Master IMLEX/COSI

2-Day Sprint

Real-Time 3D-XR Visualization

Overview

This document outlines the tasks for a 2-day sprint (14 hours total). Based on the provided code base, the objective is to implement real-time video processing, anaglyph generation, and a WebXR stereoscopic viewer.

Task 1: Image Processing Methods

Implement the following color image processing methods on videos in real-time (see the code base for more details). Each method must be implemented as a **fragment shader** and applied to all three RGB channels. Parameters should be exposed in the GUI via `lil-gui`.

1 - Convolution — Gaussian Filter

Principle. A 2D Gaussian blur smooths an image by replacing each pixel with a weighted average of its neighbours. The weights follow a 2D Gaussian distribution:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

where (x, y) is the offset from the centre pixel and σ controls the spread of the blur.

- **Kernel shape:** Square, size $k \times k$ where $k = 2r + 1$ (odd), r = radius.
- **Parameters:** `kernelSize` (odd integer, e.g. 3, 5, 7...) and `sigma` ($\sigma > 0$).
- **Weight computation:** Compute all k^2 weights from the Gaussian formula and **normalise** so they sum to 1.
- **Pixel access:** Read pixel values at exact integer coordinates from the source image (e.g. `getPixel(image, x, y)`).
- **Channels:** Apply identically to R, G, B (same weight matrix for all channels).
- **Border handling:** Clamp to edge, or skip out-of-bounds samples and renormalise.

Pseudocode:

```
// For each output pixel at position (px, py):
set sum_rgb = (0, 0, 0)
set weightSum = 0
for dy from -radius to +radius do
  for dx from -radius to +radius do
    w = exp(-(dx*dx + dy*dy) / (2 * sigma * sigma))
    pixel = getPixel(image, px + dx, py + dy)
    sum_rgb = sum_rgb + w * pixel.rgb
    weightSum = weightSum + w
  end for
end for
output = sum_rgb / weightSum
```

Complexity: $O(k^2)$ texture fetches per pixel, becomes expensive for large kernels (motivates the separable filter below).

2 - Convolution — Laplacian Filter

Principle. The Laplacian is a second-order derivative operator that highlights edges and rapid intensity changes. It approximates:

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Common 3×3 kernels:

4-connected

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

8-connected

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

- **Kernel:** Fixed 3×3 Laplacian (choose 4-connected or 8-connected, or offer both).
- **Output mode — Color:** Apply the kernel independently to R, G, B channels. The output is a signed color vector; map to $[0, 1]$ using `abs()` or $0.5 + 0.5 \times$ result for visualisation.
- **Output mode — Norm:** Compute the Laplacian per channel, then output the Euclidean norm of the resulting 3-component vector:

  $$\sqrt{L_R^2 + L_G^2 + L_B^2}$$

  displayed as a greyscale value.
- **Parameter:** A boolean or enum to toggle between *Color* and *Norm* display modes.

Pseudocode (8-connected):

```
// Fixed 3x3 kernel weights (8-connected Laplacian)
K = [1, 1, 1,  1, -8, 1,  1, 1, 1]

set laplacian_rgb = (0, 0, 0)
set idx = 0
for dy from -1 to +1 do
  for dx from -1 to +1 do
```

```
      pixel = getPixel(image, px + dx, py + dy)
      laplacian_rgb = laplacian_rgb + K[idx] * pixel.rgb
      idx = idx + 1
    end for
  end for
  if displayNorm then
    norm = sqrt(laplacian_r^2 + laplacian_g^2 + laplacian_b^2)
    output = (norm, norm, norm)
  else
    output = (abs(laplacian_r), abs(laplacian_g), abs(laplacian_b)
        )
  end if
```

Note: The Laplacian output can contain negative values; an appropriate remapping strategy is needed for display. Students may optionally combine Gaussian + Laplacian (Laplacian of Gaussian / LoG) for a more robust edge detector.

3 - Separable Filter — Gaussian

Principle. A 2D Gaussian kernel is **separable**: it can be decomposed into two successive 1D convolutions (horizontal then vertical). This reduces the complexity from $O(k^2)$ to $O(2k)$ per pixel.

$$G_{2D}(x, y) = G_{1D}(x) \cdot G_{1D}(y) \quad \text{where} \quad G_{1D}(t) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{t^2}{2\sigma^2}}$$

- **Two-pass pipeline:**
  (a) **Pass 1:** Horizontal 1D Gaussian → write to an intermediate render target.
  (b) **Pass 2:** Vertical 1D Gaussian reading from that intermediate target → write to final output.
- **Parameters:** Same as Method 1: `kernelSize` and `sigma`.
- **Architecture:** Requires **two render passes** (or one pass used twice with swapped render targets) and **two shaders** (or one shader with a `horizontal` boolean parameter to toggle direction).
- **Weight computation:** 1D Gaussian weights, normalised to sum to 1.

Pseudocode (single-direction pass):

```
input: horizontal  // true = H pass, false = V pass

set sum_rgb = (0, 0, 0)
set weightSum = 0
for i from -radius to +radius do
  w = exp(-(i * i) / (2 * sigma * sigma))
  if horizontal then
    offset = (i, 0)
  else
    offset = (0, i)
  end if
  pixel = getPixel(image, px + offset.x, py + offset.y)
  sum_rgb = sum_rgb + w * pixel.rgb
  weightSum = weightSum + w
end for
output = sum_rgb / weightSum
```

Key implementation detail: The intermediate render target from Pass 1 must be used as the input image for Pass 2. Students should verify that the result is **visually identical** to Method 1 for the same parameters, while being significantly faster for large kernel sizes.

4 - Denoising — Median Filtering

Principle. A median filter replaces each pixel value with the **median** of pixel values in its neighbourhood. Unlike linear filters (Gaussian, box), the median filter is **non-linear** and excels at removing **salt-and-pepper noise** while preserving edges.

- **Kernel shape:** Square $k \times k$ neighbourhood.
- **Parameter:** `kernelSize` (odd integer, typically 3 or 5 — larger sizes are very expensive in a shader).
- **Per-channel:** Compute the median **independently** for R, G, B.
- **Sorting:** Students must implement a **sorting network** or **insertion sort** on a fixed-size array (no built-in sort is available in shader languages).

Implementation challenge: Median filtering in a shader is significantly harder than convolution because:

(a) **No dynamic arrays** — shader languages typically require compile-time-known array sizes (9 elements for 3×3, 25 for 5×5).

(b) **Sorting** — A sorting network (e.g., Bose–Nelson or bitonic) is the most GPU-friendly approach. For 9 elements, an optimal sorting network uses only 25 comparisons.

Pseudocode (3×3 example):

```
// Collect 9 neighbourhood values for one channel
let values_r[9], values_g[9], values_b[9]
set idx = 0
for dy from -1 to +1 do
  for dx from -1 to +1 do
    pixel = getPixel(image, px + dx, py + dy)
    values_r[idx] = pixel.r
    values_g[idx] = pixel.g
    values_b[idx] = pixel.b
    idx = idx + 1
  end for
end for
// Sort each channel array (sorting network or insertion sort)
sort(values_r)
sort(values_g)
sort(values_b)
// Median is the middle element (index 4 of 9)
output = (values_r[4], values_g[4], values_b[4])
```

Tip: For a 3×3 kernel, a partial sorting network that only finds the median without fully sorting is more efficient. For a 5×5 kernel (25 values), performance will drop significantly.

Summary:

#	Method	Type	Parameters	Passes
1	Gaussian convolution	Linear, 2D	`kernelSize`, σ	1
2	Laplacian convolution	Linear, 2D	Display mode	1
3	Separable Gaussian	Linear, 1D×2	`kernelSize`, σ	2
4	Median denoising	Non-linear	`kernelSize`	1

Task 2: Color Filtered Anaglyph

Background

An anaglyph image is created by combining a stereo pair (left and right images) into a single image where the two views are encoded in different color channels. When viewed through color-filtered glasses (typically red for the left eye and cyan for the right eye), each eye sees only its corresponding image, producing a perception of depth.

All anaglyph methods can be expressed as a linear combination of the left and right image pixels:

$$\begin{pmatrix} R_a \\ G_a \\ B_a \end{pmatrix} = M_L \begin{pmatrix} R_l \\ G_l \\ B_l \end{pmatrix} + M_R \begin{pmatrix} R_r \\ G_r \\ B_r \end{pmatrix}$$

where M_L and M_R are 3×3 matrices applied to the left and right source images respectively, and (R_a, G_a, B_a) is the resulting anaglyph pixel.

Objective: Implement the following anaglyph methods as fragment shaders (one shader with a selectable method via a parameter). The user should be able to switch between methods in real-time using the GUI. The methods below are based on the comparison presented at: https://3dtv.at/Knowhow/AnaglyphComparison_en.aspx

Method Descriptions

1 - True Anaglyph (Monochrome)

Both images are converted to greyscale using the ITU-R BT.601 luminance weights before being assigned to their respective channels. This eliminates color-based retinal rivalry but produces a dark, monochrome result.

$$M_L = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_R = \begin{pmatrix} 0 & 0 & 0 \\ 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \end{pmatrix}$$

+ Little ghosting, no retinal rivalry from color
− Dark image, no color reproduction

2 - Gray Anaglyph

Similar to the True method, but uses the full luminance for both the red and cyan channels, producing a brighter image at the cost of slightly more ghosting.

$$M_L = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_R = \begin{pmatrix} 0 & 0 & 0 \\ 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \end{pmatrix}$$

Note: The matrices are the same as True Anaglyph. In some implementations, the Gray method differs by **not** converting the right image to greyscale for the cyan channels — instead using the original green and blue channels directly. An alternative formulation is:

$$M_L = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

+ Brighter than True Anaglyph
− More ghosting, still no color reproduction

3 - Color Anaglyph

The simplest method: directly maps the red channel of the left image and the green/blue channels of the right image. Preserves the most color but causes severe **retinal rivalry** (visual discomfort from conflicting color stimuli between eyes).

$$M_L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

+ Partial color reproduction, simple to implement
− Severe retinal rivalry, color tint

4 - Half-Color Anaglyph

A compromise: the left image is converted to greyscale for the red channel (reducing rivalry), while the right image keeps its original green and blue channels (preserving some color).

$$M_L = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

+ Less retinal rivalry than Color Anaglyph
− Slightly less color reproduction than Color Anaglyph

5 - Optimized Anaglyph (Dubois)

The Dubois method uses a **least-squares projection** in perceptual color space, accounting for the spectral characteristics of the display and the red/cyan glasses. This produces the best balance between color fidelity, depth perception, and minimal retinal rivalry. A gamma correction (e.g., $\gamma = 1.5$) is typically applied to brighten the red channel.

$$M_L = \begin{pmatrix} 0.437 & 0.449 & 0.164 \\ -0.062 & -0.062 & -0.024 \\ -0.048 & -0.050 & -0.017 \end{pmatrix} \quad M_R = \begin{pmatrix} -0.011 & -0.032 & -0.007 \\ 0.377 & 0.761 & 0.009 \\ -0.026 & -0.093 & 1.234 \end{pmatrix}$$

Note: These are the standard Dubois coefficients for red/cyan glasses. Students should clamp the output to $[0, 1]$ after applying the matrices.

+ Best color reproduction, almost no retinal rivalry
− Cannot reproduce red shades, more complex to implement

Implementation

All five methods share the same structure and can be implemented in a single shader with an integer parameter to select the method:

```
input: leftImage, rightImage    // Left and right eye images
input: anaglyphMethod           // 0=True, 1=Gray, 2=Color, 3=Half,
    4=Dubois

// Luminance conversion (ITU-R BT.601)
function luminance(c):
  return 0.299 * c.r + 0.587 * c.g + 0.114 * c.b

// For each output pixel at position (px, py):
L = getPixel(leftImage, px, py).rgb
R = getPixel(rightImage, px, py).rgb

if anaglyphMethod == 0 then            // True Anaglyph
  lL = luminance(L)
  lR = luminance(R)
  result = (lL, lR, lR)
else if anaglyphMethod == 1 then       // Gray Anaglyph
  lL = luminance(L)
  result = (lL, R.g, R.b)
else if anaglyphMethod == 2 then       // Color Anaglyph
  result = (L.r, R.g, R.b)
else if anaglyphMethod == 3 then       // Half-Color Anaglyph
  lL = luminance(L)
  result = (lL, R.g, R.b)
else                                   // Dubois Optimized
  ML = [[0.437, 0.449, 0.164],
        [-0.062, -0.062, -0.024],
        [-0.048, -0.050, -0.017]]
  MR = [[-0.011, -0.032, -0.007],
        [0.377, 0.761, 0.009],
        [-0.026, -0.093, 1.234]]
  result = clamp(ML * L + MR * R, 0, 1)
end if
output = result
```

Note: When translating to your target shading language, pay attention to the matrix storage convention (row-major vs. column-major) and adjust accordingly.

Summary

#	Method	Color	Rivalry	Ghosting
1	True Anaglyph	None	None	Low
2	Gray Anaglyph	None	None	Medium
3	Color Anaglyph	Good	Severe	Low
4	Half-Color	Moderate	Low	Low
5	Dubois Optimized	Best	Minimal	Low

Integration with Task 1: The image processing methods from Task 1 should be applied to each stereo half *before* the anaglyph combination. The processing pipeline is:

Stereo frame → Split into L/R → **Task 1: Processing** → **Task 2: Anaglyph** → Display

Reference: https://3dtv.at/Knowhow/AnaglyphComparison_en.aspx

Task 3: Implementation of a Stereo Viewer in WebXR

Implement a WebXR application that displays a stereo view of the video.

Objective: Display the video in a stereo view, with the left eye and right eye views being displayed separately. The user should be able to view the video in stereo by using anaglyph glasses or by using a VR headset.