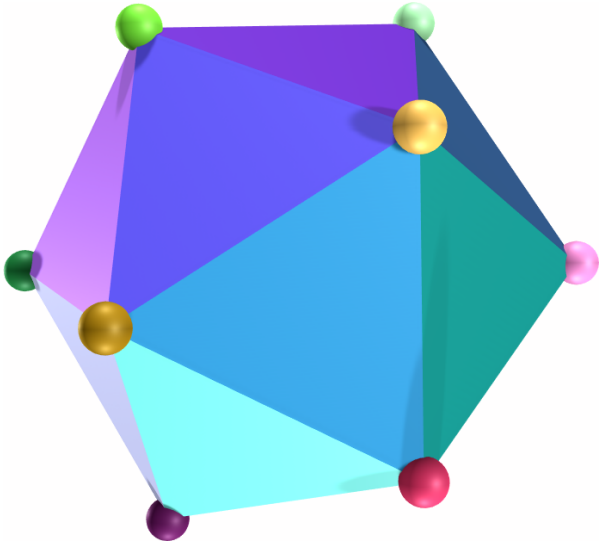


Platonics

Interactive Website

Jörg J. Buchholz

November 18, 2020

1 Manual

1.1 Introduction

In this paper, we describe the utilization and genesis of an interactive website [1] you can use to create, display and manipulate Platonic solids and other polyhedra.

"In geometry, a polyhedron [...] is a three-dimensional shape with flat polygonal faces, straight edges and sharp corners or vertices"[2].

"In three-dimensional space, a Platonic solid is a regular, convex polyhedron"[3].

The polyhedron is dynamically created by simulating physical masses (vertices) connected by springs and dampers (edges) covered by a convex hull (faces).[1] You can use the left mouse button to drag single vertices around and watch the "rubber polyhedra" dilate, translate, and rotate, in order to pull the vertex back into its hull. Pressing the right mouse button, you can orbit the camera around the scenery (section 2.4). With the mouse wheel you can zoom in and out.

You can choose different polyhedra via a button list (figure 1.1).

Figure 1.1: Choose a polyhedron.

The website has been programmed in Unity [4] in C#, compiled for WebGL, and should run in every[2] modern browser.

1.2 Polyhedra

In this chapter we describe the different polyhedra you can choose on the website.

[1] Additionally, every vertex is connected to the origin via a soft spring (section 2.2.1.1), ensuring that the polyhedron will always finally return to the center of the screen.

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

1.2.1 Tetrahedron

The Platonic solid with four vertices is called a regular tetrahedron or triangular pyramid. Obviously, the vertices are forced into a configuration of the four (Greek: tetra-) equivalent equilateral triangular faces depicted in figure 1.2.

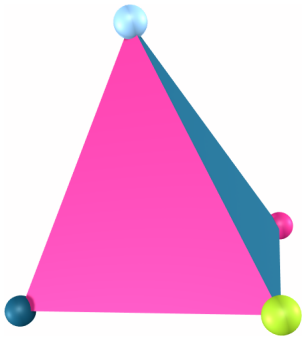


Figure 1.2: Tetrahedron ($n_{vertices} = 4$)

The colors of the vertex spheres are dynamically chosen depending on the vertex positions in the corresponding RGB color cube (figure 2.6). The colors of the faces are defined by the face normal vectors. Since the center perpendicular of every triangular face pierces the opposite vertex (sphere), both have the same color.

1.2.2 Triangular dipyramid

There is no Platonic solid with five vertices. Nevertheless, the corresponding simulation creates the highly symmetrical polyhedron in figure 1.3. It is called a triangular dipyramid, because it can be "constructed" by "gluing" two (Greek: di-) triangular pyramids (tetrahedrons) base-to-base.

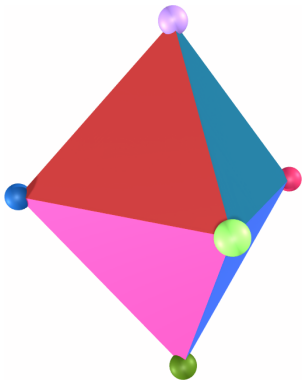


Figure 1.3: Triangular dipyramid ($n_{vertices} = 5$)

1.2.3 Octahedron

The six vertices of this simulation inflate into a regular octahedron (figure 1.4), which is one of the Platonic solids. The octahedron (Greek for "eight faces") is also called a square dipyramid because it can be dissected into two pyramids with square bases. Seen from another angle of view, the octahedron also is a triangular antiprism, because there are (four) pairs of opposing parallel triangles, of which one has its vertices where the other one has its edges.

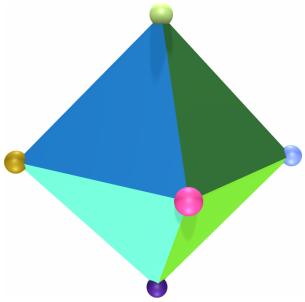


Figure 1.4: Octahedron, square dipyramid, triangular antiprism ($n_{vertices} = 6$)

1.2.4 Square antiprism

Yes – there is a Platonic solid with eight vertices; the regular hexahedron, aka the cube. But no – interestingly, this polyhedron simulation does not create a cube, if you ask it to arrange eight vertices in a minimum energy state. The astonishing result is the square antiprism shown in figure 1.5, which consists of two squares[3] (a "bottom square" and a "top square") that have been rotated (45°) into paraphase and eight triangles connecting the edges of the bottom square with the vertices of the top square and vice versa[4]. Obviously, this "twisting" of the squares leads to a stable equilibrium of the contradicting forces of the springs, while the cube, where the vertices of the squares would "face each other directly", produces an indifferent equilibrium that will never be reached in a simulation with random initial positions.

[3] You can easily identify the squares as two adjacent triangles of the same color (same plane normal).
[4] You have to try out this simulation! It is really cute to watch the vertices swarm around, until they have come to an agreement on which of them makes up the bottom and top squares.

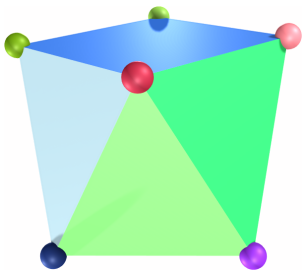


Figure 1.5: Not a cube, but a square antiprism ($n_{vertices} = 8$)

1.2.5 Icosahedron

In a simulation with twelve vertex spheres, all of them find their positions at the vertices of the regular icosahedron (Greek for "twenty faces") shown in figure 1.6, which is another one of the Platonic solids.

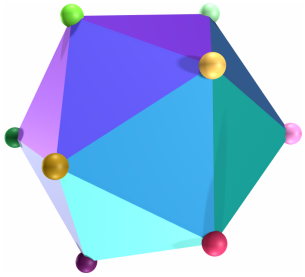


Figure 1.6: Icosahedron ($n_{vertices} = 12$)

1.2.6 Not a dodecahedron

The Platonic solid with 20 vertices and twelve (Greek: dodeca-) faces is called a regular dodecahedron. But again (as with the square antiprism) the 20 vertices minimum energy configuration depicted in figure 1.7 is not a dodecahedron. We have not even found a proper name for that irregular polyhedron. At first glance it seems to consist of triangles and three squares (adjacent triangles of the same color). But if you look more carefully at the "squares" you can discover that the colors of the adjacent triangles do not have the exact same color. Strange . . . !

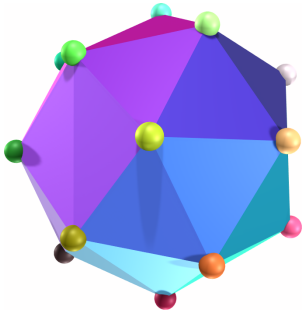


Figure 1.7: Not a dodecahedron ($n_{vertices} = 20$)

1.2.7 Corona

This program has been written in 2020 (the first year of the corona virus). The polyhedron with 128 vertices in figure 1.8 looks a little bit like the virus ... ☺

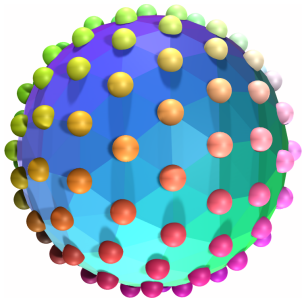


Figure 1.8: Corona ($n_{vertices} = 128$)

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).

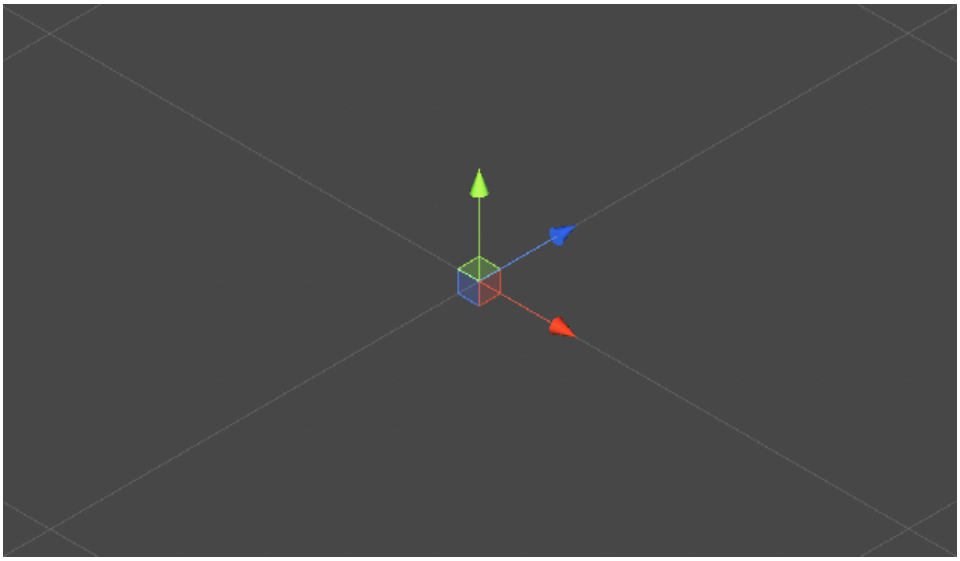


Figure 2.1: Coordinate system

2.2 Spheres

We model the vertices of the polyhedron as tiny spheres. Since the user is free to choose any polyhedron, we do not know the number of spheres we have to create. Therefore, we generate the spheres programmatically during the initialization of the program.

UNITY can create a few 3D primitives by itself:

- `Cube`

- `Sphere`
- `Capsule`
- `Cylinder`
- `Plane`
- `Quad`

Therefore, we can use a standard `Sphere` object as a prefab (figure 2.2)

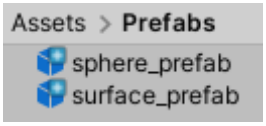


Figure 2.2: Standard `Sphere` object as a prefab

create a new empty `GameObject` called `Spheres` (figure 2.3)

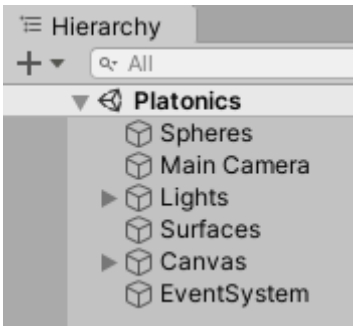


Figure 2.3: Empty `GameObject` (`Spheres`)

add a `Mesh Filter` and a `Mesh Renderer` as `Mesh` components to the empty `Spheres GameObject` (figure 2.4)

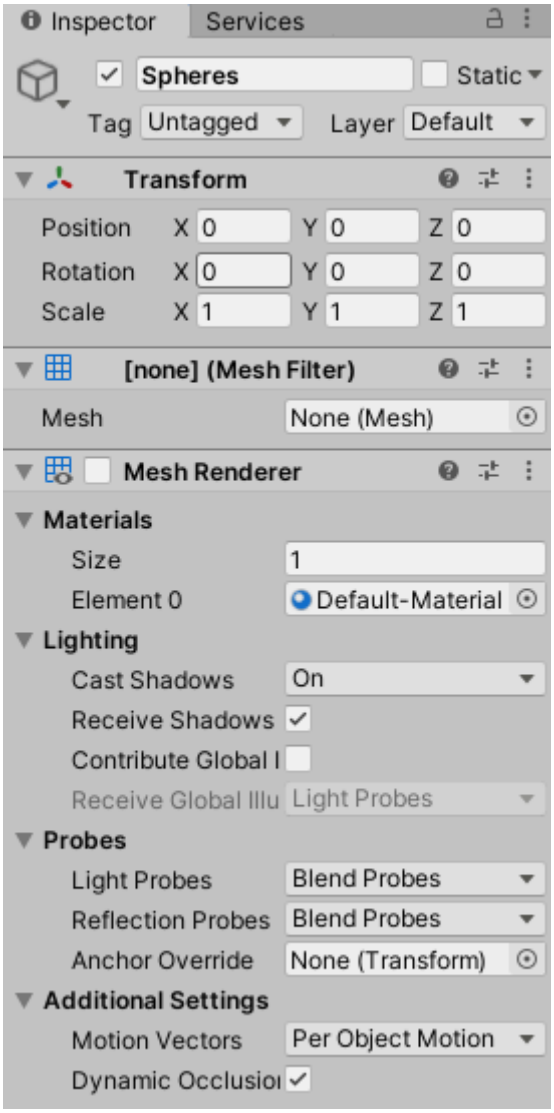


Figure 2.4: `Mesh Filter` and `Mesh Renderer` of `Spheres`

and attach a script by the name of `Spheres_class` (figure 2.5).

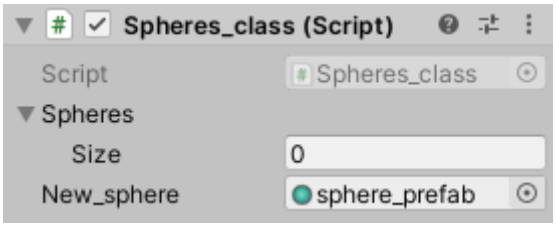


Figure 2.5: Script (`Spheres_class`)

In the script (section 2.2.1) we declare a `public GameObject` array of `spheres` that will contain the vertex sphere objects to be instantiated (and therefore has an initial size of

zero in figure 2.5) and the `public GameObject new_sphere` that we manually connect to the `sphere_prefab` in figure 2.5.

2.2.1 Spheres_class

The `Spheres_class` script is attached to the empty `Spheres GameObject`. It creates the spheres and the springs connecting the spheres to each other and the origin. In every simulation step, it computes the common center of mass of all spheres and colorizes the spheres according to their position with respect to the common center of mass.

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 some (global) objects and variables as properties. The array of `spheres` will be populated with the instantiated sphere objects:

```
public GameObject[] spheres;
```

The `new_sphere` object has been initialized with the `sphere_prefab` in figure 2.5 and will serve as an instantiation template for a new sphere:

```
public GameObject new_sphere;
```

The initial polyhedron is an icosahedron with an initial[1] vertex count of twelve:

```
public static int n_spheres = 12;
```

The `center` is computed in every simulation step as the current center of mass:

```
public static Vector3 center = new Vector3(0, 0, 0);
```

2.2.1.1 Start

The `start` function is called before the first simulation step. It creates the spheres and the springs:

```
void Start()
{
```

In the function, we create the `spheres` container as a new `GameObject` array

[1] Since we want to access the number of vertices in other classes as well, `n_spheres` has to be a `static` variable.

```
spheres = new GameObject[n_spheres];
```

and start a loop over every sphere to be created:

```
for (int i_sphere = 0; i_sphere < n_spheres; i_sphere++)
{
```

We create a new sphere in each loop pass using the `sphere prefab` (figure 2.5) and save it in the `spheres` array:

```
    spheres[i_sphere] = Instantiate(new_sphere);
```

We want to have the opportunity to drag every single sphere with the mouse. Therefore, we attach a `Drag_object` script (section 2.2.2) to every sphere:

```
    spheres[i_sphere].AddComponent<Drag_object>();
```

Even though it is not necessary for the function and it is only visible during the creation and debugging of the program, we give every sphere its own name (`Sphere_0`, `Sphere_1`, `Sphere_2`, ...)[2]

```
    spheres[i_sphere].name = "Sphere_" + i_sphere.ToString();
```

We initialize the positions of the spheres randomly inside of a sphere[3] with a radius of 10:

```
    spheres[i_sphere].transform.position =
    10 * Random.insideUnitSphere;
```

Every sphere is connected to the origin by a spring

```
    SpringJoint origin_spring =
    spheres[i_sphere].AddComponent<SpringJoint>();
```

with a fixed rest length of zero[4]:

```
    origin_spring.minDistance = 0;
    origin_spring.maxDistance = 0;
```

We want to manually attach the springs to the centers of the spheres (where the vertices of the polyhedron are located). Therefore, we switch off UNITY's automatic spring anchor point choice

```
    origin_spring.autoConfigureConnectedAnchor = false;
```

[2] Just like we do with our lawnmower robots and robot vacuum cleaners; it's not necessary at all, but it feels right.

[3] The interesting question "What happens if all spheres spawn at the same position (e. g. in the origin)?" is discussed in appendix A.

[4] Since the springs to the origin have a very small spring constant of 1 (compared to the springs between the spheres, having a spring constant of 10), the "unrealistic" rest length of zero will just slowly move the whole polyhedron back to the center of the screen after a disturbance.

and define both anchor points by ourselves:

```
    origin_spring.anchor = new Vector3(0, 0, 0);
    origin_spring.connectedAnchor = new Vector3(0, 0, 0);
```

A small spring constant of 1 tries to drag all spheres closer to the origin:

```
    origin_spring.spring = 1;
}
```

In the following double loop we define the springs that connect every sphere to every other sphere. The outer loop addresses every[5] sphere

```
for (int i = 0; i < n_spheres - 1; i++)
{
```

while the inner loop defines the connection partner[6] of the current sphere

```
    for (int j = i + 1; j < n_spheres; j++)
    {
```

Once again, we attach a spring to the current sphere

```
        SpringJoint inter_spring =
        spheres[i].AddComponent<SpringJoint>();
```

define "the other" sphere as the spring connection partner[7]

```
        inter_spring.connectedBody =
        spheres[j].GetComponent<Rigidbody>();
```

use 10 as the spring rest length

```
        inter_spring.minDistance = 10;
        inter_spring.maxDistance = 10;
```

manually connect the springs to the center of the spheres

```
        inter_spring.autoConfigureConnectedAnchor = false;
        inter_spring.anchor = new Vector3(0, 0, 0);
        inter_spring.connectedAnchor = new Vector3(0, 0, 0);
```

and give them a "stronger" spring constant of 10:

```
        inter_spring.spring = 10;
    }
}
}
```

[5] Except for the last one which has already been connected to all the others in the previous steps.

[6] A sphere should not be connected to itself and there should only be a spring connection from sphere A to sphere B (and not an additional "parallel" one from sphere B to sphere A).

[7] We did not have to do this with the origin springs. If omitted, the connected body of a spring is automatically set to the origin.

2.2.1.2 Update

The `Update` function is called in every simulation step. It computes the common center of mass of all the spheres and colorizes the spheres accordingly:

```
void Update()
{
```

In order to find the mean center of mass vector of all spheres, we add up the position vectors of all spheres

```
for (int i = 0; i < n_spheres; i++)
{
    center += spheres[i].transform.position;
}
```

and divide the sum by the number of spheres:

```
center /= n_spheres;
```

In Unity (and many other graphics programs), colors can be defined by three light intensity values (red, green, and blue, ranging from 0 to 1). According to the RGB color cube in figure 2.6, black `[0, 0, 0]` consists of no red, no green, and no blue light, while white `[1, 1, 1]` is created by full intensity red, green, and blue light. Therefore, e. g. pure red `[1, 0, 0]` contains no green and no blue light, while yellow `[1, 1, 0]` is a mixture of 100 % red and 100 % green light.

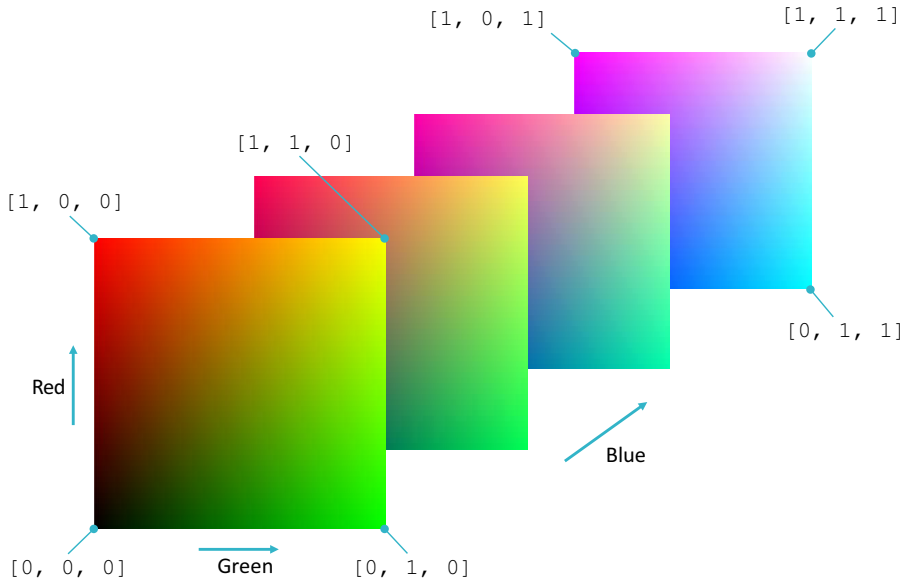


Figure 2.6: RGB color cube

We now want to colorize every sphere according to its relative position with respect to the center of the polyhedra. We start a loop over every sphere:

```
for (int i = 0; i < n_spheres; i++)
{
```

and compute its color via

```
    Vector3 color =
    (spheres[i].transform.position - center).normalized / 2  +
    new Vector3(1, 1, 1) / 2;
```

In detail: We calculate the vectorial distance between the current vertex and the center of the polyhedra (`spheres[i].transform.position - center`). Since this 3D-vector simply points from the center of the polyhedra to the current vertex, it can have any length. We make it a unit vector by dividing it by its norm (`.normalized`); this vector now describes any point on the unit sphere around the origin. Next, we cut the vector in half (`/ 2`); the vector now defines the red sphere in figure 2.7 with a radius of 0.5.

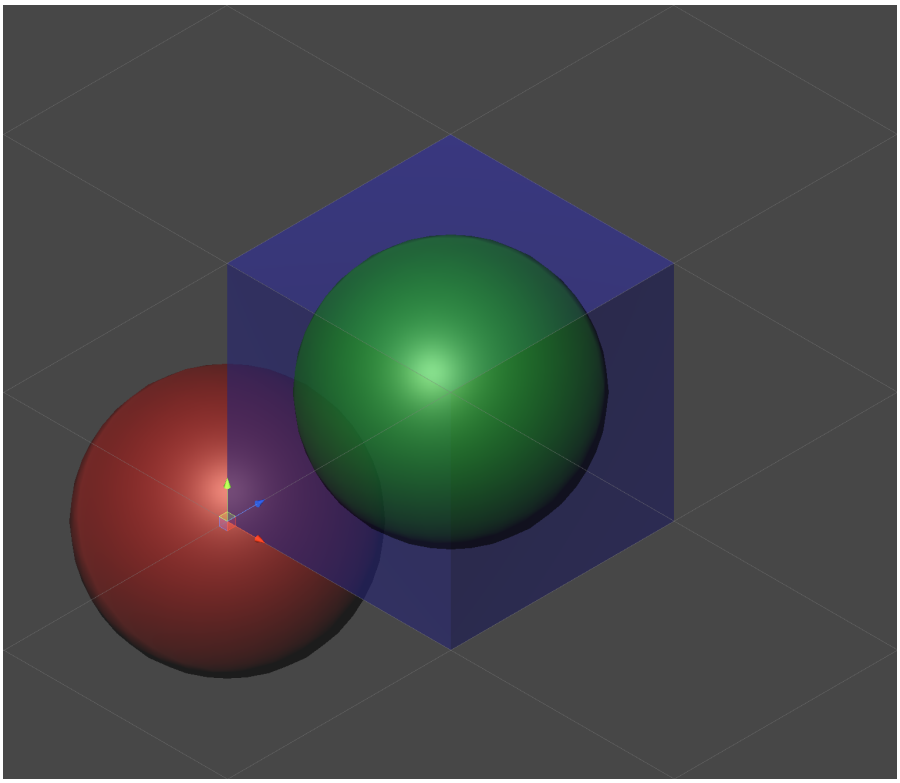


Figure 2.7: Translated color sphere

Since this vector might have negative components, we cannot directly use it as a color. Therefore, we translate the vector 0.5 along every coordinate axis, basically moving it along the spacial diagonal (`+ new Vector3(1, 1, 1) / 2`). The vector now defines the green sphere in figure 2.7 which is the inner sphere of the blue cube in figure 2.7. Since the blue cube in figure 2.7 extends from `[0, 0, 0]` to `[1, 1, 1]`, it represents the full color cube of figure 2.6. As a consequence, the green color sphere in figure 2.7

cannot reach colors in the eight vertices of the (blue) color cube in figure 2.7. The vertex spheres of the polyhedron can therefore not have pure colors like black, white, red, green, blue, yellow, cyan, and magenta; their colors are of an appealing looking pastel shade (figure 1.8).

Finally, we use the computed color value to colorize the current sphere:

```
    spheres[i].GetComponent<MeshRenderer>().material.color =
        new Color(color.x, color.y, color.z);
}
}
}
```

2.2.2 Drag_object

The `Drag_object` script is attached to every sphere during its creation (section 2.2.1.1), allowing the user to select and drag every sphere with her mouse. We use Matthias Pieroth's template [5], which works out the box: The class

```
using UnityEngine;

public class Drag_object : MonoBehaviour
{
```

declares an offset between the position of the selected game object and the mouse position when the user presses the mouse button in world coordinates

```
private Vector3 mOffset;
```

and the z-coordinate of the game object in screen coordinates:

```
private float mZCoord;
```

2.2.2.1 OnMouseDown

If the user clicks the object

```
void OnMouseDown()
{
```

the current z-coordinate of the game object in screen coordinates is saved

```
mZCoord =
Camera.main.WorldToScreenPoint(gameObject.transform.position).z;
```

and the current distance between the position of the game object and the mouse position in world coordinates is calculated:

```
mOffset = gameObject.transform.position - GetMouseAsWorldPoint();
}
```

2.2.2.2 GetMouseAsWorldPoint

`OnMouseDown` (section 2.2.2.1) uses the function

```
private Vector3 GetMouseAsWorldPoint()
{
```

that returns the current mouse position in world coordinates using the z-coordinate of the selected object as the z-coordinate of the mouse position. This allows the user to drag the object in x- and y-directions on the screen.[8] The function reads the current mouse position in screen coordinates

```
Vector3 mousePoint = Input.mousePosition;
```

substitutes the z-coordinate of the mouse position by the z-coordinate of the object

```
mousePoint.z = mZCoord;
```

and returns the modified mouse position in world coordinates:

```
return Camera.main.ScreenToWorldPoint(mousePoint);
}
```

2.2.2.3 OnMouseDrag

If the user moves the mouse while keeping the button pressed

```
void OnMouseDrag()
{
```

the current mouse position in world coordinates is computed, translated by the initial distance between object and mouse, and finally used as the new position of the object:

```
transform.position = GetMouseAsWorldPoint() + mOffset;
}
}
```

2.3 Surfaces

The surfaces of the polyhedron are triangles, each connecting three vertices. We could create and display all possible triangles formed by three vertices each: The inner vertices would not be visible; we would automatically see the hull of the polyhedron only. While this method would produce acceptable results for polyhedrons with a very small number

[8] Yes, it would be nice to control the z-coordinate of the object too. But since our traditional mouse can only move in two directions (not every user owns a space mouse [6]), we are reasonably satisfied with the x-y-drag of the spheres on the flat screen.

of vertices (tetrahedron, octahedron, ...), the number of triangles would explode for a larger number of vertices: The corona polyhedron (figure 1.8) with 128 vertices e. g. would consist of more than a quarter of a million inner (and therefore useless) triangles:

$$\texttt{nchoosek}(128, 3) = \texttt{binomial}(128, 3) = \binom{128}{3} = \frac{128!}{3! \cdot 125!} = 341\,376 \tag{2.1}$$

While modern browsers would still display and even simulate such a triangle monster, their frame rate would drastically drop to non-acceptable values even on high-performance machines.

The alternative is to calculate the 3D convex hull[9] of the polyhedron which consists of only a few triangles: Euler's polyhedron formula [7] states that the number of vertices n, the number of edges e, and the number of faces f of any polyhedron satisfy the following equation:

$$v - e + f = 2 \tag{2.2}$$

If the polyhedron is triangulated (i. e. its hull consists of only triangles) every triangle has tree edges. Therefore, the number of edges would be three times the number of faces if all faces were separated from each other. But, in a closed polyhedron every edge is shared between two faces, cutting the number of edges in half:

$$e = \frac{3}{2} f \tag{2.3}$$

If we use equation (2.3) in equation (2.2) we can find a linear relation between the number of faces with respect to the number of vertices in a triangulated polyhedron:

$$\begin{aligned} v - \frac{3}{2} f + f &= 2 \\ v - \frac{1}{2} f &= 2 \\ f &= 2v - 4 \end{aligned} \tag{2.4}$$

Using equation (2.4) for the corona polyhedron, the number of faces of its convex hull can never exceed[10] 252

$$f_{corona} = 2v_{corona} - 4 = 2 \cdot 128 - 4 = 252$$

which is significant less than the number 341 376 in equation (2.1) and quite acceptable for a dynamic simulation in a browser.

Unfortunately, finding the 3D convex hull of a given set of points is not a trivial task at all [8]. Fortunately, Oskar Sigvardsson wrote a very well documented "implementation

[9] Shrink a rubber sheet around all vertices.

[10] The number of convex hull faces of the corona polyhedron can easily be less than 252: Imagine the four "outermost" vertices forming a tetrahedron and all the other vertices lying inside this tetrahedron. The convex hull will then just consist of the four triangular tetrahedron faces.

of the Quickhull algorithm for generating 3d convex hulls from point clouds, written for Unity." [9] that works perfectly straight out of the box (section 2.3.2). The convex hull algorithm returns the vertices, triangles, and normals of a single mesh defining the convex hull. Unfortunately, UNITY does not easily[11] allow the triangles of a single mesh to have different colors. Therefore, we will use one mesh per triangle, which is much more of a burden for the GPU, but since the convex hull only consist of a few faces ...

The dynamic creation of the surfaces is very similar to the creation of the spheres:

We use a standard `Surface` object as a prefab (figure 2.2), create a new empty `GameObject` called `Surfaces` (figure 2.3), add a `Mesh Filter` and a `Mesh Renderer` as `Mesh` components to the empty `Surfaces GameObject` (figure 2.8),

[11] Yes, there are special particle shaders that can render the faces of a mesh in different colors, but these shaders might run into other problems regarding shadows, reflections, transparency, ...

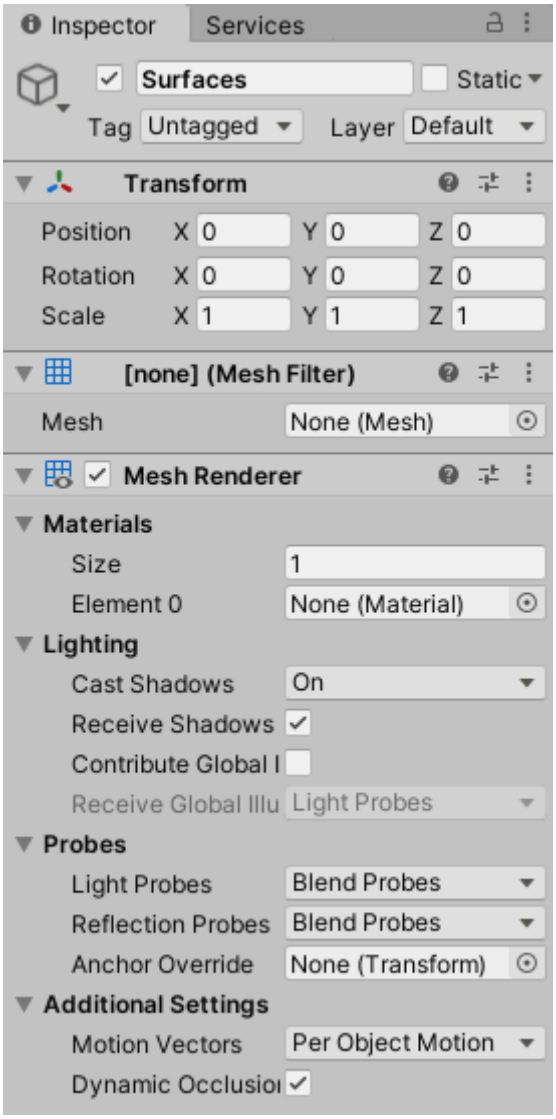


Figure 2.8: `Mesh Filter` and `Mesh Renderer` of `Surfaces`

and attach a script by the name of `Surfaces_class` (figure 2.9).

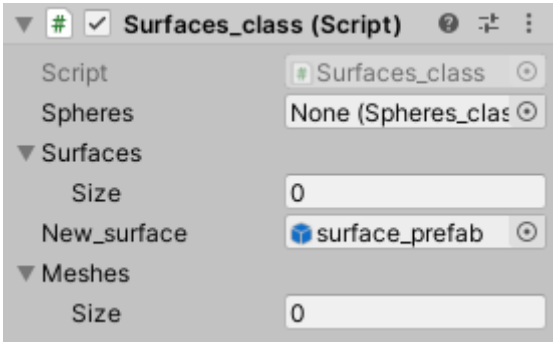


Figure 2.9: Script (`Surfaces_class`)

In the script (section 2.3.1) we declare a `public GameObject` array of `surfaces` that will contain the surface objects to be instantiated (and therefore has an initial size of zero in section 2.3.1) and the `public GameObject new_surface` that we manually connect to the `surface_prefab` in figure 2.9.

2.3.1 `Surfaces_class`

This class imports the necessary namespaces (including the `GK` namespace of the convex hull class (section 2.3.2))

```
using GK;
using System.Collections.Generic;
using UnityEngine;

public class Surfaces_class : MonoBehaviour
{
```

and declares its properties: The `spheres` object is a reference to the array holding all the spheres. We will use it to access the `Spheres_class` component in order to read the current number of spheres:

```
private GameObject spheres;
public Spheres_class Spheres;
```

The `surfaces` array will be populated by all the surfaces to be created

```
public GameObject[] surfaces;
```

while `new_surface` will use its connection to the `surface_prefab` (figure 2.9) to instantiate a new surface

```
public GameObject new_surface;
```

and `n_surfaces` will be computed as the number of surfaces via equation (2.4):

```
private int n_surfaces;
```

`vertices`, `triangles`, and `normals` will describe one of the surfaces

```
private Vector3[] vertices;
private int[] triangles;
private Vector3[] normals;
```

and the `meshes` array will hold the meshes of the surfaces:

```
public Mesh[] meshes;
```

The following properties are used by the convex hull generator:

```
private ConvexHullCalculator calc;
private List<Vector3> verts;
private List<int> tris;
private List<Vector3> norms;
private List<Vector3> points;
```

2.3.1.1 Start

During the initialization

```
void Start()
{
```

we create the objects necessary for the convex hull computation

```
calc = new ConvexHullCalculator();
points = new List<Vector3>();
verts = new List<Vector3>();
tris = new List<int>();
norms = new List<Vector3>();
```

find the game object that holds the spheres

```
spheres = GameObject.Find("Spheres");
```

access its class component

```
Spheres = spheres.GetComponent<Spheres_class>();
```

and use the number of spheres defined in the class to compute the (maximum) number of surfaces according to equation (2.4):

```
n_surfaces = 2 * Spheres_class.n_spheres - 4;
```

We can now define an array of game objects that can hold the maximum[12] number of surfaces to expect:

[12] Yes, we could create and destroy surface objects on the fly as we need them; but since we know the maximum number of surfaces and this number is not too high, it might be more performant to only have to create the surfaces once and make a few of them invisible if we do not need them.

```
surfaces = new GameObject[n_surfaces];
```

Since we want to access the meshes of the surfaces, we create an equally sized array of meshes:

```
meshes = new Mesh[n_surfaces];
```

Since we want to create single triangle meshes, each mesh has three vertices, each triangle is defined by three vertices, and there is a normal vector in each vertex:

```
vertices = new Vector3[3];
triangles = new int[3];
normals = new Vector3[3];
```

We are now ready to create the surfaces in a loop:

```
for (int i_surface = 0; i_surface < n_surfaces; i_surface++)
{
```

We instantiate the current surface

```
    surfaces[i_surface] = Instantiate(new_surface);
```

give it an individual name (footnote (2))

```
    surfaces[i_surface].name = "Surface_" + i_surface.ToString();
```

and copy its mesh into the mesh array:

```
    meshes[i_surface] =
    surfaces[i_surface].GetComponent<MeshFilter>().mesh;
}
}
```

2.3.1.2 Update

In every simulation step

```
void Update()
{
```

we wipe the `points` list of the convex hull clean

```
points.Clear();
```

and add the current position of every sphere to the list:

```
for (
int i_sphere = 0;
i_sphere < Spheres_class.n_spheres;
i_sphere++)
{
    points.Add(Spheres.spheres[i_sphere].transform.position);
}
```

We can now forward the point list to the convex hull generator (section 2.3.2) and ask it to compute the mesh of the hull:

```
calc.GenerateHull(points, true, ref verts, ref tris, ref norms);
```

The generator provides three vertex indices for every triangle, allowing us to determine the number of triangles found:

```
int n_triangles = tris.Count / 3;
```

In a loop over all triangles of the convex hull

```
for (int i_triangle = 0; i_triangle < n_triangles; i_triangle++)
{
```

we make these triangles visible:

```
    surfaces[i_triangle].SetActive(true);
```

Every surface consists of three vertices (three entries in the `verts` list), three vertex indices (three entries in the `tris` list) and one normal vector in each vertex (three entries in the `norms` list). Therefore, we start loop over the three vertices of the current surface:

```
    for (int i_vertex = 0; i_vertex < 3; i_vertex++)
    {
```

In the loop we copy the entries from the lists (`verts`, `tris`, `norms`) provided by the hull generator[13] to the corresponding components of the `vertices`, `triangles`, and `normals` arrays

```
        vertices[i_vertex] =
        verts[tris[3 * i_triangle + i_vertex]];

        triangles[i_vertex] =
        tris[3 * i_triangle + i_vertex] % 3;

        normals[i_vertex] =
        norms[3 * i_triangle + i_vertex];
    }
```

and use the current `vertices`, `triangles`, and `normals` arrays for the definition of the current surface mesh:

```
    meshes[i_triangle].vertices = vertices;
    meshes[i_triangle].triangles = triangles;
    meshes[i_triangle].normals = normals;
```

[13] The three returned lists define one big mesh; We want to color each surface individually and therefore have to split up the mesh into an array of single meshes.

We want to use the normals of the triangles to define their colors. In UNITY, every vertex has its own normal, but the hull generator provides all vertices of a single triangle with the same normals. Therefore, we just use the normal of the "first" vertex of the current mesh:

```
    Vector3 normal = meshes[i_triangle].normals[0];
```

The color of the current triangle is defined similar[14] to the colorization of the spheres in section 2.2.1.2:

```
    Vector3 color =
    -(normal).normalized / 2 + new Vector3(1, 1, 1) / 2;
```

We can now colorize the current surface:

```
    surfaces[i_triangle].GetComponent<MeshRenderer>().material.↩
       color =
        new Color(color.x, color.y, color.z);
}
```

Since we declared the array of surfaces with the maximum number of surfaces in mind (footnote (10)), the final step is to make the currently unused surfaces invisible:

```
for (
int i_triangle = n_triangles;
i_triangle < n_surfaces;
i_triangle++)
{
    surfaces[i_triangle].SetActive(false);
}
}
}
```

2.3.2 ConvexHullCalculator

Oskar Sigvardsson wrote about his convex hull calculator:

"An implementation of the quickhull algorithm for generating 3d convex hulls.

The algorithm works like this: you start with an initial 'seed' hull, that is just a simple tetrahedron made up of four points in the point cloud. This seed hull is then grown until it all the points in the point cloud is inside of it, at which point it will be the convex hull for the entire set.

All of the points in the point cloud is divided into two parts, the 'open set' and the 'closed set'. The open set consists of all the points outside of the tetrahedron, and the closed

[14] Using a negative sign gives the surface the same color as its opposite (diametral) vertex sphere. This becomes most obvious in the tetrahedron (figure 1.2).

set is all of the points inside the tetrahedron. After each iteration of the algorithm, the closed set gets bigger and the open set get smaller. When the open set is empty, the algorithm is finished."[9]

The algorithm is well documented and very reliable.

```
public void GenerateHull(
  List<Vector3> points,
  bool splitVerts,
  ref List<Vector3> verts,
  ref List<int> tris,
  ref List<Vector3> normals)
```

Its input parameters are the point cloud (`points`) as a list of 3D-vectors and the boolean `splitVerts` that decides whether every triangle should[15] have its own vertices.

Since the output parameters `verts`, `tris`, and `normals` are explicitly called by reference, we can directly address and reuse them without any extra memory overhead.

2.4 Camera

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

2.4.1 Mouse Orbit

For this purpose, we attach the `Camera_class`

```
using UnityEngine;

public class Camera_class : MonoBehaviour
{
```

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

```
float distance = 25;
```

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

```
readonly float speed_x = 3f;
readonly float speed_y = 3f;
readonly float speed_zoom = 20f;
```

[15] If two faces of a mesh share their vertices, UNITY tries to smooth the corresponding edge, which would be useful if we wanted to create a smooth surface. In our case we want single, separated triangles and call the hull generator with the `splitVerts` parameter set to `true` (section 2.3.1.2).

and the constant distance limits:

```
readonly private float distance_min = 25f;
readonly private float distance_max = 100f;
```

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 = 300f;
private float elevation = 0f;
```

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 [10] 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;
}
}
```

2.5 Canvas

We want to give the user the choice between different polyhedrons by offering her a button list (figure 1.1). Buttons are children of a canvas. Therefore, we use a `Canvas GameObject` with the default `Screen Space Overlay Render Mode` that keeps the canvas fixed on the screen even if we move the camera around (figure 2.10).

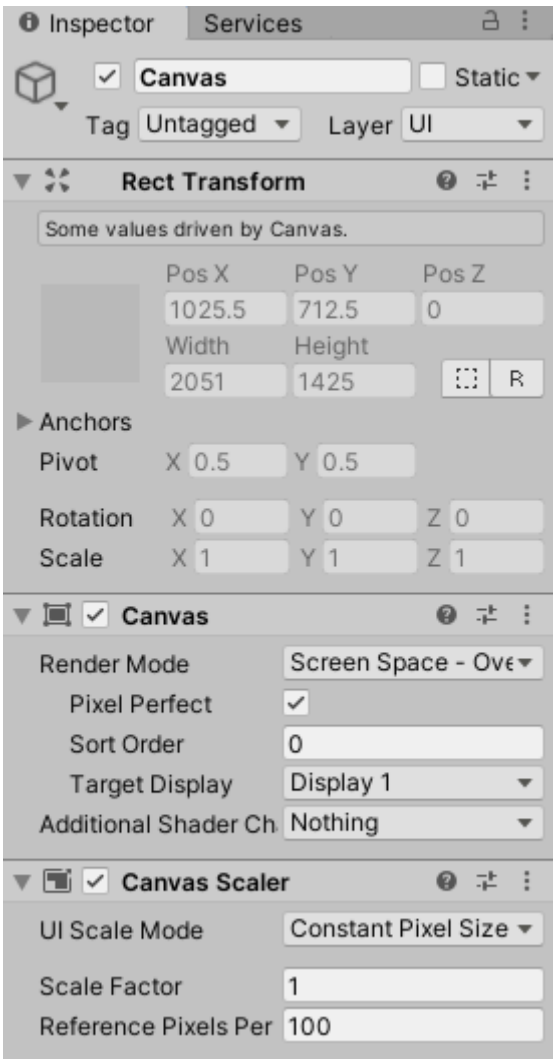


Figure 2.10: Canvas

We tell the `Canvas Scaler` to use a `Constant Pixel Size` as its `UI Scale Mode` which allows the button list (figure 1.1) to scale with the browser zoom level.

2.5.1 Buttons

We define seven `Button GameObjects` as children of the `Canvas` and name them `4`, `5`, `6`, `8`, `12`, `20`, and `128` (figure 2.11).

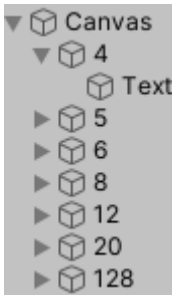


Figure 2.11: `Buttons` as children of the `Canvas`

Every button gets a width of 120 and a height of 30 pixels and an appropriate position with respect to the bottom center of the canvas (figure 2.12).

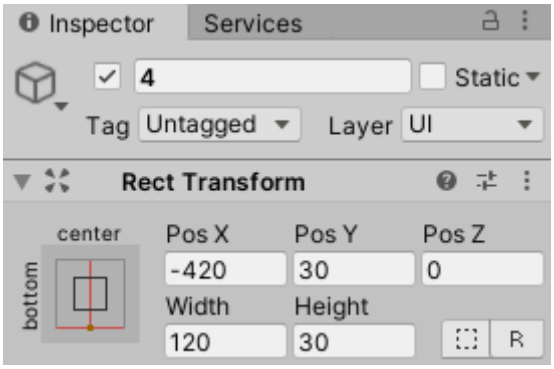


Figure 2.12: Button size and position

Additionally, we define the `On Click` event of the button by calling the `Restart_Scene` method of the `Button_clicked` class (figure 2.13).

Figure 2.13: Button `On Click` event

2.5.1.1 `Button_clicked`

The `Button_clicked` class additionally imports the `SceneManagement` namespace

```
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
public class Button_clicked : MonoBehaviour
{
```

and defines the `Restart_Scene` method that is called when the user presses the corresponding button:

```
public void Restart_Scene()
{
```

In the method, we reload the whole scene

```
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
```

and interpret the name[16] of the button as the number of vertices of the polyhedron to be created:

```
Spheres_class.n_spheres = int.Parse(name);
}
}
```

2.5.1.2 Button text

As indicated in figure 2.11, every button automatically gets a `Text` object as a child, that we provide with the corresponding information (figure 2.14)

Figure 2.14: Button text

2.6 Lights

For the illumination of the scene, we introduce six `Directional Light GameObjects` (figure 2.15)

[16] Using the name of an object for the transfer of relevant information might probably not win the first price in the Best Object Oriented Programming Style competition – but it does the job.

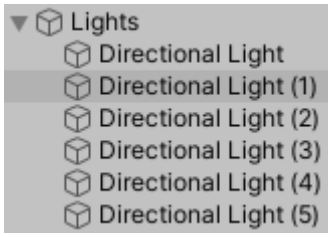


Figure 2.15: Six directional lights

pointing in all positive and negative axis directions (figure 2.16) leading to quite realistic shadows and reflections (figure 1.6).

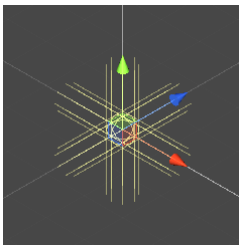
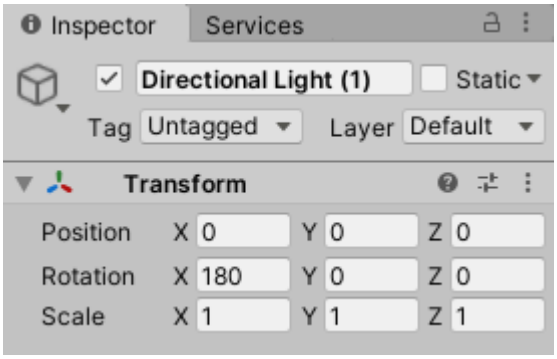


Figure 2.16: Directions of the lights

We achieve the different light directions by rotating the default light - pointing in the positive z-direction (figure 2.17a) - about appropriate axes (e. g. figure 2.17b).

(a) Light in positive z-direction



(b) Light in negative z-direction

Figure 2.17: 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 2.17), the hulls of the polyhedrons are also illuminated “on the outside”.

Bibliography

[1] J. J. Buchholz. (2020) Platonics. [Online]. Available: https://m-server.fk5.hs-bremen.de/platonics/index.html

[2] Wikipedia. (2020) Polyhedra. [Online]. Available: https://en.wikipedia.org/wiki/Polyhedron

[3] ——. (2020) Platonic Solid. [Online]. Available: https://en.wikipedia.org/wiki/Platonic_solid

[4] Unity. (2020) Unity. [Online]. Available: https://unity.com

[5] Jayanam. (2020) Unity 3D tutorial : Drag Gameobject with mouse. Jayanam Games. [Online]. Available: https://www.patreon.com/posts/unity-3d-drag-22917454

[6] 3Dconnexion. (2020) Spacemouse. 3Dconnexion. [Online]. Available: https://3dconnexion.com/de/spacemouse/

[7] Wikipedia. (2020) Euler characteristic. [Online]. Available: https://en.wikipedia.org/wiki/Euler_characteristic

[8] L. Toma. (2020) 3D convex hulls. Bowdoin College. [Online]. Available: http://www.bowdoin.edu/~ltoma/teaching/cs3250-CompGeom/spring17/Lectures/cg-hull3d.pdf

[9] Oskar Sigvardsson. (2020) unity-quickhull. GitHub. [Online]. Available: https://github.com/OskarSigvardsson/unity-quickhull

[10] Wikipedia. (2020) Gimbal lock. [Online]. Available: https://en.wikipedia.org/wiki/Gimbal_lock

[11] ——. (2020) Z-fighting. [Online]. Available: https://en.wikipedia.org/wiki/Z-fighting

[12] ——. (2020) Quickhull. [Online]. Available: https://www.wikiwand.com/en/Quickhull

[13] ——. (2020) PhysX. [Online]. Available: https://en.wikipedia.org/wiki/PhysX

A Initial spheres positions

What happened if we initially positioned all spheres (vertices of the polyhedron) at the origin?

If we use

```
spheres[i_sphere].transform.position = new Vector3(0, 0, 0);
```

instead of

```
spheres[i_sphere].transform.position =
10 * Random.insideUnitSphere;
```

in section 2.2.1.1, all spheres are initially positioned at $\begin{bmatrix}0 & 0 & 0\end{bmatrix}$ and the springs connecting the spheres immediately start pushing them apart along the x-axis[1] until they reach their final positions along a straight line (figure A.1).

Figure A.1: All twelve spheres are positioned along the x-axis.

Even though this indifferent[2] configuration is not an energy-optimal equilibrium, the NVIDIA PHYSX engine [13] that UNITY uses for the simulation of its 3D-physics is stable enough to keep the state of figure A.1 infinitely[3] long.

If we now grab one of the spheres with the mouse and move it up or down or if we use initial random positions in the x-y-plane

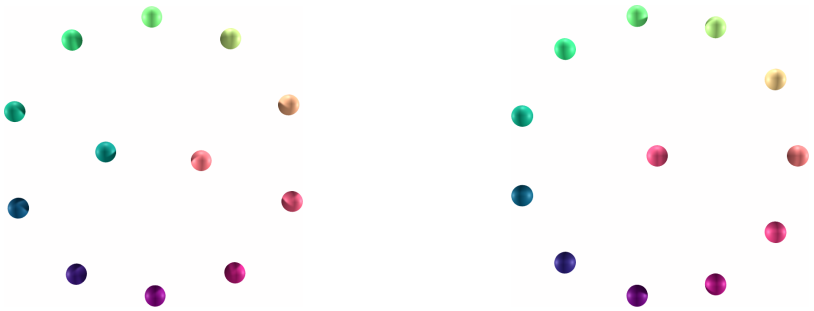
```
spheres[i_sphere].transform.position =
new Vector3(Random.value, Random.value, 0);
```

all spheres magically arrange themselves in the x-y-plane into one of the stable configurations depicted in figure A.2.

[1] The fact that the springs only push in x-direction seems pretty arbitrary ... One of the many unsolved UNITY mysteries ...

[2] Think of a perfect sphere on a perfect pinhead: As long as you do not breathe and not even think about giving the sphere the slightest push, the sphere could balance on the pinhead forever.

[3] The inappropriate use of the word "infinitely" is typical engineer's hubris. Numerical rounding errors should definitely destabilize the indifferent equilibrium after a while. On the other hand, even after "many" minutes of simulation, all spheres remain in the straight line configuration of figure A.1.



(a) Two center spheres (b) One center sphere

Figure A.2: All spheres are positioned in the x-y-plane.

We can manually switch between figure A.2a and figure A.2b by dragging spheres from the “circle” to the “center” and vice versa. We could not manually create configurations with less than one or more than two center spheres.

If we use 20 (or more) spheres, stable arrangements with two (or more) outer circles are possible (figure A.3).

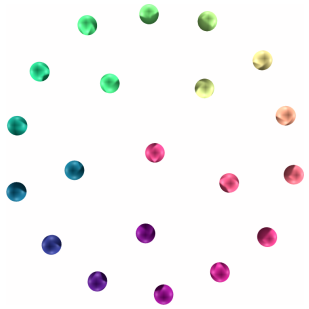


Figure A.3: Stable configuration of 20 spheres in the x-y-plane

All previously described linear or planar sphere arrangements have one major drawback: The convex hull generator (section 2.3.2) relies on points defining a true 3D-space and therefore complains if all points lie on a 2D-plane or even on a 1D-line: ”`Can't generate hull, points are coplanar.`” in every simulation step.