Manuel d'utilisation

Fluxus 0.16

Table des matières

Introduction

Fluxus est un environnement qui vous permet de créer des animations en temps réel et des programmes audio, que vous pourrez manipuler de manière flexible. Fluxus tire son nom de l'idée que l'on peut modifier en continu le code, de manière « fluide ».

Fluxus s'appuie sur le langage de programmation Scheme, qui est très flexible; et une interface qui met à votre disposition une zone d'édition de code qui flotte au dessus du résultat graphique que produira votre programme. Cette interface permet à Fluxus d'être utilisé pour du livecoding, l'art de programmer en temps réel. La plupart des utilisateurs de Fluxus sont des livecodeurs, et certains programment devant des spectateurs, mais il est tout à fait possible de l'utiliser pour du prototypage rapide de fonctions. Cela rend Fluxus plus convivial pour l'apprentissage et la compréhension de la programmation d'animations et de contenu graphique – d'ailleurs il est souvent utilisé lors de

séminaires sur le sujet.

Ce manuel d'utilisation est vaguement organisé autour du fait que ce que vous devez absolument savoir est au début et les parties plus complexes sont ensuite abordées vers la fin, mais comme il ne requiert pas vraiment d'avoir lu le début, je vous recommanderais de sauter aux parties qui vous intéresse.

Première prise en main

Au lancement de Fluxus, vous verrez apparaître un message d'accueil et une invite de commande – ceci est appelé le toplevel, ou terminal. En général les scripts Fluxus seront écrit dans des tampons de mémoire, desquels vous pourrez vous déplacer avec la combinaison de touches Ctrl-[0-9] (le tampon 0 étant votre toplevel).
Déplacez vous dans le premier tampon en pressant simultanément Ctrl et 1. Essayez maintenant de taper la commande suivante:

```
(build-cube)
```

Pressez F5 (ou Ctrl-e) – Le script serra exécuté, et un cube blanc devrait apparaître au milieu de l'écran. Utilisez la souris pour naviguer autour du cube, en pressant sur les boutons de la souris, vous pourrez effectuer plusieurs mouvements.

```
; la taille du tampon et la fréquence d'échantillonnage doivent
; correspondent aux paramètres de Jack
(start-audio “port-jack-sur-lequel-lire” 256 44100)

(define (render)
       (colour (vector (gh 1) (gh 2) (gh 3)))
       (draw-cube))

(every-frame (render))
```

Pour expliquer rapidement, la fonction (every-frame) prend en paramètre une fonction qui serra appelée toute les secondes par le moteur interne de fluxus. Dans notre cas elle rappelle la fonction (render) qui elle va définir une couleur en fonction des harmoniques du flux audio d'entrée (avec la fonction (gh)) et dessine ensuite un cube de la couleur spécifiée. Vous remarquerez que l'on utilise pas la fonction (build-cube) mais (draw-cube), l'explication à cela est plus bas.

Maintenant essayez de manipuler les exemples. Vous pourrez les ouvrir en pressant la combinaison de touches Ctrl-l ou directement à partir du terminal , en donnant un script d'exemple en paramètre de Fluxus.

Guide d'utilisation

Quand vous utilisez les tampons de Fluxus, l'idée est qu'on aura besoin d'une

seule fenêtre pour l'écriture de scripts, et leurs exécution. Quand vous aurez fini de taper votre script, lancez le avec F5 (ou Ctrl-e). Si vous sélectionnez une portion du code (avec Maj), et que vous exécutez votre script, seulement le code sélectionné serra évalué. C'est très pratique pour évaluer quelques fonctions sans avoir à relancer tout le script à chaque fois.

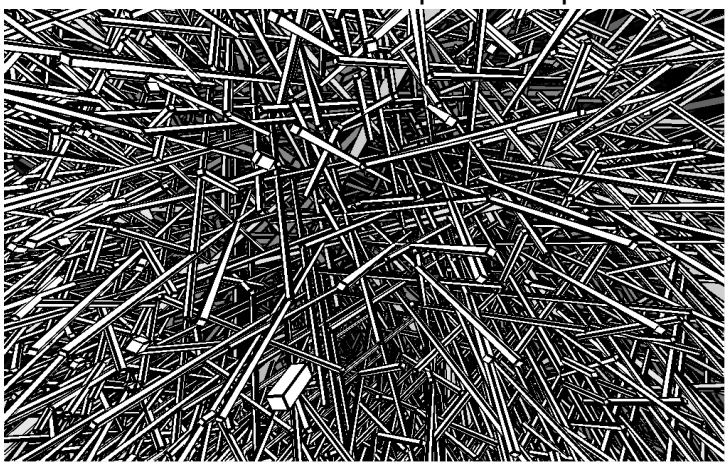

Illustration 1: un mikado programmé

Contrôle de la caméra

Le contrôle de la caméra se fait à l'aide de la souris et de ses boutons.

- Bouton gauche: Rotation
- Bouton du milieu: Mouvement transversal
- Bouton droit: Zoom

Espaces de travail

L'éditeur de scripts vous permet d'éditer jusqu'à 9 scripts simultanément grâce aux différents espaces de travail. Pour naviguer entre scripts, utilisez Ctrl-numéro. Néanmoins un seul script pourra être lancé à la fois, F5 exécutera le script de l'espace de travail courant.

Les scripts de vos différents tampons peuvent être enregistrés sous différents noms, pour cela, appuyez sur Ctrl-s pour le sauvegarder, ou Ctrl-d pour spécifier un nom a votre script avant de l'enregistrer. (le nom par défaut avec Ctrl-s est temp.scm)

La boucle TOPLEVEL

Si vous faites Ctrl-0, au lieu d'avoir un espace de travail, vous aurez une invite de commandes. Vous pourrez entrer du code directement évalué en tapant Entrée. Ce code serra évalué par le même interpréteur qui s'occupe des scripts, vous pouvez donc l'utiliser pour debugger vos scripts. C'est aussi sur la sortie de la boucle TOPLEVEL que les erreurs de vos scripts seront envoyés, à l'instar de la console qui a servi à lancer Fluxus.

Une des utilisations principales de la boucle TOPLEVEL est l'utilisation de l'aide sur les commandes de Fluxus. Par exemple, pour avoir de l'aide sur la commande build-cube, tapez:

```
(help "build-cube")
```

Vous pouvez avoir plus de détails en tapant:

```
(help "sections")
```

qui vous retournera une liste de sous-sections, par exemple pour avoir un détail des fonctions mathématiques:

```
(help "maths")
```

Vous renverra la liste de toutes les fonctions mathématiques disponible, et vous pourrez en plus demander le détail de chaque fonction. Vous pouvez copier l'exemple donné par l'aide en utilisant les touches du clavier (Ctrl-c, Ctrl-v) et naviguer sur un autre espace de travail pour y coller le code.

Les raccourcis clavier

- Ctrl-f : Mode plein écran.
- Ctrl-w : Mode fenêtré.
- Ctrl-h : Affiche/masque l'éditeur.
- Ctrl-l : Ouvre un script (navigation avec les flèches du clavier).
- Ctrl-s : Sauvegarde le script courant (sous le nom temp.scm).
- Ctrl-d : Sauvegarde sous: le nom spécifié.
- Ctrl-1 to 9 : Naviguer entre les différents espaces de travail.
- Ctrl-0 : Aller à la boucle TOPLEVEL.
- Ctrl-p : Auto-indentation du code.
- F3 : Reset de la caméra.
- F4 : Exécute le code sélectionné.

- F5/Ctrl-e : Exécute le code sélectionné, ou tout le script si rien n'est sélectionné.
- F6 : remise à zéro de l'interpréteur, puis exécution du script.
- F9 : Change la couleur du texte (avec une couleur aléatoire).
- F10 : Rendre le texte plus transparent.
- F11 : Rendre le texte moins transparent.

Scheme

Scheme est un langage de programmation inventé par Gérald J. Sussman et Guy L. Steel Jr. en 1975. Scheme est basé sur un autre langage (Lisp, qui remonte aux années 50). C'est un langage de haut niveau, ce qui signifie qu'il est simplifié pour l'utilisateur, au lieu de l'être pour le compilateur. Le tampon de Fluxus embarque un interpréteur Scheme (il peut faire tourner des scripts Scheme) et les modules de Fluxus étendent le langage Scheme de par ses commandes pour le graphisme 3D.

Ce chapitre fait une brève introduction à la programmation Scheme, et vous introduira vite dans Fluxus, même si vous n'avez aucune connaissance spécifique. Pour un apprentissage plus en profondeur du langage Scheme, je vous recommande vraiment les livres suivant (dont deux sont en libre consultation sur internet):

« The Little Schemer » de Daniel P. Friedman et Matthias Felleisen

« How to Design Programs »

« An Introduction to Computing and Programming » de Matthias Felleisen, Robert Bruce Findler Matthew Flatt Shriram Krishnamurthi qui est en ligne: http://www.htdp.org/2003-09-26/Book/

« Structure and Interpretation of Computer Programs » de Harold Abelson et Gérald Jay Sussman avec Julie Sussman qui est en ligne: http://mitpress.mit.edu/sicp/full-text/book/book.html

Nous commencerons par quelques bases du langage, on utilisera la boucle TOPLEVEL de Fluxus en y accédant par Ctrl-0.

Utiliser Scheme comme calculateur

Les langages comme Scheme sont composés de deux types d'éléments: les opérateurs (les éléments qui font quelque chose), et des valeurs (sur lesquelles les opérateurs travailleront).

```
fluxus> (+ 1 2)
3
```

Ça peut paraître étrange au début et on ca peut prendre un peu de temps à comprendre, mais cela implique que le langage est soumis à moins de règles, ce qui rendra la programmation plus aisée au fur et à mesure. Il y a heureusement des avantages à cela, par exemple pour l'addition de 1, 2 et 3, on ferrait simplement:

```
fluxus> (+ 1 2 3)
6
```

Il est assez fréquent de voir des parenthèses imbriquées dans d'autres parenthèses, comme par exemple:

```
fluxus> (+ 1 (* 2 3))
7
```

Nommer des valeurs

Si nous voulons spécifier des valeurs et leur attribuer un nom, on utilisera la commande define:

```
fluxus> (define taille 2)
fluxus> taille
2
fluxus> (* taille 2)
4
```

Nommer des valeurs est sans doute la partie la plus importante de la programmation, et représente la forme la plus simple d'abstraction, ce qui veut dire que l'on sépare les détails du sens (dans notre exemple taille). Pour la machine cela ne fait aucune différence, mais ça en fait pour les personnes qui liront votre code. Par exemple ici, on a spécifié une bonne fois pour toute la valeur de la taille, et dans notre code nous nous y réfèrerons seulement par son nom, ce qui rend le code plus facile à lire.

Nommer des fonctions

Nommer des fonctions est très utile, mais on peut aussi nommer des fonctions (ou un ensemble de fonctions) pour nous faciliter la tâche:

```
fluxus> (define (carré x) (* x x))
fluxus> (carré 10)
100
fluxus> (carré 2)
4
```

Observez attentivement cette définition, il y a plusieurs choses à prendre en compte.
Premièrement, on pourrait traduire en français la définition de la fonction par:

« définissons la fonction carré qui à x associe son carré »

Ce « x » est appelé un argument de notre fonction, et son nom importe peu à la machine, donc:

```
fluxus> (define (carré toto) (* toto toto))
```

ferra exactement la même chose. Encore une fois, il est préférable de donner un nom évocateur à l'argument, auquel cas vous vous embrouilleriez très vite. On vient de faire abstraction des opérations, comme nous l'avons fait pour les valeurs. On peut voir ça comme « ajouter » du vocabulaire au langage Scheme avec nos propres mots, alors maintenant que l'on a notre fonction carré, on peut l'utiliser pour faire d'autres opérations:

```
fluxus> (define (somme-des-carrés x y)
       (+ (carré x) (carré y)))
fluxus> (somme-des-carrés 10 2)
104
```

les retours à la ligne et les espaces avec le define ne sont juste qu'une convention de mise en page, cela veut dire que vous pourrez séparer visuellement la description et ses arguments du reste de la fonction. Scheme ne fait pas attention aux blancs dans le code, le but est encore une fois de rendre le code compréhensible.

Dessiner quelques figures

Maintenant nous en savons assez pour pouvoir dessiner quelques formes avec Fluxus. Pour commencer, sortez de la boucle TOPLEVEL avec Ctrl-1, vous pourrez y revenir à n'importe quel moment avec Ctrl-0.
Fluxus est maintenant en mode édition de scripts. Vous pouvez écrire un script et l'exécuter en tapant F5, puis le modifier, puis l'exécuter, etc... c'est la façon usuelle de coder avec Fluxus.

Entrer ce script:

```
(define (render)
       (draw-cube))

(every-frame (render))
```

Puis tapez F5, vous devriez voir apparaître un cube à l'écran, tout en maintenant appuyé un bouton de la souris, faites la glisser: cela aura pour effet de faire bouger la caméra.

Ce script définit une fonction qui dessine un cube, et qui l'appelle grâce à la commande every-frame, d'où le cube qui est apparu à l'écran.

Vous pouvez changer la couleur du cube comme cela:

```
(define (render)
       (colour (vector 0 0.5 1))
       (draw-cube))

(every-frame (render))
```

La commande colour fixe la couleur courante et prend un vecteur en argument (on pourra appeler ça un tableau aussi). Les vecteurs sont beaucoup utilisés dans Fluxus pour représenter des positions et des directions en 3D, mais aussi pour représenter la couleur sous le triplet <rouge, vert, bleu>. Dans notre exemple le cube devrait prendre une couleur bleu clair.

Les transformations

Rajoutez la commande scale à votre script:

```
(define (render)
        (scale (vector 0.5 0.5 0.5))
        (colour (vector 0 0.5 1))
(draw-cube))

(every-frame (render))
```

Votre cube devrait avoir rapetissé. Pour mieux le mettre en évidence, comparons le à un autre cube:

```
(define (render)
        (colour (vector 1 0 0))
        (draw-cube)
        (translate (vector 2 0 0))
        (scale (vector 0.5 0.5 0.5))
        (colour (vector 0 0.5 1))
        (draw-cube))

(every-frame (render))
```

Vous devriez voir apparaître deux cubes, l'un rouge et l'autre bleu, mis côte à côte (avec la commande translate) et avec sa taille réduite de moitié.

```
(define (render)
        (colour (vector 1 0 0))
        (draw-cube)
        (translate (vector 2 0 0))
        (scale (vector 0.5 0.5 0.5))
        (rotate (vector 0 45 0))
        (colour (vector 0 0.5 1))
        (draw-cube))

(every-frame (render))
```

Pour en rajouter encore, j'ai rajouté une rotation de 45° sur le cube bleu.

La récursivité

Pour faire des choses plus intéressantes, nous allons écrire une fonction qui dessine une ligne de cubes. Cela serra par récursivité, c'est à dire que notre fonction va s'appeler elle-même, elle gardera en compte le nombre de fois qu'elle s'est appelée et pourra s'arrêter au bout d'un certain nombre de fois.

Pour que la fonction arrête de s'appeler nous devons établir une condition, pour cela nous allons utiliser cond:

```
        (define (ligné-de-cubes compteur)
                (cond
                        ((not (zero? compteur))
                                (draw-cube)
                                (translate (vector 1.1 0 0))
                                (draw-row (- compteur 1)))))
```

```
        (every-frame (ligné-de-cubes 10))
```

Faîtes attention aux parenthèses, l'éditeur de Fluxus devrait vous aider en mettant en surbrillance la région contenue par chaque paire de parenthèses. En exécutant ce script, vous devriez voir apparaître une ligné de 10 cubes. Vous pouvez imaginer beaucoup de choses en utilisant ce concept, alors prenez le temps de bien le comprendre.

cond est utilisé pour établir des conditions en posant des questions, et vous pourrez en poser autant que vous souhaitez. cond les vérifiera dans l'ordre donné, et exécutera la première action dont la question rend vrai. Dans notre script on ne pose qu'une question: « est ce que le compteur n'est pas nul ? ». Si c'est vrai, c'est que le compteur est supérieur à zéro, alors on dessine un cube, on se déplace un petit peu, et on s'appelle encore. Il est important de remarquer qu'au prochain appel de la fonction, on soustrait 1 à notre compteur, il arrivera un moment ou l'on appellera la fonction avec un compteur à zéro, dans ce cas, la fonction arrête de dessiner.

En résumé, ligné-de-cubes est appelée la première fois avec un compteur à 10 par every-frame. On entre dans la fonction: « est ce que le compteur est à zéro ? » non, alors on dessine on bouge un peu, et on rappelle ligné-de-cubes avec un compteur à 9, etc... et à la fin, on aura dessiné 10 cubes.

La récursivité est un concept très puissant, très adapté au graphisme. Il est de plus très utilisé pour construire des scripts de graphiques très complexes avec un code pas si énorme que ça au final.

Animation

Bien, maintenant qu'on est arrivé jusque là, on va pouvoir faire bouger notre script:

```
(define (ligné-de-cubes compteur)
        (cond
                ((not (zero? compteur))
                        (draw-cube)
                                (rotate (vector 0 0 (* 45 (sin (time)))))
                                (translate (vector 1.1 0 0))
                                (ligné-de-cubes (- compteur 1)))))

        (every-frame (ligné-de-cubes 10))
```

time est une fonction qui retourne le temps (en secondes) qui s'est écoulé depuis le lancement de Fluxus. sin le converti en une vague sinusoïdale et une multiplication y est appliquée pour se ramener à un angle qui varie de -45° à +45° (rappel: sinus varie sur [-1, 1]). Votre ligne de cubes devrait se tordre de haut en bas dans un mouvement perpétuel. Essayez de changer le nombre de cubes et la portée de l'angle de 45°.

Encore plus de récursivité

Pour vous donner un exemple visuellement plus intéressant, ce script s'appelle deux fois, il en résulte une sorte d'arbre animé.

```
(define (ligné-de-cubes compteur)
       (cond
              ((not (zero? compteur))
                     (translate (vector 2 0 0))
                     (draw-cube)
                     (rotate (vector (* 10 (sin (time))) 0 0))
                     (with-state
                            (rotate (vector 0 25 0))
                            (ligné-de-cubes (- compteur 1)))
                     (with-state
                            (rotate (vector 0 -25 0))
                            (ligné-de-cubes (- compteur 1))))))

       (every-frame (ligné-de-cubes 10))
```

Le détail de la fonction with-state est dans la section suivante.

Les commentaires

En Scheme, les commentaires sont insérés grâce au caractère « ; » :

```
; ceci est un commentaire
```

Tout ce qui suit « ; » jusqu'à la fin de la ligne ne serra as lue par l'interpréteur.

De même en utilisant « #; » vous pourrez commenter toute une expression en Scheme, par exemple:

```
(with-state
       (colour (vector 1 0 0))
       (draw-torus))
(translate (vector 0 1 0))
#;(with-state
       (colour (vector 0 1 0))
       (draw-torus))
```

Cela empêchera l'interpréteur d'exécuter la deuxième expression with-state, ce qui annulera aussi l'affichage du tore vert.

Let

let sert à stocker des résultats temporaires, par exemple:

```
(define (animate)
    (with-state
        (translate (vector (sin (time)) (cos (time)) 0))
```

```
        (draw-sphere))
    (with-state
        (translate (vmul (vector (sin (time)) (cos (time)) 0) 3))
        (draw-sphere)))

(every-frame (animate))
```

Ce script dessine deux sphères en orbite autour de l'origine de la scène. Vous remarquerez qu'il y a un petit calcul qui est réalisé deux fois. Il serrait plus simple et plus rapide de faire une fois pour toute ce calcul et de le stocker quelque part. Une manière de le faire:

```
(define x 0)
(define y 0)

(define (animate)
    (set! x (sin (time)))
    (set! y (cos (time)))
    (with-state
        (translate (vector x y 0))
        (draw-sphere))
    (with-state
        (translate (vmul (vector x y 0) 3))
        (draw-sphere)))

(every-frame (animate))
```

C'est mieux, mais x et y sont globalement définies et peuvent être utilisées et changées autre part dans le code, ce qui peut prêter à confusion. Le meilleur moyen est d'utiliser let:

```
(define (animate)
    (let ((x (sin (time)))
          (y (cos (time))))
        (with-state
            (translate (vector x y 0))
            (draw-sphere))
        (with-state
            (translate (vmul (vector x y 0) 3))
            (draw-sphere))))

(every-frame (animate))
```

Cela n'autorise l'accès à x et y seulement aux fonctions à l'intérieur du let. Il peut aussi être imbriqué à l'intérieur d'autres let, qui dépendrait de valeurs définies en amont. Par contre si vous définissez des variables (ou valeurs) dépendantes d'autres variables, il faut utiliser let*.

```
(define (animate)
    (let* ((t (* (time) 2)) ; on définie le temps par t
          (x (sin t))  ; on utilise t ici
          (y (cos t))) ; et encore ici
        (with-state
            (translate (vector x y 0))
```

```
            (draw-sphere))
         (with-state
            (translate (vmul (vector x y 0) 3))
            (draw-sphere))))

(every-frame (animate))
```

Lambda

Même si ce nom vous fait peur, son utilisation est plutôt facile à comprendre. D'habitude quand vous créez une fonction, vous lui donnez un nom, et ensuite dans le script vous l'appelez:

```
(define (carré x) (* x x))
(display (carré 10))(newline)
```

lambda vous permet de créer et d'utiliser une fonction temporaire à un point donné dans votre script:

```
(display ((lambda (x) (* x x)) 10))(newline)
```

Ça peut paraître un peu compliqué à première vue, mais tout ce qu'on vient de faire, c'est remplacer carré par (lambda (x) (* x x)). C'est très utile dans le cadre de petites fonctions spécialisées qui ne seront plus utilisées dans le reste du script, autrement ça deviendrait vite lourd de devoir définir chaque petite fonction spécifiquement.

La machine à états

La machine à états est le point clé dans la compréhension du système Fluxus. Tout ce qu'il fait, c'est qu'en appelant certaines fonctions, vous changerez le contexte courant du script, ce qui aurra un effet sur les fonctions qui suivront. Cette manière de procéder est très utile et fonctionne à peut prêt comme l'API d'OpenGl. Par exemple:

```
(define (dessin)
       (colour (vector 1 0 0))
       (draw-cube)
       (translate (vector 2 0 0)) ; on bouge un peu le « crayon »
       (colour (vector 0 1 0))
       (draw-cube))

(every-frame (dessin))
```

Dessinera un cube rouge, puis un cube vert (vous pouvez imaginer l'appel de colour comme un changement de crayon d'une couleur différente). Les états peuvent aussi être « empilés », par exemple:

```
(define (dessin)
       (colour (vector 1 0 0))
       (with-state
              (colour (vector 0 1 0))
              (draw-cube))
       (translate (vector 2 0 0))
       (draw-cube))

(every-frame (dessin))
```

Dessinera un cube vert, puis un cube rouge. with-state « isole » un état, lui donne une durée de vie, grâce à ses parenthèses. C'est pourquoi les changements à l'intérieur du with-state n'affectent que le build-cube enveloppé, et pas les autres. Il vaut mieux utiliser l'indentation pour se faire une idée plus concrète.

La scène

Les deux exemples que l'on vient de voir utilisent ce qu'on appelle le mode « à la volée ». On dispose d'une pile, le sommet étant le contexte courant, et tout est dessiné une fois par image. Fluxus propose une structure que l'on appellera la « scène » qui servira à stocker des objets ainsi que leurs états.

Un exemple:

```
(clear) ; on nettoie la scène
(colour (vector 1 0 0))
(build-cube) ; on ajoute un cube rouge à la scène
(translate (vector 2 0 0))
(colour (vector 0 1 0))
(build-cube) ; on ajoute un cube vert à la scène
```

Ce code n'as pas besoin d'être appelé par every-frame, et nous utilisons build-cube à la place de draw-cube. Les fonctions build-* créent des objets primitifs, copient leurs états courant et rajoutent ces informations dans un conteneur appelé une « cellule de scène ».

En fait les fonctions build-* retournent l'identifiant de l'objet, qui au final n'est qu'un numéro, ce qui vous permettra de référencer la cellule de scène après qu'elle soit crée. Vous pouvez spécifier des objets comme cela:

```
(define monObjet (build-cube))
```

Ce cube serra désormais rattaché à la scène jusqu'à ce qu'on le détruise:

```
(destroy monObjet)
```

with-state retourne l'état de la dernière expression qu'elle contient, donc pour créer de nouvelles primitives avec un état donné, vous pourrez faire:

```
(define mon-object (with-state
               (colour (vector 1 0 0))
               (scale (vector 0.5 0.5 0.5))
               (build-cube)))
```

Si vous voulez modifier l'état d'un objet après l'avoir chargé dans la scène, vous pouvez utiliser la fonction with-primitive pour établir un nouvel état à un objet. Cela vous permet d'animer des objets que vous aurez crée:

```
; construisons quelques cubes

(colour (vector 1 1 1))
(define obj1 (build-cube))
```

```
(define obj2 (with-state
               (translate (vector 2 0 0))
               (build-cube))

(define (animate)
       (with-primitive obj1
               (rotate (vector 0 1 0)))

       (with-primitive obj2
               (rotate (vector 0 0 1))))

(every-frame (animate))
```

Une commande très importante de Fluxus est parent qui « chaîne » un objet à un autre, pour qu'ils puissent se suivre. Vous pouvez utiliser parent pour monter une hiérarchie d'objets, par exemple:

```
(define A (build-cube))

(define B (with-state
       (parent A)
       (translate (vector 0 2 0))
       (build-cube)))

(define C (with-state
       (parent B)
       (translate (vector 0 2 0))
       (build-cube)))
```

ce qui aura pour effet de créer trois cubes, tous reliés les uns aux autres comme une chaîne. Les transformations sur A se répercuteront sur B et sur C, celles appliquées sur B affecteront C.

De même détruire A détruirait B et C.

Vous pouvez tout de même modifier la hiérarchie de vos objets en appelant ka fonction parent à l'intérieur de with-primitive. Pour retirer l'identifiant d'un objet de sa mère, il faut appeler:

```
(with-primitive monObjetFils
        (detach))
```

Ce qui « détachera » l'objet monObjetFils de sa mère, tout en restant dans la scène.

Remarque: Par défaut, quand vous créez des objets, ils sont apparentés à la cellule « souche » de la scène. La cellule souche possède l'identifiant 1. Alors une autre façon de briser la parenté d'un objet serra d'appeler (parent 1).

Remarque sur push et pop

Fluxus possède aussi quelques commandes moins propres pour arriver au même résultat. Ces commandes étaient très utilisées dans les versions antérieures à 0.14 donc ne soyez pas surpris de tomber sur elles dans la documentation ou dans d'anciens scripts.

```
(push)...(pop)
```

revient à faire:

```
(with-state ...)
```

et

```
(grab maPrimitive)...(ungrab)
```

est la même chose que

```
(with-primitive maPrimitive ...)
```

Ces fonctions sont moins sûres car on peut appeler push sans appeler pop à la fin, et leur utilisation est déconseillé. Néanmoins dans certains cas vous ne pourrez pas faire autrement. En fait with-* les utilisent, c'est pourquoi on ne les enlève pas.

Les entrées

Très vite, vous allez vouloir utiliser des données externes à Fluxus, dans le but d'interagir avec vos animations, et c'est ce qui va rendre Fluxus très intéressant.

Le son

La principale fonction de Fluxus (et surement sa meilleure utilisation) est la modélisation de sons pour du vjing. Pour cela, Fluxus doit pouvoir récupérer en temps réel des valeurs représentatives d'un son (ou d'une source de son). On configure l'entrée du son comme cela:

```
(start-audio “port-jack-à-écouter” 2048 44100)
```

Lancez une application qui nous fournira du son, ou récupérez votre entrée audio de la carte son avec Jack. Utilisez alors:

```
(gh numéro-d'un-harmonique)
```

Qui nous renverra un réel que l'on pourra utiliser comme paramètre d'animation. Vous pourrez vérifier que gh vous renvoie bien autre chose que des zéros en affichant sa valeur:

```
(every-frame (begin (display (gh 0)) (newline)))
```

Vous pourrez aussi utiliser gain pour contrôler le gain de vos entrées:

```
(gain 1)
```

N'oubliez pas que cela aurra un effet sur la grandeur de valeurs retournés par gh.

Le clavier

Il est très intéressant d'utiliser les touches du clavier dans certains cas, comme pour des jeux par exemple:

```
(key-pressed touche-clavier)
```

Retournera #t si touche-clavier est appuyée, par exemple:

```
(every-frame
    (when (key-pressed “x”)
       (colour (rndvec))
```

```
        (draw-cube)))
```

Dessinera un cube qui aurra sa couleur aléatoirement changée lorsque l'on appuiera sur la touche « x »:

```
(keys-down)
```

Retourne la liste des touches enfoncées depuis le lancement du script.

Pour les touches qui ne peuvent pas être décrites par une String, il y a aussi:

```
(key-special-pressed key-code-number)
```

Par exemple:

```
(key-special-pressed 101)
```

retournera #t quand on appuiera sur la touche HAUT du clavier. Pour connaître le nom de ces touches là, on pourra utiliser ce script:

```
(every-frame (begin (display (keys-special-down)) (newline)))
```

qui retournera la liste des touches actuellement enfoncées. Vous n'aurez qu'à appuyer sur la touche que vous voulez pour voir apparaître son code.

Remarque: les codes des touches peuvent différées selon le système d'exploitation utilisé.

Mouse (La souris)

Vous pouvez demander les coordonnées de la souris avec:

```
(mouse-x)
(mouse-y)
```

Et savoir si un bouton de la souris est enfoncé.

```
(mouse-button button-number)
```

vous retournera #t si ce bouton est pressé.

Select (choix)

Pendant que nous sommes au niveau de la souris, l'une de ses meilleurs utilisations est de choisir des primitives, avec lesquelles on peut faire :

```
(select screen-x screen-y size)
```

Ce qui rendra le morceau de l'écran autour de ses coordonnées x (screen-x) et y (screen-y); Un exemple:

```
; click on the donuts!
(clear)

(define (make-donuts n)
    (when (not (zero? n))
        (with-state
            (translate (vmul (srndvec) 5))
            (scale 0.1)
            (build-torus 1 2 12 12))
        (make-donuts (- n 1))))

(make-donuts 10)
```

```
(every-frame
    (when (mouse-button 1)
        (let ((s (select (mouse-x) (mouse-y) 2)))
            (when (not (zero? s))
                (with-primitive s
                    (colour (rndvec)))))))
```

OSC

OSC (Open Sound Control) est un protocole standard de communication entre les programmes courants et le réseau. Fluxus a la quasi-totalité des droits d'envoi et de réception des messages OSC.

Pour commencer, voici un exemple de script qui lit les messages OSC à partir d'un port et utilise la première valeur d'un message pour positionner un cube:

```
(osc-source "6543")
(every-frame
    (with-state
        (when (osc-msg "/zzz")
            (translate (vector 0 0 (osc 0))))
        (draw-cube)))
```

Et ceci est un pd patch qui peut être utilisé pour contrôler la position du cube.

```
#N canvas 618 417 286 266 10;
#X obj 58 161 sendOSC;
#X msg 73 135 connect localhost 6543;
#X msg 58 82 send /zzz \$1;
#X floatatom 58 29 5 0 0 0 - - -;
#X obj 58 54 / 100;
#X obj 73 110 loadbang;
#X connect 1 0 0 0;
#X connect 2 0 0 0;
#X connect 3 0 4 0;
#X connect 4 0 2 0;
#X connect 5 0 1 0;
```

Time (le temps)

J'ai décidé d'inclure le temps comme source de saisie, de telle sorte qu'il vous paraisse venir de l'extérieur; c'est aussi une façon de faire de l'animation plus fiable.

```
(time)
```

Cette fonction retourne le temps (en secondes) qui s'est écoulé depuis le début de l'animation. Elle retourne un nombre approché et est enregistré une fois par fenêtre.

```
(delta)
```

Celle là retourne le temps entre la fenêtre actuelle et sa précédente. (delta) est une commande très importante du point de vue qu'elle te permette de faire une animation avec un taux de fenêtrage indépendant.

Considérons le script suivant:

```
(clear)
(define my-cube (build-cube))
(every-frame
	(with-primitive my-cube
		(rotate (vector 5 0 0))))
```

Ce script fait tourner le paramètre my-cube d'un degré à chaque fenêtre. Le problème est que cette rotation est rapide lorsque le taux de fenêtrage augmente et est lente quand le taux diminue. Si tu exécute ce script sur un autre ordinateur, il va marcher avec un taux différent. Ceci fût un problème lorsqu'on exécutais des jeux sur les anciens ordinateurs -Ils devenaient injouables comme ils vont aussi vite sur les nouveaux matériels !

La solution était d'utiliser la commande (delta):

```
(clear)
(define my-cube (build-cube))
(every-frame
	(with-primitive my-cube
		(rotate (vector (* 45 (delta)) 0 0))))
```

Le cube tournera à la même vitesse partout avec un angle de 45° par seconde.

La commande normale (time) n'est pas trop utile ici; Puisqu'elle retourne un nombre toujours croissant, ce qui est insuffisant pour une animation. Néanmoins, pour l'utiliser on la passe en paramètre à la fonction sinus pour avoir une fonction sinusoïdale:

```
(clear)
(define my-cube (build-cube))
(every-frame
	(with-primitive my-cube
		(rotate (vector (* 45 (sin (time))) 0 0))))
```

Ce script également a un taux de fenêtrage indépendant et donne une valeur comprise entre -1 et 1. Vous pouvez aussi enregistrer du début du script aux événements a venir:

```
(clear)
(define my-cube (build-cube))
(define start-time (time)) ; enregistre le temps maintenant
(every-frame
	(when (> (- (time) start-time) 5) ; 5 secondes après le début du script...
		(with-primitive my-cube
			(rotate (vector (* 45 (delta)) 0 0)))))
```

Propriétés Matérielles

Maintenant vous pouvez créer et illuminer quelques primitives, jusqu'à pouvoir changer leur apparence, autre que simplement modifier leur couleur.

Les paramètres de surface peuvent être traités exactement comme le reste des définitions locales; Dans ce cas ils peuvent être modifiés en utilisant les piles locales, en construisant des fonctions ou les modifier ultérieurement en utilisant la fonction (with-primitive). Il existe également d'autres modificateurs

tels que:

Les modificateurs de dessin de lignes et de points (width est en pixels):

```
(wire-colour n)
(line-width n)
(point-width n)
```

Les modificateurs d'éclairage:

```
(specular v)
(ambient v)
(emissive v)
(shinyness n)
```

Opacité:

```
(opacity n)
(blend-mode src dest)
```

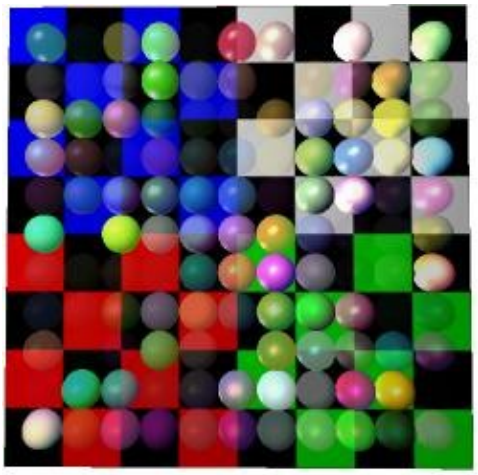

Illustration 2: des modificateurs sur quelques sphères au dessus d'un plan texturé

La commande opacity permet a l'animation d'être à moitié transparent (à travers la texture avec un composant alpha). Mais cette alpha transparence apporte quelques problèmes, du fait qu'elle expose que les primitives soient tirés dans un certain ordre, ce qui changera le résultat final où l'opacité est concerné. Ceci entraîne des scintillements à l'écran et le manque de morceaux d'objets derrière les surfaces transparentes, et c'est un problème commun de l'interprétation en temps réel.

La solution est généralement l'utilisation de la primitive hint, soit en faisant (hint-ignore-depth) qui permettra à la primitive de rendre à chaque fenêtre la même illustration sans tenir compte des autres obstacles, ou en faisant (hint-depth-sort) qui permettra à la primitive de classer la fenêtre actuelle avant de rendre la prochaine; ainsi elles apparaitront de la plus ancienne à la plus récente. La profondeur est prise en compte à partir de l'origine de la primitive. Vous pouvez la voir avec (hint-origin).

Texturing

Texturing est une partie des propriétés matérielles locales, mais c'est un noble thème complexe avec ses propres droits.

Chargement de textures

Avoir une texture chargée et appliquée à une primitive est très simple:

```
(with-state
        (texture (load-texture “test.png”))
        (build-cube))
```

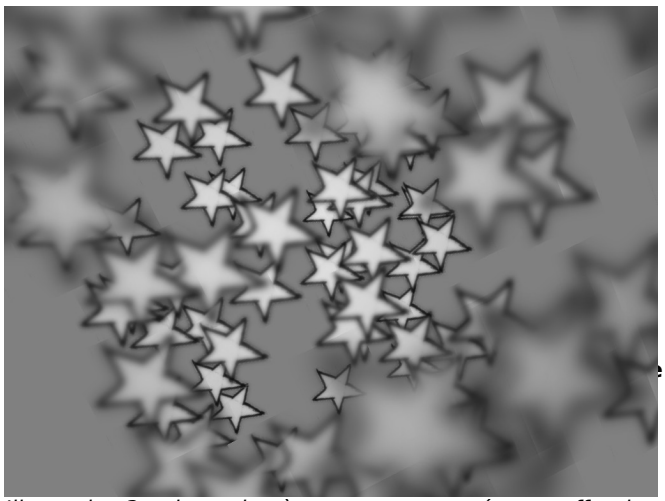

Illustration 3: mipmaping à outrance pour créer cet effet de profondeur

ce script appliquera la texture standard test de Fluxus à un cube. Fluxus lit les fichiers png files pour les utiliser comme ses textures et elles peuvent être avec ou sans un composant alpha.

Le chargement de texture est une mémoire cachée. Ce qui signifie que tu peux appeler (load-texture) avec les mêmes textures autant de fois que tu désires, Fluxus les chargera seulement que la première fois du disque. Ceci es généré jusqu'à ce que vous changiez la texture vous-même, c'est-à-dire si vous modifiez le fichier à partir de Gimp et exécutez à nouveau le script, la texture ne changera pas dans ta scène; pour contourner ça, utilise:

```
(clear-texture-cache)
```

Ce qui forcera un rechargement de la texture; il est souvent nécessaire de le mettre au début du script.

Coordonnées de la texture

Dans le but d'appliquer une texture à la surface d'une primitive, les coordonnées d'une texture sont indispensables, car elles disent au compileur quelle partie de la primitive doit être couverte et par quelles parties de la texture . Les coordonnées de la texture sont de deux valeurs comprises entre 0 et 1; (par convention, on les appelle les coordonnées "s" et "t"). Lorsque ces coordonnées sont en dehors de cet intervalle, vous pouvez les changer avec la commande (texture-params); mais par défaut, la texture est répétée.

La forme du polygone et des NURBS vous fournissent toutes les coordonnées par défaut. Ces coordonnées sont une partie du pdata pour la primitive, qui sont couvert de plus de détails plus tard. La coordonnée du tableau du pdata de la texture est appelée "t", un simple exemple qui amplifiera la texture par un facteur de deux:

```
(pdata-map!
       (lambda (t)
              (vmul t 0.5)) ; diminue la taille des coordonnées de la texture dans la texture
       "t")
```

Paramètres de la texture

Il y a assez de paramètres extra que vous pouvez utiliser pour la manière dont la texture est définie.

```
(texture-params texture-unit-number param-list)
```

Nous couvrirons le numéro d'unité de texture en plusieurs texturing ci-dessous, mais la liste des paramètres (params) peut ressembler à ça:

```
(texture-params 0 '(min nearest mag nearest)) ; super aliased & blocky texture :)
```

La meilleure approche est d'avoir un jeu et voir ce que ceux-ci font pour toi, mais voici les paramètres en totalité:

tex-env : un parmi [modulate decal blend replace]

change la manière dont une texture est appliquée à la couleur d'une surface existante.

min : [nearest linear nearest-mipmap-nearest linear-mipmap-nearest linear-mipmap-linear]

mag : [nearest linear]

Illustration 4: Une texture avec wrap-s et wrap-t

Ceux-ci modifient le type de filtres qui traitent la miniature quand les pixels de la texture (appelés texels) sont plus petits que les pixels de l'écran. Cela manque à ses engagements à linear-mipmap-linear. Aussi la grandeur, quand les texels sont plus larges que les pixels de l'écran. Cela manque à ses engagements à linear qui mélangent les couleurs des texels. Les modifier à nearest signifie que tu peux proprement voir les pixels agrandissant la texture quand c'est grand dans l'espace d'écran.

wrap-s : [clamp repeat]

wrap-t : [clamp repeat]

wrap-r : [clamp repeat] (pour afficher le cube)

Quoi faire quand les coordonnées de la texture sont en dehors de l'intervalle 0-1? régler à repeat par défaut, utiliser l'attache "smears" pour ajuster les pixels du bord de la texture.

border-colour : (vecteur de longueur 4)

Si la texture est faite pour avoir une bordure (see load-texture), ceci est la couleur qu'il devrait avoir.

priority : 0 -> 1

Je crois que ce paramètre contrôle comment probablement une texture doit être enlevée de la carte graphique quand la mémoire est basse.

env-colour : (vecteur de longueur 4)

La couleur avec laquelle on mélange la texture.

min-lod : nombre approché (for mipmap blending – par défaut c'est -1000)

max-lod : nombre approché (for mipmap blending – par défaut c'est 1000)

Ceux-ci mettent comment le mipmaping se passe.

Ce script est le code qui a produit l'image d'attache ci-dessus:

```
(clear)
(texture-params 0 '(wrap-s clamp wrap-t clamp))
(texture (load-texture "refmap.png"))
(with-primitive (build-cube)
```

```
(pdata-map!
    (lambda (t)
        (vadd (vector -0.5 -0.5 0) (vmul t 2)))
"t"))
```

Plusieurs texturing

Fluxus vous permet d'appliquer plus d'une texture à une primitive en même temps. Vous pouvez combiner les textures ensembles, les multiplier , les ajouter ou faire un mélange alpha d'eux différemment. C'est utilisé dans les jeux surtout comme une façon d'utiliser la mémoire de texture plus efficacement, en séparant des cartes claires des cartes chromatiques diffuses, ou en appliquant des cartes de détail qui se répètent excessivement des cartes de couleur de résolution plus basses.

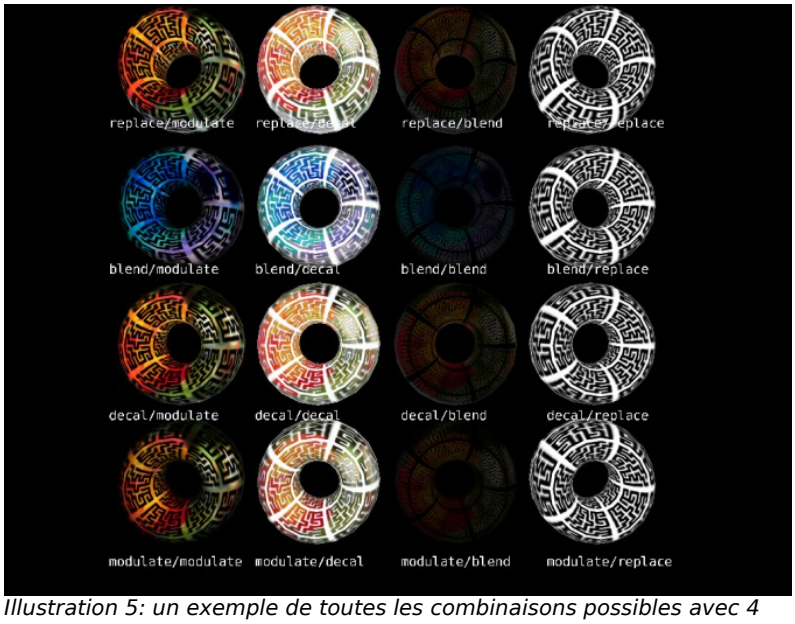

Illustration 5: un exemple de toutes les combinaisons possibles avec 4 textures

Vous avez 8 fentes pour installer des textures et les mettre ensemble :

```
(multitexture texture-unit-number texture)
```

L'unité de texture par défaut est 0, donc:

```
(multitexture 0 (load-texture "test.png"))
```

C'est exactement le même que:

```
(texture (load-texture "test.png"))
```

Ce script est un simple exemple de multitexturing:

```
(clear)
(with-primitive (build-torus 1 2 20 20)
    (multitexture 0 (load-texture "test.png"))
    (multitexture 1 (load-texture "refmap.png")))
```

Par défaut, toutes les textures partagent le pdata "t" comme leurs coordonnées, mais chaque texture cherche aussi de préférence ses propres coordonnées pour les utiliser. Ils sont nommés "t1", "t2", "t3" jusqu'à "t7".

```
(clear)
(with-primitive (build-torus 1 2 20 20)
    (multitexture 0 (load-texture "test.png"))
    (multitexture 1 (load-texture "refmap.png")))
    (pdata-copy "t" "t1") ; fait une copie des coordonnées de la texture existante
```

```
(pdata-map!
    (lambda (t1)
        (vmul t1 2)) ; rend la texture refmap.png plus petite que test.png
    "t1"))
```

C'est là où multitexturing entre vraiment dans ses propriétés, car vous pouvez déplacer et déformer individuellement les textures sur la surface d'une primitive. Une utilisation pour cela applique les textures comme des autocollants sur le bas des arrières plans des textures, une autre est d'utiliser un alpha séparé qui découpe la texture d'une de ses couleurs, et avoir la couleur de la texture nagé tant que le découpage est fait.

Mipmapping

Il y a plus pour faire le chargement de textures que ce que nous avons expliqué très loin. Ça peut aussi prendre des listes de paramètres pour changer la manière dont la texture est générée. Par défaut, quand la texture est chargée, un jeu de mipmaps est produit à cet effet. Les mipmaps sont les petites versions de la texture à utiliser quand l'objet est loin de la caméra et est pré calculé dans le but d'accélérer l'interprétation.

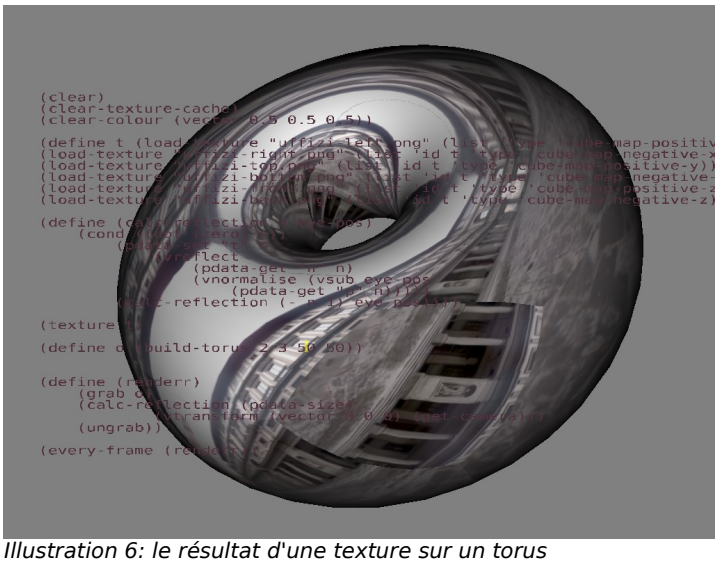

Illustration 6: le résultat d'une texture sur un torus

Une des choses communes que vous pouvez vouloir faire est d'éteindre le mipmapping, comme le blurriness peut être un problème quelquefois.

```
(texture (load-texture "refmap.png"
       '(generate-mipmaps 0 mip-level 0))) ; don't make mipmaps, and send to the top mip level
(texture-params 0 '(min linear)) ; turn off mipmap blending
```

Une autre sorte de texture est de fournir vos propres niveaux de mipmap:

```
; régler une texture mipmappée avec tes propres images
; you need as many levels as it takes you to get to 1X1 pixels from your
; level 0 texture size
(define t2 (load-texture "m0.png" (list 'generate-mipmaps 0 'mip-level 0)))
(load-texture "m1.png" (list 'id t2 'generate-mipmaps 0 'mip-level 1))
(load-texture "m2.png" (list 'id t2 'generate-mipmaps 0 'mip-level 2))
(load-texture "m3.png" (list 'id t2 'generate-mipmaps 0 'mip-level 3))
```

Ceci montre comment charger plusieurs images dans une seule texture, et comment la profondeur des effets de champs ont été accomplis dans l'image au-dessus. Vous pouvez aussi utiliser un GLSL shader pour choisir des niveaux de mip selon les autres sources.

Cubemapping (affichage de cube)

Le Cubemapping est monté d'une façon semblable au mipmapping, mais est utilisé pour des raisons un peu différentes. Cubemapping est une approximation d'un effet où une surface reflète un environnement autour de lui, qui dans ce cas est décrit par six textures représentant les six côtés d'un cube autour de l'objet.

```
(define t (load-texture "cube-left.png" (list 'type 'cube-map-positive-x)))
(load-texture "cube-right.png" (list 'id t 'type 'cube-map-negative-x))
(load-texture "cube-top.png" (list 'id t 'type 'cube-map-positive-y))
(load-texture "cube-bottom.png" (list 'id t 'type 'cube-map-negative-y))
(load-texture "cube-front.png" (list 'id t 'type 'cube-map-positive-z))
(load-texture "cube-back.png" (list 'id t 'type 'cube-map-negative-z))
(texture t)
```

Éclairage

Nous sommes arrivés très loin avec l'ignorance du thème important d'éclairage, et ce parce que Fluxus apporte par défaut de la lumière, qui est blanc pure et est attachée à la caméra, donc ça te permet toujours de voir clairement les primitives. Pourtant, c'est un réglage très ennuyeux, et si tu configure ton propre éclairage, tu peux utiliser des lumières avec de très intéressantes et créatives manières, donc c'est bien d'expérimenter avec ce qui est possible.

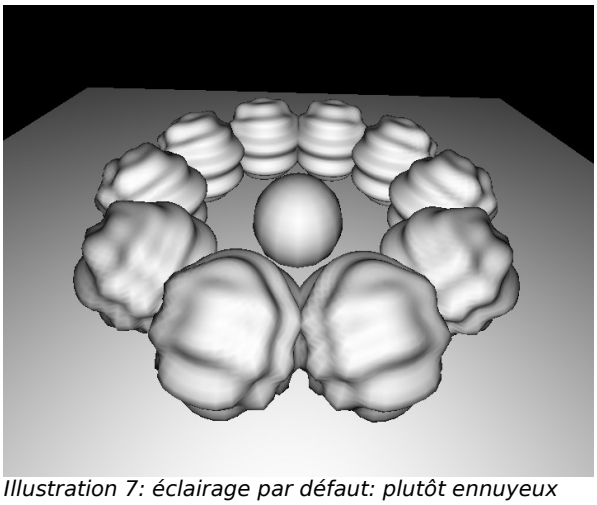

Illustration 7: éclairage par défaut: plutôt ennuyeux

L'approche standard dans l'infographie vient de la photographie, et est renvoyé vers l'éclairage des 3 points. Les trois lumières dont vous avez besoin sont:

- Key – La lumière key est la lumière principale pour l'illumination du sujet. Elle est mise du côté du sujet comme la caméra, mais éteinte sur un côté.
- Fill – La lumière fill enlève les durs ombres de la lumière fill et fournit quelques éclairages diffus. Elle est mise du côté de la caméra, mais à l'opposé de la lumière key.
- La lumière Rim ou Back, pour séparer le sujet de l'arrière plan. Elle est mise derrière le sujet, sur un côté pour illuminer les bords du sujet.

Voici un exemple de script de Fluxus pour monter un tel éclairage:

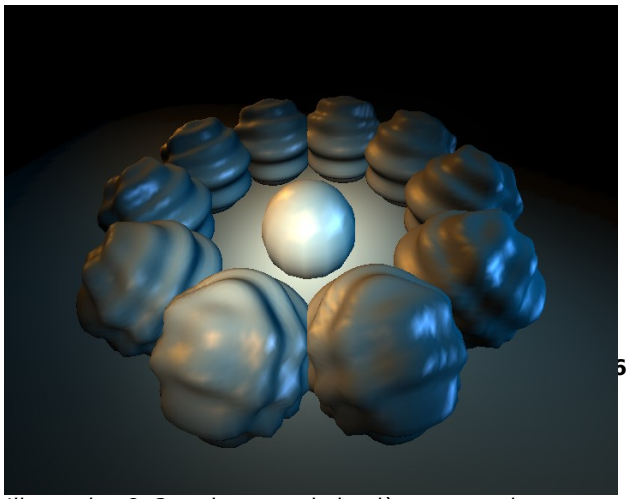

Illustration 8: 3 projecteurs de lumière pour attirer l'attention sur la sphère du centre

```
; la lumière 0 est la lumière par défaut, monté à un niveau bas
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))

; faire une grande grosse lumière key
(define key (make-light 'spot 'free))
(light-position key (vector 5 5 0))
(light-diffuse key (vector 1 0.95 0.8))
(light-specular key (vector 0.6 0.3 0.1))
(light-spot-angle key 22)
(light-spot-exponent key 100)
(light-direction key (vector -1 -1 0))

; faire une lumière fill
(define fill (make-light 'spot 'free))
(light-position fill (vector -7 7 12))
(light-diffuse fill (vector 0.5 0.3
0.1))
(light-specular fill (vector 0.5 0.3
0.05))
(light-spot-angle fill 12)
(light-spot-exponent fill 100)
(light-direction fill (vector 0.6 -0.6
-1))

; faire une lumière rim
(define rim (make-light 'spot 'free))
(light-position rim (vector 0.5 7 -12))
(light-diffuse rim (vector 0 0.3 0.5))
(light-specular rim (vector 0.4 0.6 1))
(light-spot-angle rim 12)
(light-spot-exponent rim 100)
(light-direction rim (vector 0 -0.6 1))
```

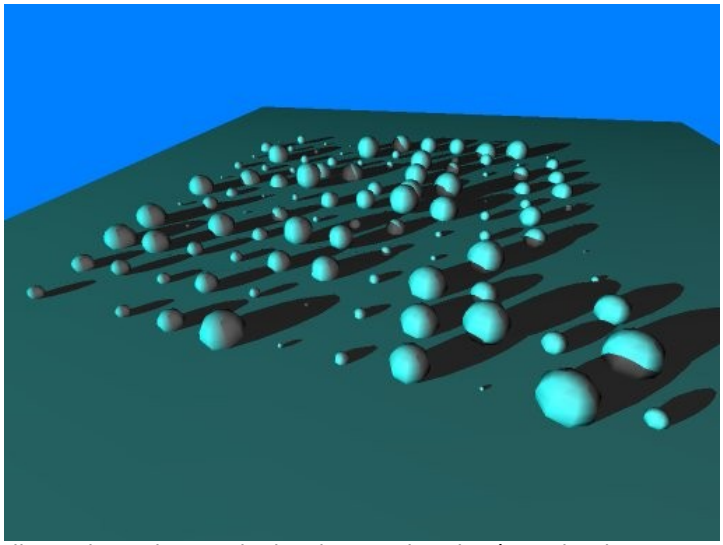

Illustration 9: l'exemple d'ombrages dans le répertoire des exemples

L'ombre

Le manque d'ombres est un grand problème avec l'infographie. Il sont complexes et prennent du temps pour produire, mais ajoutent beaucoup à la profondeur et à la forme.

Pourtant, il est tout à fait facile de les ajouter à une scène Fluxus:

```
(clear)
(light-diffuse 0 (vector 0 0 0)) ; éteint la lumière principale
(define l (make-light 'point 'free)) ; génère une nouvelle lumière
(light-position l (vector 10 50 50)) ; déplace la
(light-diffuse l (vector 1 1 1))
(shadow-light l) ; enregistre la comme la lumière des ombres

(with-state
    (translate (vector 0 2 0))
    (hint-cast-shadow) ; demande un cast pour l'ombre
    (build-torus 0.1 1 10 10))
```

```
(with-state ; génère quelque chose pour que l'ombre tombe par dessus .
    (rotate (vector 90 0 0))
    (scale 10)
    (build-plane))
```

Les problèmes liés aux ombres

Il y a quelques ennuis avec l'utilisation de l'ombre dans Fluxus. Primo, ils exigent quelques données à être produites pour la primitive qui vont caster l'ombre. Cela peut prendre du temps pour calculer les mailles complexes; bien qu'il soit seulement calculé la première fois qu'une primitive est rendue (toujours la première fenêtre).

Une édition plus problématique est l'effet de l'utilisation de la méthode shadowing de Fluxus, qui est rapide et exact, mais ne travaille pas si la caméra elle-même est à l'intérieur volume d'ombres. Vous verrez des ombres disparaître ou s'inverser. Elle peut prendre une scène prudente montée pour éviter cet incident.

Des alternatives communes aux ombres calculées (ou allumant en général) doivent les peindre à l'intérieur des textures, ceci est une meilleure approche si la lumière et les objets doivent rester fixés.

Générer des allusions

Nous avons déjà vu quelques générateurs d'allusions, mais pour les expliquer correctement, ils sont de diverses options qui peuvent changer la façon dont une primitive est rendue. Ces options sont appelées hints, comme pour quelques types de primitives, ils ne peuvent faire d'application, ou peuvent faire de différentes choses. Ils sont utiles pour le fait de déboguer et quelquefois juste pour l'amusement.

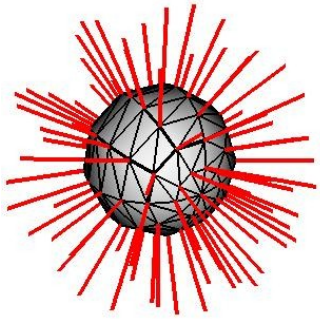

Illustration 10: Une poly-sphère avec ses normales et ses arrêtes

```
(hint-none)
```

Cette fonction hint est très importante; elle efface toutes le autres hints. Par défaut seulement une est mise hint-solid. Si tu veux la désactiver, et juste retourner un cube dans une fenêtre métallique (par exemple) tu dois exécuter ce script:

```
(hint-none)
(hint-wire)
(build-cube)
```

Pour voir la liste complète des fonctions hints, réfère toi à la section de référence de la fonction. Vous pouvez faire de très jolies choses, par exemple:

```
(clear)
(light-diffuse 0 (vector 0 0 0))
```

```
(define l (make-light 'point 'free))
(light-position l (vector 10 50 50))
(light-diffuse l (vector 1 1 1))
(shadow-light l)

(with-state
    (hint-none)
    (hint-wire)
    (hint-normal)
    (translate (vector 0 2 0))
    (hint-cast-shadow)
    (build-torus 0.1 1 10 10))

(with-state
    (rotate (vector 90 0 0))
    (scale 10)
    (build-plane))
```

Tire le torus de l'exemple shadowing mais est rendue dans une fenêtre métallique, l'affichage est normale bien que le casting de l'ombre est en cours. C'est devenu une sorte de modèle de trouver des moyens créatifs pour rendre des primitives dans Fluxus!

À propos des Primitives

Les primitives sont des objets que vous pouvez retourner. Il n'y a vraiment pas assez d'autres dans une scène de Fluxus, à part les lumières, une caméra et peu de primitives.

La primitive state

La manière normale de créer une primitive est de générer quelques state que la primitive utilisera, ensuite appellera sa fonction build puis utilisera la fonction (with-primitive) pour modifier plus tard la primitive state.

```
(define myobj (with-state
               (colour (vector 0 1 0))
               (build-cube))) ; dessine un cube de couleur verte

(with-primitive myobj
       (colour (vector 1 0 0))) ; change sa couleur en rouge
```

Donc les primitives contiennent un état qui décrit des choses tels que la couleur, la texture et qui transforme les informations. Cet état opère sur la primitive en entier, une couleur pour la primitive entière, une texture, une paie de shader et une transformation. Pour avoir une petite profondeur et faire plus, nous avons besoin d'introduire une donnée de primitive.

Les tableaux de données de primitive [aka. Pdata]

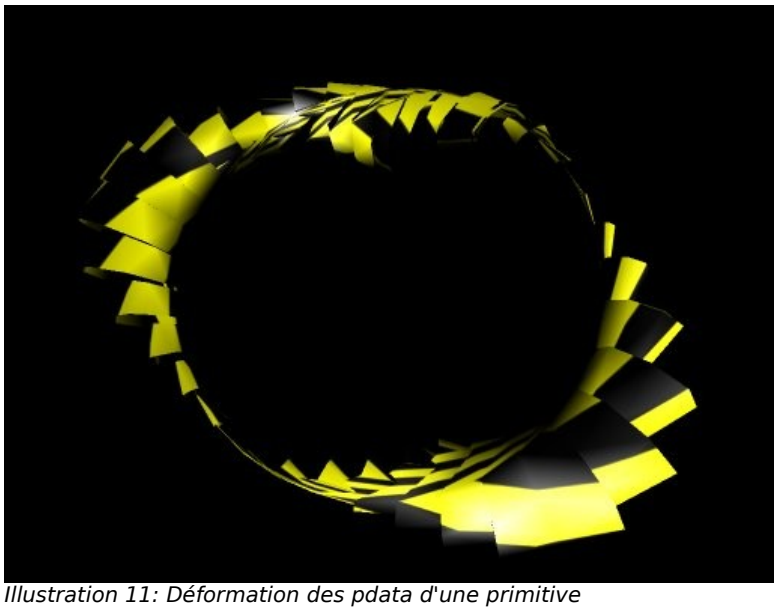

Illustration 11: Déformation des pdata d'une primitive

Un tableau de données de primitive est un tableau d'information de taille fixée qui est contenue dans la primitive. Chaque tableau a un nom, donc vous pouvez vous référer à ça, et une primitive doit contenir assez de tableaux de données de primitive différentes (qui ont toutes la même taille). Ces tableaux de données de primitive sont typés et peuvent contenir des nombres approchés, des vecteurs, des couleurs ou des matrices. Vous pouvez faire vos propres leaux de données de primitive, avec des leurs noms de votre choix ou les copié dans une commande.

Quelques tableaux sont crées lorsque la fonction build est appelée. Ceci produit automatiquement des données de primitive qui sont données avec des noms de caractères simples. Parfois ça crée automatiquement des résultats de données de primitive dans une primitive que vous pouvez utiliser par la suite (dans des commandes tel que build-cube) mais quelques primitives sont seulement utilisables si les données sont réglées et contrôlées par vous.

Dans les polygones, il y a un élément des données de primitive par vertex et un tableau séparé pour les positions, couleurs et coordonnées de textures vertex.

Donc, par exemple (build-sphere) crée un objet polygonal avec une distribution sphérique de données de points vertex , de surfaces normales à chaque vertex et coordonnées de textures; donc tu peux envelopper une texture autour de la primitive. Cette donnée (primitive data, or pdata pour raccourcir) peut être lue et écrite à l'intérieur d'une fonction (with-primitive) correspondant à l'objet courant.

```
(pdata-set! Name vertnumber vector)
```

Affecte la donnée à l'objet courant dans le vecteur utilisé

```
(pdata-ref name vertnumber)
```

Retourne le vecteur à partir des données de primitives sur l'objet courant

```
(pdata-size)
```

Retourne la taille du pdata sur l'objet courant (le nombre de verts)

Le nom définit la donnée à laquelle nous voulons accéder, par exemple “p” contient les positions vertex:

```
(pdata-set! “p” 0 (vector 0 0 0))
```

Affecte le premier point dans la primitive à l'origine (pas tout le temps)

```
(pdata-set! “p” 0 (vadd (pdata-ref “p” 0) (vector 1 0 0)))
```

Pareil que le précédent, mais l'affecte à la position originale + 1 suivant l'axe des x; le fait de compenser la position est plus utile comme il constitue une déformation du point original. (Voir le fait de Déformer, pour plus d'informations sur les déformations)

Mapping, Folding (affichage, pliage)

Les fonctions pdata-set! et pdata-ref sont utiles, mais il y a une façon plus puissante de déformer les primitives. Les fonctions Map et fold ramènent aux fonctions scheme sur les listes, c'est probablement une bonne idée de jouer avec eux pour bien comprendre ce qu'ils font.

```
(pdata-map! Procedure read/write-pdata-name read-pdata-name ...)
```

Faire un Map sur les tableaux de données de primitive semble être pour chaque pdata élément, écrire le résultat de la procédure dans le premier nom de tableau.

Un exemple de l'utilisation de pdata-map sur une primitive:

```
(define p (build-sphere 10 10))

(with-primitive p
(pdata-map!
       (lambda (n)
              (vmul n -1))
       "n"))
```

Cette méthode est plus concrète et il y a moins d'erreurs que les fonctions précédentes du fait de monter la boucle vous-même.

```
(pdata-index-map! Procedure read/write-pdata-name read-pdata-name ...)
```

Pareil que pdata-map! Mais réserve le numéro index du pdata courant à la procédure comme premier argument.

```
(pdata-fold procedure start-value read-pdata-name read-pdata-name ...)
```

Cet exemple calcule le centre de la primitive, en faisant la moyenne de toutes les positions vertex:

```
(define my-torus (build-torus 1 2 10 10))
(define torus-centre
       (with-primitive my-torus
           (vdiv (pdata-fold vadd (vector 0 0 0) “p”) (pdata-size)))))
```

```
(pdata-index-fold procedure start-value read-pdata-name read-pdata-name ...)
```

Pareil que pdata-fold mais réserve aussi le nombre index du pdata courant à la procédure comme premier argument.

En instance

Quelquefois, retenir les modes primitives peut être difficile à manier. Par exemple, si vous avez milles objets identiques, ou si vous faites les choses avec les appels récursifs, lorsque vous appelez la primitive dans plusieurs états, sauvegardez les traces de tous les objets qui seraient ennuyeux le plus.

C'est là que la méthode instance est demandée; tout ce que vous appelez est:

```
(draw-instance myobj)
```

Ce qui dessinera chaque objet dans l'état courant (mode immédiat). Un exemple:

```
(define myobj (build-nurbs-sphere 8 10)) ; fait une sphère

(define (render-spheres n)
       (cond ((not (zero? n))
              (with-state
              (translate (vector n 0 0)) ; bouge en x
              (draw-instance myobj))     ; garder une copie
              (render-spheres (- n 1)))))    ; appel recursif

(every-frame (render-spheres 10)) ; draw 10 copies
```

Construits de primitives dans le mode immédiat

Pour faciliter la vie, au lieu de faire recours aux méthodes d'instance, il y a quelques construits de primitives qui peuvent rendus à chaque moment, sans être construits:

```
(draw-cube)
(draw-sphere)
(draw-plane)
(draw-cylinder)
```

Par exemple:

```
(define (render-spheres n)
       (cond ((not (zero? n))
              (with-state
                     (translate (vector n 0 0)) ; move in x
                     (draw-sphere))            ; render a new sphere
              (render-spheres (- n 1)))))    ; recurse!

(every-frame (render-spheres 10)) ; draw 10 copies
```

Ces construits de primitives sont très restreints dans le cas où vous ne pouvez pas les éditer ou changer leur résolution etc, mais ils sont utiles pour exécuter les scripts rapidement avec des formes simples.

Types de primitives

Les primitives sont les choses les plus intéressantes dans Fluxus, comme elles représentent les objets qui sont rendus et illuminant la scène.

Illustration 12: Un poly-cube texturé avec test.png

Les primitives Polygones

Les primitives polygones sont les plus versatiles des primitives, du point de vue que Fluxus est vraiment désigné autour d'eux. Les autres primitives sont ajoutées dans le but d'achever les effets spécial et des rôle spécifiques. Les autres types de primitive ont souvent des commandes pour les convertir en primitive polygone, pour les processus futurs.

Ils y a beaucoup de commandes qui créent les primitives polygones::

```
(build-cube)
(build-sphere 10 10)
(build-torus 1 2 10 10)
(build-plane)
(build-seg-plane 10 10)
(build-cylinder 10 10)
(build-polygons 100 'triangles)
```

Le dernier est utilisé lorsque vous avez envie de construire vos propres formes de polygones, les autres sont utiles pour vous donner quelques formes prédéfinies. Toutes les fonctions build-* retournent un nombre que vous pourrez sauvegarder et l'utiliser pour modifier la primitive plus tard.

Les types de pdata

Le tableau de pdata pour les polygones est tel qu'il suit:

Utilisation	**Nom**	**Type de données**
Position vertex	p	Vecteur 3D
Vertex normal	n	Vecteur 3D
Coordonnées de texture vertex	t	Vecteur 3D pour u et v, la 3ème composante est négligeable
Couleurs vertex	c	Vecteur 4D rgba

Topologie du polygone et pdata

Avec des objets polygonaux, nous avons besoin de connecter les vertices définies par le pdata dans les faces que décrit une surface. La topologie de la primitive polygone définit comment ça se passe:

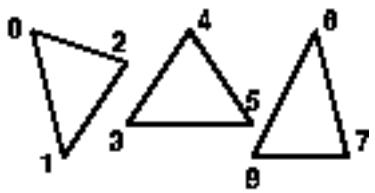

Illustration 13: topologie de liste

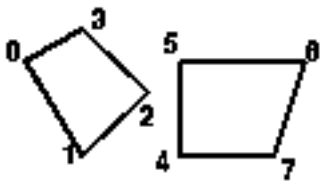

Illustration 14: topologie de liste de quadrilatères

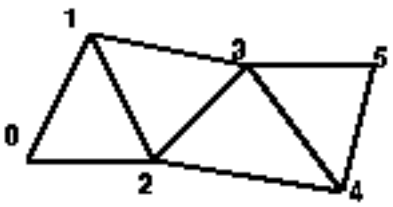

Illustration 15: topologie de bande de triangles

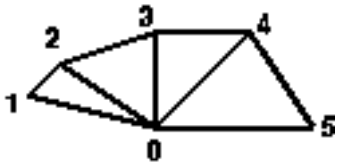

Illustration 16: topologie de triangle-fan

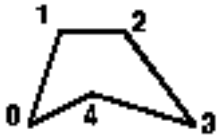

Illustration 17: topologie de polygone

Cette topologie est la même pour tous les pdata sur une primitive polygone particulière; les positions vertex, la normale, les couleurs et les coordonnées de la texture.

Bien que l'utilisation de ces topologies signifient que la primitive est optimisée à être très rapide à rendre , ça coûte assez de mémoire vu que les points sont dupliqués; Cela peut aussi causer un grand impact si vous faites assez de calcul par vertex. La plus optimale des topologies de polygone est celle de bandes de triangles car elle maximise le partage de vertices, mais vois aussi les polygones indexés pour une méthode potentiellement rapide.

Vous pouvez trouver la topologie d'une primitive polygone avec:

```
(define p (build-cube))

(with-primitive p
    (display (poly-type))(newline)) ; prints out quad-list
```

Polygones indexés

L'autre manière d'améliorer l'efficacité est de convertir les polygones en mode indexé. L'indexation signifie que les vertices dans différentes faces peuvent partager la même donnée vertex. Vous pouvez définir manuellement l'index d'un polygone avec:

```
(with-primitive myobj
       (poly-set-index list-of-indices))
```

Ou automatiquement (ce qui est recommandé) avec:

```
(with-primitive myobj
       (poly-convert-to-indexed))
```

Cette procédure compresse l'objet polygone en trouvant les les vertices dupliqués ou très proches et les collent ensemble pour former un seul vertex avec plusieurs références d'index. L'indexation a un certains nombre d'avantages, par exemple des modèles très grand prendront moins de place en mémoire; mais le plus grand avantage est que les déformations ou les autres calculs par vertex seront assez rapides.

Les problèmes liés à l'indexation automatique

Comme tous les vertex, les informations deviennent partagées pour des vertices coïncident, automatiquement les polygones indexés ne peuvent pas avoir de différentes normales, de différentes couleurs ou de différentes coordonnées de texture pour les vertices qui sont à la même position. Cela signifie qu'ils auront à être lisses et continus avec le respect d'éclairage et de

texturing. Cela peut être fixée en temps, avec un algorithme de conversion plus compliquée.

Primitives NURBS

Les NURBS sont des surfaces de pièces courbées paramétriques. Elles sont manipulées de la même manière que les primitives polygones, sauf qu'au lieu des vertices, les éléments pdata représentent les vertices de contrôle de la pièce. Changer un vertex entraînera l'objet à se déformer de façon lisse à travers sa surface.

```
(build-nurbs-sphere 10 10)
(build-nurbs-plane 10 10)
```

Illustration 18: Une sphère NURBS, avec un vertex modifié

Types de pdata

Utilisation	Nom	Type de données
Contrôleur de la position vertex	p	Vecteur 3D
Contrôleur de la normale vertex	n	Vecteur 3D
Contrôleur des coordonnées de texture vertex	t	Vecteur 3D pour u et v, la 3ème composante est négligeable

Primitives Particle(particules)

Les primitives particle utilise les pdata pour representer un point,ou un sprite face à la camera qui peut etre texturé selon les options de rendu. Cette primitive est utilise pour un nombre important d'effets comme l'eau, la fumée les nuages et les explosions.

```
(build-particles nmbre-de-particles)
```

Illustration 19: Quelques particules

Types de Pdata

Utilisation	Nom	Type de données
Position particle	p	Vecteurs 3D
Couleur Partilcle	c	Vecteurs 3D
Taille Particle	s	Vecteurs 3D pour la

		largeur et la hauteur le 3e nombre est ignoré

Astuces Geometriques

La primitive particle par defaut est un sprite, mais peut etre transformer en un point dans l'espace de travaille,avec les differentes manières qui suivent.Pour transformer en point:

```
(with-primitive myparticles
(hint-none)    ; Desactive le mode solide,qui est le mode par defaut sprite
(hint-points)) ; Active le rendu par point
```

Illustration 20: Quelques particules utilisées comme sprites en leur assignant une texture

Si vous avez aussi activer (hint-anti-alias) vous pouvez obtenir des points circulaires selon votre GPU – ils peuvent etre espacé en pixels en utilisant (point-width).

Primitives Ribbon

La Primitive Ribbon est similaire à la primitive particle par le fait q'elle est soit en rendu materiel soit des quads texturés face à la camera. Cette primitive dessine une unique ligne connectant chaque pdata connecting each pdata vertex element together. La texture est etiré le long du ruban, du debut à la fin,et en largeurtravers la ligne. La largeur peut etre defini par le sommet pour changer la forme de la ligne.

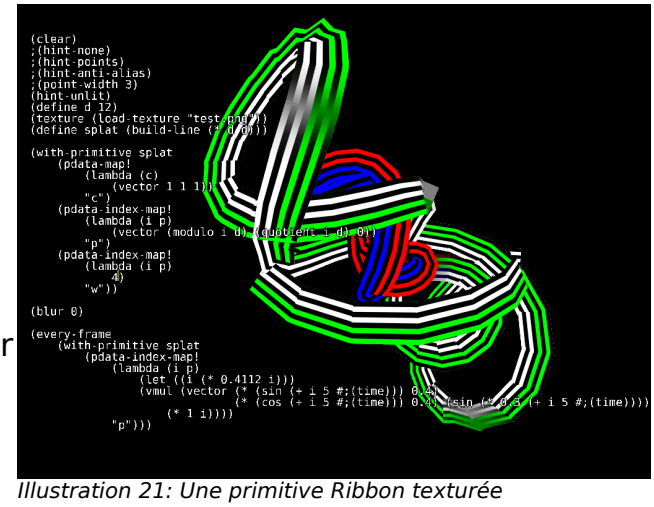

Illustration 21: Une primitive Ribbon texturée

```
(build-line num-points)
```

Types Pdata

Utilisation	Nom	Type de Données
Position du Sommet	p	Vecteurs 3D
Coleur	c	Vecteurs 3D
Largeur	w	nombre

Astuces Geometriques

Le mode par defaut de la primitive Ribbon est en rendu quads. Vous pouvez aussi choisir d'activer le rendu par ligne

```
(with-primitive myline
       (hint-none) ; desactive le mode solide, qui est le mode par defaut
```

```
        (hint-wire)) ; active le rendu par ligne
```

L'armature des lignes peut etre reduite dans l'espace en reduisant l'espace (largeur de ligne).Les lignes aussi prennent comme couleur l'information PDATA.

Primitive Text

Les primitives texte permettent de créer du texte basé sur la texture des polices.La police de caractères supposé être non porportionnelle – Il y'a un exemple de police livrée avec Fluxus.

```
(texture (load-texture “font.png”))
(build-text text-string)
```

La primitive fait une serie de quads, un pour chaque caractère, avec un ensemble de coordonnés de textures pour montrer le bon caractère de la texture. Cela fournit un moyen rapide et bon marché pour afficher du texte, et très utile pour le débogage, ou si le texte est assez faible. Vous pouvez probablement trouver aussi bien des façons creatives d'utiliser celle-ci comme un moyen de hachage d'une texture en petits carrés.

Type Primitive

Le type primitive fournit une bien meilleur qualité de rendu de caractères que la la primitive texte.Il crée à partird'une police ttf et du texte une géometrie polygonale. Vous pouvez egalement extruder le texte qui en resulte en forme 3D.

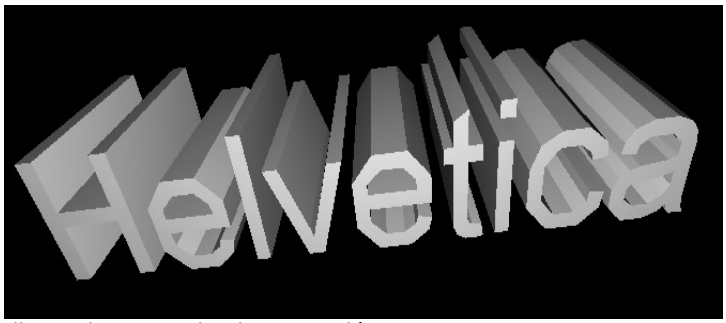

Illustration 22: Helvetica, extrudé

```
(build-type ttf-font-filename text)
(build-extruded-type ttf-font-filename text extrude depth)
```

Vous pouvez également convertir le type primitif en primitive polygone pour plus de deformation ou pour l'application de de textures avec:

```
(type->poly type-prim-id)
```

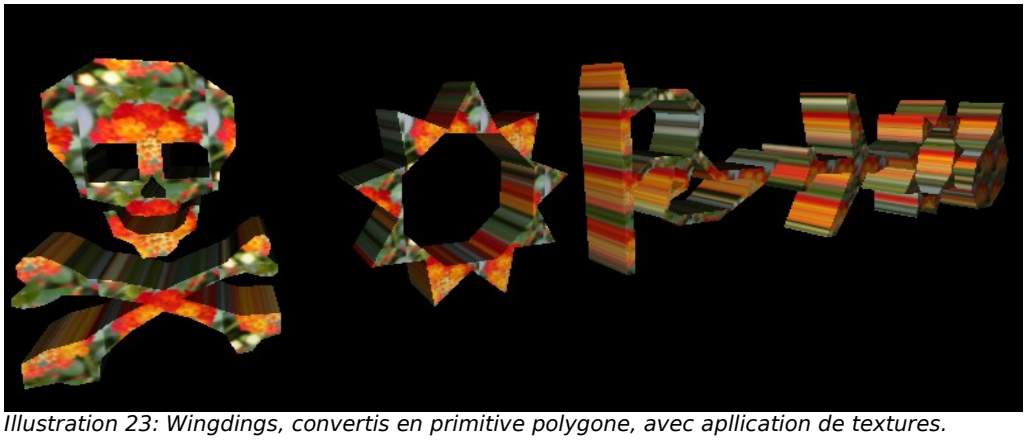

Illustration 23: Wingdings, convertis en primitive polygone, avec apllication de textures.

Primitive Locator

La primitive locator est une primitve null car elle ne rend rien du tout. Locator est utile pour diverses tâches,vous pouvez l'utiliser comme un noeud d'une scene invisble scene node pour grouper des primitives la dessus. Elle est egalement utilisés pour construire les squelettes de skin . Pour les visualiser, vous pouvez activez hint-origin (l'indication d'originr) ,qui dessine un axe représentant representant leurs transformation.

```
(hint-origin)
(build-locator)
```

Primitive Pixel

La primitive pixel est utilisé pour faire des textures procedurales, qui peuvent ensuite être appliqué à d'autres primitives.Pour cette raison, la primitive Pixel ne sera probablement pas beaucoup rendus directemetn, mais vous utiliser dans le rendu dans un aperçus de texture sur un plan

```
(pixel-primitive largeur hauteur)
```

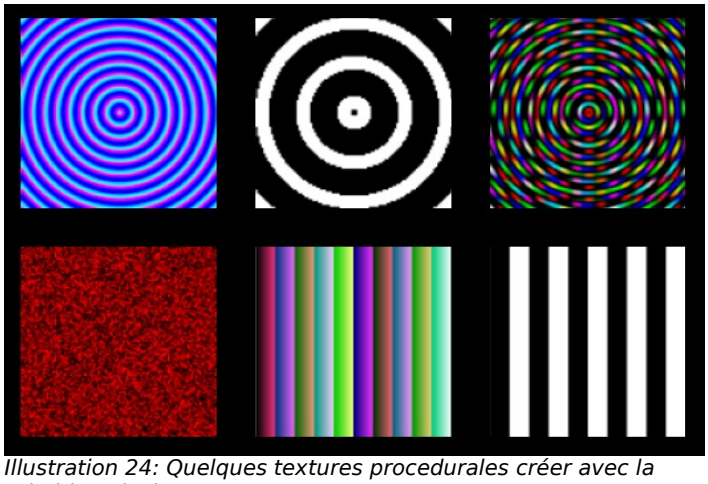

Illustration 24: Quelques textures procedurales créer avec la primitive pixel

Types Pdata

Utilisation	Nom	Type de données
Couleur Pixel	c	Vecteur 3D
Alpha de Pixel	a	nombre

Commandes Extra primitive Pixel

Le Pdata d'une primitive Pixel correspond à la valeur de pixel dans la texture, vous les ecrivez pour faire une texture procédurale de données.La primitive pixel vient avec quelques commandes:

```
(pixels-upload pixelprimitiveid-number)
```

Charge la texture de données, vous avez besoin de faire appel celle-ci lorsque vous avez fini d'ecrire la prim pix et tant qu'elle est appele.

```
(pixels->texture pixelprimitiveid-number)
```

Retourne une texture exact, vous pouvez l'utiliser exactement comme celles que vous avez charger normalement.

Voir les exemples pour quelques règles de texturing procedurale. Il est important de noter que la création de textures implque un grande quantité de temps de traitement, de sorte que vous ne voulez pas faire quelque chose sur le plan par pixel/par-frame pour les grandes textures.Les fonction des Pdata pourraient être utilisés pour nous aider à l'avenir

Ceci est un exemple simple permettant de créer un bruit de texture sur une primitive pixel.

```
(with-primitive (build-pixels 100 100)
        (pdata-map!
            (lambda (colour)
                (rndvec))
            "c")
        (pixels-upload))
```

Primitive Blobby

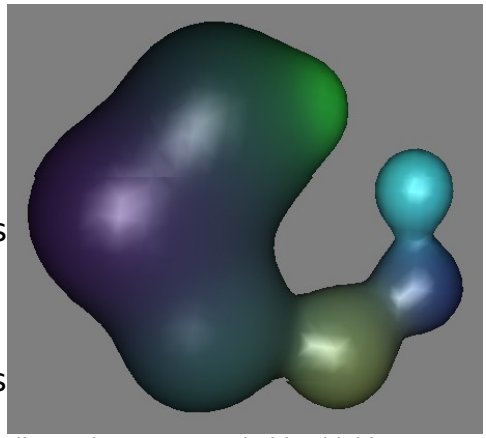

Illustration 25: Une primitive blobby

Les primitives blobby sont des representations implicite de surface d'un plus haut sont dans fluxus qui sont definies à l'aide d'influence dans l'espace. Ces influences sont additionnés ensemble, et une valeur particulière est “mailées”(en utilisant l'algorithme de marche des cube)pour former une surface lisse.Les influences peuvent êtres animés et la surface deforme pour s'adapter, ce qui donne le nom blobby à la primitive. (build-blobby) retourne un nouvel id pour une primitive blobby. Numinfluences est le nombre de "blobs".Subdivisons vous permet de contrôler la resolution de la surface de chaque dimension, tandis que boundingvec définit la zone de délimitation de la primitive dans l'espace de l'objet local.Le maillage ne sera pas calculé en dehors de cette zone.La Position des influences et les couleurs doivent être definis en utilisant pdata-set

```
(build-blobby numinfluences subdivisionsvec boundingvec)
```

Pdata Types

Utilisation	Nom	Types de données
Position	p	Vecteurs 3D
Force	s	nombre
Couleur	c	Vecteurs 3D

Conversion en Polygone

Les Blobby peuvent être lentes à calculer, si vous avez seulement besoin de maille statiques sans animation, vous pouvez les convertir en primitives polygones avec:

```
(blobby->poly blobby-id-num)
```

Deformation

Deformation dans ce chapitre diverses opérations.Il peut s'agir de changer la forme d'une primitive d'une manière pas possible par le biais d'une transformation (c'est-à-dire la flexion ,la déformation, etc) ou de modifer les coodonnées de texture ou de couleur pour obtenir un effet par sommet.La deformation de cette manière est aussi la seule façon d'obtenir des primitives particules pour faire quelque chose d'interessant

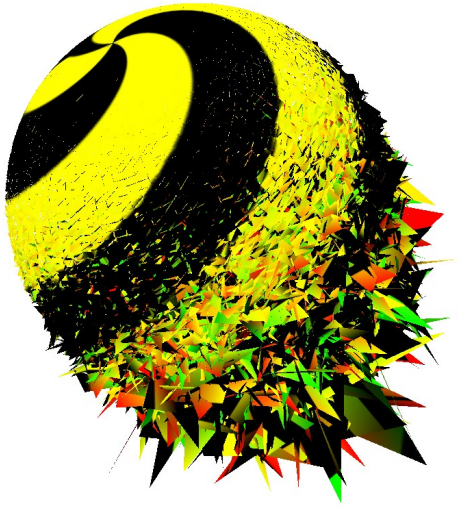

Illustration 26: Une sphère en déformation

La deformation est entierment sur les pdata, alors pour deformer un objet en entier vous devez faire comme suit:

```
(hint-unlit) (hint-wire) (line-width 4)

(define myobj (build-sphere 10 10))

(with-primitive myobj
       (pdata-map!
              (lambda (p)
                     ; add a small random vector to the original point
                     (vadd (vmul (rndvec) 0.1)) p)
             "p")))
```

Lorsque la géométrie de déformation, le déplacement des positions des sommets ne sont généralement pas assez, les normales devront être mis à jour pour que l'éclairage puissefonctionner correctement.

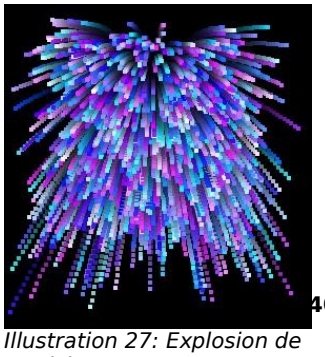

Illustration 27: Explosion de particle

```
(recalc-normals smooth)
```

Régénéront les normales pour les primitives polygones et nurbs basé sur la position des sommets. Pas particulier rapide (il est préferable de déformer les normales dans votre script, si vous le pouvez). Si le lissage est de 1, la face normale est la moyenne des faces normales coïncidant pour donner une apparence lisse.Lorsque vous travaillez sur une primitive polygone fluxus met en cache certains resultats, il sera beaucoup plus lent sur le premier calcul que les appels sur la même primitive.

User Pdata

AInsi comme les informations standard qui existe dans les primitives, fluxus vous permet également d'ajouter vos propres données par le sommet de toute primitive.User Pdata peut être écrit ou lu de la même manière que tous type de construction de tpe Pdata

```
(pdata-add name type)
```

Lorsque le nom est une chaîne avec le nom que vous voulez appeler, et le type de caractères, composé de:

f : Donnée de type float

v : Donnée de type Vecteur

c : Donnée de type couleur

m : Donnée de type matrice

```
(pdata-copy source destination)
```

Cela vous permettra de copier un tableau de pdata, ou d'ecraser un existant si il exite dejà.L' ajout de vos propres tabelaux pour le stockage des données sur les primitives, veut dire que vous pouvez l'utiliser comme un moyen rapide de lecture et d'ecriture de données même si les données ne concernent pas directement la primitive.

Un exemple est l'explosion de particle:

```
; setup the scene
(clear)
(show-fps 1)
(point-width 4)
(hint-anti-alias)

; build our particle primitive
(define particles (build-particles 1000))

; set up the particles
(with-primitive particles
       (pdata-add “vel” “v”) ; add the velocity user pdata of type vector
       (pdata-map! ; init the velocities
              (lambda (vel)
                     (vmul (vsub (vector (flxrnd) (flxrnd) (flxrnd))
```

```
                                        (vector 0.5 0.5 0.5)) 0.1))
                “vel”)
        (pdata-map! ; init the colours
                (lambda (c)
                        (vector (flxrnd) (flxrnd) 1 0))
              "c"))

(blur 0.1)

; a procedure to animate the particles
(define (animate)
        (with-primitive particles
                (pdata-map!
                        (lambda (vel)
                                (vadd vel (vector 0 -0.001 0)))
                        “vel”)
                (pdata-map! vadd “p” “vel”)))

(every-frame (animate))
```

Opération sur les Pdata

Les Operations sur les Pdata sont une optimisation pour prendre avantage de la nature de ses tableaux de stockage pour vous permettre de les traiter avec un seul appel à l'interpreter scheme.Cette deformation de primitive beaucoup plus rapide en boucle dans l'interprète scheme et il simplifie egalement votre code scheme

```
(pdata-op operation pdata operand)
```

Lorsqu'une opération est une chaîne de caractères identifiant l'opération (liste ci-dessous) et pdata est le nom de la de la cible de l'opérationis , et operand est soit une données unique (Un nombre scheme ou un vecteur (longueur 3,4 ou 16)) ou le nom d'un autre tableau pdata.

Si les fonctions (update) et (render) dans le script ci-dessus sont remplacer par ces lignes:

```
(define (update)
        ; Ajouter ce vecteur à toutes ces velocités
        (pdata-op “+” “vel” (vector 0 -0.002 0))
        ; Ajouter toutes les velocités à toutes les positions
        (pdata-op “+” “p” “vel”))

(define (render)
        (with-primitive ob
                (update)))
```

Sur ma machine, ce script fonctionne 6 fois plus rapidement que la première version.

(pdata-op) peut egalement renvoyer à votre script des informations de certaines fonctions appelé pour l'ensemble des pdata.

Opération Pdata

“+” : addition

“*” : multiplication

“sin” : ecrit le sinus d'un tableau pdata de float dans un autre.

“cos” : ecrit le cosinus d'un tableau pdata de float dans un autre.

“closest” : traite le vecteur pdata comme une position, et si c'est un seul vecteur, retourne une position proche de lui, – ou si c un float, l'utilise comme un indice dans le tableau PDATA et retourne la position la plus proche.

Pour la plupart des opérations pdata, la grande majorité des combinaisons de type d'entrée (nombre scheme, les vecteurs ou les types pdata) ne seront pas pris en charge, vous recevrez un avertissement assez mysterieux du runtime si c'est le cas.

Fonctions Pdata

Les Operations Pdata sont très utiles, mais j'avais besoin de développer l'idée en quelque chose de plus compliqué pour supporté quelque chose de plus intéressant tel que le skinning. Cette zone est en désordre, et quelque peu experimental – qu'il convient de renforcer à l'avenir.

Les fonctions Pdata (pfuncs) varie de l'usage générale au opérations specialisées que vous pouvez exécuter sur les primitives. Toutesles pfuncs partagent la meme interface pour les contrôler et les configurer.L'idée est de faire une serie d'entre eux au démarrage, puis de les exécuter sur une ou plusieurs primitives par cadre plus tard.

```
(make-pfunc pfunc-name-symbol))
```

Fait une nouvelle pfunc. Prend le symbole des noms après, e.g. (make-pfunc ‘arithmetic)

```
(pfunc-set! pfuncid-number argument-list)
```

Fixer les arguments d'une pfunc. La liste d'argument est composé des symboles et des valeurs correspondantes.

```
(pfunc-run id-number)
```

Execute une pfunc sur la primitive courante. Regardez par exemple le dépouillement pour voir comment ça fonctionne.

Type des Pfunc

Tous les types de pfunc et leurs arguments sont les suivant:

arithmetique

Pour l'application de l'arithmetique generale à tous type de tableaux Pdata

operateur string : parmis: add sub mul div

src string : nom de tableau pdata

autres string : nom de tableau pdata (optionel)

float constant : Valeur de la constante (optionel)

dst string : nom du tableau pdata

genskinweights

Génère skinweights – ajoute un pdata float appelé “s1” -> “sn” ou n est le nombre de noeuds dans le squelette – 1

skeleton-root primid-number : La racine du squellete de skinning sharpness float : Contrôle de la netteté des plis lors du skinning.

skinweights->vertcols

Un utilitaire pour visualiser skinweights pour le debogage..

Aucun Argument.

skinning

Skinner une primitive – elle se deforme pour suivre le mouvement du squelette. Les primitives sur lesquelles nous voulons l'executer doivent contenir des pdata en extra – des copies de la position du sommet appelé “pref” et de même pour les normales, si les normales sont skinne, appelé “nref”.

Skeleton-root primid-number : Primitive racine du squelette d'animation bindpose-root primid-number :Primitive racine du squelette du bindpose

Utiliser les pdata pour construire ses propres primitives:

La fonction (build_polygons) vous permet de construire une primitive vide wui peut etre utiliser pour construire d'autre type de forme procedurale qui ne sont pas supportés par fluxus en natif, ou pour charger un model de donnés du disque dur. Une fois ces primitives construites, elles peuvent etre traitées exactement de la même manière que toute autre primitive, c'est à dire des pdata peuvent etre ajouter , modifiés, et vous pouvez utiliser (recalc-normals) etc.

Camera

Sans camera vous ne seriez pas en mesure de voir quelque chose!

Elle est évidemment très importante dans l'infographie 3D, et peut être utilisé de façon très efficace, comme une camera reelle pour raconter une histoire.

Control de la Camera

Vous pouvez dejà controler la camera avec la souris mais vous aurez envie de la controler avec un script.Animer la camera de cette maniere est aussi facile, vous n'aurez qu'à verrouiller celle ci sur un objet et le deplacer comme suit:

```
(clear)
(define obj (build-cube)) ; Construit un objet sur lequel seras verouiller la camera

 (with-state ;Construisons un cube en arrierre plan pour nous permettre de savoir ce qui se passe
     (hint-wire)
     (hint-unlit)
     (texture (load-texture "test.png"))
     (colour (vector 0.5 0.5 0.5))
     (scale (vector -20 -10 -10))
     (build-cube))

 (lock-camera obj) ; Verrouillons la camera sur notre premier cube (obj)
 (camera-lag 0.1)  ; Choix de la valeur du decalage permettant de lisser le movement  du cube

 (define (animate)
     (with-primitive obj
         (identity)
         (translate (vector (fmod (time) 5) 0 0)))) ; fais un mouvement agité

 (every-frame (animate))
```

Arreter le deplacement de la camera par la souris

Meme lorsqu'elle est fixée sur un objet la camera obeït toujour la souris;

mais elle se deplace relativement à l'objet.Vous pouvez mettre fin à cette situation en fixant la transformation de la camera vous même :

```
(set-camera-transform (mtranslate (vector 0 0 -10)))
```

Cette commande prend des matrices de transformation (un vecteur de 16 nombres) qui peuventt etre generer par les commandes maths (mtranslate) , (mrotate) et (mscale) ,et les multiplie entre elles avec (mmul)

Vous avez besoin d'un seul appel de *(set-camera-transform); il donne à votre script un control complet sur la camera.et libere la souris pour faire autre chose.Pour revenir* To switch mouse à l'utilisation de la souris on utilise :

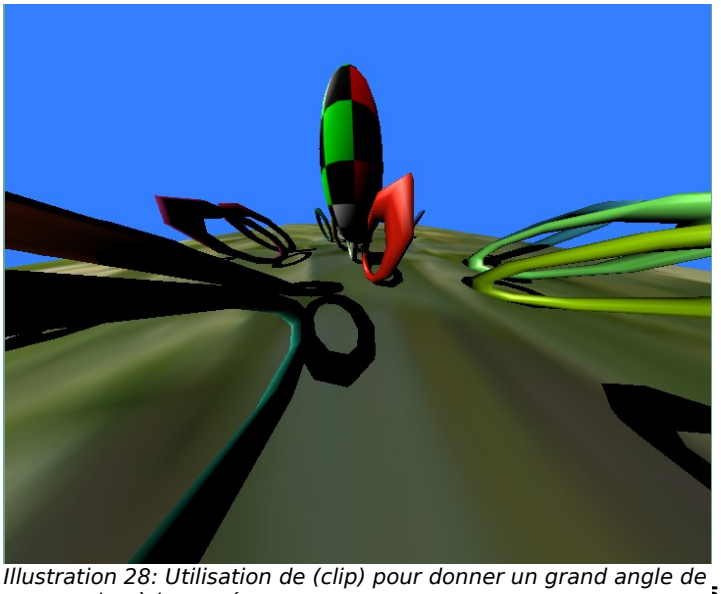

Illustration 28: Utilisation de (clip) pour donner un grand angle de perspective à la caméra

```
(reset-camera)
```

Autres propietés de la camera

Par defaut la camera est en mode perspective pour une projection en orthographique on utilise:

```
(ortho)
```

Les mouvements avant et arriere n'ont pas d'effet en mode orthographique, agrandir l'affichage on utilise:

```
(set-ortho-zoom 10)
```

Et utiliser:

```
(persp)
```

pour faire pivoter en mode perspective

L'angle de caméra peut être modifié avec la commande de confusion :

```
(clip 1 10000)
```

Qui fixe les coupures de près ou de loin de la distance de plan. La coupure de loin du plan(le second nombre) fixe ou la géometrie va commencer à etre abandonné. Le premier nombre est intéressant pour nous car ul definit la distance entre le point central de la camera au plan de coupure. Plus petit est ce nombre, plus l'anlge de camera.

Fogging(buée)

Pas strictement un réglage de camera, mais les objets en fades out lorsqu'il se deplacent hors de la camera donne l'impression d'une perspective aérienne

```
(fog (vector 0 0 1) 0.01 1 1000)
```

Le premier nombre est la couleur de la brume, suivi par la force (résistance) (à maintenir bas) puis le début et la fin du brouillard (qui n'apparait pas dans le travail sur ma carte graphique, au moins).

Fog peut être utilisé comme un moyen de cacher l'arrière plan de la coupure, ou de la rendre un peu moins choquante si les choses plutôt que de disparaître d'un coup disparaîssait en fondu- elle ajoute beaucoup de réalisme à l'exterieur des scènes, et je suis un grand fan de trouver des utilisations plus créatives pour elle.

Utilisation de multiples cameras

Cacher differentes choses dans les differents vues de camera

Bruit et l'Aléatoire

Le bruit et l'aléatoire sont utilisé dans l'animation afin d'ajouter de la "saleté" à votre travail. Ceci est très important car l'ordinateur est propre et ennuyeux par defaut

L'Aléatoire

Scheme a son propre jeu d'operation aléatoire , mais Fluxus vient avec un ensemble qui lui est propre pour vous faciliter la vie.

Operations de nombres aléatoires

Ces commandes de base retourne les nombres:

```
(rndf) ; returns un nombre entre 0 et 1
(crndf) ; (centred) retourne un nombre entre -1 and 1
(grndf) ; retourne un nombre aléatoire gaussiennce centrée sur 0 avec un ecart de 1
```

Operation de vecteurs aléatoires

Le plus souvent, lors de l'ecriture de scripts fluxus vous etes intéressés par l'obtention de vecteurs aléatoires, afin de perturber ou de générer des positions aléatoires dans l'espace, les directions pour se deplacer, ou de couleurs.

```
(rndvec) ; retourne un vecteur dont les elements sont entre 0 et 1
(crndvec) ; retourne un vecteur dont les elements sont entre -1 et 1
(srndvec) ; retourne un vecteur representé par un point dans une sphere de rayon 1
(hsrndvec) ; un vecteur representé par un point sur la surface d'une sphere de rayon 1 (sphere creuse)
(grndvec) ; Une position  Gaussienne centrée sur 0,0,0 avec pour variation 1
```

Ils sont beacoup mieux décrit par cette image:

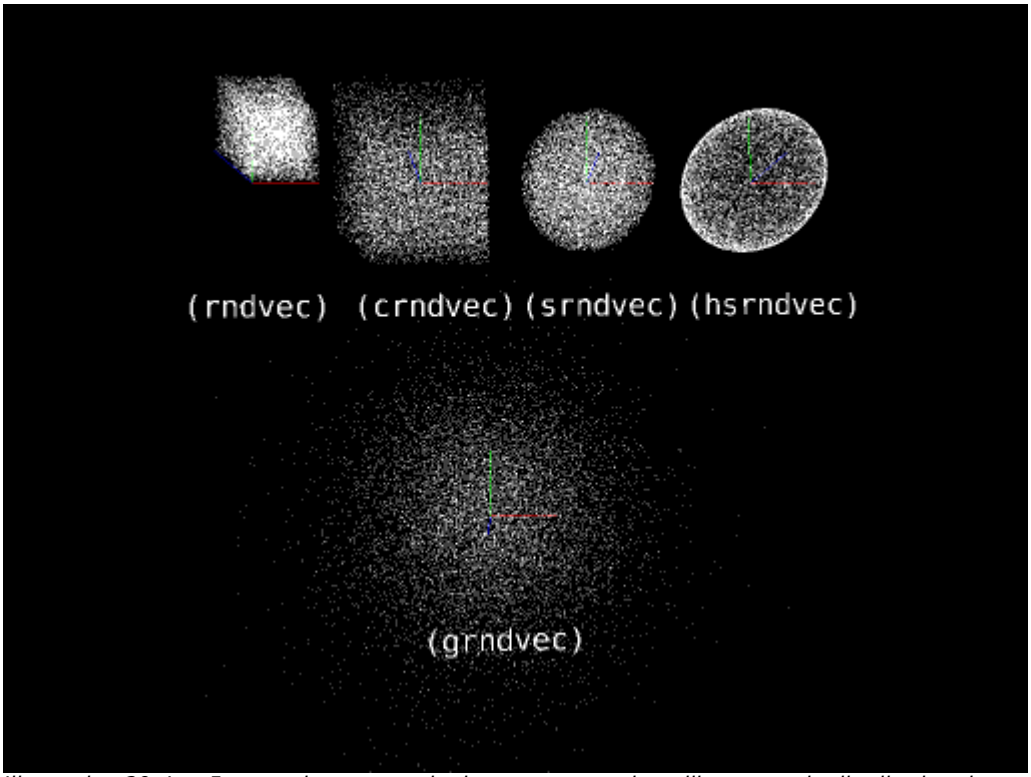

Illustration 29: Les 5 types de commande de vecteurs random illustre par la distribution de particle les utilisant.

Bruit

A faire

Inspection de Scene

Jusqu'à présent nous avons surtout cherché à decrire les objets et les déformations de fluxus, il peut aussi construire des scenes pour vous. Une autre technique efficace est de faire inspecter votre scène par fluxus et de vous donner des information sur ce qui est là.

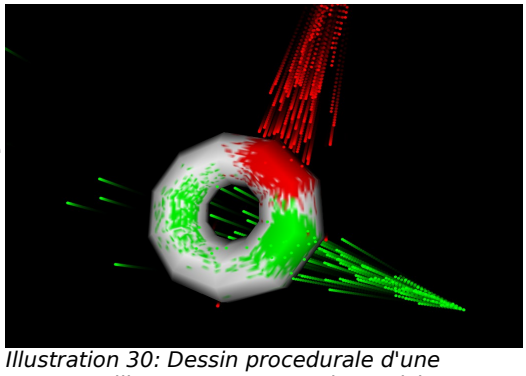

Illustration 30: Dessin procedurale d'une texture utilisant un systeme de particle et casting de rayon pour trouver des points de collision et les coordonnées de texture – et ensuite ecriture d'une primitive pixel

Scene graph inspection

Les plus simples, et peut-être les plus puissantes des commandes d'inspection sont:

```
(get-children)
(get-parent)
```

(get-children) retourne une liste de fils de la primitive, elle peut aussi vous donner une liste de fils du noeud racine de la scene si vous l'appelez de l'exterieur de (with-primitive). (get-parent) retourns le parent de la primitive courante.

Ces commandes peuvent êtres utilisées pour naviguer dans la scene et pour trouver une primitive sans que vous ayer besoin d'enregistrer manuellement les identifiants. Par exemple une primitive peut changer la couleur de son parent comme ceci :

```
(with-primitive myprim
       (with-primitive (get-parent)
              (colour (vector 1 0 0))))
```

Vous pouvez également visiter chaque primitive de la scène avec le script suivant:

```
; navigate the scene graph and print it out
(define (print-heir children)
    (for-each
        (lambda (child)
            (with-primitive child
                (printf "id: ~a parent: ~a children: ~a~n" child
(get-parent) (get-children))
                (print-heir (get-children))))
        children))
```

Detection de Collision

Une chose que vous voulez faire, est de savoir si deux objets entre en collision (particulièrement lors de

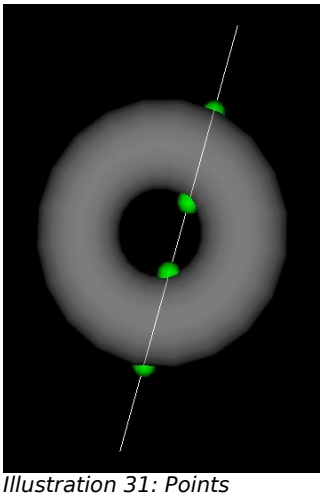

Illustration 31: Points d'intersection de la ligne avec le torus

l'ecriture de jeux). Vous pouvez savoir si l'actuelle primitive croise une autre avec cette commande:

```
(bb-intersect other-primitive box-expand)
```

Elle utilise la boîte générée automatiquement pour les primitives, et est assez rapide et assez bonne pour la plupart des detections de collisions.

Note: La boite utilisée n'est pas la meme que vous voyez avec (hint-box),qui est affecté par la transformation de la primitive. bb-intersect génère de nouveaux bounding boxes qui sont toutes alignées sur l'axe de la vitesse de comparaison.

Jet de rayons

Une autre technique extrement utile est de créer des rayons, ou des lignes dans la scène et de recuperer les informations sur l'endroit ou ils croisent les primitives. Cela peut être utilisé pour la détection détaillé de la collision ou dans des techniques plus complexes comme les techniques de raytracing.

```
(line-intersect line-start-position line-end-position)
```

Cette commande retourne une liste de valeurs pdata sur les points ou la ligne coupe une primitive. La chose ingenieuse est la valeur du point precis d'intersection – pas seulement le sommet le plus proche.

La liste retournée est conçu pour etre accessible en utilisant la commande Scheme (assoc). Une liste d'intersection ressemble à ceci:

```
(collision-point-list collision-point-list ...)
```

Ou une liste de point de collision ressemble à :

```
((p . position-vector)
 (t . texture-vector)
 (n . normal-vector)
 (c . colour-vector))
```

Les spheres vertes sur l'illustration sont positionnées sur la pdata position renvoyée par ce bout de code:

```
(with-primitive s
     (for-each
          (lambda (intersection)
              (with-state ; draw a sphere at the intersection point
                  (translate (cdr (assoc "p" intersection)))
                  (colour (vector 0 1 0))
                  (scale (vector 0.3 0.3 0.3))
                  (draw-sphere)))
       (line-intersect a b))))
```

Evaluation de Primitive

Pdata-for-each-face

pdata-for-each-triangle

pdata-for-each-tri-sample

Le moteur physique

Chargement et sauvegarde de primitives

Ca serrait pratique de pouvoir charger et sauvegarder des primitives, et cela pour plusieurs raisons. Vous voudrez peut être utiliser d'autres applications pour fabriquer des figures et les importer dans Fluxus ou inversement. Il serra assez utilse de sauvegarder les primitives dans un fichier pour ne pas avoir à les recréer dans Fluxus à chaque fois. Il y a seulement deux fonctions à connaître:

Illustration 32: Quelques particules

```
; charge une primitive dans:
(define newprim (load-primitive filename))
; et sauvegardons la dans:
(with-primitive newprim
       (save-primitive filename))
```

Pour le moment ces fonctions ne marchent qu'avec des polygones et des primitives de pixels et vous pourrez y charger/sauvegarder des objects/fichiers png avec.

Le support du format de fichier COLLADA

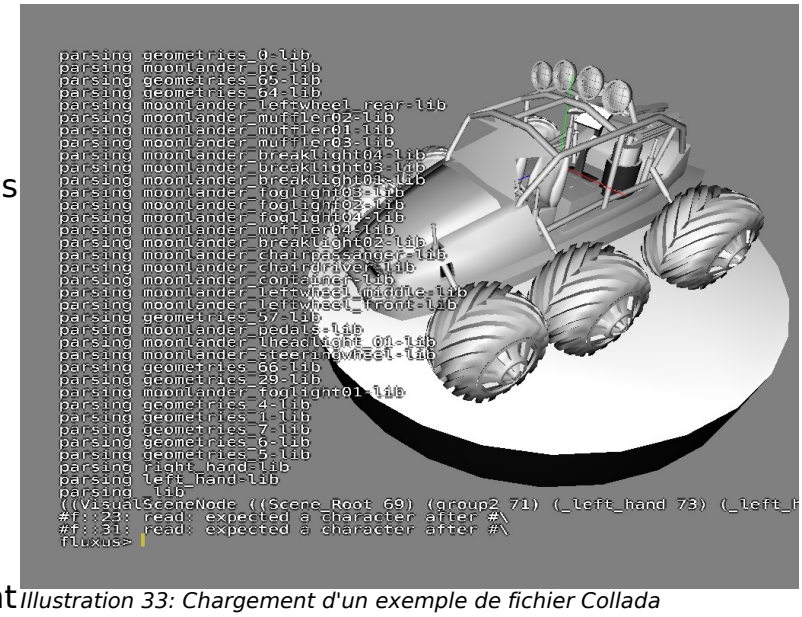

Illustration 33: Chargement d'un exemple de fichier Collada

Collada est le format de fichier utilisé pour les scènes 3d complexes. Ils peuvent être chargé dans Fluxus, actuellement les modèles géométriques supportés sont les données triangulaires, les positions de vertex, les normales, et les coordonnées de texture. L'idée ici est d'utiliser le format Collada pour des scènes complexes contenant différents types géométriques, cela inclus les animations et les données physique. L'export au format Collada est aussi prévu.

Voi la documentation pour la fonction:

```
(collada-import filename)
```

Les Shaders

Les Shaders matériels vous donnent un contrôle plus précis sur les pipelines graphique utilisé pour le rendu de vos objets. Fluxus possède des commandes pour définir et contrôler des Shaders GLSL depuis vos scripts en Scheme, il vous permet même de les modifier dans l'éditeur de Fluxus. GLSL est le protocole standard d'OpenGl pour les Shaders sur beaucoup de types de cartes graphiques. Si votre carte graphique et son pilote d'affichage supportent OpenGl2, cela devrait fonctionner pour vous.

```
(shader vertshader fragshader)
```

Cela charge, compile et lie les vertex aux Shaders dans le contexte courant, ou sur des primitives que vous aurez appelé.

```
(shader-set! paramlist)
```

Définissons des paramètres pour notre Shader dans un jeton, ou une liste de valeurs, par exemple:

```
(list “specular” 0.5 “mycolour” (vector 1 0 0))
```

C'est très simple à faire maintenant, nous n'avons qu'à déclarer dans notre Shader GLSL une valeur uniforme:

```
uniform float deformamount;
```

Qui serra ensuite appelée depuis Scheme:

```
(shader-set! (list “deformamount” 1.4))
```

Derformamount est définie une fois par objet/Shader – d'ailleurs c'est une valeur uniforme pour l'ensemble de l'objet.

Les Shaders peuvent aussi prendre des pdata comme paramètres, pour que vous puissiez ainsi utiliser ces informations avec plusieurs objets ou scripts.

En GLSL:

```
attribute vec3 testcol;
```

Pour passer ça à Scheme, il faudra tout d'abord créer un pdata avec le même nom:

```
(pdata-add “testcol” “v”)
```

Comme cela vous pourrez ainsi définir plusieurs pdata, qui contrôleront les paramètres des Shaders vertex par vertex.

Les echantillons (Samplers)

Les samplers sont aux textures ce que les Shaders matériels sont aux objets, le mot samplers est utilisé à toutes les sauces de nos jours, dans notre cas ils sont utilisés pour fire transiter des informations (qui ne sont pas forcément visuelles) entre plusieurs Shaders. Pour faire transiter des textures vers des Shaders GLSL depuis Fluxus est très simple.

Dans votre Shader GLSL:

```
uniform sampler2D mytexture;
```

En scheme:

```
(texture (load-texture “mytexturefile.png”))
(shader-set! (list “mytexture” 0))
```

Ceci dit à GLSL d'utiliser l'unité de la première texture comme sampler pour ma texture. C'est cette unité de texture que la commande texture utilise lorsqu'elle charge une texture en fait. Pour faire transiter plus qu'une seule texture:

En GLSL:

```
uniform sampler2D mytexture;
uniform sampler2D mysecondtexture;
```

En scheme:

```
; charge vers la texture d'unité 0
(multitexture 0 (load-texture “mytexturefile.png”))
; charge vers la texture d'unité 1
(multitexture 1 (load-texture “mytexturefile2.png”))
(shader-set! (list “mytexture” 0 “mysecondtexture” 1))
```

Le constructeur de tortues

La tortue constructrice de polygones est une manière encore expérimentale de construire des objets polygonaux en utilisant une « tortue » en 3d. En faisant déplacer la tortue dans l'espace vous dessinez des vertex qui formeront des figures procédurales. La tortue peut aussi être utilisée pour déformer des primitives de polygones déjà existants en les attachant aux objets que vous avez déjà crée.

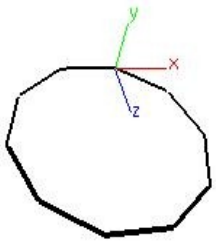

Illustration 34: Un polygone à 10 cotés

Ce script construit simplement un cercle polygonal en utilisant le bon vieux système d'appel récursif c'est à dire que nous faisons avancer le tortue un tout petit peu, puis nous la tournons un peu, nous la faisons avancer un peu, nous la tournons un peu, etc...

```
(define (build n)
(turtle-reset)
(turtle-prim 4)
(build-loop n n)
(turtle-build))

(define (build-loop n t)
       (turtle-turn (vector 0 (/ 360 t) 0))
       (turtle-move 1)
       (turtle-vert)
       (if (< n 1)
              0
              (build-loop (- n 1) t)))


(backfacecull 0)
(clear)
(hint-unlit)
(hint-wire)
(line-width 4)
```

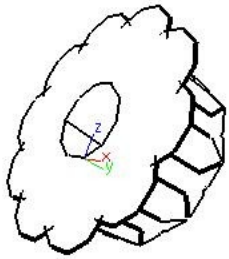

Illustration 35: Une rosace

```
(build 10)
```

Pour un exemple un peu plus complexe, modifiez la fonction build-loop comme cela:

```
(define (build-loop n t)
       (turtle-turn (vector 0 (/ 360 t) 0))

       (turtle-move 1)
       (turtle-vert)
       (if (< n 1)
              0
              (begin
                     ; ajoute un autre appel à la récursion
                     (build-loop (- n 1) t)
                     (turtle-turn (vector 0 0 45))   ; tourne un peu
                     (build-loop (- n 1) t))))
```

Remarques sur l'écriture de gros scripts dans Fluxus

En créant des scripts de taille assez importante, j'ai réalisé qu'il y avait certains aspects du langage (de PLT Scheme pour être plus précis) qui étaient essentiels à connaître dans la gestion et l'écriture de scripts.

Par exemple, on a commencé à s'habituer aux listes pour stocker des données. Utilisons une liste pour définir un robot que l'on souhaitera contrôler par la suite:

```
(define myrobot (list (vector 0 0 0) (vector 1 0 0) (build-cube)))
```

La liste stocke la position du robot, sa vitesse et sa primitive associée au robot (dans notre exemple, le robot est un cube).

On pourra alors utiliser:

```
(list-ref myrobot 0) ; retourne la position du robot
(list-ref myrobot 1) ; retourne la vitesse du robot
(list-ref myrobot 2) ; retourne la primitive du robot
```

pour obtenir les valeurs du robot pour ensuite les utiliser. Ca paraît assez simple vu comme ça mais ça deviendrait très vite lourd à gérer si l'on devait gérer un monde de robots.

```
; construit un monde de 3 robots
(define world (list (list (vector 0 0 0) (vector 1 0 0) (build-cube))
       (list (vector 1 0 0) (vector 1 0 0) (build-cube))
       (list (vector 2 0 0) (vector 1 0 0) (build-cube)))
```

Et maintenant si nous voulions obtenir la primitive du second robot:

```
(list-ref (list-ref world 1) 2)
```

vous voyez bien que ça devient impossible à gérer si l'on devait contrôler des centaines de robots.

Les structures

Les structures, ou Structs ne sont que des encapsuleurs qui vous permettront de nommer un groupe de données. Il est beaucoup plus facile et efficace de les

gérer avec des structs qu'avec des listes.

Reprenons l'exemple des robots:

```
(define-struct robot (pos vel root))
```

Ou « pos » est la position courante du robot, « vel » représente la vitesse et « root » la primitive du robot. La fonction define-struct génère automatiquement des accesseurs:

```
(define myrobot (make-robot (vector 0 0 0) (vector 0 0 0) (build-cube)) ; crée un nouveau robot
(robot-pos myrobot) ; retourne la position du robot
(robot-vel myrobot) ; retourne la vitesse du robot
(robot-root myrobot) ; retourne la primitive du robot
```

Cela rend le code beaucoup plus lisible, car toutes les données sont désormais associées à un nom évocateur. De même il serra possible de créer de nouveaux robots sans avoir à les redéfinir à chaque fois.

Un monde de robots serrait alors:

```
(define-struct world (robots))
```

Et pourrait être utilisé comme cela:

```
; construit un monde de 3 robots
(define myworld (make-world (list (make-robot (vector 0 0 0) (vector 1 0 0) (build-cube))
       (make-robot (vector 1 0 0) (vector 1 0 0) (build-cube))
       (make-robot (vector 2 0 0) (vector 1 0 0) (build-cube)))))

; retourne la primitive du second robot
(robot-root (list-ref (world-robots myworld) 1))
```

Les états mutables

jusqu'à maintenant nous n'avons programmé que fonctionnellement, et n'avons pas modifié la mémoire (du moins pas explicitement), cela notament à cause des effets de bord que Scheme souhaite éviter. Donc pour pouvoir changer « l'état » d'une struct, il faut lors de sa définition:

```
(define-struct robot ((pos #:mutable) (vel #:mutable) root))
```

Cela génèrera les fonctions suivantes:

```
(set-robot-pos! robot (vector 1 0 0))
(set-robot-vel! robot (vector 0 0.1 0))
```

Pour que vous puissiez modifier les données dans une struct.

Pour un exemple plus concret, referez vous au fichier dancing-robots.scm dans le répertoire des exemples.

Les Classes

Créer des séquences vidéo

Fluxus est destiné à une utilisation en temps réel, cea comprend les démonstrations en public et les jeux, mais vous pourrez aussi utiliser les

commandes de sauvegarde d'images pour ensuite génerer une vidéo. Cette procédure est assez complexe lorsqu'il faudra synchroniser la vidéo à l'audio, OSC ou encore au clavier.

Utilisé seul, le dump des images va seulement sauvegarder les images que votre machine doit rendre. Ca peut être utile dans certains cas, mais pas si vous voulez ensuite générer une vidéo avec un taux d'images fixe, il faudrait alors faire en sorte de synchroniser l'audio de façon à avoir un taux d'images de 25 images par seconde.

Synchroniser à l'audio

La fonction process réalise plusieurs choses: il redirige l'audio venant de votre port jack d'entrée déclaré vers un fichier sur votre disque. Mais il fait aussi en sorte que chaque tampon audio ne produise qu'un seule image à la fois. En réalité lors d'opérations en temps réel, les buffers audio seront « jetés » ou dupliqués, en fonction du taux d'images à la seconde et de l'échantillonnage.

Alors en gros si vous voulez générer une vidéo avec un framerate de 25fps, et de l'audio échantillonné à 44100 Hz, il faudrait alors calculer: 44100/25=1764 samples audio par image. Configurez la taille de votre tampon audio à cette valeur. Ainsi tout ce qui vous restera à faire serra de vous assurer que process et start-framedump sont appelés à la même image, pour que le son et l'image soient synchronisés.

Comme cette procédure ce ne déroule pas en temps réel, vous pouvez configurer la résolution comme bon vous semble, ou avoir un script aussi complexe soit-il.

Synchroniser au clavier pour l'enregistrement de sessions live

Vous pouvez utiliser l'enregistreur de touches pour sauvegarder vos démos de livecoding et ainsi pouvoir les relancer plus tard.

Pour l'utiliser, lancez Fluxus avec le paramètre -r ou -p (regardez la doc interne de Fluxus pour plus d'infos). Cela enregistrera l'ensemble des touches pressées et la durée ou elles sont appuyées. Il pourra ainsi les réinterpreter peut importe le framerate utilisé.

L'enregistreur de touches fonctionne avec la fonction process de la même façon que process avec l'audio (vous aurez toujours besoin d'une piste audio, même muette). Donc l'enregisteur « avancera » en fonction du nombre d'images qui serra rendu au lieu d'utiliser l'horloge interne de l'ordinateur. Encore une fois, vous pouvez prendre votre temps lors de la création de vos scripts, car lors de la relecture, tout serra refait de manière fluide et non-stop.

Enregistrer les messages OSC est possible (pour stocker des infos comme l'activité d'un manette de jeu). Faites le moi savoir si ça vous interesse.

Résolution des problèmes de synchronisation

Réussir à synchroniser l'audio avec le rendu peut être assez hardu, les problèmes les plus fréquents que j'ai rencontrés sont distinguables en deux catégories:

des lags de synchronisation qui augmente au fur et à mesure

l'appel de start-audio s'est fait avec la mauvaise taille de tampon. Configurez le correctement et relancez votre script. Il se peut qu'àpres 20 minutes d'animation, de courts lag commencent à apparaître, ce qui est inévitable.

Désynchronisation constante perpetuelle

Cela arrive lorsque l'audio ne commence pas en même temps que le rendu. Vous pouvez essayer d'ajouter/supprimer des « blancs » au début de votre source audio. Personnellement je n'enregistre que quelques secondes d'animation pour étalonner mon enregistrement avant de vraiment démarrer.

Fluxus dans DrScheme

DrScheme est un environnement de développement intégré pour Scheme, et est compris dans PLT Scheme, donc il est forcement installé si vous avez compilé Fluxus.

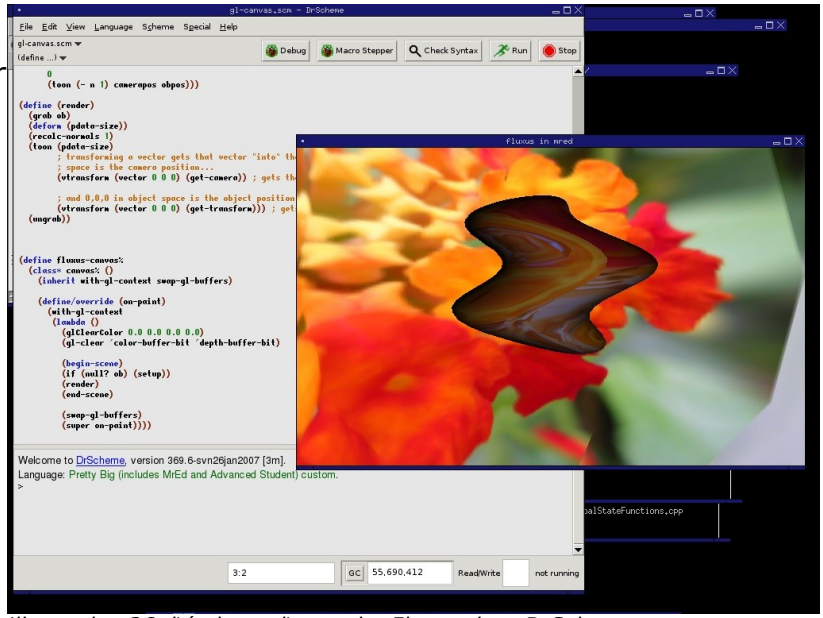

Illustration 36: l'écriture d'un script Fluxus dans DrScheme

Vous pouvez l'utiliser à la place de l'éditeur de Fluxus pour écrire vos scripts.

Les raisons pour lesquelles vous voudriez utiliser DrScheme sont les suivantes:

- possibilité de débuger
- un environnement d'édition meilleur que tout ce que pourra proposer Fluxus
- rend Fluxus très utile lorsque l'on souhaite juste rendre des objets dans Scheme

Je l'utilise très souvent lorsque je dois écrire de gros scripts. Tout ce que vous aurez à faire est rajouter cette ligne au début de votre script:

```
(require fluxus-[version]/drflux)
```

ou [version] est la version actuellement installé sans le tiret, par exemple « 016 » pour la version 0.16

Chargez un script et lancez le: une nouvelle fenêtre s'ouvre, et affiche le résultat de votre script. Relancez le script devrait alors raffraîchir cette fenêtre automatiquement.

Problèmes rencontrés

Certaines commandes font planter DrScheme: show-fps ne devrait pas être utilisé. Les Shaders matériels ne fonctionnent pas non plus. De plus DrScheme demande beaucoup de mémoire vive, ce qui pourrait poser problème.

Divers autres informations

Ce chapitre est pour des choses que je pense vraiment importantes à savoir, mais ne trouvent pas de place.

Obtenir une énorme vitesse de framerate

Par default fluxus a son framerate '' etrangle '' .Pour le supprimer vous pouvez utilisez :

```
(desiredfps 1000000)
```

Un tel framerate n'est pas garantie, mais il arretera le plafonnement de la vitesse par fluxus(qui par defaut est au tour de 50fps). Utiliser :

```
(show-fps 1)
```

pour verifier le fps avant et après. Des framerates eleve sont bon pour le VJing, il est essentiel de reduire le temps de latence des resultats des calculs audio – il est beaucoup plus réceptif.

Tests Unitaires

Si vous voulez vérifier que fluxus travaille correctement sur une nouvelle installation – ou si vous soupçonnez que quelque chose ne vas pas, essayez :

```
(self-test #f)
```

qui executera un exemple de chaque script contenue dans la documentation de la reference de fonction. Si il plante ou il y' a une erreur - lancez:

```
(self-test #t)
```

qui permettra de sauver un fichier log – veillez l'envoyer sur la mailling list et nous essayerons de le corriger.

Il est fortement recommandé aux developpeurs d'exécuter cette commande avant d'envoyer son code à la source de dépôt, de sorte que vous pouvez voir les effets de vos modifications.

Scratchpad Fluxus et modules

Ce chapitre documente sur du fluxus du niveau inferieur, uniquement si vous

voulez bricoler un peu plus.

Fluxus est constitué de deux moitié. La première moitié est une fenêtre contenant l'editeur de script rendu au dessus de la scène .C'est ce qu'on appel le fluxus scratchpad(bloc note fluxus, et c'est la façon d'utiliser fluxus pour du livecoding et une lecture generale.

L'autre moitié est un ensemble de module qui fournit les fonctions pour faire du graphisme, ils peuvent etre chargés dans n'importe quel interpreteur mzscheme, et execute en tous contexte OpenGL.

Modules

Les fonctionnalités de Fluxus sont separées entre les différents modules Scheme. Vous n'avez pas besoin de les connaître pour une utilisation simplifiée de fluxus, elles sont chargées et configure pour vous.

fluxus-engine

cet executable contient l'extension des fonctions de rendu de base, et la majorité des commandes.

fluxus-audio

Executable contenant un client jjack et les commandes du processeur fft .

fluxus-osc

Une extension binaire avec le serveur OSC et le client, et les messages de commande.

fluxus-midi

Une extension binaire avec un support entrée d'evenement midi.

Modules Scheme

Il y'a aussi de nombreux modules scheme qui viennent avec fluxus. Certains d'entre eux forment l'interface scratchpad et vous fournit les interfaces souris/clavier et les réglages camera, d'autres couches au dessus de fluxus-engine afin de le rendre plus pratique. C'est là que les choses comme with-* et pdata-map! Macros sont precisées et l'importation/exportation par exemple.

Fluxa

Fluxa est une option ajouter à fluxus qui ajoute de la synthèse audio et de la lecture d'echantillon. Il est aussi un synthetiseur non-determinist experimental où chaque 'note' est son propre synthétiseur graphique.

Fluxa est un framework pour construire et sequencer du son. Il utilise un style minimal et purement fonctionnel qui est conçu pour du liveconding. Il peut etre décompose en deux partiesy, la description des graphiques de synthese et un ensemble de formes linguistiques pour décrire les sequences procedurales

(Fluxa est aussi une sorte de primitive hommage à supercollider – voir aussi

rsc, qui est une liaison de scheme à sc)

Exemple:

```
(require fluxus-016/fluxa)

(seq
       (lambda (time clock)
              (play time (mul (sine 440) (adsr 0 0.1 0 0)))
              0.2))
```

Joue un ton sine avec une decroissance de 0,1 secondes toutes les 0.2 secondes.

Non-determinisme

Fluxa a decidement des propiétés non-determinists – la durée de vie du synthé est liée à certaines contraintes globales:

- Un nombre fixe d'operateurs, qui sont recyclés (allocation temps/espace contrainte)
- Un nombre maximum de synthe qui joue simultanement (Contraninte de temps cpu)

Ce qui veut dire que les synthé sont arretés après une periode de temps variable., en fonction de la necessité de nouvel opérateurs. Les noeuds qui composent les synthé graphiques peuvent également êtres recyclés tant qu'il sont en cour d'utilisation – aboutissant à des objets interessants (qui peut être considéré comme une fonctionnalité!)

Commandes de Synthèse

```
(play time node-id)
```

Programme l'id du noeud(node-id) qui sera joué à l'aide de time. Ceci est pour la musique en génerale.

```
(play-now node-id)
```

Joue le node-id le plus tot que possible – principalement pour tester.

Operateurs de noeuds

Toutes ces commandes crée et retourne un noeud qui peut etre jouer. Les paramètres du graph de synthèse peuvent être d'autres noeuds ou des valeurs normales.

Generateurs

```
(sine frequency)
```

Une onde sinusoïdale à la frequence specifiée.

```
(saw frequency)
```

Une onde scié à la fréquence spécifié

```
(squ frequency)
```

Une onde au carrée à la fréquence specifiée

```
(white frequency)
```

Bruit blanc

```
(pink frequency)
```

Bruit Rose

```
(sample sample-filename frequency)
```

Charge et joue un échantillon – les fichiers peuvent êtres relatifs à l'emplacement de recherche. Les échantillons vont êtres chargés de manière asynchrine, et de ne vont pas interférer avec l'audio en temps réel.

```
(adsr attack decay sustain release)
```

Genere un signal enveloppe

Maths

```
(add a b)
(sub a b)
(mul a b)
(div a b)
```

Rappele vous que ces parametres peuvent être des noeuds ou des nombres, vous pouvez faire les choses comme cela:

```
(play time (mul (sine 440) 0.5))
```

or

```
(play time (mul (sine 440) (adsr 0 0.1 0 0)))
```

Filtres

```
(mooghp input-node cutoff resonance)
(moogbp input-node cutoff resonance)
(mooglp input-node cutoff resonance)
(formant input-node cutoff resonance)
```

Audio Global

```
(volume 1)
```

N'est-ce pas que ce qui est dit sur l'étain

```
(eq 1 1 1)
```

Tweak bass, mid, treble

```
(max-synths 20)
```

Change le maximume de synthé joués en même temps – par defaut un petit 10

```
(searchpath path)
```

Ajouter un chemin pour le chargement des echantillons.

```
(reset)
```

La commande de panique – supprime tous les graph de synthé et reinitialise tous les opérateurs – pas sûr pour rt.

Commandes de Sequençages

Fluxa fournit un ensemble d'outils pour le sequençage.

```
(seq (lambda (time clock) 0.1))
```

La séquence de haut niveau – Il ne peut exister qu'un de ce genre, et tous le code fourni dans la procedure sera appelé au moment opportun. Le temps entre deux appels est définie par la valeur retourné par la procedure. – de sorte que vous pouvez changer le timing global dynamiquement.

Les paramètres de temps et d'horloge sont passés à la procedure – le temps est un float avec une valeur en temps réel , passé aux commandes play. Il s'agit en fait d'un peu d'avance sur le temps réel, afin de donner le temps au réseau pour obtenir les messages du serveur.

Vous pouvez egalement faire comme suit:

```
(play (+ time 0.5) ...)
```

Qui va compenser le temps d'une demi-seconde à l'avenir. Vous pouvez également les faire executer plus to –mais seulement un peu.

Clock est une valeur toujours croissante, qui est incrementer à chaque fois que la procedure donnée à la séquence est appelée. La valeur n'est pas très importante, mais vous pouvez utiliser zmod, qui est simplement la procedure prédefinie :

```
(define (zmod clock v) (zero? (modulo clock v)))
```

Ce qui est assez pour faire du raccourcissement utile.

```
(if (zmod clock 4) (play (mul (sine 440) (adsr 0 0.1 0 0))))
```

Jouera une note à chaque 4 beats.

```
(note 10)
```

Un utilitaire pour le mapping de numero de note en fréquence(je pense que le barème actuel est egual à la tendance)

```
(seq
	(lambda (time clock)
		(clock-map
			(lambda (n)
				(play time (mul (sine (note n)) (adsr 0 0.1 0 0)))) clock
		(list 10 12 14 15))
	0.1))
```

clock-map map la liste à la commande jouer à chaque tick de l'horloge – La liste peut etre utilisée comme une primitive sequence, et peut evidemment être utilisée pour conduire beacoup plus que la simple hauteur.

```
(seq
(lambda (time clock)
(clock-switch clock 128
(lambda ()
(play time (mul (sine (note n)) (adsr 0 0.1 0 0)))) (lambda ()
(play time (mul (saw (note n)) (adsr 0 0.1 0 0))))) 0.1))
```

Cette horloge-interrupteur bascule entre les procedures tous les 128 ticks d'horloge – pour la structure de niveau superieur.

Synchronisation

Un message osc peut être envoye au client pour une synchronisation pour des performances collaboratives.Le format du message de synchronisation est le suivant:

```
/sync [iiii] timestamp-seconds timestamp-fraction beats-per-bar tempo
```

Lors de la synchronisation ,Fluxa fournit deux autres definitions globales:

sync-clock : une horloge qui est remis à zéro lorsque le /sync iest reçu

sync-tempo : Demande le tempo actuel (vous etes libre de le modifier ou de l'ingorer)

[note: Il existe un programme qui ajoute timestamps à /sync des messages provenant d'un autre reseau, qui fait de la synchro collaborative fonctionnement correctement (comme il n'a pas besoin d'horloge pour être synchroniser aussi) envoyer moi un mail pour plus d'info]

Problemes courants/a faire

- Record execution – Les graph cycliques ne fonctionne pas
- Execution permanente de quelques noeuds – nous fixerons delay/reverb

Frisbee

Frisbee est un moteur de jeux simplifié. Il est ecrit dans un language différent du reste de fluxus, et ne necessite pas de connaissance ou l'utilisation de l'une ou l'autre commande fluxus.

Le language utilise s'appel 'Father Time' (FrTime), qui est un language de programmation fonctionnel reactif disponible dans le cadre de PLT scheme. La programmation fonctionnelle réactive (frp - Functional reactive programming) est un mode de programmation qui met l'accent sur les comportements et les evènements, et en fait un élément central du language

Programmer un environement graphique comme un jeu est une question de description de scène et des comportements qui change au fil du temps.En utilisant les languages de programmation normaux (comme la base de fluxus) vous avez généralement besoin de faire ces choses séparement, construire une scène, ensuite l'animé.En utilisant le FRP, on peut décrire la scene avec les comportements à l'interieur. L'idée est de faire de plus petit programmes et plus simples à modifier, rendant ainsi le processus de programmation plus créative.

Une simple scene frisbee

Ceci est le ce qui est le plus simple de scene frisbee :

```
(require fluxus-015/frisbee)

(scene
  (object))
```

(scene) est la commande principale de frisbee- elle est utiliser pour definir une liste d'objet et leurs comportements.

(object) créer un objet solid, par defaut le cube (bien sur! :)

Nous pouvons modifié notre objet en utilisant l'option 'keyword parameters', ils travaillent comme ça:

```
(scene
  (object #:shape 'sphere))
```

Ceci definie la forme de l'objet – Ci-après quelques formes construite en t:

```
(object #:shape 'cube)
(object #:shape 'sphere)
(object #:shape 'torus)
(object #:shape 'cylinder)
```

Ou, vous pouvez aussi charger un fichier .obj pour faire vos propres formes.

```
(object #:shape "mushroom.obj")
```

Ces fichiers objets sont relatifs à l'endroit où vous lancer fluxus, ou ils peuvent aussi vivre quelque part dans l'emplacement de recherche de fluxus (que vous pouvez definir dans votre .fluxus.scm script utilisant (searchpath)).

Si nous voulons changé la couleur de notre cube nous pouvons ajouter un nouveau parametre:

```
(object
    #:colour (vec3 1 0 0))
```

The vec3 specifies the rgb colour, so this makes a red cube. Note that frisbee uses (vec3) to make it's vectors, rather than (vector).

Voici d'autres parametres que vous pouvez regler sur l'objet:

```
(object
    #:translate (vec3 0 1 0)
    #:scale (vec3 0.1 1 0.1)
    #:rotate (vec3 0 45 0)
```

```
    #:texture "test.png"
    #:hints '(unlit wire))
```

L'odre de specification des paramètres n'a pas d'importance, le résultat est le même.La transformation de l'ordre est toujours de translater(translate) d'abord, puis pivoter (rotate), et l'echellonnage(scale).

Animation

FrTime fait un mouvement specifique très simple:

```
(object
    #:rotate (vec3 0 (integral 10) 0))
```

Fait un cube qui effectue une rotation de 10 degrees chaque seconde. Plutôt que de fixer les angles de façon explicite, précise la valeur de l'integral de la rotation à chaque seconde. Nous pouvons aussi faire cela:

```
(object
    #:rotate (vec3-integral 0 10 0))
```

Qui peut etre plus simple dans certaines situation.

Rendre les choses Reactives

Ce que nous avons fait avec la commande intégrale est ce qu'on appel comportement sa valeur depend du temps.Il s'agit d'une caracteristique essentielle de FrTime,et il existe de nombreuses façons de créer et de manipuler les comportements. Frisbee vous donne egalement certains comportements par defaut qui représentent les informations changeantes venant du monde exterieur:

```
(object
    #:rotate (vec3 mouse-x mouse-y 0))
```

Cela fait tourner le cube en fonction de la position de la souris.

```
(object
    #:colour (key-press-b #\c (vec3 1 0 0) (vec3 0 1 0)))
```

Cela change la couleur du cube lorsque vous appuyez sur la touche 'c'.

```
(object
    #:translate (vec3 0 (key-control-b #\q #\a 0.01) 0))
```

Cela bouge le cube de haut en bas lorsque vous appuyer sur les touche 'q' et 'a' , de 0.01 unités.

Objets Frais

Jusqu'à présent tous les objets que nous avons créés sont restés actifs pendant la durée du programme.Quelques fois nous voulons controler le temps de vie d'un objet, ou créer un nouveau. Ceci est evidemment important pour beaucoup de jeux! Pour faire cela, nous avons besoin d'introduire les eévènements (events).Les evenements sont d'autres éléments fondamental de FrTime, par consequence Frisbee; et les comportements peuvent peuvent êtres transformés en événement et vice-versa.Les evènements sont des chosess qui se produisent à un moment précis, contrairement au comportements à qui on peut demander leurs valeurs actuelles.Pour cette raison les evènements ne peuvent pas être utilisé directement pour la conduite des objets dans Frisbee, au même titre que les comportements le peuvent – mais ils sont utilisés pour le déclenchement de nouveaux objets en debut ou fin d'existence.

Voici un script qui cree un flot continue de cubes :

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        (metro 1) 5))
```

Il y'a beaucoup de nouvelles choses qui se passent ici. Tout d'abord il y'a la commande metro diminutif de metronome, qui créer un flux d'evenements au taux specifié (1 par seconde dans ce cas). (factory) est une commande qui ecoute un flux d'evenement – pris comme second argument, et execute une procédure passée comme son premier argument à chaqu'un(en cas de passage d'argument à la fonction fourni).

Dans ce cas chaque fois qu'un evenement se produit la fonction anonyme est executer,ce qui créer un objet qui s'eloigne de l'origine. Gauche comme ce frisbee vaudrait eventuellement ralentir à une exploration, de plus en plus de cube Lefsont créer. Ainsifactory prend le troisieme paramètre, qui est le maximum de chose qu'il peut maintenir en vie en même temps.Une fois que 5 objets ont été créer il va les recycler et supprimer les vieux objets.

Frisbee vient avec quelques evenements construits qu'on peut voir avec le script:

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        keyboard 5))
```

Qui engendre un cube à chaque fois que vous appuyer sur le clavier.
Jusqu'à present nous avons ete en ignorant l'evenement qui était placé dans notre petite fonction à faire des cube, mais comme les évènements generé par le clavier sont les touches que nous avons appuyer, on peut les utiliser ainsi :

```
(scene
    (factory
        (lambda (e)
            (if (char=? e #\a) ; make a bigger cube if 'a' is pressed
                (object
                    #:translate (vec3-integral 0.1 0 0)
                    #:scale (vec3 2 2 2))
                (object #:translate (vec3-integral 0.1 0 0))))
        keyboard 5))
```

Convertir des comportements en évenements

Vous pouvez créer des évenements quand un comportement change:

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        (when-e (> mouse-x 200)) 5))
```

Particles

Frisbee vient avec son propre systeme de primitive particle. - qui facilite la construction de differents effets particle.

Il sont créer de la même maniere que les objets solides:

```
(scene
    (particles))
```

Et vient avec un ensemble de paramètres ,que vous pouvez controler explicitement ou par les evènements:

```
(scene
    (particles
        #:colour (vec3 1 1 1)
        #:translate (vector 0 0 0)
        #:scale (vector 0.1 0.1 0.1)
        #:rotate (vector 0 0 0)
        #:texture "test.png"
        #:rate 1
        #:speed 0.1
        #:spread 360
        #:reverse #f))
```

Référence des fonctions

La référence des fonctions n'est pas encore disponible en français !