Rapport de Soutenance n^{o}3

2B2S

Thomas 'billich' De Grivel
Maxime 'loucha_m' Louchart
Bruno 'Broen' Malaquin
Julien 'Splin' Valentin

Patchwork13!

Table des matières

Chapitre 1

Introduction

Voici venue la troisième soutenance de **Patchwork13 !**, notre puissant outil de synthèse modulaire. Rappellons briévement le but du projet : permettre la génération et la manipulation de données en temps réel par la connection de *plug-ins* que nous appellons des **patches**, cela très simplement et à la souris.

Nous développons également un système de clustering pour répartir les patches sur différentes machines afin de multiplier les performances, cela sans même que l'utilisateur aie à configurer quoi que ce soit à part lancer le *daemon* sur les machines qu'il souhaite utiliser. Le but est qu'en tirant des fleches à la souris, l'utilisateur controle de manière transparente tout les serveurs.

1.1 Planning

Notre planing a été largement respecté tant au niveau du cluster que de l'interface, des patches SDL et l'OpenGL et des patchs son. Cela a été facilité par l'avance que nous avions prise par exemple sur les patches OpenGL, mais aussi compliqué par les nouvelle fonctionnalités que nous avons rajoutées en plus de celles décrites dans notre cahier des charges. Cela comprend l'exportation et l'importation dans des fichiers XML et l'avancée majeure que représente la possibilité d'associer aux patches des *méthodes* qui leurs sont propres.

De plus, l'enregistrement et du chargement des patchworks permet de ne plus perdre de temps lorsqu'on veut reprendre un patchwork et/ou le modifier, apportant un grand confort d'utilisation, surtout pour les tests ce qui accélère fortement le développement.

Concernant le cluster, l'implémentation des messages réseau que nous avions prévus est complète, ce qui permet à Patchwork13 ! d'être complètement utilisable à travers un réseau TCP/IP.

Pour la SDL, nous avons maintenant une sortie sons qui fonctionne parfaitement et nous permet, couplé avec les patchs sons crées, de pouvoir déjà produire un rendu sonore très satisfaisant.

1.2 Expérience

Entre ces deux soutenances, beaucoup de travail a donc été abattu. Certains ont découvert le confort des salles machines du sous-sol Pasteur, dont la température se prête tout à fait à un court mais réparateur sommeil lorsque les joies de la programmation cédaient face aux charmes de Morphée.

Le ryhtme de travail est donc vraiement bien installé et il n'y a aucune raison pour que cela ne continue pas à s'améliorer.

Chapitre 2

Méthodes et fonctions de rappel

2.1 Introduction

Pour pouvoir ajouter des fonctions spécifiques à une application ou à quelques patches, nous avons introduit le concept de **methode de patch**. C'est une fonction liée à un patch et identifiée par une chaine de charactères. La librairie noyau de Patchwork13! dispose désormais de fonctions d'ajout d'une méthode à un patch, de suppression et d'appel.

2.2 Avantages

Un des atouts majeurs de notre manière de gérer ces méthodes est que l'on peut associer plusieurs fonctions à un même nom. L'appel de la méthode consiste alors à appeller successivement toutes les méthodes associées au même nom de méthode. Nous avons donc pour chaque patch une liste chainée de noms de méthodes, chaqun contenant une liste chainée de pointeurs de fonctions.

Ces fonctions sont donc des *fonctions de rappel* et l'on peut ainsi associer différentes fonctions à différents patches et ces fonctions seront appellées lors dévenements définis par lápplication.

2.3 Exemple

Par exemple, dans l'interface graphique nous appellons la methode nommée "`gtk build interface`" sur chaque patch lors de sa création pour qu'il puisse éventuellement créer ses contrôles. Si le patch n'a pas associé de fonction de rappel pour ce nom de méthode, alors il ne se passe rien. Si il a défini une (ou

plusieurs) fonctions pour cette méthode alors elles sont appellées avec en paramètre le *widget* GTK parent pour que le patch crée ses controles dedans.

Nous utilisons aussi des méthodes pour gérer l'enregistrement et le chargement d'informations propres au patch.

2.4 Généralisation

Dans le chapitre suivant nous décrivons en détail comment nous avons été amenés à **généraliser** cette notion de méthode pour pouvoir également l'appliquer à un patchwork (un graphe de patches). Nous n'utilisons pour l'instant que des méthodes sur les patchs et sur les patchworks mais le programmeur peut utiliser nos fonctions pour définir des méthodes sur d'autres objets et utiliser toutes les fonctionnalités décrites plus haut.

Chapitre 3

Importation et exportation en XML

Un autre ajout à la librairie noyau est l'utilisation du format XML pour importer et exporter un patchwork (et toutes les structures qui le composent) dans un fichier. Nous disposons désormais de fonctions d'imporation et d'exportation pour chaque structure de patchwork13.

3.1 Représentation d'un graphe

Une étape intéressante a été la structure de graphe du patchwork puisqu'il faut enregistrer les sommets triés topologiquement pour pouvoir enregistrer les arcs comme des liens vers le parent. En effet, si lors de l'importation un sommet (un patch) est relié par un arc à un autre patch qui n'a pas encore été importé, nous allons vers de gros problèmes. Le tri topologique du patchwork résoud simplement ces problèmes puisque le graphe peut alors être représenté comme une foret.

3.2 Informations supplémentaires

Le problème s'est largemement complexifié lorsque nous avons voulu enregistrer pour chaque patch un peu plus que les informations du noyau dans le même fichier XML.

3.2.1 Exportation

Pour l'interface graphique, nous voulions enregistrer la position de chaque patch dans la fenêtre. Nous avons donc défini une méthode `"xml export patch"` pour chaque patch lors de sa création dans l'interface et nous l'avons associée à la fonction d'importation d'un patch de l'interface graphique.

Il faut peut-être rappeller que si un patch définissait déjà une méthode `"xml export patch"`, sa fonction puis celle de l'interface seront appellées successivement. Toutes les fonctions seront appellées avec un même paramètre d'appel, dans notre exemple c'est une structure utilisée pour l'exportation, mais aussi avec un paramètre de méthode propre à chaque fonction.

Il faut évidemment que chaque fonction exporte une balise XML ayant un nom différent pour pouvoir retrouver ses informations lors de l'importation. Un patch devra ainsi exporter une balise dont le nom contiendra son type complet (*eg.* `<std_float_sinus>` ou `<gtk_float_const>`). Les patches peuvent alors exporter toutes les informations qu'ils désirent.

L'interface graphique exporte une balise `<pw13_gtk>` ne contenant pour l'instant que les attributs `x` et `y` indiquant la position du patch.

3.2.2 Importation

Lors de l'importation de ces données, le fichier XML est parcouru pour importer chaque patch qui est d'abord initialisé. C'est à ce moment qu'il peut ajouter sa fonction d'importation à la méthode "xml import patch". La méthode est alors appelée et les informations propres au patch sont alors récupérées.

Il faut que la fonction d'importation propre au patch parcoure la liste des balises de données supplémentaires à la recherche de sa balise au nom unique pour différentier ses données de celles des autres fonctions.

Le problème

Il y a pourtant une erreur dans cette manière simpliste de récupérer les données du patch : l'interface graphique n'a dans ce processus aucun moyen d'associer sa fonction d'importation au patch puisque celui-ci vient d'être créé par la fonction d'importation du patchwork.

La solution

Il faut donc associer la méthode de récupération des données du patch non pas au patch qui n'existe pas encore mais à un autre objet qui déjà alloué lors de límportation. Le patchwork étant l'objet directement au dessus du patch, il était assez naturel de le choisir pour y associer la méthode.

Le patch recoit comme paramètre d'initialisation le patchwork le contenant. L'adaptation a donc été assez immédiate.

Le futur problème et sa solution

Le problème a été résolu pour les patches mais il risque d'y avoir le même pour importer des patchworks. La solution future à ce problème futur est d'associer les méthodes liées à l'importation des données à la structure utilisée par le noyau pour l'importation (appellée `pw13_import`).
On pourra alors récupérer des informations propres à chaque objet importé sans se soucier de l'ajout des méthodes d'importation à l'objet en train d'être importé.

3.2.3 La généralisation des méthodes

Finallement, nous avons au cours de ce chapitre détaché la notion de méthode de l'objet "patch" pour pouvoir l'appliquer à tous les types d'objets. Nous fournissons pour la convénience des fonctions de manipulation de méthodes pour les patches et les patchworks mais il est vraiement très simples de les adapter à d'autres types de données.

L'application aussi bien que le noyau ou les patches peuvent tous définir des méthodes sur des objets, et les appeller ce qui permet une adaptation phénoménale de Patchwork13 ! à tous les problèmes, et cela sans avoir à modifier les objets, les librairies ni les programmes qui ne sont pas concernés par telle ou telle application.

Chapitre 4

Patches OpenGL

4.1 Presentation

Les patchs de bases d'OpenGL de patchwork13 sont finis et permettent d'effectuer ce que l'on esperait d'eux. Soit de faire de la creation 3D en temps réel.

4.2 Finalisation

Par rapport à la dernière soutenance, où j'avais pris de l'avance dans ce domaine, il ne me restait à priori pas grand chose à faire. Tout avait été transcrit, il a donc fallu juste déboguer les problèmes apparus lors de l' utilisation des patchs et permettre enfin l' utilisation des fonctions de bases de l'Opengl. Cela permet désormais de créer des démos 3d en temps réel. On peut créer un triangle et un carré donc un cube et une pyramide soit toutes les formes que l'on veut à partir de ces deux la, changer la couleur courante, effacer le buffer de couleur et de profondeur, remettre la vue au centre, translater et rotationner. C'est assez amusant à vrai dire de faire de la création en temps réel, le système marchant avec fiabilité on peut s'amuser à compliquer l'oeuvre à la volée pendant le pump :)

Chapitre 5

Cluster

5.1 Descriptif du protocole réseau

Tout d'abord il faut au minimum un client et un serveur (normal ...). Une fois lancés, le client envoye un paquet permettant aux serveurs de se faire connaître, et ceci via un envoi de messages sur l'addresse de broadcast. Ensuite, le client initialise une connection TCP avec le(s) serveur(s) identifié(s). Par la suite, le server peut envoyer des requêtes d'éxécution de commandes via l'envoi d'un int correspondant à une fonction, qui, lorsqu'il est reç par le server, entre dans un case pour identifier et lancer la fonction désirée.

5.2 Fonctions disponibles à travers le cluster

- Une fonction de creation de patch
- Une fonction detruisant un patch
- Une fonction qui demande à un server de demarré un patch précis qu'il héberge.
- Une fonction permettant l'arrêt d'un patch
- Une fonction connectant localement deux patchs
- Une fonction permettant de connecter des patchs se trouvant sur des machines différentes
- Une fonction qui demande de pomper à tous les prédécesseurs du patchs visé.
- Une fonction permettant l'envoi de buffers
- Une nouvelle fonction send

5.3 Descriptif des fonctions et des mécanismes du cluster

- Lors de la demande de création de patch par le client, ce dernier envoie au server l'entier lui permettant de savoir qu'on lui demande cette tâche, ainsi que le chemin du patch, et pour finir son nom. Avec celà, le server utilise une fonction du noyau pour créer le patch demandé. Par a suite, le server renvoye au client un pointeur sur le patch qu'il vient de créer pour détenir une liste des patchs lancés sur chaque server.

- Pour detruire un patch, le client envoye l'ID du message correspondant au server, ainsi que le pointeur sur le patch voulu. La suppression devient effective grâce à un appel sur une fonction du noyau.

- Pour demander à un client de démarrer un patch, il suffit de préciser, en plus de l'ID du message, le temps auquel il faut lancer le patch.

- Arrêter un patch nécessite simplement d'envoyer un pointeur sur le patch concerné au client, qui sera arrêté grâce à un appel sur une fonction du noyau.

- Le transfert de buffer qui a été ecrit permet l'envoi de n'importe quel type de donnée ('a). Le principe est d'utiliser des unions, représentant chacune un bloc du buffer. Notre buffer est donc un enchaînement d'unions dans la mémoire. Pour connître la taille á recevoir, le premier block contient la taille totale du buffer.

- La fonction permettant de connecter deux servers entre eux est plus compliqué qu'un simple appel des fonctions du noyau. Il a fallu se tourner les meninges serieusement pour la pondre. Elle doit permettre de rendre transparent la connection de deux patchs entre deux serveurs. Le serveur 1 par la demande du client va etablir une connection sur le serveur 2 classiquement en tcp puis lui envoyer les données nécessaires soit le nom de l'output et le pointeur du patch. Ensuite il faut crér un patch permettant le transit des données entre les deux patchs puis le connecter au patch du serveur 1 côté input du patch. Ce patch a donc une sortie qui renvoit n'importe quelle donnée et qui permet de tout transiter avec les fonctions d'envoi et de réception de grosse données. Pour le pump tout va se faire en transparent, lorsque le patch précédent demandera un pump du fameux patch alors il demandera de pumper avec un message réseau au patch du serveur 2 puis se mettra en attente des données qui arriveront sur la sortie du path du serveur 2 et qui ressortiront directement sur l'output du patch de transmission.

- La connection localement de deux patchs fonctionnent en prenant un pointeur et une chaîne par sortie et entrée à connecter, cela permet de trouver le pointeur sur l'entrée et la sortie nécessaire à la fonction du noyau permettant de les relier.

- Étant donné que la fonction send d'UNIX ne garantit pas que toutes les données ont été emvoyées (embetant si on a des pertes lors d'envoi de gros buffers), nous avons dû nous appuyer sur cette fonction pour en créer une nouvelle permettant de tout envoyer.

5.4 Ce qu'il est possible de faire grâce au cluster

Au moment où nous rédigeons ces lignes, nous sommes capables de faire fonctionner plusieurs patchs sur différents server, le tout contrôlé par le client. Par exemple (ce qui vous à été présenté à cette soutenance), on peut faire jouer un son sur tous les ordinateurs d'epita en même temps, ou encore de mixer du son et que chaque server du cluster héberge un patch, ou plusieurs patchs appartenant au patchwork du client.

5.5 Ce qu'il reste à faire

Le cluster est à ce jour opérationnel pour tout ce dont nous avons besoin, mais nous pourrons par la suite implémenter des fonctions supplémentaires, comme par exemple, gérer plusieurs patchworcks par clients etc...

Chapitre 6

Patchs et traitement du son

6.1 Traitement du son

6.1.1 sound_out

Afin de reussir à sortir un son grâce à Patchwork13, nous utilisons le patch *sound_ out*. Une technique unique au monde créée afin de pouvoir jouer n'importe quel son à partir de son signal.

Le patch est composé de plusieurs parties. Chacune de ces parties est essentielle à la gestion de la sortie audio et nous a posé quelques problèmes afin d'avoir un son propre.

La fonction de pump.

Comme decrit précédemment, le patch prend en paramêtre un short. Plus précisément pour jouer un son audible et respectable, il nous faut un nombre de samples (shorts) minimum par seconde. Ici, on initialise cette frequence a 44100, il nous faudra donc au moins 44100 short/s. Le concept du patchwork nous permet de gérer le son ainsi.

Dans cette fonction, on ajoutera chaque sample reçu dans un buffer qui sera lu dès que l'on aura assez de samples pour avoir du son. La gestion des buffers est similaire à une file à laquelle on a relié la tête avec la queue. La technique de 2 buffers alterne a été abandonnée pour des questions d'optinmisation.

Une fois, le premier buffer rempli, on demarre la lecture dessus et on continue de remplir les autres afin de ne pas avoir de sauts dans la sortie audio.

La difficulté ici a été de trouver un systeme pour gérer les buffers. La solution utilisée fonctionne. et nous permet de traiter tous types de données.

La fonction de callback.

Cette fonction est propre a la bibliotheque SDL, et plus précisement SDL_Audio. Elle est utilisée dans la lecture du son et sert à remplir le buffer audio avec les données a lire. Chaque buffer (tableau de samples) lut y est passé en paramêtre et doit etre rempli pour la lecture. Afin de le remplir, on utilisera les buffers rempli pendant la phase de pump.

Notre probleme ici a été de comprendre le concept de la fonction et adapter notre format de données à celui demandé par SDL. Encore une fois, le probleme est resolu et la lecture du son est naturellement jolie.

Les fonctions init, start, stop, destroy.

Dans ces fonctions, on prépare les buffers à recevoir des données, on charge le patch et on le detruit en liberant la memoire.

6.1.2 wav_out

Ce patch de son, dependant de la SDL_Audio, est utilisé afin de charger un fichier wav dans le but de le jouer via le patch *sound_out*.

Les fonction init et start.

Ces fonctions sont appelées pour charger le fichier en mémoire et mettre les informations de positions, durée... à zero. On prepare le patch à decomposer et à envoyer les samples du fichier un par un. Les informations sur le sample y sont recuperées afin de pouvoir re-sampler, c'est á dire mettre au bon format audio, si nécessaire.

La fonction de pump.

Pourquoi envoyer les samples un par un? Tout simplement pour pouvoir traiter le signal du wav comme le résultat d'une fonction en fonction du temps. le re-sampling du son se fait automatiquement grâce á la gestion du patchwork dans le temps. grâce à des techniques similaires, on peut lire un wav et créer du son en même temps.

Les problèmes posés par ce patch ont été le sampling, résolu, et la gestion du volume de sortie du wav. Ce dernier a été résolu par l'ajout d'une entrée floatante permetant de faire varier l'amplitude.

6.2 Patchs de son

Afin de pouvoir commencer á obtenir un rendu intéressant, nous avons codés plusieurs patchs de traitement du son.

Premièrement, un patch de saturation qui a pour fonction de limiter un son en amplitude. Ce patch prends donc deux flotants en entrée, reprensentant la valeur de saturation maximale, ainsi que la valeur de saturation minimale. Cet effet de saturation est trés rependu et utilisé dans de nombreux styles musicaux.

Ensuite, un patch de delay qui joue un pourcentage du son courant ainsi que le son qui a été joué il y a un temps passé en input. Évidemment, on peut changer le pourcentage en live, ainsi que le delay. Tout comme la saturation, le delay est enormément utilisé lors de compositions musicales. D'un point de vue algorithmique, on crée un buffer d'une taille égale au nombre de samples de delay que l'on remplit avec l'entrée. On effectue en sortie une multiplication de l'input par le feedback (un pourcentage). Une fois qu'on a recu un nombre de samples égal au delay, on renvoye comme valeur le sample courant multiplié par le feedback plus le sample qui a été joué au temps t (t étant valant le temps courant moins la valeur de de delay).

Par la suite, nous avons élaborés un patch de sratch (comme sur des platines de DJ et dans la plupart des logiciels de son). Ce patch est composé d'une barre de scroll agissant sur le son joué, comme la main d'un DJ sur son vynil. D'un point de vue pratique, plus on deplace le scroll dans un sans ou dans l'autre, plus le temps recule ou avance. Pour finir nous avons crée un mixer, nous permettant de mixer plusieurs pistes audios á la fois avec un fader associé pour plus de précision. Pour être plus clair, toutes les éntrés voient leurs valeurs multipliées par le coefficient du fader toutes sommés pour finir.

Nous avons aussi crée un patch permettant d'appliquer une fonction replay, qui remet son prédecesseur au temps de référence. Nous utilisons pour l'instant ce patch relié à des patchs son mais il est à noter que c'est notre premier patch polymorphique (c'est à dire que le type des entrées et des sorties est 'a (comme en Caml!). La demande de replay se fait en utilisant la gestion des signaux GTK.

Il nous reste à faire une multitude de patchs sons pour la prochaine soutenance afin d'avoir des possibilités sonores étendues et présenter un rendu très apréciable pour la soutenance finale.

Chapitre 7

Interface graphique

7.1 Le patchwork

7.1.1 Nouvelles fenetres

Sauvegarder à la fermeture

Petite fenêtre supplémentaire pour les têtes en l'air. À la fermeture d'un patchwork, il y a maintenant une fenêtre de validation demandant si l'on ferme, on sauve ou on annule. Simple mais efficace.

On remerciera la magie de l'informatique. Merci.

7.2 Les patchs

7.2.1 Suppression

Mieux que l'ajout de patch, visible a la soutenance precedente, la suppression des patchs.

Et oui, nous pouvions ajouter des patchs dans le patchwork grace a un glisser-deposer lors de la derniere soutenance. Nous pouvons maintenant les supprimer grace a un clique de la molette sur le titre du patch.

Grace a une technologie avancée, nous avons meme la possibilitee d'enlever un patch pendant qu'un patchwork est lancé. Il a fallu modifier quelques éléments de la librairie noyau, qui est du coup beaucoup plus robustes.

7.3 Les connections

Mieux que les connections, les deconnections..

En effet, il est interessant de pouvoir connecter deux patchs mais il est aussi tres important pour l'utilisateur de pouvoir enlever des connections. Ca y est ! il a le choix de soit deconnecter sans trace, c'est a dire sans garder la fleche, en droppant l'entree, soit de garder une trace, fleche grisée, juste en cliquant sur l'entree ou la sortie. La reconnection se refait simplement en tirant le lien.

A l'avenir, il sera possible de desactiver une connection et de la reconnecter en un clic. Une amelioration pour la version finale du Patchwork13.

Chapitre 8

Conclusion

8.1 Les accomplissements

Patchwork13 ! à la troisième soutenance nous procure beaucoup de satisfactions étant donné qu'il est dès lors utilisable que nous nous sommes amusés avec pendant déjá plusieurs heures à faire des sons vraiement étranges, et à les modifier en *live*. Nous en avons d'ailleurs déjá réalisé un au BDE qui à été assez apprécié, cela pendant plus de vingt minutes sans interruption, ce qui prouve la stabilité du projet.

Aprés deux semaines de pseudo vacances passées á développer notre projet, nous avons beaucoup avancé vers ce que nous ttendons de patchwork13 !. Nous pouvons créer des patchwork, y déposer des patchs alors que l'on joue, les relier entre eux, les suprimer, sauvegarder des patchwork, les charger, mixer du son et de la 3d en live localement ou á traver le réseau... en somme beaucoup de choses !

8.2 Et le futur.. !

Il ne nous reste plus qu'à terminer l'interface graphique en y intégrant le cluster, clarifier le code de certaines fonctions du réseau et écrire un grand nombre de patchs pour éventuellement séduire des artistes en herbe et d'autres programmeurs pour qu'ils contribuent au projet en écrivant des patches !

Le portage ne devrait pas poser de problème puisque nous développons très régulièrement sur (liste non exhaustive) sur architecture PowerPC, sous Gentoo et Mac OS X / Darwin, et sur architecture x86 sous Gentoo, Debian, NetBSD, OpenBSD. Le portage windows était fonctionnel il y moins d'un mois et ne devrait pas être trop compliqué puisque nous n'avons pas rajouté beaucoup de librairies (libxml2).

Un programme d'installation pour Windows est presque prêt. Pour UNIX nous fournissons le format classique des packages autoconf/automake (`./configure ; make ; sudo make install`) qui est compatible avec tous les *NIX que nous avons pu tester.

L'ambiance du groupe est toujours au beau fixe nous permettant d'être confiants sur le travail à effectuer pour la soutenance finale. Le rythme de travail n'a cessé de s'accélérer depuis le début de l'année et ne devrait pas échapper à cette loi pour ces deux derniers mois. Définitivement prometteur.

MERCI