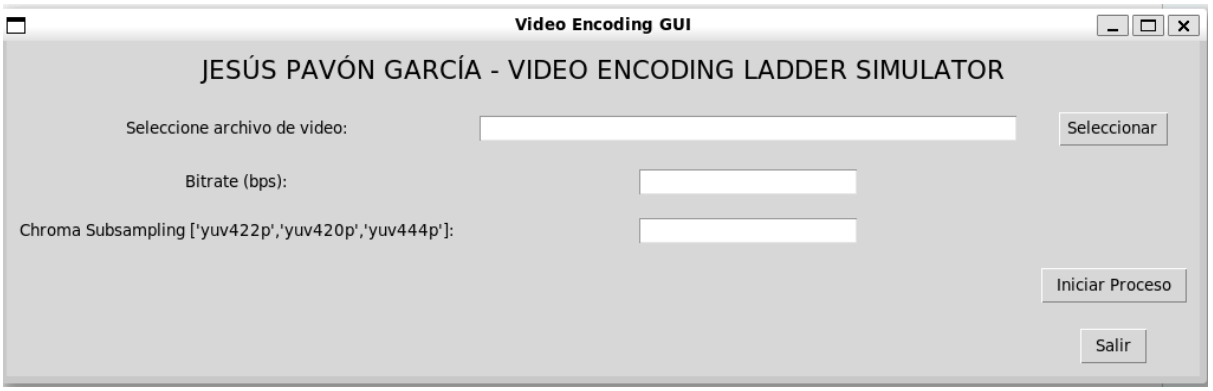
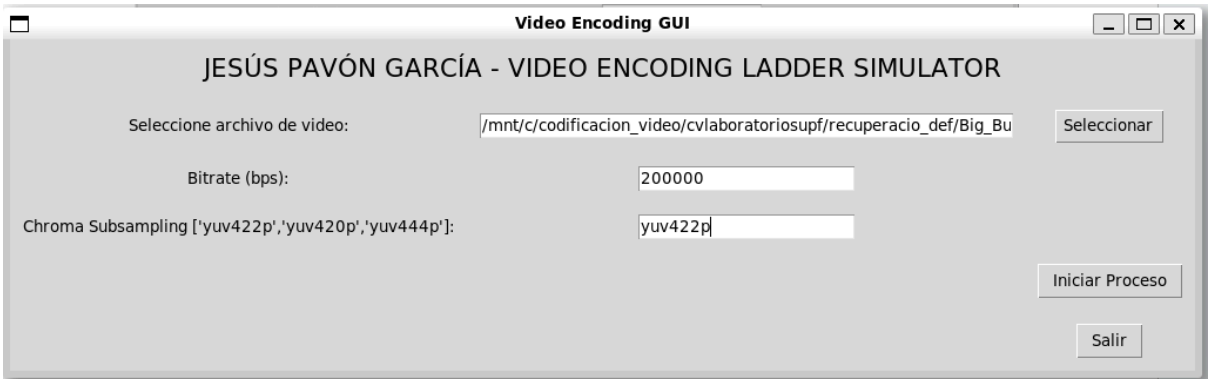
Jesús Pavón García 206933
UPF Codificació Audio i Video

SIMULADOR DE VIDEO ENCODING LADDER MITJANÇANT API

El següent programa serveix per codificar videos variant la resolució i el chroma subsampling, aconseguint entregar com a resultat un vídeo amb menor tasa de bitrate segons l'ample de banda disponible.

El funcionament del programa consisteix en:
1- Seleccionar l'arxiu de vídeo de la nostre carpeta local.
2- Introduir l'ample de banda disponible (Simulant el nostre propi ample de banda).
3- Introduir el chroma subsampling amb què volem que codifiqui el video.
4- Iniciar el procediment.





Un cop el procés és iniciat desde la Gui (*'gui2.py'*), automàticament les dades introduïdes son enviades al servidor Flask fent el següent post

```
response = requests.post(url, files=files, data={'bitrate': bitrate, 'chroma_sub': chroma_sub})
```

Ara la Api (*'app.py'*) reclama les dades enviades en la url: *'http://127.0.0.1:5002/process_video'*

```
file = request.files['video']
```

```
usr_bitrate = request.form.get('bitrate')
usr_chroma_subsampling = request.form.get('chroma_sub')
```

I comença a processar el video input amb funcions creades en (*'lab_recu.py'*).

Pseudocodi i funcionament de la Api és el següent:

1- En cas de tenir *new_bitrate*:
   - Genera un arxiu de text i copia totes les dades del video sense modificar (*'video_info_original.txt')*
   - Executa la funció de video encoding *'encoding_ladder()'*

2- En cas de tenir *chroma_subsampling*:
   - El input_video és el output_video del pas 1).
   - Executa la funció que cambia el chroma subsampling *'change_chroma_subsampling()'*
   - Genera un arxiu de text i copia totes les dades del video processat (*'video_info_temp.txt')*

3- Els arxius de text i el video processat del pas 2) s'escriuen dins d'una carpeta Zip

4- Eliminem els videos temporals processats.

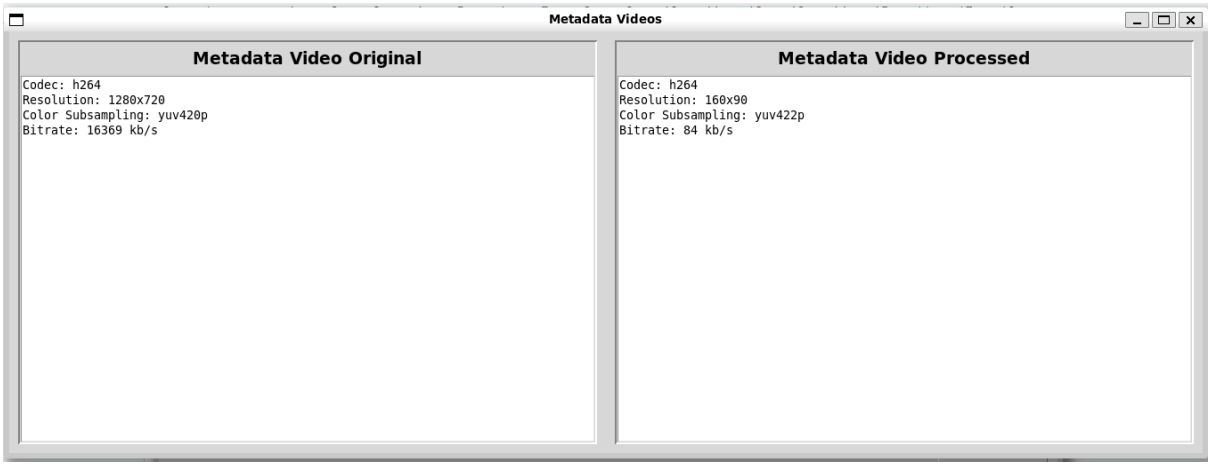
5- Enviem la carpeta Zip al servidor Flask a la url 'http://127.0.0.1:5002/process_video'

Un cop tenim el arxiu final processat, desde la Gui *('gui2.py'*) reclama la carpeta Zip al servidor Flask

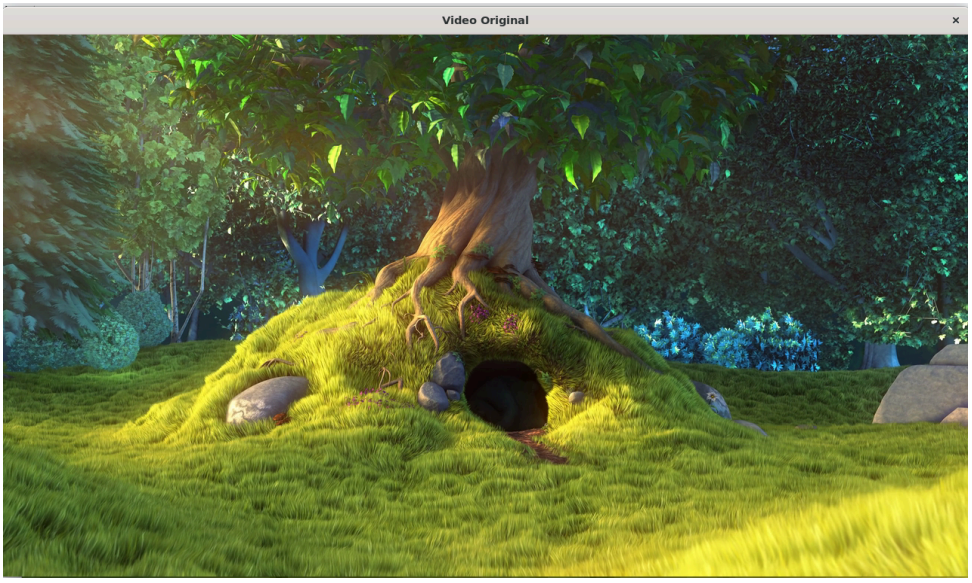
```
if response.status_code == 200:
    zip_path = os.path.join('/tmp', 'processed_video.zip')
```

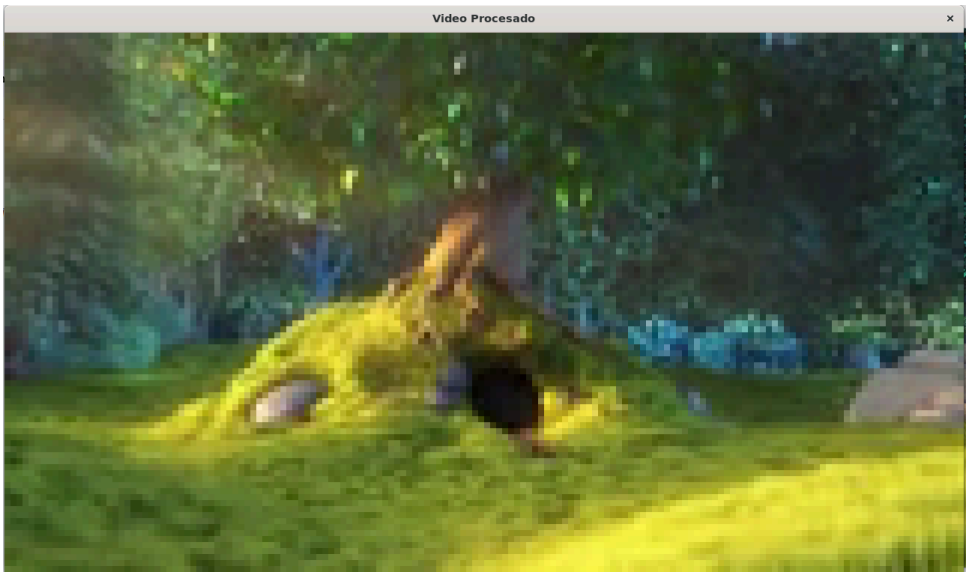
Descomprimeix els arxius de la carpeta Zip i s'encarrega de fer display:
   - Videos (Original i Processat)

Un cop els vídeos han estat reproduïts fa display de:
   - Metadades (Original i Processat)



Es pot observar que al reduir la resolució i canviar el chroma subsampling , el bandwidth necessari per executar el vídeo s'ha vist reduït, podent veure fluidament el video processat a costa de perdre resolució i detall respecte al original.





El resultat és un vídeo més pixelat que coincideix amb les dades obtingudes.

Un cop hem vist el funcionament , entraré en detall sobre parts específiques del programa:

- Video Encoding

(‘*encoding_ladder()’* ubicada a ‘*lab_recu.py’)*

És la funció encarregada d’escollir quina és la resolució més òptima segons el bandwidth proporcionat. Primer demana amb *get_video_info(),* el bitrate necessari per reproduir el vídeo input i el compara amb el bitrate introduit de l'usuari (threshold_bitrate).

si bitrate < threshold_bitrate:

Mantindrá la resolució original, ja que no és problema que nosaltres tinguem un ample de banda major que el que es necessita per reproduir el vídeo

si bitrate > threshold_bitrate

Redueix de forma lineal la resolució ,en funció a la diferència de bitrate , entre bitrate i threshold_bitrate (reduint en /2 , /4, /8).

Un cop tenim els valors de resolució per el nou video el codifiquem mitjantçant *encoding_video()* que crida a la comanda ffmpeg amb les noves dades

- Chroma subsampling

(‘change_chroma_subsampling*()’* ubicada a ‘*lab_recu.py’)*
És la funció encarregada de cambiar el chroma subsampling.
Crea la comanda ffmpeg que es nutreix del path video input , el chroma subsampling desitjat i el path del video output.

- Escriptura d’arxius

(‘parse_video_info()’ *i ‘get_video_info()’ ubicada a ‘lab_recu.py’*)
Per a l’escriptura de dades en un fitxer de text són necessaris dos processos.
En primer lloc cridar a *parse_video_info()* que rep com a input el path de vídeo i el path de l’arxiu de text.
Aquest mitjançant una comanda ffmpeg ens mostrarà l’informació del video en el terminal.
Per altre banda obté els valors de resolució, codec i bitrate cridant a *get_video_info().*
(No és necessari aquest segon pas, però és útil, per identificar ràpidament les dades rellevants al fitxer)
El recull complet d’informació s’escriu amb *file.write()* en el path que hem indicat previament.

- Lectura de dades

Les dades les rebem en un .zip de la següent forma:

Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..			Carpeta de archivos		
processed_video....	108.132	108.132	Archivo MP4	28/06/2024 14:...	B8A98749
video_info_origi...	2.615	2.615	Documento de texto	28/06/2024 14:...	6B55FE39
video_info_temp....	2.596	2.596	Documento de texto	28/06/2024 14:...	2F07EE6C

És important que quan s’ha fet l’extracció correctament , identificar els tipus d’arxiu

```
video_path = [file for file in zip_ref.namelist() if file.endswith('.mp4')][0]
metadata_txt_path_ori = [file for file in zip_ref.namelist() if file.endswith('_original.txt')][0]
metadata_txt_path = [file for file in zip_ref.namelist() if file.endswith('_temp.txt')][0]
```

Un cop tenim cada tipus de file identificat es possible processarho de diferent forma.
Per els arixus de text: *read_metadata()* i després *display_metadata()*
Per els arxius de video: *play_videos()*

- Processament dades

Quan tota l'informació ha estat identificada en gui2.py.
La funció *process_video()* és l'encarregada de contactar amb el servidor, rebre les dades, identificarles i per últim processarles. Com anteriorment comentem, es crida a *read_metadata() ,display_metadata() i play_videos()*
read_metadata(): Agafant l'arxiu de text que li diguem identifica la linea en concret:

```
stream_pattern = re.compile(
    r'Stream #\d+:\d+\(\w*\): Video: (\w+) \(.*\), (\w+), (\d+x\d+) \[.*\], (\d+) kb/s,.*')
```

ho fa amb una comparació directe linea a linea , fins a identificar la els characters identics.
Un cop fa cada match retorna esl valors de codec, colorsubsampling, resolution, bitrate.
display_metadata(): Amb els valors que retorna read_metadata() fa un display directe en una finestra.

És important que aquestes 2 funcions ho fa 2 cops un per el video original i una segona vegada per el video processat que els dos arxius de text provenen del .zip.
El display és fa en una mateixa finestra però subdividint en frames.

play_videos(): És una versió de *play_video()* però está pensada per reproduir simultaniament 2 videos en una mateixa execució. Ho fa cridant a openCV.Videocapture() en dos ocasions i genera dos finestres separades. Com a input hem d'introduir el path del video original i video_processat.

Existeix un ERROR en el programa segons el video que reproduim:

```
.
.
An error occurred: cannot unpack non-iterable NoneType object
```

Això es deu a que el reproductor no és capaç de reproduir a una resolució inferior ja que és massa petita i dispara l'error.
Si ens fixem en el .zip observarem que les dades s'han copiat i desat correctament.

processed_video....	35.174	35.174	Archivo MP4	28/06/2024 18:...	92562E0E
video_info_origi...	2.313	2.313	Documento de texto	28/06/2024 18:...	05FA7160
video_info_temp....	2.302	2.302	Documento de texto	28/06/2024 18:...	53EEF171

```
yuv422p, 80x60 [SAR 1:1 DAR 4:3], 36 kb/s,
```