbours via the 011 corner.

In other words I can say that corner 011 has 3 neighbours 001, 010 and 111. With ordinary numbers this means: Corner 3 has neighbour corners 1, 2 and 7. It is best visualized by saying that corner “xy” has neighbour corners “x”, “y” and “xyz”. In the case of 3 dimensions it is easy to see this right away, and I don’t need the maticulous numbering. But if I try to visualize the same thing in 4 or 5 dimensions, it is troublesome. When I cannot visualize it, I must follow the rules instead.

Prism 3d, corner table

Corner 0 or 000 or origo is connected to [001, 010, 100] or [1, 2, 4] or [x, y, z].
Corner 1 or 001 or x is connected to [000, 011, 101] or [0, 3, 5] or [origo, xy, xz].
Corner 2 or 010 or y is connected to [000, 011, 110] or [0, 3, 6] or [origo, xy, yz].
Corner 3 or 011 or xy is connected to [001, 010, 111] or [1, 2, 7] or [x, y, xyz].
Corner 4 or 100 or z is connected to [000, 101, 110] or [0, 5, 6] or [origo, xz, yz].
Corner 5 or 101 or xz is connected to [001, 100, 111] or [1, 4, 7] or [x, z, xyz].
Corner 6 or 110 or yz is connected to [010, 100, 111] or [2, 4, 7] or [y, z, xyz].
Corner 7 or 111 or xyz is connected to [011, 101, 110] or [3, 5, 6] or [xy, xz, yz].

Corner	Binary	z	y	x	Name	To binary corners	To corners	To names
0	000				origo	001, 010, 100	1, 2, 4	x, y, z
1	001			+	x	000, 011, 101	0, 3, 5	origo, xy, xz
2	010		+		y	000, 011, 110	0, 3, 6	origo, xy, yz
3	011		+	+	xy	001, 010, 111	1, 2, 7	x, y, xyz
4	100	+			z	000, 101, 110	0, 5, 6	origo, xz, yz
5	101	+		+	xz	001, 100, 111	1, 4, 7	x, z, xyz
6	110	+	+		yz	010, 100, 111	2, 4, 7	y, z, xyz
7	111	+	+	+	xyz	011, 101, 110	3, 5, 6	xy, xz, yz

Extend this to 4d

I need a fourth axis for 4 dimensions, and I name it q. It is perpendicular to all other axis. The 4D prism is laid out the same way as the 3D prism, but now “origo” and “xyzq” are the basic opposite corners. Altogether there are 16 corners here with names like “origo”, “y”, “yq”, “xzq” and “xyzq”. The letter indicates that there is a value (coordinate) different from zero for either x, y, z or q, while the object is still in the start position (before turning it away from the axis). The binary numbering is four digits (0000, 1011 etc.) The first digit is the eights-position (how many eights?). The range is 0 - 15 in ordinary numbers. With the same reasoning as in 3D a 4D table can be set up. This time every corner is connected to 4 other corners, and again to the corners that are only one bit (digit) away from the current corner. Based on this the rules deduct themselves without having to visualize anything.

Prism 4d, corner table

Corner 0 or 0000 or origo connects to [0001, 0010, 0100, 1000] or [1, 2, 4, 8] or [x, y, z, q].
Corner 1 or 0001 or x connects to [0000, 0011, 0101, 1001] or [0, 3, 5, 9] or [origo, xy, xz, xq].
Corner 2 or 0010 or y connects to [0000, 0011, 0110, 1010] or [0, 3, 6, 10] or [origo, xy, yz, yq].
Corner 3 or 0011 or xy connects to [0001, 0010, 0111, 1011] or [1, 2, 7, 11] or [x, y, xyz, xyq].
Corner 4 or 0100 or z connects to [0000, 0101, 0110, 1100] or [0, 5, 6, 12] or [origo, xz, yz, zq].
Corner 5 or 0101 or xz connects to [0001, 0100, 0111, 1101] or [1, 4, 7, 13] or [x, z, xyz, xzq].
Corner 6 or 0110 or yz connects to [0010, 0100, 0111, 1110] or [2, 4, 7, 14] or [y, z, xyz, yzq].
Corner 7 or 0111 or xyz connects to [0011, 0101, 0110, 1111] or [3, 5, 6, 15] or [xy, xz, yz, xyzq].
Corner 8 or 1000 or q connects to [0000, 1001, 1010, 1100] or [0, 9, 10, 12] or [origo, xq, yq, zq].
Corner 9 or 1001 or xq connects to [0001, 1000, 1011, 1101] or [1, 8, 11, 13] or [x, q, xyq, xzq].
Corner 10 or 1010 or yq connects to [0010, 1000, 1011, 1110] or [2, 8, 11, 14] or [y, q, xyq, yzq].
Corner 11 or 1011 or xyq connects to [0011, 1001, 1010, 1111] or [3, 9, 10, 15] or [xy, xq, yq, xyzq].
Corner 12 or 1100 or zq connects to [0100, 1000, 1101, 1110] or [4, 8, 13, 14] or [z, q, xzq, yzq].
Corner 13 or 1101 or xzq connects to [0101, 1001, 1100, 1111] or [5, 9, 12, 15] or [xz, xq, zq, xyzq].
Corner 14 or 1110 or yzq connects to [0110, 1010, 1100, 1111] or [6, 10, 12, 15] or [yz, yq, zq, xyzq].
Corner 15 or 1111 or xyzq connects to [0111, 1011, 1101, 1110] or [7, 11, 13, 14] or [xyz, xyq, xzq, yzq].

Corner	Binary	q	z	y	x	Name	To binary corners	To corners	To names
0	0000					origo	0001, 0010, 0100, 1000	1, 2, 4, 8	x, y, z, q
1	0001				+	x	0000, 0011, 0101, 1001	0, 3, 5, 9	origo, xy, xz, xq
2	0010			+		y	0000, 0011, 0110, 1010	0, 3, 6, 10	origo, xy, yz, yq
3	0011			+	+	xy	0001, 0010, 0111, 1011	1, 2, 7, 11	x, y, xyz, xyq
4	0100		+			z	0000, 0101, 0110, 1100	0, 5, 6, 12	origo, xz, yz, zq
5	0101		+		+	xz	0001, 0100, 0111, 1101	1, 4, 7, 13	x, z, xyz, xzq
6	0110		+	+		yz	0010, 0100, 0111, 1110	2, 4, 7, 14	y, z, xyz, yzq
7	0111		+	+	+	xyz	0011, 0101, 0110, 1111	3, 5, 6, 15	xy, xz, yz, xyzq
8	1000	+				q	0000, 1001, 1010, 1100	0, 9, 10, 12	origo, xq, yq, zq
9	1001	+			+	xq	0001, 1000, 1011, 1101	1, 8, 11, 13	x, q, xyq, xzq
10	1010	+		+		yq	0010, 1000, 1011, 1110	2, 8, 11, 14	y, q, xyq, yzq
11	1011	+		+	+	xyq	0011, 1001, 1010, 1111	3, 9, 10, 15	xy, xq, yq, xyzq
12	1100	+	+			zq	0100, 1000, 1101, 1110	4, 8, 13, 14	z, q, xzq, yzq
13	1101	+	+		+	xzq	0101, 1001, 1100, 1111	5, 9, 12, 15	xz, xq, zq, xyzq
14	1110	+	+	+		yzq	0110, 1010, 1100, 1111	6, 10, 12, 15	yz, yq, zq, xyzq
15	1111	+	+	+	+	xyzq	0111, 1011, 1101, 1110	7, 11, 13, 14	xyz, xyq, xzq, yzq

Prism 3d, face table

The 6 sides or faces of a prism are rectangles and have 4 corners each. Every corner is shared by 3 faces. Every face is paired with a parallel face on the other side of the prism. There are two rules for the corners of a face on a prism that is still in its start position (meaning the edges are parallel to one of the axis). I refer to the corners in their binary representation..

Rule 1: The edges of a face are all in the same plane and need only 2 coordinates for the corners. The third coordinate is constant. This means that the corners have a common bit-position (digit-position), one that stays on 0 or 1 and does not change.

Rule 2: Cycling the corners of a face the corner number will change one bit (digit) at a time. There will never be 2 bits changing from one corner to the next corner. Since the corner sequence is a ring or a cycle without start or end, this also applies after the fourth corner in the sequence. Only one bit is allowed to change to get back to the first corner. This rule sets a unique sequence for the corners based on their numbering only. There is no need to visualize. It is true that there is always an alternative next corner number, but that only means they could be cycled in the opposite direction. Or it means repeating a corner that is already taken.

I have sorted the faces by their natural names (the same principle as with corners): xy1, xy2, xz1, xz2, yz1, yz2. A sequence number is assigned in the same order and starts on 0. Note that the corners in the ring are not sorted by number. They are sorted by their geometrical sequence (important).

Face 0 or xy1 has corners [origo, x, xy, y] or [000, 001, 011, 010] or [0, 1, 3, 2].
Face 1 or xy2 has corners [z, xz, xyz, yz] or [100, 101, 111, 110] or [4, 5, 7, 6].
Face 2 or xz1 has corners [origo, x, xz, z] or [000, 001, 101, 100] or [0, 1, 5, 4].
Face 3 or xz2 has corners [y, xy, xyz, yz] or [010, 011, 111, 110] or [2, 3, 7, 6].
Face 4 or yz1 has corners [origo, y, yz, z] or [000, 010, 110, 100] or [0, 2, 6, 4].
Face 5 or yz2 has corners [x, xy, xyz, xz] or [001, 011, 111, 101] or [1, 3, 7, 5].

Face	Name	Ringed by corner names	Ringed by binary corners	Ringed by corners
0	xy1	origo, x, xy, y	000, 001, 011, 010	0, 1, 3, 2
1	xy2	z, xz, xyz, yz	100, 101, 111, 110	4, 5, 7, 6
2	xz1	origo, x, xz, z	000, 001, 101, 100	0, 1, 5, 4
3	xz2	y, xy, xyz, yz	010, 011, 111, 110	2, 3, 7, 6
4	yz1	origo, y, yz, z	000, 010, 110, 100	0, 2, 6, 4
5	yz2	x, xy, xyz, xz	001, 011, 111, 101	1, 3, 7, 5

Finally I can collect the faces to surround a prism like this..

Room 0 or xyz has faces [xy1, xy2, xz1, xz2, yz1, yz2] or [0, 1, 2, 3, 4, 5].
Room 0 or xyz has corners [0, 1, 2, 3, 4, 5, 6, 7].

Room	Name	Encapsulating faces (names)	Faces	Corners
0	xyz	xy1, xy2, xz1, xz2, yz1, yz2	0, 1, 2, 3, 4, 5	0, 1, 2, 3, 4, 5, 6, 7

Prism 4d, faces

I found there are 24 rectangular surfaces on the ordinary 3D prisms surrounding the 4D prism. There are no faces directly on a 4D prism since it needs 3D objects to wrap / surround / cover the 4D object. The 8 siderooms resemble the 6 sides / faces on a 3D prism. Every sideroom needs 6 faces, a total of 48 faces. Since there are only 24 faces available, each face must be shared by 2 prisms. This means that the ordinary prisms are glued together with a common surface. The entanglement between 8 prisms would be impossible if it weren't for the 4 dimensions.

There are 6 orientations for a normalized plane in a 4-dimensional space: xy, xz, yz, xq, yq, zq. This means that 4 planes of each orientation are needed since I should end up with 24 of them and since they are evenly distributed between orientations (since there is no reason why not). I name them: xy1, xy2, xy3, xy4, xz1, xz2, xz3, xz4, yz1, yz2, yz3, yz4, xq1, xq2, xq3, xq4, yq1, yq2, yq3, yq4, zq1, zq2, zq3, zq4.

Same orientation does not mean the same plane. It could be parallel planes. The explanation for the 4 planes with equal orientation is that you need 2 faces on an ordinary prism with the same orientation. They are opposite sides. On a prism in 4D there are 2 ordinary prisms with the same orientation, for example with yzq orientation. They are opposite siderooms. Each of the 2 siderooms has 2 opposite faces with the same orientation, hence 4 faces with the same orientation.

Prism 4d, face table

The binary number representations of the corners outline the whole story and is organized like this: qzyx. The last axis q has highest significance. To cycle the corners in an xy-plane I must keep the bits (digits) constant for both the z- and q-axis, like this: [0000, 0001, 0010 and 0011]. The first 00 tells me there is no value for z and q, and I am on the ground level, and I am within the xy1 face.

In the above example the corners are origo, x (1 in the x-bit), y (1 in the y-bit) and xy (1 in both x and y). If I follow this path, first an edge is drawn from origo along the x-axis, then a diagonal is drawn over to the y-axis, then an edge is drawn to the most distant corner, then a diagonal is drawn back to origo since it is a ring and I must finish the cycle.

This is not what I want. When both x and y change at the same time, it is not an edge. The only way I can cycle the corners to get edges, is doing it more gently: [0000, 0001, 0011, 0010] or [0000, 0010, 0011, 0001] (opposite direction) or [0011, 0010, 0000, 0001] (another starting point). These are all examples of the same ring (the exact same ring). The line is drawn from origo along the x-axis to the x coordinate, then to the most distant corner, then back to the y-axis, then along the y-axis back to origo. Changing one coordinate at a time assures that the lines are straight (along one of the axis or parallel to an axis).

Keeping z and q constant means the cycle is level and never moves out of the xy-plane. I can vary z and q like this: qz = 00, qz = 01, qz = 10, qz = 11. The same xy cycle is repeated within 4 different planes. The face that is furthest away on both the z-axis and the q-axis has these corners: [1100, 1101, 1111, 1110]. The last 2 bits are the same in all cycles (on all 4 faces). Expressed in a table..

Face 0 or xy1 has corners [origo, x, xy, y] or [0000, 0001, 0011, 0010] or [0, 1, 3, 2].
Face 1 or xy2 has corners [z, xz, xyz, yz] or [0100, 0101, 0111, 0110] or [4, 5, 7, 6].
Face 2 or xy3 has corners [q, xq, xyq, yq] or [1000, 1001, 1011, 1010] or [8, 9, 11, 10].

Face 3 or xy4 has corners [zq, xzq, xyzq, yzq] or [1100, 1101, 1111, 1110] or [12, 13, 15, 14].

Face 4 or xz1 has corners [origo, x, xz, z] or [0000, 0001, 0101, 0100] or [0, 1, 5, 4].
Face 5 or xz2 has corners [y, xy, xyz, yz] or [0010, 0011, 0111, 0110] or [2, 3, 7, 6].
Face 6 or xz3 has corners [q, xq, xzq, zq] or [1000, 1001, 1101, 1100] or [8, 9, 13, 12].
Face 7 or xz4 has corners [yq, xyq, xyzq, yzq] or [1010, 1011, 1111, 1110] or [10, 11, 15, 14].

Face 8 or yz1 has corners [origo, y, yz, z] or [0000, 0010, 0110, 0100] or [0, 2, 6, 4].
Face 9 or yz2 has corners [x, xy, xyz, xz] or [0001, 0011, 0111, 0101] or [1, 3, 7, 5].
Face 10 or yz3 has corners [q, yq, yzq, zq] or [1000, 1010, 1110, 1100] or [8, 10, 14, 12].
Face 11 or yz4 has corners [xq, xyq, xyzq, xzq] or [1001, 1011, 1111, 1101] or [9, 11, 15, 13].

Face 12 or xq1 has corners [origo, x, xq, q] or [0000, 0001, 1001, 1000] or [0, 1, 9, 8].
Face 13 or xq2 has corners [y, xy, xyq, yq] or [0010, 0011, 1011, 1010] or [2, 3, 11, 10].
Face 14 or xq3 has corners [z, xz, xzq, zq] or [0100, 0101, 1101, 1100] or [4, 5, 13, 12].
Face 15 or xq4 has corners [yz, xyz, xyzq, yzq] or [0110, 0111, 1111, 1110] or [6, 7, 15, 14].

Face 16 or yq1 has corners [origo, y, yq, q] or [0000, 0010, 1010, 1000] or [0, 2, 10, 8].
Face 17 or yq2 has corners [x, xy, xyq, xq] or [0001, 0011, 1011, 1001] or [1, 3, 11, 9].
Face 18 or yq3 has corners [z, yz, yzq, zq] or [0100, 0110, 1110, 1100] or [4, 6, 14, 12].
Face 19 or yq4 has corners [xz, xyz, xyzq, xzq] or [0101, 0111, 1111, 1101] or [5, 7, 15, 13].

Face 20 or zq1 has corners [origo, z, zq, q] or [0000, 0100, 1100, 1000] or [0, 4, 12, 8].
Face 21 or zq2 has corners [x, xz, xzq, xq] or [0001, 0101, 1101, 1001] or [1, 5, 13, 9].
Face 22 or zq3 has corners [y, yz, yzq, yq] or [0010, 0110, 1110, 1010] or [2, 6, 14, 10].
Face 23 or zq4 has corners [xy, xyz, xyzq, xyq] or [0011, 0111, 1111, 1011] or [3, 7, 15, 11].

Or more neatly like this..

Face	Name	Ringed by corner names	Ringed by binary corners	Ringed by corners
0	xy1	origo, x, xy, y	0000, 0001, 0011, 0010	0, 1, 3, 2
1	xy2	z, xz, xyz, yz	0100, 0101, 0111, 0110	4, 5, 7, 6
2	xy3	q, xq, xyq, yq	1000, 1001, 1011, 1010	8, 9, 11, 10
3	xy4	zq, xzq, xyzq, yzq	1100, 1101, 1111, 1110	12, 13, 15, 14
4	xz1	origo, x, xz, z	0000, 0001, 0101, 0100	0, 1, 5, 4
5	xz2	y, xy, xyz, yz	0010, 0011, 0111, 0110	2, 3, 7, 6
6	xz3	q, xq, xzq, zq	1000, 1001, 1101, 1100	8, 9, 13, 12
7	xz4	yq, xyq, xyzq, yzq	1010, 1011, 1111, 1110	10, 11, 15, 14
8	yz1	origo, y, yz, z	0000, 0010, 0110, 0100	0, 2, 6, 4
9	yz2	x, xy, xyz, xz	0001, 0011, 0111, 0101	1, 3, 7, 5
10	yz3	q, yq, yzq, zq	1000, 1010, 1110, 1100	8, 10, 14, 12
11	yz4	xq, xyq, xyzq, xzq	1001, 1011, 1111, 1101	9, 11, 15, 13
12	xq1	origo, x, xq, q	0000, 0001, 1001, 1000	0, 1, 9, 8
13	xq2	y, xy, xyq, yq	0010, 0011, 1011, 1010	2, 3, 11, 10

14	xq3	z, xz, xzq, zq	0100, 0101, 1101, 1100	4, 5, 13, 12
15	xq4	yz, xyz, xyzq, yzq	0110, 0111, 1111, 1110	6, 7, 15, 14
16	yq1	origo, y, yq, q	0000, 0010, 1010, 1000	0, 2, 10, 8
17	yq2	x, xy, xyq, xq	0001, 0011, 1011, 1001	1, 3, 11, 9
18	yq3	z, yz, yzq, zq	0100, 0110, 1110, 1100	4, 6, 14, 12
19	yq4	xz, xyz, xyzq, xzq	0101, 0111, 1111, 1101	5, 7, 15, 13
20	zq1	origo, z, zq, q	0000, 0100, 1100, 1000	0, 4, 12, 8
21	zq2	x, xz, xzq, xq	0001, 0101, 1101, 1001	1, 5, 13, 9
22	zq3	y, yz, yzq, yq	0010, 0110, 1110, 1010	2, 6, 14, 10
23	zq4	xy, xyz, xyzq, xyq	0011, 0111, 1111, 1011	3, 7, 15, 11

Prism 4d, room table

A prism in 4D has 8 ordinary prisms as a surface against the outside world of 4D. When they contribute to a 4D object, I use the term “rooms” or “siderooms”. I could perhaps say “spaces” or “volumes”, but I prefer “rooms”. I sort them by their natural names. A sequence number (a reference) is assigned for these rooms and starts on 0. The number has no deeper meaning as opposed to corners.

The rooms have 4 orientations: xyz, xyq, xzq, yzq. I use the terms “orientation” and “parallel room” for entire 3D spaces (realms) because within a 4D world these rooms act like faces on a cube. The total rooms is twice the number of orientations because there are always two opposite rooms with the same orientation (xyz orientation comes as xyz1 and xyz2). I name the 8 rooms: xyz1, xyz2, xyq1, xyq2, xzq1, xzq2, yzq1, yzq2.

Each room has 6 faces, and each face is used twice (shared by two rooms). Which faces contribute to a given room.. I found no rule, so I am including a table where all the faces are expanded to their respective corners. It was obtained by trial and error. When I gather all the corners for one prism, removing duplicates as in a set operation makes me end up with 8 corners for each room. This confirms that the faces really belong to that room (to that prism in 3D). If the faces were wrong, I would get more than 8 corners per room.

Room 0 or xyz1 has faces [xy1, xy2, xz1, xz2, yz1, yz2] or [0, 1, 4, 5, 8, 9].
Room 1 or xyz2 has faces [xy3, xy4, xz3, xz4, yz3, yz4] or [2, 3, 6, 7, 10, 11].
Room 2 or xyq1 has faces [xy1, xy3, xq1, xq2, yq1, yq2] or [0, 2, 12, 13, 16, 17].
Room 3 or xyq2 has faces [xy2, xy4, xq3, xq4, yq3, yq4] or [1, 3, 14, 15, 18, 19].
Room 4 or xzq1 has faces [xz1, xz3, xq1, xq3, zq1, zq2] or [4, 6, 12, 14, 20, 21].
Room 5 or xzq2 has faces [xz2, xz4, xq2, xq4, zq3, zq4] or [5, 7, 13, 15, 22, 23].
Room 6 or yzq1 has faces [yz1, yz3, yq1, yq3, zq1, zq3] or [8, 10, 16, 18, 20, 22].
Room 7 or yzq2 has faces [yz2, yz4, yq2, yq4, zq2, zq4] or [9, 11, 17, 19, 21, 23].

When I expand the list of faces to a longer list of corners..

Room 0 or xyz1 has corners [0, 1, 3, 2], [4, 5, 7, 6], [0, 1, 5, 4], [2, 3, 7, 6], [0, 2, 6, 4], [1, 3, 7, 5].

Room 1 or xyz2 has corners [8, 9, 11, 10], [12, 13, 15, 14], [8, 9, 13, 12], [10, 11, 15, 14], [8, 10, 14, 12],

[9, 11, 15, 13].

Room 2 or xyq1 has corners [0, 1, 3, 2], [8, 9, 11, 10], [0, 1, 9, 8], [2, 3, 11, 10], [0, 2, 10, 8], [1, 3, 11, 9].

Room 3 or xyq2 has corners [4, 5, 7, 6], [12, 13, 15, 14], [4, 5, 13, 12], [6, 7, 15, 14], [4, 6, 14, 12], [5, 7, 15, 13].

Room 4 or xzq1 has corners [0, 1, 5, 4], [8, 9, 13, 12], [0, 1, 9, 8], [4, 5, 13, 12], [0, 4, 12, 8], [1, 5, 13, 9].

Room 5 or xzq2 has corners [2, 3, 7, 6], [10, 11, 15, 14], [2, 3, 11, 10], [6, 7, 15, 14], [2, 6, 14, 10], [3, 7, 15, 11].

Room 6 or yzq1 has corners [0, 2, 6, 4], [8, 10, 14, 12], [0, 2, 10, 8], [4, 6, 14, 12], [0, 4, 12, 8], [2, 6, 14, 10].

Room 7 or yzq2 has corners [1, 3, 7, 5], [9, 11, 15, 13], [1, 3, 11, 9], [5, 7, 15, 13], [1, 5, 13, 9], [3, 7, 15, 11].

With duplicates removed the list of corners adds up to 8 distinct corners every time..

Room 0 or xyz1 has corners [0, 1, 2, 3, 4, 5, 6, 7].
Room 1 or xyz2 has corners [8, 9, 10, 11, 12, 13, 14, 15].
Room 2 or xyq1 has corners [0, 1, 2, 3, 8, 9, 10, 11].
Room 3 or xyq2 has corners [4, 5, 6, 7, 12, 13, 14, 15].
Room 4 or xzq1 has corners [0, 1, 4, 5, 8, 9, 12, 13].
Room 5 or xzq2 has corners [2, 3, 6, 7, 10, 11, 14, 15].
Room 6 or yzq1 has corners [0, 2, 4, 6, 8, 10, 12, 14].
Room 7 or yzq2 has corners [1, 3, 5, 7, 9, 11, 13, 15].

This list confirms the list of prisms in 3D around a prism in 4D and how these prisms consist of a specific set of faces (sides). These tables and lists are what I need to make a computer program. Actually I don't need these lists either because the program could deduct them. But that would be harder to program (to "code", a word that was meant for the brave men and women that converted human language into bits and bytes of machine code, now it has become the new buzzword meaning something else).

The program

index.html..
The essence of the page index.html is:

```
<canvas id="myCanvas" width="1400" height="800"></canvas>
```

The canvas tag reserves a drawing board. Everything is coded in Javascript (also called Ecmascript, which is the orginal name). There are no elements that you can place on the canvas just using tags.

Directly below the canvas the code files are declared (are included):

```
    <script src="util.js"></script>
    <script src="prism3d.js"></script>
    <script src="prism4d.js"></script>
    <script src="scene.js"></script>
    <script src="main.js"></script>
```

Their sequence is important. The script util.js present functions and classes that the other script files need. There is a dependency chain like a one way street. The file that depends on another file must succeed that file. Script main.js relies on scene.js, that relies on prism3d.js and prism4d.js, that both rely on util.js. There are no external dependencies (no need for jquery). It can run without the Internet. All files should be in the same folder. When you start (choose open or doubleclick) index.html, it expects the rest of the program to be in this folder.

The html page has a line that connects to the program:

```
<body onload="OnLoad()">
```

main.js..
Then in main.js an event with this name is triggered:

```
function OnLoad() {
    PrepareCanvas();
    Create();
}
```

The main function is Create:

```
function Create() {
    ClearCanvas();
    scene.Draw(ctx);
    window.setTimeout(Create, 10); // animation loop, repeat the call after 10 ms
}
```

This is the mechanism that drives the animation. The function invokes itself over and over with 10 milliseconds interval. It is a repeating timer event that calls the Draw() method in scene.js. The canvas is also cleared. Javascript drawing / painting is very fast, consider it being a script language (not native code). Tip: Increase 10 ms to 50 ms to slow down animation.

Drawing lines in 3d space

The structure of the program is like this..

Create(), a timer event loop, in file main.js..
calls Draw() every 10 millisec in SceneClass in file scene.js..
calls DrawFaces() in Prism3dClass in file prism3d.js..
and DrawLines() in Prism3dClass in file prism3d.js..
and DrawRooms() in Prism4dClass in file prism4d.js..
and DrawLines() in Prism4dClass in file prism4d.js.

The objects are constructed like this..

A prism of Prism3dClass is mainly defined by its corners of type Corner3dClass.

Every corner knows about itself (its own number) and its neighbours.
These are the id numbers of 3 other corners (a list / array).
Every corner also has a point object of type Point3dClass.
The point contains the 3 coordinates you need to locate the corner in 3D space.

All the classes mentioned here are contained in file prism3d.js. The data of this type support the DrawLines() method where you see an object drawn as wire mesh without colors. The drawing is accomplished simply by moving to each corner in turn and draw 3 lines out from it (rays). To avoid drawing a line twice a line is never drawn to a neighbour with a lower id number.

Code fragments below..

```
class Prism3dClass {

    constructor(Name) {
        this.Name = Name;

        this.Length = 100;
        this.Width = 100;
        this.Height = 100;

        this.Corners = [];
        this.Corners[0] = new Corner3dClass(0, [1, 2, 4]);
        this.Corners[1] = new Corner3dClass(1, [0, 3, 5]);
        this.Corners[2] = new Corner3dClass(2, [0, 3, 6]);

        this.Corners[0].Point = new Point3dClass(0, 0, 0);
        this.Corners[1].Point = new Point3dClass(this.Length, 0, 0);
        this.Corners[2].Point = new Point3dClass(0, this.Width, 0);

    DrawLines(ctx, plane) {

                            ctx.moveTo(corner.Point.X, corner.Point.Z);
                            ctx.lineTo(toCorner.Point.X, toCorner.Point.Z);

            ctx.stroke();
    }
```

Drawing lines in 4d space

A prism of Prism4dClass is defined a long way by its corners of type Corner4dClass.
Every corner knows about itself (its own number) and its neighbours.
These are the id numbers of 4 other corners (a list / array).
Every corner also has a point object of type Point4dClass.
The point contains the 4 coordinates you need to locate the corner in 4D space.

All the classes mentioned here are contained in file prism4d.js. The data of this type support the DrawLines() method where you see an object made from wire mesh without colors. The drawing is done simply by moving to each corner in turn and draw 4 lines out from it (rays). To avoid drawing a line twice the line is never drawn to a neighbour with a lower id number.

Code fragments below..

```
class Prism4dClass {
    constructor(name) {
```

```
        this.Name = name;

        this.Length = 100;
        this.Width = 100;
        this.Height = 100;
        this.Depth = 100;

        this.Corners = [];
        this.Corners[0] = new Corner4dClass(0, [1, 2, 4, 8]);
        this.Corners[1] = new Corner4dClass(1, [0, 3, 5, 9]);
        this.Corners[2] = new Corner4dClass(2, [0, 3, 6, 10]);

        this.Corners[0].Point = new Point4dClass(0, 0, 0, 0);
        this.Corners[1].Point = new Point4dClass(this.Length, 0, 0, 0);
        this.Corners[2].Point = new Point4dClass(0, this.Width, 0, 0);

    DrawLines(ctx, plane) {

                            ctx.moveTo(corner.Point.X, corner.Point.Q);
                            ctx.lineTo(toCorner.Point.X, toCorner.Point.Q);

            ctx.stroke();
    }
```

Trigonometry

There is a basic PointClass that is neither 3D or 4D. It works in 2D, but supports both 3D and 4D. It is contained in the util.js file. Real math is never performed elsewhere (trigonometry). The formula is placed only here and is called from anywhere. I also include class Lib since the Radians() function is necessary (convert from degrees to radians). Lib is not actually a class, but rather a library of functions. One important thing about the xy-plane calculating X and Y: This does not mean that it is limited to the xy-plane. The properties are only local names. They could just as well come from another plane (like the yz-plane in 3D or the zq-plane in 4D).

```
////////////////////////////////////////////////////////////////////////

class Lib {
    //Convert radians to degrees..
    static Degrees(radians) { return (radians * 180 / Math.PI); }

    //Convert degrees to radians..
    static Radians(degrees) { return (degrees * Math.PI / 180); }
}

////////////////////////////////////////////////////////////////////////

class PointClass {
    constructor(X, Y) {
        this.X = (X == undefined ? 0 : X);
        this.Y = (Y == undefined ? 0 : Y);
    }

    Rotate(Rotation) {
        Rotation = (Rotation == undefined ? 0 : Rotation);
        var angle = Lib.Radians(Rotation) * (-1);
        var tempX = this.X * Math.cos(angle) - this.Y * Math.sin(angle);
        var tempY = this.X * Math.sin(angle) + this.Y * Math.cos(angle);
        this.X = tempX;
        this.Y = tempY;
```

```
    }

    Clone() {
        var obj = new PointClass(this.X, this.Y);
        return obj;
    }

    toString() {
        var mes = "Point: " +
            this.X.toFixed(1) + ", " +
            this.Y.toFixed(1);
        return mes;
    }
}
```

Colors

To color the objects (paint them) I need to draw the lines around a face as if it was a ring, then execute fill(). The stroke-method is unnecessary since I don't want to draw the lines twice. It is all about painting a color. So instead of drawing rays from a point, I now draw a sequence of invisible lines.

A homemade class supports this paint process. It has 8 colors to differentiate the 8 prisms in 3D that surround a prism in 4D. Only 4 colors are shown below (blue, green, cyan, violet). The first line shows the rgb codes for clean colors. The 15 following lines are shades of the same color, eventually fading into white. These are like a menu to pick from so that e.g. a red cube could be shown in 6 shades, one shade of red for each face of a dice.

```
//////////////////////////////////////////////////////////////////

class PaletteClass {
    constructor() {
        //blue, green, cyan, violet, red, magenta, orange, yellow
        this.Colors = [
            ["#0000FF", "#00FF00", "#00FFFF", "#8000FF"],
            ["#1111FF", "#11FF11", "#11FFFF", "#8811FF"],
            ["#2222FF", "#22FF22", "#22FFFF", "#9022FF"],
            ["#3333FF", "#33FF33", "#33FFFF", "#9833FF"],
            ["#4444FF", "#44FF44", "#44FFFF", "#A044FF"],
            ["#5555FF", "#55FF55", "#55FFFF", "#A855FF"],
            ["#6666FF", "#66FF66", "#66FFFF", "#B066FF"],
            ["#7777FF", "#77FF77", "#77FFFF", "#B877FF"],
            ["#8888FF", "#88FF88", "#88FFFF", "#C088FF"],
            ["#9999FF", "#99FF99", "#99FFFF", "#C899FF"],
            ["#AAAAFF", "#AAFFAA", "#AAFFFF", "#D0AAFF"],
            ["#BBBBFF", "#BBFFBB", "#BBFFFF", "#D8BBFF"],
            ["#CCCCFF", "#CCFFCC", "#CCFFFF", "#E0CCFF"],
            ["#DDDDFF", "#DDFFDD", "#DDFFFF", "#E8DDFF"],
            ["#EEEEFF", "#EEFFEE", "#EEFFFF", "#F0EEFF"],
            ["#FFFFFF", "#FFFFFF", "#FFFFFF", "#FFFFFF"]
        ];
    }
}
```

Draw colors in 3d

A prism of Prism3dClass is also defined by its faces (type FaceClass), not only by the corners.
Every face (side) has an id number and a ring of 4 corners.
They are id numbers that point to corners of type Corner3dClass (a list / array).
The FaceClass is more of a generic data structure than a class.

It belongs to neither 3D or 4D, but makes it easier to pass arguments.
The id numbers could just as well point to Corner4dClass objects.
Therefore it resides in util.js.

Another class, RingClass, defines a list of integers as a ring. This means that the last element connects to the first element like there was no start or end. FaceClass extends the RingClass (inherits its methods and properties). The FaceClass data support the DrawFaces() method where you see a prism in 3D painted as six surfaces with different contrasts or shades. When faces are drawn (painted), a color is picked from the palette based on results from 2 functions, ColorGroup() for the base color and ColorIndex() for contrast or shade. If the situation changes, you can change these functions so that other colors are produced from the same behind the curtain indices.

Code fragments below..

```
        this.Faces = [];
        this.Faces[0] = new FaceClass(0, [0, 1, 3, 2]);
        this.Faces[1] = new FaceClass(1, [4, 5, 7, 6]);


    ColorGroup(index) {
        return (index % 8);
    }

    ColorIndex(index) {
        return (index * 2 % 16);
    }

    DrawFaces(ctx, plane, colorGroup) {
        var colorIndex, color;
        this.Faces.forEach((face, index) => {
            colorIndex = this.ColorIndex(index);
            color = this.Palette.Colors[colorIndex][colorGroup];
            this.DrawFace(ctx, plane, face, color);
        });
    }
```

Draw colors in 4d

A prism of Prism4dClass is also defined by its faces (type FaceClass), not only corners.
In addition it is defined as 8 rooms (type RoomClass).
Every room (prism in 3D) has an id number, a list of faces (6 ids) and a list of corners (8 ids).
The first list just groups 6 of the faces so they can roughly receive the same color.
The second list are id numbers that point to corners of type Corner4dClass.
This list is never used by the program and could be left out, but is also a good control.
Like the FaceClass, also the RoomClass is more of a generic data structure than a class.
Neither belong to 3D or 4D, they just make it easier to pass data (arguments, parameters).
Therefore they reside in util.js.

Code fragments below..

```
this.Faces = [];
// xy..
this.Faces[0] = new FaceClass(0, [0, 1, 3, 2]);
this.Faces[1] = new FaceClass(1, [4, 5, 7, 6]);
```

```
// zq..
this.Faces[20] = new FaceClass(20, [0, 4, 12, 8]);
this.Faces[21] = new FaceClass(21, [1, 5, 13, 9]);
this.Faces[22] = new FaceClass(22, [2, 6, 14, 10]);
this.Faces[23] = new FaceClass(23, [3, 7, 15, 11]);

this.Rooms = [];
// xyz..
this.Rooms[0] = new RoomClass(0, [0, 1, 4, 5, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7]);
this.Rooms[1] = new RoomClass(1, [2, 3, 6, 7, 10, 11], [8, 9, 10, 11, 12, 13, 14, 15])
;
// xyq..
this.Rooms[2] = new RoomClass(2, [0, 2, 12, 13, 16, 17], [0, 1, 2, 3, 8, 9, 10, 11]);
```

How to set the scene

The script scene.js does not follow a predestined trail, which is another technique if you want to animate for example industrial production. In this program it is all about repeating increments of rotation without knowing where it will end. But I also wish to hold still and show, with the help of colors, where the 8 ordinary cubes are within the 4D cube. Or perhaps change between 3D and 4D. Or perhaps fade the colors into a whiter shade of pale. After each drawing of a still photo the scene must return to the timer event loop in main.js. There is a chance the program could lose track of how far it has come.

I solved it with three counters. A property Counter counts down from let us say 500, a pure repetiton of cycling thru a movement or a set of colors or just holding still. When it reaches zero, the next Bullet (also a decrement) indicates the next stage, which means a different action. A “switch case” structure decides what to do. To avoid a long flat list of bullets there is a higher level, a Bulletin. This makes it easier to renumber actions since I don’t have to go thru the whole list. When there are no more bullets in a bulletin, this is signalled by resetting Bullet to zero. Then comes the next Bulletin. When there are no more bulletins, this is signalled by setting Bulletin to zero. This is the trigger that sets off the whole process over again.

One interesting thing: The rotation continues in such a way that you will hardly see exactly the same positions twice.

The program has been improved so that the user can directly manipulate the rotation speed, forward and backward rotation (to some extent, meant for close study at a certain moment), what 3D prism out of 8 prisms is shown with a color. Intervention is achieved by clicking radio buttons. When you have chosen one radio button, you can easily move within the group with the keyboard arrows.

The last half of this document lists the whole program..

index.html

```
<!DOCTYPE html>
<html lang="no">

<head>
    <meta charset="UTF-8" />
    <title>Prism4D</title>
    <link href="default.css" rel="stylesheet" />
</head>

<body onload="OnLoad()">
    <b>
        <span>Color</span>
        <input type="radio" name="myColor" value="0"> Blue
        <input type="radio" name="myColor" value="1"> Green
        <input type="radio" name="myColor" value="2"> Cyan
        <input type="radio" name="myColor" value="3"> Violet
        <input type="radio" name="myColor" value="4"> Red
        <input type="radio" name="myColor" value="5"> Magenta
        <input type="radio" name="myColor" value="6"> Orange
        <input type="radio" name="myColor" value="7"> Yellow
        <input type="radio" name="myColor" value="-1" checked> Auto
    </b>
    <hr>
    <b>
        <span>Rate</span>
        <input type="radio" name="myRate" value="-1"> -1
        <input type="radio" name="myRate" value="-0.5"> -0.5
        <input type="radio" name="myRate" value="-0.1"> -0.1
        <input type="radio" name="myRate" value="0"> 0
        <input type="radio" name="myRate" value="0.1"> 0.1
        <input type="radio" name="myRate" value="0.5"> 0.5
        <input type="radio" name="myRate" value="1" checked autofocus> 1
    </b>
    <span>          </span>
    <a href="About.txt" target="_blank">...</a>
    <hr>
    <code><b>
        <div id="myText1">ENG: Rotating cubes, 3D and 4D, projected to all planes.</div>
        <div id="myText2">NOR: Roterende kuber, 3D og 4D, projisert til alle plan.</div>
    </b></code>

    <canvas id="myCanvas" width="1400" height="800"></canvas>

    <script src="util_v1.js"></script>
    <script src="prism3d_v1.js"></script>
    <script src="prism4d_v1.js"></script>
    <script src="scene_v1.js"></script>
    <script src="main_v1.js"></script>
</body>

</html>
```

main.js

```
/*** prism main.js ***/

////////////////////////////////////////////////////////
//   Prism 4.1 - author Svein Skogland - Norway 2021   //
////////////////////////////////////////////////////////

var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
var scale = 1;
var scene = new SceneClass("My4DStory");

function PrepareCanvas() // Adjust canvas, pens and brushes..
{
    scale = scene.Scale;
    ctx.lineWidth = 1; // 2 = thicker
    ctx.strokeStyle = "#000000"; // black
    ctx.fillStyle = "#BBBBFF"; // grey blue
    ctx.scale(scale, -1 * scale); // reverse y-axis for normal geometry
    ctx.translate(0, -600); // go down to x-axis at y = 0
    ctx.globalAlpha = 1.0; // 0.7 = a little transparent
}

function ClearCanvas() {
    ctx.beginPath();
    ctx.clearRect(-500, -500, 5000, 5000);
    ctx.closePath();
}

function Create() {
    ClearCanvas();
    scene.Draw(ctx);
    window.setTimeout(Create, 10); // animation loop, repeat the call after 10 ms
}

function OnLoad() {
    PrepareCanvas();
    Create();
}
```

scene.js

```
/*** prism scene.js ***/

//////////////////////////////////////////////////////////////////////

class SceneClass {
    constructor(Name) {
        //Object state..
        this.Name = Name;
        this.Prism3 = new Prism3dClass("Prism3d");
        this.Prism4 = new Prism4dClass("Prism4d");
        this.Bulletin = 0;
        this.Bullet = 0;
        this.Counter = 0.0;
        this.Count = 0;
        this.Text1 = document.getElementById("myText1");
        this.Text2 = document.getElementById("myText2");
        this.Colors = document.getElementsByName("myColor");
        this.Rates = document.getElementsByName("myRate");
        this.Color = -1;
        this.Rate = 1;
        this.Language = 1; //0=silent, 1=eng, 2=nor
        this.Scale = 1;
        this.Paintlist = [];
        //alert(`Screen ${screen.width}, ${screen.height}`);
        this.Phone = (screen.width < 500); //or < 768 for old iPad
        if (this.Phone) this.Scale = 2;
    }

    // SlowDownRate() {
    //     this.Rate1.stepDown();
    // }

    // SpeedUpRate() {
    //     this.Rate1.stepUp();
    // }

    Draw(ctx) {
        var lang = this.Language;
        if (this.Bullet == 0) this.Bulletin++;
        if (this.Counter <= 0) this.Bullet++;
        switch (this.Bulletin) {
            case 1:
                // First 7 seconds show 3D cube..
                switch (this.Bullet) {
                    case 1:
                        // Spin 3D cube for 7 seconds in 3 projections and colors with 6 faces and
shades
                        if (this.Counter <= 0) {
                            this.Counter = 700;
                            this.Count = this.Counter;
                        }
                        this.Text1.innerHTML = "3D cube surrounded by 6 squares that started in 2
parallel XY-planes, 2 parallel XZ-planes and 2 parallel YZ-planes.";
                        this.Text2.innerHTML = "3D kube omgitt av 6 kvadrat som startet i 2
parallelle XY-plan, 2 parallelle XZ-plan og 2 parallelle YZ-plan.";
                        //var phrases = ["",
                        //"3D cube surrounded by 6 squares that started in 2 parallel XY-planes, 2
parallel XZ-planes and 2 parallel YZ-planes.",
                        //"3D kube omgitt av 6 kvadrat som startet i 2 parallelle XY-plan, 2
parallelle XZ-plan og 2 parallelle YZ-plan."];
                        //this.Text2.innerHTML = phrases[lang];
                        //this.Text2.innerHTML = this.Rate1.value;
                        this.Draw3D(ctx);
                        break;
                    case 2:
                        // Reserved for fading
```

```
                    break;
                default:
                    this.Bullet = 0;
                    break;
            }
            break;

        case 2:
            // Next 9 seconds present a frozen 4D cube, first time glued to its axis..
            this.CycleColors4D(ctx);
            break;

        case 3:
            // Next 4 minutes spin 4D cube..
            this.TurnPrism4D(ctx)
            break;

        case 4:
            // Next 9 seconds present a frozen 4D cube skewed from its axis..
            this.CycleColors4D(ctx);
            break;

        case 5:
            // Next 4 minutes spin 4D cube..
            this.TurnPrism4D(ctx)
            break;

        default:
            // Repeat the whole sequence..
            this.Bullet = 0;
            this.Bulletin = 0;
            break;
    }
    this.Paintlist = [];
    this.Colors.forEach(node => { if (node.checked) this.Color = Number(node.value); });
    if (this.Color >= 0 && this.Color <= 7) this.Paintlist[0] = this.Color;
    this.Rates.forEach(node => { if (node.checked) this.Rate = Number(node.value); });
    //if (this.Counter > 0) this.Counter--;
    if (this.Counter > 0) {
        this.Counter -= this.Rate;
        if (this.Counter > this.Count) this.Counter = this.Count;
    }
}

CycleColors4D(ctx) {
    //For about 9 seconds present a frozen 4D cube, first time glued to its axis..
    switch (this.Bullet) {
        case 1:
            // Hold 4D cube for 1 second in 6 projections, each with 8 cubes and colors, each
with 6 faces and shades.
            //if (this.Counter == 0) this.Counter = 100;
            if (this.Counter <= 0) {
                this.Counter = 100;
                this.Count = this.Counter;
            }
            this.Text1.innerHTML = "4D 'cube' with eighth color (yellow) painted last.";
            this.Text2.innerHTML = "4D 'kube' påført åttende farge (gul) sist.";
            this.Draw4D(ctx, [0, 1, 2, 3, 4, 5, 6, 7], true);
            break;
        case 2:
            // Hold 4D cube for 1 second with first color added over again.
            //if (this.Counter <= 0) this.Counter = 100;
            if (this.Counter <= 0) {
                this.Counter = 100;
                this.Count = this.Counter;
            }
            this.Text1.innerHTML = "4D 'cube' with first color (blue) painted last.";
            this.Text2.innerHTML = "4D 'kube' påført første farge (blå) sist.";
            this.Draw4D(ctx, [1, 2, 3, 4, 5, 6, 7, 0], true);
```

```
                break;
            case 3:
                // Hold 4D cube for 1 second with second color added over again.
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with second color (green) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført andre farge (grønn) sist.";
                this.Draw4D(ctx, [2, 3, 4, 5, 6, 7, 0, 1], true);
                break;
            case 4:
                // Hold 4D cube for another 5 seconds with third to seventh color added over again
(1 second each).
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with third color (cyan) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført tredje farge (cyan) sist.";
                this.Draw4D(ctx, [3, 4, 5, 6, 7, 0, 1, 2], true);
                break;
            case 5:
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with fourth color (violet) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført fjerde farge (fiolett) sist.";
                this.Draw4D(ctx, [4, 5, 6, 7, 0, 1, 2, 3], true);
                break;
            case 6:
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with fifth color (red) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført femte farge (rød) sist.";
                this.Draw4D(ctx, [5, 6, 7, 0, 1, 2, 3, 4], true);
                break;
            case 7:
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with sixth color (magenta) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført sjette farge (magenta) sist.";
                this.Draw4D(ctx, [6, 7, 0, 1, 2, 3, 4, 5], true);
                break;
            case 8:
                //if (this.Counter <= 0) this.Counter = 100;
                if (this.Counter <= 0) {
                    this.Counter = 100;
                    this.Count = this.Counter;
                }
                this.Text1.innerHTML = "4D 'cube' with seventh color (orange) painted last.";
                this.Text2.innerHTML = "4D 'kube' påført sjuende farge (oransje) sist.";
                this.Draw4D(ctx, [7, 0, 1, 2, 3, 4, 5, 6], true);
                break;
            case 9:
                // Hold 4D cube for 0.5 second while all colors fade.
                if (this.Counter <= 0) {
                    this.Counter = 50;
                    this.Count = this.Counter; //remember start count to show progress
```

```
                }
                var contrast = this.Counter * 100 / this.Count;
                this.Draw4D(ctx, undefined, true, contrast);
                break;
            default:
                this.Bullet = 0;
                break;
        }
    }

    TurnPrism4D(ctx) {
        // spin 4D cube..
        var paintlist = this.Paintlist;
        this.Text1.innerHTML = "";
        this.Text2.innerHTML = "";
        switch (this.Bullet) {
            case 1:
                // Spin 4D cube with only 2 cubes painted (and their faces shaded).
                //if (this.Counter <= 0) this.Counter = 6000;
                if (this.Counter <= 0) {
                    this.Counter = 6000;
                    this.Count = this.Counter;
                }
                if (paintlist.length == 0) {
                    paintlist = [0, 1];
                    this.Text1.innerHTML = "4D 'cube' bordered by opposite 3D-cubes that started in
2 parallel XYZ-spaces.";
                    this.Text2.innerHTML = "4D 'kube' avgrenset av motstående 3D-kuber som startet
i 2 parallelle XYZ-rom.";
                }
                this.Draw4D(ctx, paintlist);
                break;
            case 2:
                // Hold 4D cube while 2 colors fade.
                if (this.Counter <= 0) {
                    this.Counter = 50;
                    this.Count = this.Counter;
                }
                var contrast = this.Counter * 100 / this.Count;
                if (paintlist.length == 0) paintlist = [0, 1];
                this.Draw4D(ctx, paintlist, true, contrast);
                break;
            case 3:
                // Repeat 4D cube 'spin and hold' yet another 3 times so all cubes and colors get
their share.
                //if (this.Counter <= 0) this.Counter = 6000;
                if (this.Counter <= 0) {
                    this.Counter = 6000;
                    this.Count = this.Counter;
                }
                if (paintlist.length == 0) {
                    paintlist = [2, 3];
                    this.Text1.innerHTML = "4D 'cube' bordered by opposite 3D-cubes that started in
2 parallel XYQ-spaces.";
                    this.Text2.innerHTML = "4D 'kube' avgrenset av motstående 3D-kuber som startet
i 2 parallelle XYQ-rom.";
                }
                this.Draw4D(ctx, paintlist);
                break;
            case 4:
                if (this.Counter <= 0) {
                    this.Counter = 50;
                    this.Count = this.Counter;
                }
                var contrast = this.Counter * 100 / this.Count;
                if (paintlist.length == 0) paintlist = [2, 3];
                this.Draw4D(ctx, paintlist, true, contrast);
                break;
            case 5:
```

```
                //if (this.Counter <= 0) this.Counter = 6000;
                if (this.Counter <= 0) {
                    this.Counter = 6000;
                    this.Count = this.Counter;
                }
                if (paintlist.length == 0) {
                    paintlist = [4, 5];
                    this.Text1.innerHTML = "4D 'cube' bordered by opposite 3D-cubes that started in
2 parallel XZQ-spaces.";
                    this.Text2.innerHTML = "4D 'kube' avgrenset av motstående 3D-kuber som startet
i 2 parallelle XZQ-rom.";
                }
                this.Draw4D(ctx, paintlist);
                break;
            case 6:
                if (this.Counter <= 0) {
                    this.Counter = 50;
                    this.Count = this.Counter;
                }
                var contrast = this.Counter * 50 / this.Count;
                if (paintlist.length == 0) paintlist = [4, 5];
                this.Draw4D(ctx, paintlist, true, contrast);
                break;
            case 7:
                //if (this.Counter <= 0) this.Counter = 6000;
                if (this.Counter <= 0) {
                    this.Counter = 6000;
                    this.Count = this.Counter;
                }
                if (paintlist.length == 0) {
                    paintlist = [6, 7];
                    this.Text1.innerHTML = "4D 'cube' bordered by opposite 3D-cubes that started in
2 parallel YZQ-spaces.";
                    this.Text2.innerHTML = "4D 'kube' avgrenset av motstående 3D-kuber som startet
i 2 parallelle YZQ-rom.";
                }
                this.Draw4D(ctx, paintlist);
                break;
            case 8:
                if (this.Counter <= 0) {
                    this.Counter = 50;
                    this.Count = this.Counter;
                }
                var contrast = this.Counter * 100 / this.Count;
                if (paintlist.length == 0) paintlist = [6, 7];
                this.Draw4D(ctx, paintlist, true, contrast);
                break;
            default:
                this.Bullet = 0;
                break;
        }
    }

    Draw3D(ctx) {
        // if (!this.Freeze) this.Prism3.Rotate(0.23, 0.13, 0.17); //replaced 2020-06-21-19-40-00
by..

        var rate = this.Rate;
        this.Prism3.Rotate(0.23 * rate, 0.13 * rate, 0.17 * rate);

        ctx.save();

        ctx.translate(150, 450);
        this.Prism3.DrawFaces(ctx, 0, 4);
        this.Prism3.DrawLines(ctx, 0);

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism3.DrawFaces(ctx, 1, 6);
```

```
            this.Prism3.DrawLines(ctx, 1);
        }

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism3.DrawFaces(ctx, 2, 7);
            this.Prism3.DrawLines(ctx, 2);
        }

        ctx.restore();
    }

    Draw4D(ctx, rooms, pause, contrast) {
        var rate = this.Rate;
        if (!pause) this.Prism4.Rotate(0.23 * rate, 0.13 * rate, 0.17 * rate, 0.11 * rate, 0.19 *
rate, 0.29 * rate);

        ctx.save();

        ctx.translate(150, 500);
        this.Prism4.DrawRooms(ctx, 0, rooms, contrast);
        this.Prism4.DrawLines(ctx, 0, rooms);

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism4.DrawRooms(ctx, 1, rooms, contrast);
            this.Prism4.DrawLines(ctx, 1, rooms);
        }

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism4.DrawRooms(ctx, 2, rooms, contrast);
            this.Prism4.DrawLines(ctx, 2, rooms);
        }

        ctx.restore();

        ctx.save();

        if (!this.Phone) {
            ctx.translate(150, 300);
            this.Prism4.DrawRooms(ctx, 3, rooms, contrast);
            this.Prism4.DrawLines(ctx, 3, rooms);
        }

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism4.DrawRooms(ctx, 4, rooms, contrast);
            this.Prism4.DrawLines(ctx, 4, rooms);
        }

        if (!this.Phone) {
            ctx.translate(250, 0);
            this.Prism4.DrawRooms(ctx, 5, rooms, contrast);
            this.Prism4.DrawLines(ctx, 5, rooms);
        }

        ctx.restore();
    }
}
```

prism3d.js

```
/*** prism prism3d.js ***/

//////////////////////////////////////////////////////////////////////

class Point3dClass {
    constructor(X, Y, Z) {
        this.X = (X == undefined ? 0 : X);
        this.Y = (Y == undefined ? 0 : Y);
        this.Z = (Z == undefined ? 0 : Z);
    }

    RotateXY(Rotation) {
        var point = new PointClass(this.X, this.Y);
        point.Rotate(Rotation);
        this.X = point.X;
        this.Y = point.Y;
    }

    RotateXZ(Rotation) {
        var point = new PointClass(this.X, this.Z);
        point.Rotate(Rotation);
        this.X = point.X;
        this.Z = point.Y;
    }

    RotateYZ(Rotation) {
        var point = new PointClass(this.Y, this.Z);
        point.Rotate(Rotation);
        this.Y = point.X;
        this.Z = point.Y;
    }

    Clone() {
        var obj = new Point3dClass();
        obj.X = this.X;
        obj.Y = this.Y;
        obj.Z = this.Z;
        return obj;
    }

    toString() {
        var mes = "Point: " +
            this.X.toFixed(1) + ", " +
            this.Y.toFixed(1) + ", " +
            this.Z.toFixed(1);
        return mes;
    }
}

//////////////////////////////////////////////////////////////////////

class Corner3dClass {
    constructor(Id, ToIds) {
        this.Id = (Id == undefined ? 0 : Id);
        this.ToIds = (ToIds == undefined ? [] : ToIds);
        this.Point = new Point3dClass();
    }

    Clone() {
        var obj = new Corner3dClass();
        obj.Id = this.Id;
        obj.ToIds = Array.from(this.ToIds);
        obj.Point = this.Point.Clone();
        return obj;
    }
```

```
    toString() {
        var mes = "\n Corner " + this.Id +
            " to corners " +
            this.ToIds[0] + ", " +
            this.ToIds[1] + ", " +
            this.ToIds[2] + ". " +
            this.Point;
        return mes;
    }
}

//////////////////////////////////////////////////////////////////////

class Prism3dClass {

    constructor(Name) {
        this.Name = Name;

        this.Length = 100;
        this.Width = 100;
        this.Height = 100;

        this.Corners = [];
        this.Corners[0] = new Corner3dClass(0, [1, 2, 4]);
        this.Corners[1] = new Corner3dClass(1, [0, 3, 5]);
        this.Corners[2] = new Corner3dClass(2, [0, 3, 6]);
        this.Corners[3] = new Corner3dClass(3, [1, 2, 7]);
        this.Corners[4] = new Corner3dClass(4, [0, 5, 6]);
        this.Corners[5] = new Corner3dClass(5, [1, 4, 7]);
        this.Corners[6] = new Corner3dClass(6, [2, 4, 7]);
        this.Corners[7] = new Corner3dClass(7, [3, 5, 6]);

        this.Corners[0].Point = new Point3dClass(0, 0, 0);
        this.Corners[1].Point = new Point3dClass(this.Length, 0, 0);
        this.Corners[2].Point = new Point3dClass(0, this.Width, 0);
        this.Corners[3].Point = new Point3dClass(this.Length, this.Width, 0);
        this.Corners[4].Point = new Point3dClass(0, 0, this.Height);
        this.Corners[5].Point = new Point3dClass(this.Length, 0, this.Height);
        this.Corners[6].Point = new Point3dClass(0, this.Width, this.Height);
        this.Corners[7].Point = new Point3dClass(this.Length, this.Width, this.Height);

        var point = this.Corners[7].Point;
        this.Shift(-point.X / 2, -point.Y / 2, -point.Z / 2);

        // alert(this.Corners);

        this.Faces = [];
        this.Faces[0] = new FaceClass(0, [0, 1, 3, 2]);
        this.Faces[1] = new FaceClass(1, [4, 5, 7, 6]);
        this.Faces[2] = new FaceClass(2, [0, 1, 5, 4]);
        this.Faces[3] = new FaceClass(3, [2, 3, 7, 6]);
        this.Faces[4] = new FaceClass(4, [0, 2, 6, 4]);
        this.Faces[5] = new FaceClass(5, [1, 3, 7, 5]);

        this.Palette = new PaletteClass();
    }

    Shift(dX, dY, dZ) {
        this.Corners.forEach(corner => {
            corner.Point.X += dX;
            corner.Point.Y += dY;
            corner.Point.Z += dZ;
        });
    }

    Rotate(XYTurn, XZTurn, YZTurn) {
        this.Corners.forEach(corner => {
            corner.Point.RotateXY(XYTurn);
            corner.Point.RotateXZ(XZTurn);
```

```
        corner.Point.RotateYZ(YZTurn);
    });
}

ColorGroup(index) {
    return (index % 8);
}

ColorIndex(index) {
    return (index * 2 % 16);
}

DrawFaces(ctx, plane, colorGroup) {
    var colorIndex, color;
    this.Faces.forEach((face, index) => {
        colorIndex = this.ColorIndex(index);
        color = this.Palette.Colors[colorIndex][colorGroup];
        this.DrawFace(ctx, plane, face, color);
    });
}

DrawFace(ctx, plane, face, color) {
    ctx.fillStyle = color;
    ctx.beginPath();
    face.Set();
    var corner, toCorner;
    for (let index = 0; index < face.Count; index++) {
        corner = this.Corners[face.Value];
        // select projection..
        switch (plane) {
            case 0:
                if (face.Index == 0) ctx.moveTo(corner.Point.X, corner.Point.Y);
                break;
            case 1:
                if (face.Index == 0) ctx.moveTo(corner.Point.X, corner.Point.Z);
                break;
            case 2:
                if (face.Index == 0) ctx.moveTo(corner.Point.Y, corner.Point.Z);
                break;
            default:
                break;
        }
        face.Next();
        toCorner = this.Corners[face.Value];
        // select projection..
        switch (plane) {
            case 0:
                ctx.lineTo(toCorner.Point.X, toCorner.Point.Y);
                break;
            case 1:
                ctx.lineTo(toCorner.Point.X, toCorner.Point.Z);
                break;
            case 2:
                ctx.lineTo(toCorner.Point.Y, toCorner.Point.Z);
                break;
            default:
                break;
        }
    }
    ctx.fill();
}

DrawLines(ctx, plane) {
    this.Corners.forEach(corner => {
        corner.ToIds.forEach(toId => {
            if (corner.Id < toId) {
                var toCorner = this.Corners[toId];
                // select projection..
                switch (plane) {
```

```
                        case 0:
                            ctx.moveTo(corner.Point.X, corner.Point.Y);
                            ctx.lineTo(toCorner.Point.X, toCorner.Point.Y);
                            break;
                        case 1:
                            ctx.moveTo(corner.Point.X, corner.Point.Z);
                            ctx.lineTo(toCorner.Point.X, toCorner.Point.Z);
                            break;
                        case 2:
                            ctx.moveTo(corner.Point.Y, corner.Point.Z);
                            ctx.lineTo(toCorner.Point.Y, toCorner.Point.Z);
                            break;
                        default:
                            break;
                    }
                }
            });
            ctx.stroke();
        });
    }
}
```

prism4d.js

```
/*** prism prism4d.js ***/

//////////////////////////////////////////////////////////////////////

class Point4dClass {
    constructor(X, Y, Z, Q) {
        this.X = (X == undefined ? 0 : X);
        this.Y = (Y == undefined ? 0 : Y);
        this.Z = (Z == undefined ? 0 : Z);
        this.Q = (Q == undefined ? 0 : Q);
    }

    RotateXY(rotation) {
        var point = new PointClass(this.X, this.Y);
        point.Rotate(rotation);
        this.X = point.X;
        this.Y = point.Y;
    }

    RotateXZ(rotation) {
        var point = new PointClass(this.X, this.Z);
        point.Rotate(rotation);
        this.X = point.X;
        this.Z = point.Y;
    }

    RotateYZ(rotation) {
        var point = new PointClass(this.Y, this.Z);
        point.Rotate(rotation);
        this.Y = point.X;
        this.Z = point.Y;
    }

    RotateXQ(rotation) {
        var point = new PointClass(this.X, this.Q);
        point.Rotate(rotation);
        this.X = point.X;
        this.Q = point.Y;
    }

    RotateYQ(rotation) {
        var point = new PointClass(this.Y, this.Q);
        point.Rotate(rotation);
        this.Y = point.X;
        this.Q = point.Y;
    }

    RotateZQ(rotation) {
        var point = new PointClass(this.Z, this.Q);
        point.Rotate(rotation);
        this.Z = point.X;
        this.Q = point.Y;
    }

    Clone() {
        var obj = new Point4dClass();
        obj.X = this.X;
        obj.Y = this.Y;
        obj.Z = this.Z;
        obj.Q = this.Q;
        return obj;
    }

    toString() {
        var mes = "Point: " +
            this.X.toFixed(1) + ", " +
```

```
            this.Y.toFixed(1) + ", " +
            this.Z.toFixed(1) + ", " +
            this.Q.toFixed(1);
        return mes;
    }
}

////////////////////////////////////////////////////////////////////////

class Corner4dClass {
    constructor(id, toIds) {
        this.Id = (id == undefined ? -1 : id);
        this.ToIds = (toIds == undefined ? [] : toIds);
        this.Point = new Point4dClass();
    }

    Clone() {
        var obj = new Corner4dClass();
        obj.Id = this.Id;
        obj.ToIds = Array.from(this.ToIds);
        obj.Point = this.Point.Clone();
        return obj;
    }

    toString() {
        var mes = "\n Corner " + this.Id +
            " to c.s " +
            this.ToIds[0] + ", " +
            this.ToIds[1] + ", " +
            this.ToIds[2] + ", " +
            this.ToIds[3] + ". " +
            this.Point;
        return mes;
    }
}

////////////////////////////////////////////////////////////////////////

class Prism4dClass {
    constructor(name) {
        this.Name = name;

        this.Length = 100;
        this.Width = 100;
        this.Height = 100;
        this.Depth = 100;

        this.Corners = [];
        this.Corners[0] = new Corner4dClass(0, [1, 2, 4, 8]);
        this.Corners[1] = new Corner4dClass(1, [0, 3, 5, 9]);
        this.Corners[2] = new Corner4dClass(2, [0, 3, 6, 10]);
        this.Corners[3] = new Corner4dClass(3, [1, 2, 7, 11]);
        this.Corners[4] = new Corner4dClass(4, [0, 5, 6, 12]);
        this.Corners[5] = new Corner4dClass(5, [1, 4, 7, 13]);
        this.Corners[6] = new Corner4dClass(6, [2, 4, 7, 14]);
        this.Corners[7] = new Corner4dClass(7, [3, 5, 6, 15]);
        this.Corners[8] = new Corner4dClass(8, [0, 9, 10, 12]);
        this.Corners[9] = new Corner4dClass(9, [1, 8, 11, 13]);
        this.Corners[10] = new Corner4dClass(10, [2, 8, 11, 14]);
        this.Corners[11] = new Corner4dClass(11, [3, 9, 10, 15]);
        this.Corners[12] = new Corner4dClass(12, [4, 8, 13, 14]);
        this.Corners[13] = new Corner4dClass(13, [5, 9, 12, 15]);
        this.Corners[14] = new Corner4dClass(14, [6, 10, 12, 15]);
        this.Corners[15] = new Corner4dClass(15, [7, 11, 13, 14]);

        this.Corners[0].Point = new Point4dClass(0, 0, 0, 0);
        this.Corners[1].Point = new Point4dClass(this.Length, 0, 0, 0);
        this.Corners[2].Point = new Point4dClass(0, this.Width, 0, 0);
        this.Corners[3].Point = new Point4dClass(this.Length, this.Width, 0, 0);
```

```
        this.Corners[4].Point = new Point4dClass(0, 0, this.Height, 0);
        this.Corners[5].Point = new Point4dClass(this.Length, 0, this.Height, 0);
        this.Corners[6].Point = new Point4dClass(0, this.Width, this.Height, 0);
        this.Corners[7].Point = new Point4dClass(this.Length, this.Width, this.Height, 0);
        this.Corners[8].Point = new Point4dClass(0, 0, 0, this.Depth);
        this.Corners[9].Point = new Point4dClass(this.Length, 0, 0, this.Depth);
        this.Corners[10].Point = new Point4dClass(0, this.Width, 0, this.Depth);
        this.Corners[11].Point = new Point4dClass(this.Length, this.Width, 0, this.Depth);
        this.Corners[12].Point = new Point4dClass(0, 0, this.Height, this.Depth);
        this.Corners[13].Point = new Point4dClass(this.Length, 0, this.Height, this.Depth);
        this.Corners[14].Point = new Point4dClass(0, this.Width, this.Height, this.Depth);
        this.Corners[15].Point = new Point4dClass(this.Length, this.Width, this.Height,
this.Depth);

        var point = this.Corners[15].Point;
        this.Shift(-point.X / 2, -point.Y / 2, -point.Z / 2, -point.Q / 2);

        this.Faces = [];
        // xy..
        this.Faces[0] = new FaceClass(0, [0, 1, 3, 2]);
        this.Faces[1] = new FaceClass(1, [4, 5, 7, 6]);
        this.Faces[2] = new FaceClass(2, [8, 9, 11, 10]);
        this.Faces[3] = new FaceClass(3, [12, 13, 15, 14]);
        // xz..
        this.Faces[4] = new FaceClass(4, [0, 1, 5, 4]);
        this.Faces[5] = new FaceClass(5, [2, 3, 7, 6]);
        this.Faces[6] = new FaceClass(6, [8, 9, 13, 12]);
        this.Faces[7] = new FaceClass(7, [10, 11, 15, 14]);
        // yz..
        this.Faces[8] = new FaceClass(8, [0, 2, 6, 4]);
        this.Faces[9] = new FaceClass(9, [1, 3, 7, 5]);
        this.Faces[10] = new FaceClass(10, [8, 10, 14, 12]);
        this.Faces[11] = new FaceClass(11, [9, 11, 15, 13]);
        // xq..
        this.Faces[12] = new FaceClass(12, [0, 1, 9, 8]);
        this.Faces[13] = new FaceClass(13, [2, 3, 11, 10]);
        this.Faces[14] = new FaceClass(14, [4, 5, 13, 12]);
        this.Faces[15] = new FaceClass(15, [6, 7, 15, 14]);
        // yq..
        this.Faces[16] = new FaceClass(16, [0, 2, 10, 8]);
        this.Faces[17] = new FaceClass(17, [1, 3, 11, 9]);
        this.Faces[18] = new FaceClass(18, [4, 6, 14, 12]);
        this.Faces[19] = new FaceClass(19, [5, 7, 15, 13]);
        // zq..
        this.Faces[20] = new FaceClass(20, [0, 4, 12, 8]);
        this.Faces[21] = new FaceClass(21, [1, 5, 13, 9]);
        this.Faces[22] = new FaceClass(22, [2, 6, 14, 10]);
        this.Faces[23] = new FaceClass(23, [3, 7, 15, 11]);

        this.Rooms = [];
        // xyz..
        this.Rooms[0] = new RoomClass(0, [0, 1, 4, 5, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7]);
        this.Rooms[1] = new RoomClass(1, [2, 3, 6, 7, 10, 11], [8, 9, 10, 11, 12, 13, 14, 15]);
        // xyq..
        this.Rooms[2] = new RoomClass(2, [0, 2, 12, 13, 16, 17], [0, 1, 2, 3, 8, 9, 10, 11]);
        this.Rooms[3] = new RoomClass(3, [1, 3, 14, 15, 18, 19], [4, 5, 6, 7, 12, 13, 14, 15]);
        // xzq..
        this.Rooms[4] = new RoomClass(4, [4, 6, 12, 14, 20, 21], [0, 1, 4, 5, 8, 9, 12, 13]);
        this.Rooms[5] = new RoomClass(5, [5, 7, 13, 15, 22, 23], [2, 3, 6, 7, 10, 11, 14, 15]);
        // yzq..
        this.Rooms[6] = new RoomClass(6, [8, 10, 16, 18, 20, 22], [0, 2, 4, 6, 8, 10, 12, 14]);
        this.Rooms[7] = new RoomClass(7, [9, 11, 17, 19, 21, 23], [1, 3, 5, 7, 9, 11, 13, 15]);

        this.Palette = new PaletteClass();
    }

    Shift(dX, dY, dZ, dQ) {
        this.Corners.forEach(corner => {
            corner.Point.X += dX;
```

```
        corner.Point.Y += dY;
        corner.Point.Z += dZ;
        corner.Point.Q += dQ;
    });
}

Rotate(XYturn, XZturn, YZturn, XQturn, YQturn, ZQturn) {
    this.Corners.forEach(corner => {
        corner.Point.RotateXY(XYturn);
        corner.Point.RotateXZ(XZturn);
        corner.Point.RotateYZ(YZturn);
        corner.Point.RotateXQ(XQturn);
        corner.Point.RotateYQ(YQturn);
        corner.Point.RotateZQ(ZQturn);
    });
}

ColorGroup(index) {
    return (index % 8);
}

ColorIndex(index, contrast) {
    contrast = (contrast == undefined ? 100 : contrast);
    const maxIndex = 15;
    var plainIndex = (index * 2 % (maxIndex + 1));
    var distance = Math.round((maxIndex - plainIndex) * contrast / 100);
    var newIndex = maxIndex - distance;
    return newIndex;
}

DrawRooms(ctx, plane, paintlist, contrast) {
    paintlist = (paintlist == undefined ? [0, 1, 2, 3, 4, 5, 6, 7] : paintlist);
    var index, room, colorGroup;
    paintlist.forEach((roomId) => {
        index = roomId;
        room = this.Rooms[index];
        colorGroup = this.ColorGroup(index);
        this.DrawRoom(ctx, plane, room, colorGroup, contrast);
    });
}

DrawRoom(ctx, plane, room, colorGroup, contrast) {
    var face, colorIndex, color;
    room.FaceIds.forEach((faceId, index) => {
        face = this.Faces[faceId];
        colorIndex = this.ColorIndex(index, contrast);
        color = this.Palette.Colors[colorIndex][colorGroup];
        this.DrawFace(ctx, plane, face, color);
    });
}

DrawFace(ctx, plane, face, color) {
    ctx.fillStyle = color;
    ctx.beginPath();
    face.Set();
    var corner, toCorner;
    for (let index = 0; index < face.Count; index++) {
        corner = this.Corners[face.Value];
        switch (plane) {
            case 0:
                if (face.Index == 0) ctx.moveTo(corner.Point.X, corner.Point.Y);
                break;
            case 1:
                if (face.Index == 0) ctx.moveTo(corner.Point.X, corner.Point.Z);
                break;
            case 2:
                if (face.Index == 0) ctx.moveTo(corner.Point.Y, corner.Point.Z);
                break;
            case 3:
```

```
                if (face.Index == 0) ctx.moveTo(corner.Point.X, corner.Point.Q);
                break;
            case 4:
                if (face.Index == 0) ctx.moveTo(corner.Point.Y, corner.Point.Q);
                break;
            case 5:
                if (face.Index == 0) ctx.moveTo(corner.Point.Z, corner.Point.Q);
                break;
            default:
                break;
        }
        face.Next();
        toCorner = this.Corners[face.Value];
        switch (plane) {
            case 0:
                ctx.lineTo(toCorner.Point.X, toCorner.Point.Y);
                break;
            case 1:
                ctx.lineTo(toCorner.Point.X, toCorner.Point.Z);
                break;
            case 2:
                ctx.lineTo(toCorner.Point.Y, toCorner.Point.Z);
                break;
            case 3:
                ctx.lineTo(toCorner.Point.X, toCorner.Point.Q);
                break;
            case 4:
                ctx.lineTo(toCorner.Point.Y, toCorner.Point.Q);
                break;
            case 5:
                ctx.lineTo(toCorner.Point.Z, toCorner.Point.Q);
                break;
            default:
                break;
        }
    }
    ctx.fill();
}

DrawLines(ctx, plane) {
    this.Corners.forEach(corner => {
        corner.ToIds.forEach(toId => {
            if (corner.Id < toId) {
                var toCorner = this.Corners[toId];
                switch (plane) {
                    case 0:
                        ctx.moveTo(corner.Point.X, corner.Point.Y);
                        ctx.lineTo(toCorner.Point.X, toCorner.Point.Y);
                        break;
                    case 1:
                        ctx.moveTo(corner.Point.X, corner.Point.Z);
                        ctx.lineTo(toCorner.Point.X, toCorner.Point.Z);
                        break;
                    case 2:
                        ctx.moveTo(corner.Point.Y, corner.Point.Z);
                        ctx.lineTo(toCorner.Point.Y, toCorner.Point.Z);
                        break;
                    case 3:
                        ctx.moveTo(corner.Point.X, corner.Point.Q);
                        ctx.lineTo(toCorner.Point.X, toCorner.Point.Q);
                        break;
                    case 4:
                        ctx.moveTo(corner.Point.Y, corner.Point.Q);
                        ctx.lineTo(toCorner.Point.Y, toCorner.Point.Q);
                        break;
                    case 5:
                        ctx.moveTo(corner.Point.Z, corner.Point.Q);
                        ctx.lineTo(toCorner.Point.Z, toCorner.Point.Q);
                        break;
```

```
                        default:
                            break;
                    }
                }
            });
            ctx.stroke();
        });
    }
}
```

util.js

```
/*** prism util.js ***/

////////////////////////////////////////////////////////////////////

class Lib {
    //Convert radians to degrees..
    static Degrees(radians) { return (radians * 180 / Math.PI); }

    //Convert degrees to radians..
    static Radians(degrees) { return (degrees * Math.PI / 180); }
}

////////////////////////////////////////////////////////////////////

class RingClass {
    constructor(Array) {
        this.Array = (Array == undefined ? [] : Array);
        this.Count = this.Array.length;
        this.Index = 0;
        this.Value = this.Array[this.Index];
    }

    Set(Index) {
        Index = (Index == undefined ? 0 : Index);
        if (Index < 0 || Index >= this.Count) return;
        this.Index = Index;
        this.Value = this.Array[this.Index];
    }

    Next() {
        this.Index++;
        if (this.Index >= this.Count) this.Index = 0;
        this.Value = this.Array[this.Index];
    }

    Previous() {
        this.Index--;
        if (this.Index <= -1) this.Index = this.Count - 1;
        this.Value = this.Array[this.Index];
    }

    Clone() {
        var obj = new RingClass(Array.from(this.Array));
        return obj;
    }

    toString() {
        var mes = "Ring: " + this.Array +
            ", Ct=" + this.Count +
            ", Ix=" + this.Index +
            ", Vl=" + this.Value;
        return mes;
    }
}

////////////////////////////////////////////////////////////////////

class PointClass {
    constructor(X, Y) {
        this.X = (X == undefined ? 0 : X);
        this.Y = (Y == undefined ? 0 : Y);
    }

    Rotate(Rotation) {
        Rotation = (Rotation == undefined ? 0 : Rotation);
        var angle = Lib.Radians(Rotation) * (-1);
```

```
        var tempX = this.X * Math.cos(angle) - this.Y * Math.sin(angle);
        var tempY = this.X * Math.sin(angle) + this.Y * Math.cos(angle);
        this.X = tempX;
        this.Y = tempY;
    }

    Clone() {
        var obj = new PointClass(this.X, this.Y);
        return obj;
    }

    toString() {
        var mes = "Point: " +
            this.X.toFixed(1) + ", " +
            this.Y.toFixed(1);
        return mes;
    }
}

//////////////////////////////////////////////////////////////////////

class FaceClass extends RingClass {
    constructor(Id, CornerIds) {
        super(CornerIds);
        this.CornerIds = this.Array;
        this.Id = (Id == undefined ? 0 : Id);
    }

    Clone() {
        var obj = new FaceClass(this.Id, Array.from(this.CornerIds));
        return obj;
    }

    toString() {
        var mes = "\n Face " + this.Id +
            ", corners " +
            this.CornerIds[0] + ", " +
            this.CornerIds[1] + ", " +
            this.CornerIds[2] + ", " +
            this.CornerIds[3] + ". ";
        return mes;
    }
}

//////////////////////////////////////////////////////////////////////

class RoomClass {
    constructor(Id, FaceIds, CornerIds) {
        this.Id = (Id == undefined ? 0 : Id);
        this.FaceIds = (FaceIds == undefined ? [] : FaceIds);
        this.CornerIds = (CornerIds == undefined ? [] : CornerIds);
    }

    Clone() {
        var obj = new RoomClass(this.Id, Array.from(this.FaceIds), Array.from(this.CornerIds));
        return obj;
    }

    toString() {
        var mes = "\n Room " + this.Id +
            ", faces " +
            this.FaceIds[0] + ", " +
            this.FaceIds[1] + ", " +
            this.FaceIds[2] + ", " +
            this.FaceIds[3] + ", " +
            this.FaceIds[4] + ", " +
            this.FaceIds[5] +
            ", corners " +
            this.CornerIds[0] + ", " +
```

```
            this.CornerIds[1] + ", " +
            this.CornerIds[2] + ", " +
            this.CornerIds[3] + ", " +
            this.CornerIds[4] + ", " +
            this.CornerIds[5] + ", " +
            this.CornerIds[6] + ", " +
            this.CornerIds[7] + ". ";
        return mes;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////
/

class PaletteClass {
    constructor() {
        //     blue,      green,     cyan,      violet,    red,      magenta,    orange,    yellow
        this.Colors = [
            ["#0000FF", "#00FF00", "#00FFFF", "#8000FF", "#FF0000", "#FF00FF", "#FF8000",
"#FFFF00"],
            ["#1111FF", "#11FF11", "#11FFFF", "#8811FF", "#FF1111", "#FF11FF", "#FF8811",
"#FFFF11"],
            ["#2222FF", "#22FF22", "#22FFFF", "#9022FF", "#FF2222", "#FF22FF", "#FF9022",
"#FFFF22"],
            ["#3333FF", "#33FF33", "#33FFFF", "#9833FF", "#FF3333", "#FF33FF", "#FF9833",
"#FFFF33"],
            ["#4444FF", "#44FF44", "#44FFFF", "#A044FF", "#FF4444", "#FF44FF", "#FFA044",
"#FFFF44"],
            ["#5555FF", "#55FF55", "#55FFFF", "#A855FF", "#FF5555", "#FF55FF", "#FFA855",
"#FFFF55"],
            ["#6666FF", "#66FF66", "#66FFFF", "#B066FF", "#FF6666", "#FF66FF", "#FFB066",
"#FFFF66"],
            ["#7777FF", "#77FF77", "#77FFFF", "#B877FF", "#FF7777", "#FF77FF", "#FFB877",
"#FFFF77"],
            ["#8888FF", "#88FF88", "#88FFFF", "#C088FF", "#FF8888", "#FF88FF", "#FFC088",
"#FFFF88"],
            ["#9999FF", "#99FF99", "#99FFFF", "#C899FF", "#FF9999", "#FF99FF", "#FFC899",
"#FFFF99"],
            ["#AAAAFF", "#AAFFAA", "#AAFFFF", "#D0AAFF", "#FFAAAA", "#FFAAFF", "#FFD0AA",
"#FFFFAA"],
            ["#BBBBFF", "#BBFFBB", "#BBFFFF", "#D8BBFF", "#FFBBBB", "#FFBBFF", "#FFD8BB",
"#FFFFBB"],
            ["#CCCCFF", "#CCFFCC", "#CCFFFF", "#E0CCFF", "#FFCCCC", "#FFCCFF", "#FFE0CC",
"#FFFFCC"],
            ["#DDDDFF", "#DDFFDD", "#DDFFFF", "#E8DDFF", "#FFDDDD", "#FFDDFF", "#FFE8DD",
"#FFFFDD"],
            ["#EEEEFF", "#EEFFEE", "#EEFFFF", "#F0EEFF", "#FFEEEE", "#FFEEFF", "#FFF0EE",
"#FFFFEE"],
            ["#FFFFFF", "#FFFFFF", "#FFFFFF", "#FFFFFF", "#FFFFFF", "#FFFFFF", "#FFFFFF",
"#FFFFFF"]
        ];
    }
}
```