Institut d'Informatique
de la Faculté des Sciences

Lausanne, février 1999

Un RasterViewer pour SPANAKE

– Les outils de l'Informatique moderne appliqués à un problème de Biologie –

Javier Iglesias

Mémoire de Module de Spécialisation
dirigé par le Professeur Marco Tomassini

Le présent mémoire découle du travail de reconception complète de la partie de SPANAKE permettant d'afficher les séries de trains de potentiel sous la forme de rasters de points. Il a été chapeauté par l'Institut d'Informatique de la Faculté de Sciences en collaboration avec le Dr Villa du Laboratoire de Neuro-heuristique, et a été effectué au cours du semestre d'hiver 1998-1999, sous la forme d'un module de spécialisation
en vue de l'obtention du grade de Diplômé en Biologie.

I. INTRODUCTION

Parmi les approches expérimentales les mieux adaptées à l'investigation fonctionnelle du système nerveux, les techniques d'enregistrement de l'activité électrique démontrent quotidiennement leurs avantages. La diversité des échelles de temps et d'espace auxquelles les mesures peuvent être faites (mesures unitaires, champs de potentiels locaux, régions cérébrales) et leurs précisions sont des atouts majeurs.

Les études qui vont nous intéresser dans les pages qui suivent sont de type unitaire. Elles consistent à enregistrer simultanément des potentiels d'action produits par les neurones situés à proximité d'électrodes. La séquence de ces événements est plus précisément dénommée *train de potentiel d'action*. La fréquence moyenne aussi bien que la signature temporelle de ces trains de potentiels sont déterminantes pour le codage de l'information qu'ils transportent.

Un potentiel d'action est un phénomène de dépolarisation du potentiel électrique transmembranaire. Le mécanisme moléculaire qui sous-tend cette activité électrique est le passage, suite à la perméabilisation de la membrane cellulaire, d'ions divers au travers des canaux formés par des protéines spécialisées. Par le jeu des protéines membranaires s'ouvrant et se fermant tour à tour, la dépolarisation peut se déplacer le long des membranes des neurones et colporter une information.

Ces événements sont binaires, tout ou rien. Suite à leur enregistrement au moyen d'électrodes, des traitements statistiques peuvent leur être appliqués, comme s'il s'agissait de processus discrets. La recherche de motifs dans les trains de potentiels est capitale pour le processus de reconstitution du flux de l'information dans le cortex cérébral. Il s'agit d'une étape préliminaire dans la compréhension de la coordination sensori-motrice à la base de la transmission et de l'intégration de l'information.

Dans le cadre de cette problématique, le programme SPANAKE – *spike analysis* – est actuellement développé par l'institut d'Informatique de la faculté des Sciences en collaboration avec le laboratoire de Neuroheuristique. Il est bâti autour de l'idée de laboratoire virtuel appliqué au traitement et à l'analyse des trains de potentiels d'action issus des expériences neurophysiologiques.

Ses concepteurs souhaitent fournir, sous une seule interface, un ensemble d'outils permettant aux neurophysiologistes d'accomplir les fonctions suivantes :

- ➢ la récupération de données de trains de potentiels sur un réseau informatique,
- ➢ la prévisualisation des données brutes à des fins de prétraitement et de tri,
- ➢ le lancement d'analyses statistiques qui peuvent nécessité l'utilisation de machines puissantes telles que des serveurs,
- ➢ la visualisation des résultats issus de ces analyses.

A. Les données expérimentales.

Les données expérimentales sont contenues dans des fichiers dont la structure a été définie par M. Abeles (Dept. Of Physiology, Hadassah Medical School, The Hebrew University, Jerusalem). Les événements y sont définis par une clef formée d'un *type* et d'un *qualifier*, à laquelle est rattachée une valeur correspondant au *temps* écoulé depuis l'événement précédent du fichier. Le format de cette triplette dans un fichier ASCII est :

```
<type>, <qualifier>, <time>
```

où le type et le *qualifier* sont des valeurs hexadécimales, alors que le temps est une valeur décimale entière. Les virgules, les espaces, les tabulations et les retours à la ligne sont des séparateurs. On pourra ainsi trouver dans un fichier l'information suivante :

```
[...]   1,F,21  1,3,34  51,A06,27   [...]
```

où l'événement `1`, `3` c'est produit `34` unités de temps – souvent la milliseconde – après le `1`, `F` et `27` unités de temps avant le `51`, `A06`.

L'Annexe A contient la définition complète de la structure des fichiers de donnée Abeles.

Un moyen simple et élégant de regrouper en famille les événements du même type est d'appliquer sur les qualifier des *masques bit-à-bit* (`bitwise AND`). Il s'agit de comparer bit après bit les nombres qui composent le qualificatif avec ceux du masque en appliquant l'opérateur booléen AND. Si le nombre obtenu par cette opération est le même que le qualifier testé, on accepte l'événement dans la "famille". Voici un exemple concret où l'on applique le masque `51 <FA06>` aux codes `51`, `4A06` ; `51`, `03E2` et `52`, `4A06` :

Table de vérité de `AND` :

p :	0	1	0	1	**AND**
q :	0	0	1	1	
p AND q :	**0**	**0**	**0**	**1**	

Qualifier :	4	A	0	6
	0 1 0 0	1 0 1 0	0 0 0 0	0 1 1 0
Masque :	F	A	0	6
	1 1 1 1	1 0 1 0	0 0 0 0	0 1 1 0
Résultat :	**4**	**A**	**0**	**6**
	0 1 0 0	1 0 1 0	0 0 0 0	0 1 1 0

De la même manière, on trouvera que si le masque `51 <FA06>` s'applique au code `51`, `4A06`, il ne s'applique à aucun des deux autres puisque le résultat du masque sur `51`, `03E2` donne `0202` et que le `52`, `4A06` n'est pas du bon type. On remarquera que le masque `<FFFF>` couvre tous les qualifier que l'on pourrait imaginer, alors que le `<0000>` n'en couvre aucun.

Cette technique de masquage est assez courante en électronique. Son inconvénient est qu'il faut définir astucieusement les codes, de manière à pouvoir appliquer les masques selon des critères utiles. Pour un ensemble de formes de couleur par exemple, on peut chercher à trier les ronds – définis comme tels par leur code – grâce à un certain masque ou encore toutes les formes jaunes au moyen d'un autre masque. Un troisième masque peut servir à trier les ronds jaunes des autres formes et des autres couleurs.

Une autre source de données est contenue dans les *fichiers de parsing*. Ils sont obtenus suite à une analyse statistique de corrélation sur les fichiers de données expérimentales. Cette analyse permet de découvrir des synchronisations entre un neurone et lui-même ou d'autres neurones. Des séquences de potentiels d'action – *patterns* – peuvent apparaître. Par exemple, lorsque le neurone A déclenche un potentiel d'action, B fait de même 4 ms plus tard, puis à nouveau 6 ms plus tard.

Ce type de patterns est très intéressant puisqu'il permet en théorie de reconstituer le passage de l'information au travers d'un réseau de neurones et, par la suite, la structure de celui-ci.

De telles synchronisations seront enregistrées dans les fichiers de parsing de la manière suivante :

```
<fileName>  <cellNbr>  <timeInFile>
```

On trouvera par exemple dans le fichier de parsing `7bb.tim` :

```
[...]  c1lc04.002  7  29690  [...]
[...]  c1lc04.002  7  37032  [...]
[...]  c1lc04.031  7  15495  [...]
```

Il faut lire l'exemple ci-dessus comme suit : dans le fichier `c1lc04.002`, situé dans le même répertoire que le fichier de parsing, les temps `29690` et `37032` sont intéressants pour la cellule numéro `7`, puis le temps `15495` dans le fichier `c1lc04.031`. Il est à noter que les temps référencés recommencent à zéro à chaque fois qu'on se réfère à un nouveau fichier.

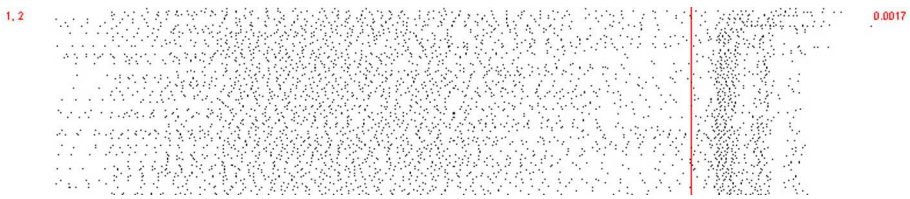

Figure 1 : exemple de dot raster

B. La visualisation des données.

La prévisualisation des données brutes s'effectue généralement sous la forme d'un graphique – *raster de points* ou *dot raster* – où chaque point marque le temps de déclenchement d'un potentiel d'action. Ainsi, on trouve le temps en abscisse et en ordonnée, les occurrences d'un type d'événement particulier appelé *trigger* (la barre rouge de la Figure 1). Le trigger est un événement le plus souvent d'origine extérieure à l'animal dont on cherche à montrer l'effet de

synchronisation sur les trains de potentiels. Sur chaque ligne du graphique, les potentiels d'action d'un ou de plusieurs neurones, symbolisés par un point ou un trait vertical, sont répartis de chaque côté du trigger, en fonction de leur temps d'apparition relatif.

Au laboratoire de Neuro-heuristique, l'expérience type consiste à placer un rat éveillé dans une caisse en plastique spécialement équipée de deux haut-parleurs, une mangeoire, un stroboscope et de cellules infrarouges. Le rat apprend qu'il doit se tenir à l'extrémité opposée à la mangeoire de la caisse, attendre qu'un son soit produit par les haut-parleurs pour se précipiter vers la mangeoire où il y reçoit une graine de tournesol. Les cellules infrarouges servent à mesurer ce déplacement, alors que le stroboscope signale à l'animal qu'il doit reprendre la position de départ. Les sons émis peuvent être graves ou aigus, à droite ou à gauche, si bien que l'animal doit reconnaître la combinaison particulière qui s'accompagne d'une graine. Ce type d'expérience est dit GO/NOGO, où l'animal est soumis à des informations contradictoires qu'il doit discriminer.

Les rats sont alors opérés afin de leur implanter des électrodes dans le cerveau, puis les expériences reprennent. C'est à ce moment que l'on raccorde les électrodes à un ordinateur et que les trains de potentiels d'action sont enregistrés, puis représentés, comme dans la Figure 1, autour d'un trigger qui peut être l'instant de production du son ou la mise en marche du stroboscope par exemple.

En empilant les occurrences d'un trigger, on pourra mettre en évidence des relations directe entre celles-ci et l'apparition d'un potentiel d'action. La Figure 1 nous montre que le neurone en question présente une synchronisation très nette autour de ce trigger : une diminution de son activité juste avant et une reprise groupée juste après.

Ce type de synchronisations ne peut réellement être établi que par des outils statistiques (auto-corrélation, crosscorrélation, …) qui sont toutefois très lourds en terme de calculs. Aussi les données sont-elles prévisualisées afin de ne lancer ces analyses statistiques que sur les seules données qui en méritent "visiblement" l'investissement. L'œil expert du scientifique peut lui permettre de gagner un temps considérable sur les analyses.

Le travail auquel ce mémoire se rapporte a consisté à reconcevoir la partie de SPANAKE affectée à la traduction de données sous la forme graphique de rasters de points. Cette partie avait été abordée dans le projet SPANAKE. Cependant, elle souffrait d'imperfections et nous avons dû la repenser dans sa totalité. De nouvelles solutions ont été proposées pour la gestion de la mémoire et de l'affichage. Le langage de programmation Java a été employé pour la mise en œuvre de ce travail, comme pour le développement de SPANAKE.

II. ETAT DES LIEUX

Les logiciels permettant la visualisation et le traitement des données neurophysiologiques ne sont pas légion. Depuis son apparition, SPANAKE fait figure de précurseur dans le domaine.

En ce qui concerne l'affichage des rasters, l'interface graphique propose déjà un certain nombre d'outils, mais ils ne semblent jamais être à leur place. Les onglets '`Settings`' – Figure 3 – et '`Output Layout`' – Figure 4 – ont des noms qui ne définissent pas clairement leurs contenus, si bien que l'on passe continuellement de l'un à l'autre à la recherche du paramètre à modifier.

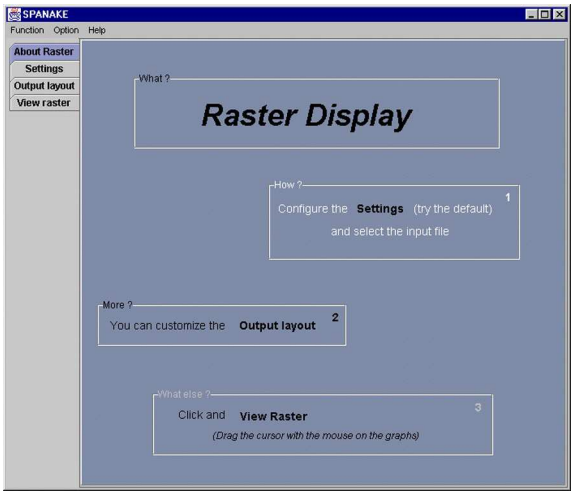

Figure 2 : l'onglet '`About Raster`'

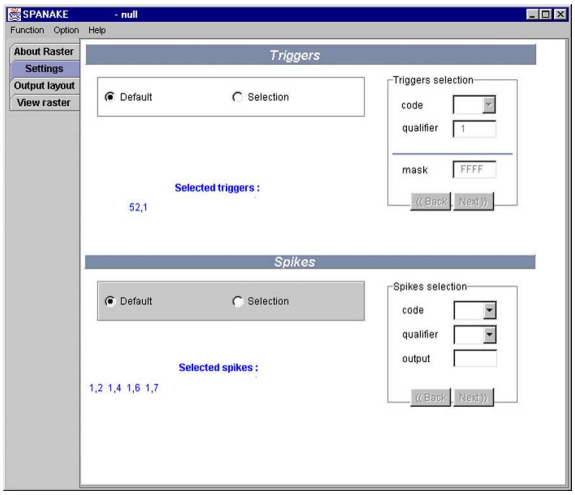

Figure 3 : l'onglet '`Settings`'

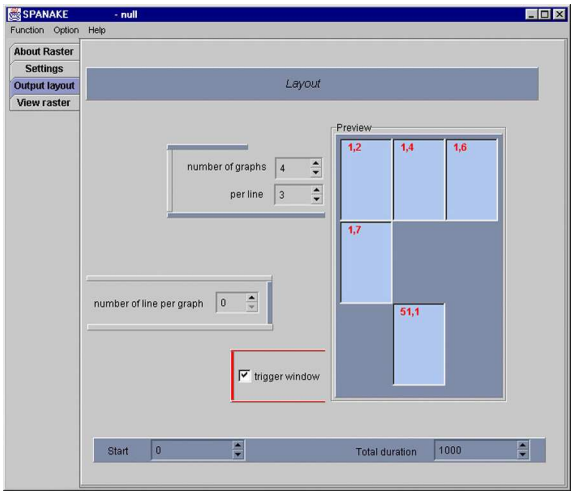

Figure 4 : l'onglet '`Output Layout`'

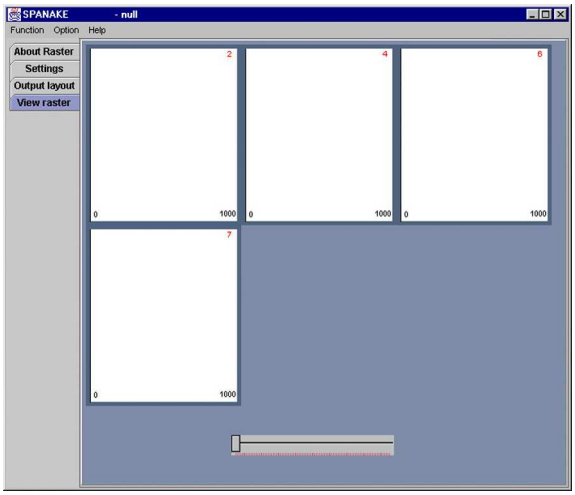

Figure 5 : l'onglet '`View Raster`'

De manière générale, on peut dire que la solution proposée ici, basée sur une fenêtre unique subdivisée en onglets, n'est pas satisfaisante – de Figure 2 à Figure 5.

Plus concrètement, il subsiste un problème de gestion de la mémoire qui rend la visualisation consécutive de deux fichiers peu envisageable – Figure 6. De plus, le travail sur les fichiers de parsing est assez laborieux, dû à l'intégration tardive de cette notion dans le projet.

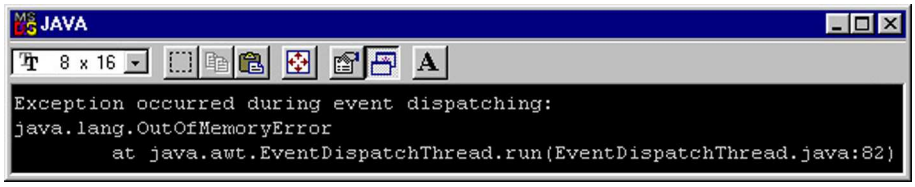

Figure 6 : la grande plaie de SPANAKE

On notera encore l'impossibilité d'utiliser les masques pour regrouper les triggers, et ce malgré deux champs de texte, non fonctionnels, qui laissent croire le contraire – Figure 3, en haut à droite. Un masque <FFFF> est en fait appliqué à la lecture du fichier. Cela découle d'une mise en œuvre posant clairement les différences entre les notions de trigger et d'événement. Ces deux aspects, pourtant intimement liés, ne sont pas interchangeables, rendant impossible certains affichages qui pourraient pourtant s'avérer instructifs.

Cependant, les paramètres importants sont clairement définis, et on dénote bon nombre de bonnes idées, comme par exemple le dynamisme des rasters que l'on peut déplacer autour du trigger à l'aide d'un curseur, permettant d'afficher plutôt ce qui c'est passé avant ou après son apparition, sans avoir besoin de paramétrer à nouveau l'affichage. Pour des questions principalement esthétiques, la solution mise en œuvre n'est pas complètement satisfaisante. On appréciera l'affichage par défaut qui permet à l'utilisateur d'avoir immédiatement quelque chose à regarder.

Il paraissait clair que la partie de SPANAKE affectée à la visualisation des rasters devait être repensée pour offrir une interface plus confortable et des fonctionnalités supplémentaires à l'utilisateur, tout en reprenant et en améliorant celles déjà disponibles.

III. SOLUTIONS PROPOSÉES

Comptes tenus des problèmes notés dans la version précédente, trois axes prioritaires ont été dégagés pour ce projet :

- ➢ trouver un modèle d'interface graphique convivial et performant,
- ➢ contrôler étroitement la gestion de la mémoire,
- ➢ intégrer à part entière la lecture des fichiers de parsing.

D'autres aspects généraux de la programmation ont dû être pris en compte, comme la modularité des objets informatiques et la réutilisation du code, pour simplifier et anticiper les modifications futures.

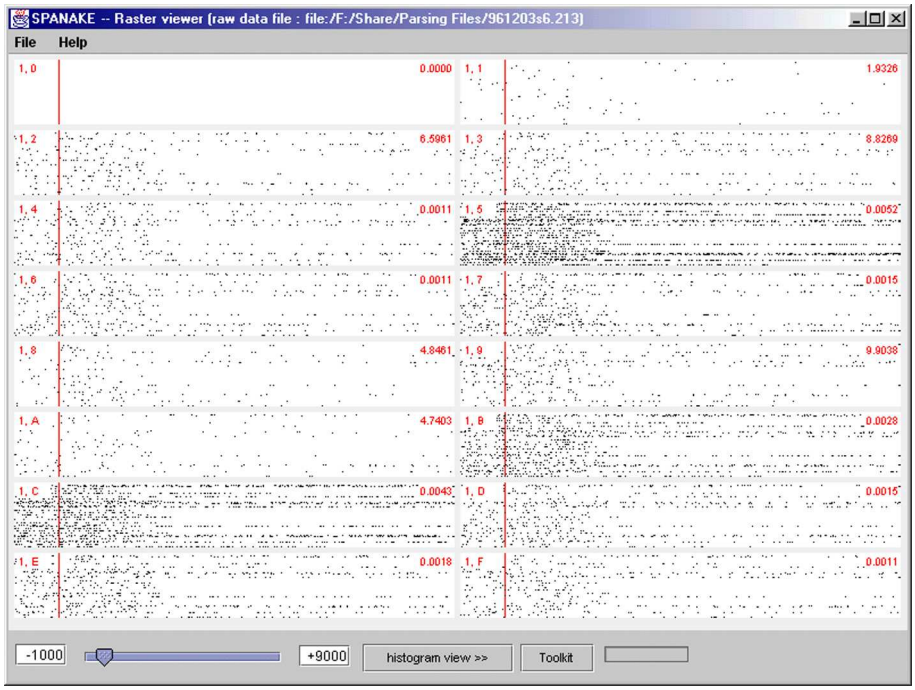

Figure 7: la fenêtre principale du Raster Viewer.

A. L'interface graphique.

L'interface graphique se décompose en deux fenêtres. La première – `RasterViewer` (Figure 7) – est exclusivement affectée à la visualisation des rasters de points, alors que la seconde – `RasterViewerToolkit` – constitue une interface semi-directe où l'on peut modifier les paramètres d'affichage de la fenêtre principale en interagissant directement avec une représentation symbolique de son contenu.

La fenêtre principale se compose d'un menu, d'une glissière, de deux boutons et d'une barre de progression.

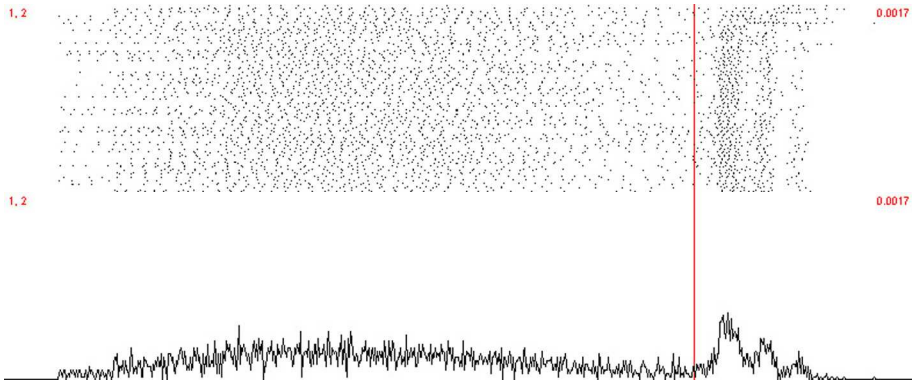

Figure 8 : un dot raster et la vue en histogramme associée

Le slider permet de faire bouger les images affichées le long de leur axe de temps afin de pouvoir observer ce qui s'est passé avant ou après l'occurrence du trigger. Deux champs de texte situés à gauche et à droite de ce slider permettent à la fois de visualiser la quantité de temps représentée avant et après le trigger, et à entrer ces paramètres à l'aide du clavier. Il faut alors presser <`enter`> pour valider ces valeurs. Si l'utilisateur entre une valeur non acceptable (un temps négatif ou plus grand que ce qu'il a été demandé d'afficher) la valeur correcte la plus proche est prise en compte. Si cela n'est pas possible, la valeur précédente est récupérée.

L'utilisation du bouton `Toolkit` rappelle la fenêtre `RasterViewerToolkit`, dans le cas où l'utilisateur l'aurait fermée par erreur ou qu'elle serait passée sous la fenêtre `RasterViewer`.

Le second bouton permet de passer d'une vue en raster à une vue en histogramme – Figure 8. Cette nouvelle option permet d'avoir une vue plus claire des densités de points à l'écran, bien qu'elle ne résulte pas d'une analyse statistique, mais du comptage des points sur une colonne de pixels de l'image. Il s'agit donc d'une représentation brute et peu précise, qui offre tout de même un confort de lecture et d'interprétation appréciables.

Finalement, la barre de progression montre l'état d'avancement de la génération des images lors de l'ouverture du fichier, afin que l'utilisateur puisse se convaincre qu'il se passe bien quelque chose pendant qu'il attend !

A l'heure actuelle, le menu n'est pas utilisable car il regroupe les fonctions qui restent encore à mettre en œuvre, telles que l'impression ou l'ouverture de nouveaux fichiers dans la même fenêtre ou dans une fenêtre différente.

La seconde fenêtre, de petite taille, est fractionnée en deux onglets : l'un servant à gérer les paramètres propres à l'affichage des données – *Layout*, Figure 9 – alors que l'autre est affecté à la sélection du ou des triggers que l'on souhaite afficher – *Triggers*,
Figure 10.

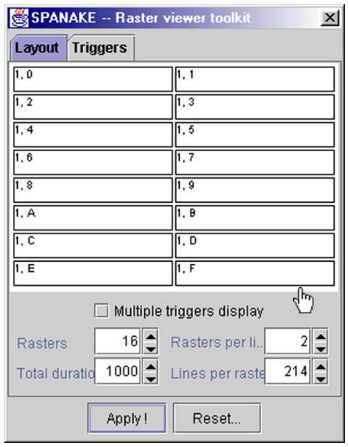

Figure 9 : l'onglet '`Layout`' de la fenêtre `RasterViewerToolkit`

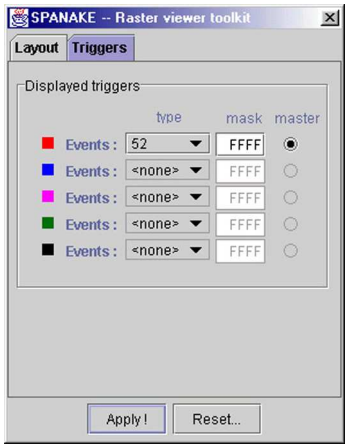

Figure 10 : l'onglet '`Triggers`' de la fenêtre `RasterViewerToolkit`

La solution consistant à faire deux fenêtres nous a semblé préférable à une solution où l'utilisateur modifie directement le contenu de l'affichage, car le processus de calcul des images peut faire intervenir des lectures de fichiers et le tri dans des tableaux de grande taille. On attend donc de l'utilisateur qu'il modifie plusieurs paramètres à la fois – le nombre de rasters à l'écran, leurs contenus, leurs dispositions et la quantité de temps à afficher, par exemple – avant de lancer la génération des images en appuyant sur le bouton '`Apply !`'.

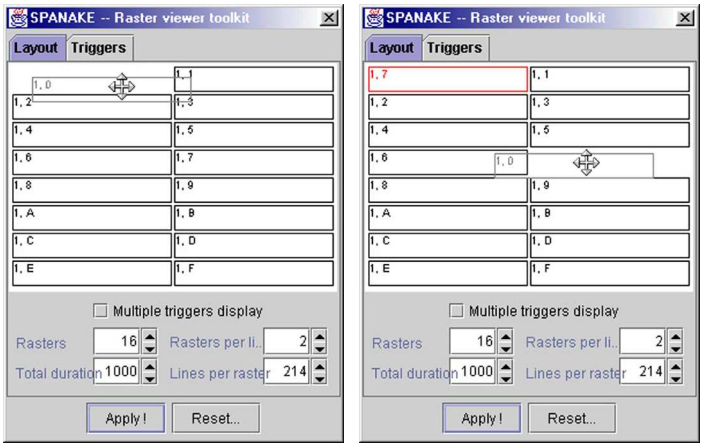

Figure 11 : *drag and drop* sur l'écran symbolique de la fenêtre `RasterViewerToolkit`

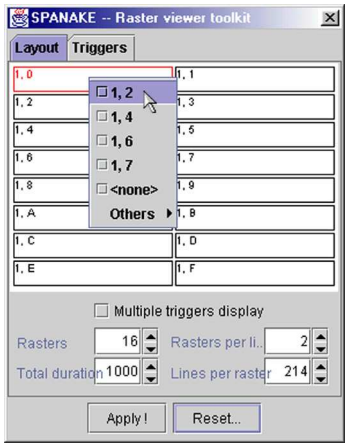

Figure 12 : sélection du contenu des rasters via un menu contextuel

L'utilisation de deux fenêtres, contrairement à la version précédente qui utilisait des onglets sur une seule fenêtre pour l'affichage et le paramétrage, permet à l'utilisateur d'avoir encore sous les yeux l'affichage précédent au moment de décider des paramètres à modifier.

Parallèlement, l'utilisateur se voit indiquer les zones avec lesquelles il peut interagir par le curseur de la souris : celui-ci prend la forme d'une main. Ce mode simple de communication permet de guider efficacement un utilisateur novice.

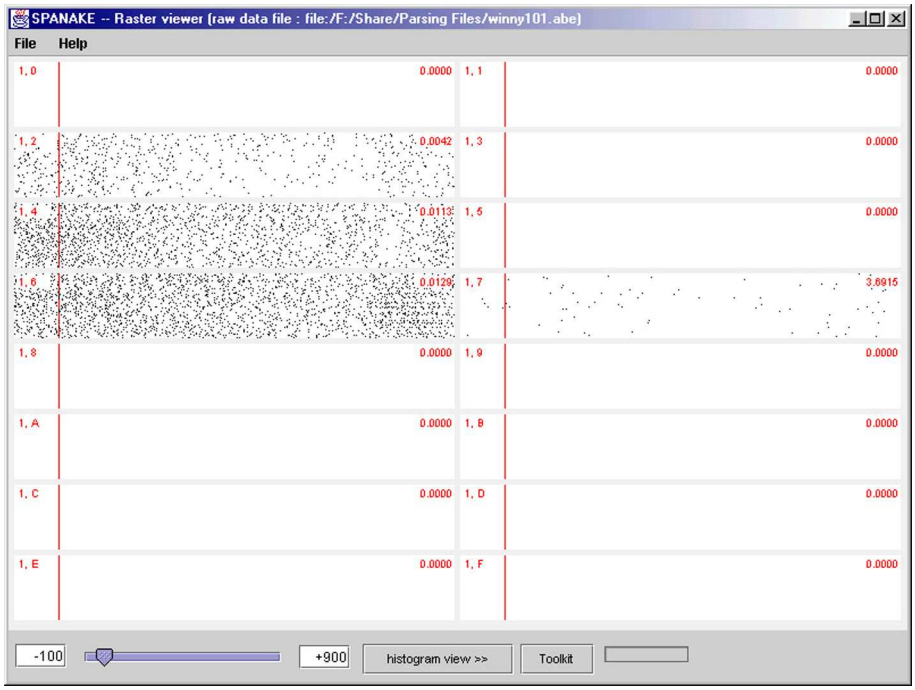

Figure 13 : la vue par défaut à l'ouverture du fichier `Winny101.abe`

Parmi les actions possibles pour l'utilisateur, on trouvera deux moyens simples de changer la disposition des graphiques dans la fenêtre. Le premier fait appel au déplacement des rasters déjà affichés, au moyen d'un drag and drop intuitif implémenté sur l'écran virtuel de la fenêtre `RasterViewerToolkit` – Figure 11. Le second consiste encore plus simplement à sélectionner parmi la liste des événements présents dans le fichier, lequel ou lesquels on souhaite voir apparaître dans un raster donné. Ce choix s'opère au moyen d'un menu contextuel qui s'ouvre lorsque le bouton droit de la souris – `opt + click gauche` sur Macintosh – est effectué au-dessus du rectangle représentant un raster – Figure 12.

i) La consultation d'un fichier de données

Une fois le fichier à visualiser, par exemple '`Winny101.abe`', sélectionné dans la fenêtre ad hoc de SPANAKE – hors du scope de ce travail –, il est entièrement lu et un rapport mentionnant son contenu est redirigé vers la sortie d'erreurs standard du système – `System.err`. C'est alors que la fenêtre `RasterViewer` apparaît et que sa barre de progression indique qu'elle a commencé à générer des images.

L'utilisation de paramètres d'affichage par défaut nous a semblé nécessaire afin que l'utilisateur ait dès l'ouverture quelque chose à l'écran, lui évitant ainsi l'angoisse de la page blanche – Figure 13.

Les calculs terminés, la fenêtre `RasterViewerToolkit` – Figure 14 – fait son apparition et invite l'utilisateur à modifier l'affichage par défaut. Ce dernier se compose de seize rasters disposés par deux, présentant les éventuelles occurrences des événements `1`, `0` à `1`, `F`, `1000` unités de temps autour du premier événement dont le `type` soit plus grand ou égal à `50`, avec un masque `<FFFF>`. 10% du temps total est affiché avant le trigger et le reste après.

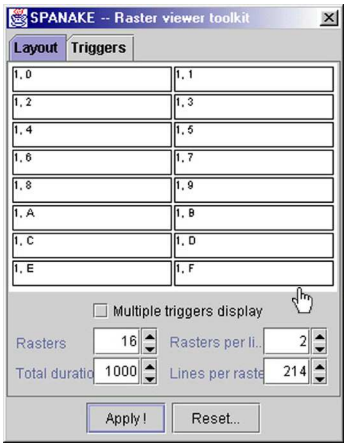

Figure 14 : la vue par défaut de la fenêtre `toolkit`

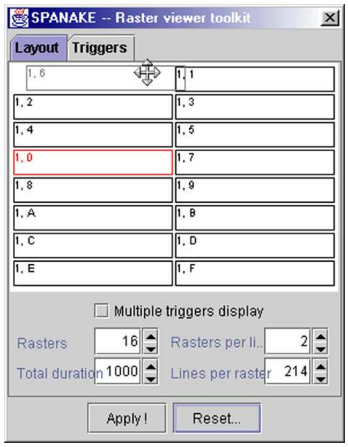

Figure 15 : l'échange des positions des rasters 1 et 8 par 'drag and drop'

On va chercher à modifier l'affichage de la Figure 13 afin d'obtenir quelque chose de plus explicite. Pour ce faire, il nous faut :

- ne laisser à l'écran que les rasters contenant des occurrences – Figure 16,
- placer les rasters un par ligne – Figure 17,
- afficher ceux-ci sur 4000 unités de temps – Figure 18.

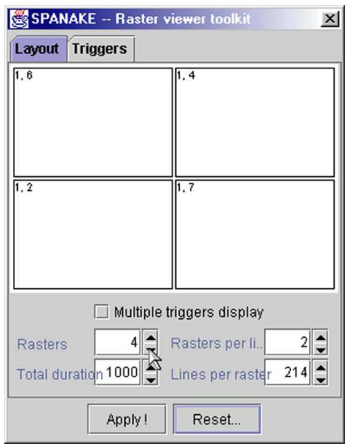

Figure 16 : seuls les rasters intéressants sont conservés

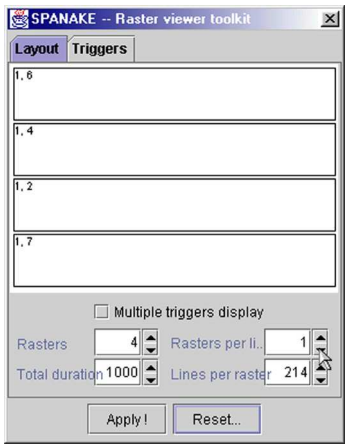

Figure 17 : on dispose les rasters un par ligne

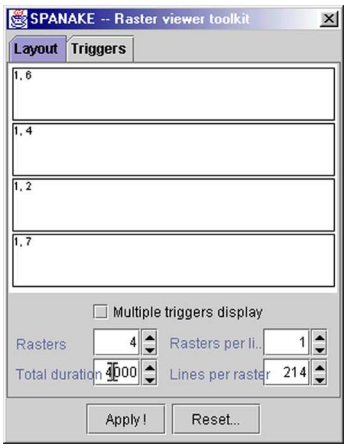

Figure 18 : on affiche 4000 unités de temps

Pour sélectionner le contenu de l'écran, nous disposons de deux méthodes : les menus contextuels pour redéfinir chaque raster l'un après l'autre ou manipuler les rectangles représentant les rasters dans la fenêtre toolkit pour les disposer correctement. C'est cette deuxième méthode que nous allons développer ici, parce qu'elle est plus rapide pour cet exemple.

Il nous faut échanger les rasters 8 et 1. La main du curseur est placée sur un des membres du couple, par exemple le 8. Celui-ci devient rouge pour indiquer qu'il est présélectionné. Le bouton de la souris est alors pressé pour le saisir et on le glisse au-dessus de celui avec lequel on souhaite l'échanger, tout en maintenant le bouton de la souris pressé – on le "drag" ! Le raster 1 a pris la place du huitième et se singularise en se colorant de rouge – Figure 15. Si on

relâche le bouton de la souris – on le "drop" ! –, les contenus des rasters 1 et 8 ont échangé leurs places. Il faut répéter l'opération pour les couples de rasters 7-2 et 5-4. A présent, les rasters qui contiennent de l'information sont placés au sommet de l'affichage.

L'étape suivante consiste à faire disparaître les rasters vides. Le paramètre '`Rasters`' est passé de 16 à 4, soit en tapant la nouvelle valeur dans le champ de texte, soit en utilisant les petites flèches placées à sa droite pour décrémenter jusqu'à la valeur désirée – Figure 16. Si on choisit d'entrer la valeur au clavier, il est nécessaire de la valider en appuyant sur la touche <`enter`>.

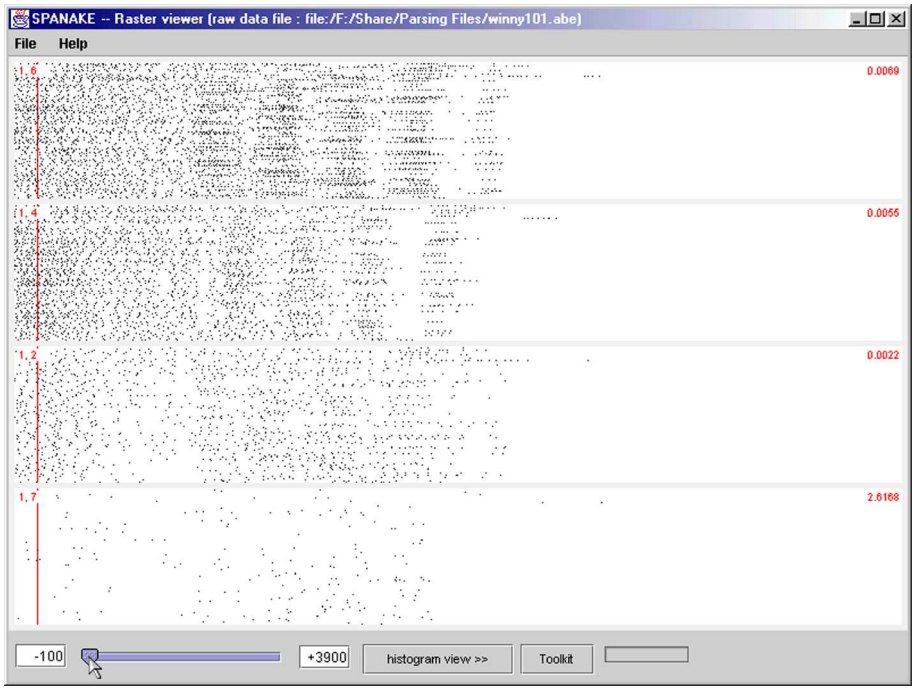

Figure 19 : `Winny101.abe` après modifications des paramètres d'affichage

Ces quatre rasters sont répartis sur quatre lignes grâce au paramètre '`Rasters per line`' – Figure 17 –, et l'affichage de 4000 unités de temps est demandé en l'introduisant dans le champ de texte de '`Total duration`', à la place de 1000 – Figure 18.

Le bouton '`Apply !`' est pressé pour que les valeurs entrées soient utilisées pour l'affichage dans la fenêtre principale qui vient se placer au sommet des fenêtres actuellement ouvertes – Figure 19. Si le bouton '`Reset...`' avait été pressé, les paramètres actuellement utilisés pour l'affichage auraient repris place dans leurs champs respectifs de la fenêtre toolkit.

La fenêtre principale nous permet de déplacer la position relative du trigger – la barre verticale de couleur – pour voir les événements précédents ou ultérieurs à l'apparition du trigger – Figure 20 –, ainsi que d'afficher une vue en histogramme des rasters – Figure 21.

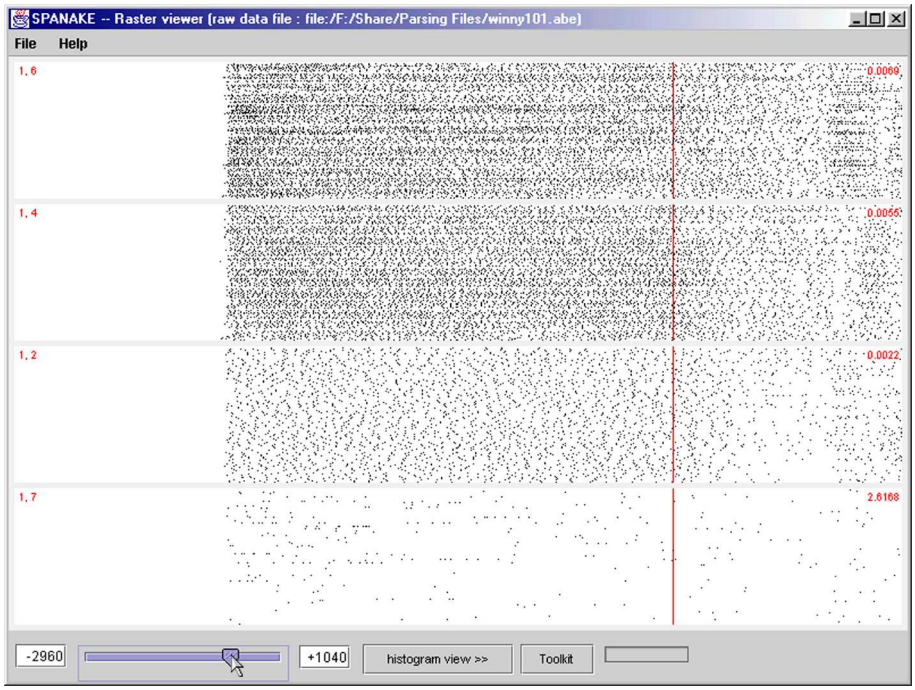

Figure 20 : la Figure 19 après modification de la position du slider

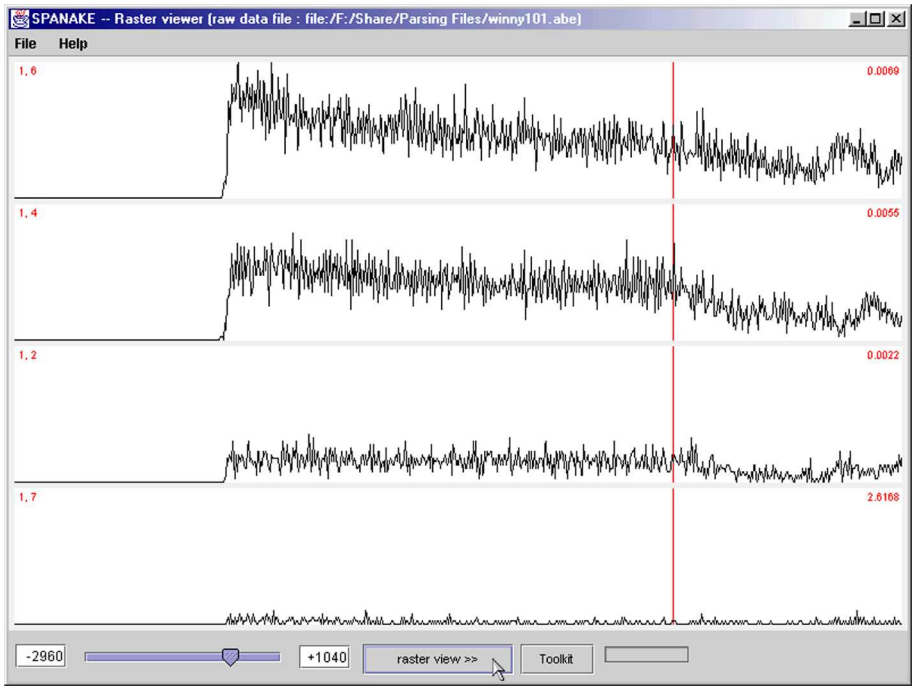

Figure 21 : l'option '`histogram view`'

Le trigger utilisé peut être rapidement modifié au deuxième onglet de la fenêtre toolkit où une seule liste déroulante est actuellement activée – Figure 22. Tous les types d'événements présents dans le fichier y sont référencés. Le trigger peut y être sélectionné. Éventuellement le masque par défaut <FFFF> peut être modifié selon les besoins dans le champ texte prévu à cet effet. La pression du bouton '`Apply !`' met fin à la manœuvre.

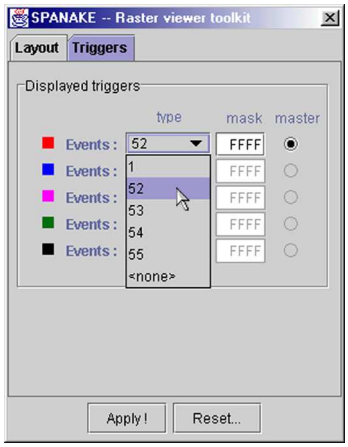

Figure 22 : sélection du '`master trigger`'

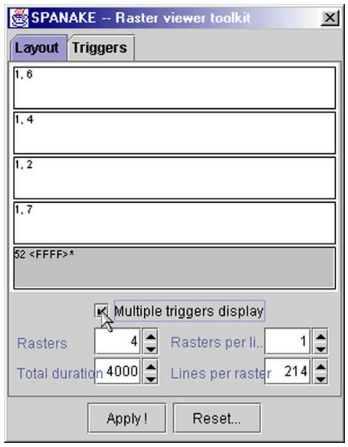

Figure 23 : la boîte de sélection '`Multiple triggers display`'

Un raster particulier autorise l'affichage de multiples triggers par rapport à un trigger principal dit '`master trigger`'. Ni son contenu, ni sa position dans la fenêtre ne peuvent être changées comme c'est le cas pour les autres. Il est centré au bas de l'écran, et les modifications se font via l'onglet '`Triggers`' de la fenêtre toolkit. Pour activer cet affichage, il faut commencer par valider la boîte de sélection '`Multiple triggers display`' – Figure 23.

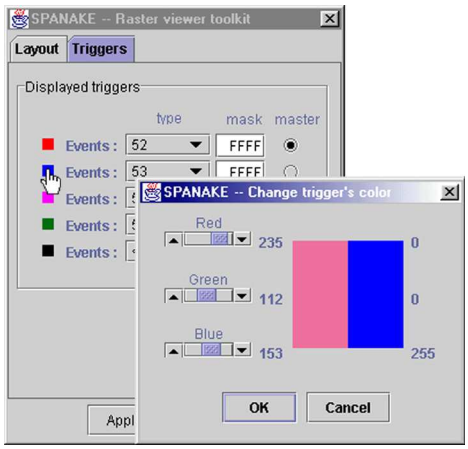

Figure 24 : pour changer la couleur d'affichage d'un trigger

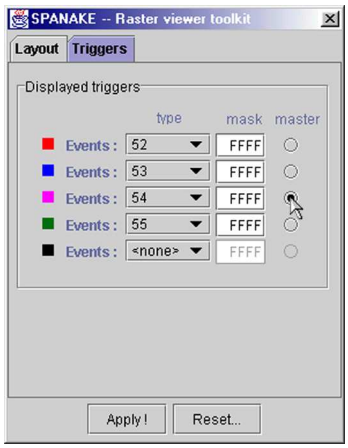

Figure 25 : le changement de '`master trigger`'

Jusqu'à cinq types d'événements et de masques peuvent être sélectionnés dans l'onglet '`Triggers`'. À chacun correspond une couleur définie par les petits carrés sur la gauche. Un click sur ces carrés ouvre une boîte de dialogue modale – Figure 24 – grâce à laquelle on peut définir une nouvelle couleur. Les boutons radio sur la droite de l'onglet servent à déterminer lequel des triggers doit servir de '`master trigger`' – Figure 25.

Ces changements peuvent être observés sur la Figure 26.

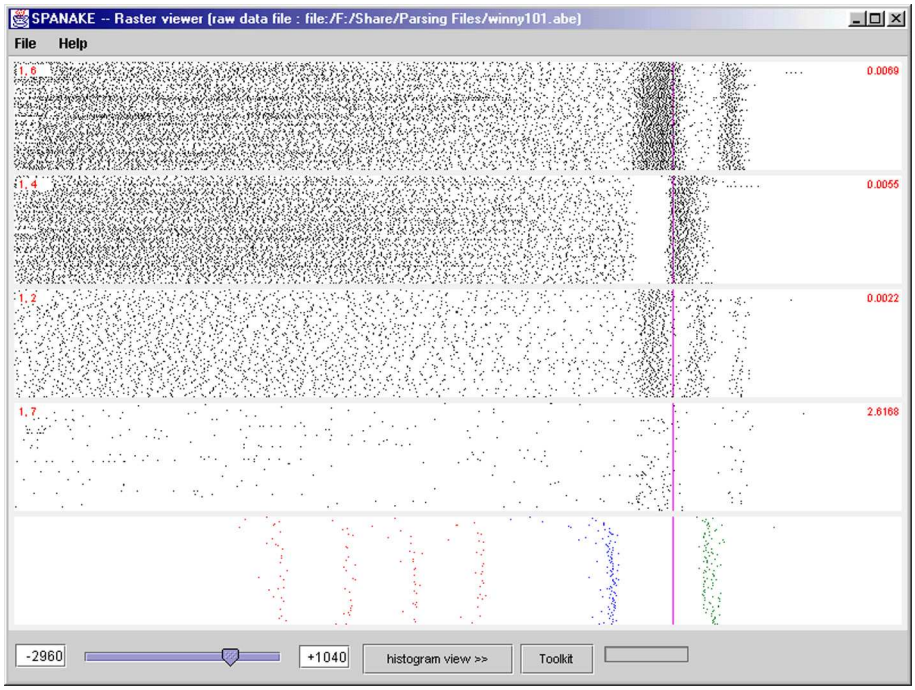

Figure 26 : l'affichage multiple de triggers

ii) La consultation d'un fichier de parsing

Une fois un fichier de parsing sélectionné et la lecture du fichier effectuée, les paramètres d'affichage par défaut sont appliqués, et tous les fichiers mentionnés dans le fichier de parsing sont également lus. Ici encore, seize rasters apparaissent à l'écran, présentant les éventuelles occurrences des événements `1`, `0` à `1`, `F` autour des temps mentionnés dans le fichier de parsing pour chaque fichier référencé.

L'utilisateur peut également changer le nombre, le contenu et la disposition des rasters pour obtenir le résultat désiré. Il peut également faire apparaître un raster avec les occurrences de certains "triggers". Ceux-ci ne sont pas réellement utilisés en tant que tels dans ce contexte, mais permette tout de même de vérifier si les synchronisations que fait apparaître le fichier de parsing le sont aussi avec un trigger ou qu'elles sont pour ainsi dire inhérent aux cellules enregistrées.

B. La gestion de la mémoire

La quantité très variable de données que l'on peut trouver dans les fichiers rend la gestion de la mémoire complexe. En effet, il faut prévoir des tableaux par défaut de grande taille afin d'être prêts à accueillir n'importe quel fichier.

On pourrait penser pallier ce problème en calculant les images parallèlement à la lecture du fichier de données, sans rien stocker en mémoire. Cette proposition s'avère inapplicable, car chaque nouvelle génération d'une image demanderait une relecture complète du fichier qui, dans le meilleur des cas, se trouve sur un disque local, et dans le pire des cas sur un autre continent !

Une solution consisterait à utiliser une structure de donnée permettant la gestion dynamique de l'allocation de la mémoire, comme la liste linéaire, pour stocker le contenu des fichiers. Malheureusement, le temps perdu dans la lecture séquentielle de cette structure est un sérieux handicap, du fait du nombre important de tris à opérer pour l'établissement des images.

Il est donc malgré tout raisonnable d'encombrer la mémoire de tableaux gargantuesques en très grande partie vides.

Le compromis choisi a été de copier les cellules occupées d'un tableau de grande taille dans un tableau ayant exactement le bon nombre de cellules et d'utiliser le *ramasse miettes* de la machine virtuelle Java pour récupérer l'espace mémoire précédemment occupé par les tableaux surdimensionnés.

C. L'intégration des parsing

L'affichage des fichiers de parsing est très proche de celui des données brutes, à l'exception que les triggers utilisés ne sont pas des événements avec un code type, qualifier, mais des temps déterminés par un algorithme statistique dans le but de mettre en évidence des motifs de potentiels d'action au cours du temps. Malgré tout, ces deux types d'affichage ont tant de choses en commun qu'il paraît normal de les faire cohabiter dans le même outil de visualisation.

Les fichiers de parsing posent à nouveau le problème de la gestion de la mémoire, dans des proportions encore plus grandes. Le nombre et la taille des fichiers à lire sont inconnus. On aurait pu imaginer de stocker en mémoire toutes les données provenant de tous les fichiers référencés pour limiter les accès disques ou réseau, mais étant incapable de prédire le volume de données représenté, on prenait le risque de ne pouvoir tout simplement pas afficher le contenu d'un fichier de parsing, simplement parce que la mémoire était insuffisante pour stocker les données.

La solution choisie, bien que pas entièrement satisfaisante, est celle de n'avoir en mémoire qu'un fichier à la fois, dans l'ordre de leur apparition dans le fichier de parsing. Ainsi, le premier fichier est-il lu, les données intéressantes triées avant que le tableau des temps des événements ne soit écrasé par celui correspondant au fichier suivant à lire. Il en découle qu'à chaque fois qu'un affichage sera généré, il faudra relire tous les fichiers référencés dans le fichier de parsing. Seuls restent en mémoire les noms des fichiers, et les temps relatifs.

IV. IMPLÉMENTATION

Le langage de programmation retenu pour ce travail a été Java. Il s'agit d'un langage orienté objet récent qui connaît depuis cinq ans une grande popularité. Plusieurs aspects propres et souvent uniques en ont fait le langage choisi pour le développement de SPANAKE. Pour ne citer que quelques-unes unes de ces particularités :

- ➢ une syntaxe claire et simple,
- ➢ le paradigme objet fortement implanté,
- ➢ la portabilité du code écrit pour une machine virtuelle, interprété par la machine hôte,
- ➢ l'existence d'un nombre déjà considérable de librairies de classes bien organisées et surtout bien documentées,
- ➢ des outils pour l'utilisation des réseaux intégrés, comme les protocoles de transfert ou la compression et décompression transparente des fichiers avant et après le passage sur le réseau,
- ➢ la possibilité d'interfacer des bouts de code écrits en C ou C++, permettant d'intégrer des algorithmes optimisés pour certaines plates-formes afin de réduire les temps d'analyse.

Cependant, Java possède quelques défauts découlant presque tous de sa relative jeunesse, et tout particulièrement à son manque d'optimisation. L'interprétation du code pour une machine virtuelle Java coûte cher en terme de vitesse.

De manière générale, l'utilisation des récents composants `swing` permet d'ajouter sur les éléments de l'interface graphique des petites bulles d'aide qui apparaissent à l'écran lorsque le pointeur de la souris stationne au-dessus de l'élément plus d'une seconde. Il s'agit d'un moyen simple pour communiquer à l'utilisateur la fonction d'un bouton sans qu'il ait besoin pour cela d'ouvrir un fichier d'aide. Finalement, la fonction `Look&Feel` permet de choisir l'apparence de l'interface graphique, quel que soit le système d'exploitation sur lequel tourne l'application.

A. La gestion de la mémoire.

L'utilisation de tableaux pour le stockage de l'information pose le problème draconien de la gestion dynamique de leurs tailles. Le compromis choisi ici est celui de stocker les données dans des tableaux de grande taille. Mais lorsque la fin du fichier est atteinte, l'ajout dans chacun d'entre eux d'une valeur donnée – une sorte d'étiquette `EOF` –, déclenche la copie des seules cellules occupées dans des tableaux qui ont exactement les bonnes tailles pour les recevoir.

Des mesures du taux d'occupation de la mémoire ont montré que ce mécanisme permettait de récupérer beaucoup de place en faisant un appel explicite au *garbage collector* de la machine virtuelle Java à des instants stratégiques. Les avantages de rapidité d'accès aux données propre aux tableaux peuvent être conservés avec pour seuls coûts la copie des cellules, et le *ramassage des miettes*. Il s'agit, pour donner un ordre de grandeur, de copier quelques centaines de cellules pour en récupérer plusieurs dizaines de millier.

B. Gestion de l'affichage

Chacun des rectangles que l'on voit sur la fenêtre principale (Figure 7) correspond à un objet de type `Stamp` héritant de `JPanel`, c'est à dire un container particulier qui s'occupe de manière autonome de produire une image à partir des données qui lui sont fournies.

À chaque fois que le contenu d'une image change, le `Stamp` qui l'affichait et jeté puis remplacé par un nouveau auquel on fournit les bonnes données et paramètres. Dans le cas de la visualisation d'un simple fichier de données brutes, ce processus demande à ce que le tableau des temps en mémoire soit trié avec les nouveaux paramètres. S'il s'agit d'un fichier de parsing, il faut alors relire tous les fichiers pour pouvoir en trier le contenu au fur et à mesure.

Si on ne fait que changer la taille de l'image, par exemple en passant le `RasterViewer` en plein écran, l'image est simplement recalculée à partir des données qui lui avaient été fournies.

Le scrolling des rasters au moyen du curseur ne nécessite aucun calcul, puisque le `Stamp` possède en mémoire une image deux fois plus large que celle visible à l'écran. Il lui suffit donc d'en afficher une autre portion, en fonction de la position du curseur sur sa course.

C. Gestion des parsing

Les fichiers de parsing ne sont composés en fait que de noms de fichiers de données brutes situés dans le même dossier, et de valeurs de temps particulières à l'intérieur de ceux-ci.

```
[...]  c1lc04.002  7  29690  [...]
[...]  c1lc04.002  7  37032  [...]
[...]  c1lc04.031  7  15495  [...]
```

Il est normal de chercher à afficher les fichiers référencés par le même mécanisme que s'ils étaient ouverts explicitement. Aussi, à la lecture d'un fichier de parsing, leurs noms et les temps associés sont-ils enregistrés dans un tableau de petite taille. Ils sont lus, l'un après l'autre, par les mêmes méthodes que celles appelées lors d'une ouverture formelle du fichier, aboutissant à des tableaux de temps en mémoire. On peut alors facilement rechercher dans ces tableaux les temps auxquels le fichier de parsing fait référence, et afficher à l'écran les événements qui les entourent sous forme de raster de points.

Ce mécanisme est rappelé pour chaque fichier référencé, c'est à dire que le tableau des temps du fichier en train d'être lu écrase le tableau du fichier précédent. A tout moment, il n'y a au plus en mémoire que les données contenues dans un fichier de données brutes.

La classe qui gère tous ces mécanismes – `DataFile` – est une classe abstraite qui demande à ce qu'on mette en œuvre deux méthodes – `readNextEventDataFromFile (StreamTokenizer)` et `readNextParsedFileFromFile (StreamTokenizer)` – qui se basent sur la grammaire des fichiers à lire. Elles sont appelées respectivement lorsqu'il faut lire un fichier de données brutes et un fichier de parsing. Ainsi, cette classe demande à être héritée pour pouvoir lire un type de fichier différent. Dans notre cas :

```
public class AbelesDataFile extends DataFile
```

La méthode `readNextEventDataFromFile` rend à chaque appel un objet de type `eventData`. Celui-ci possède quatre attributs : `type`, `qualifier`, `mask` et `time` permettant par la suite de le manipuler en mémoire comme s'il s'agissait de la triplette lue dans le fichier.

Quant à la méthode `readNextParsedFileFromFile`, utilisée pour lire un fichier de parsing, elle retourne une instance de `parsedFile`, possédant deux champs : `fileName` et `triggerTimes`, le second étant un tableau de taille précise contenant les temps absolus déterminés par l'analyse statistique. Ce temps a pour origine – temps 0 – le début du fichier.

C'est également au niveau de la classe fille héritant de `DataFile` que doivent être initialisées les constantes de taille comme `MAX_EVENTS_KINDS`, `MAX_PARSED_FILES` et `MAX_HEADER_COMMENTS` fixant le nombre maximum de codes type, qualifier différents, de fichiers référencés dans un fichier parsing et celui des commentaires en tête des fichiers de données brutes respectivement, servant bien évidemment à l'initialisation de la taille des tableaux respectifs.

D. Intégration de l'orienté objet dans le projet

Le paradigme objet offre une manière élégante de modéliser des solutions logicielles à des problèmes concrets, comme l'affichage de données par exemple. La programmation orientée objet est cependant difficile à mettre en œuvre et demande beaucoup de réflexion pour pouvoir en tirer vraiment partie. Les objectifs qu'il faut impérativement rechercher sont la modularité et la réutilisation du code.

Pour ce projet, le parti pris a été celui de l'utilisation de l'héritage de manière à regrouper dans des interfaces et des classes mères le maximum d'information pour qu'elles puissent être mises en œuvre et héritée de manière efficace. Ainsi, si à l'avenir on souhaitait à nouveau réécrire un `RasterViewer`, on pourrait récupérer par l'interface `RasterViewerInterface` les noms des méthodes et des variables nécessaires au bon fonctionnement de la fenêtre `RasterViewerToolkit` ou comme vu plus haut, pour pouvoir lire de nouveaux types de fichier.

`RasterViewer` est elle même une classe, dont l'unique constructeur – `RasterViewer (URL sourceURL, int fileTypeConstant, int fileKindConstant)` – permet de lui fournir directement le chemin du fichier à lire et des constantes entières qui définissent respectivement le type de fichier et son type de contenu.

Le fichier à ouvrir étant défini par son URL, on peut profiter pleinement des outils réseaux de Java, car quel que soit le protocole de transfert utilisé (*http*, *ftp* ou *file*), la gestion se fera de manière transparente, qu'on soit en local ou sur le réseau.

Voici un exemple de la construction d'un objet du type `RasterViewer` :

```
new RasterViewer (  new URL (http://www-iis.unil.ch/7bb.tim),
                    RasterViewer.ABELES,
                    DataFile.RAW_DATA_FILE   );
```

Ce constructeur s'occupe alors d'ouvrir, de lire et d'afficher les données brutes structurées dans un fichier Abeles situé à l'adresse fournie.

Pour permettre l'affichage d'autres types de fichiers – le type Abeles est actuellement le seul implémenté – il faudrait effectuer les modifications suivantes :

➢ modifier la classe `RasterViewer` en ajoutant une constante et un nouveau cas au `switch` du constructeur :

```
public RasterViewer ( URL fileURL,
                      int fileType,
                      int fileKind  ) {
  [...]
    public static final int ABELES   = 0;
    public static final int NEW_TYPE = 1;      // à ajouter
  [...]
  switch (fileType) {
    case ABELES : {
      file = new AbelesDataFile (fileURL,
                                 fileKind);
      break;
    }
    case NEW_TYPE : {                          // à ajouter
      file = new newTypeDataFile (fileURL,     // à ajouter
                                  fileKind);   // à ajouter
      break;                                   // à ajouter
    }                                          // à ajouter
    default : {
      System.out.println (
        "This file type hasn't been defined yet"
      );
      System.exit (-1);
    }
  }
  [...]
```

➢ Ecrire la classe `newTypeDataFile` **`extends`** `DataFile` qui définirait les deux méthodes `readNextEventDataFromFile (StreamTokenizer)` et `readNextParsedFileFromFile (StreamTokenizer)`

S'il fallait dans le futur faire face à d'autres types de contenu que les deux actuellement implémentés – `DataFile.RAW_DATA_FILE` et `DataFile.PARSING_FILE` – il faudrait alors faire des changements plus conséquents dans la classe `DataFile`. Cela est malheureusement inévitable. Cependant, la tâche serait facilitée par les nombreux `switch` répartis dans le code aux endroits stratégiques dans la décision du comportement à adopter selon le type de contenu.

V. CONCLUSION

Les buts fixés pour ce projet ont été pleinement atteints. La gestion de la mémoire donne de bons résultats, et le travail avec les fichiers de parsing est totalement transparent pour l'utilisateur, mis à part les quelques aspects qui lui sont intrinsèquement spécifiques, comme l'absence de '`master trigger`'. En outre, l'interface graphique a reçu un excellent accueil de la part des premiers testeurs, laissant préfigurer que l'outil pourrait s'avérer utile. Ce n'est pas la moindre des choses !

Sur le plan de la petite mécanique, les algorithmes de tri devront encore être optimisés pour rendre le programme plus rapide. Sur le plan esthétique, la GUI demande encore à être normalisée, en particulier en ce qui concerne les polices de caractères. Certains boutons gagneraient certainement en clarté si une petite icône y était ajoutée. Quelques fonctionnalités, comme l'impression ou l'ouverture d'un autre fichier dans la même fenêtre, restent à mettre en œuvre, ainsi que l'intégration concrète dans SPANAKE. Celle-ci ne devrait cependant pas poser de réels problèmes.

Pour ce qui est du `RasterViewer` en tant que partie du projet SPANAKE, il ne peut pas être considéré comme terminée tant qu'il n'aura pas été validé par des personnes extérieures à l'équipe de conception. Un travail d'affinement du produit est nécessaire afin qu'il réponde pleinement aux attentes des scientifiques qui pourraient l'utiliser dans leurs travaux. Les leçons tirées de cette version comme des précédentes permettront probablement de rendre cette phase aisée.

VI. ANNEXE A

DATA STRUCTURE OF SPIKE DATA FILES IN A STANDARD FORMAT

THIS INFORMATION WAS ORIGINALLY PROVIDED BY M. Abeles
Dept of Physiology, School of Medicine
The Hebrew University of Jerusalem
9 AUG 1991

General.

1. DATA STRUCTURE

The data is coded by triplets of numbers.
Two numbers describe "EVENTS"
The third describes the time of occurrence. The time is expressed as an interval from the previous event.

Example: 1 1 43 1 3 17 1 5 0 1 2 11 ...

It would read: event no. 1,1 occurred 43 msec after the recording started, then event no. 1,3 occurred 17 msec later, then event no. 1,5 occurred within less then a msec after event no.3, then event no. 1,2 occurred 11 msec later etc...

Using milliseconds as the units for time measurements can be changed, so that any other time units could be used. If nothing is said about time units they are assumed to be milliseconds.

2. SEPARATORS

<carriage returns>,<line_feeds>, blanks , tabs are allowed between numbers in any desired combination.

Example: the data in the following LINE:

1 1 43 1 3 17 1 5 0 1 2 11

can also be written as:

1 1 43
1 3 17
1 5 0
1 2 11
. . .

OR:

It is also allowed to use a comma (,) as a separator between numbers, as well as a combination of a comma and the other separators. Thus, the same data may be encoded by:

1,1,43 1,3,17 1,5,0 1,2,11
. . .

NOTE: two commas are the same as a number with value of 0. So that the following text:

1,1,,43 1,3,17
1,5, ...

would read: Event no. 1,1 occurred 0 msec after the start of file, then event no. 43,1 occurred 3 msec later, then event number 17,1 occurred 5 msec. later etc...

3. COMMENTS

In addition comments can be interspersed anywhere by enclosing them in single quotes ('). For instance the exact data as in the previous examples may look like:

1,1,43 1,3,0 1,5,0
'This is a true coincidence between units 1,3 and 1,5'

4. KEYWORDS

Some keywords have special meaning and they are put in double quotes ("). For instance the following file contains the same data as the previous one but coded in 100 microseconds units:

"TIME_UNITS = 0.0001"
1,1,430 1,3,170 1,5,0
'This is a ...

5. CODES

The meaning of the first pair of number in each data item (triplet) is always a code for some event that happened. This might be a spike that fired, a stimulus that was presented, an analog voltage that was measured (as will be described later) or any other thing that the user wishes to define and mark its time of occurrence in the file.

It is advisable to attach some significance (taken from the experimental situation) to the two numbers that describe the events. For instance the first number may be used to describe the electrode number (or track number) and the second number may be used to describe the single unit recorded from that electrode (in that track). Or, the first number may be used to describe one parameter of the stimulus (such as its frequency), while the second number may be used to describe another parameter (such as its intensity).

In the discussion that follows we shell refer to the first number of the event pair as the event type and to the second number as the event qualifier.

Each of the two numbers in the event code is described by 1 to 4 hexadecimal digits (0,1,2,...9,A,B,C,D,E,F). The user can assign any code number he wishes to any event type (the first number), except for the numbers 0. Events having the form 0,nnn are reserved for special purposes as described below.

Event number 0,0 (which may be written also 00,0 , or 0,00 etc...) means the null event. It could be used to describe very long intervals or to mark the time at which a comment is recorded in the file. Two examples will illustrate these usages.

Suppose you wish to limit yourself so as to use at most two digits to specify the interval between successive events. This might be the case if you are using FORTRAN to generate your spike_data file and you wish to use a fixed format specification I2 for the times. What if you have occasionally more than 99 msec without any event? It would be a waste of space to code all the times in 3 digits if long periods of silence occurred rarely. You could use than the empty event (0,0) to describe this silence. For instance, a file that starts:

1,1,47 1,5,32 0,0,99 1,2,17 ...

means that event no. 1,1 occurred 47 msec after we started to record, then event no. 1,5 occurred 32 msec, later and then event no. 1,2 occurred 116 (99+17) msec later.

Another use of the empty event is to mark in the file the point at which something which is not an event occurred. Suppose that you wish to record the time at which you inserted an important comment in the file. You may do it in the following way:

7,1,47 1,5,32 0,0,65
'at this point electrode no. 7 got dislocated from the cell' 1,2,11 ...

This record will be interpreted as saying that 65 msec after event no. 1,5 occurred the comment was inserted, while event no. 1,2 occurred 76 msec (65+11) after event no. 1,5.

Event number 0,2 is used to state that data collection stopped in this point of time. Event number 0,1 means that data collection was resumed. These two codes are useful if the data in the file are not continuously collected, but made of many short runs of data streams (for instance if only activity around a stimulus is collected). If the first data triplet in the file is not 0,1,0 it is assumed that this code exists there. Thus two files that start:

3,1,167 ... ,and
0,1,0 3,1,167 ...

are equivalent.

Events 0,11,xxx 0,12,xxx 0,13,xxx are special events which are used when combining files. This may happen if the users wishes to take activity around a given stimulus from several files and make a new file which contains only the selected sections from the original files. This are not generally useful, but are included here for the sake of completeness.

Event number 0,11: Marks the start original file. It is used when a new file is generated from a number of old files. It gives the position at which the data from the new (original file) started. Typically it is followed by the name of the new file. e.g.

0,11,0
"TITLE(0) = 'v20s.022'"

Event number 0,12: Marks an end of original file. It is used when a new file is generated from a number of old files. It gives the time from the last event until the end of the original file. e.g.

0,12,323

Event number 0,13: defines long period with no events. Typically, when generating a new file from pieces of old files, it is used to fill the space in which there was data in the old files but we wish to ignore it in the new file. e.g.

0,13,5000

The code 0,FFFF is reserved to state the end of file. Note that while all other event qualifiers can have any number of (hexadecimal) digits the end of file code is specified as having 4 digits exactly. Anything that appears in the file after the time delay associated with 0,FFFF is ignored. If the end of file code (0,FFFF) is not preceded by the end of data collection code (0,2), it is assumed that data was collected up to the end of the file. That is, two files that end by:

... 3,1,67 0,FFFF,29 ,and
... 3,1,67 0,2,29 0,FFFF,0

are equivalent.

A complete file of data may look like that:

"TIME_UNITS=0.001"

'1,n are spikes recorded through electrode no.1'
'3,n are spikes recorded through electrode no.3'
'A,01 is the onset of a 200 msec noise burst'
0,1,0
1,1,17 3,2,3 1,2,11 1,3,3 1,3,1 1,3,2 1,2,17
1,4,22 A,01,3 3,2,2 1,2,4 1,2,1 1,2,3 1,2,5
1,4,13
0,2,7 0,FFFF,0

This file reads as follows: In the first line we see that the units of time in this file are given in msec. The second, third and fourth lines are comments that reminds the experimenter what he was doing. Note that the programs that will analyze these data will ignore these comments. In the fifth line we see that data collection had started. Actual data starts in the sixth line and says that spike 1,1 fired 17 msec after we started the recording, spike 3,2 fired 20 (17+3) msec after we started the recording, spike 1,2 fired at time 31 (17+3+11) msec, etc... The next line tells us that spike 1,4 fired 22 msec after spike 1,2 (the last event on the previous line) and then after additional 3 msec a noise burst was sounded. Then we get some more spike data, and then the last line states that 7 msec after the last spike (1,4) we stopped the recording, and then this file ends. Note that as much as we need to know how much time elapsed from the beginning of the recording until the first event occurred, we also need to know how much time elapsed from the last recorded event until we stopped our measurements.

As is obvious from the foregoing description, the times of events are coded in decimal form (while the events were defined in hexadecimal form). There are two reasons for this seemingly awkward dichotomy. Although it would be advantageous to code all numbers in hexadecimal (because this would save some space), the user of these files might find hexadecimal times hard to decipher, especially if he wanted to run some checks on the programs that analyze the data by computing some sample data by hand and comparing the results with the computer results. A more important reason for not coding times in hexadecimal is the danger of interpreting an FFFF time delay as an end_of_file if an extra comma(,) is inserted into the file or if one of the numbers is dropped during communication. By reserving the hexadecimal codes to events we reduce somewhat this danger. On the other hand events could be coded in many ways so that there is no special advantage in using decimal notations.

To this basic scheme we have to add few features that will deal with error detection, general management, and including analog data in our file.

6. CHECKSUM

At first we shall deal with questions of error detection. One of the main reasons for coding spike data in ASCII is portability of such files between computers. However if one tries to transmit such files (which are bound to be very long) through serial lines, errors are likely to occur. Some communication programs have means to detect (or even correct) such errors. However in most cases, particularly if we think of exchanging files between different kinds of computers, this is not available. We introduce, therefore, some means to assure the integrity of our data. One of the most simple ways to test for integrity is to include a 'checksum' every now and then in the data. In our case we do the following: Starting from the beginning of the file we add up all the values of the characters to each other in 16 bits. We do not include in the checksum the end_of_line codes, the blanks, the tabs and the comments. The reasons for not including end_of_lines, blanks and tabs is that different I/O systems treat these characters differently and we do not wish to generate errors because of these different treatments. The reason for not including comments will become apparent shortly. If the added values overflow (i.e. they become bigger then FFFF hexadecimal) we take the remaining value and keep adding the characters. Every now and then we include this computed checksum in our file by inserting:

"CHKSM = value"

Where value is the hexadecimal representation of the computed checksum coded in ASCII. To fix our ideas let us look at a very small example, assume that our file starts:

1,1,4 1,2,17
...

and we want to include there the checksum. We proceed in the following way: The file starts with several blanks. These are ignored. Note that some I/O system will drop out the first character in a line (assuming that it is a

carriage control of a printer), it is therefore wise to start every line with at least one blank. The first non blank character is 1 whose hexadecimal code is 31, so we assign the value of 31 to CHKSM, then we have a comma whose value in ASCII is 2C (hexadecimal), we add it to the checksum to get 5D. Then we have another 1 whose ASCII value is 31 (hexadecimal), therefore we add 31 to CHKSM then we add 2C (for the next comma) then 34 (for 4). The two blanks that follow the 4 are ignored, then we add 31 (for the next 1), and so on until we reach the end of the line. We ignore the end-of-line character because some I/O system will use <carriage- return> to designate end-of-line, some will use <line-feed> and some will use both.

Assume that we wish to put the CHKSM here. We could just insert an additional line giving the checksum. We would get:

1,1,4 1,2,17
"CHKSM = 211"
...

Note that the characters in the line which states the checksum are within a comment and therefore not included in the computation. If we wanted to include also the "CHKSM=hhh" text in the checksum we would run into some complex computational problems. This is one of the reasons for not including comments in checksums. The other reason is to enable the owner of the file to add later any number of comments without having to worry about updating the checksums.

Of course after writing down the checksum in the file we restart to calculate it from the first character in the next line.

The places at which the checksums are placed are arbitrary and they can be put as frequently as the quality of the communication line dictates.

In the future error correcting codes may be added to this scheme.

7. VERSION AND TITLES

Management codes include (in addition to CHKSM) the version number of the code, and the titles. Version number should appear as the first item in the file. It informs the interpreting procedures what kind of statements they can expect to find in the file. The version described here is version 0. Therefore any data file that adheres to the standards proposed here should start with:

"VERSION = 0"

If the "VERSION" clause is not included version 0 is assumed. Titles are a special class of privileged comments. They are inserted into the file by:

"TITLE = 'any text you wish to type'"

or by
"TITLE(n) = 'some other important comment'"

where n can be any decimal number from 0 upward. When the (n) suffix is not added it is understood that this is title number 0. The text of the title is enclosed in single quotes, while the entire title is enclosed in double quotes. The text within the single quotes can span several lines.

The purpose of inserting comments into the data file by the TITLE keyword is to enable you, later on, to specify which titles will be included in the histograms (or other graphs) that will be computed from your data. For instance you may wish to include information about date, track number, and stimulus conditions in such titles. Your data file may look like:

"VERSION = 0"
"TITLE(0) = '12/12/85'"
"TITLE(1) = 'Track III'"
"TITLE(2) = 'moving grating

at 5 deg/sec'"

The analyzing programs should enable you to state later which titles you wish to include with the results of the analysis of the data.

8. ANALOG DATA

Analog data should be avoided as far as possible. If one wishes to record an EMG for deciding when a certain muscle starts to move it is advisable that he decides when the muscle starts to move before preparing the data file, assign an event code to that initiation of movement and record in the file only the times of movement initiation (and not the EMG itself). When this is not possible it is advisable to use only occasionally analog data. For instance if one is interested in saccades (rapid shifts of gaze), it would be wise to give a code to the event of the saccade and to include in the file only the fact that a saccade occurred (with its time) and to follow this code by two analog codes specifying the eye position coordinates to which the eyes moved.

When this is not possible, and the experiments calls for "continuous" sampling of analog data, one should attempt to include only the pieces of data which are of interest (e.g. scalp potentials around the time of a stimulus) and not the entire, uninterrupted, stream of samples. One should always bear in mind that long sequences of sampled, analog, data are bound to increase enormously the size of the file.

Analog data are recorded too by triplets of numbers. The first number (the event type) identifies the channel from which data was recorded, the second number (the event qualifier) identifies the value of the sampled data. If the second number is negative it has to be put as the number that complements it to 10,000 hexa (e.g -1 will be written FFFF). Because of this difference the file must state all the codes which will be used for analog signals.

For example the following lines:

"TIME_UNITS=0.001"
"ANALOG = A1"
"ANALOG = A2"

states that event types A1 and A2 are for analog recordings. The third number is the time interval like usual.

Let us look at the following file which includes both analog and spike data:

"VERSION=0"
"TIME_UNITS=0.001"
"ANALOG=A1"
"ANALOG_UNITS(A1)=0.000001"
'event 1,1 is a code for a spike'
'event A1 is a code for EEG recording'
0,1,0 1,1,72 1,1,49
A1,24,17 A1,2,5 A1,FFE0,5 1,1,3 A1,FFC4,2 ...

After the initialization and comments we read the following data: spike 1,1 fired 72 msec after we started the recording, then 49 msec later it fired again, then after 17 msec we started to sample our A1 analog input (at a rate of every 5 msec). The value of the first sample (sampled at time 17 msec after spike 1,1 fired) was 24 u_volts, 5 msec later we sampled again and got 2 u_volts, 5 msec later we sampled again and got -20 u_volts, 3 msec after this last sample spike 1,1 fired again, and then 5msec after the previous sample (i.e. 2 msec after spike 1,1) we sampled again and got -3B u_volts.

Note that since we made all the event qualifiers hexadecimal, the voltage of the analog channels is also in hexadecimal.

9. SYNOPSIS.

The Data_file is made of a list of constants. A constant is a string of hexadecimal (or decimal) digits or a string of text enclosed by single(') or double (") quotes. Constants are separated from each other by separators. Separators are blanks,tabs, end-of-line codes, or any combination of the above, or a comma(,), or a combination of a comma and any of the other separators.

The information in the Data_File is taken to be of two major types: Events and Comments. Events are made of triplets of numeral constants, the first two of which are coded in hexadecimal and the third in decimal. Comments are strings of text which are used for remarks and for defining parameters and terms to the analyzing programs.

9.1 EVENTS

Each Event is composed of three numbers two of which are the event code and the third one is the event time.

The event code is made up by two hexadecimal numbers. The first number specifies the events type, while the second number is the events qualifier.

Event types have three meanings: Event type 0 is a control type. All other values are either point events or analog channel numbers. An analog channel number is any hexadecimal constant hhhh that has been declared to be an analog channel by: "ANALOG = hhhh". Event types which are not 0 and are not declared as analog channels are point events.

The control event may have 4 event qualifiers (0,1,2 and FFFF), which have the following significance:
0,0 means an empty event,
0,FFFF means the end of file,
0,1 means the recording has started,
0,2 means the recording was stopped.

The event qualifier that follows an analog event is its analog value, coded in hexadecimal form. The number that follows a point event code is a qualifier for that code that may be used to describe the code in more details.

9.2 COMMENTS

Are recognized by being embedded in quotes (single or double). Comments that appear in between single (') quotes are not interpreted by any data processing programs. Comments that appear between double quotes (") must be of the form "KEYWORD = VALUE". They are used to control the operation of the interpreting programs.

Recognized KEYWORDS are

VERSION To define the version no. of the file.
TIME_UNITS A scaling factor that converts all the time information given later into seconds.
ANALOG To define analog events.
ANALOG_UNITS(xx) A scaling factor that converts all the analog values given later for channel xx into volts.
CHKSM To specify a 16 bit checksum.
TITLE(n) Any string that may be used later as a title for the display of results computed from the data in this file.

VERSION must appear at the start of a file.

ANALOG = xx must precede the "ANALOG_UNITS(xx) = yy" comment.

The CHKSM is computed by adding together the ASCII codes (evaluated in hexadecimal) for all the characters in the file except:
blanks,tabs,end-of-lines and the comments.

The definitions given here are for "VERSION = 0".

VII. BIBLIOGRAPHIE

Une liste partielle des lectures qui m'ont accompagnées, avant et pendant l'élaboration de ce projet :

La programmation orientée-objet et le langage Java

A Java GUI programmer's primer
Fintan Culwin, Prentice Hall, 1998, ISBN : 0-13-908849-0

Object-oriented software construction, 2ème édition
Bertrand Meyer, Prentice Hall, 1997, ISBN : 0-13-629155-4

Principles of object-oriented programming in Java 1.1
James W. Cooper, Ventana, 1997, ISBN : 1-56604-530-4

The Java tutorial : object oriented programming for the internet, 2ème édition
Mary Campione and Kathy Walrath, Addison Wesley 1998
http://java.sun.com/docs/books/tutorial/information/download.html

Les aspects neurophysiologiques

Corticonics : neural circuits of the cerebral cortex
Mosche Abeles, Cambridge University Press, 1991, ISBN 0-521-37617-3

Enregistrements unitaires dans le cortex auditif d'un rat soumis à une tâche de discrimination auditive de type GO/NOGO
Alexandre Kuhn, Travail de certificat de Neurophysiologie dirigé par Dr Villa, 1997

Neurobiology, 3ème édition
Gordon M. Shepherd, Oxford University Press, 1994, ISBN 0-19-508843-3

La gestion de projet

Stratégies pour développer juste (*Software project survival guide*)
Steve McConnell, Microsoft Press, 1997, ISBN : 2-840-82262-8

The mythical man-month : essay on software engineering, 20ème anniversaire
Frederick P. Brooks, Jr, Addison Wesley, 1995, ISBN : 0-201-83595-9

Spanake

Computer assisted neurophysiology by a distributed Java program
Luc Jeandenans, Computers and Biomedical Research, Academic Press, 1998, pp.465-475

VIII. REMERCIEMENTS

Par ordre chronologique des événements qui m'ont menés à pouvoir faire ce travail, je voudrais tout d'abord remercier le professeur R. Wittek, conseiller aux études de la section de Biologie, pour m'avoir immédiatement soutenu dans ma démarche. Viennent ensuite le professeur M. Tomassini, de l'IIS, pour avoir accepté de me chapeauter et m'avoir fait confiance – même s'il n'y avait pas de raison apparente de le faire – ainsi que le docteur Villa, du LNH, pour avoir trouvé un sujet à la fois passionnant et à ma mesure... pourvu que ça dure ;–)

Que le Conseil de Section de Biologie se voit également remercié d'avoir autorisé un de ses étudiants à sortir du droit chemin le temps d'un semestre de spécialisation. J'ose espérer que d'autres pourront profiter de cette aubaine pour arpenter des couloirs différents et acquérir des connaissances dans des domaines proches et alliés tels que l'Informatique ou la Physique. Ces domaines prometteurs pour la Biologie pâtissent de l'absence de gens à leurs interfaces. Mon ambition est bien de me former à cette tâche.

Je voudrais également remercier l'équipe de l'IIS de m'avoir offert un cadre de travail amical et agréable. Tout spécialement L. Jeandenans, pour ses conseils avisés et son regard critique qui m'ont été d'un grand secours, en particulier pour ce mémoire, ainsi que le Dr M. Gautero que la proximité géographique de nos bureaux et la profondeur de son puits de science ont transformé en source de ravitaillement vitale lors de ma traversée du désert. Tous mes vœux pour son nouveau poste !

Finalement, une pensée pour mes parents qui me soutiennent depuis toujours, mes amis qui me supportent et pour Caroline...

IX. TABLE DES FIGURES

Les captures d'écran ont été réalisées sur un système Microsoft Windows 98 sur lequel était installée la version 1.2 du JDK de Sun Microsystems.

X. TABLE DES MATIÈRES

CONTACTS

Javier Iglesias
Av. du Temple 8
CH-1020 Renens
Tél. : ++4121 635 54 90
mailto : javier.iglesias@etu.unil.ch

Université de Lausanne
Institut d'Informatique
Collège propédeutique
CH-1015 Lausanne
Tél. : ++4121 692 35 80
Fax : ++4121 692 35 85

Laboratoire de Neuro-heuristique
Institut de Physiologie
Faculté de Médecine
Université de Lausanne
Rue du Bugnon 7
CH-1015 Lausanne
Tél. : ++4121 692 55 32
Fax : ++4121 692 55 05

v1.0 – 01