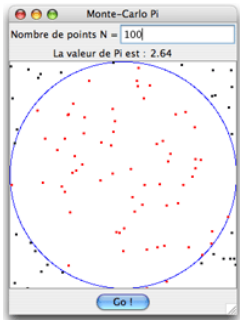




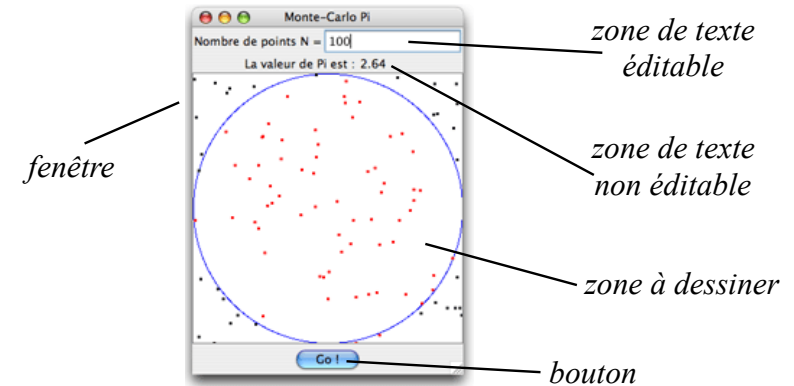
# Les Interfaces Graphiques



p. 313-318

## L'API graphique de Racket

- L'API de Racket est un ensemble de classes dont l'instanciation donne naissance à des objets qui seront les composants graphiques de l'interface.



- Nous limiterons notre étude à 3 parties de l'API :

**fenêtrage** : *windowing toolbox*. Pour obtenir des fenêtres avec des boutons, des zones de texte, de dessin, etc.

**dessin** : *drawing toolbox*. Pour dessiner avec un crayon, afficher une image gif, gérer les polices de caractères...

**éditeurs** : *editor toolbox*. Pour implémenter à moindre frais des éditeurs de texte, des navigateurs...

GUI and Graphics Libraries  
[GUI: Racket Graphics Toolkit](#)  
[Framework: Racket GUI Application Framework](#)  
  
[GL: 3-D Graphics](#)  
[PLoT: Graph Plotting](#)  
  
[Browser: Simple HTML Rendering](#)  
[Cards: Virtual Playing Cards Library](#)  
[Embedded GUI: Widgets within editor<%>](#)  
[Games: Fun Examples](#)  
[GL Board Game: 3-D Game Support](#)  
[MrLib: Extra GUI Libraries](#)  
[String Constants: GUI Internationalization](#)  
[Syntax Color: Utilities](#)  
[Turtle Graphics](#)

- Référence essentielle :

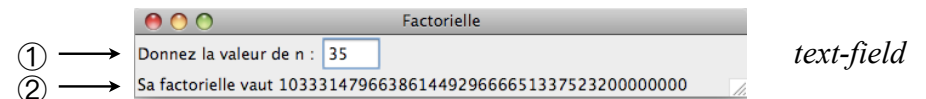
**GUI: Racket Graphics Toolkit**

dans l'aide en ligne.

## Exemple 1 : la factorielle

- Un **composant graphique** minimal, réduit à deux sous-composants :

- une **zone de texte éditable** [*text-field*] qui sera remplie par la donnée fournie par l'utilisateur.
- une **zone de texte non éditable** [*message*] mais variable, qui contiendra le résultat du calcul.



- Le tout tiendra dans une fenêtre, objet de la classe **frame%** :

```
(define FRAME (new frame% (label "Factorielle")))
```

- Les composants seront placés dans un **panneau vertical** :

```
(define VPANEL (new vertical-panel%
  (parent FRAME)
  (alignment '(left center))))
```

*left, center, right    top, center, bottom*



- On insère les composants [dans l'ordre] dans ce panneau. D'abord une **zone de texte éditable** avec un *prompt* :

```
(define TEXT-FIELD (new text-field%
  (parent VPANEL)
  (label "Donnez la valeur de n : ")
  (min-width 200)
  (init-value "0")
  (stretchable-width #f)
  .....))
```

## Le mécanisme de **callback**

- Un **callback** est une réponse à un évènement. C'est une fonction (lambda (obj evt) ...) automatiquement activée par cet évènement.
- Elle prend en arguments l'objet text-field et l'évènement. Un évènement de text-field est un objet de la classe **control-event%** :

```
(define TEXT-FIELD (new text-field%
  (parent VPANEL)
  (label "Donnez la valeur de n : ")
  (init-value "0")
  (callback (lambda (obj evt)
    ...))))
```



```
(callback (lambda (obj evt)
  (when (equal? (send evt get-event-type) 'text-field-enter)
    (let ((n (string->number (send TEXT-FIELD get-value))))
      (send MSG set-label
        (format "Sa factorielle vaut ~a" (fac n)))))))
```

- Puis une **zone de texte non éditable**, qui sera mise à jour par le programme :

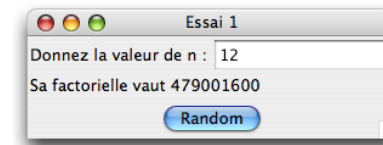
```
(define MSG (new message% (parent VPANEL)
  (label "1")
  (min-width 500)))
```

## L'interface graphique doit **REAGIR AUX EVENEMENTS** !

- Le plus important : je modifie le contenu du text-field et j'appuie sur *Entrée*. Que va-t-il se passer ?...
- Le text-field doit **réagir en effectuant une action**. Il s'agit d'un mécanisme de **callback**.

## Exemple 2 : ajouter un bouton

- Ajoutons un **bouton** dans notre interface graphique. Sa pression générera un entier aléatoire dans le text-field et mettra à jour le résultat du calcul.

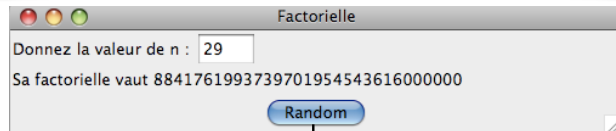


- Pour centrer le bouton, nous l'insérons seul dans un panneau horizontal centré !

```
(define HPANEL (new horizontal-panel%
  (parent VPANEL)
  (alignment '(center center))))
```

- Et nous retrouvons l'inévitable *callback* puisqu'il doit y avoir une réaction en réponse au clic dans le bouton :

```
(define BOUTON
  (new button%
    (parent HPANEL)
    (label "Random")
    ; (style '(border)) ; ne réagit pas à la pression sur Enter
    (callback (lambda (obj evt) ; ici obj et evt ne sont pas utilisés !
      (define n (+ 2 (random 34))) ; n ∈ [2,35]
      (send TEXT-FIELD set-value (number->string n))
      (send MSG set-label
        (format "Sa factorielle vaut ~a" (fac n)))))))
```



```
(send FRAME show #t)
```

Clic ! → *callback* !

- Il y aura clairement 3 composants en tout, à la verticale.

```
(define FRAME
  (new frame% (label "Editor toolbox")))

(define VPANEL (new vertical-panel% (parent FRAME)))
```

- Les deux premiers composants sont des **éditeurs de texte**. Un tel éditeur est constitué :

- d'un *écrivain*, chargé des écritures, instance de `text%`

```
(define TEXT1 (new text%))
```

- d'une *feuille à écrire*, objet de la classe `editor-canvas%`

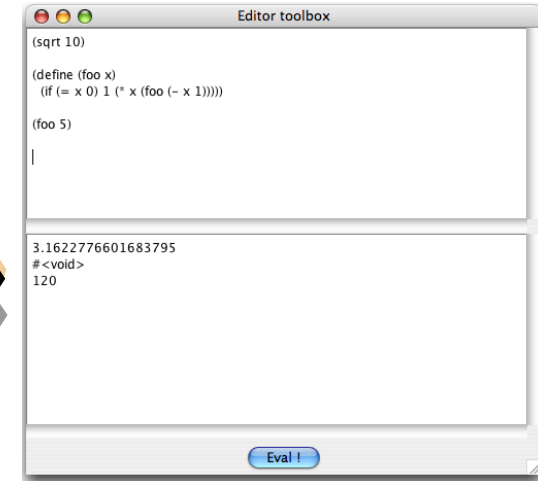
```
(define ECANVAS1
  (new editor-canvas%
    (parent vpanel) (editor TEXT1)
    (min-width 400) (min-height 200)))
```

*Idem pour le  
second éditeur !*

## Exemple 3 : les éditeurs de texte

- Proposons-nous de simuler un mini-micro-DrRacket avec ses deux fenêtres : un éditeur de code Scheme en haut, et les résultats des évaluations en bas !

eval



1

2

- Le troisième composant est un bouton. Sa pression provoque l'évaluation en séquence de toutes les expressions de l'éditeur n°1, avec envoi des résultats dans l'éditeur n°2...

```
(define BUTTON-EVAL
  (new button%
    (parent VPANEL)
    (label "Eval !")
    (callback
      (lambda (obj evt) ; obj = le bouton
        (send TEXT2 erase)
        (define L (read-from-string-all (send TEXT1 get-text)))
        (for-each (lambda (expr)
          (send TEXT2 insert (expr->string (eval expr)))
          (send TEXT2 insert "\n"))
          L))))))
```

*Cool !*

```
> (read-from-string "123 (a b c) (sqrt 3)")
123
> (read-from-string-all "123 (a b c) (sqrt 3)")
(123 (a b c) (sqrt 3))
```



## Exemple 4 : dessiner des polygones...

- Un **point** est une structure de type **posn**, à importer :

```
(require lang/posn) ↔ (define-struct posn (x y) #:transparent #:mutable)
```

- Un **polygone** sera une liste de points. Voici un constructeur de polygone sur le modèle de build-list :

```
(define (build-polygon n f g) ; n sommets ((f i) (g i))  
  (build-list n (lambda (i) (make-posn (f i) (g i)))))
```

```
> (build-polygon 5 (lambda (x) x) sqr)  
#(struct:posn 0 0) #(struct:posn 1 1) #(struct:posn 2 4)  
#(struct:posn 3 9) #(struct:posn 4 16))  
  
> (build-polygon 6 (lambda (i) (* 30 (+ i 1))) (lambda (i) (random 10)))  
#(struct:posn 30 7) #(struct:posn 60 4) #(struct:posn 90 4)  
#(struct:posn 120 9) #(struct:posn 150 7) #(struct:posn 180 7))
```

```
(define FRAME  
  (new frame% (label "Polygon")  
             (min-width 300)  
             (min-height 300)  
             (stretchable-width #f)  
             (stretchable-height #f)))
```

- La fonction de **callback** se nomme cette fois **paint-callback** et reçoit en argument le canvas obj et son peintre dc :

```
(define CANVAS  
  (new canvas%  
    (parent frame)  
    (paint-callback (lambda (obj dc)  
                     (draw-polygon POLY dc)))))
```

- La fonction de **paint-callback** est appelée par la méthode **on-paint**, qui est automatiquement invoquée par le système dès qu'un évènement nécessite de redessiner la fenêtre [recouvrement, etc].

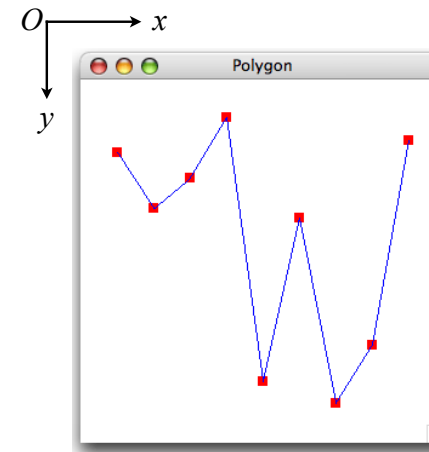
- Pour dessiner, il faudra une **toile** [**canvas%**] et un **peintre** [**dc%**]. Le **peintre** va utiliser des **crayons de couleurs** [**pen%**] :

```
(define PEN-FOR-POINTS (new pen% (color "red") (width 8)))  
(define PEN-FOR-LINES (new pen% (color "blue") (width 1)))
```

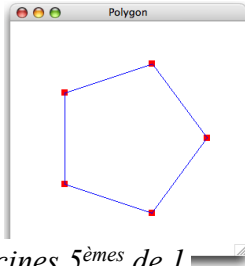
- Une fonction de dessin d'un polygone L par un peintre dc :

```
(define (draw-polygon L dc) ; dessin du polygone L = (posn ...)  
  (define (draw-vertices L) ; par un peintre dc  
    (send dc set-pen PEN-FOR-POINTS)  
    (for ([pt (in-list L)])  
      (send dc draw-point (posn-x pt) (posn-y pt))))  
  (define (draw-edges L)  
    (send dc set-pen PEN-FOR-LINES)  
    (while (not (null? (cdr L)))  
      (send dc draw-line (posn-x (car L)) (posn-y (car L))  
                       (posn-x (cadr L)) (posn-y (cadr L))))  
    (set! L (cdr L)))  
  (send dc clear)  
  (when (not (null? L))  
    (draw-vertices L)  
    (draw-edges L)))
```

```
(define POLY (build-polygon 9 (lambda (i) (* 30 (+ i 1)))  
                          (lambda (i) (random 300))))  
(send FRAME show #t)
```

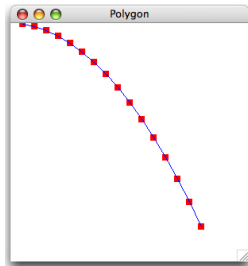


```
(define POLY
  (let ((a (* 2/5 pi)))
    (build-polygon 6 ; 6 pour le refermer !
      (lambda (i) (+ 150 (* 100 (cos (* i a))))))
      (lambda (i) (+ 150 (* 100 (sin (* i a)))))))))
```



Le polygone régulier des racines 5<sup>èmes</sup> de 1

```
(define POLY
  (build-polygon 16
    (lambda (i) (* 15 i))
    (lambda (i) (* i i))))
```



La chute parabolique...

- Sur pression du bouton gauche, on doit savoir si l'on est près d'un sommet V du polygone. Sinon, V vaut #f :

```
(define V #f)

(define MY-CANVAS%
  (class canvas% ; sous-classe de canvas%
    (define/override (on-event evt)
      (case (send evt get-event-type)
        ((left-down) (define x (send evt get-x))
                      (define y (send evt get-y))
                      (set! V (pt-proche x y POLY)))
        ((left-up) (set! V #f))
        (else (when V
                 (set-posn-x! V (send evt get-x))
                 (set-posn-y! V (send evt get-y))
                 (send this on-paint)))))) ; callback !
    (super-new)))
```

## Comment faire bouger les sommets à la souris ?

- La manipulation d'une souris génère aussi des événements, objets de la classe `mouse-event%` :
  - pression du bouton gauche : `left-down`
  - relâchement du bouton gauche : `left-up`
  - etc.
- Sur un tel événement, le système invoque automatiquement la méthode `on-event` de la classe `canvas%`. On va donc *redéfinir cette méthode dans une sous-classe* :

```
(define MY-CANVAS%
  (class canvas%
    (define/override (on-event evt)
      ...)
    (super-new)))
```

- Et l'on définira le canvas de travail comme une instance de la classe `MY-CANVAS%` avec le même `paint-callback`...

- La fonction `(pt-proche x y L)` retourne le point du polygone L assez proche de `<x,y>`, ou bien #f s'il n'y en a pas à proximité :

N.B.  $(x_1, y_1)$  est assez proche de  $(x_2, y_2) \Leftrightarrow |x_1 - x_2| + |y_1 - y_2| < 6$

```
(define (pt-proche x y L)
  (define (assez-proche? x1 y1 x2 y2)
    (< (+ (abs (- x1 x2)) (abs (- y1 y2))) 6))
  (cond ((null? L) #f)
        ((assez-proche? x y (posn-x (car L)) (posn-y (car L))) (car L))
        (else (pt-proche x y (cdr L)))))
```

```
> POLY
#(struct:posn 30 258)
#(struct:posn 60 2)
#(struct:posn 90 139)
#(struct:posn 111 235)
#(struct:posn 169 69)
#(struct:posn 202 207)
#(struct:posn 230 145)
#(struct:posn 240 76)
#(struct:posn 270 190))

> (pt-proche 88 135 POLY)
#f
> (pt-proche 88 137 POLY)
#(struct:posn 90 139)
```

- Sur pression du bouton gauche, on doit savoir si l'on est près d'un sommet V du polygone. Sinon, V vaut #f :

```
(define V #f)

(define MY-CANVAS%
  (class canvas% ; sous-classe de canvas%
    (define/override (on-event evt)
      (case (send evt get-event-type)
        ((left-down) (define x (send evt get-x))
                      (define y (send evt get-y))
                      (set! V (pt-proche x y POLY)))
        ((left-up) (set! V #f))
        (else (when V
                 (set-posn-x! V (send evt get-x))
                 (set-posn-y! V (send evt get-y))
                 (send this on-paint)))))) ; callback !
    (super-new)))
```

- La fonction `(pt-proche x y L)` retourne le point du polygone L assez proche de `<x,y>`, ou bien #f s'il n'y en a pas à proximité :

N.B.  $(x_1, y_1)$  est assez proche de  $(x_2, y_2) \Leftrightarrow |x_1 - x_2| + |y_1 - y_2| < 6$

```
(define (pt-proche x y L)
  (define (assez-proche? x1 y1 x2 y2)
    (< (+ (abs (- x1 x2)) (abs (- y1 y2))) 6))
  (cond ((null? L) #f)
        ((assez-proche? x y (posn-x (car L)) (posn-y (car L))) (car L))
        (else (pt-proche x y (cdr L)))))
```

```
> POLY
#(struct:posn 30 258)
#(struct:posn 60 2)
#(struct:posn 90 139)
#(struct:posn 111 235)
#(struct:posn 169 69)
#(struct:posn 202 207)
#(struct:posn 230 145)
#(struct:posn 240 76)
#(struct:posn 270 190))

> (pt-proche 88 135 POLY)
#f
> (pt-proche 88 137 POLY)
#(struct:posn 90 139)
```

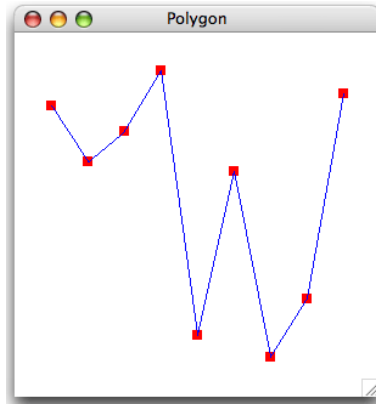
- Il ne reste alors qu'à définir le canvas de travail :

```
(define MY-CANVAS
  (new MY-CANVAS%
    (parent FRAME)
    (paint-callback (lambda (obj dc)
                      (draw-polygon POLY dc))))))
```

- et à ouvrir l'interface graphique :

```
(send FRAME show #t)
```

- Il s'agit bien d'un **éditeur**. Si je bouge un sommet à la souris, la variable POLY est automatiquement modifiée ! Je peux la récupérer au toplevel...



## La classe canvas%

```
(define a-canvas (new canvas% (parent container)
  (style '(border hscroll vscroll))
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)
  (paint-callback (lambda (c dc) ...))))
```

```
(send a-canvas get-dc)
(send a-canvas refresh)
```

↓  
dc%

- Pour gérer la souris, on programme une sous-classe de canvas% dans laquelle on redéfinit la méthode **on-event**. Puis on instancie un objet de cette sous-classe, avec un *paint-callback*.

## La classe frame%

```
(define a-frame (new frame% (label string)
  (x integer) (y integer)
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)))
```

```
(send a-frame show boolean)
(send a-frame set-label string)
```

## La classe horizontal-panel%

```
(define h-panel (new horizontal-panel% (parent frame-or-panel)
  (alignment '(l-c-r t-c-b))))
```

## La classe vertical-panel%

```
(define v-panel (new vertical-panel% (parent frame-or-panel)
  (alignment '(l-c-r t-c-b))))
```

## La classe dc%

- Un objet de dc% est le *device context* [peintre !] associé un canvas.

```
(send a-dc clear)
(send a-dc draw-point x y)
(send a-dc draw-line x1 y1 x2 y2)
(send a-dc draw-ellipse x y width height)
(send a-dc draw-rectangle x y width height)
(send a-dc draw-text string x y)
(send a-dc set-pen pen)
(send a-dc set-brush brush)
(send a-dc set-background color)
```

```
(define a-color (make-object color% color-string))
(define a-pen (make-object pen% color-string width style))
(define a-brush (make-object brush% color-string style))
```

voir color-database<%>

'solid  
'short-dash  
...

'solid  
'cross-hatch  
...



## La classe `button%`

```
(define a-button (new button% (parent container)
  (label string)
  (enabled boolean)
  (style '(...))
  (callback (lambda (b ctrl-evt) ...))))
```

```
(send a-button enable boolean)
(send a-button set-label string)
```

---

## La classe `message%`

```
(define a-msg (new message% (parent container)
  (label string)
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)))
```

```
(send a-msg get-label)
(send a-msg set-label string)
```

## La classe `radio-box%`

```
(define a-radio (new radio-box% (parent container)
  (label string) (choices string-list)
  (style '(vertical-or-horizontal))
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)
  (callback (lambda (rb ctrl-evt) ...))))
```

```
(send a-radiobox get-selection) → integer ≥ 0
(send a-radiobox enable boolean)
```

- Un seul bouton est enfoncé !

---

## La classe `gauge%`

••••••••••

## La classe `text-field%`

```
(define a-textfield (new text-field% (parent container)
  (label string) (init-value string)
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)
  (callback (lambda (tf ctrl-evt) ...))))
```

```
(send a-textfield get-value) → string
(send a-textfield set-value string)
```

- On vérifie en principe que l'évènement passé au `callback` est de type `text-field-enter` [pression sur *Entrée*] :

```
(if (equal? (send ctrl-evt get-event-type) 'text-field-enter)
    ...)
```

## La classe `slider%`

```
(define a-slider (new slider% (parent container)
  (label string) (enable boolean)
  (style '(vertical-or-horizontal))
  (min-value integer) (max-value integer)
  (init-value integer)
  (min-width integer) (min-height integer)
  (stretchable-width boolean)
  (stretchable-height boolean)
  (callback (lambda (s ctrl-evt) ...))))
```

```
(send a-slider get-value) → integer
(send a-slider enable boolean)
```

---

## La classe `check-box%`

••••••••••

## La classe `editor-canvas%`

```
(define an-ecanvas (new editor-canvas% (parent container)  
  (style '(...))  
  (min-width integer) (min-height integer)  
  (stretchable-width boolean)  
  (stretchable-height boolean)))
```

```
(send an-ecanvas set-editor text)
```

---

## La classe `text%`

```
(define a-text (new text%))
```

```
(send a-text erase)
```

```
(send a-text insert string)
```

```
(send a-text get-text) → string
```

```
(send a-text change-style style-delta)
```

```
(send a-text hide-caret boolean)
```

```
(send a-text lock boolean)
```