PROJECT

"CONTEMPORARY PROGRAMMING LANGUAGES, ENVIRONMENTS AND TECHNOLOGIES AND THEIR APPLICATION IN BUILDING UP SOFTWARE DEVELOPERS"

CONTRACT **№ DFNI-I01/12**

MECHO

~ INTERACTIVE VIRTUAL MECHANICS ~

DESIGN AND IMPLEMENTATION:

PAVEL BOYTCHEV

FACULTY OF MATHEMATICS AND INFORMATICS
SOFIA UNIVERSITY

2013

CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Mecho (MECHanical Objects) is a library for building and animating mechanical devices and linkages. It provides a set of virtual mechanical parts, called *mecholets*. They are positioned, oriented and animated programmatically. The library provides tools for building virtual devices – complex mechanisms comprising of several joined mecholets.

The library is built within the scope of project "*Contemporary programming languages, environments and technologies and their application in building up software developers*", contract № DFNI-I01/12. The design of the library is based on previous project "*Educational Logo Interface for Creative Activities*".

1. SOURCE FILES

Mecho is distributed as source files grouped in several categories:

- **Library files**: These are the main files that control the basic functionality of Mechio. The files are `Mecho.cpp` and `Mecho.h` and are in folder `…/Sources`.
- **Mecholet files**: These files define the mecholets. For each mecholet there is a pair of `.cpp` and `.h` files. For example, the `beam` mecholet is defined in `Beam.cpp` and `Beam.h`. The mecholet files are in folder `…/Sources`.
- **Device files**: These files define the devices. Devices that are closely related (in terms of OOP) are usually defined in a single file. For example, the file `hypotrochoid.cpp` (and its companion `.h`) define the `Hypotrohoid` device, as well as other devices, which are based on it, like the `Hypocycloid` and the `TusiCouple` devices. The device files are in folder `…/Devices`.
- **Demo files**: These files define demonstrational and exemplary files. They are referenced in this document and provided both as source code and as binary executable file for Windows. The demo files are in folder `…/Demos`.

2. PROGRAMMING WITH MECHO

a) The minimal program

Mecho is designed in a way to simplify the task of building and animating virtual mechanisms. Many of the functions and the classes use default values for their parameters. The minimal Mecho program is this:

```
#include "Mecho.h"
int main()
{
    initMecho();
    while(runningMecho()) { }
    finitMecho();
    return 0;
}
```

It opens a graphical window and shows an empty virtual scene – a marble ground without any mecholet or devices (see Figure 1 and `Minimal.cpp`).

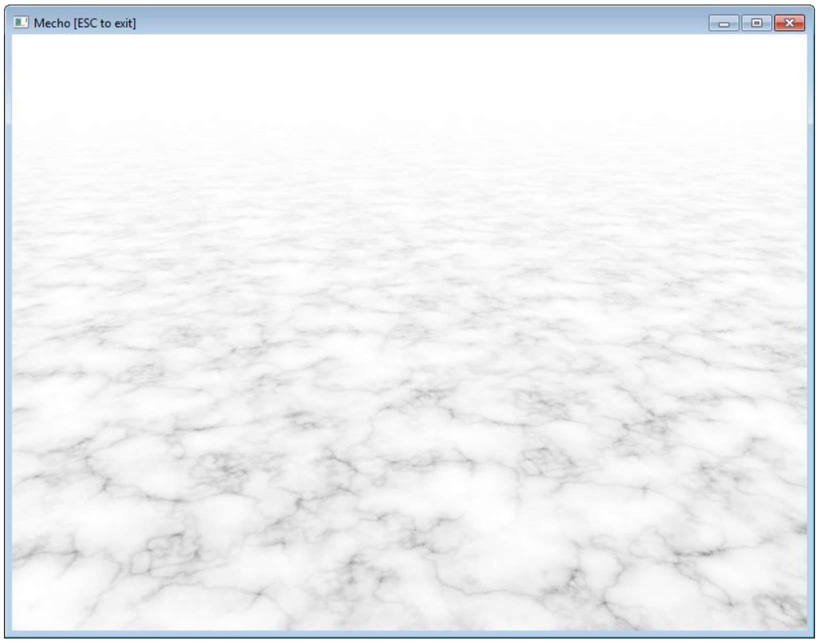


Figure 1. The minimal Mecho program.

b) Initialization

Every Mecho program includes `Mecho.h`. Then, it initializes the screen, declares and initializes the virtual mechanism and enters in a loop. The body of the loop animates the mechanism. The execution ends with the finalization of the program.

The typical Mecho program demonstrating a virtual mechanism is this:

```
#include "Mecho.h"
int main()
{
  initMecho();
  mecholet definitions
  while( runningMecho() )
    {
      mecholet motion
    }
  finitMecho();
  return 0;
}
```

Function `initMecho` builds an empty 3D virtual world, where the mechanisms exist. The complete function specification is this:

```
void initMecho( caption       = "Mecho [ESC to exit]",
                drawingStyle  = BEAUTIFUL,
                lightingStyle = DAY,
                groundStyle   = MARBLE,
                materialStyle = WOOD );
```

Parameter `caption` is the title of the graphical window.

Parameter `drawingStyle` is the quality of rendering. Possible values are `BEAUTIFUL` and `CARTOON` – see

Figure 2. The style `BEAUTIFUL` generates nicer scenery, but at lower speed; `CARTOON` generates flat-shaded, that is drawn faster.

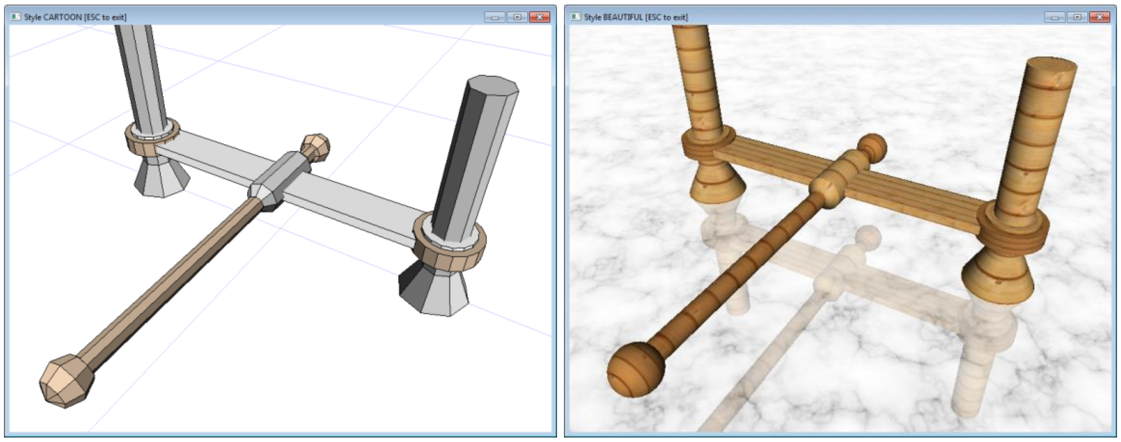


Figure 2. Drawing styles `CARTOON` *and* `BEAUTIFUL`

Parameter `dayTime` sets the lighting conditions to daytime or nighttime. Possible values are `DAY` and `NIGHT` – see Figure 3.

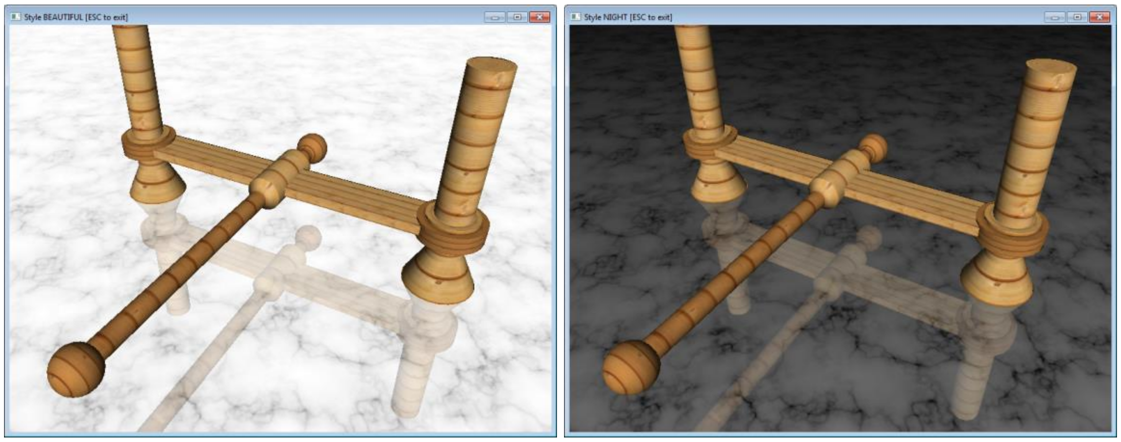


Figure 3. Lighting styles `DAY` *and* `NIGHT`

Parameter `groundStyle` is the material style of the ground. Possible values are `MARBLE`, `TILES`, `ASPHALT`, `ROCK` and `INDUSTRIAL` – see Figure 4. For `MARBLE` or `TILES` the ground is reflective and rendering is slower. For `ASPHALT`, `ROCK` and `INDUSTRIAL` the ground is non-reflective.

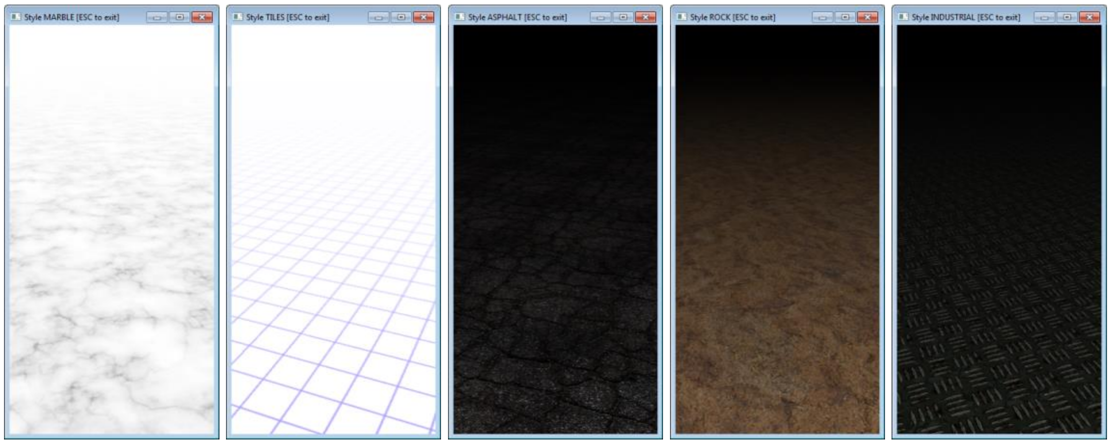

Style MARBLE [ESC to exit]
Style TILES [ESC to exit]
Style ASPHALT [ESC to exit]
Style ROCK [ESC to exit]
Style INDUSTRIAL [ESC to exit]

Figure 4. Ground styles `MARBLE`, `TILES`, `ALSPHALT`, `ROCK` *and* `INDUSTRIAL`

Parameter `materialStyle` is the style of the mecholets' material. Possible values are `METAL`, `METRIC`, `SCRATCH`, `PAPER` and `WOOD` (see Figure 5 and the corresponding `Style…` demo programs). The metric material style is used mainly for debugging texture calibrating purposes.

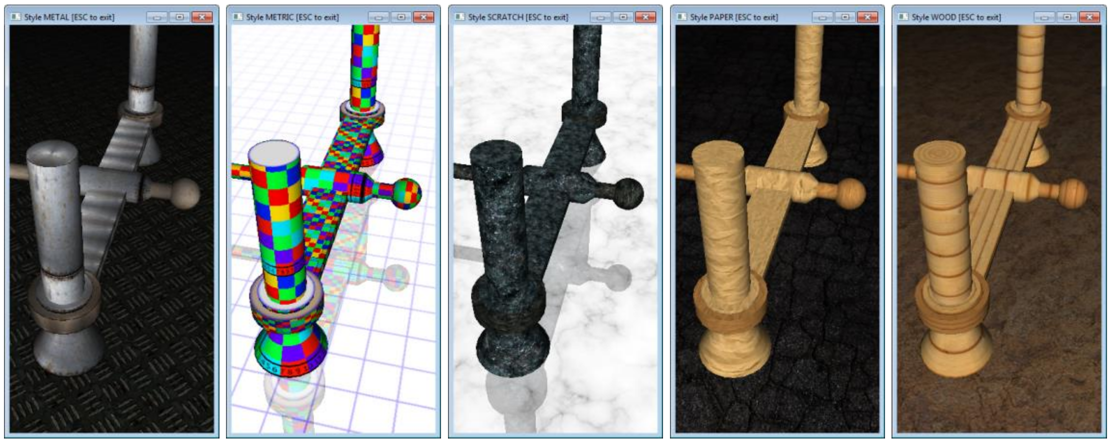

Style METAL [ESC to exit]
Style METRIC [ESC to exit]
Style SCRATCH [ESC to exit]
Style PAPER [ESC to exit]
Style WOOD [ESC to exit]

Figure 5. Material styles `METAL`, `METRIC`, `SCRATCH`, `PAPER` *and* `WOOD`

c) Per-frame management

Function `runningMecho` handles all per-frame activities including:

- Rendering mecholets
- Managing time synchronization
- Capturing mouse and keyboard events

The specification of the function is this:

```
bool runningMecho();
```

The function is called at the beginning of every frame generation. The result is `true` if the model is still active. If the user presses `ESC` key or closes the program window, the function immediately returns `false`.

d) Finalization

Function `finitMecho` is called once just before program exit. It finalizes the program and releases used resources. The specification of the function is this:

```
void finitMecho();
```

II. MECHOLETS

1. MECHOLET PROPERTIES

All mecholets share the same set of properties for managing their location and orientation in the space.

a) Elements

Every mecholet has a set of intrinsic invisible elements – several control points and axes. (see **Error! Reference source not found.**). A mecholet has *a central point*, which may or may not coincide with its geometrical center. This point is a reference point to object's motion. Changes to the central point translate the mecholet in the 3D space.

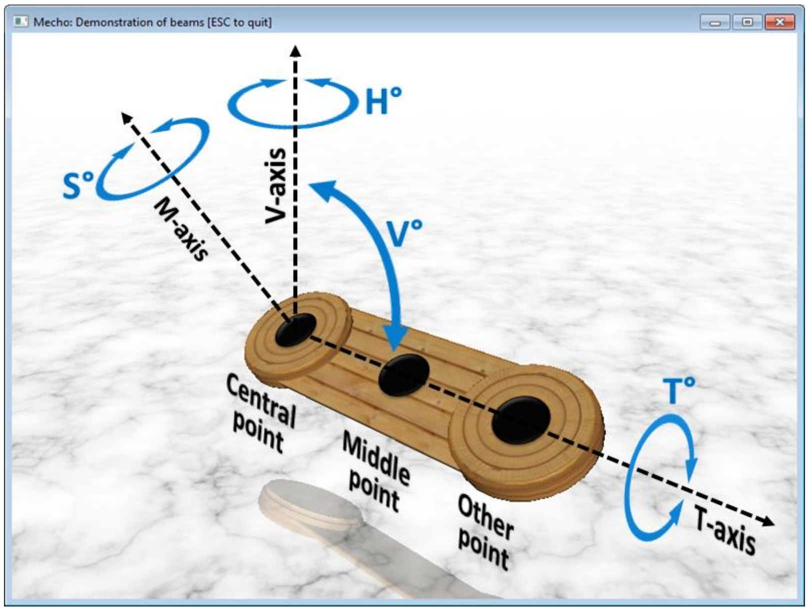


Figure 6. Elements of a mecholet for its position and orientation

There are three axes passing through the central point (see Figure 6).

- The *vertical axis* (V-axis) always points towards Z^+ direction of the virtual world independent on the location and the orientation of a mecholet.
- The *main axis* (M-axis) points to the local object-related up direction. This axis is "attached" firmly to the object, so when the objects changes its orientation, the axis changes too.
- The *twist axis* (T-axis) goes through the body of the mecholet. It is used mainly to get the coordinates of points along the object. For some mecholets the twist and the main axes are the same.

b) Location

To position a mecholet in 3D space is defined by its central point. Its coordinates are set and get with these methods:

```
vect getCenter();
void setCenter(center);
```

Individual axial coordinates of the central point can be set with these methods:

```
void setX(x);
void setY(y);
void setZ(z);
```

To get a point along the twist axis use these methods:

```
vect atPoint(pos);
vect middlePoint();
vect otherPoint();
```

where `atPoint` returns the coordinates of a point at position `pos`. Thus:

- `atPoint(0)` returns the central point, i.e. `getCenter()`
- `atPoint(0.5)` returns the middle point, i.e. `middlePoint()`
- `atPoint(1)` returns the other point, i.e. `otherPoint()`

It is possible to shift only the image of a mecholet along the main axis, without changing the central point of the mecholet. Results of `atPoint` will always be calculated as if the offset is 0. To set the offset of this shift use the method:

```
void setOffset(offset);
```

Once a point of the mecholet is found, it could be further modified by these methods:

```
vect setZ(z);
vect moveZ(z);
```

These methods modify the Z coordinate and are methods of 3D coordinates (class `vect`), not methods of mecholet instances.

c) Orientation

A mecholet has four angles that control its orientation: `H`, `V`, `T` and `S` – see Figure 6. The angles are measured in degrees.

- Spin angle `S` is the rotation of the mecholet around its main axis.
- Vertical angle `V` is the angle between the main and the vertical axes.
- Twist angle `T` is the rotation of the mecholet around its twist axis. Twisting changes the main axis direction.
- Horizontal angle `H` is the horizontal rotation of the whole mecholet together with its main and twist axis around the vertical axis.

The orientation can be set with these functions:

```
void setH(angle);
void setV(angle);
void setT(angle);
void setS(angle);
```

The orientation can be retrieved with these functions:

```
double getH();
double getV();
double getT();
double getS();
```

2. PRIMITIVE MECHOLETS

This section describes the mecholets defined in the Mecho library.

a) Invisible

An invisible mecholet is an abstract object that has no graphical representation. It is used solely for locating specific points. An invisible mecholet has all properties related to position and orientation. Additionally, it has `size`, which is used to determine the result of `otherPoint()`. The constructor is:

```
Invisible(size);
```

Invisibles are used to represent invisible elements in a construction. Figure 7 shows two pencils – one attached to a rotating beam and another one attached to a rotating invisible mecholet.

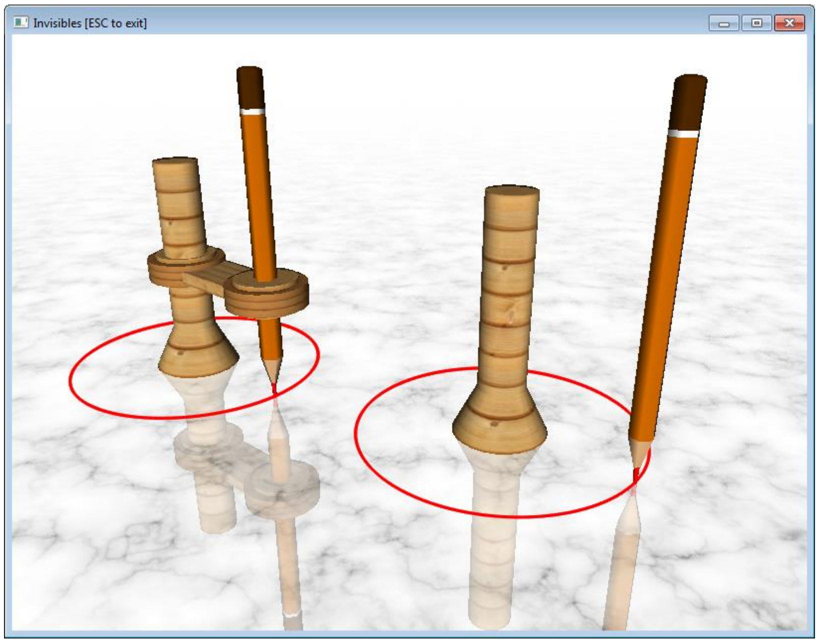


Figure 7. The right pencil is attached to a rotating invisible mecholet

An invisible mecholet is visualized by temporarily changing its type form `Invisible` to `Rail` or another suitable mecholet.

Any visible mecholet (i.e. not `Invisible`) can be made invisible or visible by using the function `setVisible`:

```
void setVisible(visibility);
```

If `visibility` is `true`, the mecholet is rendered in the virtual world, otherwise it is ignored.

b) Pencil

A pencil mecholet draws lines and curves. The properties that define its dimensions are shown in Figure 8. If `topSize` is less than 0.2, the top section of the pencil is not drawn.

The central point of a pencil is at its tip. The main and the twist axes coincide and go upwards. The other point is at the top of the pencil.

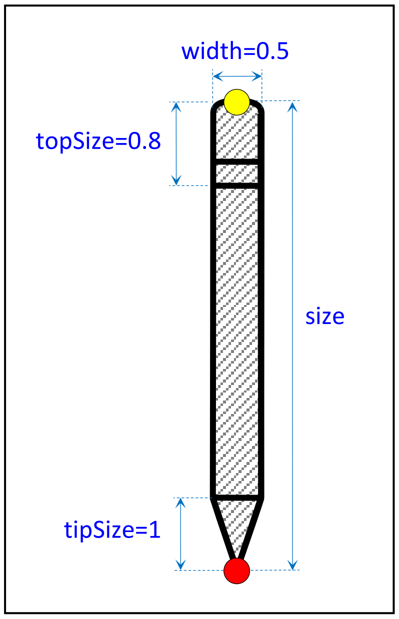


Figure 8. Pencil dimensions

The constructors of a pencil are:

```
Pencil(size);
Pencil(size, width);
Pencil(size, width, tipSize);
Pencil(size, width, tipSize, topSize);
```

Figure 8 (left) show three variants of pencils. The middle one is the default pencil.

Apart from the standard mecholet properties, a pencil leaves colour traces. The default colour is red, but it is change with this function:

```
void setColor(colour);
```

The value of parameter `colour` is one of these predefined constants: `RED`, `BLACK`, `WHITE`, `RED`, `BLUE`, `GREEN` or `YELLOW`. The current colour of a pencil is indicated as the colour of its graphite. The pencils in Figure 8 (left) have blue, red and black colours.

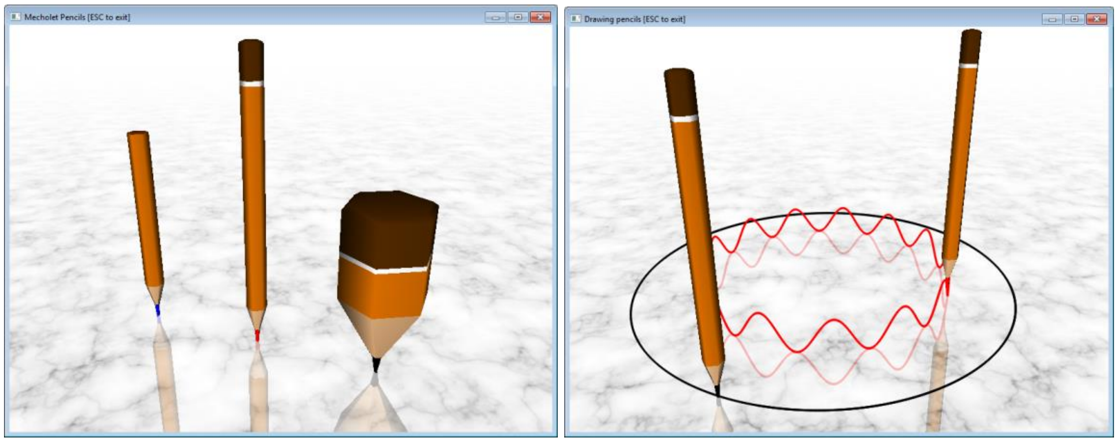


Figure 9. Various shapes of pencil (left) and their traces (right)

A pencil has two drawing states – up and down. When a pencil is up it does not draw traces while it is moving. When a pencil is down it draws traces. By default drawing state is down and drawing is turned off. To turn it on use these functions:

```
void penDown();
void penDown(duration);
```

Parameter `duration` defines how long (in terms of seconds) the pencil will be down. When the time elapses, the pencil will automatically revert to its up state. If `penDown` is used without parameter, it turns drawing on for 10^{38} seconds, which is practically forever. To explicitly and immediately turn drawing off, use this function:

```
void penUp();
```

c) Pillar

A pillar mecholet supports other mecholets. Although it is supposed to be static, it can be animated like any other mecholet. The properties that define pillar's dimensions and their default values are shown in Figure 10. The central point of a pillar is at its base.

The main and the twist axes coincide and go upwards. The other point is at the top of the pillar.

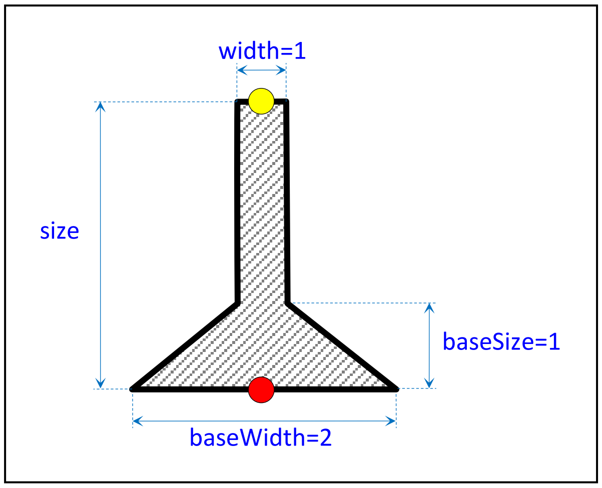


Figure 10. Pillar dimensions

The constructors of a pillar are:

```
Pillar(size);
Pillar(size, width);
Pillar(size, width, baseSize);
Pillar(size, width, baseSize, baseWidth);
```

Figure 11 shows several variations of pillars. The pillar at the far back has default sizes.

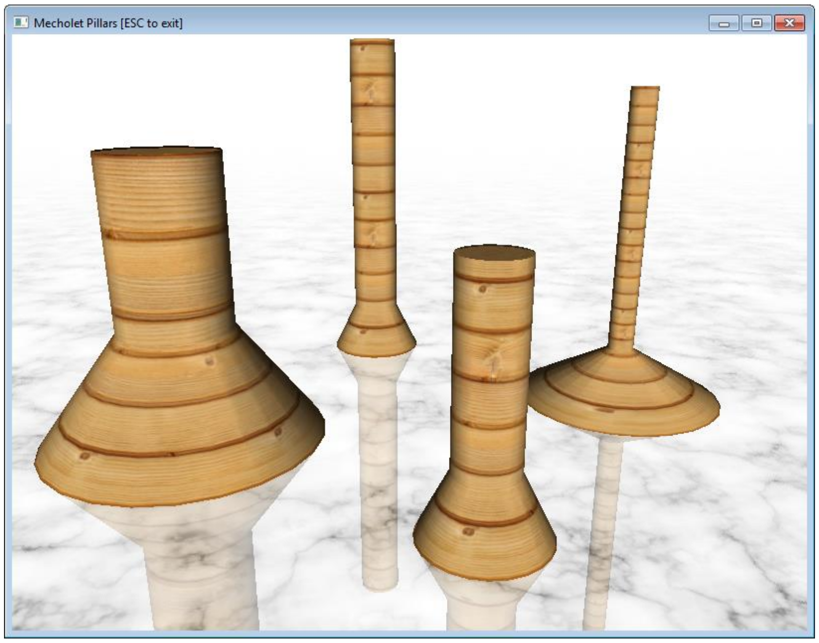


Figure 11. Various shapes of pillars

d) Rail

A rail mecholet controls and restricts the motion of other mecholets. The properties that define rail's dimensions and their default values are shown in Figure 12. If `baseSize` or `topSize` are equal to `width`, then the corresponding end of the rail is rounded. If they are smaller than `width`, then the rail terminates sharply. If they are larger, then the rail terminates with spheres.

The central point of a rail is at its base. The main and the twist axes coincide and go upwards. The other point is at the top of the pillar.

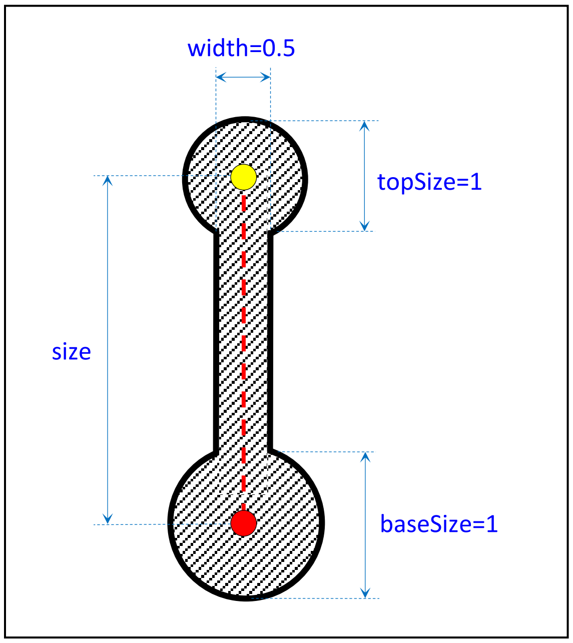


Figure 12. Rail dimensions

The constructors rail are:

```
Rail(size);
Rail(size, width);
Rail(size, width, baseSize);
Rail(size, width, baseSize, topSize);
```

Figure 13 shows several variations of rails.

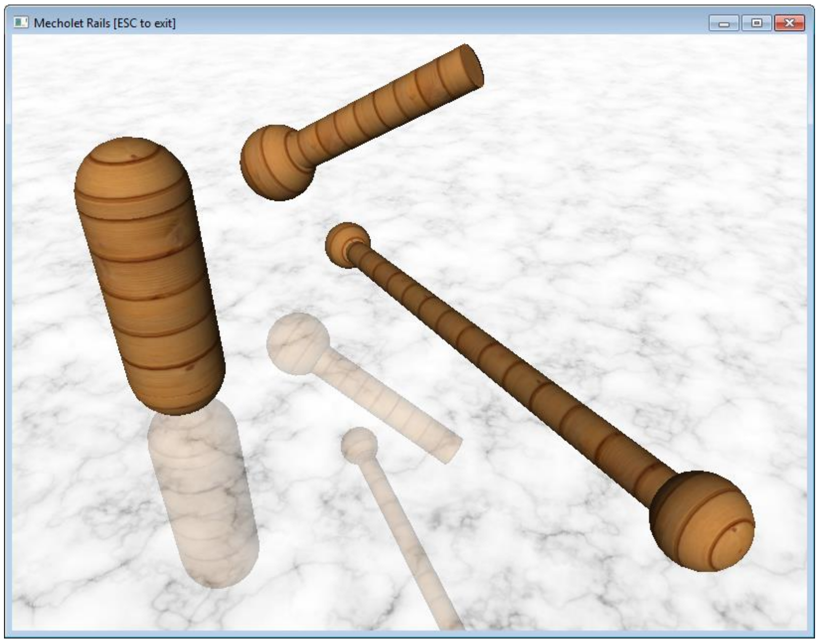


Figure 13. Various shapes of rails

e) Tube

A tube mecholet is a holder for cylindrical objects like pencils and rails. A tube rotates around the object that it holds. The object in a tube slides forward and backward through it. Also, a tube is used to connect two objects. The dimensions of a tube are shown in Figure 14.

The central point of a tube is at its middle. The main and the twist axes coincide and go upwards. The other point is at the top of the pillar.

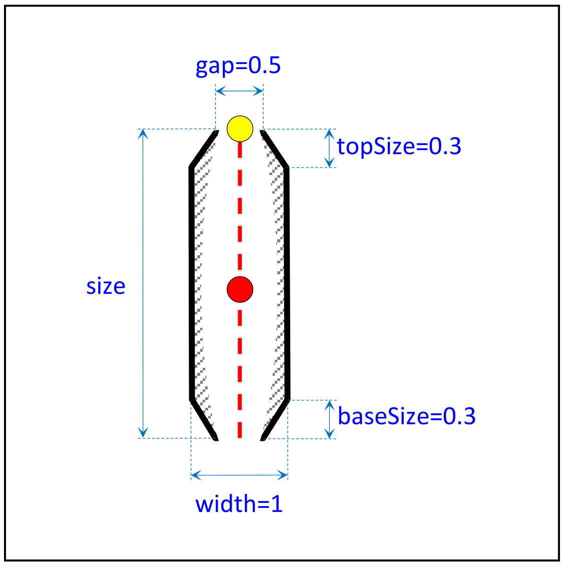


Figure 14. Tube dimensions

The constructors of a tube are:

```
Tube(size);
Tube(size, width);
Tube(size, width, baseSize);
Tube(size, width, baseSize, topSize);
Tube(size, width, baseSize, topSize, gap);
```

If `gap` of a tube is bigger than its `width`, then the ends of the tube are bent outwards and can be used as fillets – see the left tube in Figure 15.

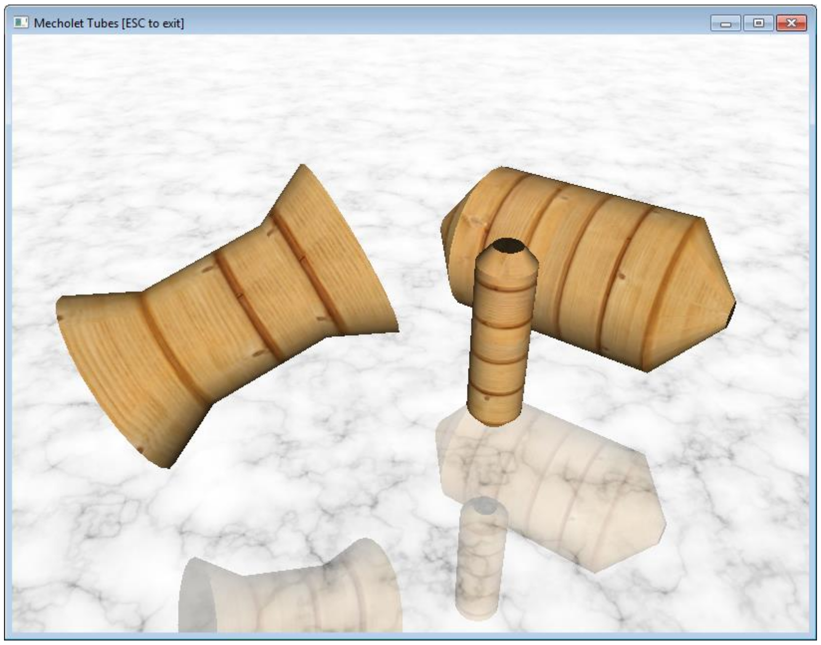


Figure 15. Various shapes of tubes

f) Beam

A beam mecholet represents a solid element connected to other elements. A beam is symmetrical, with disks at both ends. The disks model rotational joints.

The central point of a beam is at the center of its base disk. The main axis is perpendicular to the base disk, while the twist axis goes towards the other disk. The other point is at the center of the other disk.

The default size of a beam is shown in Figure 16. There are extrusions attached to the disks. Their size is set automatically and is larger than `width` and smaller than `diskSize`.

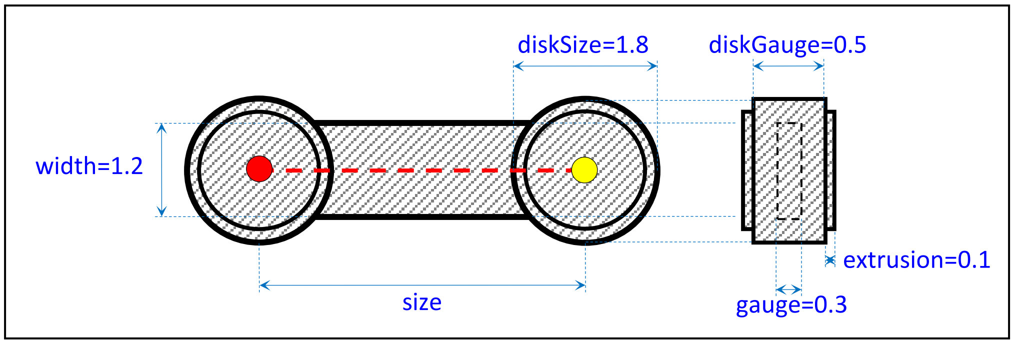


Figure 16. Beam dimensions

The constructors of a beam are:

```
Beam(size);
Beam(size, width);
Beam(size, width, gauge, diskSize);
Beam(size, width, gauge, diskSize, diskGauge);
Beam(size, width, gauge, diskSize, diskGauge, extrusion);
```

The size of the disk cannot be less than or equal to the width of a beam. The leftmost beam in Figure 17 uses the default sizes.

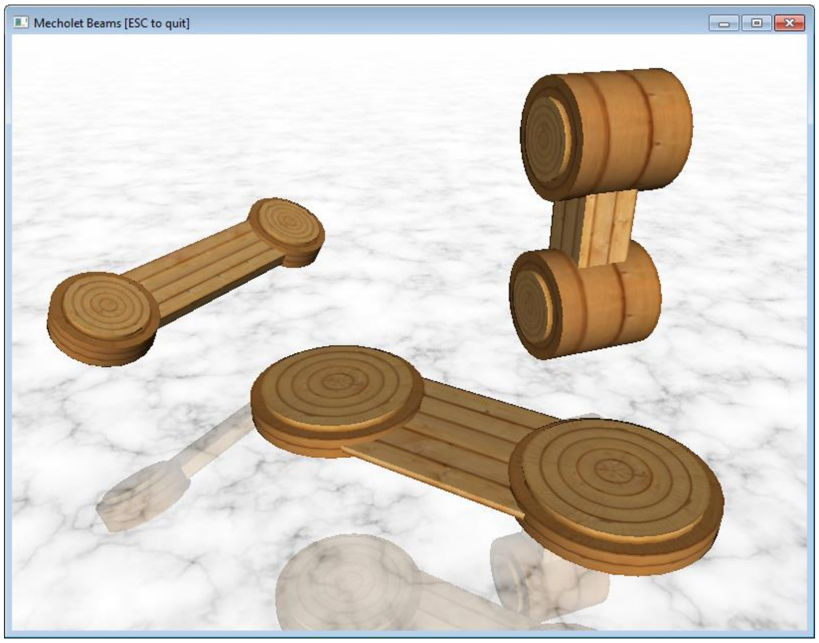


Figure 17. Various shapes of beams

A beam main and twist axes are different. As a result, rotating around the twist axes flips a beam from horizontal into vertical and vice versa. Figure 18 shows the effect of flipping. One of the beams rotates along a rail, the other – around a rail.

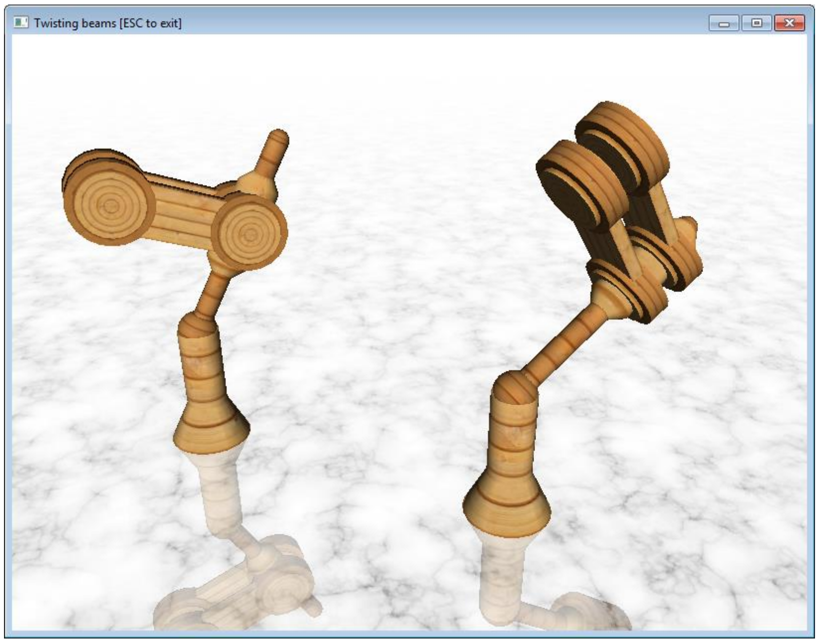


Figure 18. Beams with two different orientations

g) Ring

A ring mecholet is a circular ring with internal gear teeth. The typical use of rings is to have a gear rolling inside them. Ring teeth are dimensionless and are located on the internal surface of a ring.

The central point of a ring is at its center. The main axis is perpendicular to the plane of the ring disk, while the twist axis goes towards a point on the ring. The other point is that point.

Figure 19 shows the dimensions of a ring. The outer size is automatically set to `size+2` if `outerSize` is not provided at the time of ring construction.

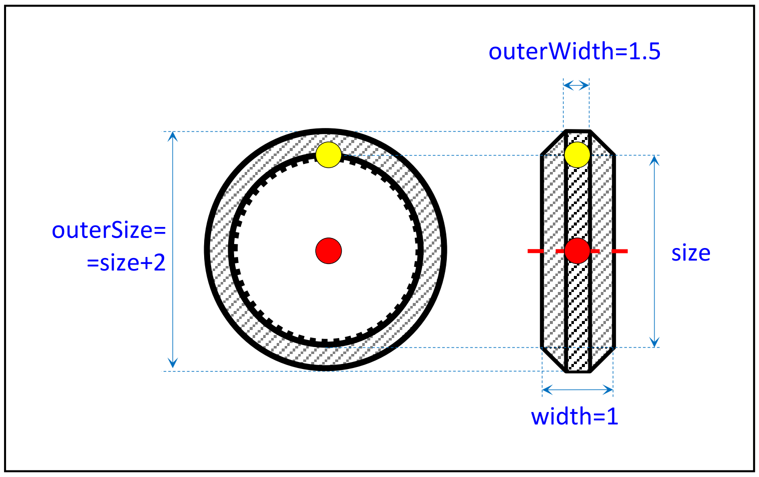


Figure 19. Ring dimensions

The constructors of a ring are:

```
Ring(size);
Ring(size, width);
Ring(size, width, outerSize);
Ring(size, width, outerSize, outerWidth);
```

A group of rings is shown in Figure 20:

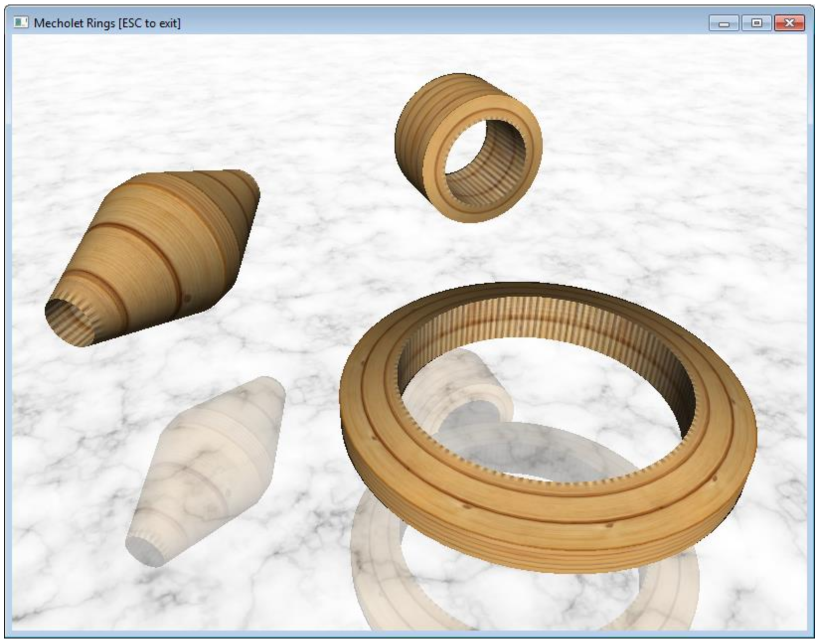


Figure 20. Various shapes of rings

h) Gear

A gear mecholet is a disk with gear teeth. The typical use of gears is to roll inside rings or to roll against other gears. Gear teeth are dimensionless and are located on the outer surface of the gear.

The central point of a gear is at its center. The main axis is perpendicular to the plane of the gear disk, while the twist axis goes towards a point on the gear. The other point is that point.

Figure 21 shows the dimensions of a gear. The inner size is automatically set to `size-2` (or 0) if `innerSize` is not provided at the time of gear construction.

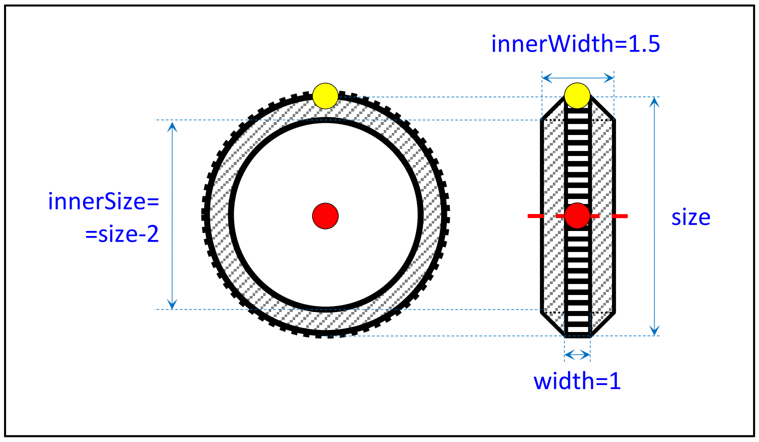


Figure 21. Gear dimensions

The constructors of a gear are:

```
Gear(size);
Gear(size, width);
Gear(size, width, innerSize);
Gear(size, width, innerSize, innerWidth);
```

Figure 22 shows several customized gears.

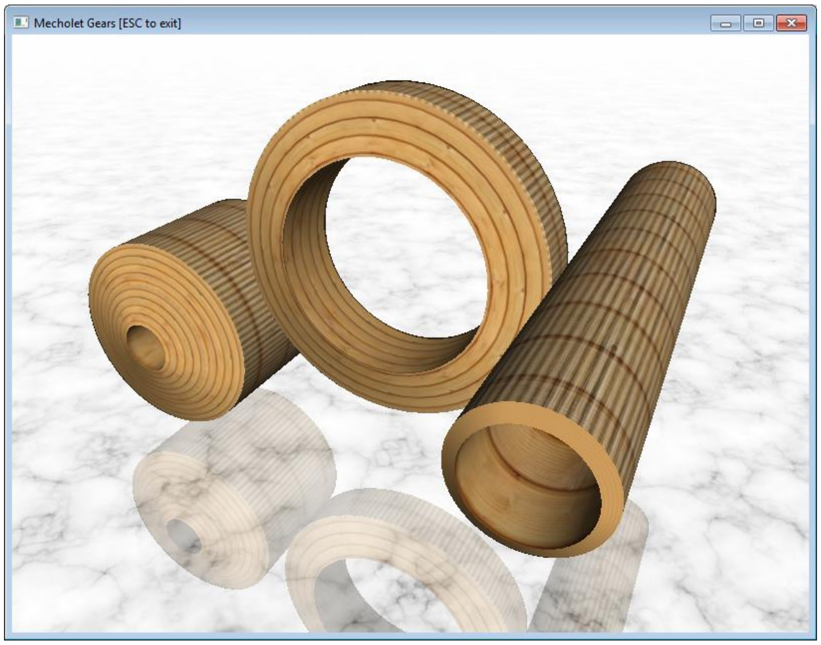


Figure 22. Various shapes of gears

i) Disk

A disk mecholet represents … a disk. It is used for joints with rotation.

The central point of a disk is at its center. The main axis is perpendicular to the plane of the disk, while the twist axis goes towards a peripheral point on the disk. The other point is that point.

Figure 23 shows the dimensions of a disk.

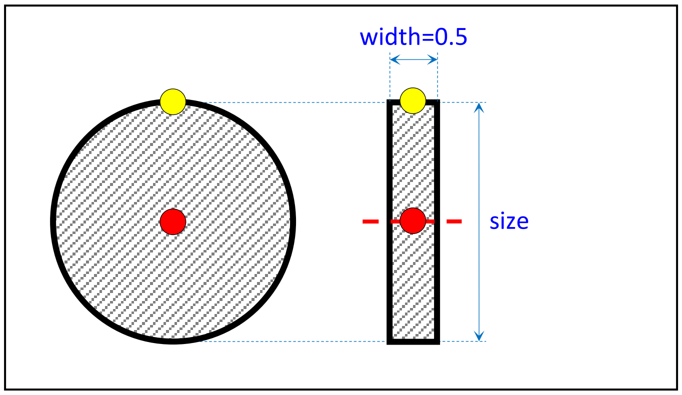


Figure 23. Disk dimensions

The constructors of a disk are:

```
Disk(size);
Disk(size, width);
```

Figure 24 shows three customized disks:

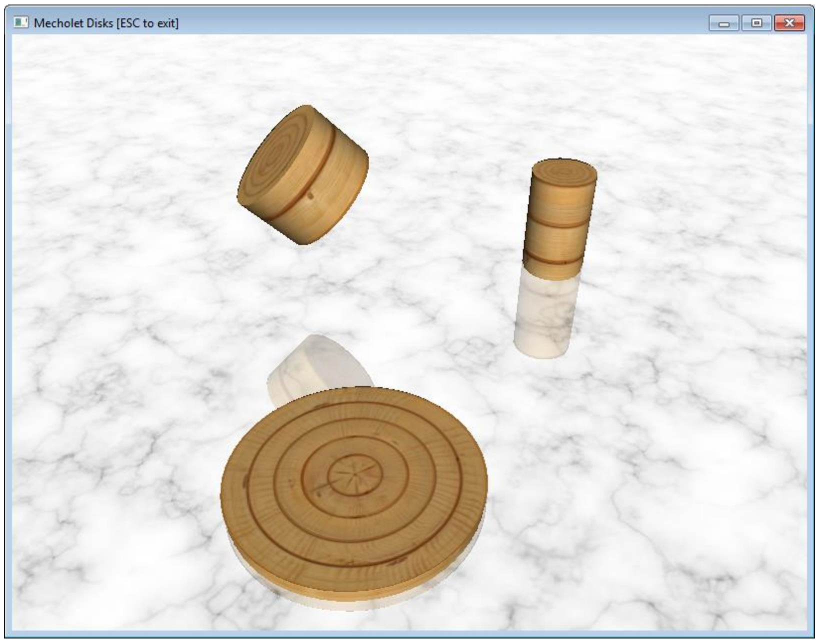


Figure 24. Various shapes of disks

III. DEVICES

A device is a virtual mechanism built of mecholets. Often, a pencil is attached to a device to trace a path.

1. CUSTOM DEVICES

a) Defining custom devices

A device is defined as a C++ class based on class `Mechodev`. It determines the used mecholets, their properties and their motion in respect to time.

A general definition of a non-drawing device is this:

```
class MyDevice : public MechoDev
{
    ‹…list of mecholets…›
public:
    MyDevice(‹…device parameters…›);
    virtual void setTime(double t);
};

MyDevice::MyDevice(‹…device parameters…›):
    ‹…mecholets constructors…›
{
    localize(‹…number of mecholets…›);
    ‹…initial mecholets properties…›
}

void MyDevice::setTime(double t)
{
    ‹…mecholets motion…›
}
```

The method `localize(n)` is used to move the latest `n` created mecholets in the current device. The virtual method `setTime` adjusts the device's configuration to match a given time. Animation is done by calling `setTime` at consecutive points in time.

A device manages its speed of motion with `setSpeed` and `getSpeed` commands. However, it is up to the definition of `setTime` to use or to ignore the speed.

```
void setSpeed(speed);
double getSpeed();
```

There are two types of devices – non-drawing and drawing. They have different sets of properties and their functionalities differ too.

b) Non-drawing devices

A non-drawing device is a device, which has no pencil attached. Such device is moved and rotated in space with the same set of functions as mecholets.

The center of a device is managed with these functions:

```
vect getCenter();
void setCenter(center);
```

Individual axial coordinates of a device are set with these functions:

```
void setX(x);
void setY(y);
void setZ(z);
```

The orientation is set with these functions:

```
void setH(angle);
void setV(angle);
void setS(angle);
void setT(angle);
```

The orientation is retrieved with these functions:

```
double getH();
double getV();
double getS();
double getT();
```

The following example shows a typical usage of a non-drawing device:

```
MyDevice dev(…);
while( runningMecho() )
{
    dev.setTime(t);
}
```

Figure 25 show a minimal non-drawing device built by a single beam. The size of the beam is hard-coded in the device.

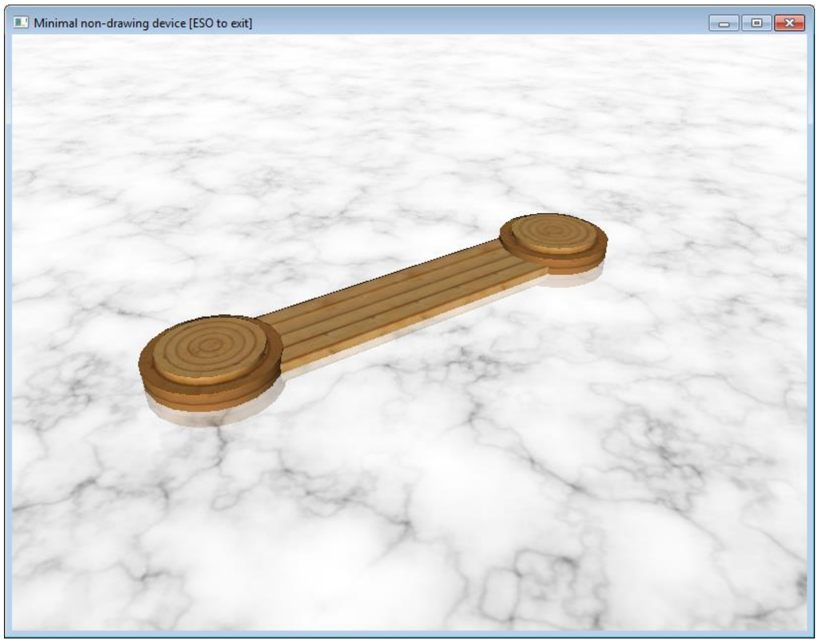


Figure 25. Minimal single-beam non-drawing device

Non-drawing devices can be "attached" to each other, thus forming more complex mechanisms. Figure 26 shows three instances of minimal non-drawing devices (like the one in Figure 25) chained into a single device.

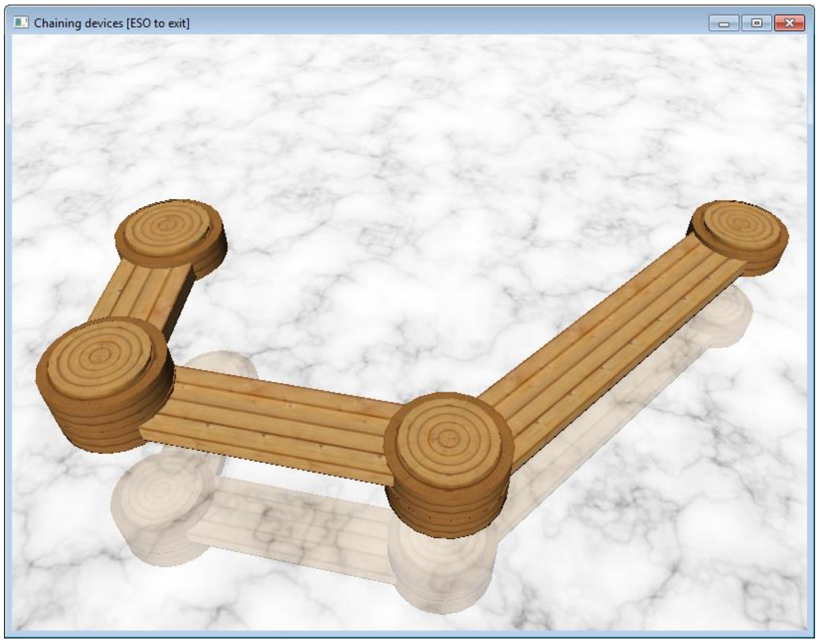


Figure 26. A chain of devices

It is possible to define several instances of externally defined device, like `Pusher` device, and control them independently (see Figure 27, left) or attach them and control them together (see Figure 27, right).

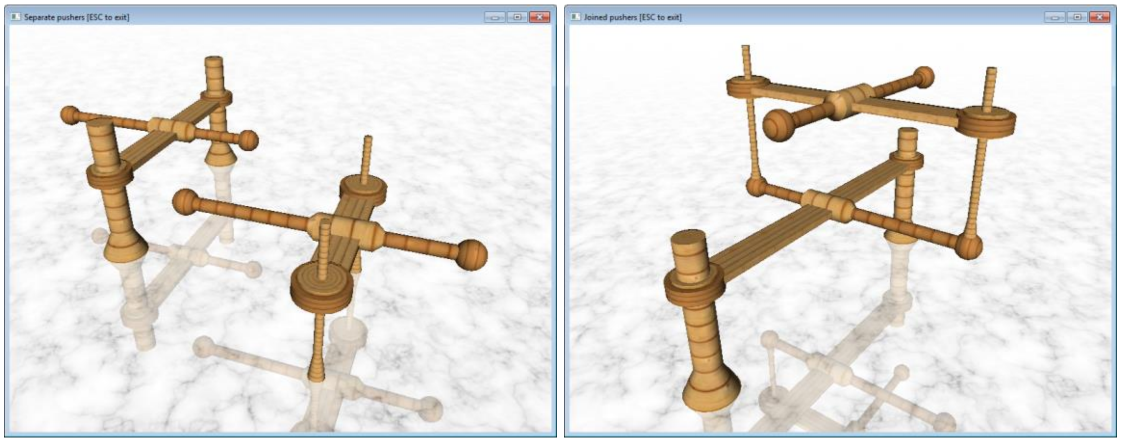


Figure 27. Separate (left) and joined (right) pusher devices

c) Drawing devices

A drawing device is a device, which has a pencil attached. Although such device could be moved and rotated in space, this will also affect the pencil traces – they will move and rotate together with the device.

Figure 28 shows a drawing device leaving 3D traces. By design, it is up to the device creator to limit pencil traces to the ground, or to let them hover above it.

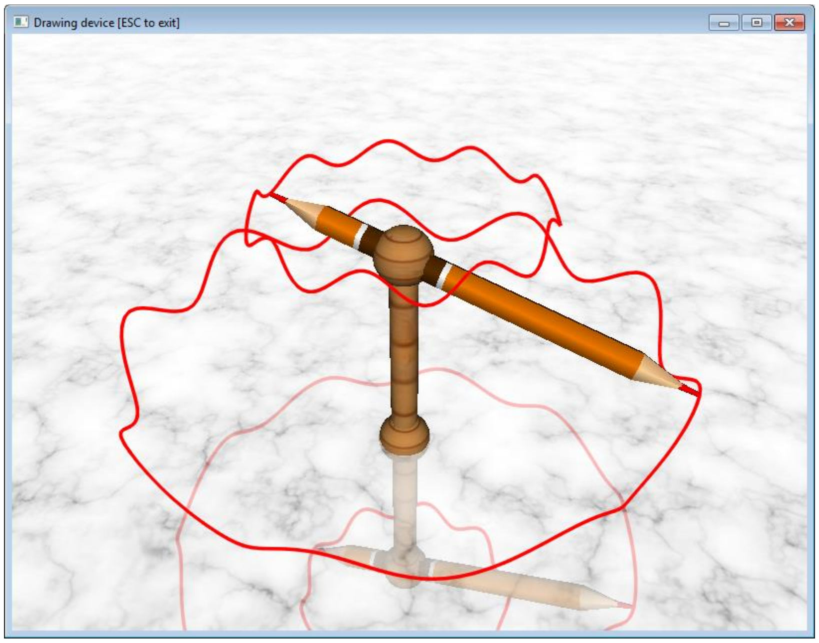


Figure 28. Drawing device

A drawing device should redefine `penUp` and `penDown` functions to allow pencil control from outside. These functions may activate or deactivate many pencils, if needed.

For example, if there are two pencils, `pencil1` and `pencil2`, in a device its `penDown` function is defined in this way:

```
void penDown(double stopTime)
{
   pencil1.penDown(stopTime);
   pencil2.penDown(stopTime);
}
```

2. PUSHER

A `Pusher` device is used for demonstration purposes. It is composed of two pillars, a beam, a tube and a rail. The beam slides up and down along the pillars. The tube is attached to the middle of the beam. The rail slides through the tube.

Pushers are shown in Figure 2 and Figure 27.

The constructor of a pusher is:

```
Pusher(size, width);
```

Parameter `size` is the distance between the two pillars and `width` is their width.

3. HYPOTROCHOID

a) Hypotrochoid

The `Hypotrochoid` device represents a gear rolling in a ring and a pencil attached relatively to the gear. The shape of a hypotrohoid is defined by *k=R/r* and *e=d/r*, see Figure 29. The device is used to build a family of hypotrochoidal devices.

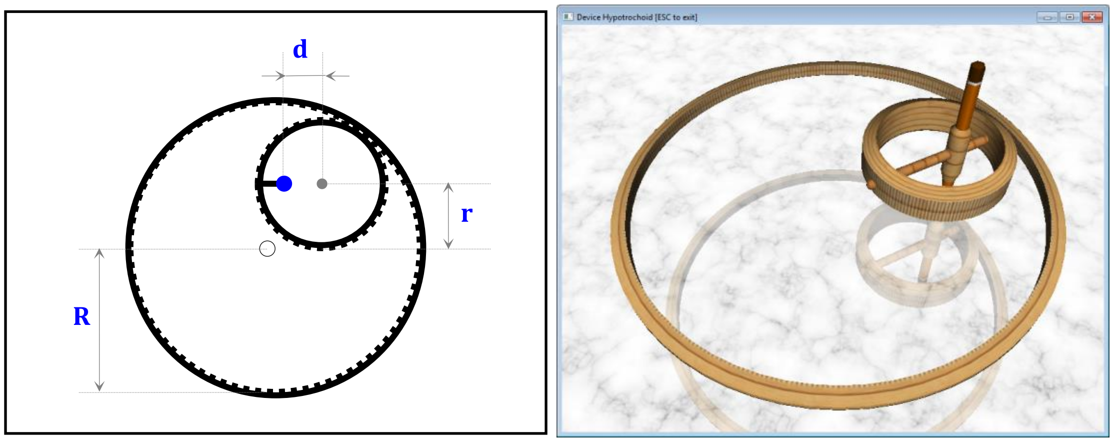


Figure 29. Hypotrochoid – dimensions and example (k=5/2, e=1/3)

The constructor of a general hypotrochoid is this:

```
Hypotrochoid(size,k,e);
```

Parameter `size` is the size of the device equal to 2*R*, `k` is *R*/*r* (and the number of cusps) and `e` is *d*/*r* (and the sharpness of the cusps).

Table 1 and Table 2 describe the visual properties of a hypotrochoid for different values of *k* and *e*.

Table 1. Visual properties of hypotrochoids for k

k	Description
$k \geq 2$	For integer *k* the shape has *k* non-overlapping cusps For rational $k = p/q$ (in simplest terms) the shape has *p* overlapping cusps at a step of *q* cusps
$1 < k < 2$	The generated shapes are similar to those when $k \geq 2$, because: • if $e \neq 0$ then $(k,e) \sim (k/(k-1), 1/e)$ • if $e = 0$ then all hypotrochoids are circles (see Table 2)
$k \leq 1$	The device is impossible

Table 2. Visual properties of hypotrochoids for e

k	Description
$e > 1$	Cusps are prolate
$e = 1$	Cusps are sharp
$0 < e < 1$	Cusps are curtate
$e = 0$	There are no cusps (the shape is a circle)
$e < 0$	Same shapes as for $e > 0$: $(k,e) \sim (k,-e)$

There is a hierarchy of other devices built upon the hypotrochoid device – see Figure 30.

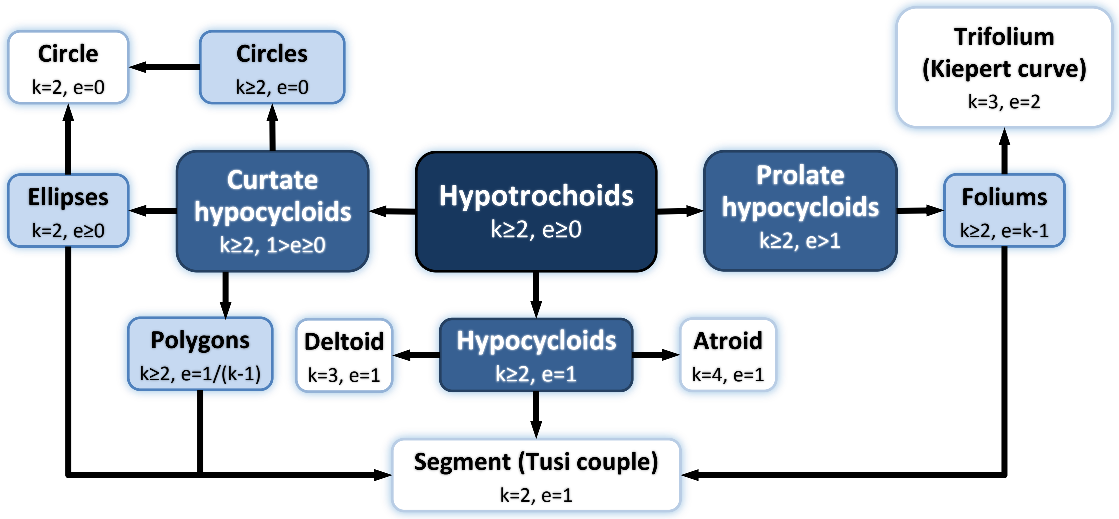


Figure 30. Hypotrochoid family

b) Hypocycloid (segment, Tusi couple, deltoid, astroid)

A hypotrochoid with *e*=1 (i.e. the pencil is on the rolling gear) is a *hypocycloid*. The shape of the curve is defined only by the number of cusps *k*.

The native constructor of a hypocycloid is this:

```
Hypocycloid(size,k);
```

Alternatively, a hypocycloid is constructed as hypotrochoid:

```
Hypotrochoid(size,k,1);
```

Figure 31 shows three hypocycloids with 7 cusps spread at different steps of 1, 2 and 3 cusps (i.e. k=7; 7/2; 7/3). Figure 32 shows some famous hypocycloids: *astroid* (*k*=4), *deltoid* (*k*=3) and *segment* (aka *Tusi couple*, *k*=2).

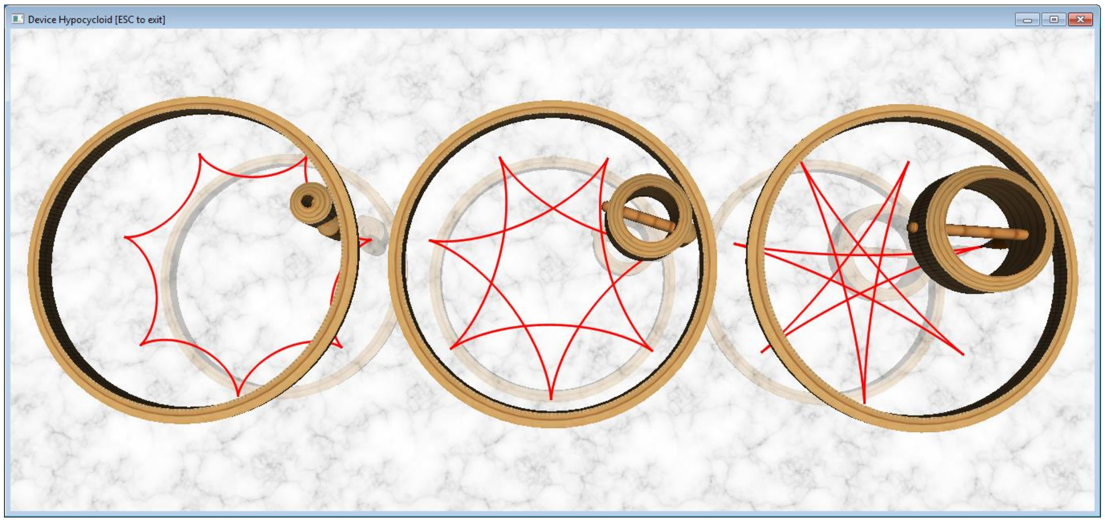


Figure 31. Hypocycloids with 7 cusps

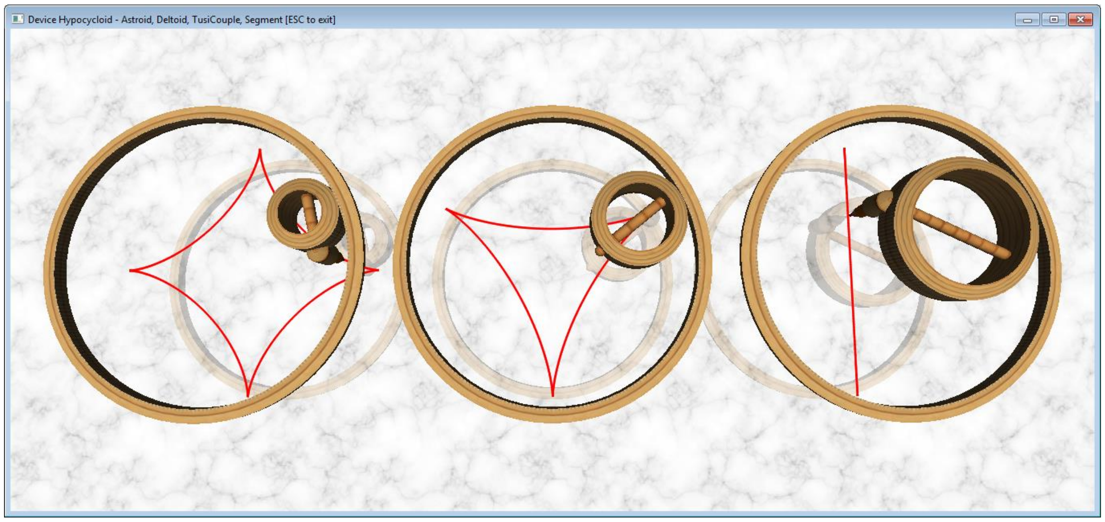

Device Hypocycloid - Astroid, Deltoid, TusiCouple, Segment [ESC to exit]

Figure 32. Asteroid, deltoid and segment/Tusi couple

The constructors of `Astroid`, `Deltoid`, `Segment` and `TusiCouple` are:

```
Astroid(size);
or Hypocycloid(size,4)
or Hypotrochoid(size,4,1);

Deltoid(size);
or Hypocycloid(size,3)
or Hypotrochoid(size,3,1);

Segment(size);
TusiCouple(size);
or Hypocycloid(size,2);
or Hypotrochoid(size,2,1);
```

A set of hypocycloid devices with the same number of cusps, but with different values of *k* draw a full hypocycloidal graph. Figure 33 shows a stack of five hypocycloid devices at *k* ranging from 11/1 to 11/5.

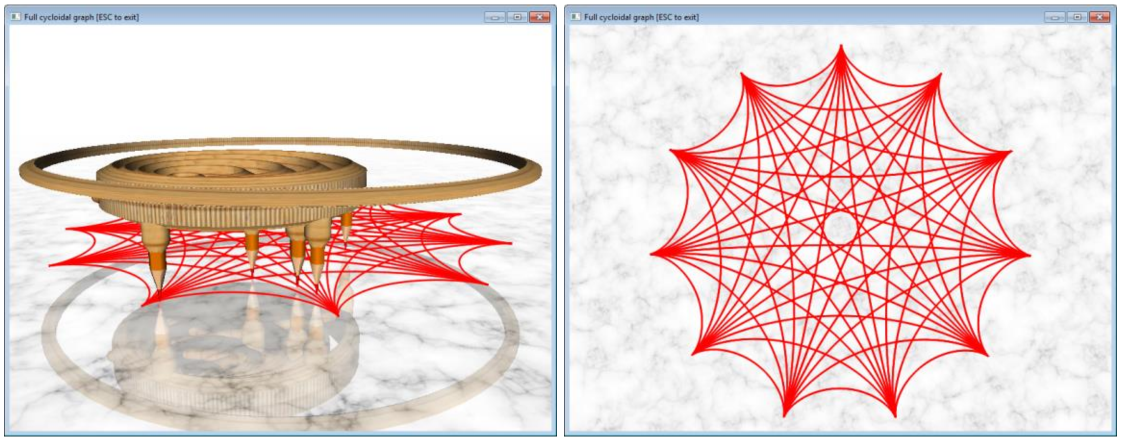

Full cycloidal graph [ESC to exit]
Full cycloidal graph [ESC to exit]

Figure 33 . Full cycloidal graph with 11 nodes *c) Curtate hypotrochoid (circle, ellipse, polygon)*

A hypotrochoid with 0≤*e*<1 is a *curtate (contracted) hypocycloid*. Its cusps, if 0<*e*, are rounded and soft.

Figure 34 shows three devices generating 7-cusp curtate hypotrochoids with e=0.5 and k=7/1, 7/2 and 7/3.

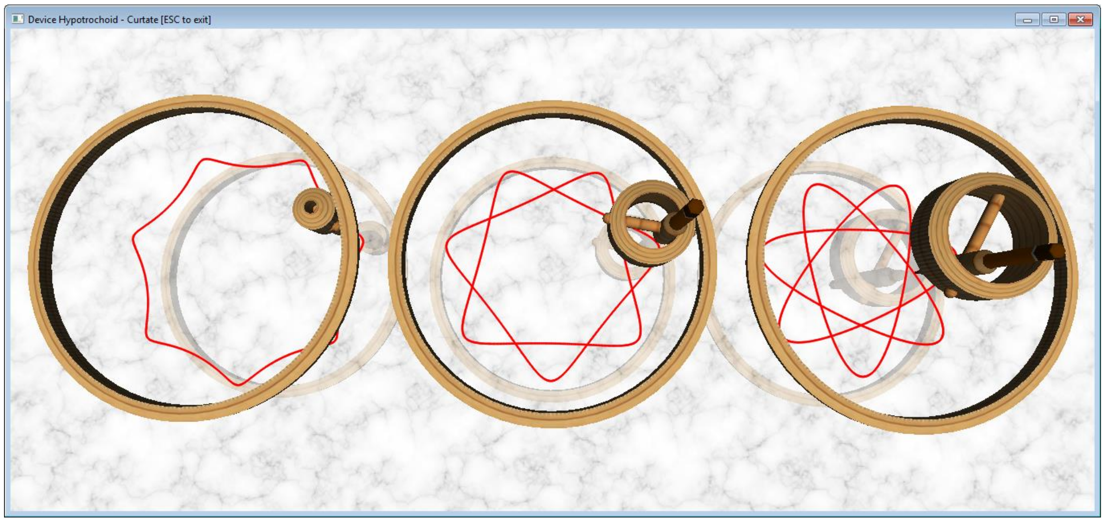


Figure 34. Curtate hypotrochoids with 7 cusps and e=0.5

Some famous curtate hypotrochoids are: *circle* (*e*=0), *ellipse* (*k*=2, *e*>0) and rounded *polygon* (*e*=1/(1-*k*)). Figure 35 shows a circle and two ellipses (*e*=0.2, *e*=0.7). Although similar, parameter e is not numerically equivalent to the ellipse eccentricity.

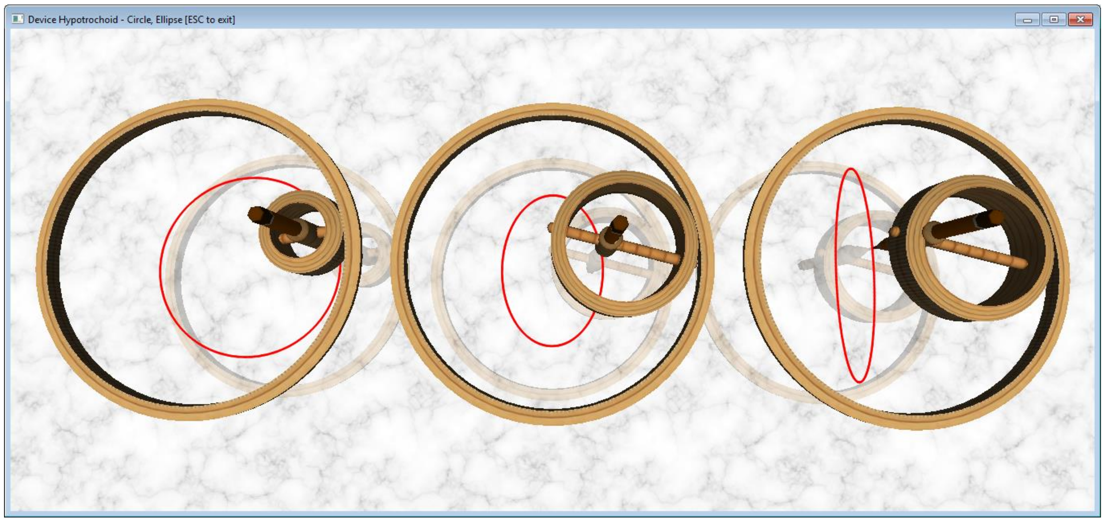


Figure 35. Circle and ellipses as curtate hypotrochoids

The constructors of `Circle` and `Ellipse` are these:

```
Circle(size, k)
or Hypotrochoid(size,k,0)

Ellipse(size,e)
or Hypotrochoid(size,2,e);
```

Figure 36 shows the construction of rounded triangle, square and hexagon. As the number of side increases, the polygon converges to a circle.

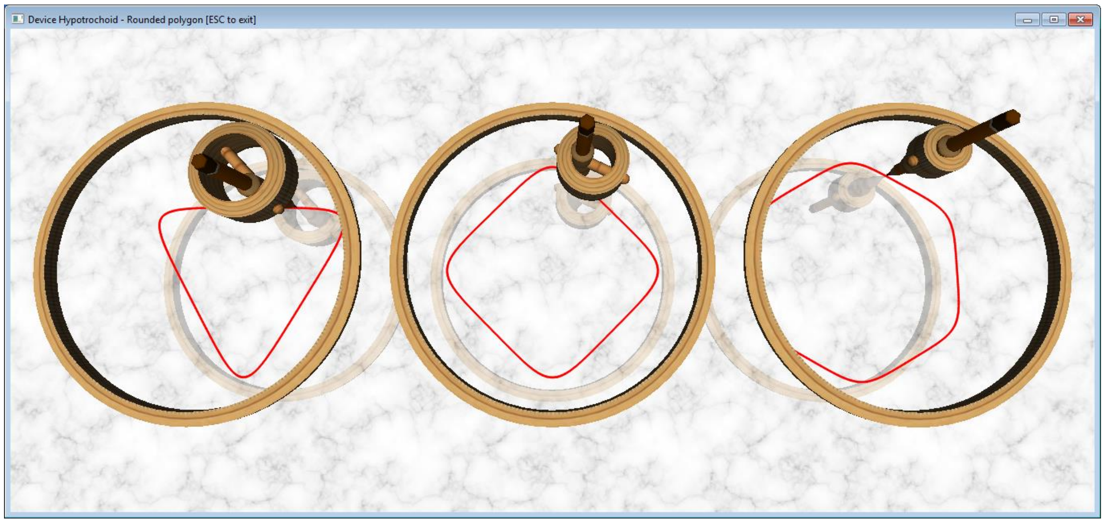


Figure 36. Triangle, square and hexagon as curtate hypotrochoidal polygons

Figure 37 shows rational polygons. The constructors of Polygon are these:

```
Polygon(size,n);
or Hypotrochoid(size,n,1/(n-1));
```

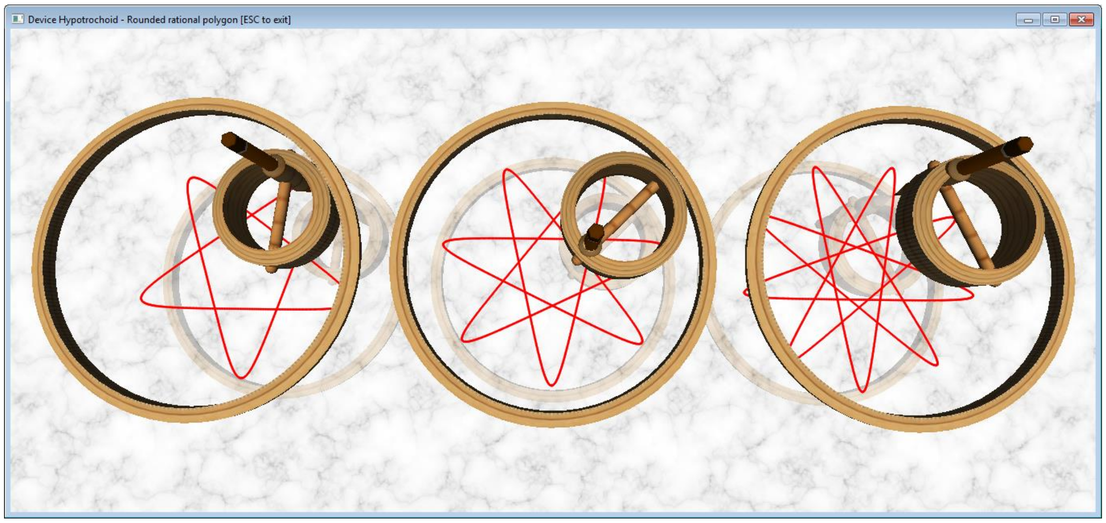

Device Hypotrochoid - Rounded rational polygon [ESC to exit]

Figure 37. Curtate hypotrochoidal polygons with fractional k: 5/2, 7/3 and 9/4

d) Prolate hypotrochoid (folium, Kiepert curve)

A hypotrochoid with *e*>1 is a *prolate (extended) hypotrochoid*. Its cusps form loops. Figure 38 shows three devices generating 5-cusp prolate hypotrochoids.

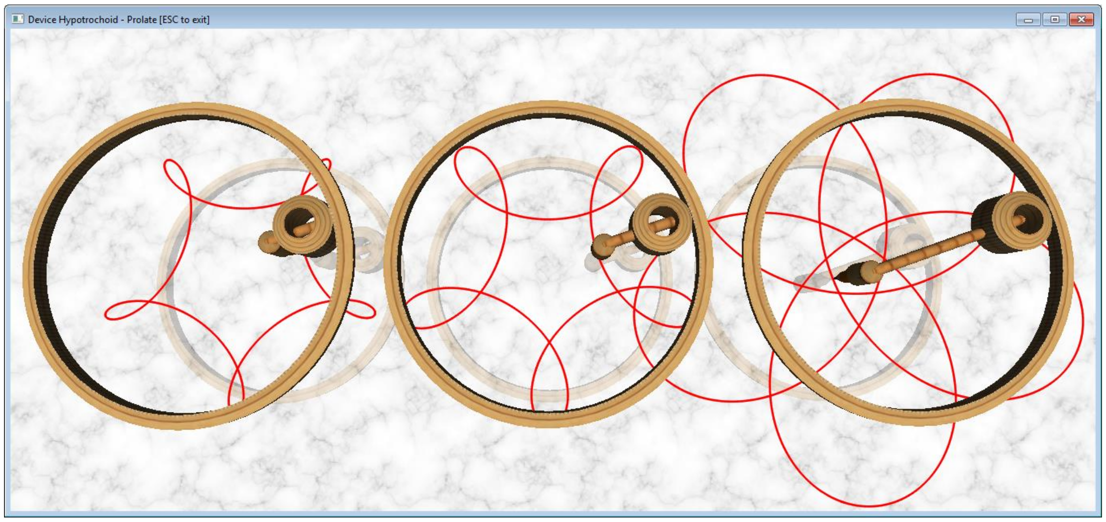


Figure 38. Prolate hypotrochoids with 5 cusps and e=1.5, e=2.0 and e=5.0

A *folium* is a prolate hypotrochoid with *e*=*k*-1. A famous folium is the *trifolium* (k=3,e=2) which is also called *Kiepert Curve*, see Figure 39. Their constructors are these:

```
Folium(size,n);
or Hypotrochoid(size,n,n-1);

Trifolium(size);
KiepertCurve(size);
or Folium(size,3);
or Hypotrochoid(size,3,2);
```

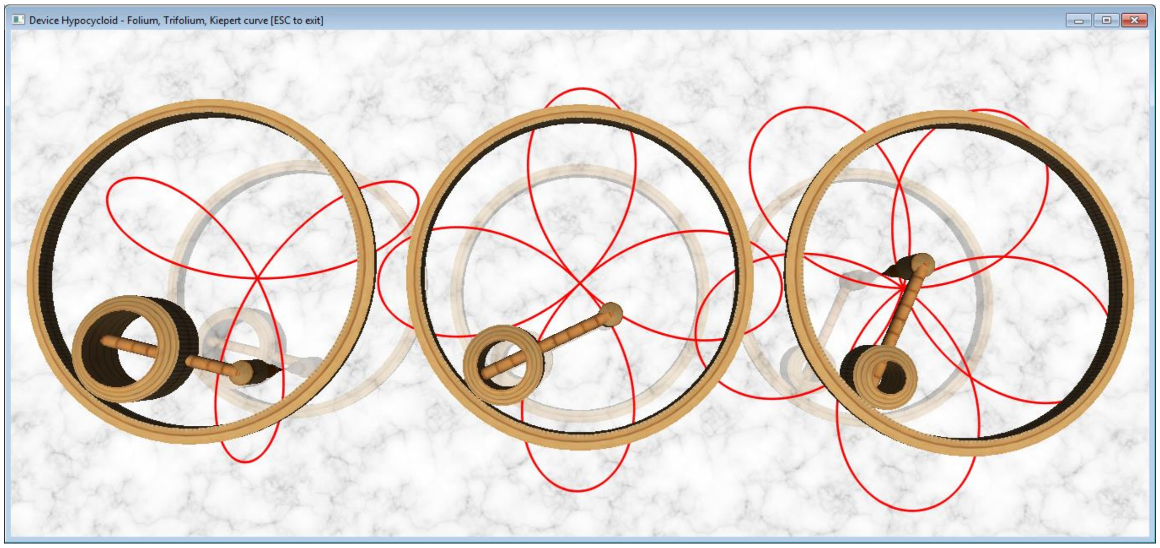

Device Hypocycloid - Folium, Trifolium, Kiepert curve [ESC to exit]

Figure 39. Foliums with k=3 (Kiepert curve), k=4 and k=5

4. EPITROCHOID

a) Epitrochoid

The `Epitrochoid` device represents a gear rolling outside another gear. There is a pencil attached to the rolling gear. The shape of an epitrochoid is defined by $k=R/r$ and $e=d/r$, see Figure 40. The device is used to build a family of epitrochoidal devices.

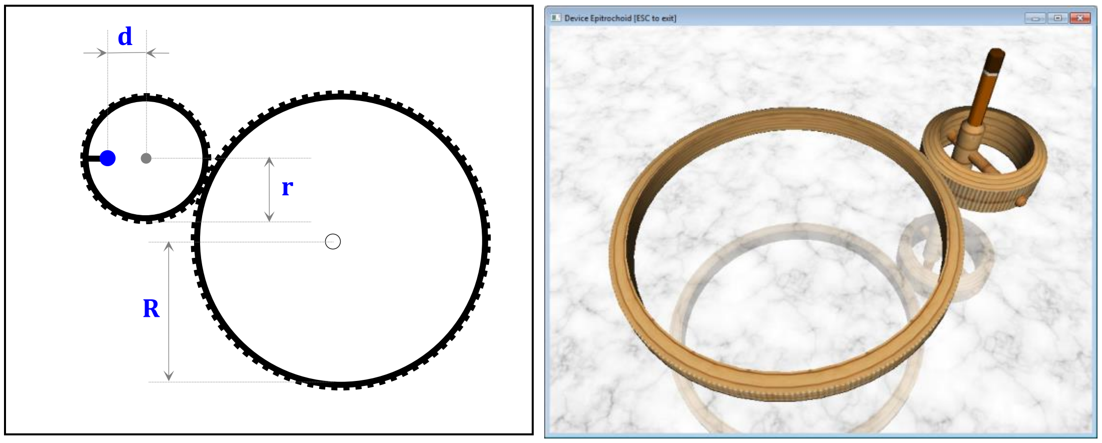


Figure 40. Epitrochoid – dimensions and example (k=5/2, e=1/3)

The constructor of a general epitrochoid is this:

```
Epitrochoid(size,k,e);
```

Parameter `size` is the size of the device equal to $2R$, `k` is R/r the number of cusps and `e` is d/r (and the sharpness of the cusps).

Table 3 and

Table 4 describe the visual properties of an epitrochoid for different values of k and e.

Table 3. Visual properties of epitrochoids for k

k	Description
$k>0$	For integer k the shape has k non-overlapping cusps For rational $k=p/q$ (in simplest terms) the shape has p overlapping cusps at a step of q cusps
$k\leq 0$	The device is impossible

Table 4. Visual properties of epitrochoids for e

k	Description
e>1	Cusps are prolate
e=1	Cusps are sharp
0<*e*<1	Cusps are curtate
e=0	There are no cusps (the shape is a circle)
e<0	Same shapes as for *e*>0: (*k*,*e*)~(*k*,-*e*)

There is a hierarchy of other devices built upon the epitrochoid device. It is shown in Figure 41.

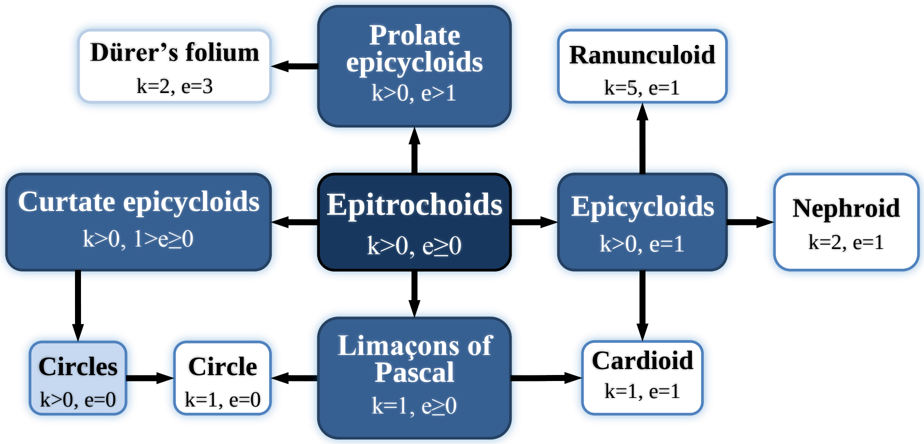


Figure 41. Hierarchy of epitrochoids

b) Epicycloid (cardioid, nephroid, ranunculoid)

An epitrochoid with *e*=1 (i.e. the pencil is on the rolling gear) is an *epicycloid*. The shape of the curve is defined only by the number of cusps *k*.

Figure 42 shows three epicycloids with 7 cusps spread at different steps of 1, 2 and 3 cusps (i.e. k=7; 7/2; 7/3).

The constructors of an epicycloid are these:

```
Epicycloid(size,k);
```
or
```
Epitrochoid(size,k,1);
```

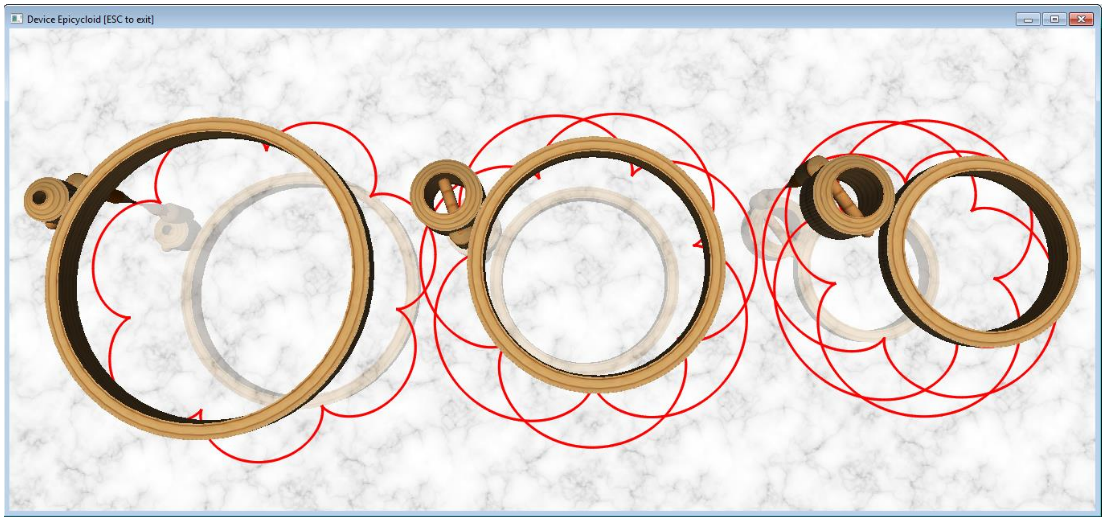


Figure 42. Epicycloids with 7 cusps

Figure 43 shows some famous epicycloids: *cardioid* (*k*=1), *nephroid* (*k*=2) and *ranunculoid* (*k*=5). Their constructors are:

```
Cardioid(size);
or Epicycloid(size,1);
or Epitrochoid(size,k,1);
```

```
Nephroid(size);
or Epicycloid(size,2);
or Epitrochoid(size,k,2);
```

```
Ranunculoid(size);
or Epicycloid(size,5);
or Epitrochoid(size,k,5);
```

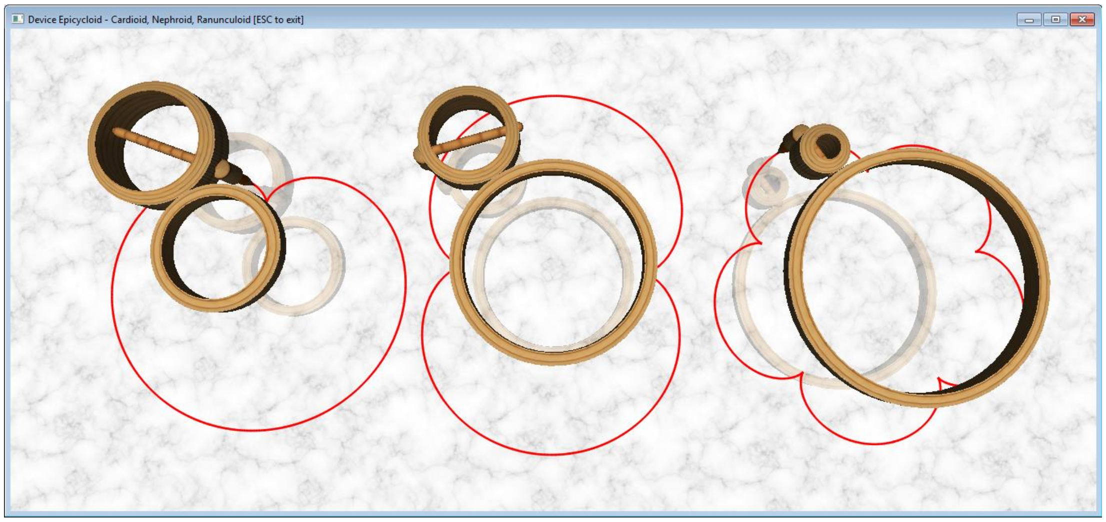

Device Epicycloid - Cardioid, Nephroid, Ranunculoid [ESC to exit]

Figure 43. Famous epicycloids – cardioid, nephroid and ranunculoid

A variety of other shapes can be achieved when 0<*k*<1, i.e. the radius of rolling gear is larger than the radius of the fixed gear.

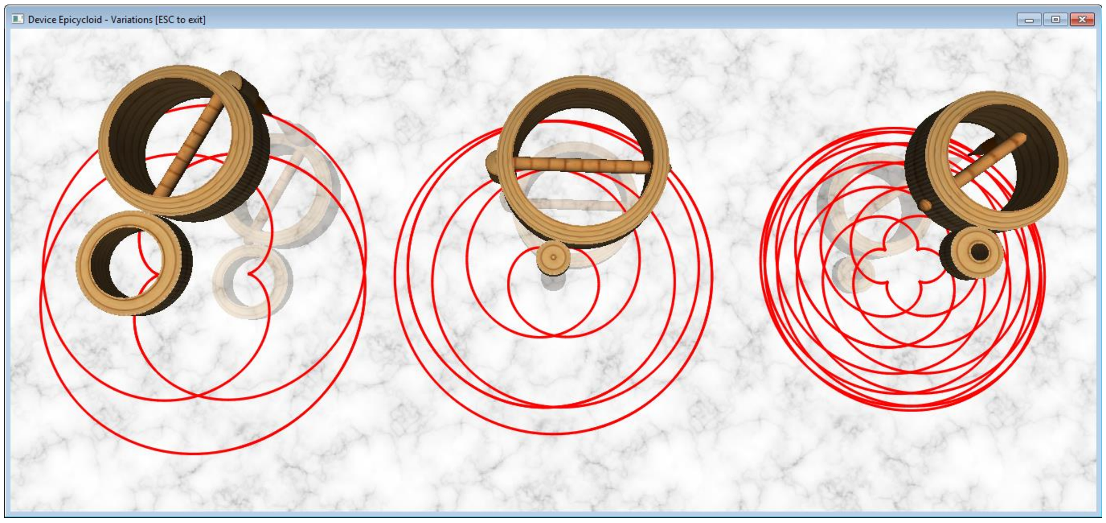


Figure 44. Epicycloids with k=2/3, k=1/5 and k=4/11

c) Curtate epitrochoid (circle)

An epitrochoid with *e*<1 is a *curtate (contracted) epicycloid*. Figure 45 shows several curtate epitrochoids with *k* equal to 7/1, 7/2 and 7/3 and *e*=0.5.

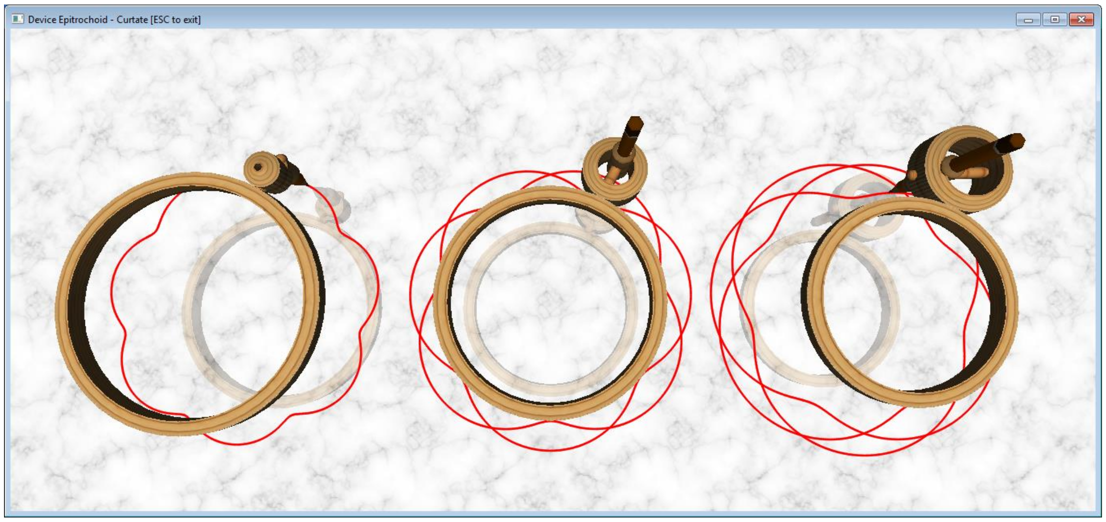


Figure 45. Curtate epitrochoids with 7 cusps and e=0.5

When *e*=0 the curtate epitrochoid is a *circle*, see Figure 46. The constructors of an epitrochoidal circle are these:

```
Circle(size,k);
```
or
```
Epitrochoid(size,k,0);
```

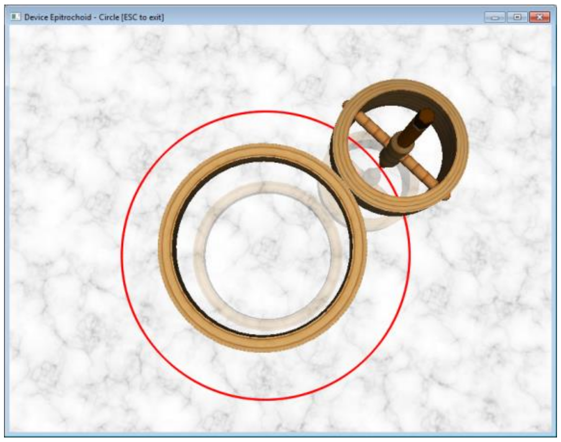


Figure 46. Circles as curtate epitrochoids

d) Prolate epitrochoid

An epitrochoid with *e*>1 is a *prolate (extended) epitrochoid*. Figure 47 shows several prolate epitrochoids with *k*=5, *e*=1.5 and 5; and *k*=5.5, *e*=6.5.

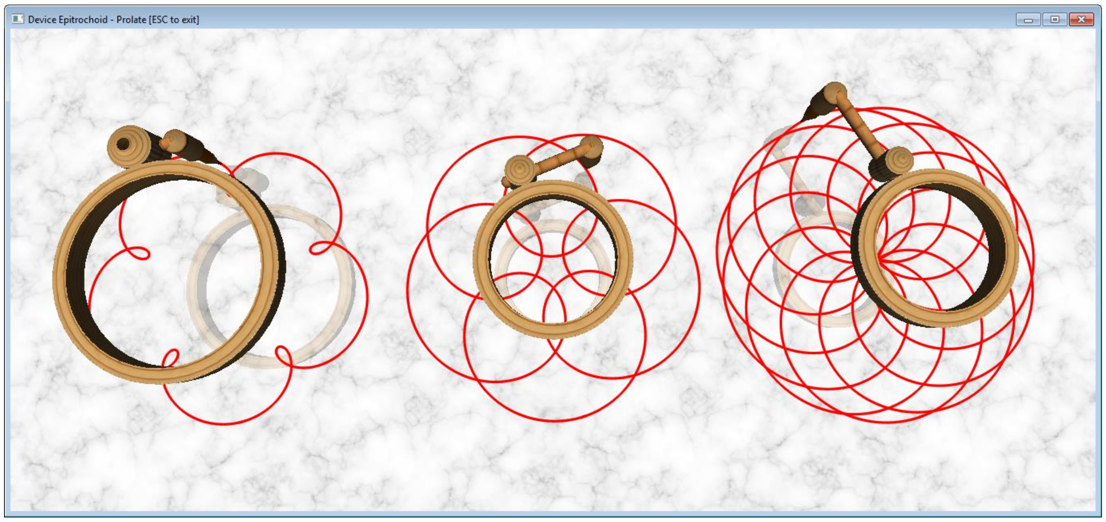


Figure 47. Prolate epitrochoids with 5 cusps and e=1.5, e=5 and k=5.5 e=6.5

A prolate epitrochoid with *k*=2 and *e*=3 is called *Dürer's folium*. It is shown in Figure 48. The constructors of *Dürer's folium* are these:

```
DurerFolium(size);
```
or
```
Epitrochoid(size,2,3);
```

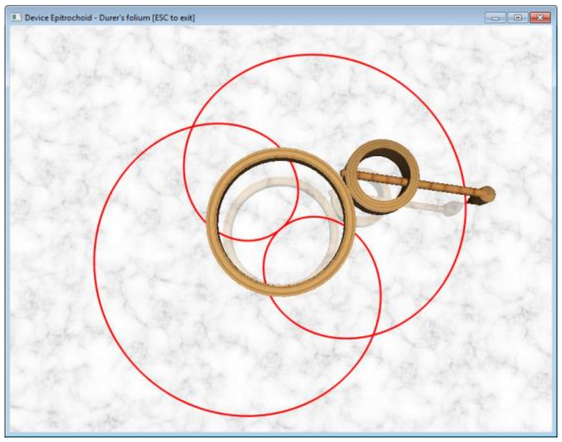


Figure 48. Dürer's folium

e) Limaçon of Pascal

An epitrochoid with *k*=1 is a *Limaçon of Pascal*. The limaçon at *e*=1 is a *cardioid*. The limaçon at *k*=1 and *e*=0 is a *circle*. The constructors of a limaçon of Pascal are these:

```
Limacon(size,e);
```
or
```
Epitrochoid(size,1,e);
```

Figure 49 shows devices drawing limaçons at e=0.75, 1.00 and 1.75.

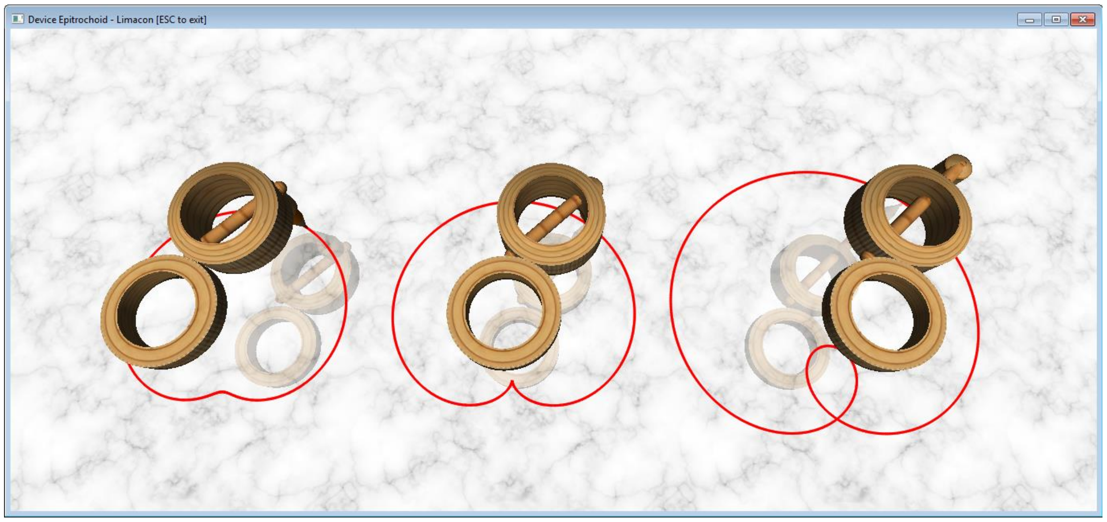
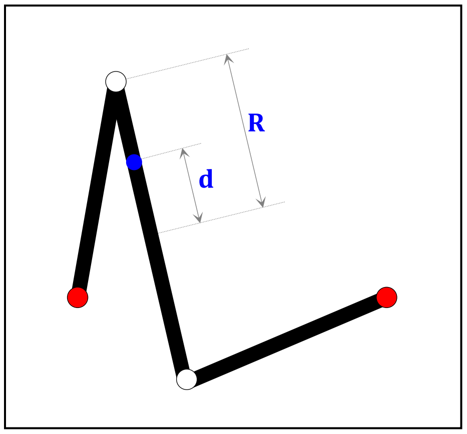


Figure 49. Limaçons of Pascal at e=0.75, e=1 and e=1.75

5. LEMNISCATE

The `Lemniscate` device represents a linkage of three beams with lengths of ratios 1:√2:1. The shorter beams have fixed ends at relative distance √2 (marked as red circles in Figure 50). One of the beams is rotating. A pencil is attached to the central (i.e. longest) beam.

The shape of the lemniscate is defined by $e=d/R$, where R is half the size of the central beam. If $|e|>1$ then the central beam is extended.



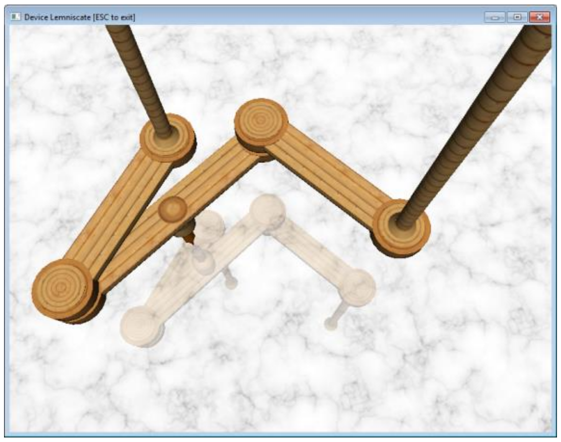


Figure 50. Lemniscate – dimensions and example (e=1/5)

The constructor of a general lemniscate is this:

```
    Lemniscate(size,e);
```

Parameter `size` is the distance between the fixed points, which is also the length of the central beam and is equal to $2R$, `e` is d/R.

Depending on the pencil position (i.e. parameter e) it is possible to generate other shapes (see Table 5).

Table 5. Visual properties of lemniscates for e

E	Description
$e=0$	Two equal loops
$0<\|e\|<1/\sqrt{2}$	Two loops
$\|e\|=1/\sqrt{2}$	Sharp cusps
$1/\sqrt{2}<\|e\|<1$	Single protruding loop
$\|e\|=1$	Circle
$\|e\|>1$	Single concave loop

Figure 51 shows two variations of lemniscates at e=0.2 (two different loops) and e=0.5 (one concave loop).

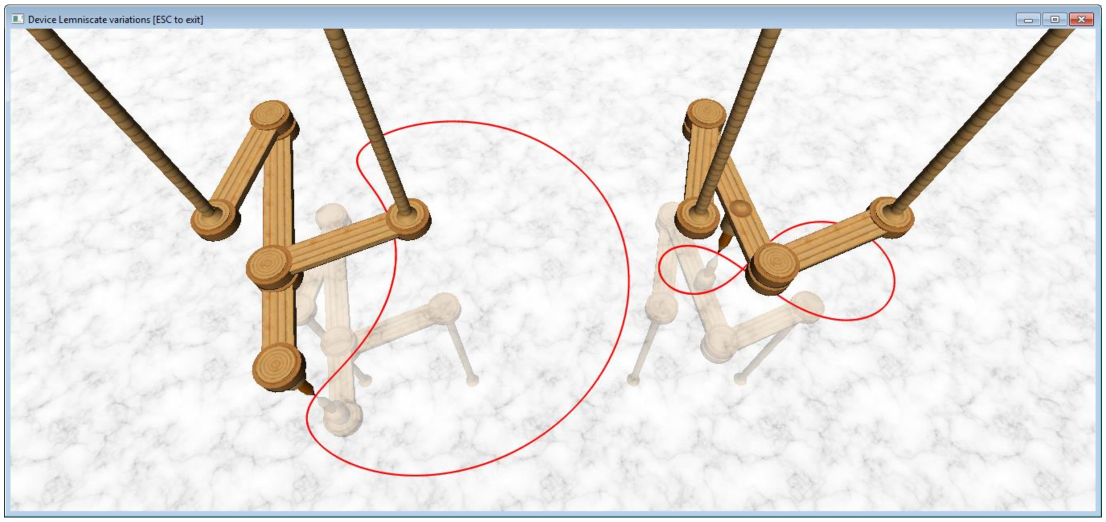


Figure 51. Lemniscate shapes at e=2 and e=0.2

There is a hierarchy of other devices which are all built upon the lemniscate device – see Figure 52. Some of the most famous one are *the lemniscate of Bernoulli*, *the lemniscate drop* and *the circle*. The transition between these curves is shown in Figure 53.

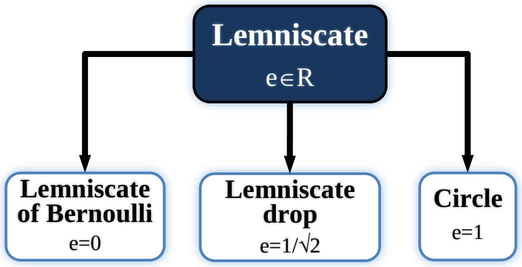


Figure 52. Hierarchy of lemniscates

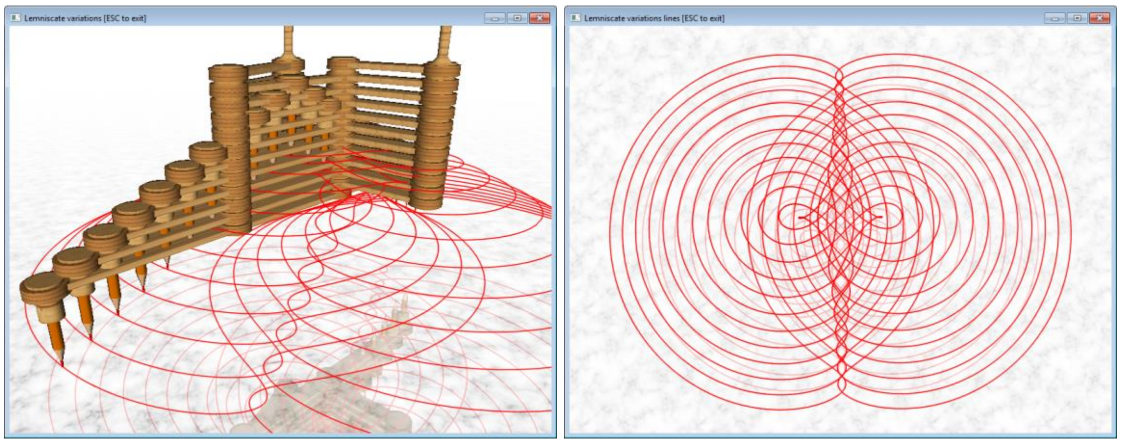

Lemniscate variations [ESC to exit]
Lemniscate variations lines [ESC to exit]

Figure 53 A full set of lemniscates

a) Lemniscate of Bernoulli

A lemniscate with *e*=0 is *a lemniscate of Bernoulli*. The loops of the lemniscate are completely symmetrical.

The constructors of the lemniscate of Bernoulli are these:

```
LemniscateOfBernoulli(size);
or Lemniscate(size,0);
```

The left device in Figure 54 draws a lemniscate of Bernoulli.

b) Lemniscate drop

A lemniscate with *e*=±1/√2 is *a lemniscate drop.* The constructors of a lemniscate drop are these:

```
LemniscateDrop(size);
or Lemniscate(size,1/sqrt(2));
or Lemniscate(size,-1/sqrt(2));
```

The middle device in Figure 54 draws a lemniscate drop.

c) Circle

A lemniscate with *e*=±1 is a *circle.* The constructors of a lemniscate circle are these:

```
Circle(size);
or Lemniscate(size,1);
or Lemniscate(size,-1);
```

The right device in Figure 54 draws a lemniscate circle.

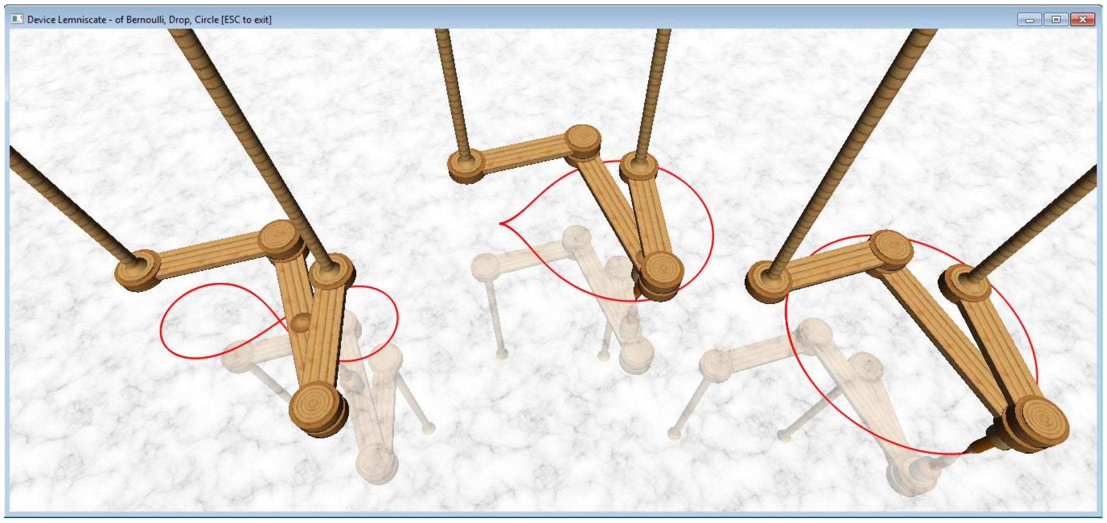

Device Lemniscate - of Bernoulli, Drop, Circle [ESC to exit]

Figure 54. A lemniscate of Bernoulli, a lemniscate drop and a lemniscate circle

6. WALKER

The Walker device represents a walking mechanisms designed by kinetic sculptor Theo Jansen. The implementation of the mechanism demonstrates the following features of Mecho:

- *Complex mechanisms* – the walker is composed of dozens of mecholets, and its walking animates most of them.
- *Submechanisms* – the four legs are instances of a leg class; they are sort of local mechanisms.
- *Subanimation* – each leg is animated independently on the others.
- *Collision* – an algorithm in the walker keeps track which leg touches the ground. This is used to define the actual walking on the ground, by keeping that leg at fixed ground position.
- *Tracking* – the view point tracks automatically the motion of the walker; yet, the user can change the view point simultaneously.
- *Parametrisation* – the walker's size and proportions of body parts are controlled by parameters. Thus different walkers are created and animated in a single virtual world.

Snapshot of the walker are shown in Figure 55, Figure 56 and Figure 57.

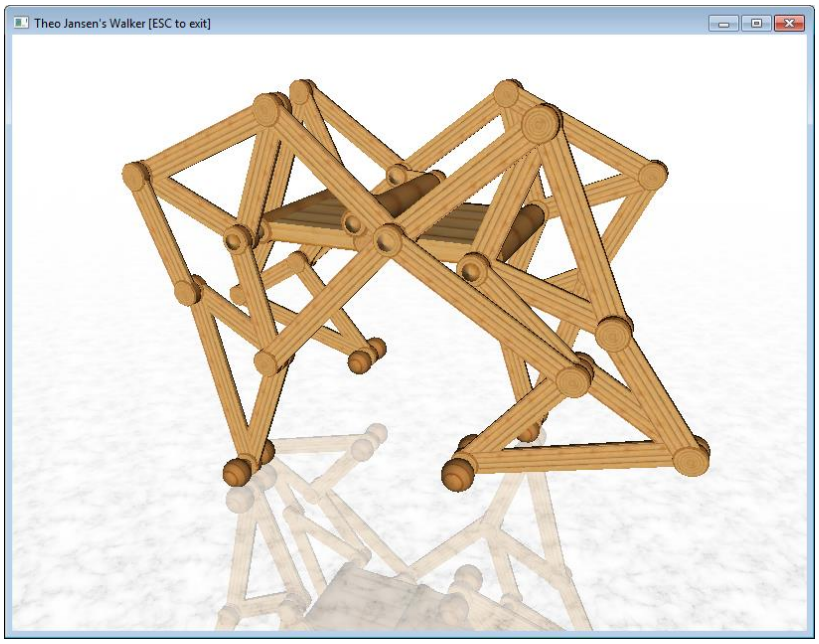


Figure 55. The walker – side view

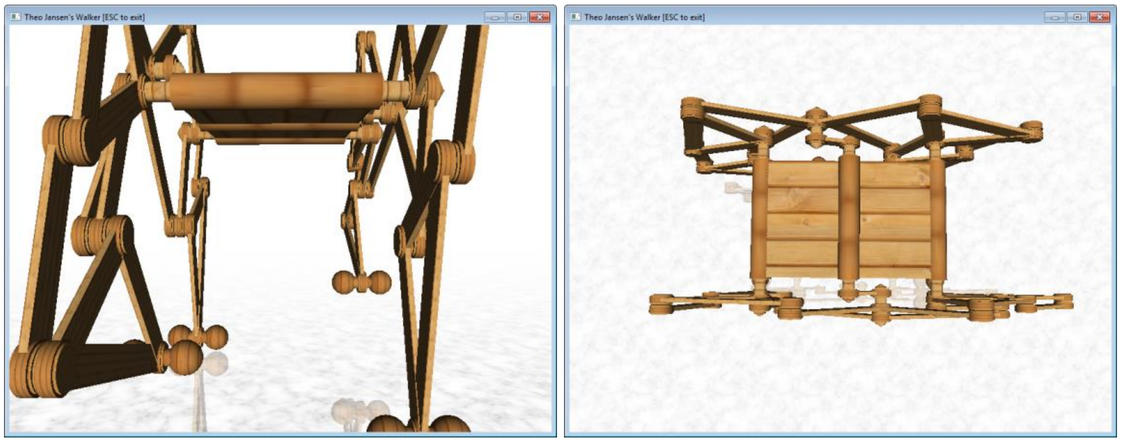

Theo Jansen's Walker [ESC to exit]
Theo Jansen's Walker [ESC to exit]

Figure 56. The walker – front and top view

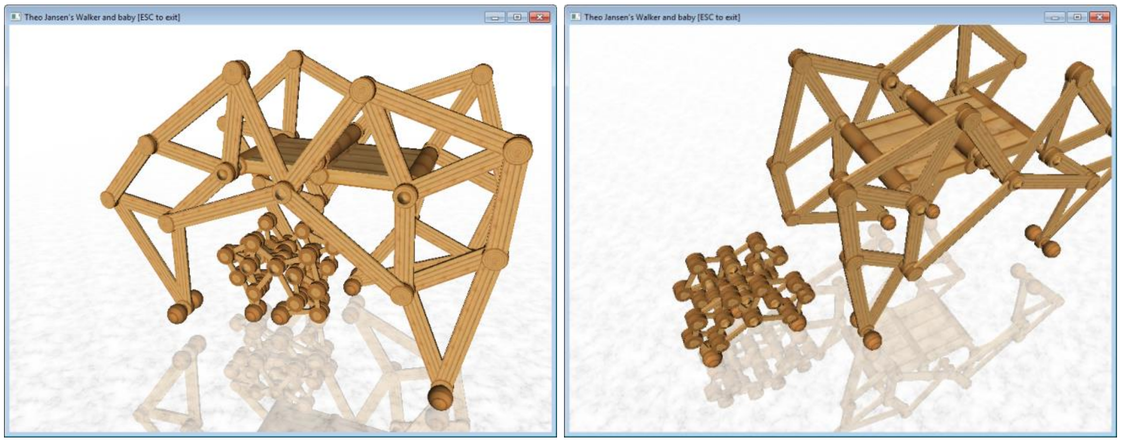

Theo Jansen's Walker and baby [ESC to exit]
Theo Jansen's Walker and baby [ESC to exit]

Figure 57. The walker and her baby