Rapport PAF final : Effets Vidéo sur GPU

Description de l'architecture de cumul des effets:

1) Introduction :

Cette architecture repose sur la propagation des informations via des objets et l'échange avec la session gl. On divise les effets à implémenter sur des slices suivant la nécessité d'un passage par un framebuffer intermédiaire. (une tranche = un fragment shader + FBO d'entrée + FBO de sortie).

On a créé 3 classes d'objets principales : → effect object
→ uniform variable object
→ fragment shader info object
→ framebuffer+texture list : FBOs

On a définit 4 listes de contrôle :
→ effects_list : liste des références sur les effets object utilisés. (list of effect object)
→ slices : liste sur les limites de chaque tranche [index_first_eff, index_last_eff+1]
→ list_fs_info : liste contenant les informations nécessaire sur chaque fragment shader
→ framebuffer+texture list : FBOs : textures intermédiares

On classe les variables uniforms sous deux types :
→ effect specific uniforms : variables spécifiques à l'effet.
(exemple: kernel pour des convolutions, coefficient fixe défini à l'initialisation, …)
!!! hypothèse de travail : une fois la valeur affectée, elle reste constante sur toute la session
RQ : les listes des specific uniform variable object sont définies par effet comme attribut de l'object effect correspondant.

→ global uniforms : variable fournies par défaut par le gestionnaire des effets
(exemple: nbframe, width, height, …)
!!! hypothèse de travail : on se restreint au cas des variables uniformes constantes par frame
RQ :une liste globale stocke les global uniform variables et les object effect garde une référence par nom des varaibles à utiliser

2) Pipeline:

1. Définition des constructeurs d'objet des chaque classe représentant un type d'effet video
2. remplir *effects_list* par la liste des effets à appliquer (Attention à ne pas avoir en dernière slice un effet avec alpha!= [1,1], dans le cas contraire on obtient un effet non souhaité)
3. définir les slices selon l'attribut *require_fbo* et le nombre de textures supplémentaires dans la tranche + noter les effets demandant une texture supplémentaire. (condition : 8 textures supplémentaires max par tranche)
4. assembler le fragment shader de chaque tranche: appel de *create_fs*
5. filter.initialize : → importer des différentes textures
   → mettre à jour les informations des effets à textures supplémentaire
   → initier les buffers primordiaux
6. filter.configure_pid : → configurations par défaut
   → initier les buffers intermédiaires nécessaires
7. filter.process : la boucle générique du code
   → mettre à jour la les info des variables uniformes globales
   → récupération des paquets d'information
   → initier la texture de la vidéo
   → créer les différents shaderProgram : appel de setup Program pour chaque slice
   → affichage de la texture finale (boucle sur les framebuffers pour appliquer les effets de chaque slice : lecture du buffer d'entrée et écriture sur le buffer de sortie)

3) Descriptif des objets et des fonctions définis:

1. Uniform variable object:

Attribut / méthode	Type	Utilité
name	string	nom générique de la variable dans le glsl (peut être changer par l'assembleur)
type	string	type des valeurs stockés par la variable
dim	[int, int]	information supplémentaire sur le type. exemple: [1,1] → scalaire [1,9] → array de scalaire [2,1] → vec2 [2,9] → array de vec2
value	this.type	passée en paramètre ou changée par *update_uniforms_values*
special_info	undefined (dépend de l'utilisation)	passée en paramètre ou changée par *update_special_info*

2. Effect object:

Attribut / méthode	Type	Utilité
name	string	nom du filtre
effect_uniforms	uniform variable object	information sur toute variable uniforme spécifique à l'effet
global_uniforms_in_use_names	un array de string	Référence par nom sur les variable uniformes globales utilisées par cet effet

alpha	[float,float]	si différent de [1,1], redéfinir la taille du framebuffer en écriture de la slice commençant par cet effet rq: commencer une nouvelle tranche par conséquent
require_fbo	boolean	true si cet effet nécessite une nouvelle slice
require_texture	boolean	true si cet effet nécessite une texture supplémentaire pour support
source	string	contient le code source de l'effet : sous forme d'une fonction glsl vec4 fx(vec4, vec2);
update_source	méthode(nom_origine, préfixe)	permet d'ajouter un préfix au nom de la variable lors de l'assemblage du fs
update_uniforms_values	méthode(liste_des_indice_des_variables_à_maj, liste_des_nouvelles_valeurs	mettre à jour les valeurs de qlq variables uniformes
update_special_info	méthode(liste_des_indice_des_variables_à_maj, liste_des_nouvelles_valeurs	mettre à jour les *specefic_info* des variables unifs

Si l'effet le nécessite, on ajoute des Attribut / méthode aux objets créés

3. create_fs:

fonction qui permet d'assembler le code source d'un fragment_shader et de garder les information d'autres informations nécessaires

```
 return {
   resize : alpha,
   source : s,
   specefic_uniforms : specefic_uniforms,
   global_uniforms_in_use_names : global_uniforms_in_use_names,

 }
```

4. add_uniform:

fonction qui permet de pousser les bonnes valeurs aux variables uniformes

4) Effets implémentés:

On travaille principalement avec 3 types d'effets:

- simple_linear_transformation(name, transformation_matrix)
    - transformation_matrix: on fait le produit de vec.rgb avec cette matrice
- kernel_convolution(name, kernel, offset_size, offset):
    - kernel: le noyau
    - offset_size:
    - offset:
- change_buffer_size(name, a,b)
    - a: coefficient pour la largeur
    - b: coefficient pour la longueur

- linear_transformation_per_zones(name, transformation_matrix,w,v):
  Séparation en 4 zones de l'écran en fonction de w et v (comprises entre 0 et 1) et effectuer un filtre de couleur sur chaque partie du code.

- linear_transformation_per_zones_v(name, transformation_matrix):
  Filtre quadricolore précédent mais défilant selon le nb_frame

---

- webgl_yuv_2D_uniform.js : filtre jouant sur l'intensité des couleurs avec un coefficient variable
- webgl_yuv_filtrage.js : filtre modifiant les pixels en prenant en compte de manière pondérée les voisins ⇒ vidéo floue
- webgl_yuv_blackandwhite.js : filtre noir et blanc sur la vidéo en attribuant la valeur moyenne (rgb) aux trois couleurs
- webgl_yuv_moyenneur.js : filtre qui attribue à chaque pixel la moyenne des pixels proches voisins ==>lisse les discontinuités mais vidéo floue.
- webgl_yuv_linear.js : filtre vert sur la vidéo
- webgl_yuv_linear4.js : filtre quadricolore (rouge, vert, bleu et n&b) sur ¼ de la vidéo
- webgl_yuv_linear4_nbframe.js : filtre quadricolore défilant
- webgl_yuv_nettete.js : filtre qui améliore la netteté de la vidéo grâce à une matrice de convolution
- webgl_yuv_prewitt_X.js : filtre qui détermine les contours des formes de l'images grâce à la matrice de prewitt
- webgl_yuv_sobel_X.js : filtre qui détermine également les contours grâce à la matrice de sobel
- webgl_yuv_laplacien.js : détermine encore une fois les contours mais de manière plus détaillée grâce au laplacien (second ordre)
- webgl_yuv_texture_effect.js : applique un masque téléchargé de textures/ sur la video
- webgl_yuv_framebuffer.js : applique un effet blur par changement de la taille du framebuffer

Rq. : Ces filtres ont été généralisés dans des classes d'objets. Les classes en question ont été déclarées sous forme de constructeurs d'objet dans la structure générique. Il suffit d'instancier le bon objet avec les bons paramètres pour insérer l'effet dans l'enchaînement.

5) Description du fonctionnement:

La session est représentée par un objet nommé gl qui gère les processus par des méthodes bien déterminées:

- un vertexshader: Les vertex shaders calculent la projection des sommets des primitives à partir de l'espace 3D dans l'espace d'écran en 2D, ils s'exécutent une fois pour chaque sommets.

- un fragmentshader: est un mini-programme qui reçoit la variable varying vec2 vTextureCoord de la part de vertexshader pour localiser le pixel puis il exécute des fonctions sur la couleur de pixel et finalement il l'affiche.

- une variable uniform: c'est la forme sous laquelle on passe des variables de l'extérieur de script
-
- un framebuffer intermédiaire: On a utilisé un framebuffer intermédiaire principalement pour 2 utilités:
    1) Faire un resize sur la taille du buffer.
    2) passer d'un filtre_1 à un filtre_2 dans le cas où le filtre 2 dépend du résultat du filtre_1.

6) Les erreurs typiques à éviter:

Dans le code GLSL:

-Au cas où on veut utiliser un float dans le code GLSL il ne faut absolument pas oublier de le mettre sous cette forme 5.0 vu que ça renvoie un conflit avec les int si on met 5.

-On ne peut pas déclarer des const arrays dans le code GLSL il faut les passer comme uniform.

-dans le code GLSL, il faut se rappeler que la vTextureCoord représente des coordonnées normalisées comprises dans le carré ([0,0],[0,1],[1,1],[1,0]).

-vérifier la cohérence d'affectation des variables (il ne faut pas mettre par exemple un vec3 dans un vec4).

-Quand on manipule la couleur d'un pixel, il faut toujours modifier les 3 premiers coordonnées du vecteur (sauf si on veut aussi modifier la transparence).

-Une différence entre code GLSL entre MAC et LINUX:

Déclaration d'un array uniform:

→ LINUX: uniform float[9] name_array;

→ MAC: uniform float name_array[9];

Dans le code JavaScript:

-Quand on veut passer une variable uniforme au fragment shader, il ne faut oublier de modifier le code sur 3 niveau:

- Déclarer la variable dans le fragment shader "uniform [type] u_uniform".
- L'ajouter dans les uniformLocation au niveau de la fonction setupPorgram: uniform: gl.getUniformLocation(shaderProgram, 'u_uniform').
- Le passer au fragment shader dans la fonction drawscene en tenant compte de son type:
  - float: gl.uniform1f(location,uniform_value)
  - int: gl.uniform1i((location,uniform_variable.value)
  - vec2: gl.uniform2fv(location,uniform_variable.value)
  - array de float: gl.uniform1fv(location,uniform_variable.value)

→ Un détail important: quand on veut passer une matrice de vec2 comme une variable uniform (dans le cas d'un filtre basé sur les convolutions par exemple) il faut la déclarer en avance comme une simple liste en mettant les couples de coordonnées en série (sans faire de crochets). puis dans la partie add_uniform du code on met gl.uniform2fv(location,liste).

-Il ne faut pas faire un bindTexture sans faire avant un activeTexture avec le bon indice.

7) Répartition des tâches

Au début (tout le monde):
-familiarisation avec les langages
-compréhension du code
-premiers effets simples

Sur les derniers jours :
-Yassine et Rached se sont chargés de construire la structure générale permettant d'utiliser des objets afin de faciliter l'input des nouveaux filtres. (automatiser le maximum la procédure)
-Zakary et Caroline se sont chargé d'implémenter différents effets pour les tests à la fin

Â la fin (tout le monde):
l'intégration des filtres dans la structure et les tests.