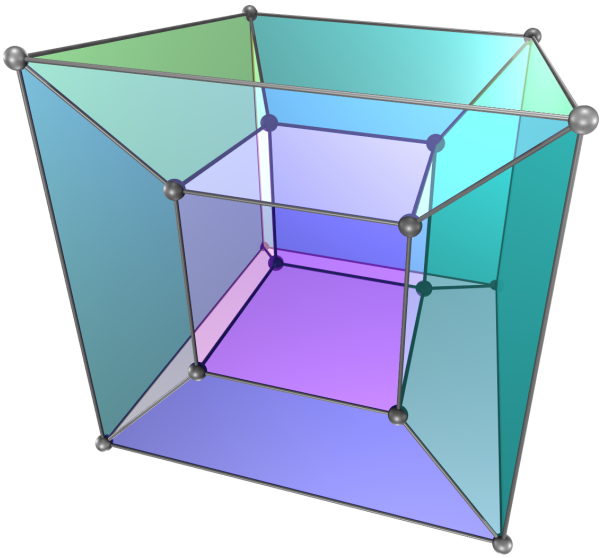


Hypercube

Interactive Website and Virtual Reality

Jörg J. Buchholz

October 6, 2021

1 Manual

1.1 Introduction

In this paper, we describe the utilization and genesis of an interactive website [1] you can use to display, rotate, and unfold (unravel) four-dimensional hypercubes (figure 1.1).

1.2 Website

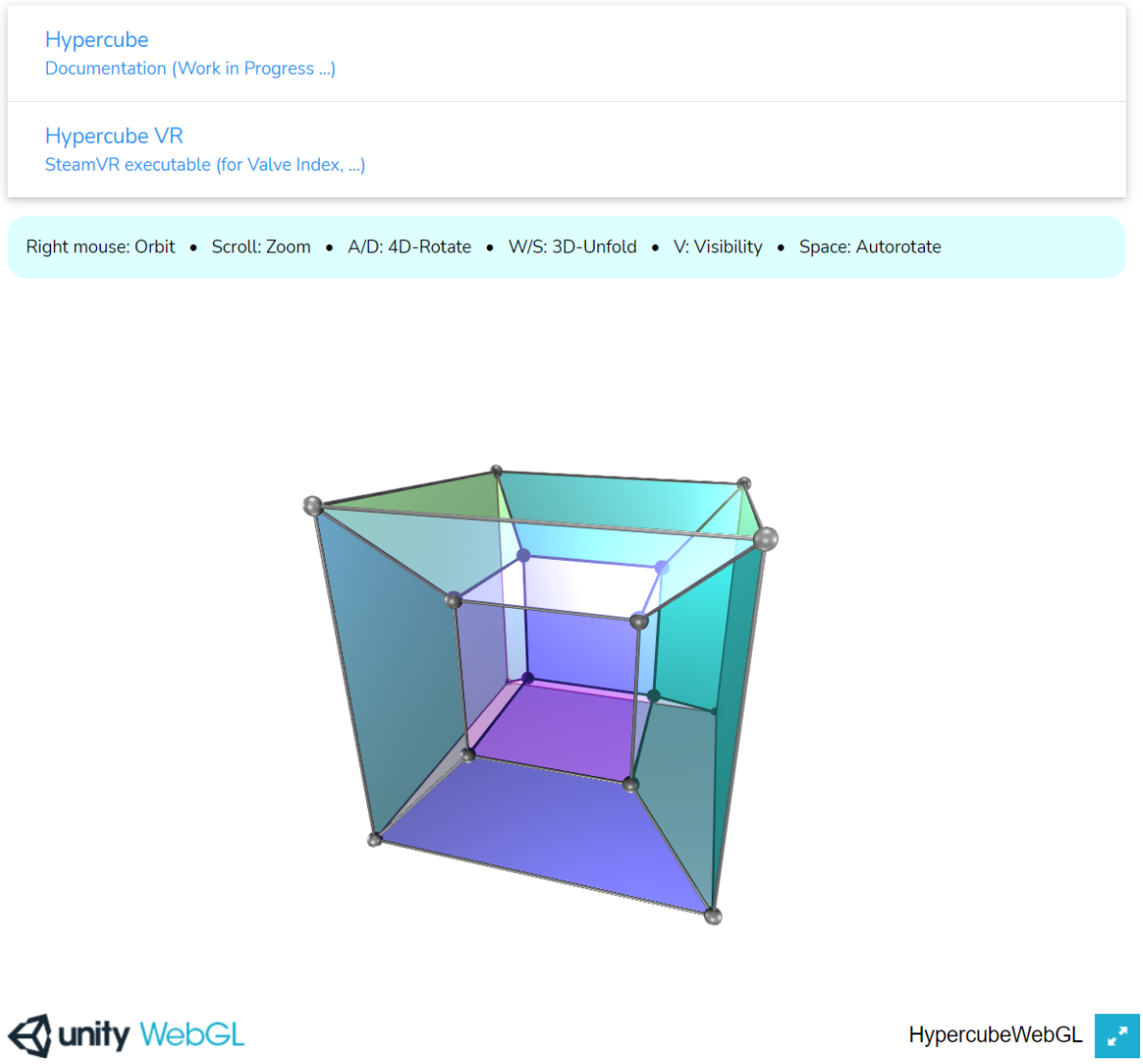


Figure 1.1: Hypercube website [1]

The website has been programmed in Unity [2] in C#, compiled for WebGL, and should run in every[1] current desktop[2] browser.

[1] Except for – who would have guessed – Internet Explorer which does not support WebAssembly.

[2] Unity says, mobile browsers are not supported but displays the scene nevertheless.

1.3 Mouse

Pressing the right mouse button, you can orbit the camera around the scenery (section 2.8). With the mouse wheel you can zoom in and out.

1.4 Rotation

The `A` and the `D` key rotate the hypercube (figure 1.2) about the x-y-plane in the four-dimensional hyperspace (section A.4.1); the `Space` key toggles an automatic rotation about this plane.

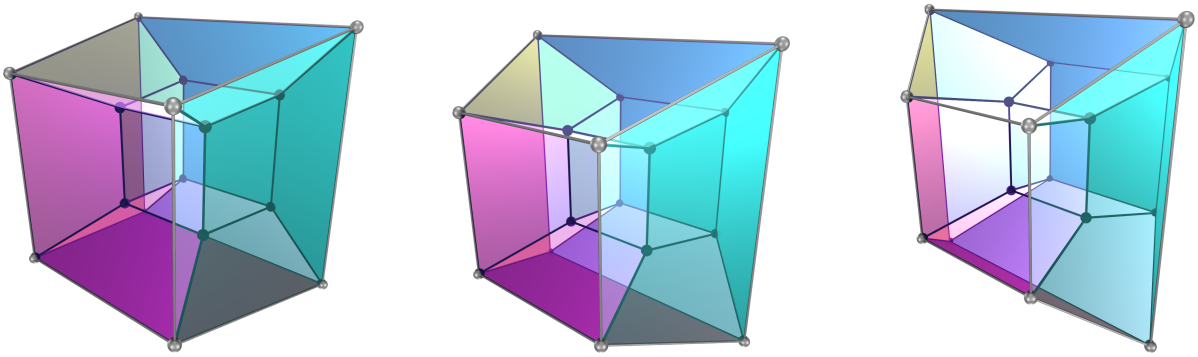


Figure 1.2: Rotation about the x-y-plane in 4D space

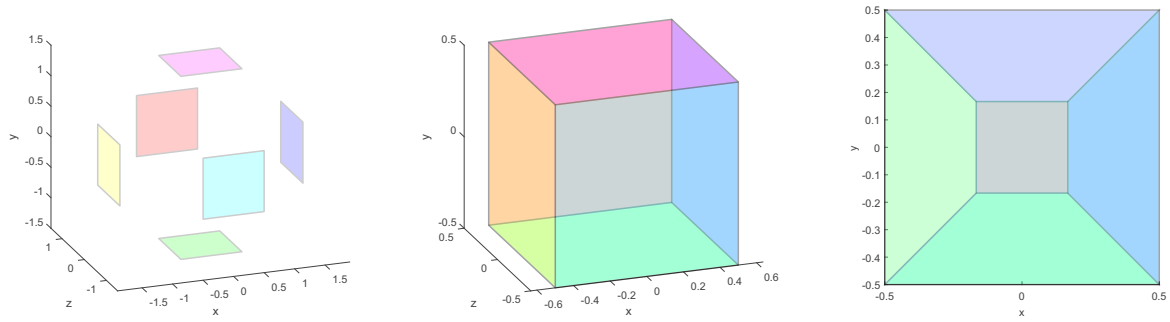
1.5 Unfolding (unraveling)

The `W` key unfolds (and the `S` key folds back) the four-dimensional hypercube into the three-dimensional space (figure 1.6).

For a better understanding of the unfolding of a 4D hypercube into the 3D space, we have to step one dimension down and discuss the unfolding of a 3D cube into the 2D plane.

1.5.1 Unfold a 3D cube into 2D

Figure 1.3 demonstrates three different ways to depict a 3D cube with transparent faces on a 2D "piece of paper".



(a) Exploded (b) Parallel projection (c) Perspective projection

Figure 1.3: 3D cube with transparent faces

The exploded view[3] in figure 1.3a highlights the actual transparent face colors (cyan front, red back, yellow left, ...) that cannot be recognized easily in the other projections:

E. g., if we look through the transparent[4] cyan front face at the transparent red back face in the parallel projection in figure 1.3b, we see a gray color. In additive color mixing, cyan (RGB: 0 1 1) light is a mixture of green (RGB: 0 1 0) light and blue (RGB: 0 0 1) light, adding up with the red (RGB: 1 0 0) light to white/gray (RGB: 1 1 1).

We can see the same back face as the small inner gray square in the perspective projection depicted in figure 1.3c. Since we directly look in positive z-direction[5], we see all other faces through the transparent cyan front face. The main property of perspective projection is that objects farther away are smaller; that's why the inner back square is actually drawn smaller than the outer front square. For the same reason, the other four square faces (left, right, top, bottom), which connect the front face to the back face, are depicted as trapezoids in figure 1.3c.

In preparation of the unfolding[6], we take a sharp virtual knife and cut the cube open along the seven edges indicated in figure 1.4.

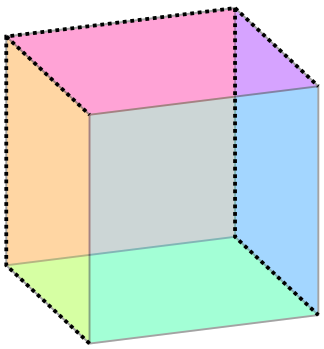


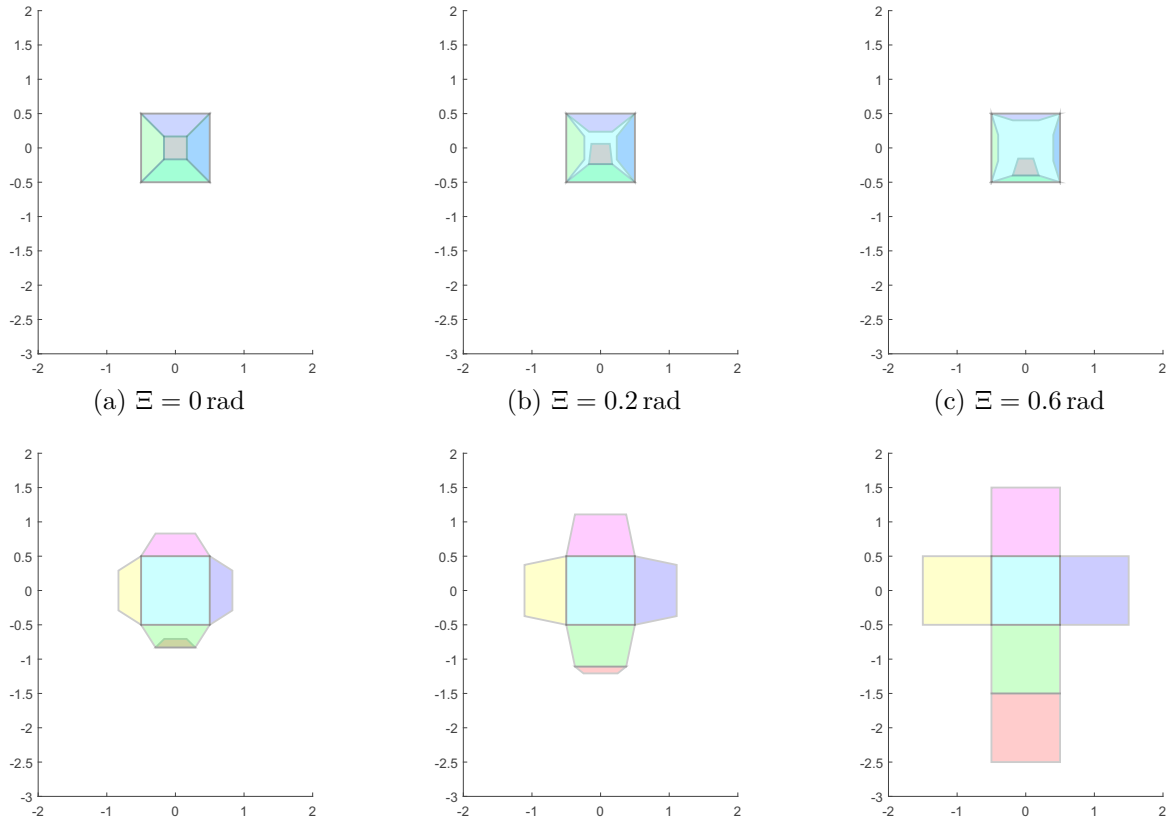
Figure 1.4: Cube cuts

[3] In fact, the exploded view is a parallel projection as well.
[4] All faces have an alpha value of 20 %, corresponding to an 80 % transparency.
[5] Remember that Unity uses a left-handed coordinate system.
[6] We can unfold a cube into 11 different nets [3].

To actually unfold, we rotate[7] all faces – except the front face – about their connecting edges for $\pm 90°$ in figure 1.5:

- The yellow left face rotates about the front left vertical y-axis for $-90°$.
- The blue right face rotates about the front right vertical y-axis for $+90°$.
- The magenta top face rotates about the front top horizontal x-axis for $-90°$.
- The green bottom face rotates about the front bottom horizontal x-axis for $+90°$.
- The red back face rotates about the back bottom horizontal x-axis for $+90°$.

The rotation of the red back face is a bit tricky: Unlike the other four faces, the back face is not connected to the fixed front face but to the bottom face which itself is rotating. Therefore, the back face actually performs a double rotation with a final angle of 180°.



(a) $\Xi = 0\,\text{rad}$ (b) $\Xi = 0.2\,\text{rad}$ (c) $\Xi = 0.6\,\text{rad}$

(d) $\Xi = 1.2\,\text{rad}$ (e) $\Xi = 1.4\,\text{rad}$ (f) $\Xi = \frac{\pi}{2}\,\text{rad}$

Figure 1.5: Unfolding a 3D cube into 2D

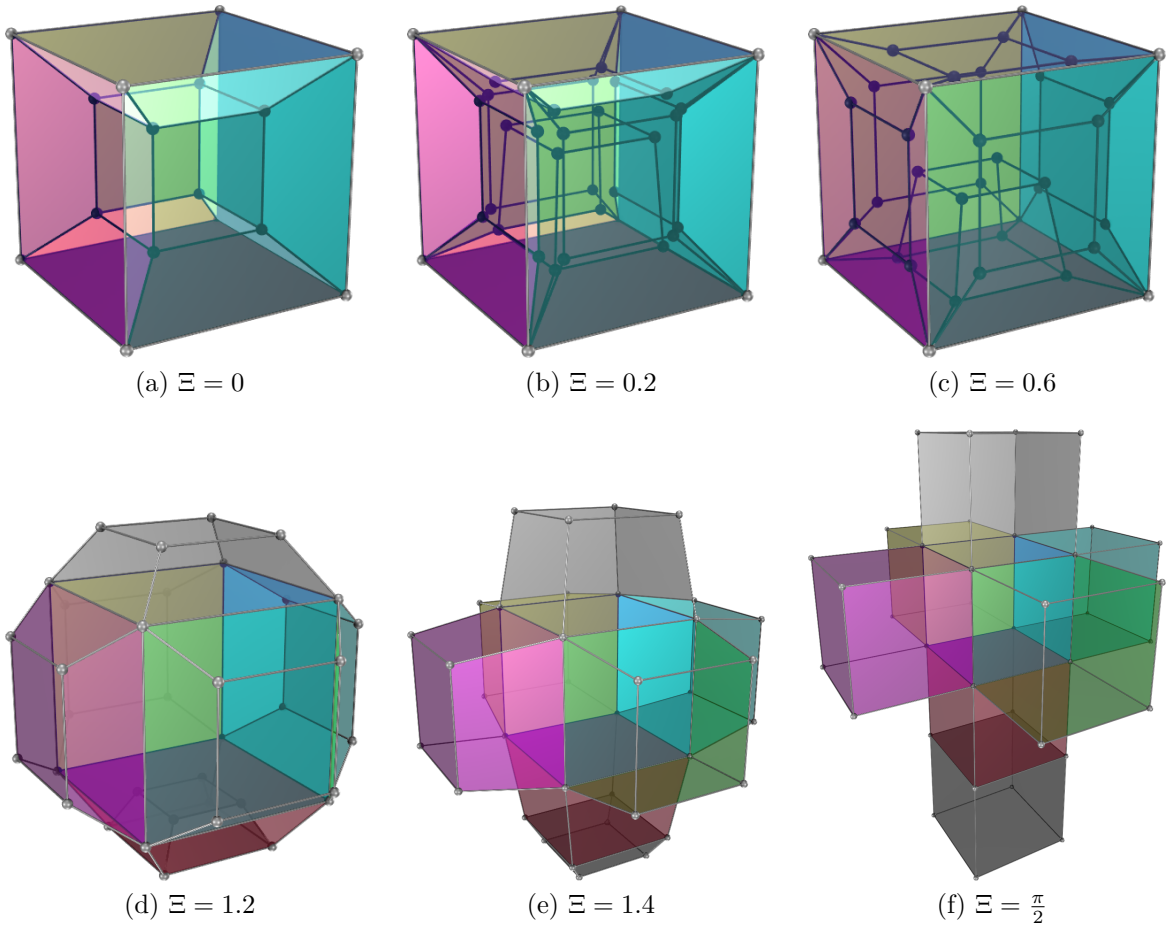
Starting with an initial rotation angle of $\Xi = 0\,\text{rad}$ in figure 1.5a, the back and side faces detach from each other in figure 1.5b. All faces are still visible through the transparent

[7] We explain the positive direction of rotation in figure 2.2.

cyan front face. In figure 1.5c, the side faces are almost invisible.[8] In figure 1.5d, the side faces have rotated "outside" the cube and are visible from "the outside". The back face is still concealed by the bottom face. In figure 1.5e, the back face is finally visible in its true red color underneath the green bottom face. The completely unfolded cube with a rotation angle of 90° is depicted in figure 1.5f.

1.5.2 Unfold a 4D hypercube into 3D

The unfolding[9] of a 4D hypercube into 3D in figure 1.6 looks very similar to the unfolding of a 3D cube into 2D in figure 1.5 and is described in detail in section 2.7.7.



(a) $\Xi = 0$ (b) $\Xi = 0.2$ (c) $\Xi = 0.6$

(d) $\Xi = 1.2$ (e) $\Xi = 1.4$ (f) $\Xi = \frac{\pi}{2}$

Figure 1.6: Unfolding a 4D hypercube into 3D space

In table 1.1, we compare the 3D case (figure 1.5) with the 4D case (figure 1.6).

[8] Depending on the parameters of the perspective projection (section 2.7.9) there is an intermediate angle with completely invisible side faces.

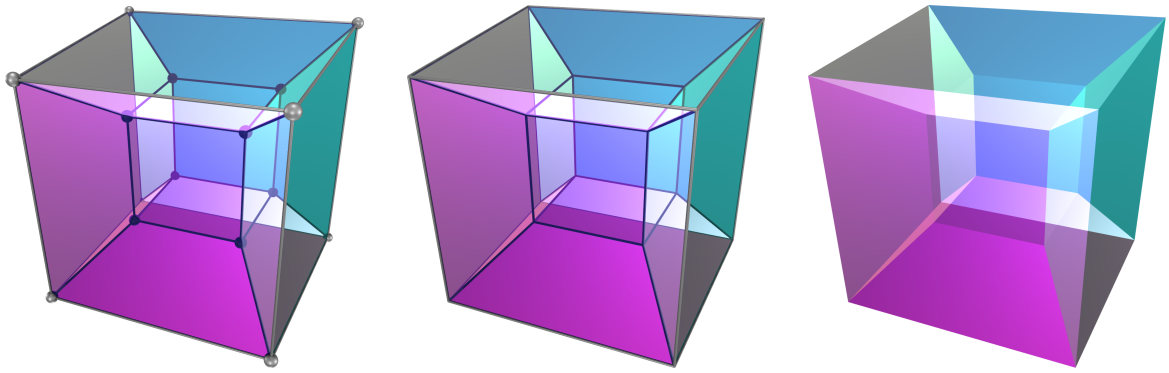
[9] The rotation angles Ξ in figure 1.6 are estimates.

3D case (figure 1.5)	**4D case (figure 1.6)**
The front face (with all of its edges) is fixed.	The outer cube (with all of its faces) is fixed.
Four side faces (depicted as trapezoids) connect the front face to the back face.	Six "side" cubes (depicted as square frustra) connect the outer cube to the inner cube.
Faces are connected to each other via their edges.	Cubes are connected to each other via their faces.
The connecting edges are the rotational axes.	The connecting faces are the rotational planes (appendix A).
Before unfolding, we cut the cube along seven of its edges, partly separating the faces.	Before unfolding, we cut the hypercube along 17 of its faces, partly separating the cubes.
Each face can now freely rotate about its fixed edge.	Each cube can now freely rotate about its fixed face.
The back face is not directly connected to the fixed front face but only to the bottom face. Therefore, the back face has to do a double rotation: together with the bottom face and about the edge connecting it to the bottom face.	The inner cube is not directly connected to the fixed outer cube but only to the bottom cube. Therefore, the inner cube has to do a double rotation: together with the bottom cube and about the face connecting it to the bottom cube.
In figure 1.5b, the faces detach from each other and start rotating.	In figure 1.6b, the cubes detach from each other and start rotating.
In figure 1.5d, the side faces have rotated "outside" the cube and are visible from the front. The back face is still concealed by the bottom face.	In figure 1.6d, the side cubes have rotated "through" the outer cube and are visible from the outside. The inner cube is still concealed by the "bottom" cube.
In figure 1.5e, the back face is finally visible underneath the bottom face.	In figure 1.6e, the inner cube is finally visible underneath the "bottom" cube.
The completely unfolded cube is depicted in figure 1.5f.	The completely unfolded hypercube is depicted in figure 1.6f [4].

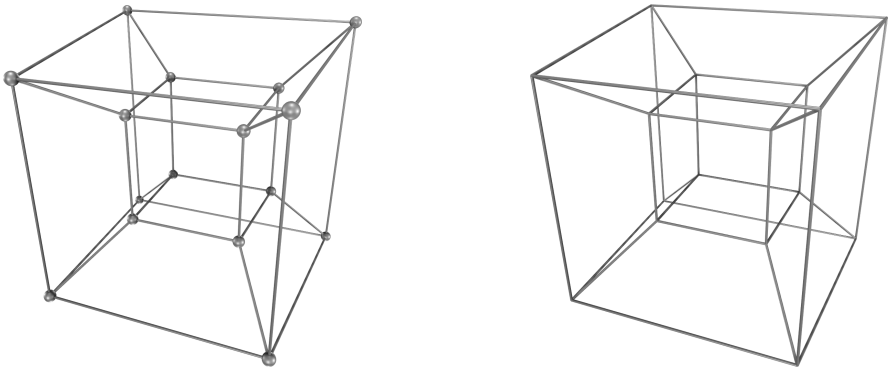
Table 1.1: Comparison of the 3D case (figure 1.5) with the 4D case (figure 1.6)

1.6 Visibility

The `V` key cycles the visibility of transparent cubes, vertex spheres, and edge cylinders (figure 1.7).



(a) Cubes, spheres, and cylinders (b) Cubes and cylinders (c) Cubes



(d) Spheres and cylinders (e) Cylinders

Figure 1.7: The `V` key cycles the visibility of cubes, spheres, and cylinders.

1.7 Genesis

In this chapter, we explain how we can create an $(n + 1)$-dimensional hypercube by moving an n-dimensional starting object towards an n-dimensional end object in a new direction of the $(n + 1)$-dimensional hyperspace, connecting the $(n - 1)$-dimensional object components by n-dimensional connection objects.

In the beginning, there was a point. And the point was a zero-dimensional hypercube (figure 1.8).

Figure 1.8: The point, a zero-dimensional hypercube

If we move the green 0D starting point in figure 1.9 in positive x-direction, to the right, towards the red 0D end point, we obtain a blue one-dimensional hypercube (aka. a line segment).

Figure 1.9: The line segment, a one-dimensional hypercube

If we move the green bottom 1D starting line segment in figure 1.10 in positive y-direction, up, towards the red top 1D end line segment, we obtain a two-dimensional hypercube (aka. a square). The 0D vertices of the starting and end line segments are connected to each other by two blue 1D connecting line segments. All line segments are the edges of the square.

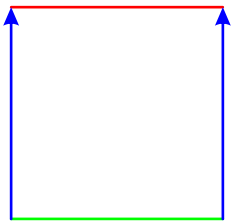


Figure 1.10: The square, a two-dimensional hypercube

If we move the green back 2D starting square in figure 1.11 in positive z-direction, backwards, towards the red back 2D square, we obtain a three-dimensional hypercube (aka. a cube). Again, the 0D vertices of the starting and end squares are connected to each other by four blue 1D connecting line segments. Additionally, the 1D edges of the starting and end squares are connected to each other by four 2D connecting squares[10]: the left square, the right square, the bottom square, and the top square in figure 1.11. All squares are the faces of the cube.

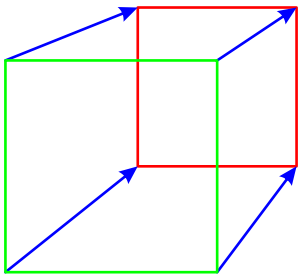


Figure 1.11: The cube, a three-dimensional hypercube

If we move the green outer 3D starting cube in figure 1.12 in positive w-direction, kata, towards the red inner 3D cube, we obtain a four-dimensional hypercube (aka. a

[10]Due to the perspective projection, the connecting squares look like trapezoids in figure 1.11.

tesseract). Again, the 0D vertices of the starting and end cubes are connected to each other by eight blue 1D connecting line segments. Again, the 1D edges of the starting and end cubes are connected to each other by twelve 2D connecting squares. Additionally, the 2D faces of the starting and end cubes are connected to each other by six 3D cubes[11]: the left cube, the right cube, the bottom cube, the top cube, the front cube, and the back cube in figure 1.12. All cubes are the hyperfaces of the hypercube.

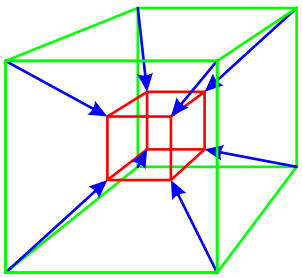


Figure 1.12: The tesseract, a four-dimensional hypercube

[11]Due to the perspective projection, the connecting cubes look like square frustra in figure 1.12.

2 Under the hood

2.1 Coordinate system

UNITY uses a left-handed coordinate system:

- The thumb of your left hand points to your right (red x-axis in figure 2.1).
- The index finger of your left hand points up (green y-axis in figure 2.1).
- The middle finger of your left hand points away from you (blue z-axis in figure 2.1).

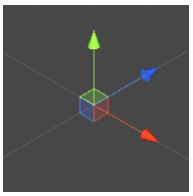


Figure 2.1: Left-handed coordinate system

The positive direction of a rotation is indicated by the curved fingers of your left hand if your left thumb points towards the positive axis of rotation (figure 2.2).

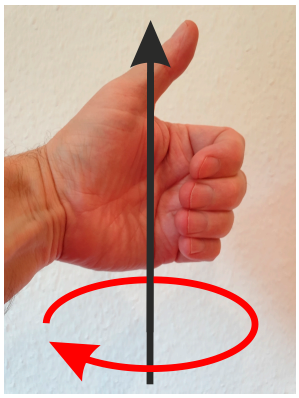


Figure 2.2: Curved fingers of left hand indicate positive rotation.

- Align your left thumb with the positive red x-axis in figure 2.1. The curved fingers of your left hand indicate that a positive rotation (of 90°) rotates the green y-axis into the blue z-axis.

- Align your left thumb with the positive green y-axis in figure 2.1. The curved fingers of your left hand indicate that a positive rotation (of 90°) rotates the blue z-axis into the red x-axis (figure 2.2).
- Align your left thumb with the positive blue z-axis in figure 2.1. The curved fingers of your left hand indicate that a positive rotation (of 90°) rotates the red x-axis into the green y-axis.

2.2 MyHypercube prefab

Besides the Camera (section 2.8) and the Lights (section 2.9), the Hypercube scene contains just one GameObject: MyHypercube (figure 2.3). The blue color indicates that MyHypercube is a prefab that we constructed before we used it in the scene.

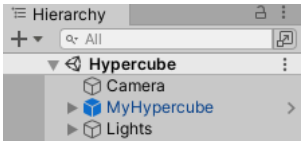


Figure 2.3: MyHypercube prefab

The MyHypercube prefab is made up of eight MyCCS prefabs (section 2.3) of different colors (figure 2.4).

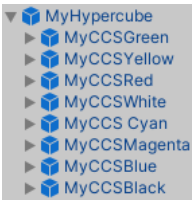


Figure 2.4: MyHypercube consists of eight MyCCS prefabs.

We attach a script (section 2.7) to the MyHypercube GameObject (figure 2.5) that is responsible for all the user interaction with the hypercube.

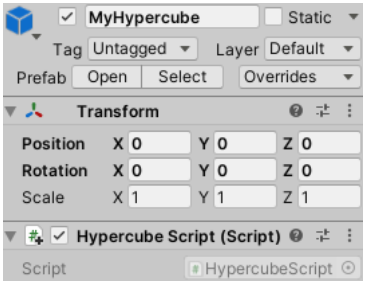
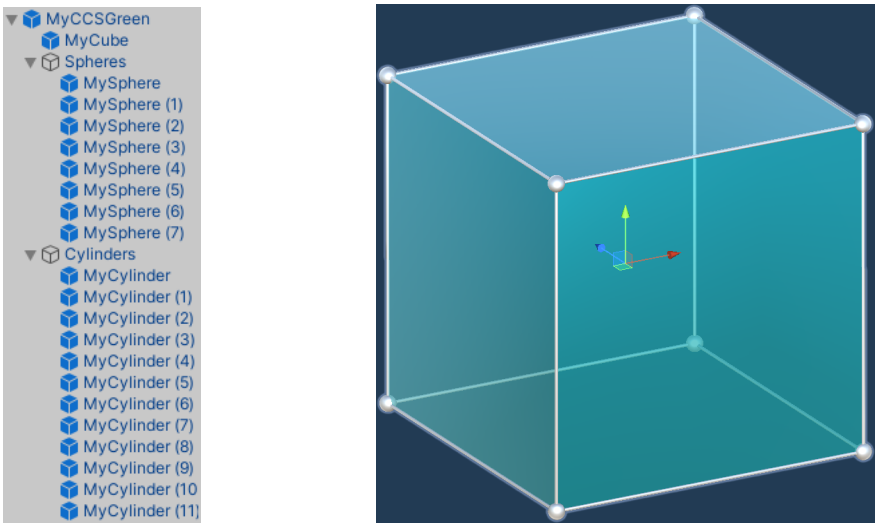


Figure 2.5: HypercubeScript

2.3 MyCCS prefab

The CCS in MyCCS stands for Cube, Cylinders, and Spheres. As depicted in figure 2.6, a MyCCS prefab consist of one MyCube prefab (section 2.4), eight MySphere prefabs (section 2.5), and twelve MyCylinder prefabs (section 2.6).



(a) MyCCS children (b) MyCCS object

Figure 2.6: A MyCCS prefab consists of a cube, spheres, and cylinders.

2.4 MyCube prefab

As in most 3D programs, we create meshes in Unity by defining vertices and triangles that use three vertices each. If we assign a material (with a color) to the mesh, we create a surface. Surfaces then make up objects.

Unity pursues an interesting concept regarding edge smoothing: If two triangles share (reuse) the same two vertices, the corresponding edge is smooth; if both triangles use their own vertices (even if these have the same coordinates), the edge is sharp.

Therefore, we need 24 vertices to define a cube with sharp edges: The cube itself has eight vertices but every vertex belongs to three independent triangular faces. Or, as seen from a different point of view: Each of the six quadratic faces of the cube uses is own four vertices.

Unity can create a few 3D primitives by itself:

- Cube
- Sphere
- Capsule

- `Cylinder`
- `Plane`
- `Quad`

Therefore, we could have used the `Cube` primitive instead of our own `MyCube` prefab. Unfortunately, UNITY's `Cube` primitive uses some unorthodox way of ordering its 24 mesh vertices (figure 2.7).

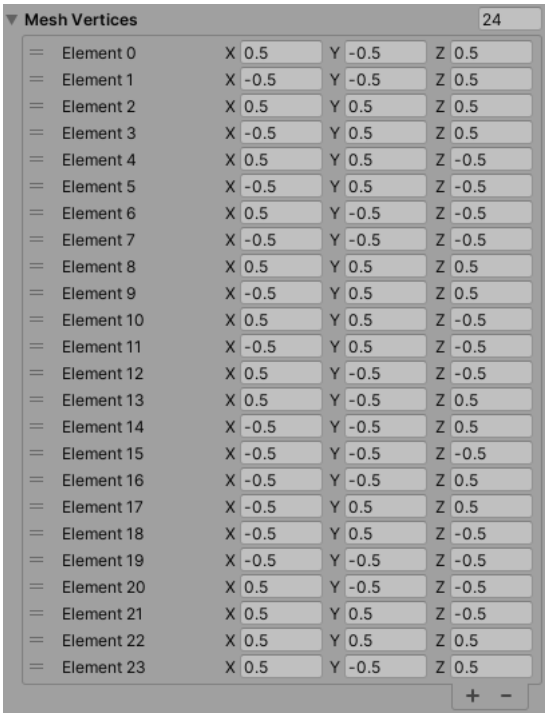


Figure 2.7: Mesh vertices of UNITY's `Cube` primitive

While we can clearly recognize the structure UNITY uses in figure 2.7, we prefer the direct repetition of the eight primary cube vertices depicted in figure 2.8.

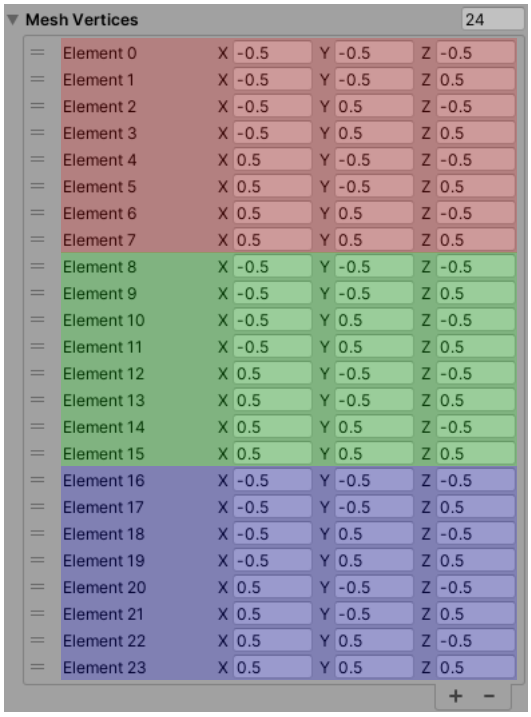


Figure 2.8: Mesh vertices of `MyCube` prefab

The red, green, and blue sections in figure 2.8 contain the same vertices in the same order. Since the cube has an edge length of 1.0 and is centered around the origin, all vertex components are either $+0.5$ or -0.5. Therefore, we can use a simple sign table (table 2.1) to display the structure of our primary cube vertices:

	0	1	2	3	4	5	6	7
x	−	−	−	−	+	+	+	+
y	−	−	+	+	−	−	+	+
z	−	+	−	+	−	+	−	+

Table 2.1: `MyCube` vertex signs

Utilizing the left-handed coordinate system of section 2.1, the first four vertices are located "on the left face" (negative x-component), the first two vertices (and the fifth and sixth vertices) are located on the bottom face (negative y-component), and the first vertex (and every vertex with an even index) is located on the front face (negative z-component) of the cube. Just like the systematic construction of the hypercube vertices in table 2.2, we could create the cube vertices by three nested `for`-loops (figure 2.9).

```
for (int ix = 0; ix <= 1; ix++)
{
  for (int iy = 0; iy <= 1; iy++)
  {
    for (int iz = 0; iz <= 1; iz++)
    {
      cubeVertices[4 * ix + 2 * iy + iz] =
          new Vector4(2 * ix - 1, 2 * iy - 1, 2 * iz - 1)
          * 0.5f;
    }
  }
}
```

Figure 2.9: Create cube vertices by three nested for-loops

The exploded view of figure 2.10 indicates which vertices we use for which faces of the cube. Additionally, figure 2.10 defines which vertices we use for which triangles.

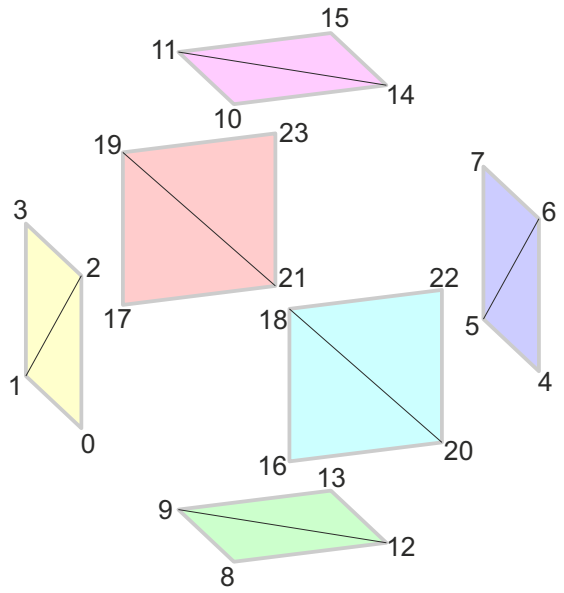


Figure 2.10: Vertex indices and triangle definitions of MyCube prefab

In Unity, a triangle is only rendered if its vertices are defined in a clockwise order. Therefore, the two triangles forming the (yellow) left face of the cube in figure 2.10 have to have the clockwise order $0 \rightarrow 1 \rightarrow 2$ and $1 \rightarrow 3 \rightarrow 2$ in order to be rendered if we look at the cube from the outside (from the left). If we looked at the yellow face from the inside of the cube (from the right) we would not see anything at all. The choice of the first triangle vertex does not matter; we could have chosen $1 \rightarrow 2 \rightarrow 0$ or $2 \rightarrow 0 \rightarrow 1$ for the first triangle. The vertex indices of all triangles (two triangles per cube face) are listed in figure 2.11 with a background color according to figure 2.10.

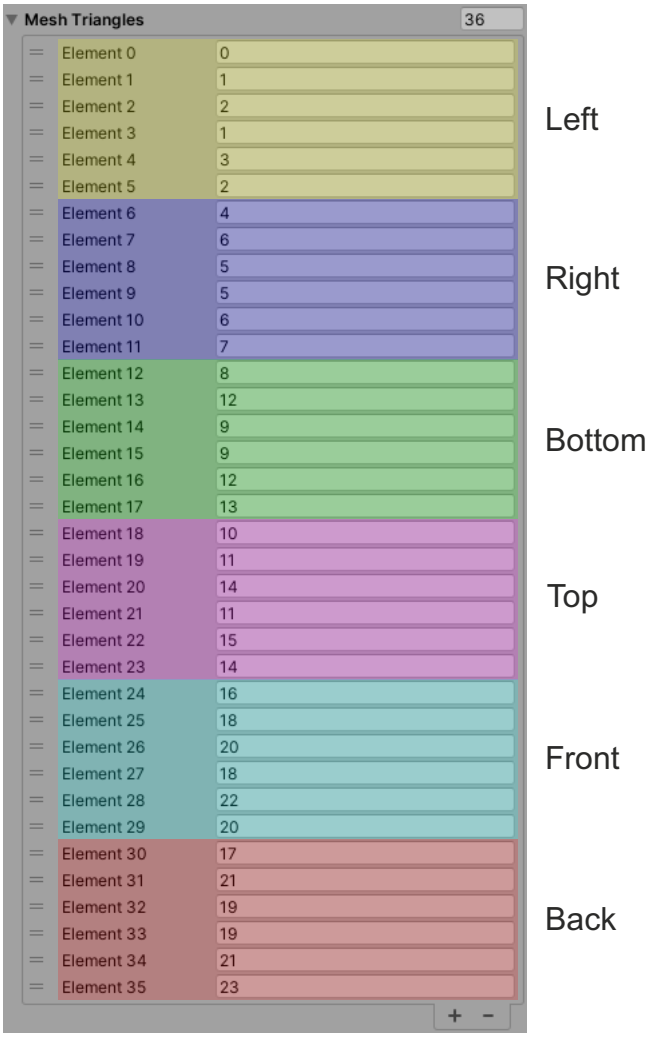


Figure 2.11: Mesh triangle indices of `MyCube` prefab

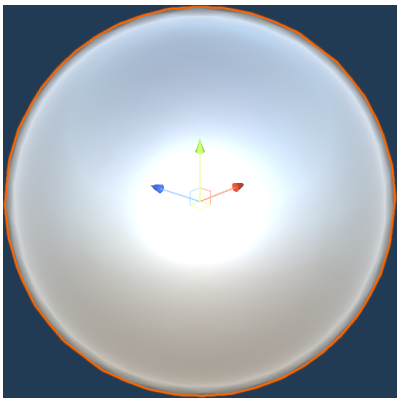
Finally, we attach a `Cube` tag to the `MyCube` prefab (figure 2.12).

Figure 2.12: All cubes have a `Cube` tag.

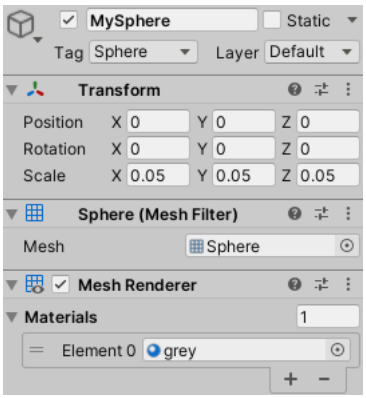
We will use these tags to find all cubes and toggle their visibility in section 2.7.14.

2.5 `MySphere` prefab

We use spheres to indicate the vertices of the cubes. Therefore, the `MySphere` prefab is just a primitive sphere scaled down to 5 % with a grey material.



(a) MySphere object



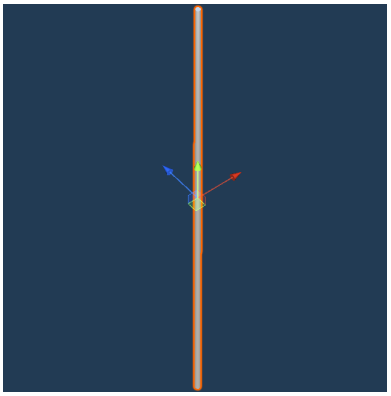
(b) MySphere data

Figure 2.13: MySphere prefab

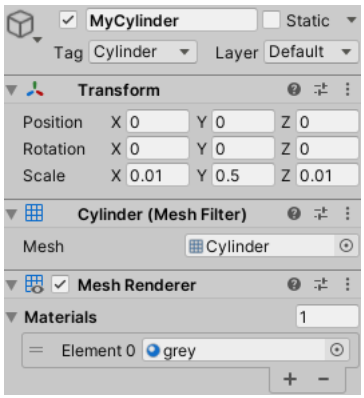
Finally, we attach a Sphere tag to the MySphere prefab (figure 2.13b). We will use these tags to find all spheres, toggle their visibility in section 2.7.14, and hide those with identical positions in section 2.7.15.

2.6 MyCylinder prefab

We use cylinders to display the edges of the cubes. Therefore, the MyCylinder prefab is just a primitive cylinder with a length (height) of 0.5, a diameter of 1 %, and grey material.



(a) MyCylinder object



(b) MyCylinder data

Figure 2.14: MyCylinder prefab

Finally, we attach a Cylinder tag to the MyCylinder prefab (figure 2.14b). We will use these tags to find all cylinders, toggle their visibility in section 2.7.14, and hide those with identical positions in section 2.7.15.

2.7 HypercubeScript

The HypercubeScript script is attached to the MyHypercube GameObject. It creates the hypercube vertices and defines the indices of the cubes and the cylinders. In every simulation step, it manages the user interaction, rotates and unfolds the hypercube and updates the hypercube projection into 3D.

UNITY scripts automatically import some standard types from predefined namespaces of which we only use the UnityEngine:

```
using UnityEngine;
```

Every UNITY script derives from the MonoBehaviour base class:

```
public class Spheres_class : MonoBehaviour
{
```

Before we define the Start function, we declare[1] some (global) objects and variables as properties:

```
private GameObject[] allCubes;
private GameObject[] allSpheres;
private GameObject[] allCylinders;
private int nAllSpheres;
private int nAllCylinders;
private float angleXY = 0f;
private float angleUnfold = 0f;
private float previousAngleUnfold = 0f;
private Vector4[] hypercubeVerticesOriginal;
private Vector4[] hypercubeVertices;
private Vector3[] cubeVertices;
private Vector3[] meshVertices;
private const int nHypercubeVertices = 16;
private const int nCubeVertices = 8;
private const int nMeshVertices = 24;
private const int nCubes = 8;
private const int nCylinders = 12;
private int[,] cylinderIndices;
private int[,] cubeIndices;
private int visibility = 0;
private bool autorotate = true;
```

2.7.1 Start

The start function is called before the first simulation step. It creates the hypercube vertices and defines the indices of the cubes and the cylinders:

[1] We explain variables, the meaning of which cannot directly be derived from their names, when we first use them. nHypercubeVertices obviously is the number of vertices in a hypercube, ...

```
void Start()
{
```

In the function, we define the constant half-length of the hypercube

```
const float sizeHypercube = 0.5f;
```

let UNITY find all cube, sphere, and cylinder `GameObjects`

```
allCubes = GameObject.FindGameObjectsWithTag("Cube");
allSpheres = GameObject.FindGameObjectsWithTag("Sphere");
allCylinders = GameObject.FindGameObjectsWithTag("Cylinder");
```

and determine[2] the number of spheres and cylinders it has found:

```
nAllSpheres = allSpheres.Length;
nAllCylinders = allCylinders.Length;
```

Additionally, we instantiate the containers for the vertices of a cube, a hypercube[3], and a cube mesh:

```
cubeVertices = new Vector3[nCubeVertices];
hypercubeVerticesOriginal = new Vector4[nHypercubeVertices];
hypercubeVertices = new Vector4[nHypercubeVertices];
meshVertices = new Vector3[nMeshVertices];
```

We use four nested for-loops to systematically define the 16 vertices of a 4D hypercube:

[2] Yes, we know the number of cubes, the number of vertices and edges per cube ...

[3] Fortunately, UNITY offers generic four-dimensional vectors (`Vector4`) and even 4×4 matrices (`Matrix4x4`). Usually, these four-dimensional entities are used for homogeneous coordinates [5] but we are very lucky to use them for true 4D vertices and rotations.

```
for (int ix = 0; ix <= 1; ix++)
{
  for (int iy = 0; iy <= 1; iy++)
  {
    for (int iz = 0; iz <= 1; iz++)
    {
      for (int iw = 0; iw <= 1; iw++)
      {
        hypercubeVerticesOriginal
        [8 * ix + 4 * iy + 2 * iz + iw] =
        new Vector4
        (2 * ix - 1, 2 * iy - 1, 2 * iz - 1, 2 * iw - 1)
        * sizeHypercube;
      }
    }
  }
}
```

Figure 2.15: All vertices of the unit hypercube are centered around the origin.

Since the unit hypercube has an edge length of 1.0 and is centered around the origin, all vertex components are either $+0.5$ or -0.5. Therefore, we can use a simple sign table (table 2.2) to display the structure of the original hypercube vertices:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+
y	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	+
z	−	−	+	+	−	−	+	+	−	−	+	+	−	−	+	+
w	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+

Table 2.2: Hypercube vertex signs

As explained in section 1.7, a hypercube consists of eight cubes. We define which hypercube vertex belongs to which cube by searching for eight vertices with the same sign in a row in table 2.2. We e. g. find the first eight vertices to have negative signs in the first row in table 2.2 ($x = -0.5$). This is the "left" cube (first row of `cubeIndices`):

```
cubeIndices = new int[nCubes, 8]
{
  {0, 1, 2, 3, 4, 5, 6, 7},
  {8, 9, 10, 11, 12, 13, 14, 15},
  {0, 1, 2, 3, 8, 9, 10, 11},
  {4, 5, 6, 7, 12, 13, 14, 15},
  {0, 1, 4, 5, 8, 9, 12, 13},
  {2, 3, 6, 7, 10, 11, 14, 15},
  {0, 2, 4, 6, 8, 10, 12, 14},
  {1, 3, 5, 7, 9, 11, 13, 15}
};
```

Figure 2.16: List of cube vertex indices

The "right" cube consists of the second half of vertices in table 2.2 all of which have positive signs in the first row ($x = +0.5$). The "bottom" cube has negative signs in the second row ($y = -0.5$) of table 2.2 resulting in cube indices of $0 - 3$ and $8 - 11$ (third row of `cubeIndices`).

The last row of `cubeIndices` addresses all vertices with odd indices (1, 3, 5, ..., 15). These are the vertices having a positive w-component ($w = +0.5$) in table 2.2. Even though we cannot directly imagine the fourth perpendicular direction (ana $\leftrightarrow$ kata), we could call this cube the "kata" cube or the "outer" cube in figure 1.1.

Finally, we define the edges of the cubes which are displayed by thin cylinders (section 2.6):

```
cylinderIndices = new int[nCylinders, 2]
{
  {0, 1},
  {0, 2},
  {0, 4},
  {1, 3},
  {1, 5},
  {2, 3},
  {2, 6},
  {3, 7},
  {4, 5},
  {4, 6},
  {5, 7},
  {6, 7}
};
}
```

Figure 2.17: List of cylinder end vertices

As depicted in figure 2.18

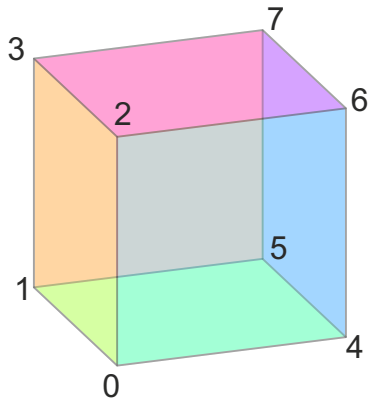


Figure 2.18: Vertices of a cube

there are edges between vertex 0 and vertex 1 (first row of `cylinderIndices`), between vertex 0 and vertex 2 (second row of `cylinderIndices`), ..., and between vertex 6 and vertex 7 (last row of `cylinderIndices`).

2.7.2 `Update`

The `Update` function is called in every simulation step:

```
private void Update()
{
```

We use it to determine if the user has pressed a key.

If they pressed the `V` key

```
if (Input.GetKeyDown(KeyCode.V))
```

we cycle the `visibility` flag through its five states (figure 1.7 and section 2.7.14):

```
{
  visibility = (visibility + 1) % 5;
}
```

If they pressed the `Space` bar

```
if (Input.GetKeyDown(KeyCode.Space))
```

we toggle the `autorotate` flag (section 2.7.3):

```
{
  autorotate = !autorotate;
}
}
```

2.7.3 FixedUpdate

In every fixed[4] rate simulation step

```
private void FixedUpdate()
{
```

we trigger the actual rotation and unfolding of the hypercube.

If the user pressed the A or the D key[5] we de- or increment the rotation angle about the x-y-plane by a small[6] amount:

```
angleXY += Input.GetAxis("Horizontal") * 0.5f * Time.deltaTime;
```

Additionally, we de- or increment the unfold angle with the W/S key pair

```
angleUnfold += Input.GetAxis("Vertical") * 0.5f * Time.deltaTime;
```

and restrict its domain at $0° \ldots 90°$:

```
angleUnfold = Mathf.Clamp(angleUnfold, 0, Mathf.PI / 2);
```

If the user has used the W or the S key, the unfold angle has changed

```
if (angleUnfold != previousAngleUnfold)
```

and we reset the rotation angle

```
{
  angleXY = 0;
```

switch off auto rotation

```
  autorotate = false;
```

and buffer the current unfold angle for comparison in the next time step:

```
  previousAngleUnfold = angleUnfold;
}
```

If the autorotate flag is on

```
if (autorotate)
```

we increment the rotation angle for a very small amount in every time step, resulting in a very slow automatic rotation of the hypercube:

[4] It seems reasonable to perform "asynchronous" actions (like switching properties on and off) in the Update method and use the FixedUpdate method for all continuous visible actions that should take place with a constant "velocity" (like rotating).

[5] The left arrow and right arrow keys (defining the "Horizontal" input axis) are automatically mapped to the A/D key pair that gamers are more familiar with.

[6] The multiplication with Time.deltaTime ensures a constant angular velocity with possibly inconsistent frame rates.

```
{
  angleXY += 0.05f * Time.deltaTime;
}
```

If the rotation angle is not zero any more (because the user has pressed the `A` or the `D` key without pressing the `W` or `S` key)

```
if (angleXY != 0)
```

we reset[7] the unfold angle

```
{
  angleUnfold = 0;
}
```

We now want to do the rotation of the hypercube about the x-y-plane in the 4D space (figure 1.2). Inside a loop over all vertices of the hypercube

```
for (
  int iHypercubeVertex = 0;
  iHypercubeVertex < nHypercubeVertices;
  iHypercubeVertex++)
```

we rotate each hyperspace vertex[8] by multiplying it (from the left) with the 4D rotation matrix calculated from the current rotation angle in section 2.7.4:

```
{
  hypercubeVertices[iHypercubeVertex] =
    RotationXY(angleXY) *
    hypercubeVerticesOriginal[iHypercubeVertex];
}
```

Additionally, we unfold (section 1.5.2), project, and display the hypercube cube by cube utilizing the `UnfoldCube` method (section 2.7.7):

```
UnfoldCube(0, new Vector4(0.5f, 0, 0, 0.5f),
  RotationYZ(-angleUnfold)); // Left cube

UnfoldCube(1, new Vector4(-0.5f, 0, 0, 0.5f),
  RotationYZ(angleUnfold)); // Right cube

UnfoldCube(2, new Vector4(0, 0.5f, 0, 0.5f),
  RotationXZ(angleUnfold)); // Bottom cube
```

[7] This ensures that rotating and unfolding does not happen at the same time; simultaneous rotating and unfolding looks weird and does not provide any further insights.

[8] Please note that we rotate the original vertices by an increasing rotation angle instead of rotating the current vertices incrementally by a small rotation angle increments. Therefore, we do not overwrite the original vertices but use a second variable for the current vertices.

```
UnfoldCube(3, new Vector4(0, -0.5f, 0, 0.5f),
  RotationXZ(-angleUnfold)); // Top cube

UnfoldCube(4, new Vector4(0, 0, 0.5f, 0.5f),
  RotationXY(angleUnfold)); // Back cube

UnfoldCube(5, new Vector4(0, 0, -0.5f, 0.5f),
  RotationXY(-angleUnfold)); // Front cube
```

Unfolding involves a translation of the 4D rotation plane into the origin (second parameter of the `UnfoldCube` method), the rotation (section 2.7.6, section 2.7.5 and section 2.7.4) about the rotation plane of that specific cube (third parameter of the `UnfoldCube` method), and the translation of the rotation plane back to its original position.

The outer cube of figure 1.6a (index 6) does not[9] have to be unfolded at all; it becomes the concealed inner cube after the unfolding (figure 1.6f):

```
UnfoldCube(6, new Vector4(0, 0, 0, 0),
  Matrix4x4.identity); // Outer cube, no unfolding
```

Just like the back plane in figure 1.5, the inner cube (index 7) has to be rotated twice; we use a separate method (section 2.7.8) for that:

```
UnfoldInnerCube(7, new Vector4(0, 0.5f, 0, 0.5f),
  RotationXZ(angleUnfold)); // Inner cube, double unfolding!
```

Finally, the `VisibilityCubesSpheresCylinders` method (section 2.7.14)

```
VisibilityCubesSpheresCylinders();
```

updates the visibility of the cubes, spheres, and cylinders and the `AntiZFighting` method (section 2.7.15) hides duplicate objects to prevent z-fighting [6]:

```
AntiZFighting();
}
```

2.7.4 `RotationXY`

We only use three of the six 4D rotations defined in section A.4: the rotation about the x-y-plane (section A.4.1), the rotation about the x-z-plane (section A.4.5), and the rotation about the y-z-plane (section A.4.2).We define a separate method for each rotation.

The method to rotate about the x-y-plane

[9] No translation (`Vector4(0, 0, 0, 0)`) and no rotation (`Matrix4x4.identity`) is necessary for the outer cube. Nevertheless, we still have to call the `UnfoldCube` method because we perform the projection of the cube and the update of the mesh, spheres, and cylinders inside the `UnfoldCube` method.

```
private Matrix4x4 RotationXY(float angle)
{
```

gets the rotation angle via its parameter list and computes the cosine (`c`), the sine (`s`), and the negative sine (`m`) of the rotation angle:

```
float c = Mathf.Cos(angle);
float s = Mathf.Sin(angle);
float m = -s;
```

After instantiating a new 4D rotation matrix

```
Matrix4x4 rotationMatrix = new Matrix4x4();
```

we define the matrix[10] using `c`, `s`, and `m` according to equation (A.4):

```
rotationMatrix.SetRow(0, new Vector4(1, 0, 0, 0));
rotationMatrix.SetRow(1, new Vector4(0, 1, 0, 0));
rotationMatrix.SetRow(2, new Vector4(0, 0, c, m));
rotationMatrix.SetRow(3, new Vector4(0, 0, s, c));
```

Finally, we return the 4×4 rotation matrix:

```
return rotationMatrix;
}
```

2.7.5 RotationXZ

We compute[11] the the rotation matrix about the x-z-plane in a similar way according to equation (A.6):

```
private Matrix4x4 RotationXZ(float angle)
{
float c = Mathf.Cos(angle);
float s = Mathf.Sin(angle);
float m = -s;
Matrix4x4 rotationMatrix = new Matrix4x4();
rotationMatrix.SetRow(0, new Vector4(1, 0, 0, 0));
rotationMatrix.SetRow(1, new Vector4(0, c, 0, m));
rotationMatrix.SetRow(2, new Vector4(0, 0, 1, 0));
rotationMatrix.SetRow(3, new Vector4(0, s, 0, c));
return rotationMatrix;
}
```

[10] Using one-letter variables, the matrix code structure bears a clear resemblance to the matrix definition in equation (A.4).

[11] Since we compute all rotation matrices for the same (negative) angle, we could save some computing time by calculating the cosine, sine, and negative sine outside these methods and provide the methods with the calculated values via their parameter lists. Nevertheless, for reusability reasons, we do the computation of the trigonometric functions inside the methods.

2.7.6 RotationYZ

We compute the the rotation matrix about the y-z-plane in a similar way according to equation (A.5):

```
private Matrix4x4 RotationYZ(float angle)
{
float c = Mathf.Cos(angle);
float s = Mathf.Sin(angle);
float m = -s;
Matrix4x4 rotationMatrix = new Matrix4x4();
rotationMatrix.SetRow(0, new Vector4(c, 0, 0, s));
rotationMatrix.SetRow(1, new Vector4(0, 1, 0, 0));
rotationMatrix.SetRow(2, new Vector4(0, 0, 1, 0));
rotationMatrix.SetRow(3, new Vector4(m, 0, 0, c));
return rotationMatrix;
}
```

2.7.7 UnfoldCube

The eight calls to the `UnfoldCube` method on page 27

```
private void UnfoldCube(
  int iCube, Vector4 translation, Matrix4x4 rotationMatrix)
{
```

are done in every simulation step in order to unfold the hypercube (if the user has initiated the unfolding). Additionally, `UnfoldCube` calls the perspective projection method and the update method for the selected cube's mesh, its spheres, and its cylinders.

Each call to `UnfoldCube` translates and rotates a specific cube. The translation is necessary because all cubes are initially centered around the origin (22); a pure rotation[12] would therefore rotate the cube about its center in the origin as well. If we want to 4D rotate the cube about one of its faces[13], we have to translate the cube with this face into the origin, do the rotation and translate the fixed face (together with the rotated cube) back to its former position.

After declaring a buffer for a single 4D vertex of the selected cube

```
Vector4 hypercubeVertexBuffer;
```

we start a loop over all eight vertices of a cube:

[12] Rotations (section A.4) are always done around the origin.

[13] The contra-intuitive 4D rotation about a fixed face is explained in appendix A. Even though it is very hard to imagine for us limited 3D beings, we have to understand that the rotation plane of a 4D rotation is indeed fixed. The whole rotation plane itself is not rotated; just like the rotation axis of a 3D rotation!

```
for (
  int iCubeVertex = 0;
  iCubeVertex < nCubeVertices;
  iCubeVertex++)
{
```

Inside the loop, we buffer the current vertex of the selected cube (`iCube`) using the cube vertex index array `cubeIndices` (figure 2.16):

```
  hypercubeVertexBuffer =
    hypercubeVertices[cubeIndices[iCube, iCubeVertex]];
```

Each vertex is then translated to move the fixed rotation face into the origin

```
  hypercubeVertexBuffer += translation;
```

rotated, using the specific rotation matrix from the parameter list

```
  hypercubeVertexBuffer =
    rotationMatrix * hypercubeVertexBuffer;
```

and translated back to move the fixed rotation face back into its former position:

```
  hypercubeVertexBuffer -= translation;
```

Finally, all we have to do is to project the current vertex from 4D to 3D using the perspective projection method described in section 2.7.9

```
  cubeVertices[iCubeVertex] =
    PerspectiveProjectionOfOneVertex(hypercubeVertexBuffer);
}
```

and update the mesh, the vertex spheres, and the edge cylinders of the current cube (section 2.7.10):

```
MeshSpheresAndCylindersUpdate(iCube);
}
```

2.7.8 `UnfoldInnerCube`

The unfolding of the inner cube is a bit more tricky. Just like the red back face in figure 1.5 is not directly connected to the fixed front face but only to the bottom face, the inner cube is not directly connected to the fixed outer cube but only to the bottom cube (figure 1.6). Therefore, the inner cube has to be rotated twice: One rotation together with the bottom cube and another one about the face connecting it to the bottom cube. Since these two rotations depend on each other, we define an separate method for the rotation of the inner cube:

```
private void UnfoldInnerCube(
  int iCube,
  Vector4 translation,
  Matrix4x4 rotationMatrix)
{
```

We use the result of the first rotation in the second rotation. Therefore, we instantiate a buffer for all the vertices of the cube after the first rotation

```
Vector4[] hypercubeVerticesBuffer = new Vector4[nCubeVertices];
```

and a 4D vector for the second translation

```
Vector4 secondTranslation = new Vector4();
```

The first rotation is just a standard translation/rotation/translation about the face of the fixed outer cube. It is similar to the rotation of the bottom cube on page 27. The only difference is the fact that we save all rotated vertices in the buffer:

```
for (
  int iCubeVertex = 0;
  iCubeVertex < nCubeVertices;
  iCubeVertex++)
{
hypercubeVerticesBuffer[iCubeVertex] =
  hypercubeVertices[cubeIndices[iCube, iCubeVertex]];

hypercubeVerticesBuffer[iCubeVertex] += translation;

hypercubeVerticesBuffer[iCubeVertex] =
  rotationMatrix * hypercubeVerticesBuffer[iCubeVertex];

hypercubeVerticesBuffer[iCubeVertex] -= translation;
}
```

The second rotation is done about the "lower" face[14] of the currently rotating inner cube. Inside a second loop over all cube vertices

```
for (
  int iCubeVertex = 0;
  iCubeVertex < nCubeVertices;
  iCubeVertex++)
{
```

we define the second translation that translates the lower face of the currently rotating inner cube to the origin:

[14] At first, the face that connects the inner cube to the bottom cube is the "lower" face of the inner cube. During the unfolding, this connecting face becomes the upper face of the inner cube.

```
  secondTranslation.y = -hypercubeVerticesBuffer[1].y;
  secondTranslation.w = -hypercubeVerticesBuffer[1].w;
```

Now we can do the second standard translation/rotation/translation

```
  hypercubeVerticesBuffer[iCubeVertex] += secondTranslation;

  hypercubeVerticesBuffer[iCubeVertex] =
  rotationMatrix * hypercubeVerticesBuffer[iCubeVertex];

  hypercubeVerticesBuffer[iCubeVertex] -= secondTranslation;
```

and the projection and update of mesh, spheres, and cylinders:

```
  cubeVertices[iCubeVertex] =
    PerspectiveProjectionOfOneVertex
    (hypercubeVerticesBuffer[iCubeVertex]);
}
MeshSpheresAndCylindersUpdate(iCube);
}
```

2.7.9 PerspectiveProjectionOfOneVertex

We want to use perspective projection to project the 4D hypercube into our 3D space, just like the perspective projection of the 3D cube onto the 2D plane in figure 1.3c. To derive the equations of the perspective projection of an n-dimensional object into the $(n-1)$-dimensional subspace, we demonstrate the perspective projection of a 2-dimensional object onto a 1-dimensional "screen" in figure 2.19.

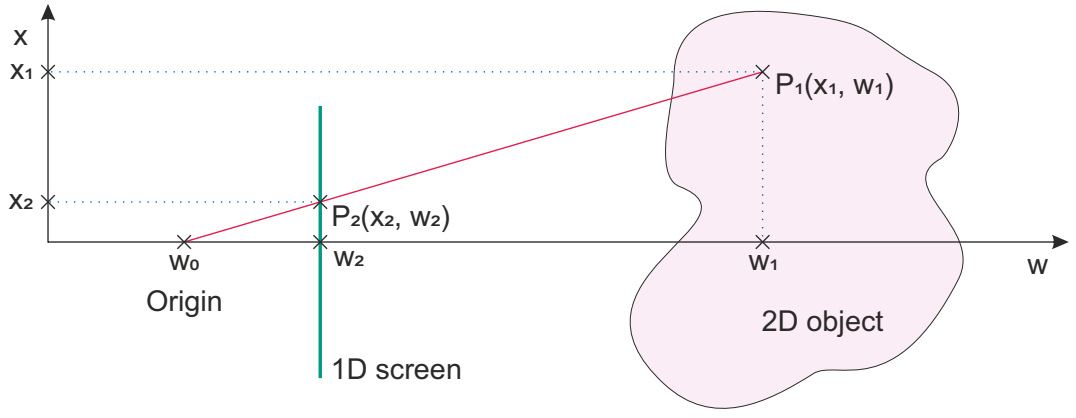


Figure 2.19: Perspective projection of a 2D object onto a 1D screen

We connect an arbitrary point $P_1(x_1, w_1)$ of the 2D object in figure 2.19 with the origin (center of projection, camera, eye, spectator, ...) at w_0 by a red line. This red line

intersects the green 1D screen line (canvas, image, view "plane", projection "plane", ...) at the projection point $P_2(x_2, w_2)$. Since we can arbitrarily choose w_0 and w_2 (as long as they are outside the object), we are looking for the unknown x_2-component of the projection point P_2 on the screen. We can identify two similar triangles $\Delta(w_0, w_2, P_2)$ and $\Delta(w_0, w_1, P_1)$ with equal ratios of corresponding side

$$\frac{x_2}{w_2 - w_0} = \frac{x_1}{w_1 - w_0}$$

resulting in the unknown projection coordinate:

$$x_2 = \frac{x_1(w_2 - w_0)}{w_1 - w_0} \tag{2.1}$$

Since we are already projecting the object along the fourth (ana ↔ kata) w-axis, we can scale up the problem two dimensions and compute the other two coordinates of a 4D object in our 3D projection space in the same way:

$$y_2 = \frac{y_1(w_2 - w_0)}{w_1 - w_0} \tag{2.2}$$

$$z_2 = \frac{z_1(w_2 - w_0)}{w_1 - w_0} \tag{2.3}$$

We are now ready to write the method to project a single 4D vertex of the hypercube into our 3D space:

```
private Vector3 PerspectiveProjectionOfOneVertex(
  Vector4 hypercubeVertex)
{
```

Since the minimum w-coordinate of a hypercube vertex is -0.5, we position the origin (figure 2.19) and the 3D screen "a little bit to the ana" of the hypercube:

```
const float wOrigin = -1.5f;
const float wScreen = -0.5f;
```

We declare a three-dimensional vector for the 3D projection of the hypercube vertex

```
Vector3 cubeVertex;
```

and use equations (2.1), (2.2), and (2.3) to compute and return its components:

```
cubeVertex.x = hypercubeVertex.x * (wScreen - wOrigin) /
  (hypercubeVertex.w - wOrigin);

cubeVertex.y = hypercubeVertex.y * (wScreen - wOrigin) /
  (hypercubeVertex.w - wOrigin);
```

```
cubeVertex.z = hypercubeVertex.z * (wScreen - wOrigin) /
  (hypercubeVertex.w - wOrigin);

return cubeVertex;
}
```

2.7.10 MeshSpheresAndCylindersUpdate

As the name of

```
private void MeshSpheresAndCylindersUpdate(int iCube)
{
```

implies, this method is just a container for the update methods of the mesh (section 2.7.11), the vertex spheres (section 2.7.12), and the edge cylinders (section 2.7.13) of the current cube:

```
MeshUpdate(iCube);
SpheresUpdate(iCube);
CylindersUpdate(iCube);
}
```

2.7.11 MeshUpdate

As indicated in figure 2.20, `MyCube` prefabs (section 2.4) are children of `MyCCS` prefabs (section 2.3) which are children of the `MyHypercube` prefab (section 2.2).

Figure 2.20: The cube is a grandchild of the hypercube.

Therefore, if we want to update the mesh of a single cube in

```
private void MeshUpdate(int iCube)
{
```

we have to address (read from right to left) the `mesh` of the `meshfilter` of the first child (the cube) of a specific child (the CCS) of the `transform` of the hypercube:

```
Mesh mesh = transform.GetChild(iCube).GetChild(0).
  GetComponent<MeshFilter>().mesh;
```

Inside a loop over all vertices of the current cube

```
for (
  int iCubeVertex = 0;
  iCubeVertex < nCubeVertices;
  iCubeVertex++)
{
```

we use another loop to address all three mesh vertices (figure 2.8) of every cube vertex:

```
  for (
    int iMeshVertex = 0;
    iMeshVertex < nMeshVertices;
    iMeshVertex += nCubeVertices)
  {
```

Inside the loop we copy the current vertex of the current cube to its mesh vertex:

```
    meshVertices[iCubeVertex + iMeshVertex] =
      cubeVertices[iCubeVertex];
  }
}
```

Finally, we copy the just computed mesh vertices to the corresponding mesh property

```
mesh.vertices = meshVertices;
```

and recalculate the normals in order to update lights, shadows, and reflections on the cube:

```
mesh.RecalculateNormals();
}
```

2.7.12 SpheresUpdate

In

```
private void SpheresUpdate(int iCube)
{
```

we move all vertex spheres of the current cube to its current vertex. The spheres are great grandchildren of the hypercube (figure 2.21).

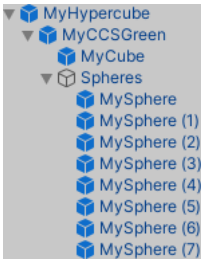


Figure 2.21: The eight vertex spheres of a cube are great grandchildren of the hypercube.

Inside a loop over all vertices of the current cube

```
for (
  int iCubeVertex = 0;
  iCubeVertex < nCubeVertices;
  iCubeVertex++)
{
```

we set the position of the current sphere to the position of the current vertex:

```
  transform.GetChild(iCube).GetChild(1).GetChild(iCubeVertex).
    position = cubeVertices[iCubeVertex];
}
}
```

2.7.13 CylindersUpdate

The update of the edge cylinders is a bit more difficult. In

```
private void CylindersUpdate(int iCube)
{
```

we start a loop over all edge cylinders of the current cube:

```
for (
  int iCylinder = 0;
  iCylinder < nCylinders;
  iCylinder++)
{
```

Inside the loop, we define both end points of the current cylinder using the list in figure 2.17

```
  Vector3 startPoint =
    cubeVertices[cylinderIndices[iCylinder, 0]];

  Vector3 endPoint =
    cubeVertices[cylinderIndices[iCylinder, 1]];
```

and compute a vector to the center of the edge

```
  Vector3 midPoint = (startPoint + endPoint) / 2;
```

and another one from the start point of the edge to its end point:

```
  Vector3 connection = endPoint - startPoint;
```

As indicated in figure 2.22, the 12 edge cylinders of a cube are great grandchildren of the hypercube

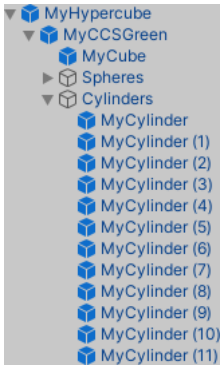


Figure 2.22: The 12 edge cylinders of a cube are great grandchildren of the hypercube.

and can therefore be accessed in the same way as the spheres:

```
  Transform cylinderTransform =
    transform.GetChild(iCube).GetChild(2).GetChild(iCylinder);
```

Since the projected "cubes" are square frustra in 3D, their edges might not have unit length. Therefore, we have to access the scale of the current edge

```
  Vector3 scale = cylinderTransform.localScale;
```

and fit[15] the cylinder to the distance between its start vertex and its end vertex:

```
  scale.y = connection.magnitude / 2;
```

The pivot point of UNITY's prefab cylinder is at its center. Therefore, we position the current cylinder at the already computed mid point between its cube vertices:

```
  cylinderTransform.position = midPoint;
```

Now we have to align the cylinder with the vector between its cube vertices. Fortunately, we can do that by simply aligning the cylinders `up` direction with the already computed vector from the start vertex to the end vertex:

```
  cylinderTransform.up = connection;
```

Finally, we write the updated scale back to the `transform` of the current cylinder:

```
  cylinderTransform.localScale = scale;
}
}
```

[15] UNITY's prefab cylinder has a height of 2 (from $y = -1$ to $y = +1$); hence we have to divide its y-component by 2.

2.7.14 VisibilityCubesSpheresCylinders

In section 2.7.2 we cycle the visibility flag (`visibility`) through its five states if the user presses the `V` key. The VisibilityCubesSpheresCylinders method

```
private void VisibilityCubesSpheresCylinders()
{
```

uses the visibility flag to actually enable or disable the visibility of transparent cubes, vertex spheres, and edge cylinders (figure 1.7).

After defining the default visibility values of the objects

```
bool visibilityCubes = false;
bool visibilitySpheres = false;
bool visibilityCylinders = false;
```

we define the visibilities of the cubes, spheres, and cylinders separately depending on the visibility flag:

```
switch (visibility)
{
```

The five different cases resemble figure 1.7:

```
    case 0:
        visibilityCubes = true;
        visibilitySpheres = true;
        visibilityCylinders = true;
        break;

    case 1:
        visibilityCubes = true;
        visibilitySpheres = false;
        visibilityCylinders = true;
        break;

    case 2:
        visibilityCubes = true;
        visibilitySpheres = false;
        visibilityCylinders = false;
        break;

    case 3:
        visibilityCubes = false;
        visibilitySpheres = true;
        visibilityCylinders = true;
        break;
```

```
    case 4:
        visibilityCubes = false;
        visibilitySpheres = false;
        visibilityCylinders = true;
        break;
}
```

We found all cube, sphere, and cylinder objects in section 2.7.1. Therefore, we can now easily set the visibilities of these object groups separately:

```
foreach (GameObject cube in allCubes)
{
    cube.GetComponent<Renderer>().enabled =
    visibilityCubes;
}

foreach (GameObject sphere in allSpheres)
{
    sphere.GetComponent<Renderer>().enabled =
    visibilitySpheres;
}

foreach (GameObject cylinder in allCylinders)
{
    cylinder.GetComponent<Renderer>().enabled =
    visibilityCylinders;
}
}
```

2.7.15 AntiZFighting

If two triangles share the same vertices, the renderer has to decide for every single pixel which triangle to display. Sometimes, the renderer randomly switches between the triangles, leading to the spotted appearance displayed in figure 2.23. Since even in figure 1.6f some of the cubes are still joined face to face, they share some vertex spheres and edge cylinders.

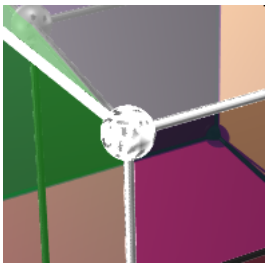


Figure 2.23: Z-fighting makes multiple spheres at the same position appear spotted.

The easiest[16] way to prevent this z-fighting is to check for every sphere and cylinder if another one is already present at the same position and then disable the visibility of the doublet. Therefore, in

```
private void AntiZFighting()
{
```

we start a loop over every sphere:

```
for (
  int isphere = 0;
  isphere < nAllSpheres;
  isphere++)
{
```

Inside the loop, we start another loop over every remaining[17] sphere:

```
  for (
    int iisphere = isphere + 1;
    iisphere < nAllSpheres;
    iisphere++)
  {
```

If both spheres share the same position

```
    if (
      allSpheres[isphere].transform.position ==
      allSpheres[iisphere].transform.position)
    {
```

we simply make the second sphere invisible:

```
      allSpheres[iisphere].GetComponent<Renderer>().enabled =
        false;
    }
  }
}
```

The same is done for every cylinder:

```
for (
  int icylinder = 0;
  icylinder < nAllCylinders;
  icylinder++)
```

[16] Yes, we could a priori analyze the positions of all spheres and cylinders and then only display (or even create) the necessary ones. And yes, we would have done that if we were only talking about the 3D projection of the hypercube. But since during the unfolding, some of the cube face joints are cut and some spheres and cylinders become separated, we would have to keep track for different cases. Therefore, it seems more appropriate to do the analysis a posteriori.

[17] By that, we compare every sphere with every other sphere.

```
{
  for (
    int iicylinder = icylinder + 1;
    iicylinder < nAllCylinders;
    iicylinder++)
  {
    if (
      allCylinders[icylinder].transform.position ==
      allCylinders[iicylinder].transform.position)
    {
      allCylinders[iicylinder].GetComponent<Renderer>().enabled =
        false;
    }
  }
}
}
}
```

2.8 Camera

The user can use their mouse (with the right button pressed) to orbit the camera around the scenery.

2.8.1 `OrbitCamera`

For this purpose, we attach the `OrbitCamera` script

```
using UnityEngine;
public class OrbitCamera : MonoBehaviour
{
```

to the default camera object and declare and define the initial distance of the camera from the origin

```
float distance = 2f;
```

the constant factors translating the mouse (scroll) speed to the orbit angles and the zoom rate

```
readonly float speed_x = 1f;
readonly float speed_y = 1f;
readonly float speed_zoom = 2f;
```

and the constant distance limits:

```
readonly private float distance_min = 2f;
readonly private float distance_max = 5f;
```

Finally, we initialize the orbit angles:

`azimuth:` left and right in the x-z-plane

`elevation:` up and down with respect to the x-z-plane

```
private float azimuth = 30f;
private float elevation = 30f;
```

In every simulation step

```
void LateUpdate()
{
```

we check if the user pressed the right mouse button

```
if (Input.GetMouseButton(1))
{
```

in which case we in- or decrease the orbit angles according to the mouse position change:

```
    azimuth += Input.GetAxis("Mouse X") * speed_x;
    elevation -= Input.GetAxis("Mouse Y") * speed_y;
}
```

With the help of UNITY we compute the quaternion representation of the current attitude from the Euler angles:

```
Quaternion rotation = Quaternion.Euler(elevation, azimuth, 0);
```

We use a scroll wheel input to alter the distance of the camera from the origin and limit it to the already declared limits:

```
distance =
Mathf.Clamp(distance - Input.GetAxis("Mouse ScrollWheel")
* speed_zoom, distance_min, distance_max);
```

Using the negative distance as its z-component, we define the position vector in the local camera coordinate system:

```
Vector3 neg_distance = new Vector3(0.0f, 0.0f, -distance);
```

Since we need the position of the camera in the global coordinate system, we have to transform the position vector from the local camera system to the global world system. UNITY makes this very easy: The multiplication operator of a quaternion object is overloaded, enabling us to multiply a quaternion and a 3D vector, just like we would multiply a transformation matrix with the vector, while at the same time avoiding the gimbal lock problem [7] with the Euler angles in the transformation matrix:

```
Vector3 position = rotation * neg_distance;
```

Finally, we transfer the new attitude and position to the actual camera:

```
transform.rotation = rotation;
transform.position = position;
}
}
```

2.9 Lights

For the illumination of the scene, we introduce six `Directional Light GameObjects` (figure 2.24)

Figure 2.24: Six directional lights

pointing in all positive and negative axis directions (figure 2.25) leading to quite realistic shadows and reflections (figure 1.1).

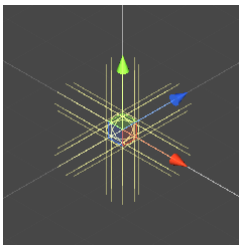
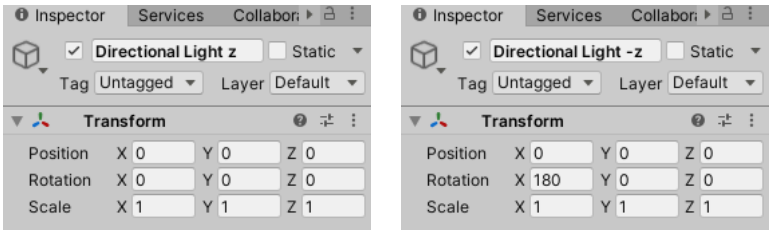


Figure 2.25: Directions of the lights

We achieve the different light directions by rotating the default light - pointing in the positive z-direction (figure 2.26a) - about appropriate axes (e. g. figure 2.26b).



(a) Light in positive z-direction (b) Light in negative z-direction

Figure 2.26: Lights in positive and negative z-directions

Interestingly, the positions of the lights do not matter at all. Even if we leave them all at their default positions at the origin (figure 2.26), the faces of the hypercube are also illuminated "on the outside".

It is very important that the *Pixel Light Count* parameter in UNITY's *Edit* | *Project Settings* | *Quality* | *Rendering* menu must be greater or equal the number of lights that illuminate the scene; otherwise some cubes might look darker than others from certain angles. Since we use six directional lights, we have to set *Pixel Light Count* to a minimum of 6.

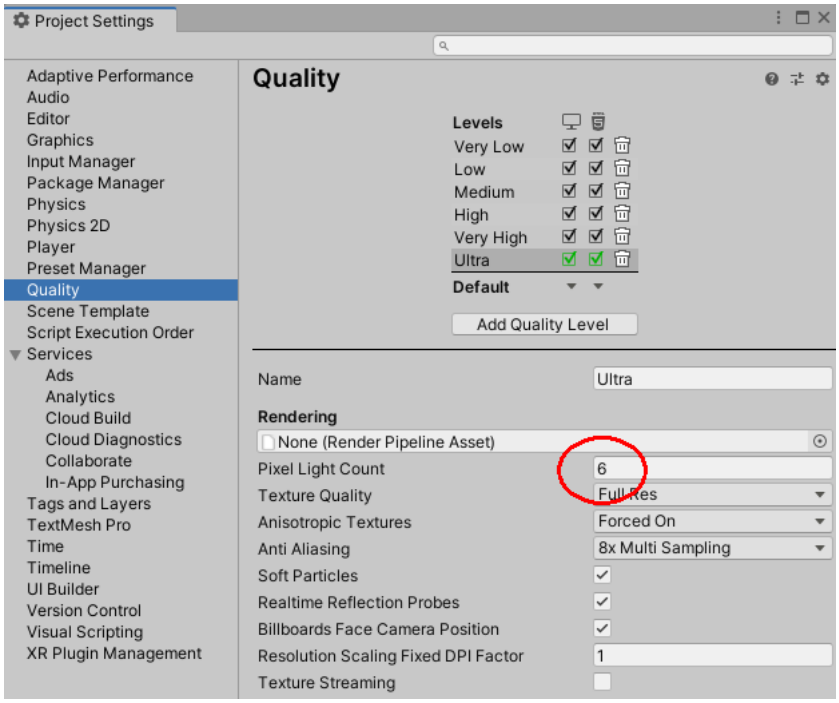


Figure 2.27: *Pixel Light Count* must correspond to the number of lights.

Bibliography

[1] J. J. Buchholz. (2021) Hypercube. [Online]. Available: https://m-server.fk5.hs-bremen.de/hypercube/index.html

[2] Unity. (2021) Unity. [Online]. Available: https://unity.com

[3] Wikipedia. (2021) Cube. [Online]. Available: https://en.wikipedia.org/wiki/Cube

[4] ——. (2021) Crucifixion (corpus hypercubus). [Online]. Available: https://en.wikipedia.org/wiki/Crucifixion_(Corpus_Hypercubus)

[5] ——. (2021) Homogeneous coordinates. [Online]. Available: https://en.wikipedia.org/wiki/Homogeneous_coordinates

[6] ——. (2021) Z-fighting. [Online]. Available: https://en.wikipedia.org/wiki/Z-fighting

[7] ——. (2021) Gimbal lock. [Online]. Available: https://en.wikipedia.org/wiki/Gimbal_lock

[8] ——. (2021) Outer product. [Online]. Available: https://en.wikipedia.org/wiki/Outer_product

[9] ——. (2021) Drehmatrix. [Online]. Available: https://de.wikipedia.org/wiki/Drehmatrix

A Rotation matrices

The German Wikipedia [9] demonstrates the computation of the n-dimensional rotation matrix $\boldsymbol{R}$ that rotates an n-dimensional vector about an $(n-2)$-dimensional fixed hyperspace[1]. The invariant[2] plane in which the rotation takes place is defined by two orthogonal unit vectors $\boldsymbol{g_1}$ and $\boldsymbol{g_2}$.

We define two matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ using outer products[3] of the span vectors $\boldsymbol{g_1}$ and $\boldsymbol{g_2}$

$$\boldsymbol{V} = \boldsymbol{g_1} \otimes \boldsymbol{g_1} + \boldsymbol{g_2} \otimes \boldsymbol{g_2} \tag{A.1}$$

$$\boldsymbol{W} = \boldsymbol{g_1} \otimes \boldsymbol{g_1} - \boldsymbol{g_2} \otimes \boldsymbol{g_1} \tag{A.2}$$

A rotation matrix $\boldsymbol{R}$ that rotates by the rotation angle α in the $\boldsymbol{g_1}$-$\boldsymbol{g_2}$-plane in $\mathbb{R}^n$ is then defined as

$$\boldsymbol{R} = \boldsymbol{I_n} + (\cos(\alpha) - 1) \cdot \boldsymbol{V} + \sin(\alpha) \cdot \boldsymbol{W} \tag{A.3}$$

where $\boldsymbol{I_n}$ is the n-dimensional identity matrix

$$\boldsymbol{I_n} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

A.1 Matlab program `rotation_matrix`

We utilize equation (A.3) in the Matlab function

```
function R = rotation_matrix (dimension, index_1, index_2)
```

[1] Points in the fixed hyperspace are not affected by the rotation.

[2] Points in the invariant plane are rotated into other points in the same plane.

[3] The outer product of two vectors $\boldsymbol{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_m \end{bmatrix}^T$ and $\boldsymbol{v} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}^T$ is defined [8] as the matrix

$$\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} \cdot \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{bmatrix}$$

to compute the rotation matrix `R` from the `dimension` of the requested hyperspace and the indices (`index_1` and `index_2`) of the invariant plane in which the rotation should take place.

Inside the function we declare the rotation angle as a symbolic variable

```
alpha = sym ('alpha');
```

and define the n-dimensional identity matrix:

```
I = eye (dimension);
```

The indices `index_1` and `index_2` pick corresponding column vectors of the identity matrix as the orthogonal unit vectors $\boldsymbol{g_1}$ and $\boldsymbol{g_1}$ defining the invariant plane:

```
g_1 = I(:, index_1);
g_2 = I(:, index_2);
```

Using the unit vectors, we compute the matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ according to equations (A.1) and (A.2)

```
V = g_1 * g_1' + g_2 * g_2';
W = g_1 * g_2' - g_2 * g_1';
```

and finally use equation (A.3) to return the rotation matrix:

```
R = I + (cos (alpha) - 1) * V + sin (alpha) * W;
end
```

A.2 Rotations in $\mathbb{R}^2$

In the 2D-plane a vector has only two coordinates

$$\boldsymbol{a_2} = \begin{bmatrix} x \\ y \end{bmatrix}$$

The Matlab function call

```
rotation_matrix (2, 2, 1)
```

returns the well-known 2-dimensional rotation matrix

$$\boldsymbol{R}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

that rotates the x-axis (first column of the 2-dimensional identity matrix)

$$\boldsymbol{g_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

counterclockwise (mathematically positive) into the y-axis (second column of the 2-dimensional identity matrix

$$\boldsymbol{g_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

if $\alpha = \frac{\pi}{2}$

$$\begin{aligned} \boldsymbol{g_2} &= \boldsymbol{R}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_1} \\ &= \begin{bmatrix} \cos\left(\frac{\pi}{2}\right) & -\sin\left(\frac{\pi}{2}\right) \\ \sin\left(\frac{\pi}{2}\right) & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{aligned}$$

In $\mathbb{R}^2$, the $(n-2)$-dimensional fixed hyperspace is just the 0-dimensional origin.

If we exchanged the order of the span vectors $\boldsymbol{g_1}$ and $\boldsymbol{g_1}$ by calling

```
rotation_matrix (2, 1, 2)
```

the function would return the inverse (transposed) matrix

$$\boldsymbol{R^{-1}}(\alpha) = \boldsymbol{R^T}(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$$

that would perform an undesirable clockwise (mathematically negative) rotation, e. g. rotating the y-axis into the x-axis:

$$\begin{aligned} \boldsymbol{g_1} &= \boldsymbol{R^T}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_2} \\ &= \begin{bmatrix} \cos\left(\frac{\pi}{2}\right) & \sin\left(\frac{\pi}{2}\right) \\ -\sin\left(\frac{\pi}{2}\right) & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{aligned}$$

A.3 Rotations in $\mathbb{R}^3$

In 3D space, a vector has three coordinates:

$$\boldsymbol{a_3} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

A.3.1 Rotation matrix $\boldsymbol{R_x}$

The Matlab function call

```
rotation_matrix (3, 3, 2)
```

returns a 3-dimensional rotation matrix about the x-axis

$$\boldsymbol{R_x}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

that rotates the y-axis (second column of the 3-dimensional identity matrix)

$$\boldsymbol{g_1} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

counterclockwise (mathematically positive) into the z-axis (third column of the 3-dimensional identity matrix

$$\boldsymbol{g_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

if $\alpha = \frac{\pi}{2}$:

$$\begin{aligned} \boldsymbol{g_2} &= \boldsymbol{R_x}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_1} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\left(\frac{\pi}{2}\right) & -\sin\left(\frac{\pi}{2}\right) \\ 0 & \sin\left(\frac{\pi}{2}\right) & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

In $\mathbb{R}^3$, the $(n-2)$-dimensional fixed hyperspace is a 1-dimensional straight line. In the case of $\boldsymbol{R_x}$, the x-axis is the fixed rotation axis; all points on the x-axis stay in place:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ 0 \\ 0 \end{bmatrix}$$

The y-z-plane (together with all its parallel planes) is orthogonal to the fixed x-axis and is called the invariant plane because every point in the y-z-plane is rotated into another point in the y-z-plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} 0 \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ y\cos\alpha - z\sin\alpha \\ y\sin\alpha + z\cos\alpha \end{bmatrix}$$

A.3.2 Rotation matrix $\boldsymbol{R_y}$

The rotation about the fixed y-axis (rotating the z-axis into the x-axis by $\alpha = \frac{\pi}{2}$) is computed by

```
rotation_matrix (3, 1, 3)
```

resulting in

$$\boldsymbol{R_y}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

A.3.3 Rotation matrix $\boldsymbol{R_z}$

The rotation about the fixed z-axis (rotating the x-axis into the y-axis by $\alpha = \frac{\pi}{2}$) is computed by

```
rotation_matrix (3, 2, 1)
```

resulting in

$$\boldsymbol{R_z}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A.4 Rotations in $\mathbb{R}^4$

In 4D hyperspace, a vector has four[4] coordinates

$$\boldsymbol{a_4} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

[4]We introduce w as the fourth coordinate.

A.4.1 Rotation matrix $\boldsymbol{R_{xy}}$

The Matlab function call

```
rotation_matrix (4, 4, 3)
```

returns a 4-dimensional rotation matrix about the x-y-plane[5]

$$\boldsymbol{R_{xy}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{A.4}$$

that rotates the z-axis (third column of the 4-dimensional identity matrix)

$$\boldsymbol{g_1} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

counterclockwise[6] into the w-axis (fourth column of the 4-dimensional identity matrix

$$\boldsymbol{g_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

[5]Most/all(?) human beings cannot imagine a fourth direction (ana/kata) orthogonal on all of our three standard directions (left/right, up/down, back/forth). As a consequence, we cannot imagine a rotation about a plane.

[6]The expression "counterclockwise" presumes that we look at the rotation plane from a certain point of view into a certain direction. In 2D, this is not a problem; there is just the one x-y-plane we look at. In 3D, we look into the negative direction of the third axis which is the normal vector of the plane: If the rotation takes place in the x-y-plane we look into the negative z-direction. And in 4D? There is a third and a fourth axis, both of them orthogonal to the rotation plane. Therefore, the rotation plane does not have a single normal vector but an orthogonal plane spanned by the third and fourth axis. There is no cross product of two 4D vectors. How do we define "counterclockwise" now? From where and into which direction do we look at the rotation plane? It seems useful to call a 4D rotation matrix "counterclockwise" or "positive" if it rotates an axis into "the next" axis for a rotation angle of $\frac{\pi}{2}$: $x \to y, \quad y \to z, \quad z \to w, \quad w \to x$.

if $\alpha = \frac{\pi}{2}$

$$\begin{aligned}
\boldsymbol{g_2} &= \boldsymbol{R_{xy}}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_1} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\left(\frac{\pi}{2}\right) & -\sin\left(\frac{\pi}{2}\right) \\ 0 & 0 & \sin\left(\frac{\pi}{2}\right) & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\end{aligned}$$

In $\mathbb{R}^4$, the $(n-2)$-dimensional fixed hyperspace is a 2-dimensional plane. In the case of $\boldsymbol{R_{xy}}$, the x-y-plane is the fixed rotation plane; all points in the x-y-plane stay in place:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 0 \end{bmatrix}$$

The z-w-plane (together with all its parallel planes) is orthogonal to the fixed x-y-plane and is called the invariant plane because every point in the z-w-plane is rotated into another point in the z-w-plane

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ z \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ z\cos\alpha - w\sin\alpha \\ z\sin\alpha + w\cos\alpha \end{bmatrix}$$

There are three more 4D rotations that rotate an axis into "the next one":

A.4.2 Rotation matrix $\boldsymbol{R_{yz}}$

The rotation about the fixed y-z-plane (rotating the w-axis into the x-axis by $\alpha = \frac{\pi}{2}$) is computed by

```
rotation_matrix (4, 1, 4)
```

resulting in

$$\boldsymbol{R_{yz}}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & 0 & \sin\alpha \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin\alpha & 0 & 0 & \cos\alpha \end{bmatrix} \tag{A.5}$$

A.4.3 Rotation matrix $\boldsymbol{R_{zw}}$

The rotation about the fixed z-w-plane (rotating the x-axis into the y-axis by $\alpha = \frac{\pi}{2}$) is computed by

```
rotation_matrix (4, 2, 1)
```

resulting in

$$\boldsymbol{R_{zw}}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.4.4 Rotation matrix $\boldsymbol{R_{xw}}$

The rotation about the fixed x-w-plane (rotating the y-axis into the z-axis by $\alpha = \frac{\pi}{2}$) is computed by

```
rotation_matrix (4, 3, 2)
```

resulting in

$$\boldsymbol{R_{xw}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.4.5 Rotation matrix $\boldsymbol{R_{xz}}$

The remaining two 4D rotation matrices are a bit tricky:

The rotation about the fixed x-z-plane is computed by

```
rotation_matrix (4, 4, 2)
```

resulting in

$$\boldsymbol{R_{xz}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & 0 & -\sin\alpha \\ 0 & 0 & 1 & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{A.6}$$

This matrix rotates the y-axis (second column of the 4-dimensional identity matrix)

$$\boldsymbol{g_1} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

into the w-axis (fourth column of the 4-dimensional identity matrix)

$$\boldsymbol{g_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

if $\alpha = \frac{\pi}{2}$:

$$\begin{aligned} \boldsymbol{g_2} &= \boldsymbol{R_{xz}}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_1} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\left(\frac{\pi}{2}\right) & 0 & -\sin\left(\frac{\pi}{2}\right) \\ 0 & 0 & 1 & 0 \\ 0 & \sin\left(\frac{\pi}{2}\right) & 0 & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

On the other hand, the inverse matrix, computed by

```
rotation_matrix (4, 2, 4)
```

resulting in

$$\boldsymbol{R_{xz}^{-1}}(\alpha) = \boldsymbol{R_{xz}^{T}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & 0 & \sin\alpha \\ 0 & 0 & 1 & 0 \\ 0 & -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

rotates the w-axis (fourth column of the 4-dimensional identity matrix)

$$\boldsymbol{g_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

into the y-axis (second column of the 4-dimensional identity matrix)

$$\boldsymbol{g_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

if $\alpha = \frac{\pi}{2}$:

$$\begin{aligned}
\boldsymbol{g_2} &= \boldsymbol{R_{xz}^T}\left(\frac{\pi}{2}\right) \cdot \boldsymbol{g_1} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\left(\frac{\pi}{2}\right) & 0 & \sin\left(\frac{\pi}{2}\right) \\ 0 & 0 & 1 & 0 \\ 0 & -\sin\left(\frac{\pi}{2}\right) & 0 & \cos\left(\frac{\pi}{2}\right) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}
\end{aligned}$$

Since the y-axis and the w-axis are not adjacent to each other (the "next" axis of the y-axis is the z-axis and the "next" axis of the w-axis is the x-axis) there is no reason to prefer $\boldsymbol{R_{xz}}$ to $\boldsymbol{R_{xz}^{-1}}$. In this case, our definition of "anticlockwise" is not applicable.

A.4.6 Rotation matrix $\boldsymbol{R_{yw}}$

The same dilemma exist for the remaining rotation matrix $\boldsymbol{R_{yw}}$. We arbitrarily choose

```
rotation_matrix (4, 3, 1)
```

resulting in

$$\boldsymbol{R_{yw}}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

for the rotation about the fixed y-w-plane (rotating the x-axis into the z-axis by $\alpha = \frac{\pi}{2}$).

A.4.7 Rotation matrix $\boldsymbol{R_{xyzw}}$

In 4D, we can concurrently rotate about two independent planes by multiplying the corresponding rotation matrices. Therefore, the matrix that concurrently rotates about the x-y-plane and the z-w-plane is computed as the product of $\boldsymbol{R_{xy}}$ and $\boldsymbol{R_{zw}}$:

$$
\begin{aligned}
\boldsymbol{R_{xyzw}}(\alpha,\beta) &= \boldsymbol{R_{xy}}(\alpha)\cdot\boldsymbol{R_{zw}}(\beta) \\
&= \boldsymbol{R_{zw}}(\beta)\cdot\boldsymbol{R_{xy}}(\alpha) \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix}
\end{aligned}
\tag{A.7}
$$

In contrast to the 3D case, where the sequence of the rotations about two axes is important ($\boldsymbol{R_x}\cdot\boldsymbol{R_y} \neq \boldsymbol{R_y}\cdot\boldsymbol{R_x}$), the product of two independent 4D rotation matrices is commutative[7] (equation (A.7)).

A.4.8 Rotation matrix $\boldsymbol{R_{xzyw}}$

In a similar way, the matrix that concurrently rotates about the x-z-plane and the y-w-planes is computed as the product of $\boldsymbol{R_{xz}}$ and $\boldsymbol{R_{yw}}$:

$$
\begin{aligned}
\boldsymbol{R_{xzyw}}(\alpha,\beta) &= \boldsymbol{R_{xz}}(\alpha)\cdot\boldsymbol{R_{yw}}(\beta) \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & 0 & -\sin\alpha \\ 0 & 0 & 1 & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & \cos\alpha & 0 & -\sin\alpha \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha \end{bmatrix}
\end{aligned}
$$

[7] In the 3D case, we only have a single 1 in each rotation matrix. In the 4D case, every single rotation matrix contains a two-dimensional identity matrix block that is "substituted" by the trigonometric functions of the "other" rotation. From a more abstract point of view: In 3D, there cannot be two independent invariant planes in which the rotation takes place, according to the first paragraph of appendix A. In 4D, we can define two sets of unit span vectors g_1, g_2, and g_3, g_4, all orthogonal to each other, defining two independent invariant planes.

B SteamVR

Unity makes it very easy to develop for different platforms. To interact with the hypercube using a Virtual Reality headset (Valve Index, Oculus Rift, ...), all we have to do is to import the SteamVR library into the `HypercubeScript`

```
using Valve.VR;
```

and set the platform to `Windows` (figure B.1)

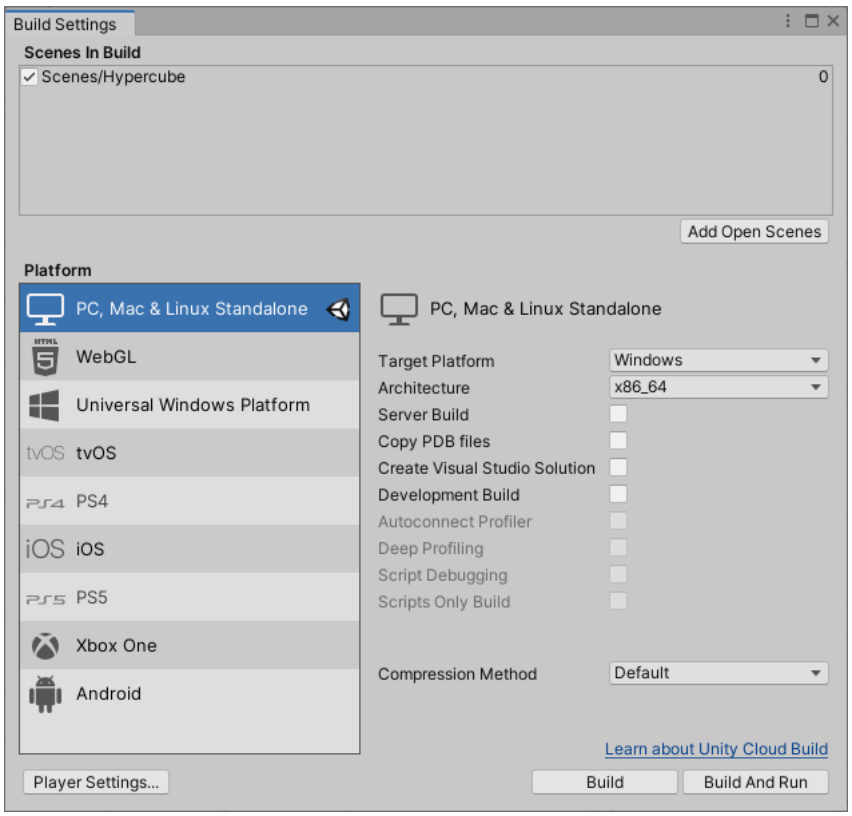


Figure B.1: SteamVR build settings

Additionally, we define a few SteamVR input actions like `RotateHypercube4D`, `UnfoldHypercube`, and `Autorotate` (figure B.2)

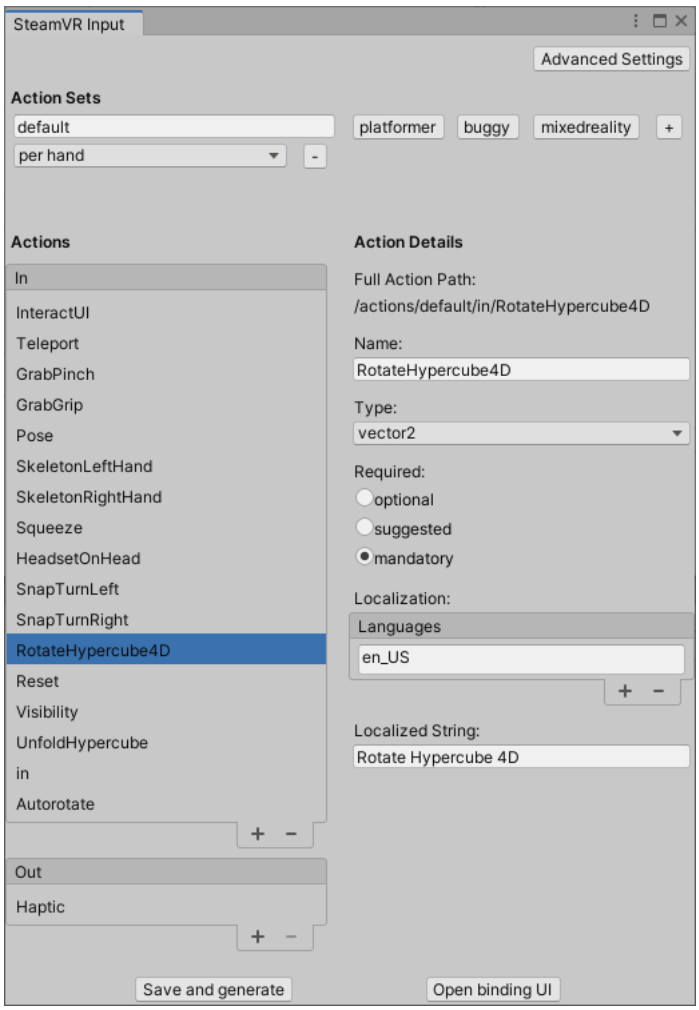


Figure B.2: SteamVR input actions

and replace the keyboard and mouse interactions with the corresponding controller inputs. We define controller button push inputs in the `Update` method

```
private void Update()
{
if (SteamVR_Input.GetStateDown(
  "Reset", SteamVR_Input_Sources.RightHand))
{
  ResetHypercubeRotation();
}

if (SteamVR_Input.GetStateDown(
  "Visibility", SteamVR_Input_Sources.RightHand))
{
  visibility = (visibility + 1) % 5;
}
```

```
if (SteamVR_Input.GetStateDown(
  "Autorotate", SteamVR_Input_Sources.LeftHand))
{
  autorotate = !autorotate;
}
}
```

and controller thumbstick inputs in the `FixedUpdate` method:

```
private void FixedUpdate()
{
angleYW += SteamVR_Input.GetVector2(
  "RotateHypercube4D", SteamVR_Input_Sources.RightHand).x / 200f;

angleXY += SteamVR_Input.GetVector2(
  "RotateHypercube4D", SteamVR_Input_Sources.RightHand).y / 200f;

angleUnfold += SteamVR_Input.GetVector2(
  "UnfoldHypercube", SteamVR_Input_Sources.LeftHand).y / 200f;
```

Instead of the mouse controlled camera orbiting (section 2.8), we use a 4D rotation about the y-w-plane in VR, which has a "similar" effect:

```
for (
  int iHypercubeVertex = 0;
  iHypercubeVertex < nHypercubeVertices;
  iHypercubeVertex++)
{
  hypercubeVertices[iHypercubeVertex] =
    RotationYW(angleYW) *
    hypercubeVerticesOriginal[iHypercubeVertex];

  hypercubeVertices[iHypercubeVertex] =
    RotationXY(angleXY) *
    hypercubeVertices[iHypercubeVertex];
}
```

That's all, Folks!