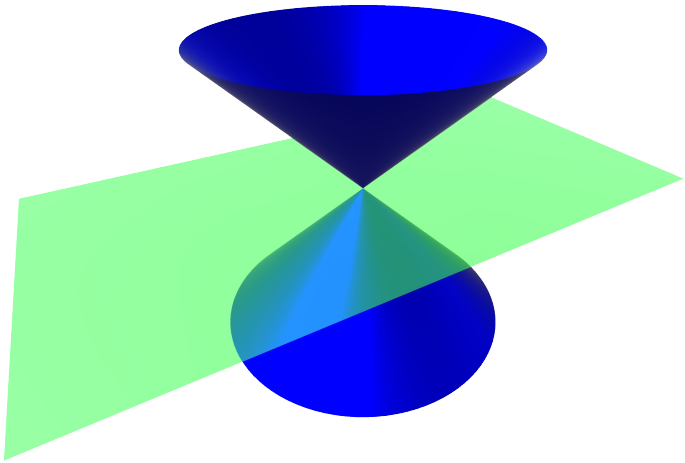


Conic Sections

Interactive Website

Jörg J. Buchholz

June 24, 2021

1 Manual

1.1 Introduction

In this paper, we describe the utilization and genesis of an interactive website [1] you can use to create and display different conic sections.

A conic section (conic) result from a plane (section 2.3) intersecting a double cone (section 2.2). Depending on the height and the angle of the plane, the conic can be an ellipse, a parabola, or a hyperbola.

You can use the keyboard keys `W` and `S` to move the plane up and down. `A` and `D` rotate the plane (section 2.3.1). Pressing the left mouse button, you can orbit the camera around the scenery (section 2.4). With the mouse wheel you can zoom in and out.

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

1.2 Ellipse

We get an ellipse (figure 1.1) if the plane angle is less than half the cone opening angle.

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

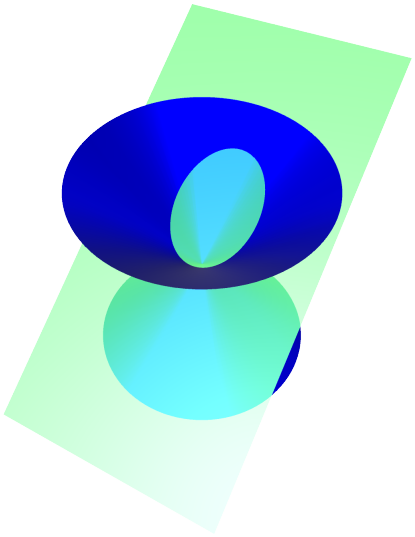


Figure 1.1: Ellipse

1.2.1 Circle

The circle (figure 1.2) is a special case of an ellipse if the plane angle is zero.

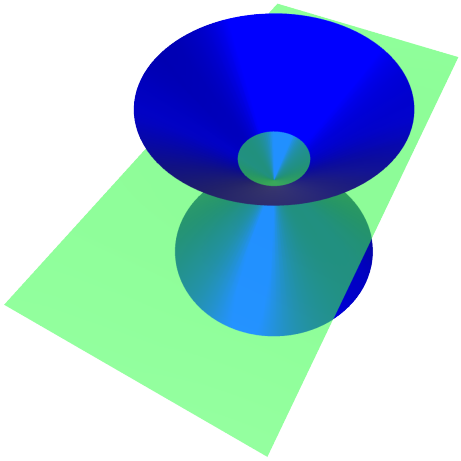


Figure 1.2: Circle

1.2.2 Point

The ellipse degenerates[2] into a point (figure 1.3) if the height of the plane is zero, i. e. the plane intersects the twin tip of the double cone.

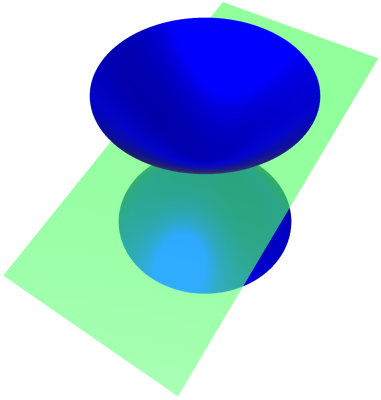


Figure 1.3: Point (invisible)

1.3 Parabola

We get a parabola (figure 1.4) if the plane angle equals half the cone opening angle, i. e. if the plane is parallel to the surface of the cone.

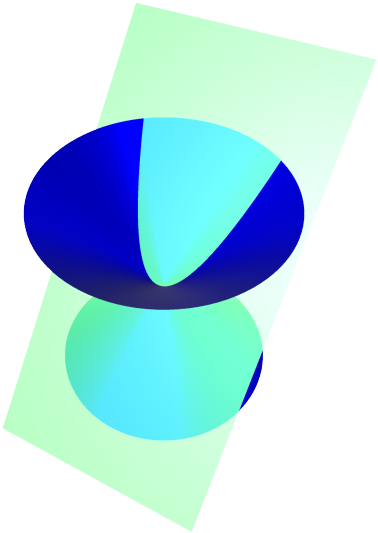


Figure 1.4: Parabola

[2] We cannot see the point in figure 1.3 since a point is just a mathematical object and does not have a visible area. Seen from the graphic card's point of view, we run into the problem of z-fighting [3]: Since the point now belongs to both the double cone and the plane, the graphic card arbitrarily has to decide whether to render the single pixel blue or green.

1.3.1 One straight line

If – additionally – the height of the plane is zero, the parabola degenerates[3] into a single straight line (figure 1.5).

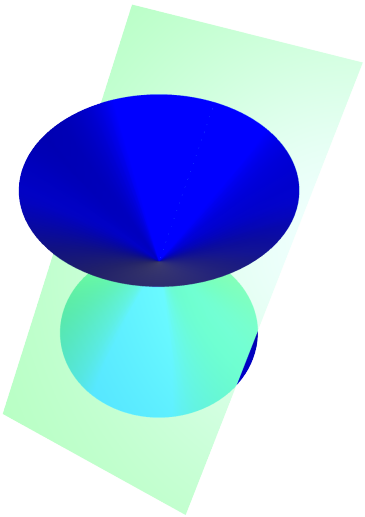


Figure 1.5: One straight line (nearly invisible)

1.4 Hyperbola

We get a hyperbola (figure 1.6) if the plane angle is greater[4] than half the cone opening angle.

[3]The straight line is part of the double cone as well as the plane. If you zoom deeply into the upper cone in figure 1.5 you might see a few green pixel where the graphic card has decided to let the plane shine through.

[4]If the plane angle is only slightly greater than half the cone opening angle, the plane does not visibly intersect the lower cone in the program, hiding the lower branch of the hyperbola. In reality, the cones are infinitely extended and do very well intersect the plane.

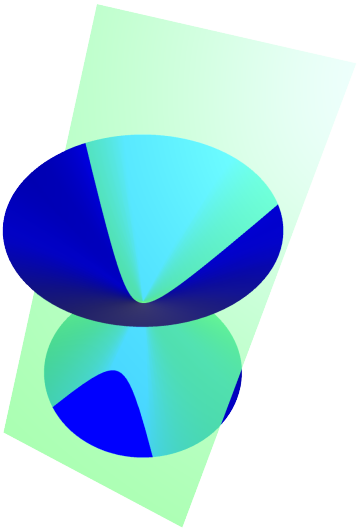


Figure 1.6: Hyperbola

1.4.1 Two straight lines

If – additionally – the height of the plane is zero, the hyperbola degenerates into two straight lines (figure 1.7).

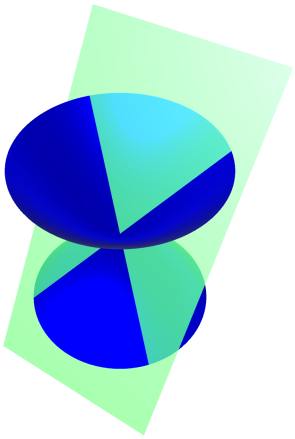


Figure 1.7: Two straight lines

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).

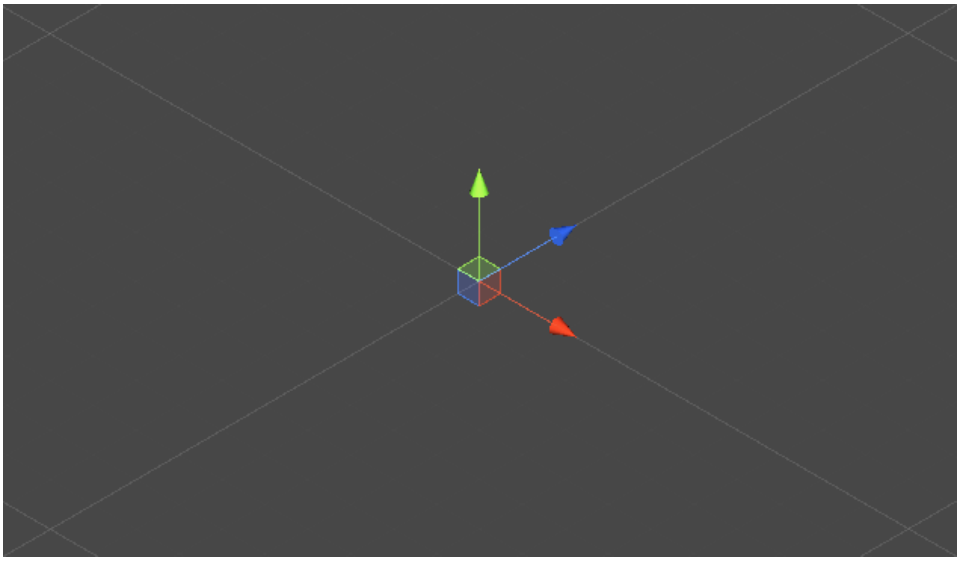


Figure 2.1: Coordinate system

2.2 Cone

We model the double cone as two single cones with a height of 1 and an opening angle of 90°. Then, we move the lower cone 1 unit down and the upper cone 1 unit up and rotate the upper cone 180° around the Z-axis. This generates the double cone in figure 2.2 with its twin tip in the origin.

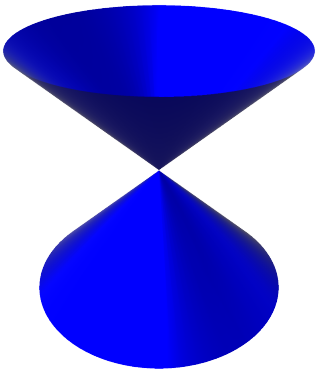


Figure 2.2: Double cone

UNITY can only create a few 3D primitives by itself:

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

We could model the cone in another 3D modeling program like SKETCHUP [4] and import it as a new object into UNITY. However, we generate the cone programmatically as a `mesh` with `vertices` and `triangles`. We create a new empty `GameObject` in UNITY, add a `Mesh Filter` and a `Mesh Renderer` as `Mesh` components, dress it with a blue material and add a `Script` by the name of `Cone` (figure 2.3).

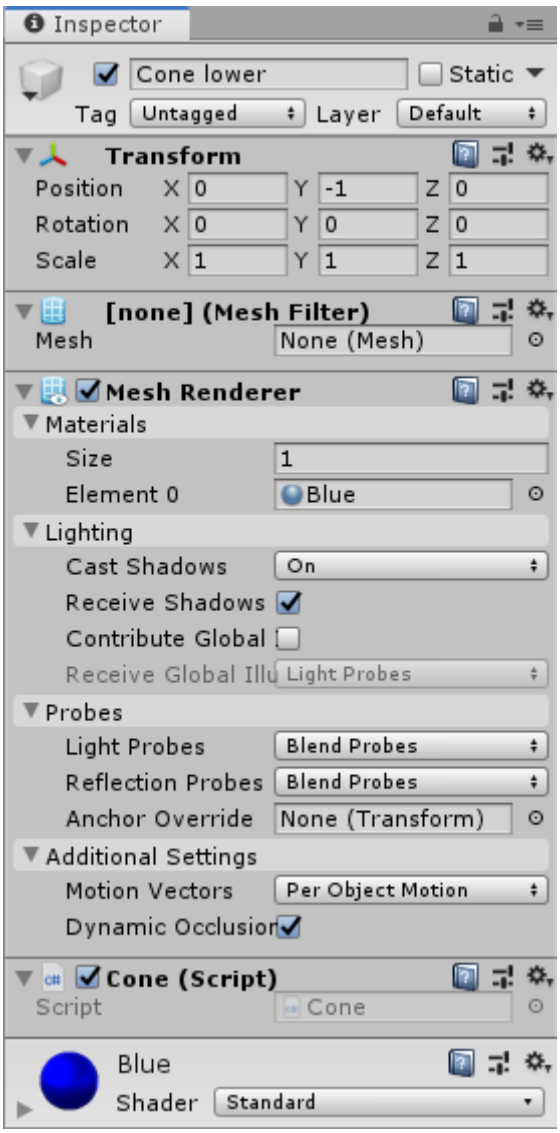


Figure 2.3: Cone lower

2.2.1 Cone class

UNITY scripts automatically import some standard types from predefined namespaces:

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

Every UNITY script derives from the `MonoBehaviour` base class:

```
public class Cone : MonoBehaviour
{
```

Before we define the `Start` function, we declare some (global) objects. The `cone` itself is a `Mesh` object

```
Mesh cone;
```

that is defined by a vertex array of 3D points and a triangle index array of integers:

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

Additionally, we declare the arrays of vertices and triangles of "the other side":[1]

```
Vector3[] vertices_double_sided;
int[] triangles_double_sided;
```

The `start` function is called before the first simulation step:

```
void Start()
{
```

In the function, we create the `cone` object as a new `Mesh`

```
cone = new Mesh();
```

and define it to be the `mesh` used by the `MeshFilter`:

```
GetComponent<MeshFilter>().mesh = cone;
```

For reasons of clarity, we outsource the rest of the cone creation into three additional functions:

```
CreateShape();

MakeDoublesided();

UpdateMesh();
}
```

2.2.2 CreateShape

The `CreateShape` function

```
void CreateShape()
{
```

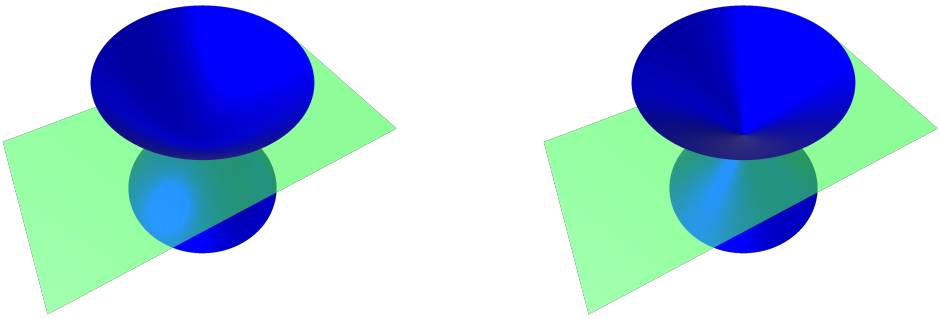
defines the vertices and the triangles that make up the cone. We want the base[2] of the cone to be a regular polygon with 100 vertices:

[1] Defining double-sided meshes in Unity is a bit awkward. We could either use special double-sided shaders, that unrealistically duplicate the light conditions from the front face to the back face or we can (and do) duplicate every triangle giving its vertices the opposite normal vectors in section 2.2.3.

[2] As seen from a mathematical point of view, the base of a cone is a circle. In computer graphics we approximate a circle by a regular polygon.

```
const int n_vertices_base = 100;
```

UNITY pursues an interesting concept regarding edge smoothing: If two triangles reuse the same two vertices, the corresponding edge is smooth; if both triangles use their own vertices (even if they have the same coordinates), the edge is sharp. Therefore, we run into a dilemma: We want the surface of the cone to be smooth but the tip of the cone to be distinctive. If we reused a single tip vertex for all surface triangles, the tip would be rendered unrealistically smooth (figure 2.4a).



(a) Single tip vertex (b) Multiple tip vertices

Figure 2.4: Multiple vertices (figure 2.4b) make the tip of the cone look more realistic.

We want the tip to look more like figure 2.4b. We can achieve this by defining as many tip vertices (with identical coordinates) as there are base vertices and use a separate tip vertex for each surface triangle. Therefore, the total number of vertices is twice the number of base vertices:

```
vertices = new Vector3[2 * n_vertices_base];
```

In the loop over the number of base vertices

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

we compute the angle of the current base vertex

```
    float angle = 2f * Mathf.PI * i_vertex / n_vertices_base;
```

and use it to compute the vertex coordinates on a circle in the X-Z-plane:

```
    vertices[i_vertex] =
    new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle));
```

As just explained, we additionally define a separate tip vertex for every base vertex:

```
    vertices[i_vertex + n_vertices_base] =
  new Vector3(0, 1, 0);
}
```

Unity wants us to define the triangles by index triplets in one large integer vector:

```
triangles = new int[3 * n_vertices_base];
```

There are as many triangles as there are base vertices; however, we have to define the last triangle separately because as its last vertex we have to use the very first vertex of the very first triangle to smoothly close the surface of the cone. Therefore, we start a loop over one less than the number of base vertices:

```
for (
int i_triangle = 0;
i_triangle < n_vertices_base - 1;
i_triangle++)
{
```

In the loop, we define the three vertex indices of the current triangle. The first index is the index of the tip vertex:

```
    triangles[3 * i_triangle] = i_triangle + n_vertices_base;
```

The second and third vertex indices are the indices of the two base vertices:

```
    triangles[3 * i_triangle + 1] = i_triangle;
    triangles[3 * i_triangle + 2] = i_triangle + 1;
}
```

The last triangle closes the surface using the last tip vertex

```
triangles[3 * n_vertices_base - 3] = 2 * n_vertices_base - 1;
```

the last base vertex

```
triangles[3 * n_vertices_base - 2] = n_vertices_base - 1;
```

and the first base vertex:

```
triangles[3 * n_vertices_base - 1] = 0;
}
```

2.2.3 MakeDoublesided

By default, Unity renders a mesh as a one-sided surface. If we look at the cone from the top, everything seems alright; however, if we look at the "inside" of the cone from the bottom, there is no surface at all, the cone becomes invisible. The best way to avoid this problem is to duplicate the mesh and give all vertices of the "other side" opposite normal vectors.

We want the function

```
void MakeDoublesided()
{
```

to be general enough to use it for arbitrary meshes by calling it right after the `CreateShape` function.

Vertices We determine the number of vertices in the mesh

```
int n_vertices = vertices.Length;
```

and create a new vertex vector twice as long:

```
vertices_double_sided = new Vector3[2 * n_vertices];
```

Finally, we copy all original vertices into the new vector twice:

```
vertices.CopyTo(vertices_double_sided, 0);
vertices.CopyTo(vertices_double_sided, n_vertices);
```

Triangles In a similar step, we determine the number of triangles in the mesh

```
int n_triangles = triangles.Length;
```

and create a new triangle vector twice as long:

```
triangles_double_sided = new int[2 * n_triangles];
```

Next, we copy the original triangle vector into the new triangle vector:

```
triangles.CopyTo(triangles_double_sided, 0);
```

Finally, we copy the original triangle vector again, but in this case we invert the order[3] of the vertices for each triangle which automatically orients the normal vectors in the opposite direction. In a loop over every original triangle

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

we use the tip vertex as the new first vertex

```
    triangles_double_sided[i + n_triangles] =
    triangles[i] + n_vertices;
```

the original third vertex as the new second vertex

```
  triangles_double_sided[i + n_triangles + 1] =
    triangles[i + 2] + n_vertices;
```

[3]Think of this as cycling through the vertices in the opposite sequence: If the original triangle had an anti-clockwise vertex order, this triangle now has a clockwise vertex order.

and the original second vertex as the new third vertex

```
  triangles_double_sided[i + n_triangles + 2] =
    triangles[i + 1] + n_vertices;
}
}
```

2.2.4 UpdateMesh

Finally, we use the UpdateMesh function

```
void UpdateMesh()
{
```

to clear the mesh of all previous data

```
cone.Clear();
```

copy the new vertices and triangles into the mesh

```
cone.vertices = vertices_double_sided;
cone.triangles = triangles_double_sided;
```

and ask UNITY to recalculate the normals of all vertices:

```
cone.RecalculateNormals();
}
}
```

2.3 Plane

As stated in section 2.2, Unity offers a `Plane` primitive as one of its 3D game objects. Nevertheless, for reasons of flexibility, we write our own `Plane` class. As with the cone (section 2.2), we create a new empty `GameObject` in UNITY (figure 2.5), add a `Mesh Filter` and a `Mesh Renderer` as `Mesh` components, dress it with a transparent green material and add two `Scripts` by the name of `Plane` (section 2.3.1) to create the plane and `WASD` (section 2.3.2) to translate and rotate the plane.

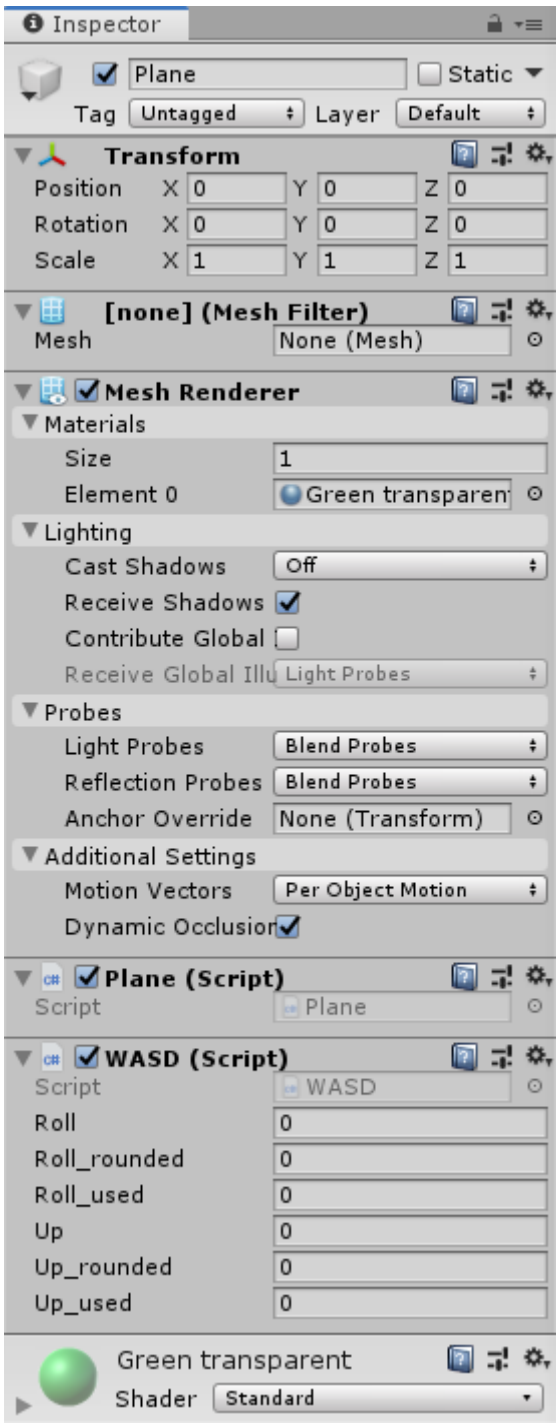


Figure 2.5: Plane

2.3.1 Plane class

The procedure in the Plane class

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

public class Plane : MonoBehaviour
{
```

is very similar to the creating of the cone in section 2.2.1. We declare the mesh, single- and double-sided vertex and triangle vectors

```
Mesh plane;

Vector3[] vertices;
int[] triangles;

Vector3[] vertices_double_sided;
int[] triangles_double_sided;
```

initialize the mesh

```
void Start()
{
plane = new Mesh();

GetComponent<MeshFilter>().mesh = plane;
```

and call the functions to create the shape, make it double-sided and refresh the mesh:

```
CreateShape();

MakeDoublesided();

UpdateMesh();
}
```

The function to create the plane

```
void CreateShape()
{
```

defines the length and width

```
float x_max = 2;
float z_max = 1;
```

the four vertices

```
vertices = new Vector3[]
{
    new Vector3(x_max, 0, z_max),
```

```
    new Vector3(-x_max, 0, z_max),
    new Vector3(-x_max, 0, -z_max),
    new Vector3(x_max, 0, -z_max)
};
```

and the two triangles[4] of the plane:

```
triangles = new int[]
{
    0, 1, 2,
    2, 3, 0
};
```

The function to make the plane double-sided is identical to section 2.2.3 and in the UpdateMesh we just replace `Cone` by `Plane`:

```
void UpdateMesh()
{
    plane.Clear();

    plane.vertices = vertices_double_sided;
    plane.triangles = triangles_double_sided;

    plane.RecalculateNormals();
}
}
```

2.3.2 WASD class

The WASD class

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

public class WASD : MonoBehaviour
{
```

translates and rotates the plane if the user presses one of the keys `W`, `A`, `S`, or `D`. We declare (and initialize) a few variables

```
public float roll = 0;
public float roll_rounded;
public float roll_used;

public float up = 0;
```

[4]By reusing the adjacent vertices we ensure the smoothness between both triangles.

```
public float up_rounded = 0;
public float up_used = 0;
```

and initially access the Euler angles (i. e. the orientation of the plane with respect to the global coordinate system):

```
void Start()
{
Vector3 angles = transform.eulerAngles;
```

Since we only want to alter the roll angle of the plane, we extract it as the rotation about the Z-axis:

```
roll = angles.z;
```

Additionally, we access the position of the plane

```
Vector3 position = transform.position;
```

and extract its Y-coordinate (`up`):

```
up = position.y;
}
```

In ever simulation step

```
void Update()
{
```

we call Unity's general `Input.GetAxis("Horizontal")` function[5] to figure out if the user pressed the `A` or `D` key. Utilizing the simulation step size `Time.deltaTime`, we can in- or decrease the plane's roll angle up to 20° per second:

```
roll += Input.GetAxis("Horizontal") * 20 * Time.deltaTime;
```

We limit the roll angle between 0° and 90°

```
roll = Mathf.Clamp(roll, 0, 90f);
```

and store its rounded value in a separate variable:

```
roll_rounded = Mathf.Round(roll);
```

We want roll angle to snap to the specific values 0°, 45°, and 90° if its floating point value gets close to these values. For this purpose we buffer the current roll angle in a separate variable

```
roll_used = roll;
```

[5] Unity's `Input.GetAxis` function is frame-rate-independent, auto-accelerates, works with keyboard, joystick, and controllers and therefore offers a smooth way to transform objects.

and analyze if the rounded roll angle equals one of the specific values. If this is the case, we set the buffered value to this specific value:

```
if (roll_rounded == 45)
{
    roll_used = 45;
}
else if (roll_rounded == 0)
{
    roll_used = 0;
}
else if (roll_rounded == 90)
{
    roll_used = 90;
}
```

We can now use the buffered value to actually perform a snapping rotation of the plane, while maintaining a smooth internal variation of the commanded roll angle:

```
transform.eulerAngles = new Vector3(0, 0, roll_used);
```

We use the same technique (frame rate independent motion, limitation, and snapping[6] to specific values) for the translation of the plane if the user presses the `W` or `S` key:

```
up += Input.GetAxis("Vertical") * 50 * Time.deltaTime;

up = Mathf.Clamp(up, 0, 50);

up_rounded = Mathf.Round(up);

up_used = up;

if (up_rounded == 0)
{
    up_used = 0;
}

transform.position = new Vector3(0, up_used / 100, 0);
}
}
```

2.4 Camera

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

[6] We internally upscale the translation value by factor of 100 to allow us to use the same simple snapping algorithm.

2.4.1 Mouse Orbit

For this purpose, we attach the `MouseOrbit` class

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

public class MouseOrbit : MonoBehaviour
{
```

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

```
float distance = 4.0f;
```

the constant factors translating the mouse speed to the orbit angles rate

```
readonly float xSpeed = 5f;
readonly float ySpeed = 5f;
```

and the constant distance limits:

```
readonly float distanceMin = 2f;
readonly float distanceMax = 10f;
```

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

```
float azimuth = 30;
float elevation = 30;
```

In every simulation step

```
void LateUpdate()
{
```

we check if the user pressed the left mouse button

```
if (Input.GetMouseButton(0))
{
```

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

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

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 potential 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"),
distanceMin,
distanceMax);
```

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

```
Vector3 negDistance = 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 [5] with the Euler angles in the transformation matrix:

```
Vector3 position = rotation * negDistance;
```

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

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

2.5 Canvas

We want to inform the user about the different ways to interact with the website and about the current height and angle of the plane and the type of the current conic section (figure 2.6).

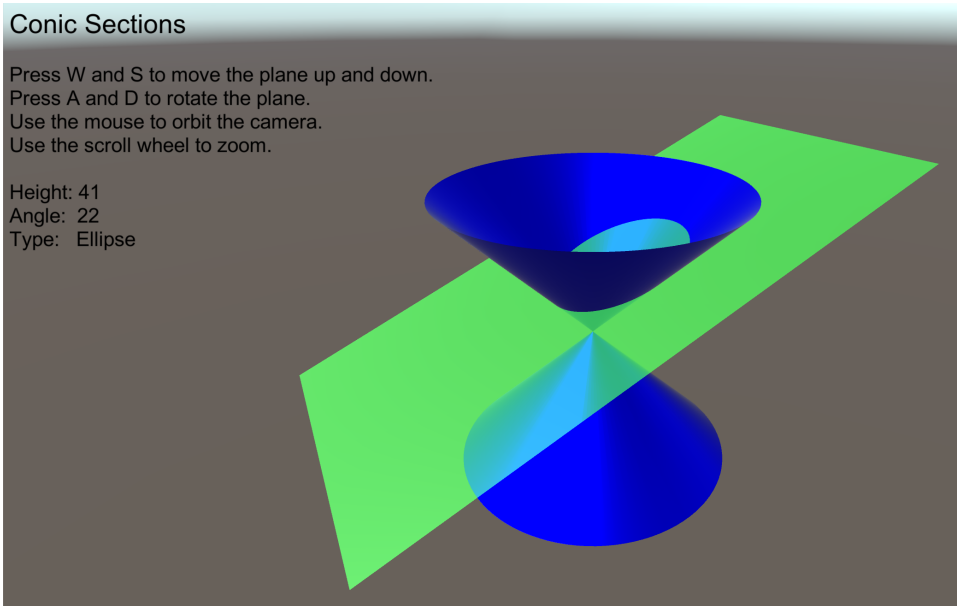


Figure 2.6: Text on Canvas

For this purpose 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 2.7).

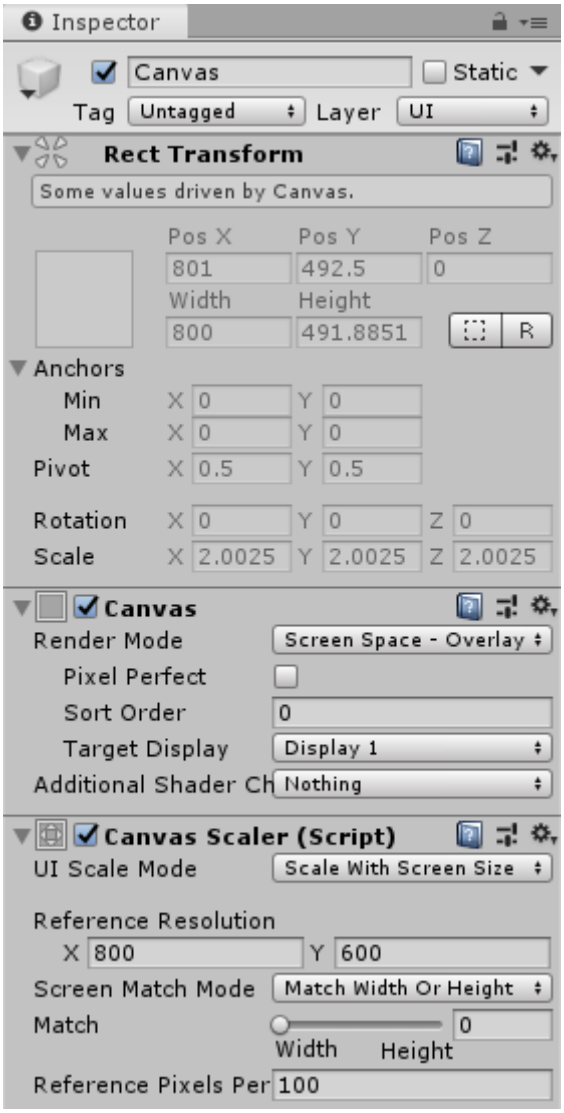


Figure 2.7: Canvas

We tell the `Canvas Scaler` to `Scale With Screen Size` and chose a `Reference Resolution` of 800×600 (figure 2.7) which seems to produce acceptable results in most (desktop) browsers.

2.5.1 Text Display

The `Text GameObject` is a child of the `Canvas`.

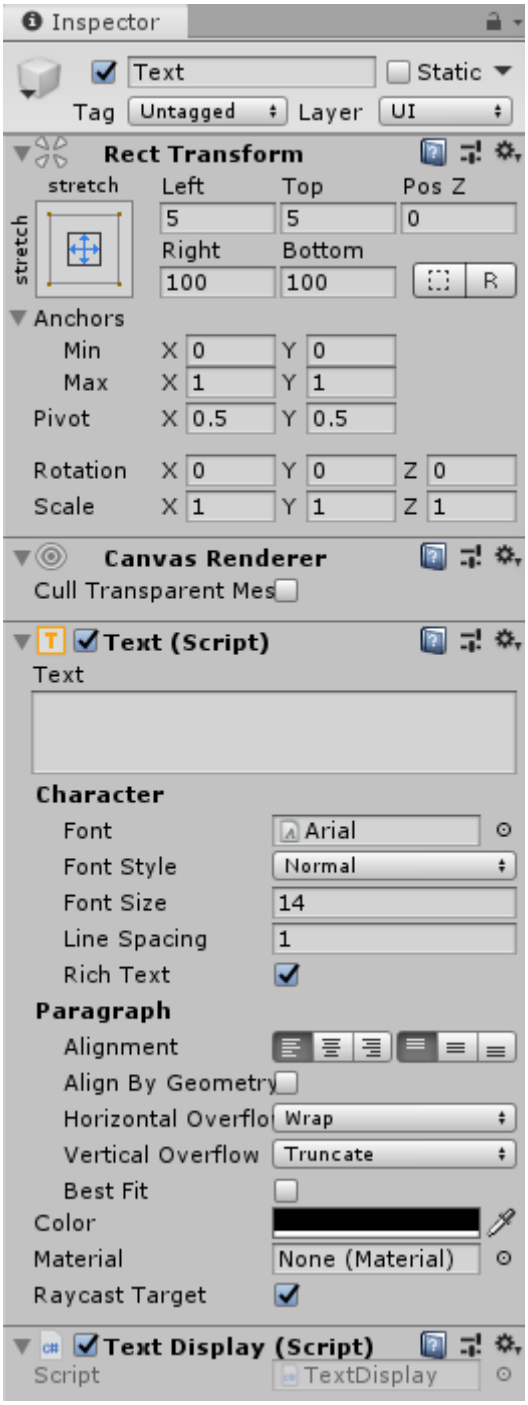


Figure 2.8: Text

We pad the `Text` with 5 pixels on the top and the left (figure 2.8 and figure 2.6) and stretch it onto the `Canvas`. We attach a `TextDisplay` script to the `Text` (figure 2.8)

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.UI;

public class TextDisplay : MonoBehaviour
{
```

in which we declare the text itself

```
Text text;
```

and the plane:

```
GameObject plane;
```

During the initialization

```
void Start()
{
```

we access the text

```
text = GetComponent<Text>();
```

and the plane:

```
plane = GameObject.Find("Plane");
}
```

In every simulation step

```
void Update()
{
```

we read the rounded roll angle and the rounded height value of the plane and convert it to an integer:

```
int roll = (int)(plane.GetComponent<WASD>().roll_rounded);

int up = (int)(plane.GetComponent<WASD>().up_rounded);
```

Additionally, we declare a string variable we will use to display the type of the conic:

```
string type;
```

We now distinguish the different cases depending on the height and the roll angle of the plane. If the height is zero

```
if (up == 0)
{
```

we can have a point, a straight line or two straight lines, depending on the roll angle of the plane:

```
    if (roll < 45)
    {
        type = "Point";
    }
    else if (roll == 45)
    {
        type = "One straight line";
    }
    else
    {
        type = "Two straight lines";
    }
}
```

If the height is not zero

```
else
{
```

the roll angle decides whether we get a circle, an ellipse, a parabola, a hyperbola, or two straight lines:

```
    if (roll == 0)
    {
        type = "Circle";
    }
    else if (roll < 45)
    {
        type = "Ellipse";
    }
    else if (roll == 45)
    {
        type = "Parabola";
    }
    else if (roll < 90)
    {
        type = "Hyperbola";
    }
    else
    {
        type = "Two straight lines";
    }
}
```

Finally, all we have to do is to generate the text string using the current height and roll angle of the plane and the type of the conic and copy the string into the corresponding property of the text:

```
text.text =
"<size=18>Conic Sections</size>" +
"\n" +
"\n" +
"Press W and S to move the plane up and down." +
"\n" +
"Press A and D to rotate the plane." +
"\n" +
"Use the mouse to orbit the camera." +
"\n" +
"Use the scroll wheel to zoom." +
"\n" +
"\n" +
"Height: " + up.ToString() +
"\n" +
"Angle:  " + roll.ToString() +
"\n" +
"Type:   " + type;
}
}
```

Bibliography

[1] J. J. Buchholz. (2019) Conic Sections. [Online]. Available: https://m-server.fk5.hs-bremen.de/conic/index.html

[2] Unity. (2019). [Online]. Available: https://unity.com

[3] Wikipedia. (2019) Z-fighting. [Online]. Available: https://en.wikipedia.org/wiki/Z-fighting

[4] Sketchup. (2019). [Online]. Available: https://www.sketchup.com/

[5] Wikipedia. (2019) Gimbal lock. [Online]. Available: https://en.wikipedia.org/wiki/Gimbal_lock