Chapter 15
Special Operators

This section presents a collection of operators that do not fit into any of the other categories. Nevertheless, most of them are extremely useful. Some of them are even necessary for the interaction of Cinderella and CindyScript.

15.1 Interaction with Geometry

CindyScript can interact in different ways with a geometric drawing that was created with Cinderella. We already saw that it can read the numerical data and appearance of geometric elements. However it can also change the position of the free elements of a construction. CindyScript may even inquire and change the construction sequence by creating and deleting new geometric elements.

15.1.1 Moving Elements

The calculations within CindyScript can be used to control the positions of free elements in a Cinderella construction. One way of doing this is to explicitly set the position information of a free element. For instance, if `A` is a free point, the line `A.xy=[1,1]` sets this point to the coordinates `[1,1]`. Another way of moving an element is with the `moveto` operator.

Moving a Free Element: `moveto(<geo>,<pos>)` **`moveto`**

Description: In this operator, `<geo>` is a free geometric object and `<pos>` (usually a vector) describes a position to which this object should be moved. Calling this operator simulates a move for this geometric object.

If `<geo>` is a free point, then `<vec>` can be a list `[x,y]` of two numbers or a list `[x,y,z]` of three numbers. The first case is interpreted as Euclidean coordinates, while the second case is interpreted as homogeneous coordinates and sets the point to `[x/z,y/z]`.

If `<geo>` is a free line, then `<vec>` has to be a list of three numbers `[a,b,c]`, and the line is set to the line described by the equation `a*x + b*y + c = 0`.

J. Richter-Gebert and U.H. Kortenkamp, *The Cinderella.2 Manual: Working with The Interactive Geometry Software*, DOI 10.1007/978-3-540-34926-6_15,

Examples: The following code lines summarize possible ways to move geometric elements (we also include the possibilities of moving elements by accessing their data fields):

```
//A is a free point
moveto(A,[1,4]);        //moves A to Euclidean coordinates [1,4]
A.xy=[1,4];             //moves A to Euclidean coordinates [1,4]
A.x=5;                  //sets the x coordinate of A to 5, lets the y
    coordinate unchanged
A.y=3;                  //sets the y coordinate of A to 3, lets the x
    coordinate unchanged
moveto(A,[2,3,2]);      //moves A to homogeneous coordinates [2,3,2]
A.homog=[2,3,2];        //moves A to homogeneous coordinates [2,3,2]

//a is a free line
moveto(a,[2,3,4]);      //moves a to homogeneous coordinates [2,3,4]
a.moveto=[2,3,4];       //moves a to homogeneous coordinates [2,3,4]

//b is a line through a point
a.slope=1;              //sets the slope of the line to 1

//C is a circle with free radius
C.radius=1;             //sets the radius of the circle to 1
```

15.1.2 Handles to Objects

mover

Who has moved: `mover()`

Description: This operator gives a handle to the element that is currently moved by the mouse.

elementsatmouse

Elements close to the mouse: `elementsatmouse()`

Description: This operator gives a list with handles to all the elements that are close to the current mouse position.

Example: The following script is a little mean. Putting it into the mouse move slot will make exactly those elements disappear that are close to the mouse. They reappear if the mouse moves away again.

```
apply(allelements(),#.alpha=1);
apply(elementsatmouse(),#.alpha=0);
repaint();
```

incidences

Incidences of an object: `incidences(<geo>)`

Description: This operator returns a list all the elements that are generically incident to a geometric element `<geo>`.

element

Getting an element by name: `element(<string>)`

Description: This operator returns the geometric object identified by the name given in `<string>`.

15.1.3 Creating and Removing Elements

Creating a free point: `createpoint(<string>,<pos>)` **`createpoint`**

Description: This operator creates a new point with label `<string>`. The point will beset to position `<pos>`. If an element with this name is already exists then no new element is created. However, if there already exists a free point with this name, then this point is moved to the specified position.

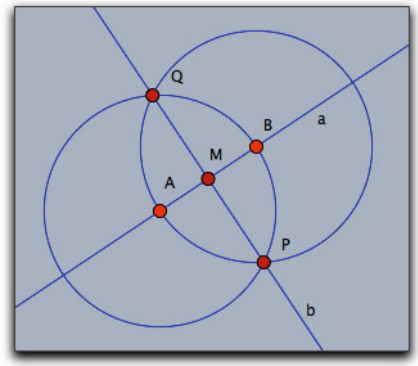
Creating a geometric element: `create(<list1>,<string>,>list2>)` **`create`**

Description: With this operator it is possible to generate arbitrary geometric elements that are functional in a geometric construction. Due to the fact that algorithms may create multiple outputs several subtleties arise. This function is meant for expert use only.

The first list contains a list `<list1>` of element names for the generated output objects of the algorithm. `<string>` is the internal name of the geometric algorithm. The second list `<list2>` is a list of the parameters that are needed for the definition. The following table shows a few possible creation statements.

```
create(["A"],"FreePoint",[[1,1,1]]);
create(["B"],"FreePoint",[[4,3,1]]);
create(["a"],"Join",[A,B]);
create(["X"],"CircleMP",[A,B]);
create(["Y"],"CircleMP",[B,A]);
create(["P","Q"],"IntersectionCircleCircle",[X,Y]);
create(["b"],"Join",[P,Q]);
create(["M"],"Meet",[a,b]);
```

This sequence of statements creates the fully functional construction shown below. Observe that in the sixth statement when two circles are intersected there must be a list of two output elements specified.



You can find the valid parameters for elements by constructing them manually and using the `algorithm` and `inputs` functions described below.

Removing a geometric element: `removeelement(<geo>)` **`removeelement`**

Description: Removes a geometric element together with all its dependent elements from a construction.

`inputs`

Input elements of an element: `inputs(<geo>)`

Description: This operator returns a list all the elements that are needed to define the object `<geo>`. These may be other geometric, elements, numbers or vectors.

`algorithm`

Algorithm of an element: `algorithm(<geo>)`

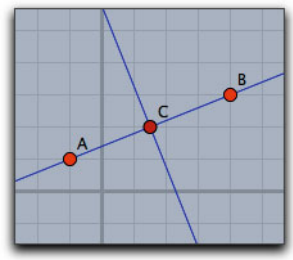
Description: This operator returns a string that resembles the algorithm of the definition the object `<geo>`.

Example: The following piece of code generates all information contained in a construction sequence.

```
els=allelements();
data=apply(els,([[#.name],algorithm(#),inputs(#)]));
```

Applied to the construction of a perpendicular bisector in the picture below it generates the following output:

```
 [[["A"],"FreePoint",[[4,4,-4]]],
  [["B"],"FreePoint",[[4,-3,1]]],
  [["a"],"Join",[A,B]],
  [["C"],"Mid",[A,B]],
  [["b"],"Orthogonal",[a,C]]
]
```



15.1.4 Accessing Element Properties

Element properties like color, size, etc. are conveniently accessible via operators like `.color`, `.size` etc. However, elements have by far more properties. All of them can be generically accessed by the following operators.

`inspect`

List all inspectable properties: `inspect(<geo>)`

Description: Returns a list of names of all private properties of a geometric element.

Example: The operator `inspect(A)` applied to a the free point *A* returns the following list of property names.

```
[name,definition,color,color.red,color.blue,color.green,alpha,visibility
    ,
drawtrace,tracelength,traceskip,tracedim,render,isvisible,
```

```
text.fontfamily,plane,pinning,incidences,labeled,textsize,textbold,
    textitalics,
ptsize,pointborder,printname,point.image,
point.image.media,point.image.rotation,freept.pos]
```

Accessing an inspectable property: `inspect(<geo>,<string>)` **inspect**

Description: Accesses an arbitrary inspectable property.

Example: One can access the color of a point *A* by `inspect(A,"color")`

Setting an inspectable property: `inspect(<geo>,<string>,<data>)` **inspect**

Description: Setting the value of inspectable property.

Example: One can set the color of a point *A* to white by `inspect(A,"color",(1,1,1))`

Forcing a repaint operation: `repaint()` **repaint**

Description: This operator causes an immediate repaint of the drawing surface. This operator is meant to be used whenever a script has updated a construction and wants to display the changes. It is not allowd to use this operator in the `draw` or in the `move` slot.

Forcing a delayed repaint operation: `repaint(<real>)` **repaint**

Description: As `repaint` but with a time delay of as many milliseconds as given by he parameter

Points on a locus: `locusdata(<locus>)` **locusdata**

Description: This operator returns a list of points in *xy*-coordinates that are all on a locus given by the name `<locus>` of a geometric element.

15.2 File Management

There is a number of operations that allow for the interaction of CindyScript with files that are stored elsewhere on the computer. Please note that these commands will not work with applets in HTML pages.

15.2.1 Reading Files

Loading data: `load(<string>)` **load**

Description: This operator takes the argument `<string>`, which is considered to be a file name (possibly preceded by directory information). If the file name is legitimate, then the entire information contained in the file will be returned as a string. This operator is particularly useful together with the `tokenize` operator, which

helps to analyze structured data. The data are read from the currently active directory, which can be set with the `setdirectory` operator.

Example: Assume that in the file `LoadDemo.txt` contains the data

```
abc,gfdg;1,3,5.6,3.141;56,abc,xxx,yyy
```

The following code reads the data and creates a list by tokenizing it with respect to ";" and ",".

```
x=load("LoadTest.txt");
y=tokenize(x,(";",","));
apply(y,println(#));
```

The resulting output is

```
[abc,gfdg]
[1,3,5.6,3.141]
[56,abc,xxx,yyy]
```

`import` **Importing program code: `import(<string>)`**

Description: This operator takes the argument `<string>`, which is considered to be a file name (including directory information). If the file name is legitimate, then the whole content of the file is assumed to be able to be parsed by CindyScript code, and it is immediately executed. In this way, one can load libraries with predefined functionality. It is advisable to use the `import` operator only in the "Init" section of CindyScript, since otherwise, the file will be read for each move.

`setdirectory` **Setting the directory: `setdirectory(<string>)`**

Description: This operator sets the directory for all subsequent file operations.

15.2.2 Writing Files

It is also possible to write files by a sequence of Cindy script commands. The usual cycle for writing is: Open a file – write to it – close the file. This can be done using the following commands.

`openfile` **Opening a file: `openfile(<string>)`**

Description: Opens a file with the specified name. The function returns a handle to the file that is needed for subsequent print operations.

`println` **Println to a file: `println(<file>,<string>)`**

Description: Identical to the `println(...)` command. However this command prints to the file specified by `<file>`.

`print` **Print to a file: `print(<file>,<string>)`**

Description: Identical to the `print(...)` command. However this command prints to the file specified by `<file>`.

Print to a file: `closefile(<file>)` **`closefile`**

Description: This command finally closes the file.

Example: The following example illustrates a file write cycle:

```
f=openfile("myFile");
println(f,"Here are some numbers");
forall(1..15,print(f,#+" ");
println(f,"");
closefile(f);
```

This code generates a file with the following content:

```
Here are some numbers
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

15.2.3 Connection to HTML

Opening a web page: `openurl(<string>)` **`openurl`**

Description: Opens a browser with the webpage given in <string>.

Calling javascript: `javascript(<string>)` **`javascript`**

Description: In exported applets this statement calls a statement in the Javascript environment of the browser. The statement is given by the content of the `<string>`. In the standalone application this statement does nothing.

Example: The following piece of script will cause a message window to pop up in the browser:

```
  javascript("alert('Hi from Cinderella!!')");
```

15.2.4 Network Connections

The TCP commands of Cinderella are rudimentary at best, but they provide the basic functionality necessary for simple networking. You should be able to send and retrieve data over the internet.

Open a TCP port: `openconnection(<string>,<int>)` **`openconnection`**

Description: Opens a bidirectional tcp connection to the server specified by the first argument and the port specified by the second argument. The return value is a handle to this network connection.

Write to a TCP connection: `print(<handle>,<string>)` **`print`**

Write to a TCP connection: `println(<handle>,<string>)` **`println`**

Description: The `print` and `println` functions not only support writing to a file, but also to a network connection created by `openconnection`.

flush **Flush output to a TCP port: `flush(<handle>)`**

Description: Flushes the output buffer of the given connection.

readln **Read from a TCP connection: `readln(<handle>)`**

Description: Reads a line from the given connection. If no data can be read, this command times out after 5 seconds.

closeconnection **Close a TCP connection: `closeconnection(<handle>)`**

Description: Closes the connection given by the handle.

Example: In the following example we open a connection to a web server and read the HTML code from there.

```
x=openconnection("cermat.org",80);
println(x,"GET /");
y="";
while(!isundefined(y),y=readln(x);println(y));
closeconnection(x);
```

15.3 Console Output

In the CindyScript input window you find a console for text output. For most practical purposes this will not be used for the final construction. However, it is extremely useful for debugging.

print **Printing text: `print(<expr>)`**

Description: This operator prints the result of evaluating `<expr>` to the console.

err **Printing text: `err(<expr>)`**

Description: Prints the result of evaluating `<expr>` to the console. If `<expr>` is a variable, the variable name is printed as well. Very useful for debugging.

println **Printing text: `println(<expr>)`**

Description: This operator prints the result of evaluating `<expr>` to the console and adds a newline character to the end of the text.

println **Printing a newline: `println()`**

Description: This operator prints a newline character to the console.

clearconsole **Clearing the console: `clearconsole()`**

Description: Removes all text from the console.

Conditional print: `assert(<bool>,<expr>)` **assert**

Description: This operator is mainly meant for convenience purpose when generating own error mesages. it is equivalent to `if(!<bool>,println(<expr>))`. It can be used to test wheter a condition is met and otherwise generate an error message.

Example: A typical usage of this operator is the following:

```
assert(isinteger(k),"k is not an integer");
```

Output a status message: `message(<expr>)` **message**

Description: This operator shows the result of evaluating `<expr>` in the status line of the application, or in the status line of the browser for Cinderella applets.

15.4 Timing and Animations

15.4.1 Time and Date

CindyScript has an internal clock that provides access to the current date and time. The clock can be also used to synchronize some automated animations. Furthermore, an operator is provided that is synchronized with the current timestamp of a running animation or physics simulation.

Accessing time: `time()` **time**

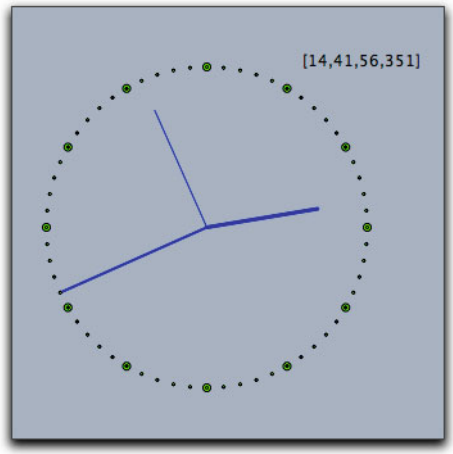
Description: This operator returns a list `[h,m,s,ms]` of four integers. The four values correspond to "hour," "minute," "second," "millisecond" on the computer's clock.

Example: The following code produces a simple clock on a Cinderella view. The variable `t` contains the time information. The subsequent code is used to produce a clocklike drawing on the view. An auxiliary function `p(w)` is defined that produces points on the unit circle. The code must be placed in the "Tick" section of CindyScript in order for it to run continuously.

```
t=time();

p(x):=[sin(2*pi*x),cos(2*pi*x)];
O=[0,0];
S=p(t_3/60)*4;
M=p(t_2/60)*5;
H=p((t_1*60+t_2)/(12*60))*3.5;
draw(O,S);
draw(O,M,size->2);
draw(O,H,size->3);
apply(1..12,draw(p(#/12)*5));
apply(1..60,draw(p(#/60)*5,size->1));

drawtext((3,5),t);
```



`date`

Accessing date: `date()`

Description: This operator returns a list `[y,m,d]` of three integers. The three values correspond to "year," "month," and "day" on the computer's calendar.

`seconds`

Timestamp: `seconds()`

Description: Returns the time elapsed since the last evaluation of `resetclock()`. The time is scaled in a way such that one unit corresponds to one second. The time's resolution is on the millisecond scale.

`resetclock`

Resetting the internal seconds: `resetclock()`

Description: Resets the value of the `seconds()` operator.

`simulationtime`

Accessing the timestamp of a simulation: `simulationtime()`

Description: This operator gives a handle to the running time clock synchronized with the progression of an animation or simulation.

Caution: This operator is still experimental.

`wait`

Pause the script for a specified time: `wait(<real>)`

Description: Will completely halt every script execution for a number of milliseconds as given by the parameter.

Example: The following code produces an acoustic jingle.

```
repeat(25,i,
  playtone(72+i);
  wait(100);
)
```

15.4.2 Animation Control

Starting the animation: `playanimation()` `playanimation`

Description: This statement starts the animation. Also physics simulation in CindyLab (p. 167) depend on running animations.

Pausing the animation: `pauseanimation()` `pauseanimation`

Description: This statement pauses the animation.

Stopping the animation: `stopanimation()` `stopanimation`

Description: This statement stops the animation. Stopping an animation also causes the geometric elements to be restored to there original position.

15.5 User Input

Sometimes it is necessary to handle user input by mouse or the keyboard explicitly. There are special evaluation times “Mouse Down,” “Mouse Up,” “Mouse Click,” “Mouse Drag,” and “Key Typed” for this (see Entering Program Code (p. 225)). These evaluation times are captured exactly when the corresponding events occur. If one wants to react to the corresponding event data, there are several operators that read the input data.

15.5.1 Mouse and Key

Mouse position: `mouse()` `mouse`

Description: Returns a vector that represents the current position of the mouse if the mouse is pressed. The vector is given in homogeneous coordinates (this allows also for access of infinite objects). If one needs the two-dimensional euclidean coordinates of the mouse position one can access them via `mouse().xy`.

Key input: `key()` `key`

Description: Returns a string that contains the last typed character.

Is a certain key pressed: `iskeydown(<int>)` `iskeydown`

Description: This operator returns a boolean value that is true if a certain key is pressed. The key in question is specified by the integer in the argument. This operator can be used to determine for instance the *shift* key is pressed. The codes for keys are usually 65, 66, 66, ... for ’A’, ’B’, ’C’,... Codes for ’shift’, ’crtl’ and ’alt’ are usually 16, 17, 18.

keydownlist **List of all pressed keys: `keydownlist()`**

Description: This operator returns a list of the codes of all pressed keys. An interesting application of the keydown list id given in the chapter on MIDI functions, where you find an example keyboard piano (p. 375).

15.5.2 AMS Data on Gravity

On Apple hardware, CindyScript can access the gravity sensor of a laptop and determine its relative orientation in space. The gravity sensor returns a three dimensional vector.

amsdata **Getting raw AMS data: `amsdata()`**

Description: This operator returns the raw data of the AMS sensor.

calibratedamsdata **Getting calibrated AMS data: `calibratedamsdata()`**

Description: This operator returns a calibrated version of the AMS sensor data. The calibrated data is a vector of unit length that represents the orientation of the computer in space.

15.5.3 Creating Custom Toolbars in a View

Cinderella can be used to export interactive worksheets to an html page. Very often it is desirable not only to export an interactive construction but also a set of construction tools along with it (like buttons for constructing points, lines or circles). By using the following set of CindyScript operations it is easily possible to create (and remove) custom toolbars that reside within an applet window.

Toolbars are in particular important for creating interactive student exercises. An example for this is given in Interactive Exercises (p. 433).

createtool **Creating a custom toolbar: `createtool(<string>,<int>,<int>)`**

createtool **Creating a custom toolbar: `createtool(<list>,<int>,<int>)`**

Description: Creates one or many toolbuttons in a Cinderella view. The first argument is either a string that describes a single construction tool or a list or matrix of strings that describe an entire toolbar. The other two arguments describe the position relative to a corner of the screen in pixel distances. Normally a createtoolbar statement is located in the *init* slot of the script editor.

The following string identifiers that correspond to the construction tools are available:

- *Moving:* `"Move"`
- *Points:* `"Point"`, `"Intersection"`, `"Mid"`, `"Center"`
- *Lines:* `"Line"`, `"Segment"`, `"Line Through"`, `"Parallel"`, `"Orthogonal"`, `"Angle Bisector"`
- *Circles:* `"Circle"`, `"Circle by Radius"`, `"Compass"`, `"Circle by 3"`, `"Arc"`
- *Conics:* `"Conic by 5"`, `"Ellipse"`, `"Hyperbola"`, `"Parabola"`

- *Special:* `"Polar Point"`, `"Polar Line"`, `"Polygon"`, `"Reflection"`, `"Locus"`
- *Measure:* `"Distance"`, `"Angle"`, `"Area"`

It is also possible to add construction tools from CindyLab (p. 167):

- *Local:* `"Mass"`, `"Velocity"`, `"Rubberband"`, `"Spring"`, `"Coulomb"`
- *Environmentsl:* `"Gravity"`, `"Sun"`, `"Floor"`, `"Bouncer"`, `"Magnet"`

The position of the tools is fixed relative to the construction view. By default the upper left corner is chosen. By using the modifyer `reference` one can also choose the other corners. Allowed values for this modifier are `"UL", "UR", "LL", "LR"`. Here the first letter stands for *upper/lower* and the second letter stands for *left/right*.

Examples: The simplest usage is for instance given by the following piece of code. The tool created by

```
createtool("Move",2,2);
```

More complicated examples that create toolbars with several tools are given below

```
createtool(["Move","Point","Line","Circle"],2,2);
```

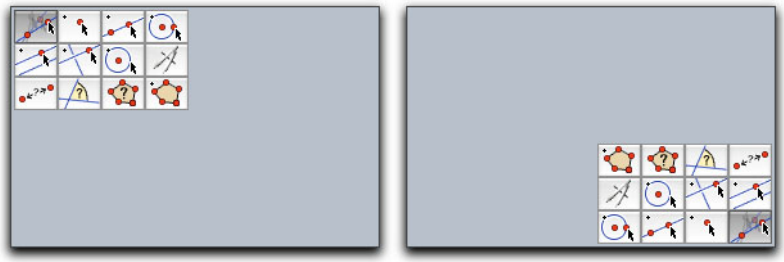
```
createtool(["Move","Point","Line","Circle"],2,2,flipped->true);
```

```
createtool(
 [
  ["Move","Point","Line","Circle"],
  ["Parallel","Orthogonal","Circle by Radius","Compass"],
  ["Distance","Angle","Area","Polygon"],
 ]
,2,2,flipped->false);
```

```
createtool(

...same as example above...

,reference->"LR");
```



Modifiers: The createtool operator can handle the modifiers summarized in the following table:

reference	<*string*>	reference position
flipped	<*bool*>	`flipped->true` exchanges rows and columns
space	<*int*>	spacing (in pixels) between tools

`removetool`

Removing a tool from a custom toolbar: `removetool(<string>)`

Description: Removes a tool from the custom toolbar.

`removetools`

Removing all custom toolbars: `removetools()`

Description: Removes all tools from the custom toolbar.

15.6 Interaction with CindyLab

`simulation`

The simulation environment: `simulation()`

Description: This operator provides a handle to the simulation environment. The simulation environment offers several fields than can be used to access its global properties.

Property	Writeable	Type	Purpose
`friction`	yes	real	total friction of the simulation
`gravity`	yes	real	total gravity of the simulation
`kinetic`	no	real	total kinetic energy of the simulation
`ke`	no	real	total kinetic energy of the simulation
`potential`	no	real	total potential energy of the simulation
`pe`	no	real	total potential energy of the simulation

`addforce`

Applying a force: `addforce(<mass>,<vec>)`

Description: Applying a force `<vec>` to an existing mass `<mass>`. This operator is useful to implement user defined force fields. It should be called in the `Integration Tick` slot.

`setforce`

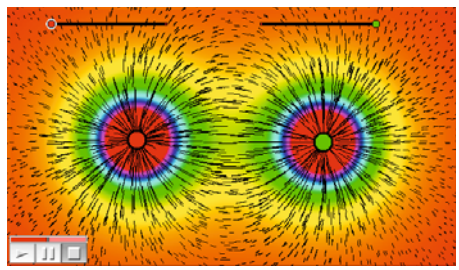
Applying a force: `setforce(<mass>,<vec>)`

Description: Setting the force `<vec>` for an existing mass `<mass>`. This operator is very useful to implement user defined force fields. It should be called in the `Integration Tick` slot.

Probing particle forces: `force(<vector>)` **`force`**

Description: The operator `force` is closely related to physics simulations in CindyLab (p. 167). It can be used for testing the force that would affect a mass particle at a specific position. The vector `<vector>` represents the position. The operator returns a two-dimensional vector that is the force at this position. If no modifiers are used, the operator assumes that the probe particle has `mass=1`, `charge=1` and `radius=1` (see Free Mass (p. 175)).

Example: The following picture was generated using the `drawforces` operator and a color plot of the `force` operator. It shows the force field and force strength of the electrostatic field of two charges.



```
A.charge=(|C,G|-3)*3;
B.charge=(|E,H|-3)*3;
f(x):=max((0,min([x,1])));
colorplot([0.1,0.1,0.1]+hue(f(abs(force(#)/3))),(-10,-10),(20,10));
drawforces(stream->true,move->0.2,color->[0,0,0],resolution->10);
```

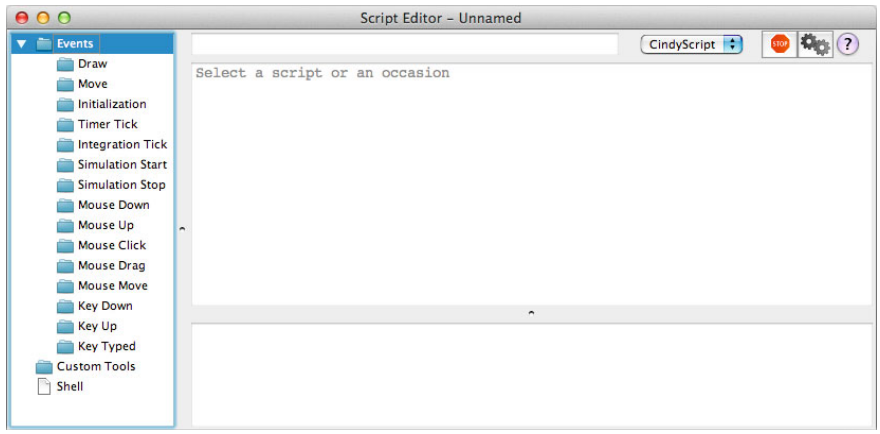
Modifiers: It is also possible to set the values of mass, charge and radius explicitly. Each of these values can be set by a modifier of the same name. If at least one of these values is set explicitly, then all unset values are set to zero. Thus `force([0,0],charge->2)` tests the force that would be present for a particle of `charge=2`, `mass=0`, and `radius=0` at point `[0,0]`.

15.7 Entering CindyScript Code

15.7.1 The CindyScript Editor

To enter CindyScript (p. 219) one can use the editor that is available from the menu *Scripting/Edit Scripts*. Here we explain briefly how to use the editor.

15.7.1.1 The Input Window



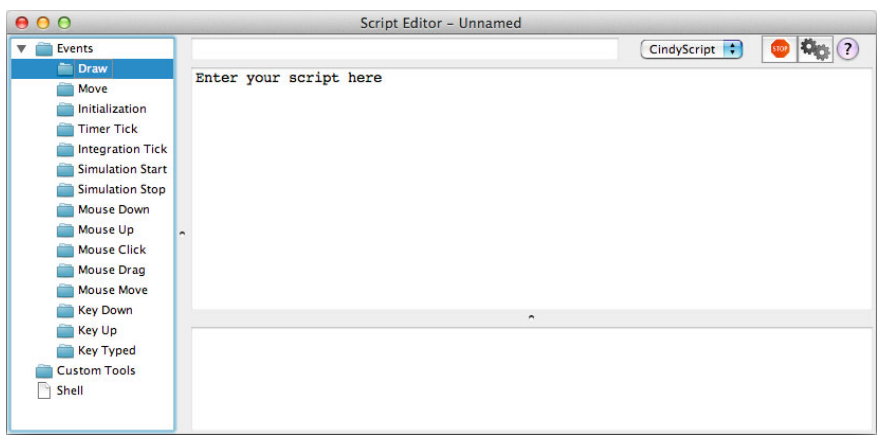
The Script Editor

The script editor shows a three pane view. On the left you see a an overview over all occasions (see below) and the associated scripts. On the right you see, below a panel that features a start, stop and help button as well as a field to enter script names, a large text area which is used to edit (i.e. type) scripts, and a smaller text area that shows any output from the scripts.

15.7.1.2 Occasions

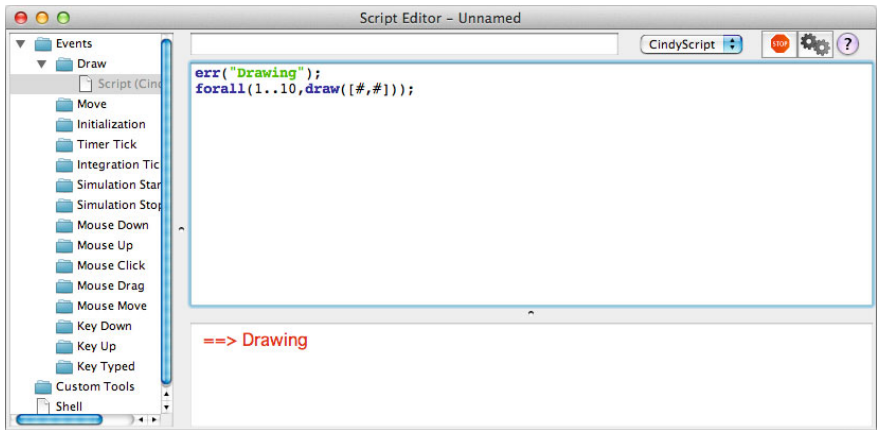
Cinderella is highly interactive, and that is the reason for many "occasions" that are suited for triggering the execution of CindyScript (p. 219) commands. On the left side of the script editor you see the available occasions.

Usually, you write scripts for the "Draw" occasion. These are executed whenever the view (p. 116) is rendered. To edit a script, first click on "Draw".

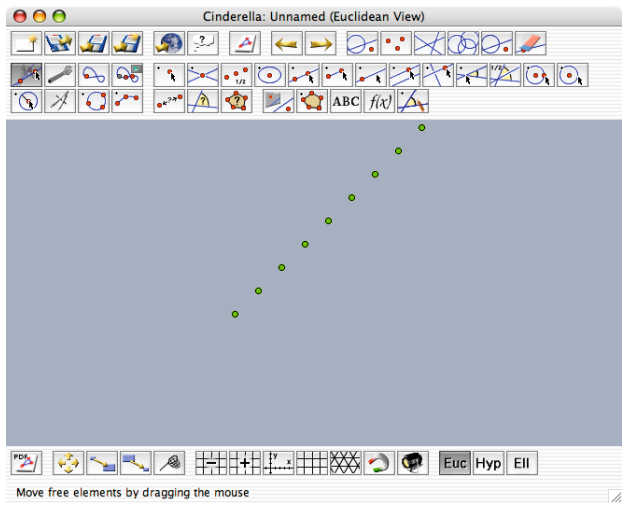


Entering scripts for the draw occasion.

The edit area will display the message "Enter your script here". Click there, and start entering text.



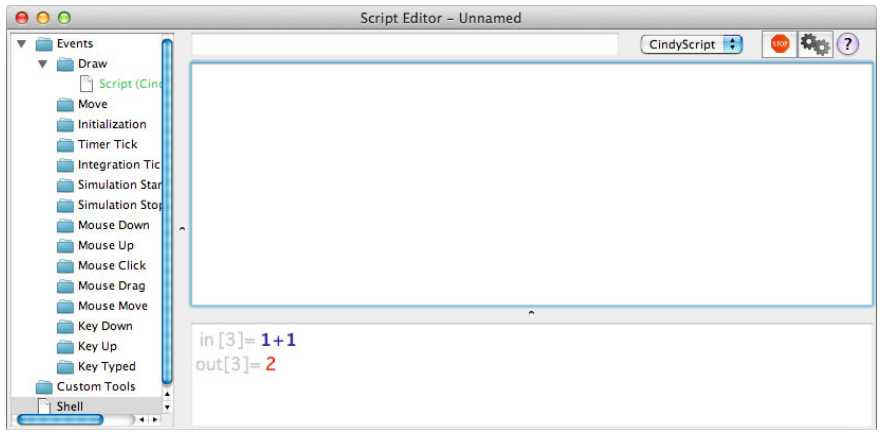
In the view you will notice a diagonal of green points that were created by the script.



You find an overview over all occasions in the introduction to CindyScript (p. 219).

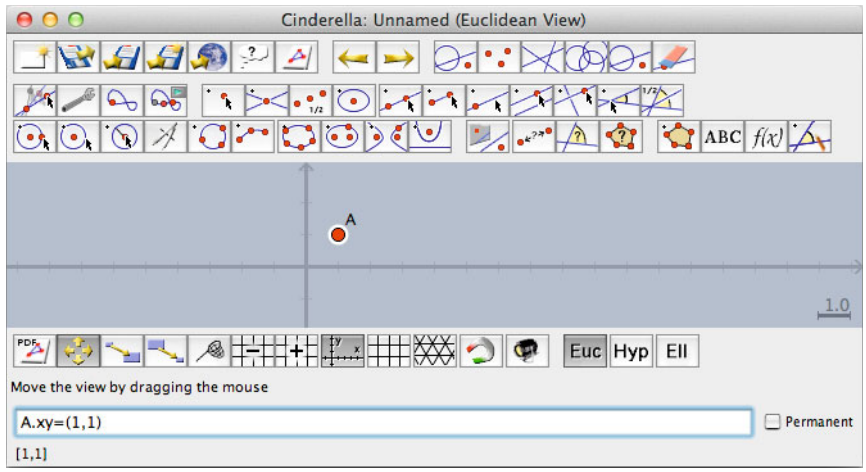
15.7.2 The Shell

You can also enter CindyScript (p. 219) commands and have them executed immediately. Just choose the “Shell” item from the left panel, and type the commands into the text area on the right. Pressing shift+enter will execute the command you typed, and you will see the in- and output in the lower text area. You can use shift-up and shift-down to scroll through a history of commands entered.



15.7.3 The Command Line

Sometimes it is very convenient to manipulate constructions using CindyScript commands. You can either do this using the shell window as described above, or, for short commands, you can use the command line integrated into the construction view. To enable it, choose "Scripting/Command Line" from the menu, or press the corresponding keyboard shortcut Ctrl-Enter (Windows and Unix) or CMD-Enter (Mac OS X).



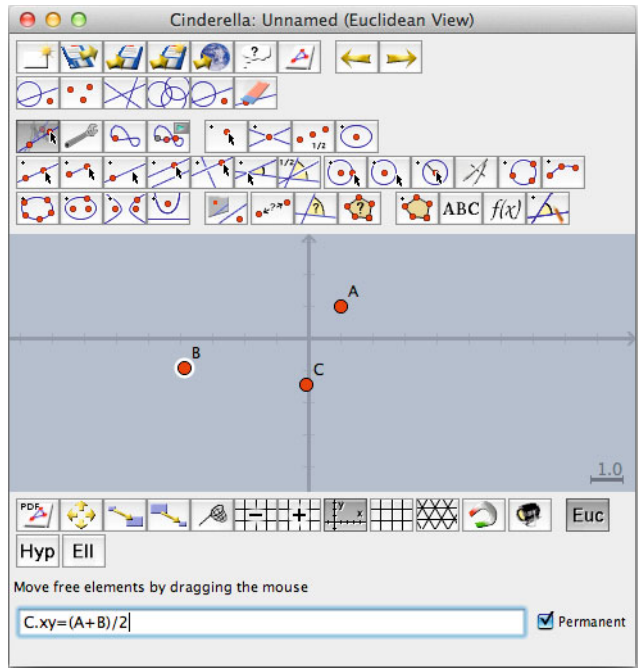
Using the command line to move a point

The command line is located below the status line. You can enter arbitrary CindyScript functions and evaluate them by pressing the enter key. If you press shift-enter, the code will be evaluated in the same way, but the command line text field will not be cleared. You can use this in case you want to issue several similar commands or if you look for the correct syntax by trial-and-error.

If you check the *permanent*-checkbox to the right of the command line, then the code will be stored in the draw occasion and thus will be evaluated whenever the

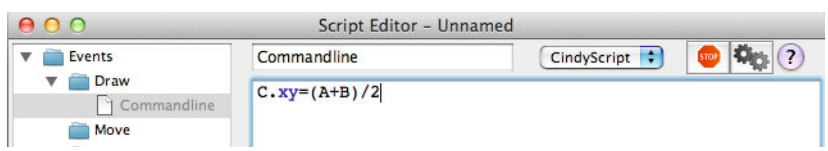
construction is redrawn. You can access the code by opening the script editor as explained above.

As a quick example we show how to move a point *C* permanently to the midpoint of two points *A* and *B*: Draw the three points, activate the command line, enter `C.xy=(A+B)/2`, and check the *permanent* checkbox.



Using the permanent checkbox

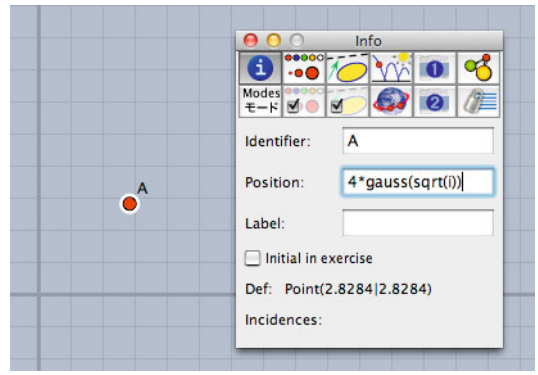
After pressing enter, *C* will move to the midpoint of *A* and *B* and stays there, even if *A* or *B* moves. Checking the Script Editor reveals the automatically generated draw occasion script named CommandLine.



The automatically generated command line script

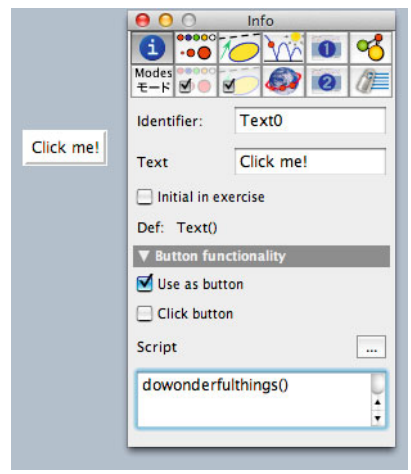
15.7.4 CindyScript and the Inspector

Many text input fields of the Inspector (p. 153) window accept CindyScript code as input. The script will only be evaluated once – if you want to make permanent changes you have to use either the command line or the draw occasion in the Script Editor. After pressing enter, you can still see your CindyScript (p. 219) code, but if the input field looses the input focus its value will be replaced with the evaluation result.



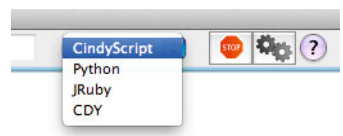
15.7.5 Clickable Buttons

Text objects can be transformed into a clickable button by checking the *Use as button* box in the inspector. You can attach CindyScript code to the button that will be executed every time the user clicks on it. As the field for script code is very small we recommend to just call functions defined in the *init* script of the construction.



15.7.6 Other Languages

You can choose the programming language used to interpret a script using the choice box in the top panel. Available languages are CindyScript (p. 219), Python, JRuby and CDY, the internal language that is used to store constructions. However, currently we only support CindyScript, and this is also the only language you can use in Cinderella applets (p. 419).



http://www.springer.com/978-3-540-34924-2

The Cinderella.2 Manual
Working with The Interactive Geometry Software
Richter-Gebert, J.; Kortenkamp, U.H.
2012, XIV, 458 p. 377 illus., 300 illus. in color.,
Hardcover
ISBN: 978-3-540-34924-2