Outils modernes pour le traitement d'images

d'OpenGL à Quartz Composer : rendre des technologies de pointe accessibles

Pierre Chatelier

chatelier@laic.u-clermont1.fr

Plan

- Problématique
- OpenGL & Shading Langage
- Core Image et Quartz Composer
- Application à la détection de couleur

Traitement d'images (I)

Traitement d'images (2)

- Outils du flux de traitement
  - Acquisition (caméra, webcam, film, image)
  - Transformation en un format manipulable
  - Algorithmes en langage de haut niveau
  - Algorithmes sur GPU
- API privilégiée : OpenGL et les « shaders »

OpenGL

- Dessin image par image
- Principe de transformations par multiplications matricielles
  - matrice <modelView>
  - matrice <projection>
- Définition de paramètres pour chaque vertex
- Interpolation automatique

Exemple d'OpenGL (1)

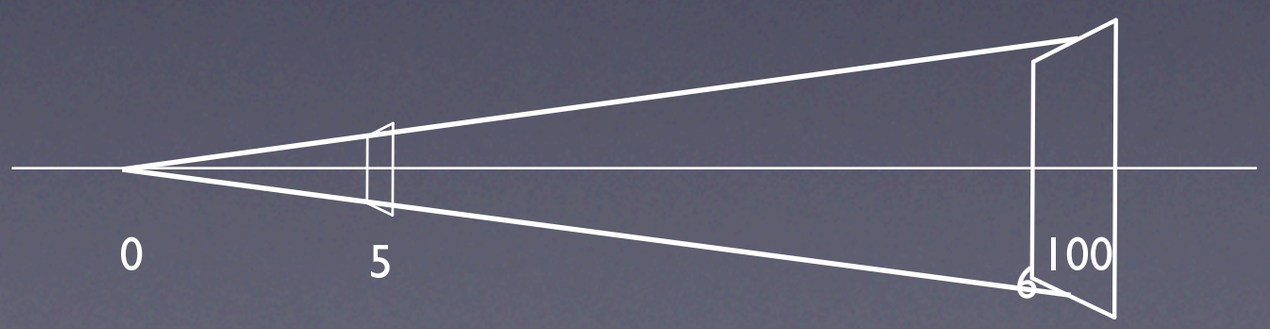
dans le code de dessin (appelé x fois par seconde) :

Réglages caméra

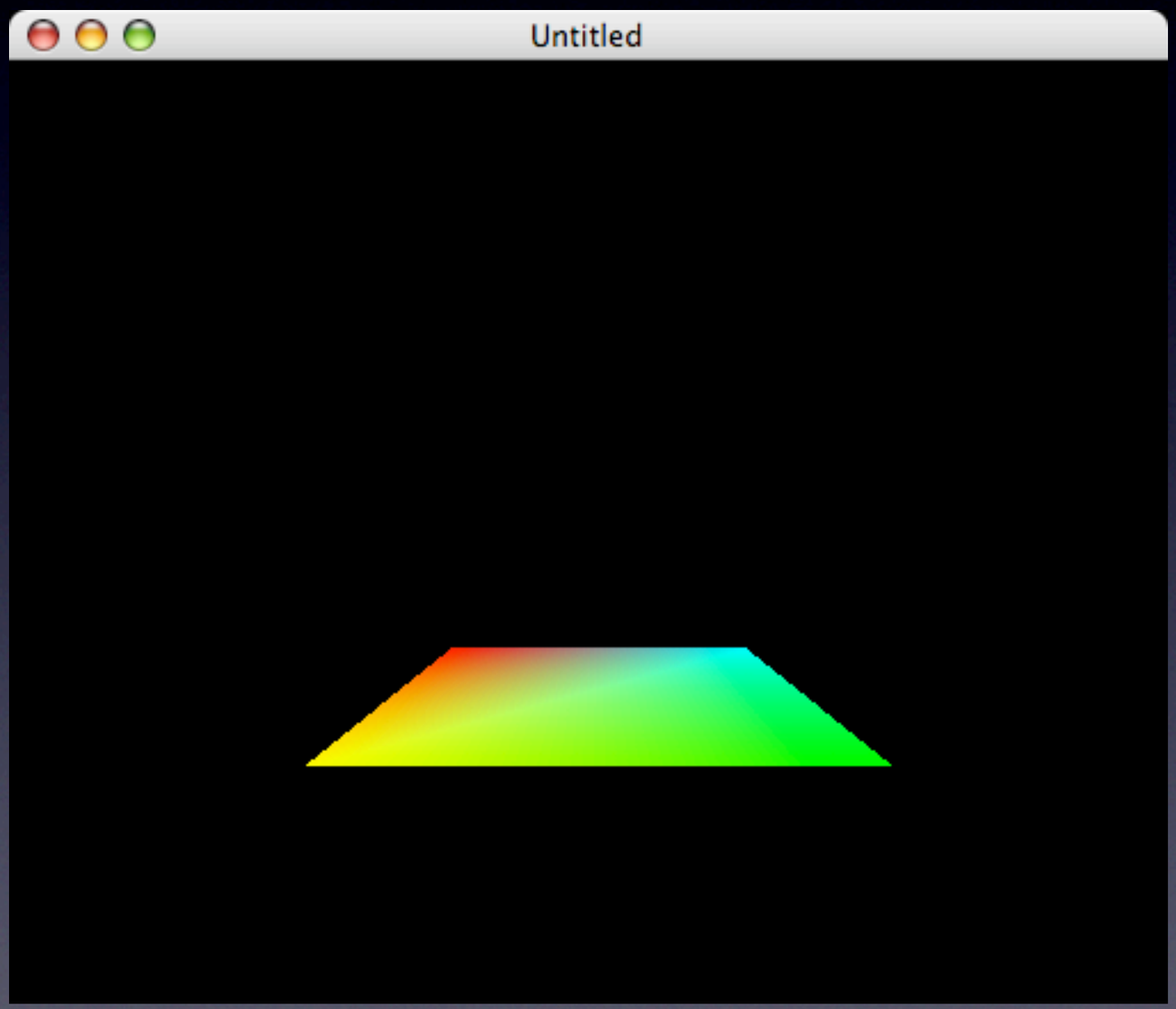
```
//définition de la zone de dessin
glViewport(0, 0, 640, 480);

//définition de la matrice de projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1, 1, -1, 1, 5, 100);

//définition de la matrice de transformation
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslate(0, -1, -20);
```

Création de l'image

```
glBegin(GL_QUAD);
        glColor3f(1, 0, 0);
        glVertex3f(-1, 0, 0);
        glColor3f(1, 1, 0);
        glVertex3f(-1, 0, 10);
        glColor3f(0, 1, 0);
        glVertex3f(1, 0, 10);
        glColor3f(0, 1, 1);
        glVertex3f(1, 0, 0);
glEnd();
```



Exemple d'OpenGL (2)



OpenGL et traitement d'images

- Image d'entrée = texture plein écran face à la caméra
- Image de sortie = ce qu'affiche OpenGL
- Traitement = opérateurs de texture
  - Limité en OpenGL standard (interpolation)
  - Utilisation des « shaders » et du shading language
- OpenGL n'est pas la seule API d'affichage, mais nous allons voir qu'elle convient parfaitement

Shading Language (1)

- Deux actions majeures effectuées par OpenGL:
  - Transformation 3D→3D→2D (local→global→projection) des coordonnées de chaque vertex (Vertex program)
  - Calcul des couleurs interpolées (Fragment program : un fragment est une valeur interpolée entre plusieurs vertices, pas seulement couleur)

Shading Language (2)

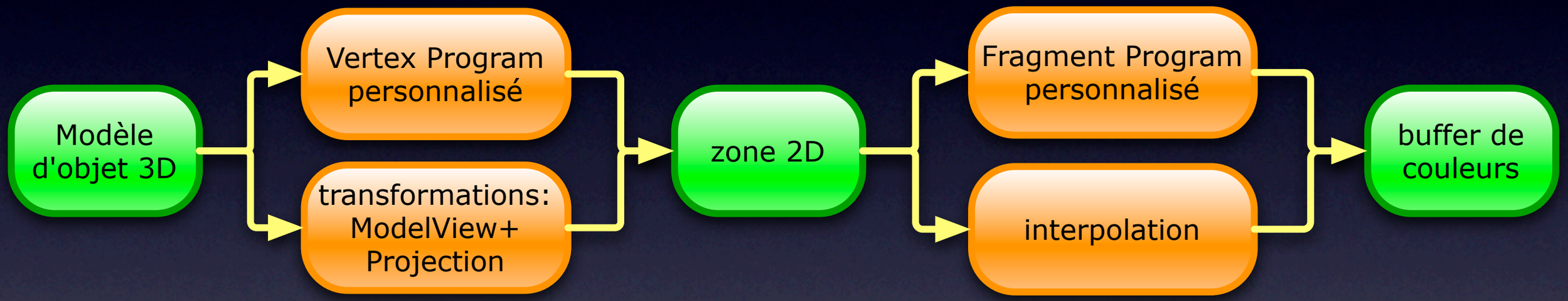
- Les cartes graphiques récentes permettent de spécifier des vertex program/fragment program personnalisés (code C/ OpenGL)
- On définit des vertex shaders/fragment shaders, et on les envoie à la carte au moment voulu
- Remarque :

  vertex program = vertex shader

  fragment program = fragment shader = pixel shader

Shading Language (3)

- Emplacement des shaders dans le flux d'affichage :



- Nota : en réalité, les cartes implémentant OpenGL sont hautement parallélisées

Shading Language (4)

- Démonstration GLSLShowPiece...

Shading Language (5)

- Quel langage pour les shaders ?
  - Assembleur (Spécifique à la carte ATI/NVidia)
  - OpenGL Shading Language (GLSL) : proche du C
  - CG (NVidia) : proche du C, compilable pour différentes architectures (même ATI)
  - HLSL : shaders Microsoft pour DirectX

Shading Language (6)

- Assembleur : trop spécifique, trop difficile, pas pratique
- HLSL : spécifique Microsoft, trop spécialisé pour nos besoins
- CG : très bien, mais un peu lourd à mettre en place (compilations...)
- GLSL : standard, proche du C... et bien suffisant pour notre but final. DIrectement intégrable dans du code C/OpenGL

Créer des shaders (1)

- On écrit un programme :

```
void main(void)
{
   //actions sur des variables internes à la carte
}
```

- Variables OpenGL disponibles en lecture et/ou écriture (gl_Vertex, gl_Normal, glFragCoord...)
- Pas les mêmes pour Vertex et Fragment Shader
- écrire gl_Position pour un Vertex Shader
- écrire gl_FragColor pour un Fragment Shader

Créer des shaders (2)

- Possibilité de créer constantes, variables locales, fonctions
- Possibilité de définir des variables dans les vertex shaders, accessibles interpolées dans les fragment shaders (varying)
- Vertex Shader sans effet:
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
- Fragment Shader sans effet:
  gl_FragColor = gl_color passé par varying

Créer des shaders (3)

- Le programme GLSL est une chaîne de caractères

```
//écriture du shader
const GLcharARB* fragment_string = “...le code du shader...”;

//compilation du shader
GLhandleARB fragment_shader, program_object;
glShaderSourceARB(fragment_shader, 1, &fragment_string, NULL);
glCompileShaderARB(fragment_shader);
glGetObjectParameterivARB(fragment_shader, GL_OBJECT_COMPILE_STATUS_ARB,
                          &fragment_compiled);
//chargement du shader
program_object = glCreateProgramObjectARB();
glAttachObjectARB(program_object, fragment_shader);
glDeleteObjectARB(fragment_shader);
glLinkProgramARB(program_object);
glGetObjectParameterivARB(program_object, GL_OBJECT_LINK_STATUS_ARB, &linked);

//utilisation
glUseProgramObjectARB(program_object); //activation du shader
//code d’affichage personnel (dessin d’un teapot
...
glUseProgramObjectARB(NULL); //désactivation du shader
```

Créer des shaders (4)

- Code OpenGL bien difficile pour utiliser un shader
- Programme de création en temps réel
  - exemple : OpenGL Shader Builder sous MacOS X
  - démonstration...

Créer des shaders (5)

- Code difficile à lire et à débugguer (pas de printf)
- Des effets compliqués (fourrures) reposent sur des idées simples (duplication de l'objet)
- Un effet peut être obtenu en appelant plusieurs shaders, en faisant de "faux" tracés d'objets... L'algorithme d'un effet compliqué ne repose pas sur un seul shader

Application au traitement d'images (I)

- Image = texture sur un rectangle face à l'écran
- Pas besoin du vertex shader
- Représenter l'algorithme avec un pixel shader, et l'appliquer
- Seuls les traitements simples sont concernés, notamment les filtres (flou, extraction de contours...)

Application au traitement d'images (2)

Problème : investissement technique énorme

- Le langage GLSL est « simple », mais les algorithmes difficiles à tester
- OpenGL n'est pas pratique à l'usage

- Écrire un code pour transformer l'image en texture, envoyer les shaders à la carte...
- Interfacer cette partie OpenGL avec le reste d'un programme (injecter l'image, chaîner des traitements, récupérer l'image, mixer avec des algorithmes écrits en C)

Application au traitement d'images (3)

Solution avancée sous MacOS X

- Librairie Core Image
- Logiciel Quartz Composer
- Intégration à du code C/Objective-C

Core Image (1)

- API Cocoa
- trois classes principales : CIImage, CIKernel et CIFilter
- CIImage encapsule une image (lue sur disque, ou sur flux vidéo)
- Un CIKernel exécute un filtre écrit en GLSL
- Un CIFilter encapsule des CIKernels, CISamplers...
- Création d'une chaîne de CIKernels, injection de CIImage, récupération de CIImage
- Toute la partie difficile d'intégration disparaît

Core Image (2)

API « intelligente » (1/2)

- Core Image réalise une compilation des Kernels à la volée
- Core Image analyse les CIKernel utilisés pour les appliquer dans l'ordre le moins coûteux (crop avant flou par exemple)

implémentation naïve

Core Image (2)

API « intelligente » (1/2)

- Core Image réalise une compilation des Kernels à la volée
- Core Image analyse les CIKernel utilisés pour les appliquer dans l'ordre le moins coûteux (crop avant flou par exemple)

implémentation intelligente

Core Image (3)

API « intelligente » (2/2)

- Une CIImage n’est qu’une « recette »
- La CIImage sortante n’est calculée qu’au besoin
- Il n’y a pas de duplication des données images lors d’un chaînage des traitements : économie mémoire

Core Image (4)

Exemple de code (Objective-C):

```
NSString*  path = @"/User/chatelier/image.png";
NSURL*      url = [NSURL fileURLWithPath:path];

//charger l'image
CIImage* source = [CIImage imageWithContentsOfURL:url];

//choisir un filtre pré-existant
CIFilter *hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
[hueAdjust setValue:source forKey:@"inputImage"];
[hueAdjust setValue:[NSNumber numberWithFloat:2.094] forKey:@"inputAngle"];

//récupérer le résultat
CIImage* result = [hueAdjust valueForKey:@"outputImage"];
```

Tout aussi simple pour injecter des CIKernels.

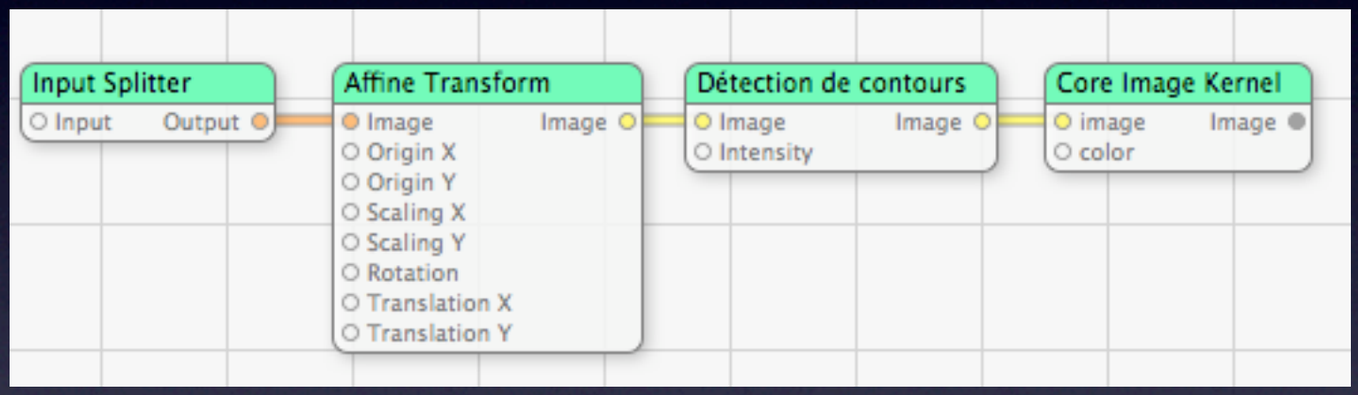
Core Image (5)

- Pas besoin de contexte graphique
- Code simple, lisible, peu technique
- Tests grandement facilités
- Chaînage très simple, et facilité d'insertion d'algorithmes écrits en C
- Résultat temps réel (Démonstration CoreImage Fun House)

Quartz Composer (1)

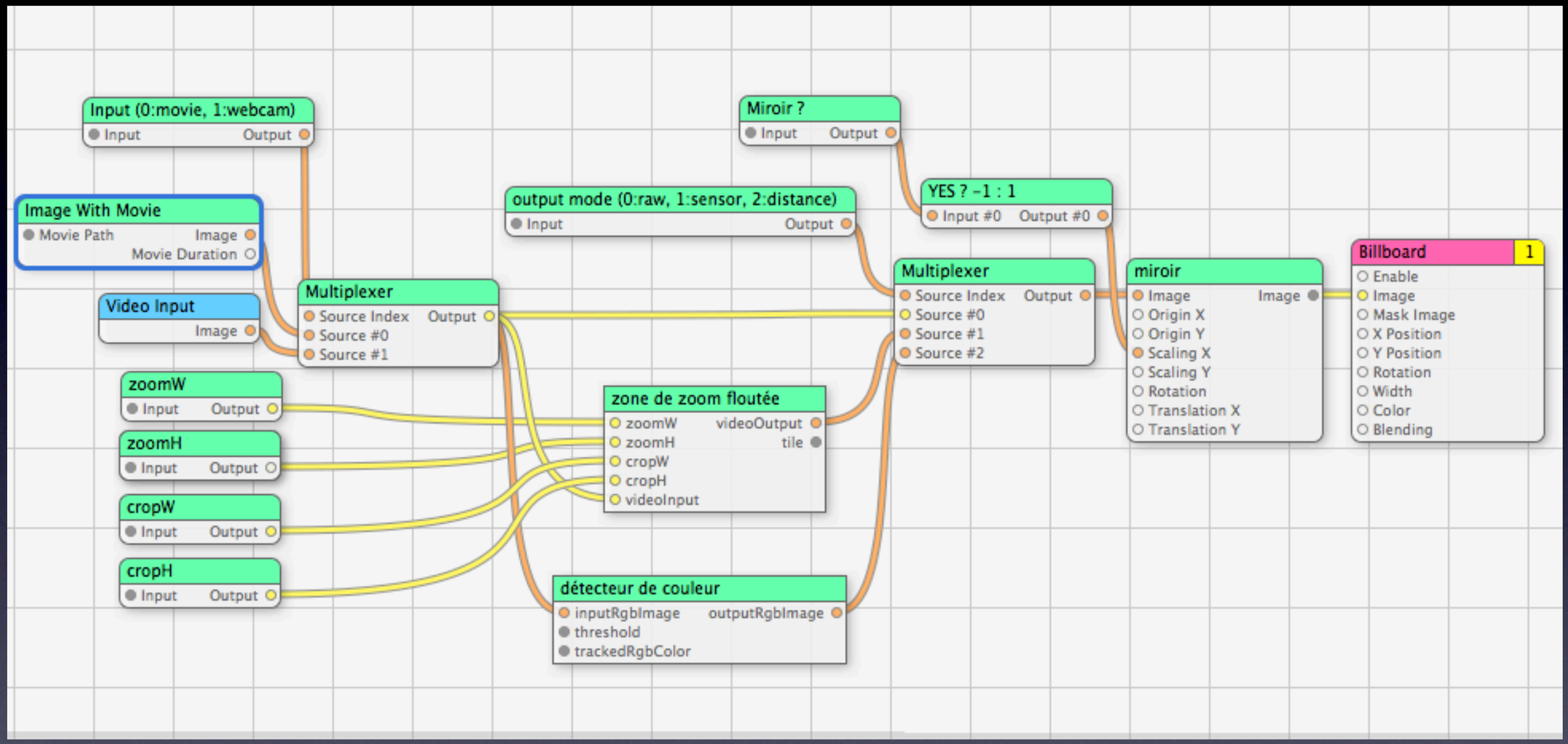
- Outil de création de « compositions »
- LabView pour CoreImage
- Tests temps réel de CIKernels
- Sauvegardable comme « boîte de traitement » à insérer dans du code
- Démonstration de Quartz Composer

Quartz Composer (2)



- LabView graphique : une boîte = une fonction
- Une composition = une boîte à insérer dans du code

Quartz Composer (2)



- LabView graphique : une boîte = une fonction
- Une composition = une boîte à insérer dans du code

Quartz Composer (3)

Exemple de code (Objective-C):

```
//charger l'image
NSString* ipath = @"/Users/chatelier/image.png";
NSURL*      url = [NSURL fileURLWithPath:ipath];
CIImage* source = [CIImage imageWithContentsOfURL:url];

//chargement de la composition
NSString* cpath = @"/Users/chatelier/composition.qtz";
NSOpenGLContext* glContext = ... //création contexte hors écran
QCRenderer* qc =
  [[QCRenderer alloc] initWithOpenGLContext:glContext pixelFormat:pixelFormat file:cpath];

//application de la composition
[qc setValue:ipath forKey:@"inputImage"];

//récupération du résultat
NSImage* result = [qc valueForOutputKey:@"outputImage"];
```

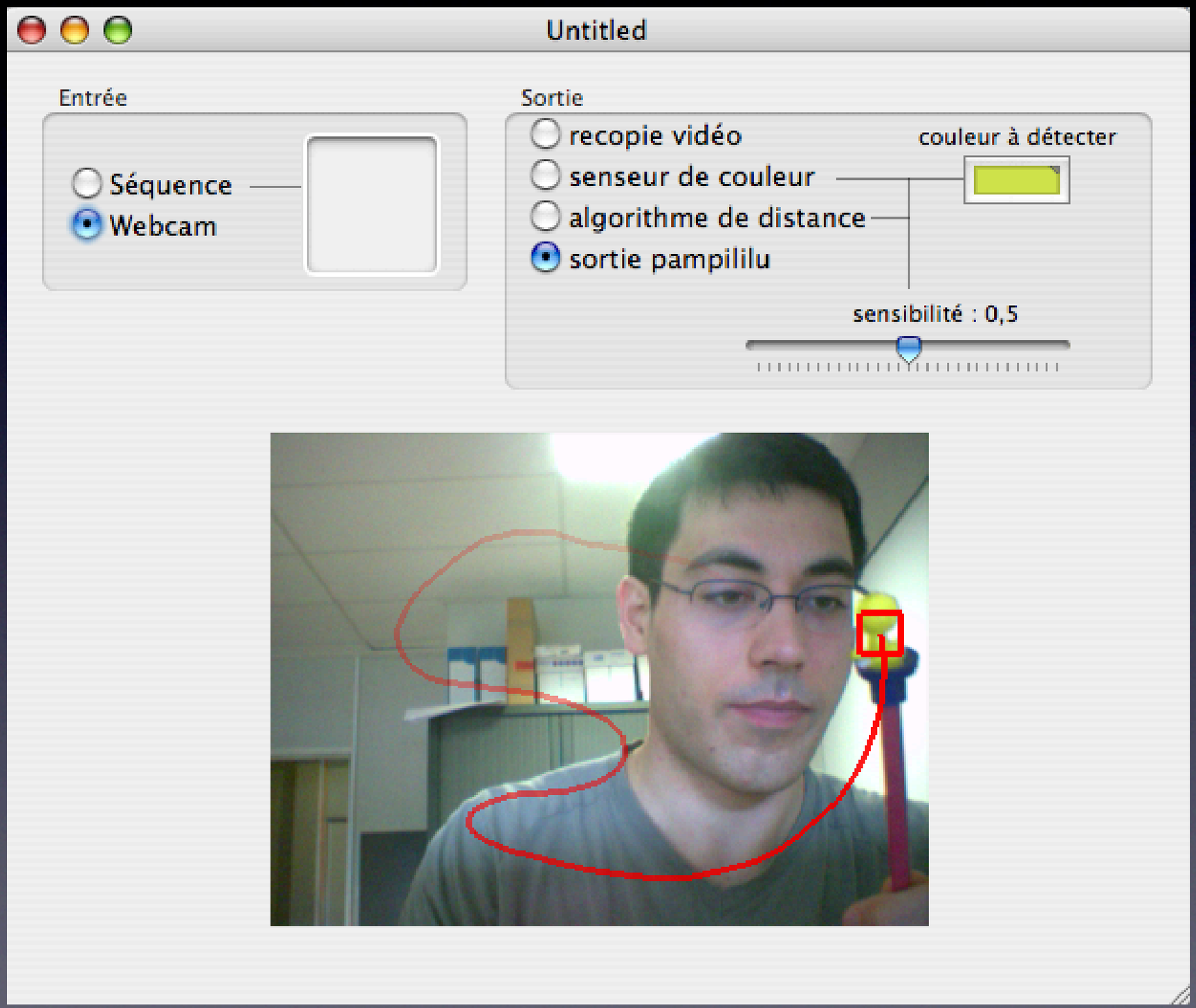
Quartz Composer (4)

- Chaînage de compositions pour insérer des algorithmes en C
- Traitement vidéo aisé à mettre en place
  - méthode *renderAtTime* d'une composition
  - appeler *valueForOutputKey* à intervalles réguliers, ou interception du PixelBuffer du contexte OpenGL

Remarques

- Actuellement, CoreImage supporte un sous-ensemble du GLSL (manque les boucles, les matrices)
- Le système de CISampler permet de gérer des déformations de l'image (plus dur en GLSL direct)
- Bilan positif toutefois : les développements deviennent accessibles

Application : Pampililu (I)



Application : Pampililu (2)

- Programme de suivi de couleur
- Section à développer avec Quartz Composer
  - Acquisition Webcam
  - Changer d'espace de couleur (RGB → Lab)
  - Appliquer une distance + filtre
- Section à développer en C
  - Détection de patates
  - Détection de chemins
- Affichage des chemins

Application : Pampililu (3)

Un shader pour RGB → Lab

- 1) Transformation linéaire RGB → XYZ (matrice*vecteur)

```
vec4 rgb2xyz(vec4 rgba)
{
  const vec4 line1 = vec4(0.49, 0.31, 0.20, 0.);
  const vec4 line2 = vec4(0.17697, 0.81240, 0.01063, 0.);
  const vec4 line3 = vec4(0.0, 0.01, 0.99, 0.);
  const float factor = 1./0.17697;
  vec4 xyz = vec4(dot(rgba, line1), dot(rgba, line2),
                  dot(rgba, line3), 0.)*factor;
  xyz.a = rgba.a;
  return xyz;
}
```

Application : Pampililu (4)

Un shader pour RGB → Lab

- 2) Transformation non linéaire XYZ → Lab

```
float f(float t)
{
  return (t > 0.008856) ? pow(t, 1./3.) : 7.787*t+16./116.;
}

vec4 rgb2lab(vec4 rgb)
{
  const vec4 xyzwhite = rgb2xyz(vec4(1., 1., 1., 1.));
  const vec4 xyz      = rgb2xyz(rgb);
  const float yyn     = xyz.g/xyzwhite.g;
  float l = ((yyn > 0.008856) ? 116.*f(yyn)-16. : 903.3*yyn);
  float a = 500.*(f(xyz.r/xyzwhite.r)-f(yyn));
  float b = 200.*(f(yyn)-f(xyz.b/xyzwhite.b));
  return vec4(l, a, b, rgb.a);
}
```

Conclusion

- Les shaders forment une technologie intéressante mais difficile à manipuler
- QuartzComposer+Core Image sont des outils d'accès très simple par rapport à ce qu'on peut en retirer
- Investissement technique faible
- Simplification du développement, et des tests, de traitements effficaces

Perspectives

- GPGPU (General Purpose on Graphical Processing Unit)
- Image = données (matrice)
- Déjà possible : Décomposition LU, résolution de systèmes linéaires...
- Kits GPGPU, par Microsoft
- La carte graphique devient un processeur supplémentaire
- Attention aux coûts de multiples transferts RAM/VRAM : favoriser un gros traitement à beaucoup de petits traitements

http://www.gpgpu.org/