Introduction

This program provides a native C graphs in Picat. It uses a Picat constraint solver to find exact locations and size of objects to be drawn. Drawing the individual components is done with IUP and CD, C graphics libraries created by Tecgraf.

Compilation Instructions

This program is currently only supported on 64-bit Linux systems.

The first step to compile is to download and install the IUP and CD binaries from:
http://sourveforge.net/projects/iup/files
http://sourveforge.net/projects/cd/files

These two installation can be completed by running the installation script as sudo and hitting enter when prompted.

```
sudo ./install
```

Then some header files of IUP and CD need to added to the include folder or a path to them must be added to the bash profile. The header files are iup.h, iupkey.h, and iupdef.h from IUP and cd.h and cdiup.h from CD.

Then the program can be compiled with

```
make -f Makefile.picat.linux64
```

Hello World

The compilation will create a program called picat_linux64 which can be run from the command line with

```
./picat_linux64
```

The following is a simple program to make an text object displaying “hello world”. The numbers at the end of each line are line numbers and should not be included in the program.

```
import cg.                          (1)
  main =>                           (2)
  Hello = new_text_box(),           (3)
  Hello.text = ‘hello world’,       (4)
  show(Hello).                      (5)
```

Example Program 1: Hello World

Our Hello World program starts by importing the graphics module cg in line (1). Line (2) creates a program named `main` which is defined by the code in lines (3) to (5). On line (3)

`new_text_box()` creates a new text object which is assigned to the variable `Hello`. Initially no words are written in this text box. The words `hello world` are set to the string in line (4) by using the `text` attribute of the text box object. Line (5) calls the `show()` function with the variable `Hello` to display the object in its own window.

We need to compile and run this program. We can load the program when opening Picat by adding the name of the file to the command line when opening Picat. Assuming the file name was hello.py and it is in the current directory, this can be done by:

```
./picat_linux64 hello
```

Alternatively the program can be manually compiled and run inside Picat. Once we have started running Picat, compilation is done by typing `cl(hello)` if the file name is hello.pi and it is in the current directory. To run the program we need to call on the `main` function, by simply typing `main` after the compilation finished. When hello program is displayed the following image should be displayed:

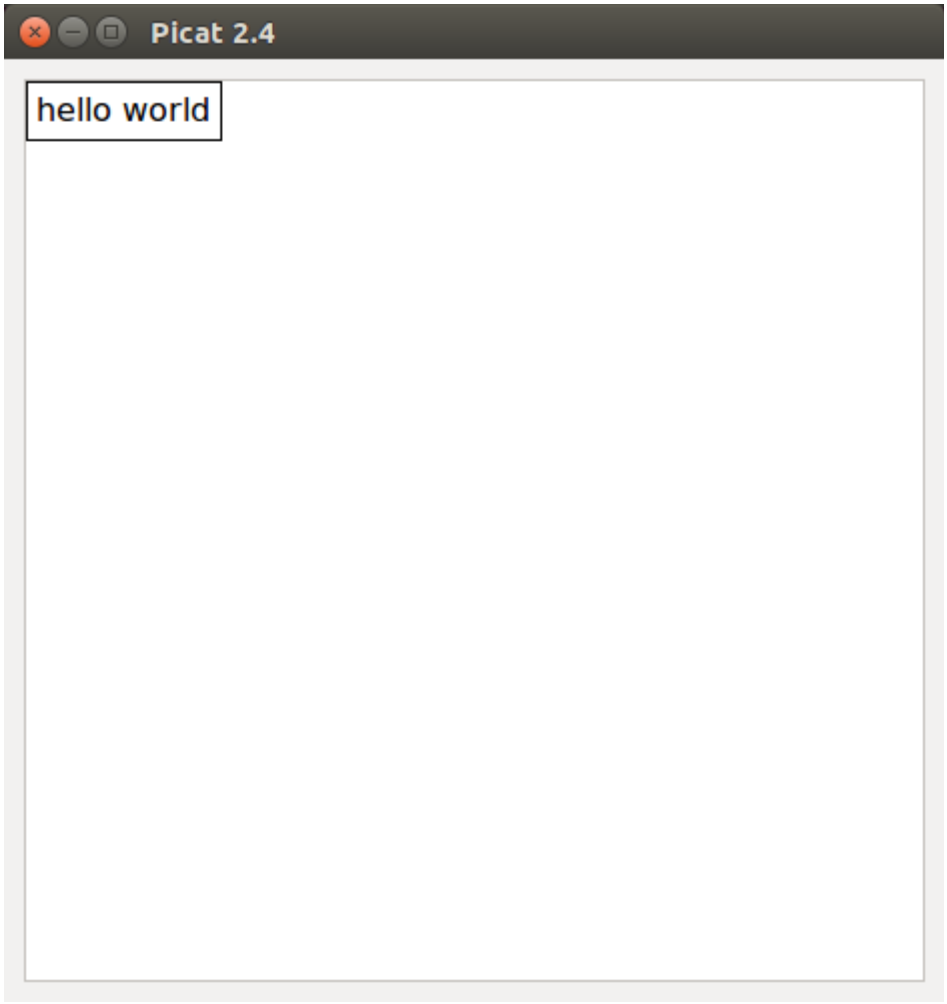

Figure 1: Hello World Program output

Usage

When creating programs using the graphical library first you need to create some objects. If no other specifications are given, all objects except polygon, triangle and line will find default values for size and location. The default location is the upper left corner, which corresponds to the origin, with an x-coordinate and y-coordinate of 0. Objects can be resized or moved by creating some constraints using their attributes. For example the `x` and `y` attributes set the horizontal and vertical starting point value of the object. Objects are drawn below the `x` value and to the right of the `y` value.

Once any desired constraints are specified, the object can be displayed with the `show(O)`,where `O` is the objects or list of objects you wish to display. Objects at the beginning of the list will be drawn first and may be covered by objects drawn after. So `show([new_circle(),new_square()])` will display a square object on top of a circle object, as the default location of the two objects overlap.

The graphical library contains functions to make the following objects:

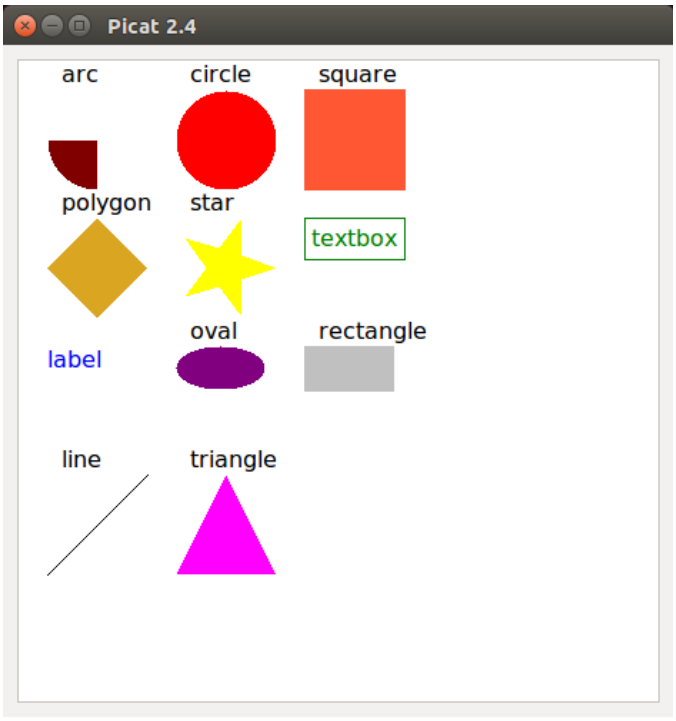

Figure 2: Examples of the displayable objects

`Arc` – The edge of part or all of a circle or a solid piece of a circle

`Circle` – A circle object with equal width and height

`Label` – A text without boarder

`Line` – A solid line connecting two points

`Oval` – A solid of hollow circular object that can have uneven width and height

`Polygon` – A solid or hollow shape connecting a number of point locations

`Rectangle` - A solid or hollow rectangular object that can have uneven width and height

`Square` - A solid or hollow rectangular object with equal width and height

`Star` — A solid or hollow star object

`Text box` — A text object with a hollow rectangular boarder

`Triangle` — A solid or hollow triangle defined by the connection of three points

Components and Attributes

All objects have some quantity of attributes that describe how they look. All attributes have some type with only certain values accepts. For example `x`, `y`, `width` and `height` are all integer values between `0` and `1000`, while the `fill` attribute of circle, rectangle, and triangle only accepts integer values of `0` or `1`. If the value of an attribute is not specified by the program, default values provided by cg will be selected and used. For example, in most objects `width` and `height` have a default of `20`, while `x` and `y` default to `0`.

Attributes can be accessed by dot notation, so an attribute `attr` of an object `O` is accessed by `O.attr`. Attributes may be assigned values by a constraint as in the `Hello.text #= 'Hello World'` of the Hello World example program.

All the objects are based around a shared component which determines some of the basic location, size, and color of the object. If specific values are not given for these attributes when `show()` is called, values will be selected. In the case of the objects of `Triangle`, `Polygon`, and `Line` the default values make all the points of the objects zero, so nothing will be displayed.

There are two type of attribute, base and derived. Base attributes are stored explicitly while derived attributes will be calculated from the stored values of the base attributes.

The following attributes are base attributes and stored explicitly:

`x` - the horizontal starting position measured in pixels from the left hand corner. Default value of `0`

`y` - the vertical starting position measured in pixels from the top of the canvas. Default value of `0`

`width` - the width of the object, so the end of the component is `x+width` pixels from the left. Default value of `20` for all objects except Triangle, Polygon, and Line

`height` - the height of the object. The end of the component will be at `y+height` pixels below the top of the canvas. Default value of `20` for all objects except Triangle, Polygon, and Line which do not have default values

`color` - the color of the object, currently the supported colors are
`dark red`, `red`, `orange`, `salmon`, `yellow`, `gold`, `lime`, `green`, `cyan`, `blue`, `purple`, `pink`, `white`, `silver`, `gray` and `black`
The default color is `black`.

Colors

The RGB values of the colors are as follows:

```
dark red -      (128,  0,  0)
red -           (255,  0,  0)
orange -        (255, 87, 51)
salmon -        (250,128,144)
yellow -        (255,255,  0)
gold -          (218,165, 32)
green -         (  0,128,  0)
lime -          (  0,255,  0)
cyan -          (  0,255,255)
blue -          (  0,  0,255)
purple -        (128,  0,128)
pink -          (255, 87, 51)
white -         (255,255,255)
silver -        (192,192,192)
gray -          (128,128,128)
black -         (  0,  0,  0)
```

Figure 3: Colors in Picat

Derived attributes are created from the base attributes by creating one or several constraints for their locations behind the scenes. For example we will look at how `centerX` is calculated. When the `centerX` attribute is accessed the system replaces the attribute with a variable and add a constraint to calculate the variable from the base attributes.

```
O.centerX #< 50,
```

Is internally changed in the following way

```
O.centerX #= V,
V #= O.x + O.width//2,
```

The following derived attributes are calculated from base attributes:

`centerX` - the center horizontal location of the component

Applies the following constraints: `O.centerX #= O.x + O.width//2,`

`centerY` - the center vertical location of the component

Applies the following constraints: `O.centerY #= O.y + O.height//2,`

`rightX` - the rightmost dimension of the component. Note there is no `leftX` value as it will have the same value as the base attribute `x`

Applies the following constraints: `O.rightX #= O.x + O.width,`

`bottomY` - the lowermost dimension of the component. There is no `topY` as the base attribute y has the same value

Applies the following constraints: `O.bottomY #= O.y + O.height,`

`leftTopPoint` - point at the upper left side

`rightTopPoint` - point at the upper right side

Applies the following constraints:

```
O.rightTopPoint #= P,
    P.x #= O.x + O.width,
    P.y #= O.y,
```

`leftBottomPoint` - point at the lower left side

Applies the following constraints:

```
O.leftBottomPoint #= P,
    P.x #= O.x,
    P.y #= O.y + O.height,
```

`rightBottomPoint` - point at the lower right side

Applies the following constraints:

```
O.rightBottomPoint #= P,
    P.x #= O.x + O.width,
    P.y #= O.y + O.height,
```

`centerPoint` - located in the center of the component

Applies the following constraints:

```
O.centerPoint #= P,
    P.x #= centerX,
    P.y #= centerY,
```

Different objects have additional attributes, such as text having a font attribute which sets the font for that object. Both component attributes and the ones specific to the object for each object can be accessed with dot notation.

Object Specific Attributes

Arc

Arc is a solid or hollow portion of a circle. A graphical object of type Arc is created with
`O = new_arc(),`

Attributes:

`fill` - Filled if `1` and hollow if `0`. Default value of `1`

`startAngle` - the starting angle of the arc in degrees. Accepts integer values between `0` and `360`. The angles lie on a coordinate system with an origin of the center of the arc

`arcAngle` - how far the arc continues in degrees. Accepts integer values between `0` and `360`. An `arcAngle` of `360` degrees will be a full circle, `180` a half circle starting at the `startAngle`

An unfilled arc with a `startAngle` of `0` and `arcAngle` of `360` will be a hollow circle.
A filled arc with a `startAngle` of `180` and an `arcAngle` of `360` will be the solid bottom half of a circle.

Circle

A graphical object of type circle is created with:
`O = new_circle(),`

Attributes:

`fill` - A solid filled-in circle if `1` and a hollow circle if `0`. Default of `1`
`diameter` - The diameter of the circle. Must be a positive integer

Creating a circle also creates an additional constraint that the `width` and `height` must be equal.

```
O.width #= O.height,
```

If this is not desired use the `Oval` object, which allows different values for `width` and `height`.

Label

A graphical object of type Label is created with:
`O = new_label(),`

Attributes:

`text` - the words to be written. Can either be a single word with no spaces such as `O.text #= hello,` or one or more words surrounded by single quotes `O.text #= 'hello world',`

font - the font of the text. Accepted values are courier, helvetica, times, or system. Defaults to system

font_size - the size of the font. Accepts integer values between 1 and 100. Defaults to size 12

font_style - the style of the font. Accepts values bold, italic or plain. Defaults to plain

Label has an internal calculation of the size of the text based on the font, font_size and font_style attributes.

align - where the text box is drawn in relation to the x and y values of the component. Possible alignment values are left, right and center. Defaults to left

Line

A graphical object of type Line is created with:
O = new_line(),

Attributes:

x1 - the x-coordinate of the starting point. Non-negative integer values accepted. Defaults to 0
x2 - the x-coordinate of the ending point. Non-negative integer values accepted. Defaults to 0
y1 - the y-coordinate of the starting point. Non-negative integer values accepted. Defaults to 0
y2 - the y-coordinate of the ending point. Non-negative integer values accepted. Defaults to 0
point1 - the first point in the line, an alternative way to assign x1 and y1
point2 - the second point in the line, an alternative way to assign x2 and y2

If no values are given for the attributes, x1, x2, y1, and y2 will all be set to 0, so no line will be drawn

x1, x2, y1 and y2 can also be accessed by the point1 and point2 attributes. point1 is [x1,y1] and point2 is [x2,yx]. As such x1 can also be accessed by point1[1]

Line has constraints to ensure that the locations of the points cannot exceed the location specified by the x, y, width and height attributes if any are assigned. For example if you make the following constraints:

```
L= new_line(),
L.x #= 0,
L.y #= 0,
L.width #=100,
L.height #= 100,
```

The location of x1, y1, x2, and y2 must all be values between 0 and 100.

This is done by additional constraints in the initiation of the object that ensure the x of the component must be less than or equal to the values for x1 and x2 and that rightX is greater

than or equal to x1 and x2. Similarly constraint are added ensuring y of the component must be less than or equal to the values of y1 and y2 and that bottomY must be greater than or equal to y1 and y2.
In lines the component attributes of x, y, width and height will not be assigned if not set by user.

Oval

A graphical object of type Oval is created with:

```
O = new_oval(),
```

Attributes:

fill - A solid filled-in circle if 1 and a hollow circle if 0. Default of 1

Polygon

A graphical object of type Polygon is created with:

```
O = new_polygon(N),
```

where N is the number of points in the polygon.

Attributes:

fill - a filled polygon if 1, a hollow polygon if 0. Default value of 1
xs - the x-coordinates of the points in the polygon. Must be in a list of length N
ys - the y-coordinates of the points in the polygon. Must be in a list of length N
If the length of xs and ys are not equal to N, the constraint solver fail and return no. If xs or ys are not set by the user, they will be set to a list of zeros of length N, therefore nothing will be displayed. The component attributes of x, y, width and height will not be assigned if not set by user in polygon objects.

Rectangle

A graphical object of type rectangle is created with:

```
O = new_rectangle(),
```

Attributes:

fill - a solid rectangle if 1, a hollow rectangle if 0. Default value of 1

Star

A new star object is created with:

```
O = new_star(N),
```

Where `N` is the number of points on the star.

Attributes:

`N` - the number of points on the star. Can only be assigned in the function `new_star(N)`
`angle0` - integer values between `0` and `360`. A value of `0` will have the first point of the star aligned with a y-axis starting at the center of the star. Other values will rotate the star so that the starting point will make an angle of `angle0` degrees from the y-axis

`fill` - a solid star if `1`, or a hollow star if `0`. Default value of `1`

Figure 4: angle0 of a five-pointed star

Square

A graphical object of type Square is created with:

```
O = new_square(),
```

Attributes:

`fill` - a solid square if `1`, a hollow square if `0`. Default value of `1`
Square enforces the constraint that the `width` and the `height` of the object must be equal. If this is not desired use rectangle.

Text Box

A graphical object of type Text Box is created with:

```
O = new_text_box(),
```

Attributes:

`align` - where the text box is drawn in relation to the `x` and `y` values of the component. Possible alignment values are `left`, `right`, and `center`. Defaults to `left`
`text` - the words to be written. Can either be a single word with no spaces such as `O.text #= hello`, or one or more words surrounded by single quotes `O.text #= 'hello world'`,

`font` - the font of the text. Accepted values are `courier`, `helvatica`, `times`, or `system`. Defaults to `system`

`font_size` - the size of the font. Accepts integer values between `1` and `100`. Defaults to size `12`

`font_style` - the style of the font. Accepts values `bold`, `italic` or `plain`. Defaults to `plain`

`padding` - the spacing between the edges of the text box and the text inside. Must be an integer between `0` and `100`. Defaults to `5`
Text Box has an internal calculation of the size of the text based on the `font`, `font_size`, `font_style` and `padding` attributes.

Triangle

A graphical object of type Triangle is created with:

```
O = new_triangle()
```

Attributes:

`fill` - a solid triangle if `1`, a hollow triangle if `0`. Defaults to `1`
`x1` - the x-coordinate of the first point of the triangle
`x2` - the x-coordinate of the second point of the triangle
`x3` - the x-coordinate of the third point of the triangle
`y1` - the y-coordinate of the first point of the triangle
`y2` - the y-coordinate of the second point of the triangle
`y3` - the y-coordinate of the third point of the triangle
`point1` - the first point in the triangle. Equivalent to [x1,y1]
`point2` - the second point in the triangle. Equivalent to [x2,y2]
`point3` - the third point in the triangle. Equivalent to [x3,y3]

Assigning triangle by points

```
import cg.
main =>
      Tri = new_triangle(),
      Sq = new_square(),
      Tri.point1 #= Sq.leftTopPoint,
```

```
Tri.point2 #= Sq.rightTopPoint,
Tri.point3 #= Sq.leftBottomPoint,
show(Tri).
```

Similar to Line and Polygon, if the points of a Triangle object are not assigned, they will be set to `0`.

Displaying Objects

Objects that have been created are shown with a call to `show()`. `show()` must be sent a single object or a list of objects ordered in the order in which they will be drawn. `show()` can be called multiple times, each time only displaying the objects sent to it. If an object has previously been shown, it will still maintain its attributes. The call to `show()` ends when the window is closed.

Constraints

The cg module uses a constraint-based solver for selecting the location of components. Specifications of the attributes enforces constraints on the solution.
So far we have only looked at simple equality constraints `#=`, however several other options exist for creating constraints. These are

`#!=`	Not equal
`#<`	Less than
`#>`	Greater than
`#<=`	Less than or equal to
`#>=`	Greater than or equal to

Grid

Grid automatically assigns locations so that the objects are laid out in a grid pattern. Only objects with their height equal to their width can be assigned with Grid. To specify the shape of the grid, use a nested lists. For example a grid with 2 columns and 3 rows containing objects O1 to O6 is assigned by:

```
grid([[O1,O2],
      [O3,O4],
      [O5,O6]]),
```

Grid also allows the user to specify a certain amount of spacing between the objects, by giving two integer values after the nested list, which must be equal to each other. The spacing is determined in pixels between each object. So the following grid would have 10 pixels added to the total width and 20 pixels added to the total height, with 10 pixels between each of the rows.

```
grid([[O1,O2],
      [O3,O4],
      [O5,O6]], 10, 10),
```

With `grid()` it is not necessary to show all the objects used in the assignment, but before grid can be called each object must be initialized with function creating the object such as `new_circle()`. This means additional components must be created to handle blank spaces between objects. If you do not wish objects displayed simply do not add them to the list that is sent to `show()`.

```
import cg.
main =>
      Show =[Show1,Show2,Show3,Show4,Show5],
      Extra = [Extra1,Extra2,Extra3,Extra4],
      foreach (I in Show)
            I = new_rectangle()
      end,
      foreach (J in Extra)
            J = new_circle()
      end,
      grid([[Show1, Extra1, Show2],
            [Extra2, Show3, Extra3],
            [Show4, Extra4, Show5]]),
      show(Show).
```

Example Program 2: Grid with blank spaces

In Example Program 2, only the objects `Show1-5` will be drawn and the pattern will look like Figure 5 shown below if no other constraints are given.

Figure 5: Grid of squares with blank spaces from Example Program 2

Tree

The `tree()` function creates visualizations of tree-like relationships. A tree is composed of nodes, each of which has an object, which is the shape of the node and a list of its children objects. The simplest possible tree is a node of some object with an empty list

```
Node = new_circle(),
node(Node, []),
```

The function signature for tree is as follows:

`tree(Tree,Type,DisX,DisY,Centered)`

`Tree` - the nodes of the tree, where each node is of the form `node(Object, Children)` where `Object` is the graphical component of the node and `Children` is a list of sub-tree in the same form as `Tree`. `Children` can have any number of nodes within it The simplest possible `Tree` is `node(Object, [])`

`Type` - determines the direction the tree is drawn. Can be one of the following:
`top_down, left_right, bottom_up, right_left`

`DisX` - the horizontal distance between each object. If `top-down` or `bottom-up` it is the distance between the nodes on a single layer. If `left-right` or `right-left`, it specifies the distance between a parent node and its children nodes

`DisY` - the vertical distance between each object. If `left-right` or `right-left` it is the distance between the nodes on a single layer. If `top-down` or `bottom-up`, it specifies the distance between a parent node and its children nodes

`Centered` – How the nodes are organized. Can be either `centered` or `itemized`. `centered` places the nodes centered below its parent node. `itemized` places adds a slight indent for each layer, so that each node on the same level has the same indent

Figure 6 – Different visualizations of tree structures

Example tree program without recursion:

```
import cg.

main =>
  node(C),
  node(L),
  node(R),
  L1 = new_line(),
  L2 = new_line(),
  Lines = [L1,L2],
  L1.point1 #= C.centerPoint,
  L2.point1 #= C.centerPoint,
  L1.point2 #= L.centerPoint,
  L2.point2 #= R.centerPoint,
  Tree = $node(C,[$node(L,[]),$node(R,[])]),
  tree(Tree,top_down,0,4,centered),
  show([C,L,R|Lines]).

node(C) =>
  C = new_circle().
```

Example Program 3: Simple Tree Program

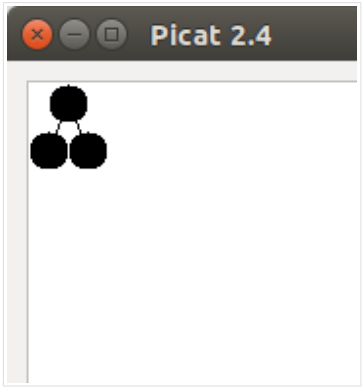

Figure 7: Output of simple tree program (Example Program 3)

Licenses

IUP and CD are under the following license: