Les Evénements

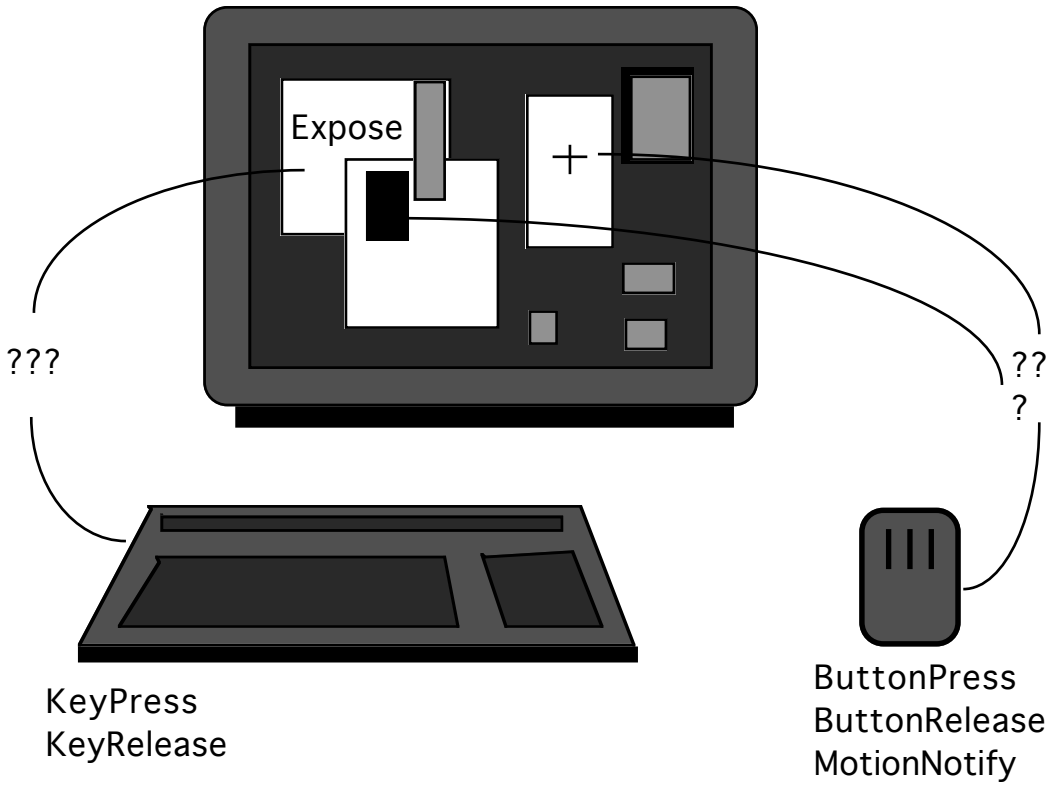

fig11. *A quelles fenêtres et quelles applications sont destinés les événements survenant sur le clavier, la souris ?*

Partage des événements

Les événements sont des structures utilisées par le serveur pour communiquer des informations aux différents clients. Il existe trois grandes catégories d'événements:

- les événements résultant d'actions de l'utilisateur sur les dispositifs d'entrée (souris, clavier).
- les événements résultant de modifications des structures internes du serveur (table de couleurs, tables des clés) ou indiquant des modifications dans la géométrie ou la visibilité des fenêtres (besoins de réaffichage).
- les événements permettant la communication entre clients. En particulier toute une série d'événements transmettent les requêtes des clients au window manager.

Les événements les plus importants sont ceux qui sont engendrés par une action de l'utilisateur sur les dispositifs d'entrées et ceux qui concernent l'affichage. On se familiarisera donc tout particulièrement avec les événements suivants:

Clavier:	Souris:	Fenêtre:
KeyPress	**ButtonPress**	**Expose**
KeyRelease	**ButtonRelease**	**ConfigureNotify**
	MotionNotify	

Focus Clavier:	Focus Souris:
FocusIn	**EnterNotify**
FocusOut	**LeaveNotify**


Le partage des dispositifs d'entrées (clavier, souris) et de sortie (écran) s'effectue selon le principe suivant: à un instant donné une seule fenêtre reçoit les événements du clavier (on dit qu'elle a le **focus**) et une seule fenêtre (pas nécessairement la même) reçoit les événements de souris. Par défaut, la fenêtre dans laquelle se trouve la souris a le focus du clavier et reçoit les événements souris, sauf si des mécanismes d'appropriation du clavier ou de la souris ont été mis en oeuvre.

Sélection et réception des événements

Par défaut, un client ne reçoit aucun événement. Pour recevoir un événement, il faut bien sûr chercher à le récupérer. La récupération des événements envoyés par le serveur s'effectue au moyen de la fonction **XNextEvent**(dpy, &event).

Mais en outre, la plupart des événements ne sont envoyés à l'application que si celle-ci en a indiqué le désir au moyen d'une opération appelée sélection. La sélection des événements doit être effectuée par l'application au niveau de chaque fenêtre - soit au moyen de la fonction **XSelectInput**(dpy, win, event_mask), soit par une modification directe de l'attribut *event_mask* de la fenêtre concernée (cf. XCreateWindow ou XChangeWindowAttributes). En quelque sorte, l'application informe le serveur qu'elle souhaite être informée de l'occurence de certains types d'événement sur une fenêtre donnée. Différents masques permettent de spécifier le type des événements intéressant une fenêtre particulière.

Les fenêtres doivent être affichées pour être susceptibles de recevoir des événements. On sélectionne cependant les événements *avant* que d'afficher les fenêtres pour garantir que le serveur n'ignore pas d'événement concernant la fenêtre.

La structure générale d'un programme d'interface sous X-Window sera donc toujours plus ou moins celle-ci:

```
/* ouverture de la connexion et initialisations diverses */
     dpy = XOpenDisplay(NULL);
/* Création de fenêtres  */
     win = XCreateSimpleWindow(…);
/* sélection d'événements associés  */
     XSelectInput(dpy,win,ButtonPressMask | ExposureMask);
/* puis affichage de la fenêtre  */
     XMapWindow(dpy, win);
/* enfin boucle de contrôle destinée à recevoir les événements */
     events();
```

La boucle de contrôle destinée à recevoir les événements est en général une boucle infinie (on prévoit cependant un cas de sortie sur une action de l'utilisateur) et a l'allure suivante:

```
events() {
XEvent ev;
     for (  ;  ;  )  {
          XNextEvent(dpy, &ev);
          switch (ev.type) {
               case ButtonPress:
                         if (ev.xbutton.button == Button1)
                              /* traitement */ ...
                         break ;
```

```
                case Expose:
                        width = ev.xexpose.width;
                        height = ev.xexpose.height;
                        /* traitement */
                        break ;
        }
    }
}
```

Propagation des événements

Le champ window de la plupart des structures d'événements contient la fenêtre à laquelle est rapporté l'événement (event window) qui est également la fenêtre dans laquelle s'est produit l'événement (source window). Cependant pour les événements concernant les dispositifs d'entrées (ButtonPress, ButtonRelease, KeyPress, KeyRelease, MotionNotify) la fenêtre à laquelle l'événement est rapporté *n'est pas nécessairement* la fenêtre source dans laquelle l'événement s'est produit.

La fenêtre à laquelle est rapporté un événement dépend du résultat de la propagation de l'événement dans toute la hiérarchie ascendante des fenêtres et est déterminée par les valeurs des attributs **event_mask** et **do_not_propagate_mask** de toutes les fenêtres ancêtres. La fenêtre source est la fenêtre contenant le pointeur (la plus basse dans la hiérarchie). Si cette fenêtre a sélectionné l'événement (champ event_mask de la structure d'attributs), l'événement lui sera transmis; sinon l'événement sera transmis à son ascendant direct, ascendant pour lequel on testera à nouveau le champ *event_mask*. Et ainsi de suite, jusqu'à trouver une fenêtre intéressée par l'événement.

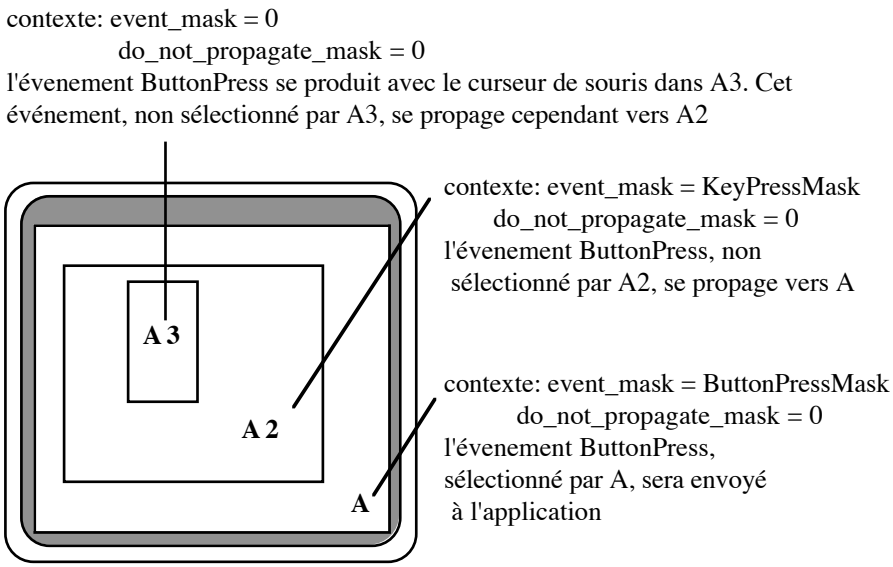

fig12a. *La propagation d'un événement ButtonPress*

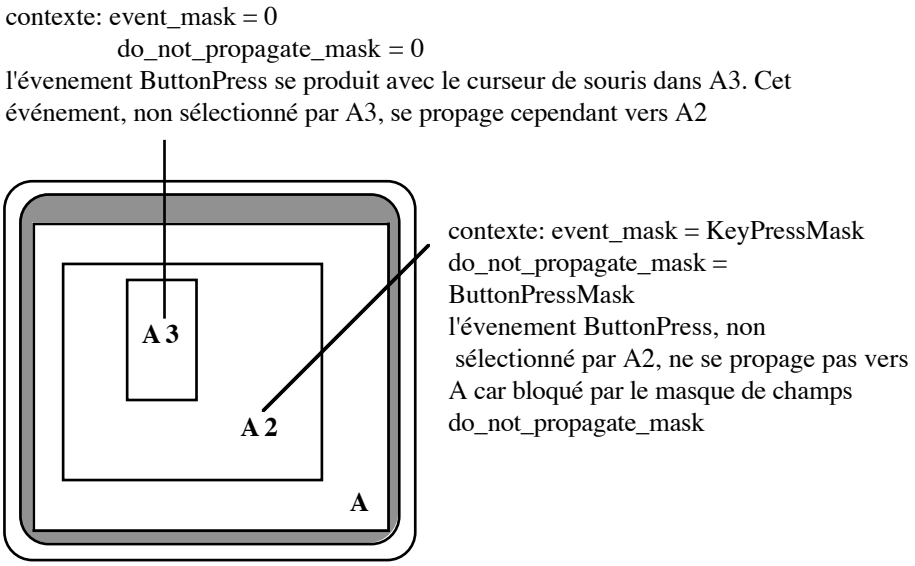

fig12b. *La non propagation d'un événement ButtonPress*

Cependant, ce processus de propagation est interrompu dès qu'une fenêtre aura masqué l'événement dans l'attribut *do_not_propagate_mask* - voir figure ci-dessus - auquel cas l'événement bloqué est perdu.

La pile des événements

Tout comme les requêtes, les événements sont empilés. En premier lieu au niveau du serveur, qui détermine vers quels clients il doit rediriger ces événements, puis localement dans la Xlib, au niveau de chaque client (cf. figure 13). Ainsi la fonction **XNextEvent** peut être bloquante car elle retourne le sommet de la pile d'événements et attend qu'un événement se produise si cette dernière est vide.

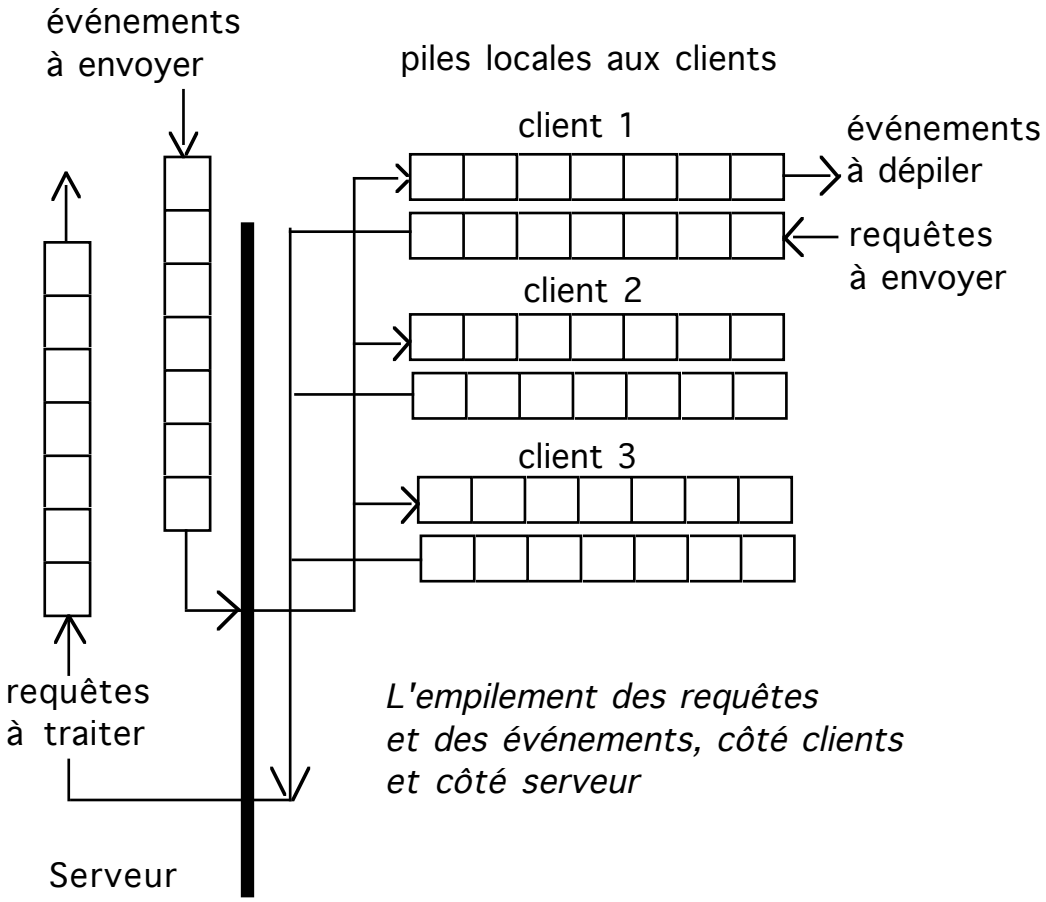

fig13. *L'empilement des événements et des requêtes*

On dispose de toute une série de fonctions permettant de regarder ou d'extraire des événements de la pile. Dans la pratique, on utilisera cependant presque toujours XNextEvent pour récupérer les événements. Indiquons cependant que **XPending** et **XEventsQueued** permettent de retourner le nombre d'événements en attente de traitement. De même il existe une fonction permettant de regarder sans l'extraire le prochain événement (avec attente): **XPeekEvent**. On peut

également regarder, avec ou sans attente, si la pile contient des événements particuliers, non nécessairement en tête de file, grâce aux fonctions suivantes:

XIfEvent	(par prédicat)
XCheckIfEvent	
XWindowEvent	(par fenêtre)
XCheckWindowEvent	
XMaskEvent	(par masque d'événement)
XCheckMaskEvent	
XCheckTypeEvent	(par type d'événement)
XCheckTypeWindowEvent	(par type et fenêtre)

Il existe également une fonction permettant de remettre un événement dans la queue: **XPutBackEvent**.

Types et structures associées aux événements

Il y a 33 noms ou types d'événements différents et 31 types effectifs de structures C associées. Le type **XEvent** est en réalité une union (en C) de ces 31 types.

```
typedef union _XEvent {
           int type;
           XAnyEvent               xany;
           XKeyEvent               xkey;
           XButtonEvent            xbutton;
           XMotionEvent            xmotion;
           XCrossingEvent          xcrossing;
           XFocusChangeEvent       xfocus;
           XExposeEvent            xexpose;
           XGraphicsExposeEvent    xgraphicsexpose;
           XNoExposeEvent          xnoexpose;
           XVisibilityEvent        xvisibility;
           XCreateWindowEvent      xcreatewindow;
           XDestroyWindowEvent     xdestroywinodw;
           XUnmapEvent             xunmap;
           XMapEvent               xmap;
           XMapRequestEvent        xmaprequest;
           XReparentEvent          xreparent;
           XConfigureEvent         xconfigure;
           XGravityEvent           xgravity;
           XResizeRequestEvent     xresizerequest;
           XConfigureRequestEvent  xconfigurerequest;
           XCirculateEvent         xcirculate;
```

```
                XCirculateRequestEvent   xcirculaterequest;
                XPropertyEvent           xproperty;
                XSelectionClearEvent     xselectionclear;
                XSelectionRequestEvent   xselectionrequest;
                XSelectionEvent          xselection;
                XColormapEvent           xcolormap;
                XClientMessageEvent      xclient;
                XMappingEvent            xmapping;
                XXErrorEvent             xerror;
                XKeymapEvent             xkeymap;
    }  XEvent ;
```

fig14. L'union de structures XEvent

A chaque type d'événement correspond une structure C dont les champs contiennent les informations relatives à l'événement de ce type. Les noms des types de structures C coïncident pratiquement avec les noms des types d'événements. Ainsi au type 'Name' d'événement correspond a priori une structure C de type 'XNameEvent' dont le champ d'accès dans l'union XEvent s'appelle 'xname'. Ainsi par exemple à l'événement de type **Expose** correspond une structure de type **XExposeEvent** adressable via le champ **xexpose** d'un événement de type générique XEvent. L'allure générale de la boucle d'événement sera donc la suivante:

```
events() {
XEvent ev;
     for (  ;  ;  )  {
          XNextEvent(dpy, &ev);
          switch (ev.type) {
               case Expose:        pos_x = ev.xexpose.x;
                                   pos_y = ev.xexpose.y;
                                   /* traitement */
               case ButtonPress:   pos_x = ev.xbutton.x;
                                   pos_y = ev.xbutton.y;
                                   /* traitement */
               ...
          }
     }
}
```

Cependant, il n'y en a que 31 noms de types de structures, car certains types différents d'événements peuvent être décrits par une même structure C. Ainsi la structure permettant d'accéder aux informations relatives aux événements de type ButtonPress et ButtonRelease est une structure C de type **XButtonEvent**

adressable par le champ **xbutton** d'un événement générique XEvent. De même à KeyPressed et KeyRelease est associée une structure de type **XKeyEvent** adressable via le champ **xkey**. Même chose pour les événements concernant le focus du clavier: à FocusIn et FocusOut correspond une structure de type **XFocusChangeEvent** adressable via le champ **xfocus**. De même les événements d'entrée et sortie de la souris EnterNotify et LeaveNotify sont décrits par une structure de type **XCrossingEvent** adressable par le champ **xcrossing.** On a donc a priori 4 types de structures de moins correspondant aux quatre cas énumérés précédemment. Cependant, il y a deux autres types de structures associées aux événements utilisés par la Xlib: le type **XErrorEvent** et le type **XAnyEvent**. D'où le bilan final: 33 types particuliers d'événements, moins 4 cas regroupés (pour cause de similarité), plus 2 cas particuliers, égale 31 types de structures associées.

```
#define KeyPress            2
#define KeyRelease          3
#define ButtonPress         4
#define ButtonRelease       5
#define MotionNotify        6
#define EnterNotify         7
#define LeaveNotify         8
#define FocusIn             9
#define FocusOut            10
#define KeymapNotify        11
#define Expose              12
#define GraphicsExpose      13
#define NoExpose            14
#define VisibilityNotify    15
#define CreateNotify        16
#define DestroyNotify       17
#define UnmapNotify         18
#define MapNotify           19
#define MapRequest          20
#define ReparentNotify      21
#define ConfigureNotify     22
#define ConfigureRequest    23
#define GravityNotify       24
#define ResizeRequest       25
#define CirculateNotify     26
#define CirculateRequest    27
#define PropertyNotify      28
#define SelectionClear      29
#define SelectionRequest    30
#define SelectionNotify     31
```

```
#define ColormapNotify        32
#define ClientMessage         33
#define MappingNotify         34
#define XError                0
#define Response              1
```

fig15. Les définitions C des différents types d'événements

La structure **XAnyEvent** comporte cinq champs qui sont accessibles dans tout type d'événement:

```
typedef struct {
    int type;            /* type de l'événement */
    unsigned long serial;/* dernière requête traitée */
    Bool send_event; /* vrai si envoyé par XSendEvent */
    Display *display;/* où l'événement a eu lieu */
     Window window;   /* fenêtre recevant l'événement */
} XAnyEvent;
```

Les autres structures correspondent à un type particulier d'événement mais comportent ces cinq mêmes champs (déclarations dans le même ordre[10]), suivis de champs plus spécifiques au type d'événement considéré: position de la souris pour les événement de type XButtonEvent, détail sur les touches pour XKeyEvent, etc.

La sélection des événements et les masques

Les masques d'événements servent généralement à sélectionner le type des événements reçus (ou bloqués) par une fenêtre. Cependant ne perdons pas de vue que:

1. Certains événements ne sont pas sélectionnés par des masques. Ainsi par exemple les événements ClientMessage et MappingNotify sont toujours envoyés aux clients concernés.
2. Certains masques ne sélectionnent aucun événement mais donnent des indications sur la manière dont les événements seront envoyés. Ainsi **PointerMotionHintMask** restreint le nombre d'événements de mouvement de souris à envoyer au client (cf. la description de MotionNotify).

[10] Cette disposition, bien connue des programmeurs C, permet des accès aux données indépendantes du type de la structure.

3. Plusieurs masques peuvent également sélectionner l'envoi d'un même type d'événement (ex: **ButtonMotionMask** et **PointerMotionMask**) et enfin un même masque peut permettre de sélectionner plusieurs événements (exemple **StructureNotifyMask** ou **SubstructureNotifyMask**).

Les deux figures suivantes énumèrent les différents masques que l'on peut utiliser et indiquent brièvement la correspondance entre ces masques et les événements. Dans la section consacrée à la description des événements, les masques seront indiqués en caractères gras en face du type d'événement considéré, le champ d'accès à la structure étant indiqué en italique.

```
#define NoEventMask                 0L
#define KeyPressMask                (1L<<0)
#define KeyReleaseMask              (1L<<1)
#define ButtonPressMask             (1L<<2)
#define ButtonReleaseMask           (1L<<3)
#define EnterWindowMask             (1L<<4)
#define LeaveWindowMask             (1L<<5)
#define PointerMotionMask           (1L<<6)
#define PointerMotionHintMask       (1L<<7)
#define Button1MotionMask           (1L<<8)
#define Button2MotionMask           (1L<<9)
#define Button3MotionMask           (1L<<10)
#define Button4MotionMask           (1L<<11)
#define Button5MotionMask           (1L<<12)
#define ButtonMotionMask            (1L<<13)
#define KeymapStateMask             (1L<<14)
#define ExposureMask                (1L<<15)
#define VisibilityChangeMask        (1L<<16)
#define StructureNotifyMask         (1L<<17)
#define ResizeRedirectMask          (1L<<18)
#define SubstructureNotifyMask      (1L<<19)
#define SubstructureRedirectMask    (1L<<20)
#define FocusChangeMask             (1L<<21)
#define PropertyChangeMask          (1L<<22)
#define ColormapChangeMask          (1L<<23)
#define OwnerGrabButtonMask         (1L<<24)
```

fig16. Les différents masques d'événements

Masques	Evénements
KeyPressMask	KeyPress
KeyReleaseMask	KeyRelease
ButtonPressMask	ButtonPress
ButtonReleaseMask	ButtonRelease
EnterWindowMask	EnterNotify
LeaveWindowMask	LeaveNotify
PointerMotionMask	MotionNotify
Button<n>MotionMask	MotionNotify
ButtonMotionMask	MotionNotify
KeymapStateMask	KeymapNotify
ExposureMask	Expose
VisibilityChangeMask	VisibilityNotify
StructureNotifyMask	CirculateNotify ConfigureNotify DestroyNotify GravityNotify MapNotify UnmapNotify ReparentNotify
ResizeRedirectMask	ResizeRequest
SubstructureNotifyMask	comme StructureNotifyMask plus CreateNotify
SubstructureRedirectMask	CirculateRequest MapRequest ConfigureRequest
FocusChangeMask	FocusIn FocusOut
PropertyChangeMask	PropertyNotify
ColormapChangeMask	ColormapNotify
NoEventMask	ne sélectionne pas d'ev.
OwnerGrabButtonMask	ne sélectionne pas d'ev.
PointerMotionHintMask	ne sélectionne pas d'ev.
pas de masque associé	ClientMessage
pas de masque associé	GraphicsExpose NoExpose
pas de masque associé	MappingNotify
pas de masque associé	SelectionClear
pas de masque associé	SelectionNotify
pas de masque associé	SelectionRequest

fig17. La correspondance Masques/Evenements

Conventions générales et appropriation des dispositifs d'entrées

Nous avons pour l'instant brossé un tableau général expliquant les mécanismes mis en jeu pour la communication avec le serveur. Les clients émettent leurs demandes sous forme de requêtes (fonctions) et ils reçoivent des informations sous forme d'événements (structures récupérées avec XNextEvent). Nous avons ainsi introduit la nature des informations communiquées (type d'événement et structures associées). Cependant les mécanismes à mettre en oeuvre pour recevoir ces informations sont parfois plus complexes.

Nous avons vu dans le paragraphe précédent que pour recevoir un événement il fallait (condition nécessaire en général[11]) en avoir fait la demande auprès du serveur en *sélectionnant cet événement* pour une fenêtre. En réalité cette condition n'est qu'une condition nécessaire et ne sera que rarement une condition suffisante. En effet, comme nous l'avons vu dans le paragraphe sur la propagation des événements, toutes les fenêtres ayant sélectionné un même événement ne le recevront en général pas. Le problème se pose tout particulièrement en ce qui concerne les événements sur le clavier et la souris. En effet, ces deux dispositifs d'entrées sont partagés par tous les clients (i.e. toutes les applications tournant simultanément à l'écran) et le problème de savoir à qui est destinée l'information provenant des actions de l'utilisateur se pose ici de manière cruciale.

Conventions générales

1. Une seule fenêtre à un instant donné recevra les événements concernant les entrées du clavier ou de la souris. La fenêtre recevant les entrées de touches au clavier est dite *avoir le focus du clavier*. La convention naturelle étant que la fenêtre située sous le curseur de souris (i.e. la plus basse dans la hiérarchie des fenêtres donc la plus en avant sur l'écran) aura le focus. On pourra cependant empêcher la mise en oeuvre de cette convention[12] en faisant un appel à la fonction **XSetInputFocus**().

2. La fenêtre recevant les événements de souris sera également la fenêtre située sous le curseur de souris, sauf entre deux événements de type ButtonPress et ButtonRelease - auquel cas les événements de souris sont acheminés vers la fenêtre ayant reçu l'événement ButtonPress (comportement du système par défaut). On parlera alors *d'appropriation* ou de *saisie automatique* du clavier (*automatic grab* dans la documentation).

---

[11] En réalité certains événements sont envoyés systématiquement à une l'application" et n'ont pas besoin d'être sélectionnés par masque.

[12] De nombreux window manager préfèrent que l'utilisateur désigne la fenêtre ayant le focus du clavier en cliquant préalablement dans la fenêtre concernée.

Les conventions précédentes décrivent un comportement par défaut du système. En réalité, on peut sortir de ces conventions et demander au serveur le monopole sur un dispositif d'entrée comme le clavier ou la souris. En dernière instance, ce sera alors le client le dernier en date en ayant exprimé la demande qui aura gain de cause.

On peut ainsi s'approprier un bouton, la souris, le clavier ou une combinaison de touches du clavier. le terme *Active Grab* pour un dispositif désigne l'appropriation des événements relatifs à ce dispositif par une fenêtre (appelée *grab-window*). Les événements concernés par le dispositif monopolisé ne suivent plus alors le mécanisme standard de propagation et sont toujours envoyés à la fenêtre accaparante quelque soit la position de la souris[13].

Une appropriation peut être activée par:

1. L'enfoncement d'un bouton de souris (c'est le comportement de monopole par défaut appelé "automatic grab" dans la documentation)
2. Un appel d'un client à **XGrabPointer** ou **XGrabKeyboard**
3. La réalisation des actions définies lors d'une appropriation (passive) par **XGrabButton** ou **XGrabKey.**

Des fonctions symétriques permettent d'interrompre ces mécanismes d'appropriation :

XUngrabPointer
XUngrabKeyboard
XUngrabKey
XUngrabButton

Signalons aussi **XChangeActivePointerGrab** qui permet de modifier les paramètres d'une appropriation active de la souris (par exemple le curseur de souris qui concrétise cette appropriation à l'écran).

Lors d'un monopole actif de la souris, on peut rendre normal l'acheminement des événements *à l'intérieur de l'application* par la sélection du masque **OwnerGrabButtonMask** (en association avec **ButtonPressMask)** ou bien en donnant la valeur True au booléen *owner_events* dans la requête d'appropriation ou de modification.

[13] En outre, si la fenêtre accaparante n'a pas sélectionné ces événements, ces derniers seront perdus.

Le déclenchement d'un monopole sur la souris générera l'envoi d'événements EnterWindow et LeaveWindow aux fenêtres concernées (le champ detail indiquera qu'il s'agit d'un "grab"). De même, un monopole du clavier est caractérisé par l'attribution du focus du clavier et il générera l'envoi d'événements FocusIn et FocusOut[14].

Communication entre clients

Plusieurs mécanismes de communication entre clients ont été prévus. Tous ces mécanismes suivent les conventions définies dans l'*Inter-Client Communication Convention Manual*. Ils sont basés sur la notion de propriété. Chaque propriété à un unique identificateur (appelé atome). Une propriété est individualisée de manière unique par un atome et une fenêtre. Elle possède en outre un nom (chaîne ASCII), un type (qui est lui-même une propriété) et un certains nombre de données associées. Il existe certains types de propriétés prédéfinies. Dès la Release 2, il y avait 68 propriétés prédéfinies pour la communication avec le window manager, les mécanismes de sélections, la table de couleur standard et les spécifications de fontes.

Voici une brève énumération des principaux mécanismes de communication:

1. La fonction **XSendEvent**(dpy, win, propagate, event_mask, event) permet à une application d'envoyer un événement à une fenêtre indépendamment des mécanismes d'appropriation. On peut spécifier la fenêtre par son identificateur ou par sa situation relativement au système (PointerWindow ou InputFocus) . Cette fonction est utilisée, pour la communication entre clients intéressés par la même sélection (voir plus loin), pour simuler des actions d'utilisateur lors de démonstrations, ou pour tout autre communication conventionnellement fixée par des applications. L'événement de type **ClientMessage** permet d'échanger une information quelconque.

2. Des **propriétés** permettent de communiquer avec le **window manager.** On peut ainsi indiquer des préférences concernant la taille des fenêtres, les icônes à utiliser ou le titre à placer dans la bannière de décoration. On pourra par exemple utiliser les fonctions suivantes: **XStoreName**(dpy, w, name), **XSetIconName**(dpy, win, name), **XSetWMHints**(dpy, win, wmhints), **XSetNormalHints**(dpy, win, hints), etc.

3. Deux propriétés (XA_PRIMARY et XA_SECONDARY) ont permi d'implanter un mécanisme de communication de données appelée **sélection** qui utilise trois

[14] Ces événements sont habituellement envoyés aux fenêtres concernées lors d'un appel à XSetInputFocus() - fonction permettant d'attribuer le focus du clavier à une fenêtre particulière.

événements de types particuliers: **SelectionNotify**, **SelectionRequest** et **SelectionClear**. Le principe est le suivant: une seule fenêtre est propriétaire à un instant donné de la sélection. **XGetSelectionOwner** retourne ce propriétaire. On peut se rendre propriétaire en faisant appel à **XSetSelectionOwner** ou à **XGetWindowProperty**. Le serveur envoie alors un événement de type SelectionClear à l'ancien propriétaire qui peut désélectionner l'ancienne sélection à l'écran. Par ailleurs la fonction **XConvertSelection** permet de demander au propriétaire de la sélection de la convertir en un certain format pour en récupérer les données. Le serveur envoie alors un événement de type SelectionRequest au propriétaire courant de la sélection qui est tenu d'envoyer en réponse, par XSendEvent à (l'application propriétaire de) la fenêtre intéressée, un événement de type SelectionNotify.

4. Un mécanisme de communication de chaînes de caractères basés sur l'utilisation de tampons appelés *cut buffer* (il en existe 8) permet également d'échanger des chaînes de caractères plus facilement en faisant appel à des requêtes particulières comme **XStoreBytes**, **XFetchBytes**, etc.

Description des événements

Evénements concernant le clavier et la souris

KeyPress (clavier) **KeyPressMask**
KeyRelease **KeyReleaseMask**
type: XKeyEvent *structure : event.xkey*

Une touche du clavier est enfoncée ou relâchée. Pour recevoir ces événements il faut bénéficier du focus du clavier[15] et avoir sélectionné l'événement par KeyPressMask (ou KeyReleaseMask). La structure de l'événement contient le code émis par la touche (*keycode*) et un code symbolique (*keysym*). Le code symbolique est décrit par des constantes définies dans le fichier /usr/include/X11/keysym.h. L'événement permet de connaître l'état des modifieurs (touches Shift, Control, MetaLeft, Compose etc.). On interprète généralement le code émis par la touche directement à partir de l'événement grâce à la fonction **XLookupString**. On peut modifier les caractères ASCII associés par XLookupString (dans le buffer) au code symbolique *keysym* grâce à la fonction **XRebindKeysym**. Cette redéfinition est locale au client. Pour reconnaître les constantes retournées par XLookupString il faut inclure également le fichier X11/Xutil.h.

Remarques:
1. Toutes les machines ne sont pas capables de générer l'événement KeyRelease.
2. **XSetInputFocus** attribue le focus à une fenêtre donnée quelque soit la position de la souris. On peut désigner une fenêtre par des constantes (ex: PointerWindow, InputFocus, etc.).
3. Il ne peut y avoir d'appropriation du clavier entre un KeyPress et un KeyRelease.

---

[15] Par défaut, le focus du clavier est attribué à la fenêtre qui contient la souris à un moment donné; cependant, certains window manager aiment que le focus soit indiqué par l'utilisateur. Cette deuxième convention utilise la fonction XSetInputFocus.

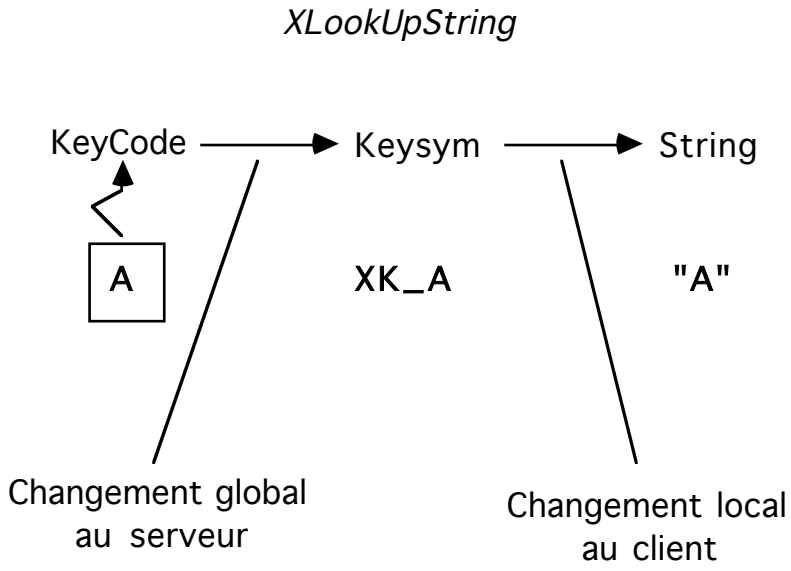

fig18. Codage des touches du clavier

ButtonPress (souris) **ButtonPressMask**
ButtonRelease **ButtonReleaseMask**
type: XButtonEvent *structure : event.xbutton*
Un bouton de la souris est enfoncé ou relâché. On sélectionne ces événements avec ButtonPressMask et ButtonReleaseMask. Les événements sont envoyés à la fenêtre dans laquelle se trouve la souris, sauf en cas d'appropriation de la souris. La structure event.xbutton contient les informations suivantes: heure de l'événement, fenêtre associée, état des modifieurs, numéro de bouton de souris, etc.


Attention:: L'événement ButtonPress déclenche lui-même un monopole de la souris par la fenêtre qui a récupéré cet événement - i.e. cette fenêtre recevra tous les événements de souris jusqu'à ce que l'événement ButtonRelease se produise. Durant toute cette période, le curseur de souris gardera la forme qui lui était attribué dans la fenêtre accaparante même s'il se trouve à l'extérieur et aucune autre fenêtre ne recevra d'événement souris. On peut cependant supprimer cette redirection automatique des événements vers la fenêtre accaparante au niveau de l'application et transmettre les événements à une fenêtre de l'application située sous le curseur de souris (comme c'est le cas habituellement) en rajoutant le masque **OwnerGrabButtonMask** lors de la

sélection du ButtonPress[16]. Dans ce cas, les événements souris sont envoyés à la fenêtre se trouvant sous le curseur (si elle appartient à l'application et si elle a sélectionné ces événements) jusqu'au ButtonRelease suivant.

MotionNotify (souris) **PointerMotionMask**
ButtonMotionMask
Button<N>MotionMask
type: XMotionEvent *structure : event.xmotion*
La souris a bougé. Cet événement peut être envoyé pour un mouvement quelconque de la souris s'il a été sélectionné par PointerMotionMask. On peut restreindre l'envoi des événements aux mouvements se produisant avec un bouton enfoncé en utilisant ButtonMotionMask, ou Button<n>MotionMask (n prévu de 1 à 5) au lieu de PointerMotionMask. Les cinq masques Button<n>MotionMask permettent de spécifier quel bouton est précisément enfoncé et on peut les combiner entre eux.

En outre, on peut combiner les masques de sélection précédents avec le masque **PointerMotionHintMask** qui précise[17] qu'un nouveau mouvement de souris ne doit être envoyé que si l'on est dans la situation suivante:
1. un changement d'état (enfoncé/relâché) d'une touche ou d'un bouton se produit
2. la souris sort de la fenêtre
3. le client fait un appel à **XQueryPointer** (ou à **XGetMotionEvents**).
Ce mécanisme permet de suivre plus rapidement la position de la souris.

EnterNotify (souris) **EnterWindowMask**
LeaveNotify **LeaveWindowMask**
type: XCrossingEvent *structure : event.xcrossing*
La souris est entrée ou sortie d'une fenêtre. Ces événements se produisent lorsqu'on déplace la souris d'une fenêtre à l'autre ou lorsqu'on affiche (ou retire) une fenêtre sous la souris. En outre, des événements EnterNotify et LeaveNotify sont également envoyés aux fenêtres qui sont traversées *virtuellement*. Ces fenêtres sont les fenêtres qui se trouvent entre la fenêtre d'origine et la fenêtre destination dans la hiérarchie. Dans ce cas, le champ detail de la structure event.xcrossing contiendra la valeur NotifyVirtual. De même des événements de type EnterNotify et LeaveNotify sont générés quand la souris est soumise à un monopole et que le pointeur ne se trouvait pas déjà dans la fenêtre accaparante. Dans ce cas, la fenêtre accaparante reçoit un EnterNotify et la fenêtre dans laquelle se trouvait le pointeur un LeaveNotify (avec le champ mode à NotifyUngrab). La position du pointeur dans les deux événements est celle qui précédait l'appropriation. Quand le monopole se

[16] Il y a un deuxième cas d'exception: si l'application avait invoqué un monopole passif du bouton sur une fenêtre ancêtre de la fenêtre dans laquelle se produit le buttonPress.
[17] Ce masque ne sélectionne pas l'événement.

termine, on a émission d'événements symétriques indiquant la fin du processus d'appropriation.

FocusIn (clavier) **FocusChangeMask**
FocusOut *structure : event.xfocus*

Ces événements sont envoyés quand le focus d'une fenêtre change après un appel à **XSetInputFocus**. Ces événements sont analogues à EnterNotify et LeaveNotify mais ils concernent le clavier alors que EnterNotify et LeaveNotify concernent la souris. La fenêtre ayant le focus du clavier et ses descendantes sont les seules à pouvoir recevoir des événements d'enfoncement de touches. Par défaut c'est la fenêtre racine qui a le focus. X inclut également une notion de focus virtuel basé sur la hiérarchie des fenêtres. On peut également recevoir des événements de type focus en cas d'appropriation du clavier. Les champs mode et detail de la structure indiqueront s'il s'agit d'événements simples, virtuels ou s'ils résultent d'une appropriation.

KeymapNotify (clavier) **KeymapStateMask**
structure : event.xkeymap

Cet événement est émis juste après un EnterNotify ou un FocusIn. Il donne l'état initial des modifieurs et indique que l'application peut recevoir des entrées.

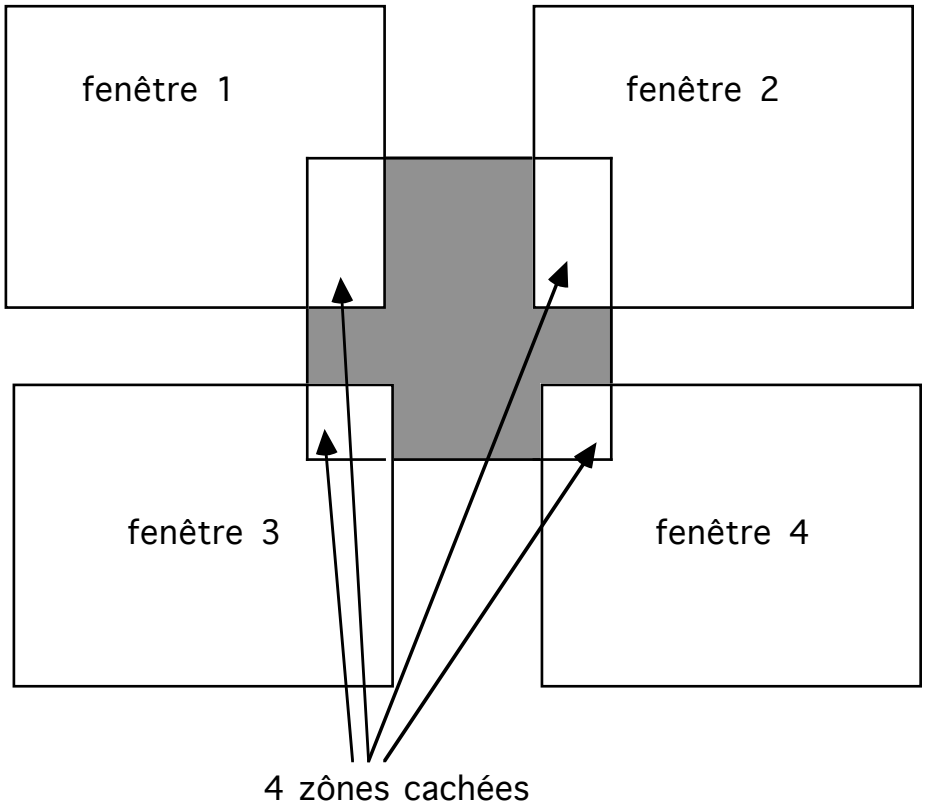

fig13. Evénements Expose
Si la fenêtre grise passe devant ses soeurs, elles recevra (au moins) quatre événements de type Expose correspondant à chacune des zones antérieurement cachées.

Evénements pour l'affichage des dessins

Expose **ExposureMask**
type: XExposeEvent *structure : event.xexpose*
Un événement de type Expose est envoyé quand une fenêtre devient visible ou qu'une partie précédemment invisible devient visible. Cet événement est sélectionné avec ExposureMask et ne concerne que les fenêtres **InputOutput**. La structure associée contient la description précise de la partie devenue visible et le champ count indique le rang dans la pile des événements de type Expose en attente. Si l'application effectue des dessins à l'écran, ces dessins seront tracés sur la réception d'un événement de type Expose. Pour optimiser l'affichage, on pourra stocker les différents rectangles exposés dans un tableau de XRectangle et on ne dessinera que sur réception d'un événement dont le champ count est à zéro (cf. la fonction **XSetClipRectangles**).

GraphicsExpose *structure : event.xgraphicsexpose*
NoExpose *event.xnoexpose*
type: XNoExposeEvent
Ces événements indiquent si une partie copiée par XCopyArea ou XCopyPlane était effectivement accessible dans la source et avertissent du succès (NoExpose) ou de l'échec (GraphicsExpose) de la copie. (Ils ne disent rien par contre de la destination). Ces événements ne sont pas sélectionnés par des masques. Pour les recevoir, il faut avoir positionné l'attribut *graphic_exposures* du contexte graphique à **True** dans la requête de copie.

Evénements concernant les propriétés et la géométrie des fenêtres

Remarques:
1. La plupart des événements en ...Notify rapportent une modification dans les attributs des fenêtres tandis que les événements en ...Request rapportent des appels de requêtes effectués par les clients sur ces fenêtres. Les événements en ...Request sont principalement sélectionnés par le window manager.
2. Les événements sélectionnables par StructureNotifyMask sur une fenêtre peuvent être sélectionnés globalement pour toutes les fenêtres soeurs en sélectionnant SubstructureNotifyMask sur la fenêtre parent.

ConfigureNotify **StructureNotifyMask**
SubstructureNotifyMask
type: XConfigureEvent *structure : event.xconfigure*
Cet événement annonce tout changement dans la configuration géométrique d'une fenêtre (taille, position, bord, ordre d'empilement). On le sélectionne avec StructureNotifyMask. Pour le recevoir pour tous les enfants d'une fenêtre, il suffit de sélectionner SubstructureNotifyMask sur la fenêtre parent.

VisibilityNotify **VisibilityChangeMask**
type: XVisibilityEvent *structure : event.xvisibility*
rapporte tout changement de visibilité dans une fenêtre de type InputOutput.

CreateNotify **SubstructureNotifyMask**
type: XCreateWindowEvent *structure : event.xcreatewindow*
informe de la création de fenêtre. Cet événement est sélectionné par SubstructureNotifyMask sur la fenêtre parent.

DestroyNotify **SubstructureNotifyMask**
type: XDestroyWindowEvent *structure : event.xdestroywindow*
informe de la destruction de fenêtre. Cet événement est sélectionné par SubstructureNotifyMask sur la fenêtre parent.

MapNotify **StructureNotifyMask**
UnmapNotify **SubstructureNotifyMask**
types: XMapEvent *structure : event.xmap*
XUnmapEvent *event.xunmap*
sont envoyés par le serveur quand une fenêtre passe de l'état Map à Unmap et vice versa.

MapRequest **SubstructureRedirectMask**
type: XMapRequestEvent *structure : event.xmaprequest*
événement généré par un appel à XMapRaised ou à XMapWindow. L'attribut *override_redirect* de la sous-fenêtre considérée doit en outre être à **False** pour que cet événement soit reçu. Si cet événement est sélectionné par le window manger, la fenêtre n'est pas affichée par le serveur, mais cet événement est envoyé au window manager (redirection de la requête) pour lui permettre de modifier la taille ou la position de cette fenêtre selon ses propres critères.

ReparentNotify **StructureNotifyMask**
SubstructureNotifyMask
type: XReparentEvent *structure : event.xreparent*
est émis pour indiquer que le parent de la fenêtre a changé. Il est important de sélectionner cet événement si on s'intéresse à la position d'une fenêtre fille de la racine, car celle-ci est susceptible d'être "reparentée" par un window manager.

ConfigureRequest **SubstructureRedirectMask**
type: XConfigureRequestEvent *structure : event.xconfigurerequest*
indique qu'un client a essayé de modifier la taille, position bord et/ou rang d'empilement de la fenêtre.

GravityNotify **StructureNotifyMask**
SubstructureNotifyMask
type: XGravityEvent *structure : event.xgravity*
est émis quand un déplacement de fenêtre est causé par un changement de la taille de son parent

ResizeRequest **ResizeRedirectMask**
type: XResizeRequestEvent *structure : event.xresizerequest*
indique qu'un client souhaite changer la taille de la fenêtre (celle-ci ne l'a pas été).

CirculateNotify **StructureNotifyMask**
SubstructureNotifyMask

type: XCirculateEvent *structure : event.xcirculate*
Une fenêtre a été déplacée par XCirculateWindowUp ou Down. Si le gestionnaire de fenêtre empêche cette opération, l'événement n'est pas envoyé.

CirculateRequest **SubstructureRedirectMask**
type: XCirculateRequestEvent *structure : event.xcirculaterequest*
rapporte l'appel d'un client à une fonction modifiant l'empilement des sous-fenêtres XCirculateSubwindows,
XCirculateSubWindowsDown ou XCirculateSubWindowsUp).

PropertyNotify **PropertyChangeMask**
type: XPropertyEvent *structure : event.xproperty*
indique qu'une propriété de la fenêtre a été modifiée - à tout le moins qu'il y a été rajouté une chaîne de longueur zéro. Cet événement rapporte l'heure de la modification.

Evénements concernant la communication entre clients

ColormapNotify **ColormapChangeMask**
type: XColormapEvent *structure : event.xcolormap*
Quelqu'un a modifié la table des couleurs standard ou l'attribut colormap de la fenêtre.

MappingNotify *structure: event.xmapping*
type: XMappingEvent
informe qu'un changement d'association a été effectué par un autre client sur les ressources concernant le clavier ou la souris. Par exemple, un appel à **XChangeKeyboardMapping** (qui modifie les liens entre touches physiques et touches symboliques) ou un appel à **XSetModifierMapping** (qui modifie les liens entre touches modifieurs et modifieurs logiques), ou encore un appel à **XSetPointerMapping** (qui modifie les liens entre les boutons physiques et les bouton logiques de la souris) ont été traités par le serveur. La réaction normale à un changement sur les touches est un appel à **XRefreshKeyboardMapping** (tester le champ *request* de l'événement) pour remettre à jour les tables locales.

SelectionClear *structure: event.xselectionclear*
type: XSelectionClearEvent
rapporte au propriétaire courant de la sélection qu'un nouveau propriétaire est défini. Cet événement n'est pas sélectionné par masque, mais toujours envoyé à l'ancien propriétaire d'une sélection lorsqu'un autre client appelle **XSetSelectionOwner** pour la même sélection.

SelectionRequest *structure: event.xselectionrequest*
type: XSelectionRequestEvent
est envoyé au propriétaire de la sélection quand un autre client en demande le contenu par **XConvertSelection**.

SelectionNotify *structure: event.xselection*
type: XSelectionEvent
est envoyé directement par les clients avec **XSendEvent**(dpy, win, propagate, event_mask, event).(La fonction XSendEvent permet d'indiquer un identificateur de fenêtre comme PointerWindow ou InputFocus à laquelle envoyer le message). Le propriétaire d'une sélection l'envoie aux clients demandeurs (ceux qui auront fait **XConvertSelection** d'un certaine propriété) quand celle-ci aura été convertie (ou non: un champ permet de l'indiquer).

ClientMessage *structure: event.xclient*
type: XClientMessageEvent
est envoyé directement à une fenêtre par **XSendEvent**. La fenêtre cible peut être désignée par **PointerWindow** ou **InputFocus**.

Les Dessins

Le contexte graphique

La librairie Xlib propose un certain nombre de fonctions de dessin. Ces fonctions sont assez variées: contours de figures géométriques, morceaux de textes, remplissages de figures, etc. Toutes ces fonctions graphiques prennent en argument un contexte graphique (type **GC**) qui permet de faire varier un certain nombre de paramètres du dessin. Un contexte graphique est donc une sorte de paramètre multiple qui contient des attributs graphiques. les attributs du contexte graphique utilisés par une primitive de dessin varient d'une primitive à l'autre.

Ainsi par exemple, le contexte graphique comporte un attribut *font* et un attribut permettant de définir un style de ligne (*line_style*). Les fonctions graphiques qui dessinent des contours de figures (sous forme de lignes) utiliseront l'attribut *line_style* alors que les fonctions qui dessinent du texte utiliseront l'attribut de type Font qui spécifie la police de caractères.

Cependant, un principe général sous-tend l'usage du contexte graphique pour toutes les primitives de dessin. Une primitive graphique engendre un bitmap représentant les points du dessin origine (par exemple, une droite avec XDrawLine). Cette primitive de dessin indique en outre par ses arguments d'appel où doit être effectué ce dessin (la destination) et dans quel contexte graphique (GC) il doit être effectué. Le dessin viendra alors modifier la destination (en général une fenêtre) en fonction des paramètres du contexte graphique. Pur toutes les primitives, la destination est un argument de type **Drawable**, i.e. quelque chose sur lequel on peut dessiner: soit une fenêtre de type **InputOutput**, soit un **Pixmap**. Les paramètres du contexte graphique sont consultés (par exemple les couleurs) pour déterminer les valeurs des pixels des points effectivement modifiés sur la destination. Le Pixmap qui en résulte est placé finalement sur les points de la destination.

Remarques:
1. Une primitive de dessin utilise fréquemment plusieurs paramètres du contexte graphique.
2. Un contexte graphique est en général une variable globale au programme et n'est associé à aucune fenêtre particulière.
3. Pour faire des dessins différents on pourra disposer de plusieurs contextes graphiques ou bien faire varier un même contexte graphique. Dans le premier cas, on minimise les échanges avec le serveur, dans le deuxième, on économise la place mémoire.

Les paramètres du contexte graphique

La structure de données permettant de définir les paramètres d'un contexte graphique est de type **XGCValues** (cf. figure). Les premiers paramètres sont utilisés par toutes les primitives de dessin.

La fonction de transfert (***function***) indique une opération logique à effectuer entre le Pixmap d'origine et le Pixmap de destination. Il existe 16 fonctions de transfert correspondant aux 16 connecteurs binaires de la logique. Pour chacune d'elles un nom (entier) a été défini en C.

```
GXclear             /* 0 */
GXand               /* src ET dst */
GXandReverse        /* src ET NON dst */
GXcopy              /* src */
GXandInverted       /* NON src ET dst */
GXnoop              /* dst */
GXxor               /* src OU exclus. dst */
GXor                /* src OU dst */
GXnor               /* NON src ET NON dst */
GXequiv             /* NON src OU exclus.dst */
GXinvert            /* NON dst */
GXorReverse         /* src OU NON dst */
GXcopyInverted      /* NON src */
GXorInverted        /* NON src OU dst */
GXnand              /* NON src OU NON dst */
GXset               /* 1 */
```

fig19. Les fonctions de transfert

```
typedef struct {
     int function;  /* opération logique entre
                         la source et la destination */
     unsigned long plane_mask;     /*  masque des plans */
     unsigned long foreground;     /* couleur du dessin */
     unsigned long background;     /* couleur du fond */
     int line_width;               /* largeur de ligne */
     int line_style; /*LineSolid,LineOnOffDash,
                                   LineDoubleDash */
     int cap_style; /* CapNotLast, CapButt,
                              CapRound, CapProjecting */
     int join_style;/* JoinMiter, JoinRound, JoinBevel */
```

```
    int fill_style;/* FillSolid, FillTiled,
                      FillStippled,FillOpaqueStippled */
    int fill_rule; /* EvenOddRule, WindingRule */
    int arc_mode;  /* ArcChord, ArcPieSlice */
    Pixmap tile;   /* tuile pour opération de pavage */
    Pixmap stipple; /* pochoir (Pixmap plan)  */
    int ts_x_origin; /* décalage du premier pavé */
    int ts_y_origin;
    Font font;      /*  fonte utilisée pour le texte */
    int subwindow_mode;/*  ClipByChildren,
                             IncludeInferiors*/
    Bool graphics_exposures; /*doit-on générer des ev.? */
    int clip_x_origin; /* origine de la zone du dessin */
    int clip_y_origin;
    Pixmap clip_mask; /* masque de la zone de dessin */
    int dash_offset; /*  pour le mode pointillé */
    char dashes;
} XGCValues;
```

fig20. La structure de données XGCValues qui permet de spécifier un contexte graphique

La fonction de transfert sera effectuée bit à bit pour chaque pixel de la zone concernée par le dessin (cf. figure suivante). Ainsi la fonction **GXCopy** permet d'ignorer la couleur des points de la destination. Inversement, **GXnoop** ne tiendra compte que de la couleur des points d'arrivée. On utilisera également volontiers **GXxor** (ou **GXInvert**) pour faire de l'inverse video.

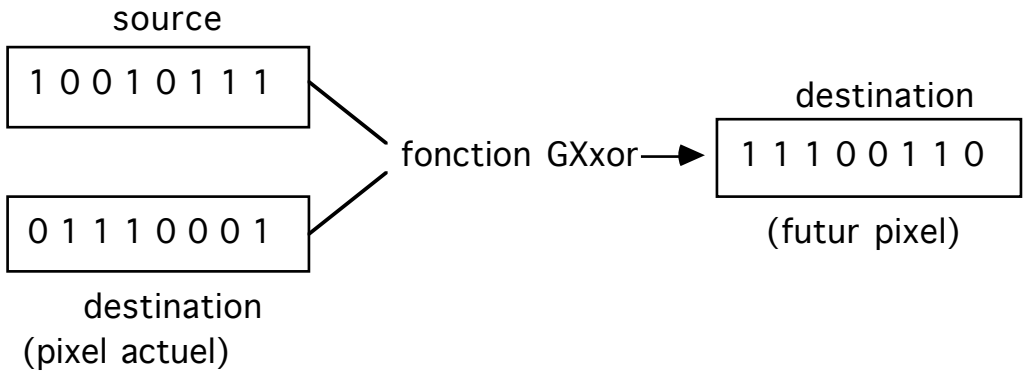

fig21. Effet de la fonction GXxor sur un pixel

Le masque des plans (***plane_mask***) permet de modifier encore le Pixmap obtenu après avoir fait opérer la fonction de transfert. On utilise ce champ sur

les machines couleur. Cette fois, on ne reporte le dessin sur la destination que pour les plans indiqués par le masque des plans. Les autres plans de la destination restent inchangés, i.e les bits des pixels concernés par le dessin gardent les valeurs qu'ils avaient précédemment sur la destination dans les plans non masqués (cf. figure 22).

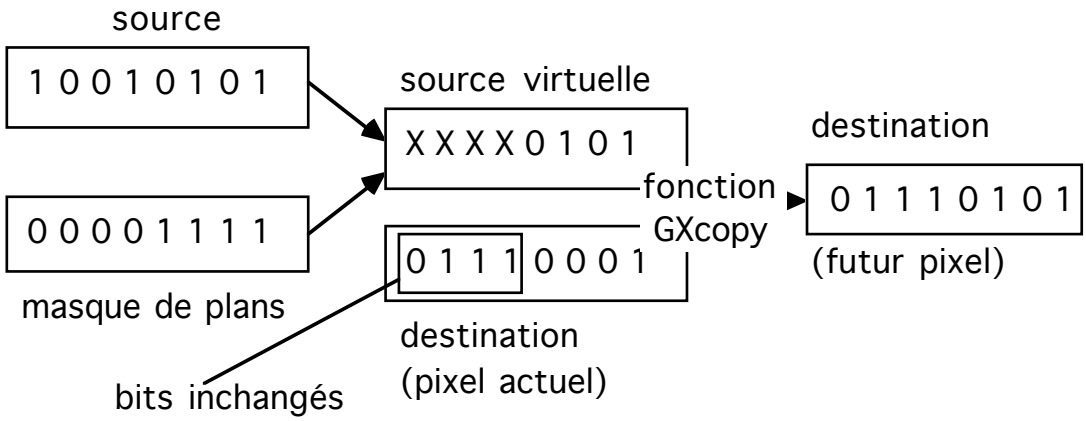

fig22. Effet du masque des plans sur un pixel (avec la fonction GXcopy)

L'attribut ***foreground*** permet d'indiquer un pixel identifiant la couleur dans laquelle le dessin sera peint.

L'attribut ***background*** indique la couleur du fond utilisée par les primitives de remplissage.

Viennent ensuite quatre attributs destinés à préciser la forme des contours de lignes pour les primitives dessinant des contours de polygones ou des segments:

line_width: indique l'épaisseur des lignes

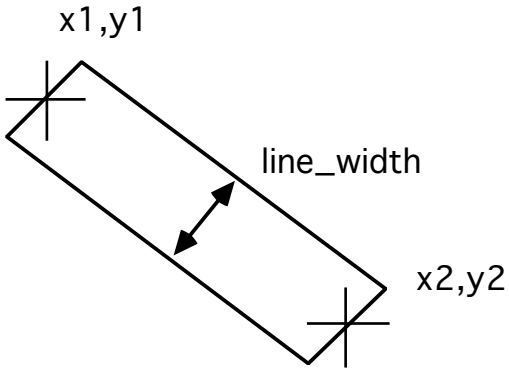

fig23. L'épaisseur de ligne (line_width)

line_style: indique le style de ligne, normal (LineSolid), pointillé transparent (LineOffOnDash) ou pointillé opaque (LineDoubleDash)

fig24. Le style de ligne (line_style)

cap_style: indique la manière dont se terminent les lignes.

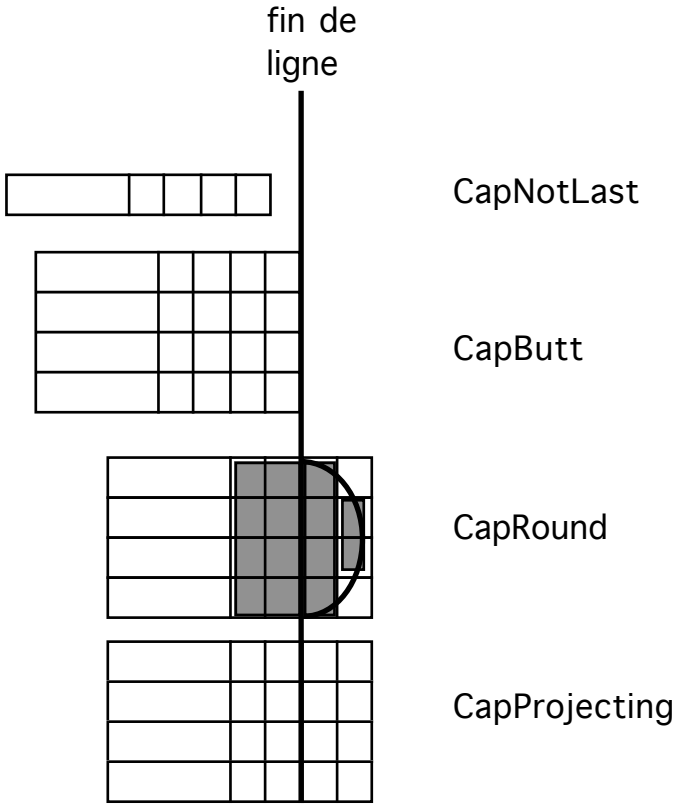

fig25. Les fins de lignes (cap_style)

join_style: indique la manière dont se joignent les segments.

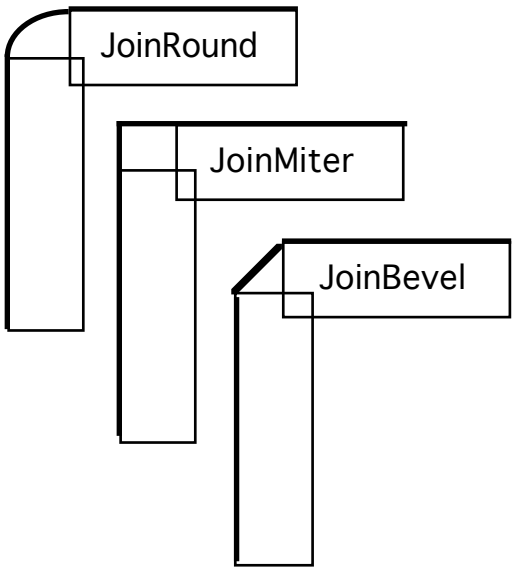

fig26. Les styles de raccords de lignes (join_style)

Pour les lignes pointillées, deux autres attributs seront utilisés: ***dash_offset***, et ***dashes***. On pourra ainsi préciser le nombre de points à partir duquel effectuer le pointillé et la longueur d'un pointillé. La fonction **XSetDashes** permet d'indiquer des alternances plus complexes.

Les attributs ***fill_style***, ***fill_rule***, ***arc_mode***, ***tile***, ***stipple***, ***ts_x_origin*** et ***ts_y_origin*** concernent les primitives de remplissage. Les primitives de remplissage utilisent a priori la couleur du fond (***background***) pour remplir le dessin mais peuvent également utiliser un motif et effectuer un pavage du dessin en le reproduisant régulièrement. Ce motif est indiqué dans un Pixmap appelé ***tile*** (qui signifie tuile en anglais) pour rappeler la disposition des tuiles sur un toit. Un autre motif est également disponible, le Pixmap ***stipple*** (pochoir). Ce dernier est un Pixmap plan dont les couleurs sont déterminées par celles indiquées dans les champs ***foreground*** et éventuellement ***background***. La manière dont est en fait effectué le remplissage du dessin est indiqué dans le champ ***fill_style*** et est illustré par la figure 27.

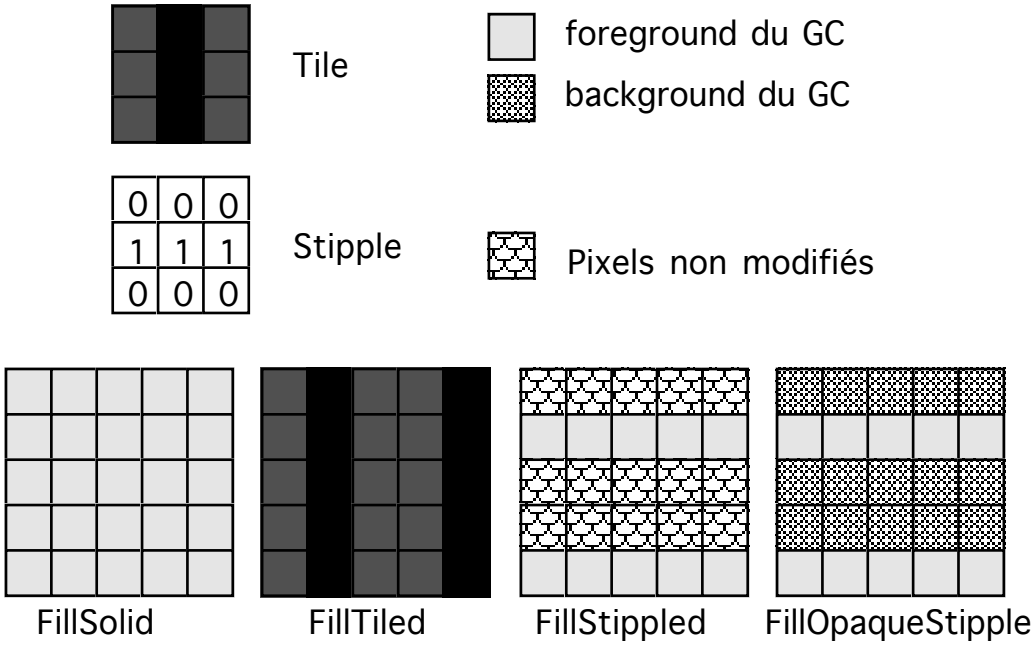

fig27. Le style de remplissage (fill_style)

Les attributs ***ts_x_origin*** et ***ts_y_origin*** indiquent l'origine à partir de laquelle est reporté le motif dans le cas d'un pavage du dessin.

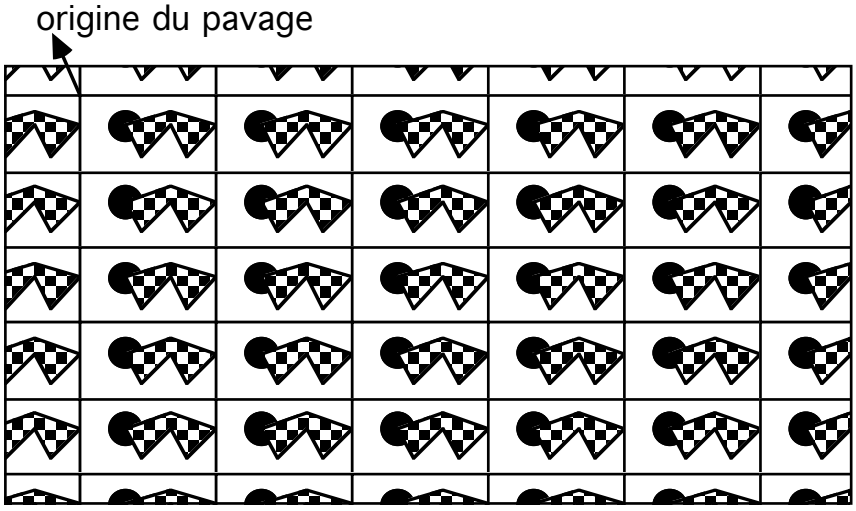

fig28. L'origine du pavage

L'attribut ***fill_rule*** concerne le dessin des polygones. Il peut prendre deux valeurs: **EvenOddRule** et **WindingRule**. EvenOddRule, littéralement règle

Pair/Impair, remplit les surfaces internes superposées un nombre impair de fois alors que la règle WindingRule remplit les surfaces internes quel que soit le nombre de superpositions.

L'attribut ***arc_mode*** concerne le remplissage des arcs. On pourra remplir l'arc à la corde (**ArcChord**) ou comme une part de gâteau (**ArcPieSlice**).

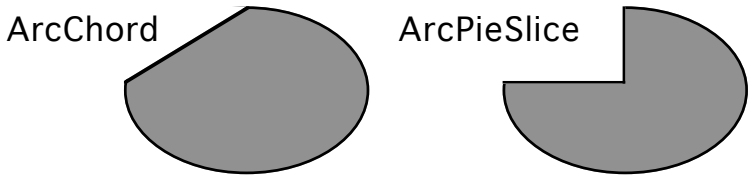

fig29. Modes des arcs d'ellipses

L'attribut ***font*** indique la fonte qui est utilisée par les primitives dessinant du texte.

L'attribut ***subwindow_mode*** indique si une requête effectuée sur une fenêtre doit également atteindre les fenêtres filles (**IncludeInferiors**) ou les ignorer (**ClipByChildren**).

Le booléen ***graphics_exposures*** permettra de recevoir les événements de type **GraphicExpose** et **NoExpose** après un appel à une fonction de transfert de dessin comme **XCopyArea** ou **XCopyPlane**. On pourra ainsi être averti du succès ou de l'échec de la copie. (Un booléen similaire est passé en argument à **XClearArea** de même un attribut booléen analogue est consulté dans la fenêtre pour l'exécution de **XClearWindow**).

Viennent ensuite trois attributs permettant de définir une zone restreignant les points affectés en dernière instance par la primitive de dessin. ***clip_x_origin*** et ***clip_y_origin*** indiquent l'origine de cette zone et ***clip_mask*** délimite la zone de dessin proprement dite.

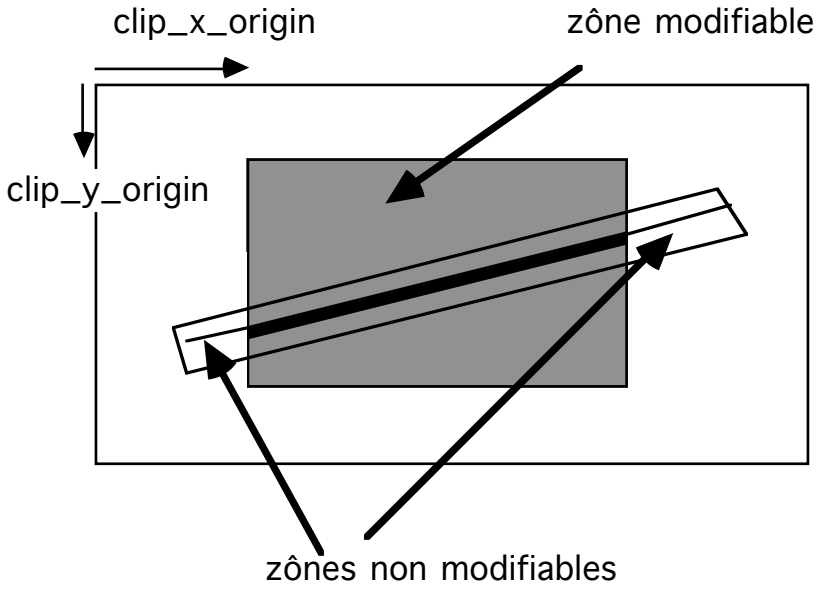

fig30. Effet du cadrage du dessin sur le tracé d'un segment

Manipulation de contextes graphiques

On pourra créer un contexte graphique grâce à la fonction **XCreateGC** (display, drawable, valuemask, &xgcvalues). L'argument drawable sert ici à récupérer des attributs liés au traitement de la couleur sur l'écran. On pourra en général passer la fenêtre racine. L'utilisation de valuemask et xgcvalues est analogue à celle de valuemask et xswa dans XCreateWindow. Ici encore, on passe l'adresse d'une structure dont certains champs ont été initialisés et un masque permettant d'identifier quels champs doivent être pris en considération dans la structure.

```
#define GCFunction          (1L<<0)
#define GCPlaneMask         (1L<<1)
#define GCForeground        (1L<<2)
#define GCBackground        (1L<<3)
#define GCLineWidth         (1L<<4)
#define GCLineStyle         (1L<<5)
#define GCCapStyle          (1L<<6)
#define GCJoinStyle         (1L<<7)
#define GCFillStyle         (1L<<8)
#define GCFillRule          (1L<<9)
#define GCTile              (1L<<10)
#define GCStipple           (1L<<11)
```

```
#define GCTileStipXOrigin   (1L<<12)
#define GCTileStipYOrigin   (1L<<13)
#define GCFont              (1L<<14)
#define GCSubwindowMode     (1L<<15)
#define GCGraphicsExposures (1L<<16)
#define GCClipXOrigin       (1L<<17)
#define GCClipYOrigin       (1L<<18)
#define GCClipMask          (1L<<19)
#define GCDashOffset        (1L<<20)
#define GCDashList          (1L<<21)
#define GCArcMode           (1L<<22)
```

fig31. Masques pour la définition des champs du contexte graphique

On peut également ne pas créer de contexte graphique et utiliser le contexte graphique par défaut en appelant la macro **DefaultGC**(dpy,screen).

On peut également copier certains attributs d'un contexte graphique dans un autre à l'aide de **XCopyGC** ou encore modifier un contexte graphique existant sur quelques attributs à l'aide de **XChangeGC**. Ces deux procédures utilisent une structure de type XGCValues et un masque. Signalons également l'existence de requêtes permettant de changer un contexte graphique sur un ou plusieurs attributs, sans passer par l'intermédiaire d'une structure de type XGCValues:

XSetState (dpy, gc, fg, bg, function, plane_mask)
XSetForeground (dpy, gc, fg)
XSetBackground (dpy, gc, bg)
XSetFunction (dpy, gc, function)
XSetPlaneMask (dpy, gc, planemask)
XSetLineAttributes (dpy, gc, lineWidth, lineStyle, capstyle, joinstyle)
XSetDashes(dpy,gc,dash_offset, dash_list, n)
XSetFillStyle(dpy, gc, fillStyle)
XSetFillRule(dpy, gc, fillRule)
XSetTile(dpy, gc, tile)
XSetStipple (dpy, gc, stipple)
XSetTSOrigin (dpy, gc, tsX, tsY)
XSetFont(dpy, gc, font)
XSetClipOrigin (dpy, gc, cX, cY)
XSetClipMask (dpy, gc, pixmap)
XSetClipRectangles (dpy, gc, cx, xy, rects, n, ordre)

XSetArcMode (dpy, gc, arcMode)
XSetSubwindowMode (dpy, gc, subwinMode)

On notera en particulier, **XSetDashes** pour définir des styles de pointillés complexes et **XSetClipRectangles** pour définir la zone de dessin effective comme réunion de rectangles (sans passer par l'intermédiaire d'un Pixmap, comme avec XSetClipMask). XSetClipRectangles est utilisée pour optimiser le réaffichage du dessin dans les événements de type Expose. Le principe est le suivant: tant que le champ *count* de l'événement Expose n'est pas nul, on accumule les rectangles dans un tableau de XRectangle. Une fois le champ count à zéro, on réaffiche le contenu en utilisant un contexte graphique précédemment modifié par XSetClipRectangles avec les rectangles accumulés. On prendra soin ensuite de remettre à jour la zone du contexte graphique (avec XSetClipMask (dpy, gc, None)).

Le dessin des lignes et des figures

On dispose des fonctions suivantes pour dessiner (sur un objet de type *Drawable* noté d):

XDrawPoint(dpy,d,gc,x,y)
XDrawPoints(dpy,d,gc,points, npoints, mode)
XDrawLine(dpy,d,gc,x1,y1,x2,y2)
XDrawLines(dpy,d,gc,points, npoints, mode)
XDrawSegments(dpy,d,gc,segments, nsegments)
XDrawRectangle(dpy,d,gc,x,y,width,height)
XDrawRectangles(dpy,d,gc,rects, nrects)
XDrawArc(dpy,d,gc,x,y,width,height,ang1,ang2)
XDrawArcs(dpy,d,gc,x,y,arcs, narcs)

Les fonctions suivantes permettent de remplir des figures ou des régions:

XFillRectangle(dpy,d,gc,x,y,width,height)
XFillRectangles(dpy,d,gc,rects,nrects)
XFillArc(dpy,d,gc,x,y,width,height,ang1,ang2)
XFillArcs(dpy,d,gc,x,y,arcs,narcs)
XFillPolygon(dpy,d,gc,points,npoints,shape,mode)

Signalons également les fonctions permettant de repeindre le fond d'une fenêtre (dans la couleur ou le motif indiqué par les attributs correspondants):

XClearArea(pdy,win,x,y,width,height,exposures)
XClearWindow(dpy,win)

Les fonctions **XcopyArea** et **XCopyPlane** permettent de recopier partiellement le contenu d'une fenêtre ou d'un Pixmap vers une autre fenêtre ou un autre Pixmap.

Le dessin du texte

Le dessin de texte s'effectue à partir d'une des trois primitives suivantes:

XDrawString(dpy, d, gc, x, y, string, length)
XDrawImageString(dpy, d, gc, x, y, string, length)
XDrawText(dpy, d, gc, x, y, items, nitems)

Les deux premières permettent de dessiner une chaîne dans la fonte indiquée par le contexte graphique, avec ou sans fond autour des caractères. La dernière permet d'afficher une ligne de texte composée de plusieurs items pour lesquels la fonte peut varier. On passe en argument un tableau d'items formés à partir de la structure suivante:

```
typedef struct {
     char* chars: /*pointeur sur une chaîne */
     int nchars; /* la longueur de la chaîne */
     int delta; /* l'espacement avec l'item suiv. */
     Font font; /* la fonte à utiliser */
} XTextItem;
```

Pour savoir où afficher une ligne de texte, on aura besoin de récupérer les informations concernant la taille prise par le morceau de texte dans la fonte considérée. Pour cela on aura recours aux fonctions:

XTextWidth(fontstruct, string, lenght)
XTextExtents (fontstruct, string, nchars, direction, ascent, descent, size)
XQueryTextExtents (dpy, fid, string, nchars, direction, ascent, descent, size)

Le dessin d'un morceau de texte s'effectue toujours sur l'horizontale passant par le point d'origine indiqué dans la primitive de dessin. On appelle cette ligne **la ligne de base**. Selon la fonte, la *direction* peut varier (gauche à droite où l'inverse), ainsi que l'espacement vertical occupé par le dessin du texte au dessus (*ascent*) et en dessous (*descent*) de la ligne de base. La taille (*size*) désigne la longueur prise par le dessin.

Les fontes

Il y a plusieurs manières de parler des fontes. On prendra garde à utiliser l'argument du bon type dans les fonctions manipulant des fontes. Ainsi, une fonte peut être référencée par un nom (de type **char***), par un identificateur de

fonte (type **Font**) ou par l'intermédiaire d'une structure de donnée contenant des informations sur la fonte (le type **XFontStruct**). Les fonctions précédentes par exemple avaient besoin respectivement d'un pointeur sur une structure de type *XFontStruct* ou d'un identificateur de fonte (*Font*).

Les principaux champs de la structure XFontStruct (cf. figure) sont les suivants:

fid: l'identificateur de la fonte correspondante.

direction: sens dans laquelle le texte est dessiné.

ascent: espacement maximal occupé par les caractères de cette fonte au dessus de la ligne de base (celle sur laquelle on écrit).

descent: espacement minimal occupé par les caractères de cette fonte au dessous de la ligne de base.

min_bounds, ***max_bounds*** qui indiquent les dimensions minimales et maximales des caractères de la fonte.

```
typedef struct {
   XExtData     *ext_data;                    /* pour réaliser des  (extension) */
   Font         fid;                          /* l'identificateur de la fonte */
   unsigned     direction;                    /* direction selon laquelle écrit cette fonte */
   unsigned     min_char_or_byte2;            /* premier caractère */
   unsigned     max_char_or_byte2;            /* dernier caractère */
   unsigned     min_byte1;                    /* première rangée existante */
   unsigned     max_byte1;                    /* dernière rangée existante */
   Bool         all_chars_exist;              /* True si tous sont de taille non nulle */
   unsigned     default_char;                 /* caractère à imprimer pour les indéfinis */
   int          n_properties;                 /* nombre de propriétés associées */
   XFontProp    *properties;                  /* pointeur vers le tableau des propriétés */
   XCharStruct min_bounds; /* minimum des dimensions de tous les caractères */
   XCharStruct max_bounds;/* maximum des dimensions de tous les caractères */
   XCharStruct *per_char;                    /* pointeur sur la table des info par caractère */
   int          ascent;                      /* espace maximum au dessus de la ligne de base */
   int          descent; /* espace minimum au dessous de la ligne de base */
} XFontStruct;
```

fig13. Les informations relatives à une police de caractères

La structure XFontStruct contient également un tableau comportant les informations relatives à chaque caractère de la fonte (tableau de structures **XCharStruct**) si leurs dimensions sont susceptibles de varier (cas des fontes dites *proportionnelles*). Une structure **XCharStruct** comporte les champs suivants:

```
typedef struct {
    short           lbearing;   /* distance (origine, pt de gauche) */
    short           rbearing;   /* distance (origine, pt de droite) */
    short           width;/* avancée jusqu'à l'origine suivante */
    short           ascent;     /* distance (ligne de base, sommet) */
    short           descent;    /* distance (ligne de base, pied) */
    unsigned short  attributes;
} XCharStruct;
```

fig13. La structure XCharStruct

lbearing: indique le distance de l'origine du caractère au point le plus à gauche
rbearing: indique la distance de l'origine au point le plus à droite.
width: indique la distance de l'origine du caractère à l'origine du caractère suivant. Cette distance est constante dans les fontes non proportionnelles.
ascent: est la hauteur du point le plus haut du caractère par rapport à la ligne de base.
descent: est la distance du point le plus bas du caractère par rapport à la ligne de base.

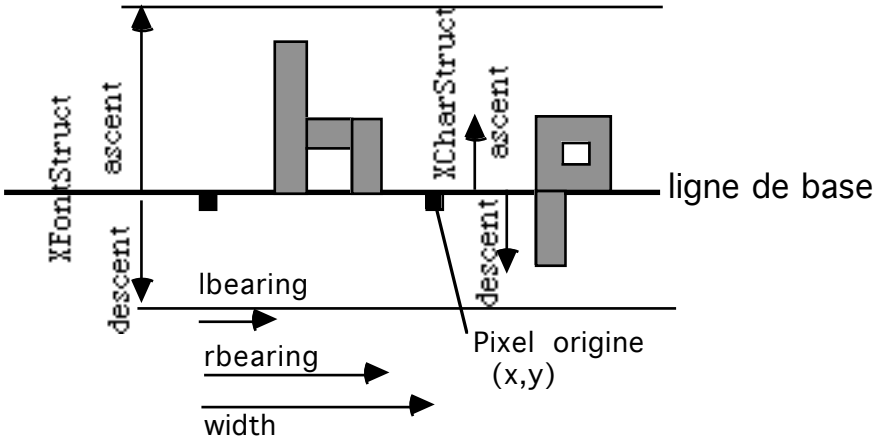

fig32. Les dimensions des caractères et de la fonte

On ne consultera que si nécessaire ces informations au niveau d'un caractère car on dispose de toutes façons des minima et maxima (sur tous les caractères de la fonte) dans les champs **min_bounds** et **max_bounds** de la structure XFontStruct.

Chargement et libération des fontes:

Les fontes (ou polices de caractères) sont stockées dans des fichiers. On peut demander au serveur de les charger à l'aide des fonctions suivantes:

Font **XLoadFont**(dpy,fontname)
XFontStruct* **XQueryFont**(dpy, fid)
XFontStruct* **XLoadQueryFont**(dpy,fontname)

et les libérer avec:

XFreeFont(dpy,fontstruct)
XUnloadFont(dpy,fid)

Sous shell, on peut obtenir la liste des fontes disponibles avec la fonction **xlsfonts**. On pourra les visualiser à l'écran grâce à l'utilitaire **xfd** (X Font Description). On peut modifier le chemin d'accès aux répertoires de fontes avec la fonction **xset** (option +fp et -fp). Ces opérations peuvent également être effectuées dans le programme, grâce aux fonctions suivantes:

XListFonts(dpy,pattern,nmax,n_return)
XSetFontPath(dpy, directories, ndirs)
XGetFontPath(dpy, ndirs)

On peut également charger des fontes avec:

XListFontsWithInfo(dpy,pattern,nmax,n_return, info_return)

Pour libérer les fontes ainsi chargées, on utilisera:

XFreeFontInfo(names,free_info, ninfo)
XFreeFontNames(list)

Les curseurs

Un curseur est défini par la donnée d'un motif rectangulaire bicolore (appelé Pixmap *source*), d'un masque de même dimension (Pixmap plan *mask*) indiquant quels points doivent être pris en considération dans le Pixmap source, et d'un point de référence (appelé *hot spot*) qui sert à indiquer la position de la souris.

L'intérêt du masque est de permettre un découpage plus fin du dessin. On peut ainsi avoir des curseurs non rectangulaires. On peut créer un curseur à partir de Pixmaps grâce à la fonction suivante: **XCreatePixmapCursor**(dpy, source, mask, fg, bg, x_hot, y_hot)

Attention: 1. Les arguments fg et bg ici ne sont pas des pixels, mais des pointeurs sur des structures **XColor**.
2. Le curseur se déplaçant fréquemment sur l'écran, le serveur en gère lui-même l'affichage. Il doit sauver en permanence le contenu de l'écran situé sous le

curseur. Sa capacité de sauvegarde variant avec les machines, il est préférable de s'enquérir de la taille favorite de curseur du serveur à l'aide de la fonction **XQueryBestCursor**(dpy,d,w,h,pw,ph).

Pour créer un curseur, on utilisera en fait généralement la fonte de curseurs (nom *cursor*), qui permet de charger un curseur à partir d'un nom de forme. Les noms utilisables se trouvent définis dans cursorfont.h. et la fonction s'appelle **XCreateFontCursor**(dpy,forme). Pour visualiser les curseurs à l'écran, on pourra utiliser l'utilitaire **xfd**.

Les couleurs d'un curseur peuvent être modifiées grâce à **XRecolorCursor**(dpy, cursor, fg, bg).

Pour associer un curseur à une fenêtre, on utilisera la fonction **XDefineCursor**(dpy,win,cursor) et pour revenir au curseur précédent **XUndefineCursor**(dpy,win). Pour libérer les ressources utilisées par un curseur, on appellera **XFreeCursor**(dpy,cursor).

Images et Régions

Signalons pour le principe l'existence d'autres structures que les fenêtres ou Pixmap pour stocker des dessins: les images (type **XImage**). Les fonctions sur les images permettent de modifier la valeur d'un pixel point par point. Une structure image contient toutes les données nécessaires à la compréhension de son format. Il vaut mieux cependant dans la pratique réécrire les fonctions d'accès aux images si leur format est connu, car les fonctions standards étant prévues pour un format quelconque, elles sont très lentes. Un champ est prévu à cet effet dans la structure XImage.

Les régions (type **XRegion**) sont à la base de l'implémentation des fenêtres et les fonctions opérant sur les régions sont très rapides. On utilisera les régions pour définir des zones sensibles non rectangulaires et non connexes. Les régions possèdent en effet des opérateurs ensemblistes intéressants (réunion, intersection, etc.) et de nombreux opérateurs booléens permettant de savoir si un point ou un rectangle se trouve ou non dans une région, etc. Les fonctions disponibles sur les régions sont les suivantes:

Region **XCreateRegion**()
Region **XPolygonRegion**(points, npoints, fill_rule)
XDestroyRegion(region)
XSetRegion(dpy, gc, region)
XClipBox(region, rect)
XOffsetRegion(region,dx,dy)
XShrinkRegion(region, dx, dy)

XIntersectRegion(r1,r2, rdest)
XUnionRegion(r1,r2, rdest)
XXorRegion(r1,r2, rdest)
XUnionRectWithRegion(rect, rsrc, rdest)
Bool **XEmptyRegion**(region)
Bool **XEqualRegion**(r1, r2)
Bool **XPointInRegion**(region, x, y)
int **XRectInRegion**(region, x, y, w, h)

La Couleur

Le traitement de la couleur

La couleur est constituée à l'écran par la composition de trois faisceaux de phosphore : un rouge, un vert et un bleu (*red*, *green*, *blue*). Ces couleurs sont dites fondamentales car elles permettent, par mélange, d'obtenir toutes les autres[18]. Si l'on produit l'intensité maximale pour chacun des faisceaux, on obtient du blanc. Si l'on combine des intensités moyennes et égales pour les trois faisceaux, on obtient divers niveaux de gris, et le noir correspond aux trois intensités nulles. On dit que la couleur est produite par *synthèse additive* car il faut augmenter les intensités pour augmenter la coloration.

```
typedef struct {
  unsigned long pixel;/* indice dans la colormap */
  unsigned short red, green, blue;/* composantes */
  char flags;/* combinaison de DoRed DoGreen DoBlue */
  char pad;          /* caractère de complétion */
} XColor;
```

fig33. *La structure XColor*

Un *pixel* est un indice dans une table de couleurs (type `Colormap`). Une `Colormap` est un tableau d'entrées fournissant dans trois *cellules*, les valeurs respectives des intensités des faisceaux de rouge, vert et bleu. Sur les machines à différents niveaux de gris, il n'y a qu'une cellule (celle correspondant au rouge) qui se trouve utilisée. On trouvera un champ `pixel` et un niveau d'intensité pour chacune des trois couleurs fondamentales dans la structure **XColor** utilisée pour manipuler la couleur dans la Release 4 (cf. figure 33).

On voit que la librairie a prévu de stocker les niveaux d'intensités dans cette structure sur des `short`. Un `short` étant codé sur 16 bits, on a virtuellement 65536 valeurs possibles pour chaque faisceau, soit $(65536)^3$ couleurs virtuellement différentes ! Si les pixels sont stockés sur 8 bits, on aura 256^3 couleurs possibles, soit plus de 16 millions de couleurs. En réalité, ni l'oeil, ni les machines ne peuvent distinguer tant de niveaux de couleurs et le nombre de couleurs possiblement rendues par une machine dépend en fait du nombre de niveaux distinguables par la machine. Soit Max le nombre de niveaux possibles pour chaque composante, on aura finalement Max^3 possibilités de couleurs différentes[19].

---

[18] et aucune d'entre elles ne peut être obtenue par le mélange des deux autres.

[19] Dans les `XColor`, on renormalisera ces niveaux de 0 à N* 65535/(Max - 1) avec N variant de 0 à Max-1.

Quoiqu'il en soit, le nombre de couleurs stockées dans une table de couleur est rarement égale au nombre de couleurs qui peuvent être rendues par une machine. Le nombre d'entrées dans une `Colormap` est le nombre de couleurs que l'on peut coder *simultanément* sur un écran. Ce nombre est en réalité fonction du *nombre de plans* de la mémoire écran et du *modèle* utilisé pour coder la couleur.

Dans le modèle illustré figure 34 un pixel est obtenu à partir des valeurs des bits dans les différents plans et ce pixel indexe *directement* une entrée dans une `Colormap`. Dans un tel modèle, un système comportant 4 plans ne peut indexer que 2^4 valeurs de cellules différentes (puisque les pixels sont stockés sur 4 bits). On aura donc seulement 16 couleurs différentes simultanément à l'écran[20]. Un système comportant 8 plans permet de stocker 256 couleurs et un système comportant 24 plans peut *a priori* coder plus de 16 millions de couleurs (2^{24} cellules).

Sur une machine de performance moyenne on a de 4 à 8 plans, ce qui conduit à des palettes de 16 à 256 couleurs. En outre chaque application peut définir une ou plusieurs palettes. On a vu en effet dans le chapitre sur les fenêtres que l'on pouvait associer une `Colormap` à chaque fenêtre.

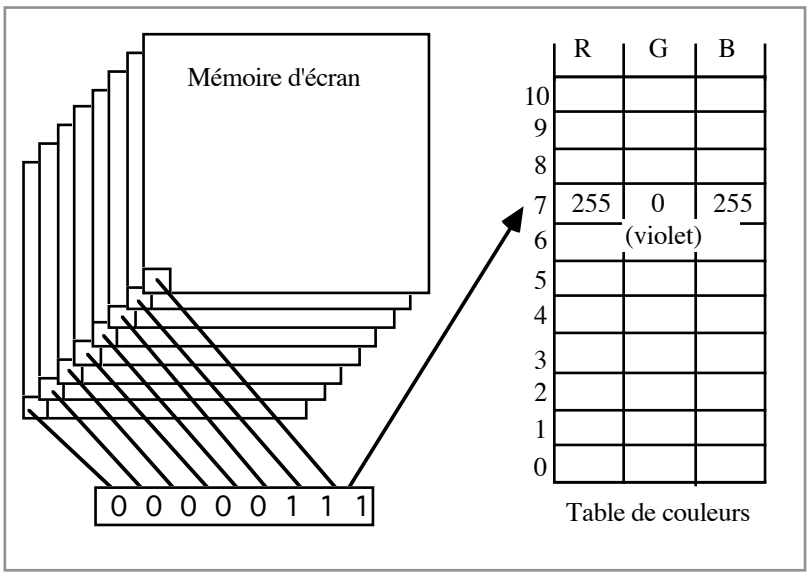

fig34. *Le modèle à indexation directe*

[20] qui pourront cependant être prises, parmi Max^3 possibilités, où Max est le nombre de niveaux d'intensités distinguables.

C'est le *window manager* qui est chargé d'installer la ou les tables de couleurs sur l'écran[22]. Si la machine ne permet pas d'installer plusieurs tables et que plusieurs clients ont créé leurs propres tables, le *window manager* installe à l'écran la table de la fenêtre qui a le focus des entrées. Dans la pratique, cela revient malheureusement à avoir les couleurs souhaitées par l'application ayant le focus et de "mauvaises" couleurs pour les autres, puisque les valeurs stockées dans les pixels de la table installée risquent de ne pas correspondre à celles utilisées par les autres clients. L'inconvénient pour l'utilisateur, outre que les couleurs de certaines applications soient fausses, est que l'écran change sans arrêt d'apparence quand le focus est modifié.

C'est pourquoi l'utilisation de tables standards permettant aux différentes applications de partager des couleurs est vivement recommandé, même dans les modèles les plus performants. Les tables standards sont des tables dans lesquelles les liens entre les pixels et les valeurs stockées sont calculables. Ces tables proposent une gamme raisonnable de couleurs régulièrement réparties et doivent donc satisfaire la plupart des clients. En outre, on peut tirer avantage du caractère prévisible du codage pour modifier les couleurs sans consulter le serveur au préalable.

De manière générale, partager une table de couleur revient simplement à demander l'allocation de cellules en lecture. Idéalement un seul client, le *window manager*, installe une table de couleurs standard sur la racine. Les autres clients se contentent alors d'utiliser les couleurs présentes[23]. L'inconvénient d'une allocation restreinte à la lecture est que l'on doit entièrement redessiner une figure pour en modifier les couleurs. A l'inverse, l'allocation privée en écriture présente l'avantage de la rapidité, car il suffit de changer les valeurs stockées dans les cellules pour modifier tout l'écran simultanément, sans avoir

---

21 Sur les machines les plus performantes, on peut avoir plusieurs tables de couleurs installées sur différentes fenêtres simultanément actives à l'écran, mais ce n'est pas le cas courant. La fonction `XListInstalledColormaps` permet de récupérer des informations sur le nombre de `Colormap` installées dans le *Hardware*.

22 Avant la *Release* 4, les *window manager* ne se chargeaient pas toujours d'installer les `Colormap` des fenêtres. A partir de cette version cependant, la convention utilisée par X est que les clients ne doivent pas installer eux-mêmes de tables de couleur. Les fonctions utilisées par les *window mabager* pour installer les tables de couleurs sont `XInstallColormap` et `XUninstallColormap`.

23 Il est en effet toujours possible de demander une allocation en lecture simple, même dans une table de couleurs autorisant l'écriture. Les tables standards offrent également parfois quelques cellules vierges que certains clients pourront s'allouer.

à redessiner. On retiendra cependant que le partage de tables standards est préférable du fait des limitations de ressources du serveur.

Les Ecrans couleur : indexation directe ou séparée

Selon les machines, le traitement de la couleur varie. Il existe deux grands modèles d'organisation des tables de couleurs. Le premier modèle dit *à indexation directe* spécifie directement dans la table pour un pixel donné les intensités de Rouge, Vert et Bleu qui lui correspondent (cf. figure avec 8 plans). Dans ce modèle, il y a autant d'entrées dans la table que de couleurs possibles simultanément à l'écran (il faut donc des tables de Max^3 entrées où Max est le nombre de niveaux d'intensités possibles). Pour 256 niveaux d'intensités, cela conduirait à une table de plus de 16 millions d'entrées.

Dans un modèle *à indexation séparée*, la couleur d'un point est déterminée par un pixel décomposé cette fois en trois entiers. Un pixel spécifie, sur trois séries de plans, trois indices dans une table permettant d'indexer les trois niveaux d'intensités fondamentaux. Par exemple figure 35, les 24 plans ont été partagés en trois séries de 8 plans. La première série fournit un index permettant de déterminer le niveau de rouge, la deuxième série fournit un index permettant de déterminer le niveau de vert, et la troisième série fournit un index permettant de déterminer le niveau de bleu.

En outre, que l'on soit dans un modèle à indexation directe ou à indexation séparée, on distingue encore les machines selon que l'on peut ou non écrire dans une table de couleur. Le type de traitement de la couleur est spécifié dans la structure display. Il est caractérisé par la classe du type Visual.

Le traitement de la couleur varie donc beaucoup selon les machines. Il est en effet fonction du nombre de plans et de la possibilité d'écrire ou non dans la table des couleurs. Dans un modèle à *indexation directe*, il y a autant d'entrées dans la table que de couleurs codables simultanément à l'écran. Pour 8 plans de mémoire, cela conduit, on l'a vu, à des tables de 256 couleurs.

L'indexation séparée est particulièrement économique puisque le nombre d'entrées dans la table diminue exponentiellement : cette fois, si Max est le nombre de niveaux distinguables par la machine pour chacune des trois couleurs, Max entrées suffisent à coder toutes les couleurs possibles.

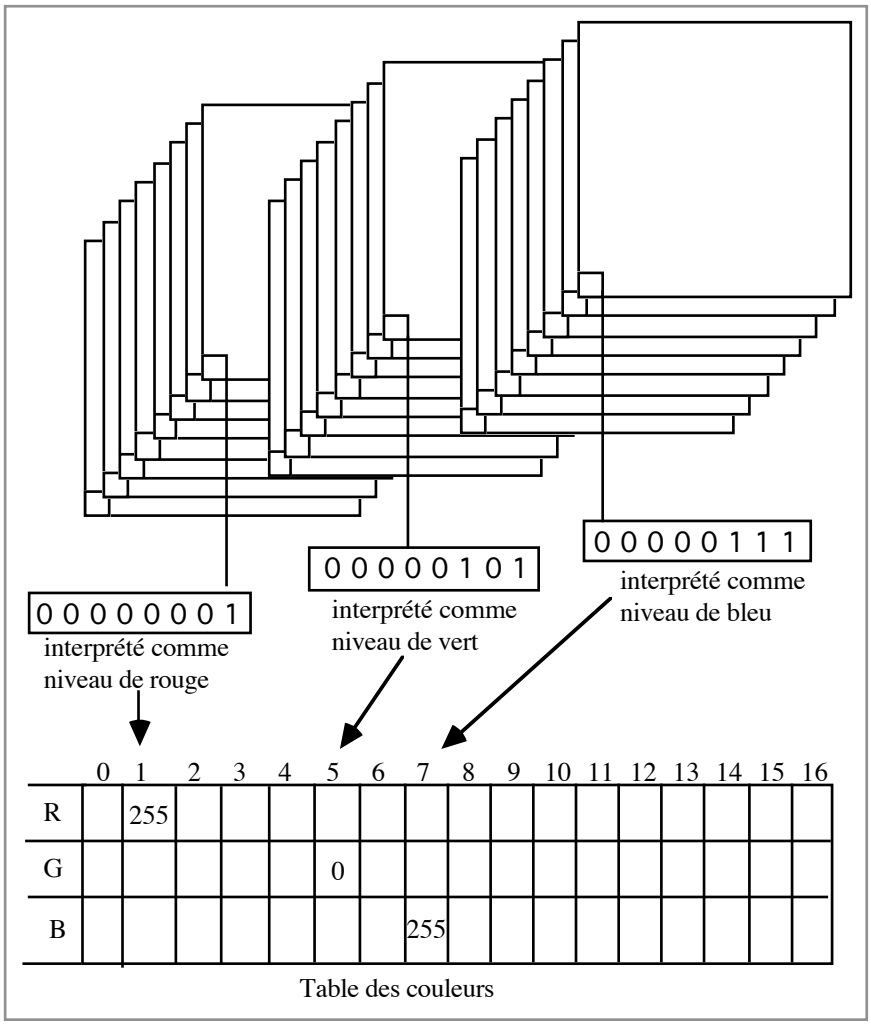

fig35. *Le modèle à indexation séparée*

Avec 8 plans seulement, on peut faire de l'indexation séparée en réservant par exemple 3 plans au codage du rouge, 3 plans au codage du vert et 2 plans à celui du bleu. Ce système est alors appelé modèle 3/3/2 et il existe une table standard pour ce modèle. Dans ce modèle, on a 8 niveaux de rouge (codés de 0 à 7 sur 3 bits), 8 niveaux de vert et 4 niveaux de bleu. On peut alors avoir une table de seulement 8 entrées, permettant de définir 256 couleurs différentes (8x8x4).

Le type visuel

La structure `Visual` permet de caractériser le type de traitement de la couleur. La classe d'un **Visual** indique si le modèle est à indexation directe ou à indexation séparée et si l'on peut ou non écrire dans la table des couleurs. Il y a quatre types de visuel couleur (DirectColor, TrueColor, PseudoColor et StaticColor) et deux types de visuels gris (GrayScale et StaticGray). Seuls les modèles `DirectColor` et `TrueColor` permettent de faire de l'indexation séparée. Les classes **DirectColor**, **PseudoColor** et **GrayScale** permettent d'écrire dans les tables de couleurs alors que les classes **TrueColor**, **StaticColor** et **StaticGray** supportent une table qui n'est accessible qu'en lecture.

Le modèle le plus général correspond aux visuels de classe DirectColor. On peut alors lire et écrire dans la table de couleurs et y faire de l'indexation séparée. Les machines supportant ce modèle supportent en réalité tous les autres. De manière générale, les modèles les plus performants peuvent supporter les modèles les moins performants et l'on a les inclusions de modèles illustrées figure 36.

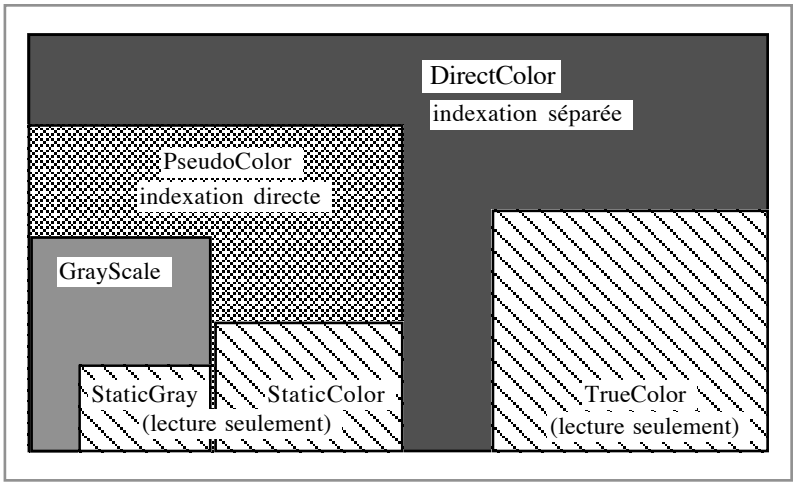

fig36. *Les inclusions de classes de Visual*

Le type de traitement de la couleur doit être spécifié à la création de chaque `Colormap` par une structure de type `Visual`. On doit également indiquer une structure `Visual` à la création de la fenêtre qui utilisera cette `Colormap`. Ces deux `Visual` doivent être les mêmes où des erreurs se

produiront à l'exécution. Pour utiliser un type Visual, on pourra se servir des fonctions:

DefaultVisual(dpy, screen)
Status **XMatchVisualInfo** (dpy, screen, depth, class,
&vinfo_return)
XVisualInfo * **XGetVisualInfo** (dpy, vinfo_mask,
&vinfotemplate, &nitems_return)

Une fois trouvé un visuel de caractéristiques optimales, on pourra en passer l'adresse à XCreateColormap et à XCreateWindow (c'est le premier champ de la structure XVisualInfo). Si le type visuel ou les couleurs requises n'existent pas sur cet écran, on pourra toujours se rabattre sur une version en noir et blanc ou quitter le programme en avertissant l'utilisateur.

Le noir et blanc

Cette section est destinée à attirer l'attention des programmeurs qui ne possèdent pas de machine couleur sur les questions qui, bien que relevant de la couleur, les concernent. En premier lieu, il important de noter qu'on ne connait pas *a priori* la valeur du pixel noir et celle du pixel blanc. C'est pourquoi on a utilisé dans les exemples des précédents chapitres les macros WhitePixel (dpy, screen) et BlackPixel (dpy, screen)

On sait simplement que sur une machine noir et blanc l'une de ces valeurs est 0 et l'autre 1. En conséquence, on peut peindre en *inverse video* sur ces machines en utilisant un contexte graphique dans lequel on a mis la fonction de transfert à GXxor (avec 2 x).

Par contre, si l'on dessine en noir et blanc sur une machine couleur, les valeurs du noir et du blanc peuvent être quelconques et rien ne garantit que le ou exclusif WhitePixel ^ BlackPixel ne fournisse pas une troisième couleur. Les programmes écrits pour du noir et blanc sans tenir compte de ce problème ne fonctionnent donc plus correctement quand ils sont portés sur une machine couleur.

Les solutions à ce problème sont :

• soit de mettre l'attribut foreground du contexte graphique au *ou exclusif* BlackPixel ^ WhitePixel et utiliser la fonction de transfert GXxor.

• soit de mettre la fonction de transfert à GXinvert et utiliser un masque des plans égal au *ou exclusif* WhitePixel ^ BlackPixel.

Ces deux méthodes se généralisent très bien à un affichage en inverse video couleur.

Un développeur peut aussi avoir besoin sur machines noir et blanc d'utiliser quelques fonctions de traitement de la couleur. Par exemple pour créer un curseur avec XCreatePixmapCursor on a besoin d'initialiser des structures XColor au noir et au blanc de la machine.

Pour connaître les valeurs d'intensités associées à un pixel donné on utilise généralement la fonction `XQueryColor` (dpy, cmap, &color_in_out).
On pourra appeler cette fonction avec la table de couleur par défaut (en utilisant la macro `DefaultColormap` (dpy, screen)) et passer l'adresse d'une variable de type `XColor` dans laquelle le champ *pixel* aura été spécifié (pour le noir ou blanc grâce aux macros `WhitePixel` et `Blackpixel`). La procédure remplit alors les valeurs des champs d'intensités RGB associées, d'où le nom de *color_in_out* de ce paramètre[24]. Comme on va le voir, cet usage des structures `XColor` comme paramètre d'entrée/sortie est fréquente dans toutes les procédures traitant de la couleur.

Allocation, Lecture et Ecriture des cellules

Il existe également des noms standards de couleurs. Pour lire des couleurs dans une colormap, on dispose des fonctions suivantes: **XQueryColor** et **XQueryColors** qui spécifient les pixels et reçoivent les RGB associés. Inversement **XParseColor** permet de récupérer un pixel à partir d'une spécification de couleur donnée soit par un nom standard, soit par un pattern du type #RGB, #RRGGBB ou #RRRGGGBBB dans lequel les lettres R,G et B désignent des nombres hexadécimaux qui forment des valeurs d'intensités dans les couleurs considérées.

Même si l'on ne compte utiliser des pixels qu'en lecture, on doit en déclarer la liste au serveur. Cette déclaration s'appelle l'*allocation de couleur*. La raison de cette déclaration est que les cellules allouées en lecture sont partagées entre tous les clients. Le serveur doit maintenir la liste des clients qui utilisent un pixel donné pour libérer les cellules quand aucun client n'aura plus l'intention d'utiliser ce pixel.

Dans les modèles où on peut écrire, on pourra s'allouer des couleurs grâce aux fonctions suivantes:
XAllocColor(dpy, cmap, color_in_out) alloue un pixel.
XAllocNamedColor qui permet de s'allouer la couleur la plus proche d'une couleur donnée par son nom disponible sur cet écran et d'en obtenir la

[24] Le champ *pixel* étant initialisé, il s'agit bien d'un paramètre d'entrée, et les champs *red*, *green*, *blue* étant remplis en sortie de procédure, il s'agit également d'un paramètre de sortie.

définition exacte. **XLookupColor** qui ramène les mêmes résultats mais sans faire d'allocation.
XAllocColorCells (modèle à indexation directe) qui permet de s'allouer des cellules vierges (pixels) dans une table moyennant certaines contraintes sur la manière dont elles sont disposées. On peut en effet spécifier un nombre de plans tels que les pixels retournés et leurs paires masquables pour ces plans soient alloués.

Ainsi par exemple, si on demande deux pixels masquables par un plan
on aura en retour un tableau de 2 pixels et un masque de plan:
pixels[0] = ----------0-0----------
pixels[1] = ----------0-1----------
(où les autres bits sont quelconques) et
plane_mask[0] = ----------1----------
(tous les autres bits sont cette fois à zéro).

On pourra en fait utiliser les 4 pixels de la table:
pixels[0] = ----------0-0----------
pixels[1] = ----------0-1----------
et
pixel_2 = ----------1-0----------
pixel_3 = ----------1-1----------

en sachant que l'on peut passer de pixels[0] à pixel_2 simplement en faisant opérer le masque du plan:

pixel_2 = pixels[0] | planemask[0]

Cette fonction est prévue pour appliquer une technique dite de recouvrement (*overlay*) dans un modèle à indexation directe. Les pixels vierges étant ainsi allouer, on peut ensuite aller y placer les valeurs d'intensités (RGB) que l'on souhaite pour les apparier relativement aux masquages des plans. On pourra effectuer cette opération de stockage grâce aux fonctions: **XStoreColors** ou **XStoreColor** ou encore **XStoreNamedColor**.

Ainsi dans notre exemple, on pourra aller placer en pixels[0] et pixels[1] des intensités donnant la couleur du dessin (foreground) et celle du fond (background) et leur faire correspondre ensuite deux couleurs pour une mise en relief spéciale du motif. Pour faire apparaître le motif en relief, il suffira alors de le peindre dans un contexte graphique masqué par le plan de recouvrement, sans modifier pour autant les spécifications des attributs *foreground* et *background*. On utilisera également cette technique pour tracer un dessin dans une couleur variant selon le fond sur lequel il est dessiné.

XAllocColorPlanes permet d'appliquer la même technique dans un modèle à indexation séparée.

Pour libérer des cellules de couleurs, on utilisera la fonction **XFreeColors**.

Annexe : Index alphabétique des événements

Dans cette annexe extraite de mon livre X-Window, Manuel de Programmation aux éditons Eyrolles, les événements sont présentés par ordre alphabétique sur les types. Les masques permettant de sélectionner les événements (s'il en existe) sont indiqués en caractères machine en face du type considéré. Le type de la structure qui leur est associée figure en italique et son champ d'accès dans un événement event de type XEvent est indiqué en face sur la même ligne.

ButtonPress ButtonPressMask
ButtonRelease ButtonReleaseMask
type : XButtonEvent *event.xbutton*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* fenêtre à laquelle est rapporté l'ev. */
        Window root;            /* racine sur laquelle l'ev. s'est produit */
        Window subwindow;       /* fenêtre fille dans laquelle s'est produit l'ev. */
        Time time;              /* heure de l'ev. en millisecondes */
        int x, y;               /* coordonnées du pointeur relatives à window */
        int x_root, y_root;     /* coordonnées relatives à root */
        unsigned int state;     /* masque d'état des boutons et modifieurs  */
        unsigned int button;    /* numéro du bouton */
        Bool same_screen;       /* vrai si la souris est encore sur le même écran */
} XButtonEvent;
```

Un bouton de souris est enfoncé ou relâché. On sélectionne ces événements par ButtonPressMask et ButtonReleaseMask. Ces événements sont normalement envoyés à la fenêtre située sous le curseur de souris, sauf en cas de saisie (monopolisation) de la souris. La structure event.xbutton contient les informations suivantes : heure de l'événement, fenêtre recevant l'événement, état des modifieurs, numéro du bouton enfoncé, position du pointeur relativement à la fenêtre ayant reçu l'événement et relativement à la fenêtre racine, etc. Le champ subwindow contient l'identificateur de la fenêtre dans laquelle s'est produit l'événement s'il a eu lieu dans une fenêtre fille de window (cf. chapitre 4, section 4.4. *La propagation des événements*).


L'événement ButtonPress déclenche automatiquement un monopole de la souris par la fenêtre qui a récupéré cet événement. Cette fenêtre sera la seule à recevoir des événements souris *jusqu'à ce que tous les boutons de la souris soient relâchés*. Durant toute cette période, le curseur de souris garde la forme qui lui était attribuée dans la fenêtre accaparante (même s'il se trouve à l'extérieur de la fenêtre) et aucune fenêtre ne reçoit plus d'événement souris. La fenêtre accaparante, à l'inverse, reçoit les événements qu'elle a sélectionnés même s'ils se produisent dans une autre fenêtre.

On peut supprimer cette redirection automatique des événements vers la fenêtre accaparante *au niveau de l'application* et rendre les événements à la fenêtre de l'application située sous le curseur de souris en ajoutant le masque OwnerGrabButtonMask lors de la sélection de l'événement ButtonPress[25]. Dans ce cas, les événements souris sont envoyés à la fenêtre située sous le curseur de souris, pourvu qu'elle appartienne à l'application propriétaire de la saisie et qu'elle ait sélectionné ces événements. Pour plus d'informations sur ces événements voir le chapitre 5, en particulier les sections 5.1 et 5.2.

[25] Il y a un deuxième cas d'exception : si l'application avait invoqué un monopole passif du bouton sur une fenêtre ancêtre de la fenêtre dans laquelle s'est produit l'événement `ButtonPress`.

CirculateNotify StructureNotifyMask
SubstructureNotifyMask
type : XCirculateEvent *event.xcirculate*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* la fenêtre recevant l'ev. */
        Window window;          /* la fenêtre réempilée */
        int place;              /* PlaceOnTop, PlaceOnBottom */
} XCirculateEvent;
```

Une fenêtre a été déplacée par suite d'un appel d'un client à XCirculateWindowUp ou XCirculateWindowDown. Si le *window manager* a empêché cette opération, l'événement ne sera pas envoyé. Cet événement indique l'identificateur de la fenêtre qui a été empilée dans le champ window, et la fenêtre qui a reçu l'événement dans le champ event. L'entier place indique si la fenêtre a été placée en tête ou en queue de file.

CirculateRequest SubstructureRedirectMask
type : XCirculateRequestEvent *event.xcirculaterequest*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window parent;                  /* le parent de la fenêtre ré-empilée */
        Window window;          /* la fenêtre empilée */
        int place;              /* PlaceOnTop, PlaceOnBottom */
} XCirculateRequestEvent;
```

L'événement CirculateRequest rapporte qu'un client a fait appel à une requête d'empilement de sous-fenêtres comme XCirculateSubwindowsDown ou Up, ou XCirculateSubwindows. Quand un tel événement est reçu, la requête d'empilement n'a pas été traitée par le serveur, ce qui donne au parent qui a sélectionné l'événement (en général, le *window manager*) l'opportunité de reformuler la requête (cf. chapitre 1, la redirection des requêtes).

ClientMessage
type : XClientMessageEvent *event.xclient*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* fenêtre à qui l'ev. a été envoyé */
        Atom message_type;      /* type de la propriété envoyée */
        int format;             /* format de la propriété */
        union {
                char b [20];
                short s [10];
                long l [5];
                } data;         /* les données elles-mêmes */
} XClientMessageEvent;
```

L'événement ClientMessage est envoyé directement par un client à une fenêtre quelconque avec la fonction XSendEvent. N'importe quel type d'événement peut être envoyé par cette requête. La fenêtre cible a pu être désignée par PointerWindow (fenêtre contenant le pointeur de souris) ou InputFocus (fenêtre possédant le focus).

Le champ message_type contient un atome qui spécifie une propriété et indique comment les données data doivent être interprétées par le client. Cette interprétation se fera suivant un protocole d'accord entre les clients. On peut imaginer par exemple que ce champ serve à indiquer une sorte de type secondaire pour l'événement. Le champ format spécifie le format de la propriété spécifiée par message_type. Il peut prendre les valeurs 8, 16 ou 32. Le serveur n'interprète pas les données mais impose qu'elles se présentent sous la forme d'une liste d'éléments de 8, 16 ou 32 bits. Les données se présentent toujours sous la forme d'une liste de vingt données dans le format 8-bits, dix données dans le format 16-bits et cinq dans le format 32-bits, que tout l'espace mémoire soit ou non utilisé.

Si votre application souhaite communiquer avec d'autres applications qu'elle-même (ou une autre occurrence d'elle-même), consultez la documentation sur les ICCCM pour connaître les protocoles les plus standards.

ColormapNotify ColormapChangeMask
type : XColormapEvent *event.xcolormap*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* celle dont l'attribut change */
        Colormap colormap;      /* une Colormap ou None */
        Bool new;
        int state;              /* ColormapInstalled, ColormapUninstalled */
} XColormapEvent;
```

Quelqu'un a modifié la table de couleurs standard ou l'attribut colormap de la fenêtre. S'il n'y a plus de table, le champ colormap est à None. Le champ new indique par True que l'attribut de la fenêtre a changé, et par False que la table a été installée ou désinstallée. Le champ state indique si la table spécifiée dans le champ *colormap* est ou n'est pas installée.

ConfigureNotify StructureNotifyMask
SubstructureNotifyMask

type : XConfigureEvent *event.xconfigure*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* celle qui reçoit l'ev. */
        Window window;          /* celle qui est modifiée */
        int x, y;               /* les nouvelles composantes relatives au parent */
        int width, height;      /* les nouvelles dimensions */
        int border_width;
        Window above;                   /* la sœur juste dessous */
        Bool override_redirect;         /* l'attribut de la fenêtre modifiée */
} XConfigureEvent;
```

Cet événement annonce tout changement survenu dans la configuration géométrique d'une fenêtre (taille, position, bord, ordre d'empilement). On le sélectionne avec le masque StructureNotifyMask. Pour le recevoir pour tous les enfants d'une fenêtre, il suffit de sélectionner SubstructureNotifyMask sur la fenêtre parent. Le champ above précise la fenêtre devant laquelle se trouve la fenêtre reconfigurée dans la pile de ses sœurs. S'il est à None, c'est que la fenêtre se trouve en fond de pile.

ConfigureRequest SubstructureRedirectMask
type : XConfigureRequestEvent *event.xconfigurerequest*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window parent;                  /* celle qui a sélectionné l'ev. */
        Window window;          /* la fenêtre à reconfigurer */
        int x, y;               /* les arguments de la requête */
        int width, height;
        int border_width;
        Window above;
        int detail;             /* Above, Below, TopIf, BottomIf, Opposite */
        unsigned long value_mask;
} XConfigureRequestEvent;
```

L'événement ConfigureRequest informe qu'un client a essayé de modifier la taille, la position, le bord et/ou le rang d'empilement de la fenêtre window par une requête (comme XConfigureWindow). La requête n'a cependant pas été satisfaite et ce type d'événement est normalement sélectionné par le *window manager*. C'est lui qui décide de reformuler la requête ou de l'ignorer. Le champ above indique la sœur devant laquelle la fenêtre à reconfigurer souhaite être placée. S'il vaut None, la fenêtre devrait être placée en fond de pile.

CreateNotify SubstructureNotifyMask
type : XCreateWindowEvent *event.xcreatewindow*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window parent;                  /* parent de la fenêtre */
        Window window;          /* identificateur de la fenêtre créée */
        int x, y;               /* position de la fenêtre */
        int width, height;      /* taille de la fenêtre */
        int border_width;       /* épaisseur du bord */
        Bool override_redirect;         /* indique si la création devrait être ignorée du
                                        window manager*/
} XCreateWindowEvent;
```

L'événement CreateNotify informe le client qui le souhaite (a priori le *window manager*) de la création d'une fenêtre fille. Cet événement est sélectionné par SubstructureNotifyMask sur la fenêtre parent (habituellement la racine de l'écran). Si l'attribut override_redirect est à True, le *window manager* doit normalement ignorer cet événement. Sinon, il récupère l'identificateur de la fenêtre créée et y sélectionne les événements qui l'intéressent (par exemple ConfigureRequest).

DestroyNotify SubstructureNotifyMask
type : XDestroyWindowEvent *event.xdestroywindow*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* fenêtre qui a selectionné l'ev. */
        Window window;          /* la fenêtre détruite */
} XDestroyWindowEvent;
```

L'événement DestroyNotify informe de la destruction d'une fenêtre. Cet événement est sélectionné par SubstructureNotifyMask sur la fenêtre parent.

EnterNotify EnterWindowMask
LeaveNotify LeaveWindowMask
type : XCrossingEvent *event.xcrossing*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* la fenêtre à laquelle est rapportée l'ev. */
        Window root;            /* racine sur laquelle l'ev. s'est produit */
        Window subwindow;       /* fenêtre fille */
        Time time;              /* millisecondes */
        int x, y;               /* coordonnées du pointeur relatives à window */
        int x_root, y_root;     /* coordonnées relatives à root */
        int mode;               /* NotifyNormal, NotifyGrab, NotifyUngrab */
        int detail;             /* NotifyAncestor, NotifyVirtual, NotifyInferior,
                                 * NotifyNonlinear, NotifyNonlinearVirtual */
        Bool same_screen;       /* si le pointeur n'a pas changé d'écran */
        Bool focus;             /* si la fenêtre a le focus */
        unsigned int state;     /* masque d'état des boutons + modifieurs */
} XCrossingEvent;
```

La souris est entrée ou sortie d'une fenêtre. Ces événements se produisent lorsque l'on déplace la souris d'une fenêtre à l'autre ou lorsque l'on affiche (ou retire) une fenêtre sous la souris. La fenêtre qui reçoit EnterNotify est avertie qu'elle peut recevoir ensuite des événements souris. Normalement, elle reçoit aussi le focus du clavier, sauf en cas de saisie du clavier ou du focus. Le champ focus de l'événement permet de savoir si la fenêtre bénéficie du focus du clavier. En outre, des événements EnterNotify et LeaveNotify sont envoyés aux fenêtres qui sont traversées *virtuellement*. Ces fenêtres sont celles qui se trouvent entre la fenêtre d'origine et la fenêtre de destination dans la hiérarchie des fenêtres. (En effet, les ancêtres des fenêtres concernées par les événements d'entrée/sortie sont elles aussi concernées par ces événements, car elles peuvent normalement recevoir les événements souris par propagation.) Dans ce cas, le champ detail de la structure event.xcrossing contient la valeur NotifyVirtual.

Des événements de type EnterNotify et LeaveNotify sont également envoyés quand la souris est saisie et que le pointeur ne se trouvait pas déjà dans la fenêtre accaparante. Dans ce cas, la fenêtre accaparante reçoit un EnterNotify qui lui indique qu'elle va recevoir les événements souris et la fenêtre dans laquelle se trouvait le pointeur reçoit un LeaveNotify. La position du pointeur dans les deux événements est

celle qui précédait au déclenchement de la saisie. Quand le monopole se termine, on a émission d'événements symétriques indiquant la fin du processus d'appropriation.

Pour une description complète de ces événements, voir chapitre 5 la section 5.5. qui leur est consacrée.

Expose ExposureMask
type : XExposeEvent *event.xexpose*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* la fenêtre à réafficher */
        int x, y;               /* la zone à réafficher */
        int width, height;
        int count;              /* nb d'ev. du même type encore à venir */
} XExposeEvent;
```

Si l'application effectue des dessins à l'écran, ces dessins devront être tracés sur la réception d'un événement de type Expose. Un événement de type Expose est envoyé quand une fenêtre affichée devient visible ou quand une partie précédemment invisible devient visible. C'est l'indication pour l'application qu'elle peut dessiner ou redessiner le contenu de la fenêtre ayant reçu l'événement, dans la zone indiquée par l'événement. Cet événement est sélectionné avec ExposureMask et ne concerne que les fenêtres InputOutput. La structure associée contient la géométrie du rectangle devenu visible et le champ count indique le nombre des autres événements de type Expose encore en attente de réception.

Pour optimiser l'affichage, il est recommandé de ne réafficher les dessins que dans la zone indiquée par l'événement Expose en affectant la *zone de clipping* du contexte graphique passé en argument à la requête de dessin. Pour optimiser encore davantage, on peut ne dessiner que sur réception d'un événement Expose dont le champ count est à zéro. Dans ce cas, il faut stocker les différents rectangles exposés dans un tableau de XRectangle (tant que le champ count n'est pas à zéro) et dessiner ensuite avec un contexte graphique dont la *zone de clipping* aura été affectée par la réunion des différents rectangles exposés grâce à la fonction XSetClipRectangles). Pour plus de précision sur ces événements, se reporter à la section 6.5.

FocusIn FocusChangeMask
FocusOut
type : XFocusChangeEvent *event.xfocus*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* fenêtre de l'événement */
        int mode;               /* NotifyNormal, NotifyGrab, NotifyUngrab */
        int detail;             /* NotifyAncestor, NotifyVirtual, NotifyInferior,
                                 * NotifyNonlinear, NotifyNonlinearVirtual,
                                 * NotifyPointer, NotifyPointerRoot,
                                 * NotifyDetailNone  */
} XFocusChangeEvent;
```

FocusIn informe l'application qu'elle a saisi le focus du clavier et FocusOut qu'elle l'a perdu. Ces événements sont envoyés quand le focus d'une fenêtre change après un appel à XSetInputFocus. Cette fonction permet de restreindre le focus à une fenêtre affichée et ses descendantes. (Pour revenir au comportement par défaut, on passera la valeur *None* à XSetInputFocus comme identificateur de fenêtre.) Ces événements sont analogues à EnterNotify et LeaveNotify mais ils concernent le clavier alors que EnterNotify et LeaveNotify concernent la souris. La fenêtre ayant le focus du clavier et ses descendantes sont les seules à pouvoir recevoir les événements clavier. (Par défaut c'est la fenêtre racine qui a le focus, ce qui a pour effet de transmettre le focus à toute fenêtre dans laquelle se trouve la souris.)

On peut également recevoir des événements de type Focus en cas d'appropriation du clavier. Le champ mode de la structure indiquera s'il s'agit d'événements simples, virtuels ou s'ils résultent d'une appropriation. Le champ detail indique la position exacte de la fenêtre par rapport aux fenêtres suivantes : origine (ancien focus), destination (nouveau focus) et fenêtre contenant la souris au moment du changement de focus. La section 5.6. de ce manuel décrit en détail ces événements.

GraphicsExpose
NoExpose
type : XGraphicsExposeEvent, *event.xgraphicsexpose*
XNoExposeEvent *event.xnoexpose*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Drawable drawable;      /* fenêtre InputOutput ou Pixmap */
        int x, y;               /* zone à réafficher */
        int width, height;
        int count;              /* nb d'ev. de même type encore à venir */
        int major_code;                 /* X_CopyArea ou X_CopyPlane */
        int minor_code;                 /* indéfini */
} XGraphicsExposeEvent;
```

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Drawable drawable;
        int major_code;                 /* X_CopyArea ou X_CopyPlane */
        int minor_code;                 /* zéro, sauf extension */
} XNoExposeEvent;
```


On ne sélectionne pas ces événements par des masques. Pour recevoir ces événements, il faut avoir positionné l'attribut graphic_exposures du contexte graphique à True dans la requête de copie (cf. section 6.4.). Pour tester le champ *major_code* il faut avoir inclu le fichier <X11/Xproto.h>.

Ces événements indiquent si la partie copiée par XCopyArea ou XCopyPlane était effectivement accessible dans la source au moment du traitement de la requête de transfert. Les événements de type GraphicsExpose avertissent de l'échec partiel de la copie quand il est dû à une indisponibilité de la source. En cas d'échec, le serveur envoie un ou plusieurs événements de type GraphicsExpose pour indiquer les zones qui devront être réexposées à la requête de copie.

L'événement de type NoExpose signifie qu'aucun événement de type GraphicsExpose n'est envoyé. Cela peut donc signifier que la source était entièrement disponible (et que la requête a été exécutée avec succès), ou bien que la destination était

indisponible (dans ce cas le serveur considère que l'état de la source n'importe pas bien qu'en un certain sens il y ait eu échec total du transfert).

GravityNotify StructureNotifyMask
SubstructureNotifyMask
type : XGravityEvent *event.xgravity*

```
typedef struct {
        int type;
        unsigned long serial;  /* n° de la dernière requête traitée par le serveur */
        Bool send_event;       /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;      /* terminal sur lequel l'ev. s'est produit */
        Window event;                  /* la fenêtre qui a sélectionné l'ev. */
        Window window;         /* la fenêtre qui a bougé */
        int x, y;              /* nouvelle position relative au parent */
} XGravityEvent;
```

Un événement de type GravityNotify est émis quand un déplacement de fenêtre est causé par un changement de taille de son parent (cf. section 3.4. *Attributs pour les changements de taille des fenêtres*).

KeymapNotify KeymapStateMask
type : XKeymapEvent *event.xkeymap*

```
typedef struct {
        int type;              /* type de l'événement */
        unsigned long serial;  /* n° de la dernière requête traitée par le serveur */
        Bool send_event;       /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;      /* terminal sur lequel l'ev. s'est produit */
        Window window;         /* celle qui a reçu FocusIn ou EnterNotify */
        char key_vector [32]; /* l'état des touches */
} XKeymapEvent;
```

Cet événement est envoyé pour permettre aux applications qui le souhaitent de récupérer l'état du clavier avant de lire des entrées. Cet événement est émis juste après un EnterNotify ou un FocusIn — ces deux types d'événements étant susceptibles d'indiquer à l'application qu'elle va recevoir des entrées.


KeymapNotify ne garantit pas que la fenêtre qui le reçoit ait le focus mais

donne l'état initial de toutes les touches (cf. section 5.6). Le champ key_vector donne les 256 bits correspondant aux diverses touches. Une touche donnée a pour keycode la position de son bit dans le key_vector. On peut également lire ce vecteur en appelant XQueryKeymap (mais les performances sont moins bonnes). En fait, cet événement est peu utilisé car l'état des modifieurs figure dans l'événement rapportant l'enfoncements d'une touche.

KeyPress KeyPressMask
KeyRelease KeyReleaseMask
type : XKeyEvent *event.xkey*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* fenêtre à laquelle est rapporté l'ev. */
        Window root;            /* racine sur laquelle l'ev. s'est produit */
        Window subwindow;       /* fenêtre fille */
        Time time;              /* millisecondes */
        int x, y;               /* coordonnées du pointeur relatives à window */
        int x_root, y_root;     /* coordonnées relatives à root */
        unsigned int state;     /* masque d'état des boutons et modifieurs */
        unsigned int keycode;   /* détail */
        Bool same_screen;       /* si le pointeur est encore sur le même écran */
} XKeyEvent;
```

Une touche du clavier est enfoncée ou relâchée. Pour recevoir ces événements il faut bénéficier du focus du clavier[26] et avoir sélectionné l'événement par KeyPressMask (ou KeyReleaseMask). La structure de l'événement contient le code émis par la touche dans le champ keycode.

L'événement permet en outre de connaître l'état des modifieurs (ce sont les touches *Shift*, *Control*, *MetaLeft*, *MetaRight, Compose* etc.) et l'état des boutons grâce à la valeur du masque fournie dans le champ state.

On interprète généralement le code émis par la touche à partir de l'événement

[26] Par défaut, le focus du clavier est attribué à la fenêtre qui contient la souris à un moment donné. Cependant, certains *window manager* autorisent l'utilisateur à préciser le focus avec la souris. Cette deuxième convention nécessite l'utilisation de la fonction `XSetInputFocus`.

grâce à la fonction XLookupString (XmbLookupString et XwcLookupString dans la *Release* 5). Ces fonctions permettent de récupérer à la fois le code symbolique associé à l'événement compte tenu de l'état des modifieurs, et la chaîne ASCII qui lui est associée, si elle existe[27]. (La fonction XKeycodeToKeysym permet également d'effectuer la conversion mais nécessite de tester l'état des modifieurs.)

 Pour reconnaître les codes symboliques et les constantes de retour fournies par XLookupString il faut inclure les fichiers <X11/keysym.h> et <X11/Xutil.h>.

LeaveNotify LeaveWindowMask
type : XCrossingEvent *event.xcrossing*

cf. la description de EnterNotify.

MapNotify StructureNotifyMask
UnmapNotify SubstructureNotifyMask
types : XMapEvent *event.xmap*
XUnmapEvent *event.xunmap*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* celle qui a sélectionné l'ev. */
        Window window;          /* fenêtre affichée */
        Bool override_redirect;         /* valeur de l'attribut de window */
} XMapEvent;

typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* celle qui a reçu l'ev. */
        Window window;          /* fenêtre supprimée de l'affichage */
        Bool from_configure;/* pour indiquer l'origine de l'ev. */
} XUnmapEvent;
```

[27] On peut modifier les caractères ASCII associés par `XLookupString` au code symbolique `keysym` grâce à la fonction `XRebindKeysym`. Cette redéfinition est locale au client.

Ces événements sont envoyés par le serveur quand une fenêtre passe de l'état Map à Unmap et vice versa. Le booléen from_configure permet de distinguer les événements de type UnmapNotify qui proviennent d'une requête de reconfiguration de ceux qui proviennent d'un changement de taille du parent quand la fenêtre a positionné l'attribut win_gravity à UnmapGravity (cf. la description de win_gravity dans le chapitre 3 sur les fenêtres).

MappingNotify
type : XMappingEvent *event.xmapping*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* non utilisé */
        int request;            /* MappingModifier, MappingKeyboard,
                                 * ou MappingPointer */
        int first_keycode;      /* premier code modifié s'il y en a */
        int count;              /* nb des codes modifiés */
} XMappingEvent;
```

L'événement MappingNotify informe qu'un changement d'association a été effectué par un autre client sur les liens concernant le clavier ou la souris. Cet événement peut provenir d'un appel à :

• XChangeKeyboardMapping, qui modifie les liens entre les codes physiques keycode et les codes symboliques keysym.

• XSetModifierMapping, qui modifie les liens entre les touches et les modifieurs logiques.

• XSetPointerMapping, qui modifie les liens entre les boutons physiques et les bouton logiques de la souris.

Le champ request de l'événement indique le type de modification effectuée. La réaction normale à un changement sur les touches (champ request à MappingKeyboard) est un appel à XRefreshKeyboardMapping pour remettre à jour les tables locales.

Une application ne doit normalement pas réagir aux autres requêtes (MappingPointer ou MappingModifier) car elle doit n'utiliser que les liens symboliques pour permettre à l'utilisateur de les changer au besoin. (Si cependant l'application

utilise des liaisons physiques particulières elle peut appeler XGetModifierMapping ou XGetPointerMapping pour connaître les nouvelles associations.)

MapRequest SubstructureRedirectMask
type : XMapRequestEvent *event.xmaprequest*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window parent;                  /* parent de la fenêtre à afficher */
        Window window;          /* fenêtre à afficher */
} XMapRequestEvent;
```

Cet événement est envoyé pour informer qu'un appel à XMapRaised ou à XMapWindow a été effectué par un client, mais la fenêtre qui était l'objet de la requête n'a cependant pas été affichée quand cet événement est reçu. Cet événement est sélectionné par le *window manager* avec le masque SubstructureRedirectMask sur la fenêtre parent. L'attribut override_redirect de la fenêtre considérée doit normalement être à False pour que l'événement soit reçu. Au lieu d'afficher la fenêtre sur la requête, le serveur envoie cet événement à l'unique client (le premier) qui l'aura sélectionné. Ce mécanisme porte le nom de *redirection* des requêtes. Il permet au *window manager* de réviser la taille ou la position de la fenêtre avant de l'afficher lui-même en y rajoutant des décorations.

MotionNotify PointerMotionMask
ButtonMotionMask
Button<N>MotionMask
type : XMotionEvent *event.xmotion*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* la fenêtre à laquelle est rapportée l'ev. */
        Window root;            /* racine sur laquelle s'est produit l'ev. */
        Window subwindow; /* fenêtre fille */
        Time time;              /* millisecondes */
```

```
    int x, y;           /* coordonnées du pointeur relatives à window */
    int x_root, y_root; /* coordonnées relatives à root */
    unsigned int state; /* masque d'état des boutons + modifieurs */
    char is_hint;       /* si c'est une trace */
    Bool same_screen;   /* si le pointeur est resté sur le même écran */
} XMotionEvent;
```

La souris a bougé. Cet événement peut être envoyé pour un mouvement quelconque de souris s'il a été sélectionné avec PointerMotionMask. On peut cependant restreindre l'envoi des événements aux mouvements se produisant avec un bouton enfoncé en utilisant ButtonMotionMask, ou Button<n>MotionMask (n prévu de 1 à 5) au lieu de PointerMotionMask. Les cinq masques Button<n>MotionMask permettent de spécifier précisément quel bouton est enfoncé et on peut les combiner entre eux.

En outre, on peut combiner les masques de sélection avec le masque PointerMotionHintMask qui précise qu'un nouveau mouvement de souris doit être envoyé quand on sera dans la situation suivante :

- un changement d'état (enfoncé/relâché) d'une touche ou d'un bouton se produit
- la souris sort de la fenêtre
- le client fait un appel à XQueryPointer ou à XGetMotionEvents.


Le masque PointerMotionHintMask ne sélectionne pas en lui-même d'événement mais il instaure un mécanisme d'envoi d'événements qui permet de mieux suivre les mouvements de la souris (cf. sections 5.2. et 5.3.).

NoExpose
type : XNoExposeEvent *event.xnoexpose*

cf. la description de GraphicsExpose.

PropertyNotify PropertyChangeMask
type : XPropertyEvent *event.xproperty*

```
typedef struct {
    int type;               /* type de l'événement */
    unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
    Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
    Display *display;       /* terminal sur lequel l'ev. s'est produit */
    Window window;          /* la fenêtre dont la propriété à été modifiée */
    Atom atom;              /* la propriété modifiée */
    Time time;              /* heure de la modification */
```

```
        int state;              /* NewValue, Deleted */
} XPropertyEvent;
```

L'événement PropertyNotify indique qu'une propriété de la fenêtre a été modifiée - à tout le moins qu'il y a été rajouté une chaîne de longueur zéro. Cet événement rapporte l'heure de la modification et le nom (Atom) de la propriété modifiée.

ReparentNotify StructureNotifyMask
SubstructureNotifyMask
type : XReparentEvent *event.xreparent*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window event;                   /* celle qui reçoit l'ev. */
        Window window;          /* celle dont le parent a changé */
        Window parent;                  /* le nouveau parent */
        int x, y;               /* coordonnées relatives au nouveau parent */
        Bool override_redirect;         /* celui de la fenêtre "reparentée" (window) */
} XReparentEvent;
```

L'événement ReparentNotify indique que le parent de la fenêtre window a changé. Il est important de sélectionner cet événement si l'on s'intéresse à la position d'une fenêtre fille de la racine, car celle-ci est susceptible d'être "reparentée" par un *window manager*. L'événement contient l'identificateur du nouveau parent et les coordonnées de la fenêtre relativement à ce nouveau parent.

ResizeRequest ResizeRedirectMask
type : XResizeRequestEvent *event.xresizerequest*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* la fenêtre devant être retaillée */
        int width, height;      /* les dimensions souhaitées, sans les bords */
} XResizeRequestEvent;
```

L'événement ResizeRequest indique qu'un autre client souhaite changer la taille de la fenêtre (alors que celle-ci n'a pas changée). Un seul client (habituellement le *window manager*) peut sélectionner cet événement car il entraîne un mécanisme de redirection. Quand il a été sélectionné, le serveur émet cet événement au lieu de valider la requête de changement de taille.

SelectionClear
type : XSelectionClearEvent *event.xselectionclear*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;          /* la fenêtre qui reçoit l'ev. et perd la sélection */
        Atom selection;                 /* la sélection dont on perd la propriété */
        Time time;              /* l'instant du changement enregistré */
} XSelectionClearEvent;
```

L'événement SelectionClear rapporte au propriétaire courant de la sélection qu'un nouveau propriétaire a été défini. Cet événement n'est pas sélectionné par masque, mais toujours envoyé à l'ancien propriétaire d'une sélection lorsqu'un autre client appelle XSetSelectionOwner pour la même sélection. La fonction XSetSelectionOwner admet les arguments suivants :

```
XSetSelectionOwner (dpy, selection, owner, time)
Atom        selection ;
Window      owner ;
Time        time ;
```

Si l'application elle-même fait appel à XSetSelectionOwner elle recevra cet événement.

SelectionNotify
type : XSelectionEvent *event.xselection*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
```

```
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window requestor;       /* la fenêtre de l'appel à XConvertSelection */
        Atom selection;
        Atom target;            /* type des données */
        Atom property;                  /* un atome ou None */
        Time time;
} XSelectionEvent;
```

Cet événement est envoyé directement par un client (avec XSendEvent) à la fenêtre de l'application qui a fait un appel à XConvertSelection (cf. la description de l'événement SelectionRequest). L'envoi aura été effectué après que la sélection ait été convertie et stockée dans une propriété, ou bien même lorsque la conversion demandée n'aura pu être effectuée (ce qui sera alors indiqué par un champ property à None). Si l'application elle-même appelle XConvertSelection, elle doit recevoir cet événement. Rappelons que la fonction XConvertSelection admet les arguments suivants :

```
XConvertSelection (dpy, selection, target, property, requestor, time)
Atom        selection, target, property ;
Window      requestor ;
Time        time ;
```

Les champs de l'événement ont les mêmes valeurs que celles spécifiées dans la requête à XConvertSelection qui a provoqué l'envoi de cet événement par le propriétaire — à ceci près que le champ property peut, soit spécifier l'atome de la propriété indiquée par le demandeur (le type cible étant alors indiqué dans target), soit être à None, ce qui indique que les données n'ont pu être converties dans le type cible (target).

SelectionRequest
type : XSelectionRequestEvent *event.xselectionrequest*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window owner;                   /* le propriétaire */
        Window requestor;       /* le demandeur */
        Atom selection;                 /* la sélection demandée */
        Atom target;            /* le type cible de conversion */
```

```
        Atom property;
        Time time;
} XSelectionRequestEvent;
```

SelectionRequest est envoyé au propriétaire de la sélection quand un autre client en demande le contenu par XConvertSelection. Le propriétaire est alors tenu d'envoyer en réponse un événement SelectionNotify à la fenêtre intéressée. Les champs de cet événement ont les valeurs spécifiées dans la requête à XConvertSelection qui a déclenché l'envoi de cet événement. Le propriétaire est tenu de convertir (si possible) la sélection en se basant sur le type spécifié dans target. Si une propriété est spécifiée, le propriétaire doit stocker les résultats sur cette propriété de la fenêtre requestor et envoyer un événement de type SelectionNotify à cette fenêtre par XSendEvent. S'il ne peut satisfaire la demande spécifiée, le propriétaire doit envoyer un événement SelectionNotify de champ property à None.

UnmapNotify SubstructureNotifyMask
StructureNotifyMask
types : XUnmapEvent *event.xunmap*

cf. description de MapNotify.

VisibilityNotify VisibilityChangeMask
type : XVisibilityEvent *event.xvisibility*

```
typedef struct {
        int type;               /* type de l'événement */
        unsigned long serial;   /* n° de la dernière requête traitée par le serveur */
        Bool send_event;        /* vrai si l'ev. provient d'un appel à XSendEvent */
        Display *display;       /* terminal sur lequel l'ev. s'est produit */
        Window window;
        int state;              /* constante d'état de visibilité */
} XVisibilityEvent;
```

L'événement VisibilityNotify rapporte tout changement de visibilité dans une fenêtre de type InputOutput. Les fenêtres de classe InputOnly ne peuvent recevoir ce type d'événement et les sous-fenêtres d'une fenêtre donnée sont ignorées dans le calcul de la visibilité. Le champ state peut valoir VisibilityUnobscured, VisibilityPartiallyObscured ou VisibilityFullyObscured.

Table des Matières