A3: Image Warping

CS4365 Applied Image Processing
Assignment 3
2023/2024

TU Delft

Petr Kellnhofer

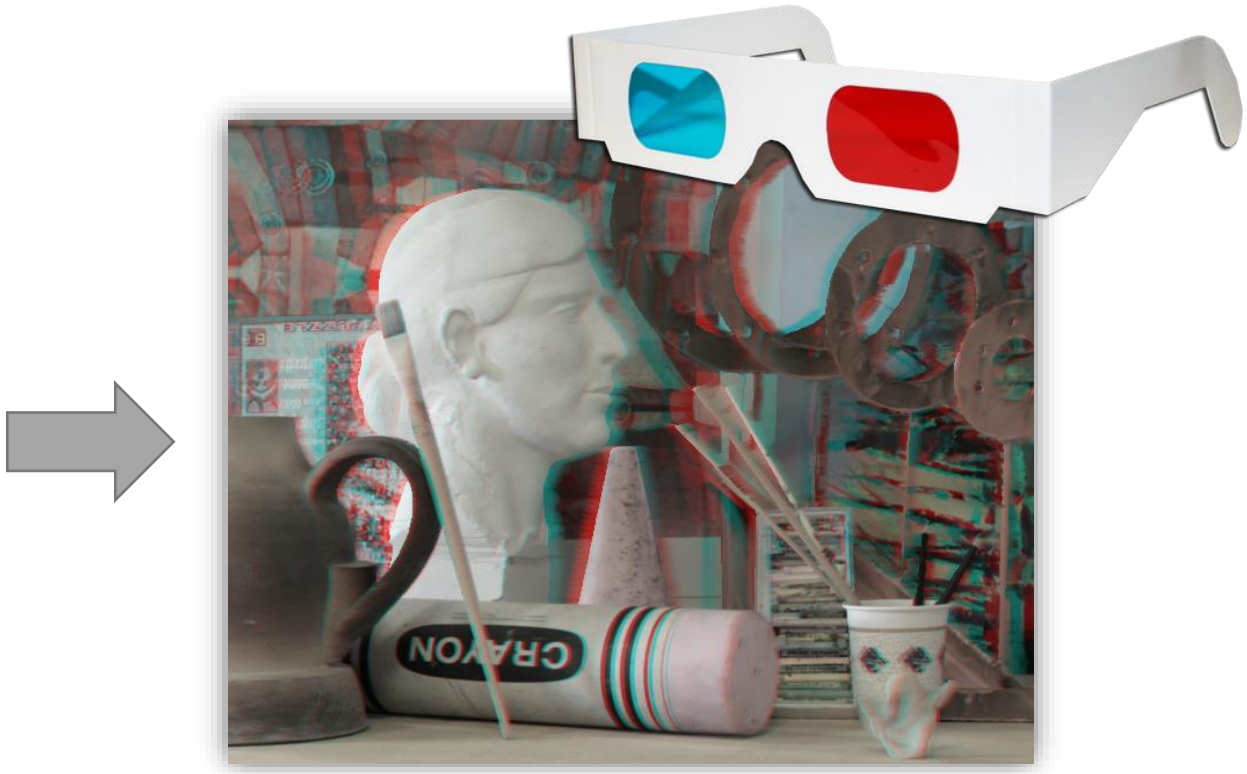
COMPUTER GRAPHICS AND VISUALIZATION CGV

Goal

- Convert an RGB image with depth information into a **stereoscopic anaglyph image** using **mesh-based warping**.
  - See **Lecture 6** for more info.

RGB

Depth



Anaglyph stereo

What we DO implement

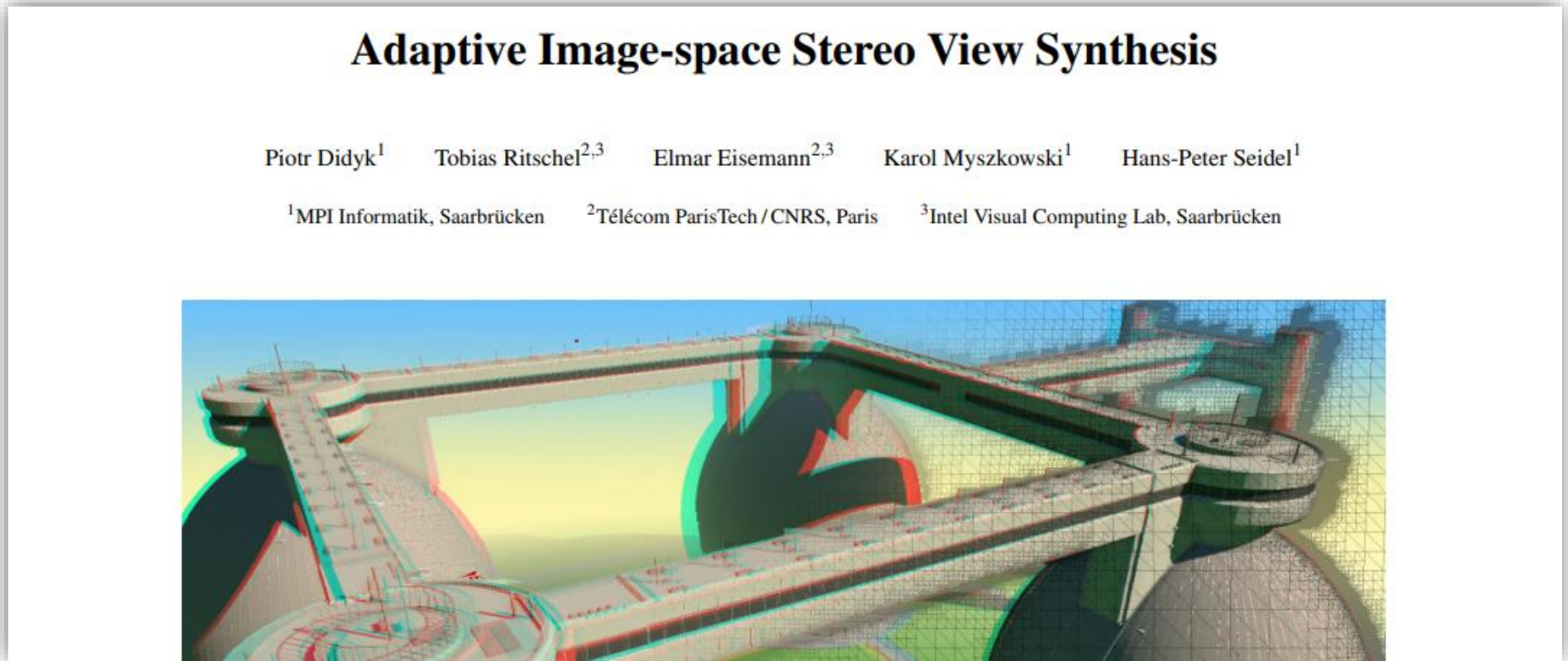
- **Bilateral filtering** of depth artifacts.
- Compute stereoscopic **disparity** from depth.
- A simple **forward warping** to get left and right stereo pair.
- Generate **anaglyph** image.

- Build a **warping grid**.
- **Warp the grid** using disparity.
- Use **mesh-based warping** to get left and right stereo pair.

- **Rotate** an image.

Very loosely inspired by

- Didyk, Piotr, et al.:
  "**Adaptive Image-space Stereo View Synthesis**." VMV 2010.
  - Link: https://resources.mpi-inf.mpg.de/AdaptiveStereoViewSynthesis/AdaptiveStereoViewSynthesis.pdf



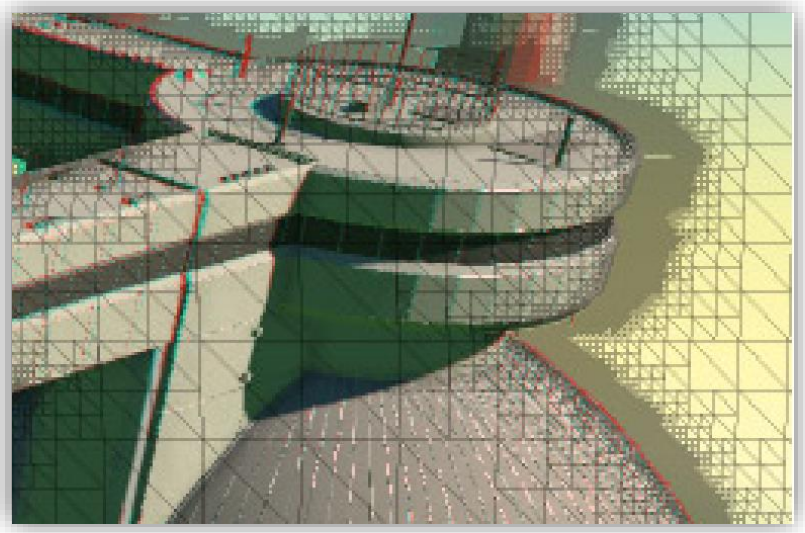
Adaptive Image-space Stereo View Synthesis

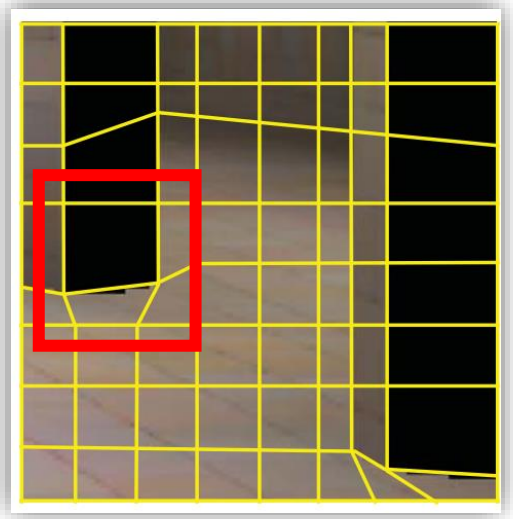
Piotr Didyk[1] Tobias Ritschel[2,3] Elmar Eisemann[2,3] Karol Myszkowski[1] Hans-Peter Seidel[1]

[1]MPI Informatik, Saarbrücken [2]Télécom ParisTech / CNRS, Paris [3]Intel Visual Computing Lab, Saarbrücken

What we DO NOT implement

- Content-**adaptive grid resolution**.
  - Our grid has a fixed very high resolution.
  - This has impact on **performance**.
- Content-aware grid “**snapping**”.
  - Our grid always aligns with pixel borders.
  - This increases **artifacts** for large disocclusions.





I) Input preprocessing

Input RGB

Remove holes using a guided bilateral filter

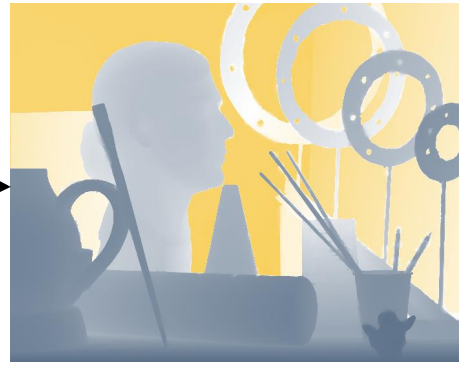
Compute normalized depth

Compute pixel disparities for comfortable stereo

Input disparity

Clean disparity

Normalized depth



Clean disparity

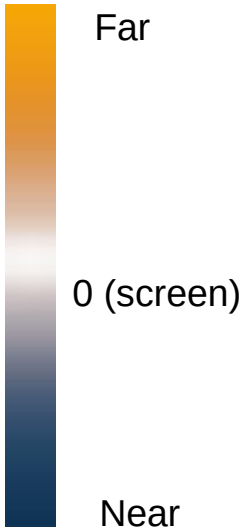
0. RGBD input image pair

Get more data:
https://vision.middlebury.edu/stereo/data/

- RGB and Depth loaded separately from 2 PNG files (code provided).
- Disparity saved with a tone mapper for negative/positive values.

Input RGB

Input disparity




```
auto image = ImageRGB(dataDirPath / "half/view1.png");
auto src_disparity = loadDisparity(dataDirPath / "half/disp1.png");
```

1. Filter disparity

Refer to the text-book for general info about Bilateral filtering:
Computer Vision: Algorithms and Applications, 2nd edition,
© 2022 Richard Szeliski: Sec. 3.3.2 Bilateral filtering

- Disparity map contains **holes** – we should remove them.
  - Invalid disparity pixels == `INVALID_VALUE`.

- Write a **joint bilateral filter** for the disparity D_0 using the image I as guide signal.
  - i.e., **relative** contribution of a neighboring pixel $\boldsymbol{x}_i$ is:

Normalize it!

$$w_i = gauss(\|\boldsymbol{x}_i - \boldsymbol{x}\|_2, \sigma) \cdot gauss(\|I(\boldsymbol{x}_i) - I(\boldsymbol{x})\|_2, \sigma_{guide})$$

L2 (=Euclidean) distance

- **Skip neighbors** which have `INVALID_VALUE`.
  - => $w_i = 0$ if $D_0(\boldsymbol{x}_i)$ == `INVALID_VALUE`

- If all neighbors are invalid, mark output pixel also as invalid.

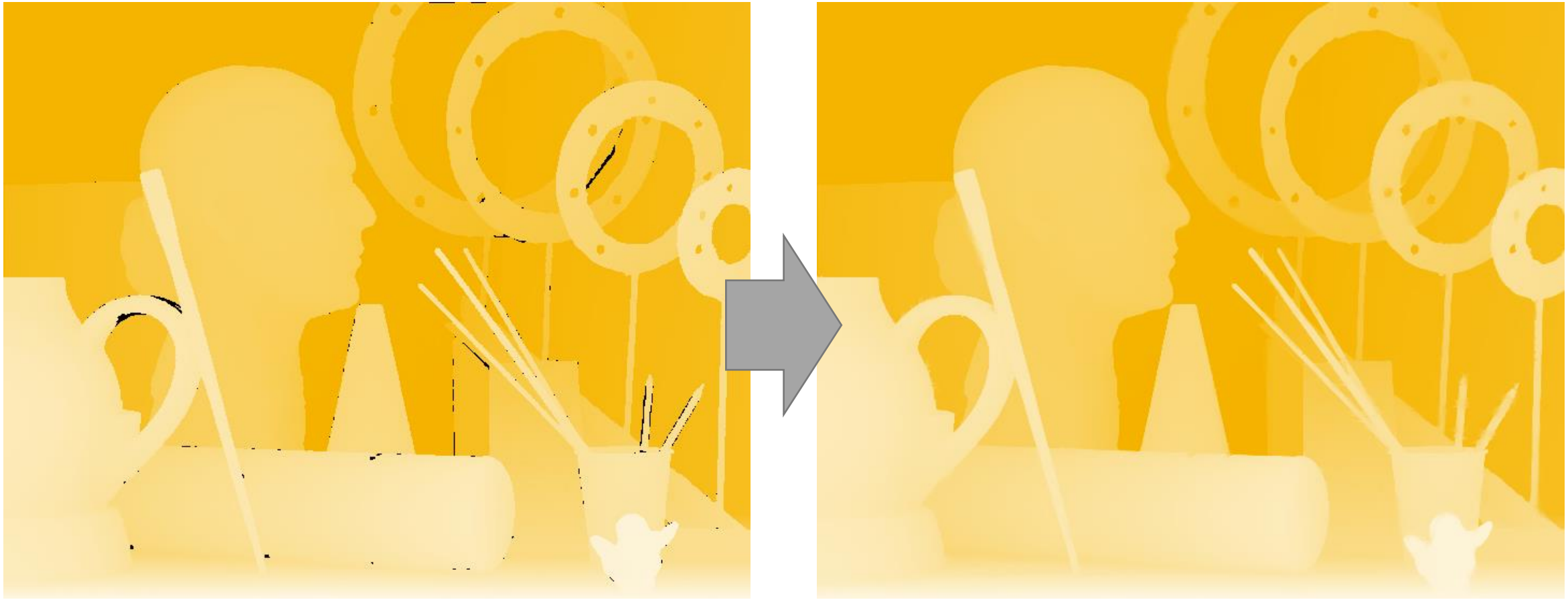
1. Filter disparity

The filter should have shape [size x size] and be symmetric. Size is always odd.

```
ImageFloat jointBilateralFilter(const ImageFloat& disparity, const ImageRGB& guide,
                                const int size, const float guide_sigma)
{
    // We assume both images have matching dimensions.
    assert(disparity.width == guide.width && disparity.height == guide.height);

    // Rule of thumb for gaussian's std dev.
    const float sigma = (size - 1) / 2 / 3.2f;

    // Empty output image.
    auto result = ImageFloat(disparity.width, disparity.height);

    // Notes:
    //   * If a pixel has no neighbor (all were skipped), assign INVALID_VALUE to the output
    //   * One point awarded for a correct OpenMP parallelization.

    //
    //    YOUR CODE GOES HERE
    //

    auto example = gauss(0.5f, 1.2f); // This is just an example

    // Return filtered disparity.
    return result;
}
```

OpenMP points

1. Filter disparity



Before

After

2. Compute normalized linear depth

- We do not know the intended scale of the disparity so we can only extract the depth up to a scale.
- Depth z is **inversely proportional** to disparity d => $z \sim 1/d$
- **Rescale** the depth to [0,1] range.
- If d is `INVALID_VALUE` then `z` is `INVALID_VALUE`

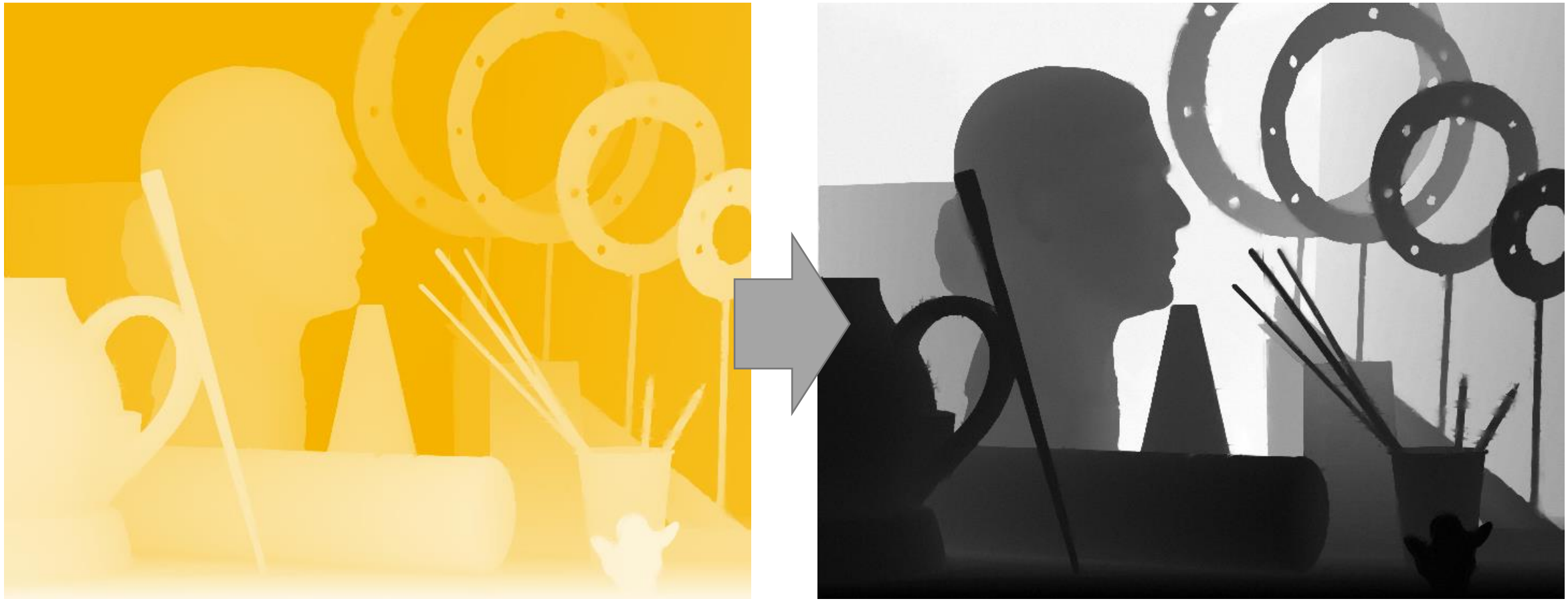
2. Compute normalized linear depth

```
ImageFloat disparityToNormalizedDepth(const ImageFloat& disparity)
{
    auto depth = ImageFloat(disparity.width, disparity.height);

    //
    //    YOUR CODE GOES HERE
    //

    // Rescales valid depth values to [0,1] range.
    normalizeValidValues(depth);

    return depth;
}
```

2. Compute normalized linear depth

```
void normalizeValidValues(ImageFloat& scalar_image)
{
    //
    //    YOUR CODE GOES HERE
    //
}
```

2. Compute normalized linear depth



Disparity

Normalized linear depth

II) Forward warping

Input RGB

Normalized depth

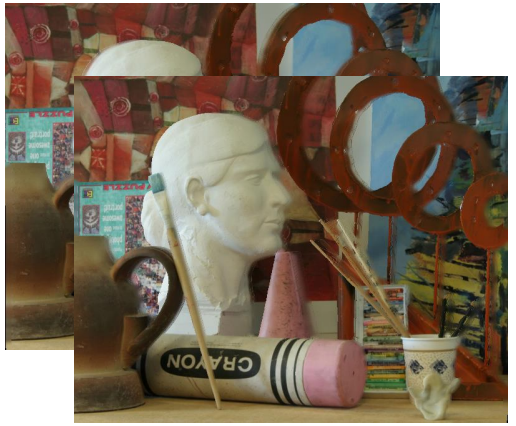
Clean disparity

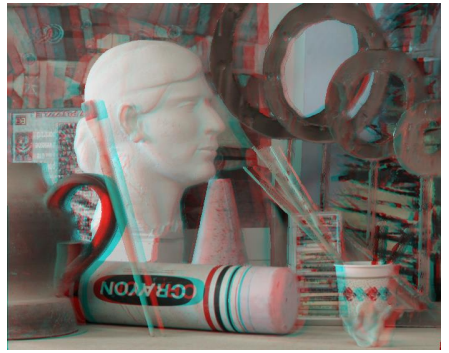
Forward warp (2x)

Inpaint holes (2x)

2x

Stereo pair





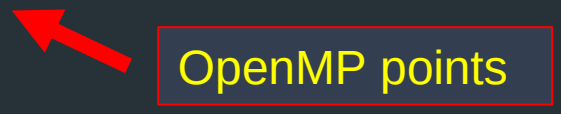
Anaglyph

2.1 Forward warping

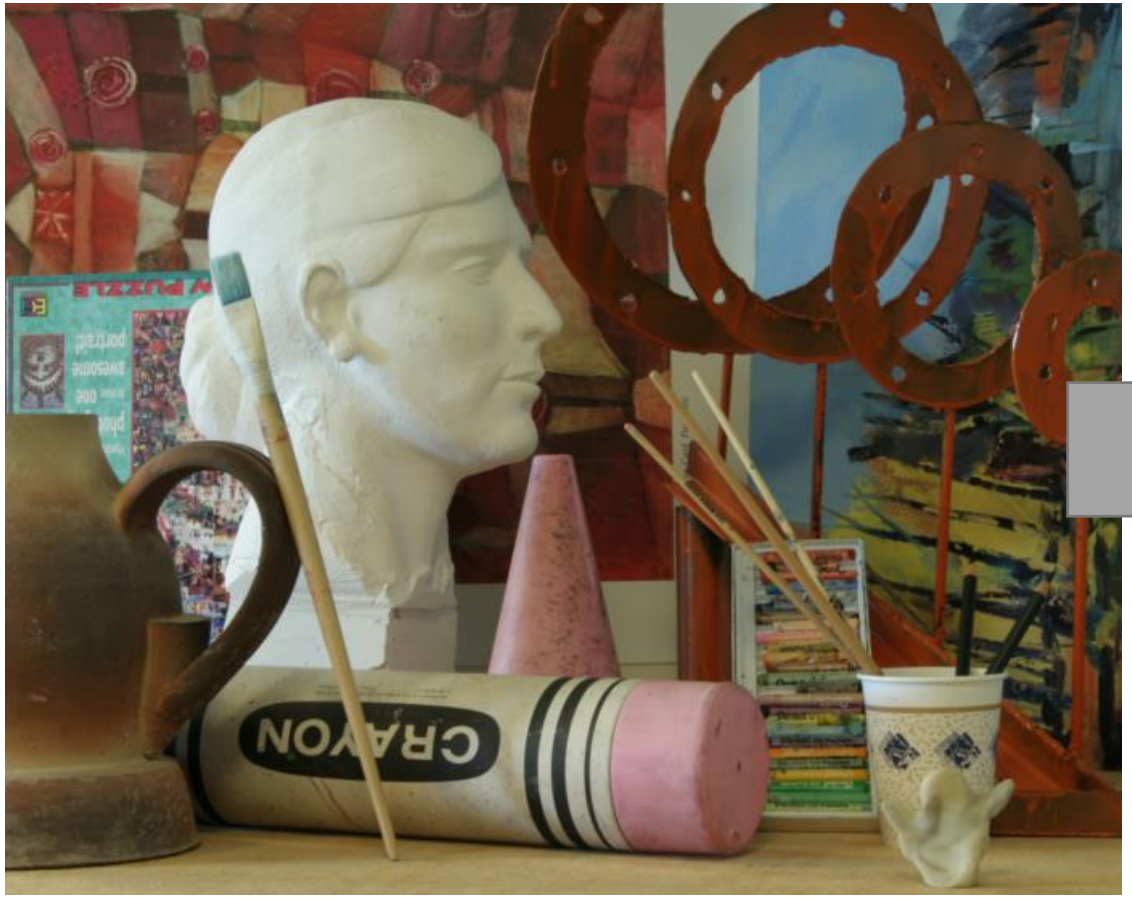
- **Forward warp** the image pixels I using the filtered disparity d.
- Use the common **half-up rounding** (0.5 goes up) for selecting the nearest output pixel coordinates.
  - It is the common school rounding (3.499 down, 4.500 up).
- Leave **empty pixels** set to 0 (black).
- Resolve conflicts using **depth-test**.
  - Overwrite if $z_{src} < z_{dst}$.

2.1 Forward warping

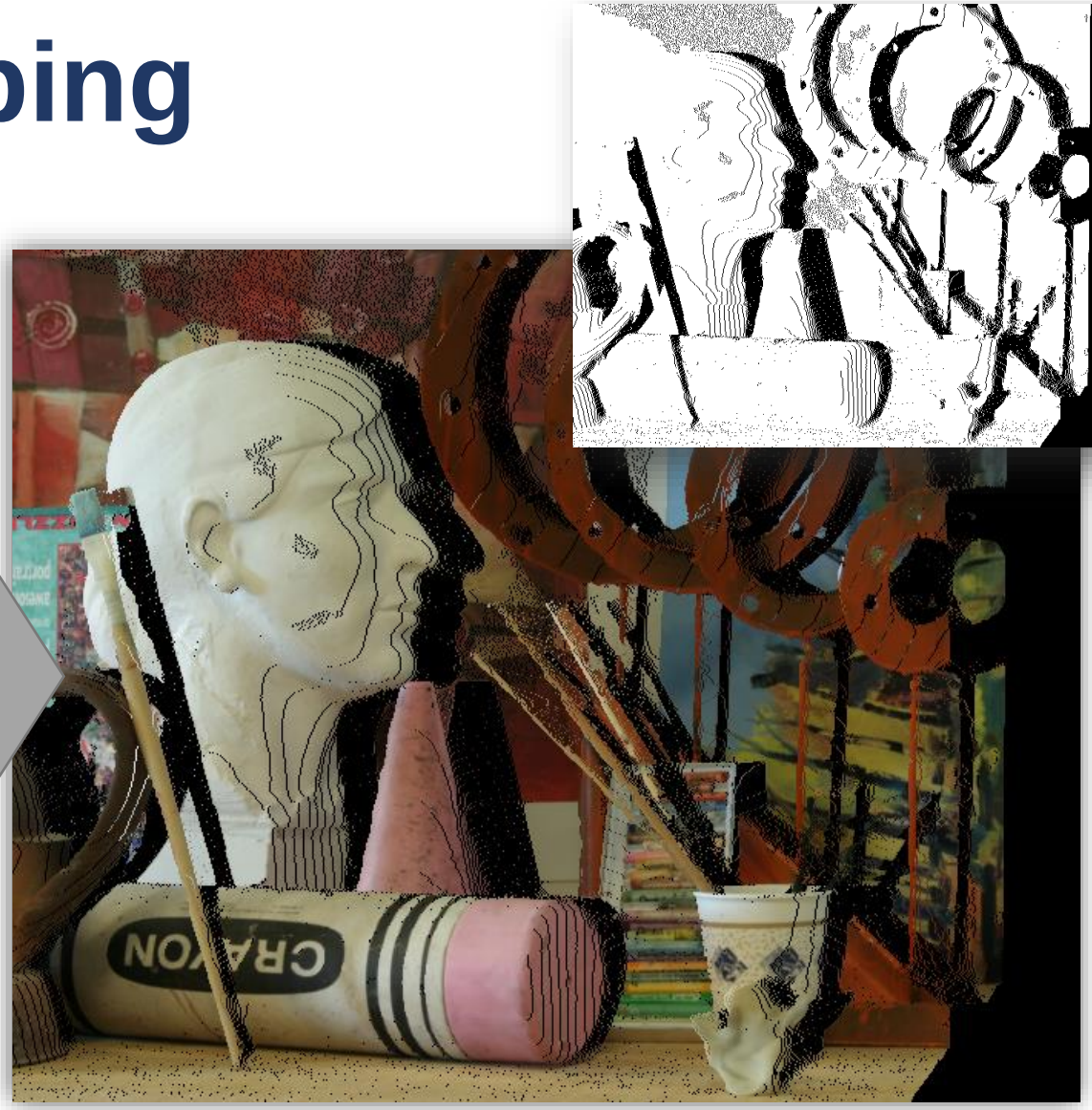
```
ImageWithMask forwardWarpImage(const ImageRGB& src_image, const ImageFloat& src_depth,
                               const ImageFloat& disparity, const float warp_factor)
{
    // The dimensions of src image, src depth and disparity maps all match.
    assert(src_image.width == disparity.width && src_image.height == disparity.height);
    assert(src_image.width == disparity.width && src_depth.height == src_depth.height);

    // Create a new image and depth map for the output.
    auto dst_image = ImageRGB(src_image.width, src_image.height);
    auto dst_mask = ImageFloat(src_depth.width, src_depth.height);
    // Fill the destination depth mask map with zero.
    std::fill(dst_mask.data.begin(), dst_mask.data.end(), 0.0f);
    auto dst_depth = ImageFloat(src_depth.width, src_depth.height);
    // Fill the destination depth map with a very large number.
    std::fill(dst_depth.data.begin(), dst_depth.data.end(), std::numeric_limits<float>::max());

    // Note: Point(s) awarded for a correct and efficient parallel solution using OpenMP.

    //
    //    YOUR CODE GOES HERE
    //

    // Return the warped image and mask.
    return ImageWithMask(dst_image, dst_mask);

}
```



2.1 Test forward warping



Original



Forward Warped

Comparison: Original vs. Warp (animated)

Comparison: Original vs. Warp

Original

Comparison: Original vs. Warp

Forward Warped

Comparison: Z-test vs. flipped Z-test (animated)

Comparison: Z-test vs. flipped Z-test

Correct Z-test

Comparison: Z-test vs. flipped Z-test

Flipped
Z-test

Comparison: Ground truth vs. Warp (animated)

Comparison: Ground truth vs. Warp

Ground truth

Comparison: Ground truth vs. Warp

Forward Warped

2.2 Inpaint holes

- Blur the image with a gaussian filter.
  - **Ignore invalid pixels.**
- Replace empty pixels (mask == 0) with pixels from the blurred image.
- Empty pixels with empty neighborhood stay empty.

2.2 Inpaint holes

```
ImageRGB inpaintHoles(const ImageWithMask& img, const int size, const float guide_sigma)
{
    // The filter size is always odd.
    assert(size % 2 == 1);

    // Rule of thumb for gaussian's std dev.
    const float sigma = (size - 1) / 2 / 3.2f;

    // The output is initialized by copy of the input.
    auto result = ImageRGB(img.image);

    //
    //    YOUR CODE GOES HERE
    //

    return result;
}
```

2.2 Inpaint holes

Inputs

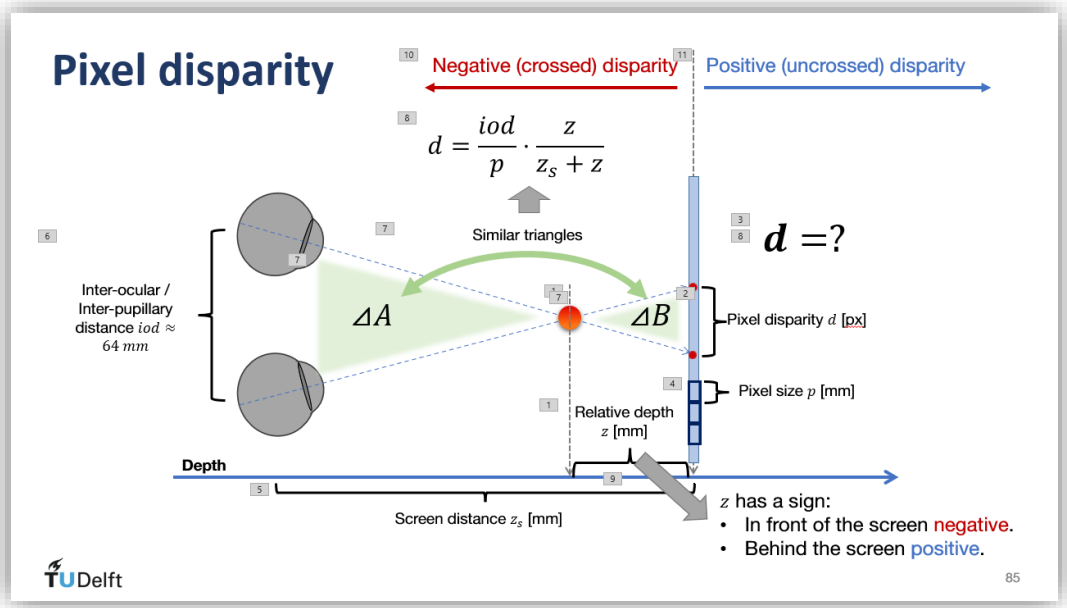
Output

3. Disparity remapping

- **Input**: Normalized depth, target min and max depth in mm, inter-ocular distance, pixel size and screen distance.

- **Goal**: Pixel disparity values that reproduce desired per-pixel depths.

- **Algorithm**:
  1. Linearly **remap the depth** to the target range.
  2. Compute **pixel disparity** for each computed depth pixel.
     - See **Lecture 6** for details.



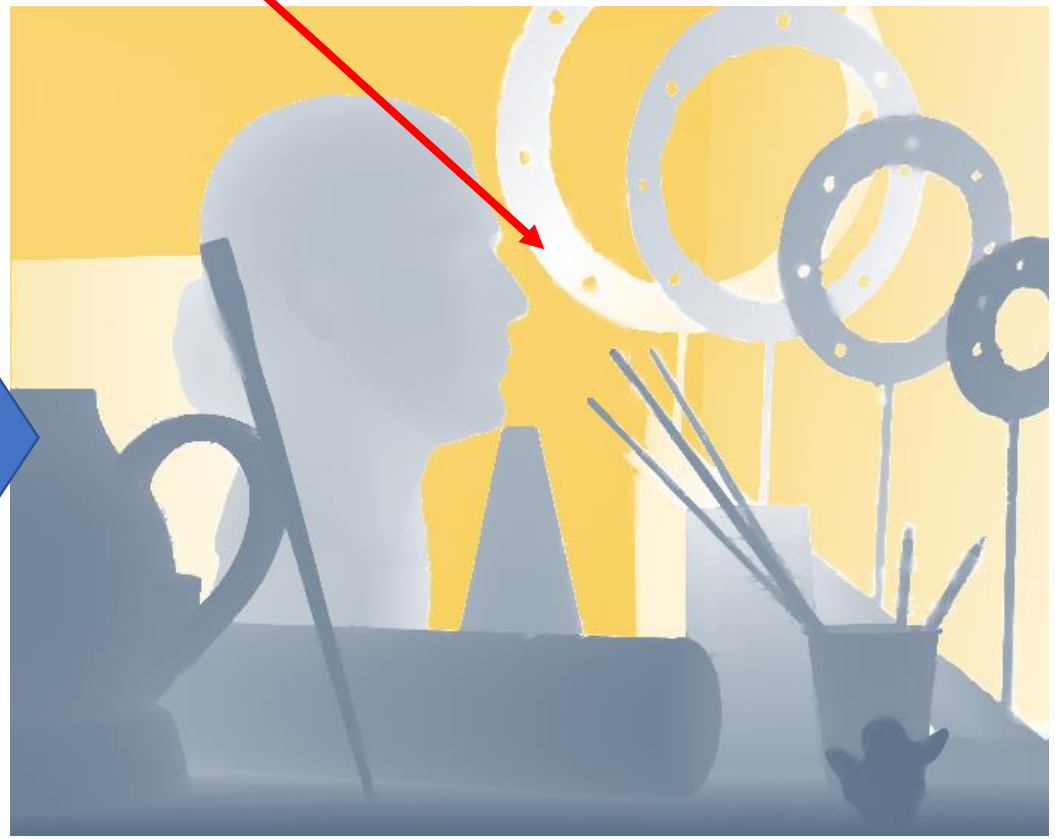
3. Disparity remapping

```
ImageFloat normalizedDepthToDisparity(
    const ImageFloat& depth, const float iod_mm,
    const float px_size_mm, const float screen_distance_mm,
    const float near_plane_mm, const float far_plane_mm)
{
    auto px_disparity = ImageFloat(depth.width, depth.height);

    //
    //    YOUR CODE GOES HERE
    //

    return px_disparity; // returns disparity measured in pixels
}
```

3. Disparity remapping

Hint: Check that zero values (white) are where you expect them.

Normalized depth



Remap pixel disparities

Intermezzo

(just to explain what happens, nothing to do here)

- The **main.cpp** code runs our backward **warping twice** with our new disparity equally distributed between left and right image:

```
std::vector<ImageRGB> image_pair;
for (int i = 0; i < 2; i++) {
    // The total disparity is split in half between both images as each gets shifted in an
    // opposite direction.
    auto warp_factor = i == 0 ? -0.5f : 0.5f;

    // Forward warp the image. We use the scaled disparity.
    auto img_forward = forwardWarpImage(image, linear_depth, target_disparity, warp_factor);
    // Inpaint the holes in the forward warping image.
    ImageRGB dst_image = inpaintHoles(img_forward, scene_params.bilateral_size);

    image_pair.push_back(std::move(dst_image));
}
```

Intermezzo

(just to explain what happens, nothing to do here)

$\times (-0.5)$

$\times (+0.5)$

Left

Right

Intermezzo – Left vs. Right (animated)

Intermezzo – Left vs. Right

Left

Intermezzo – Left vs. Right

Right

Intermezzo – Free viewing

Left

Right

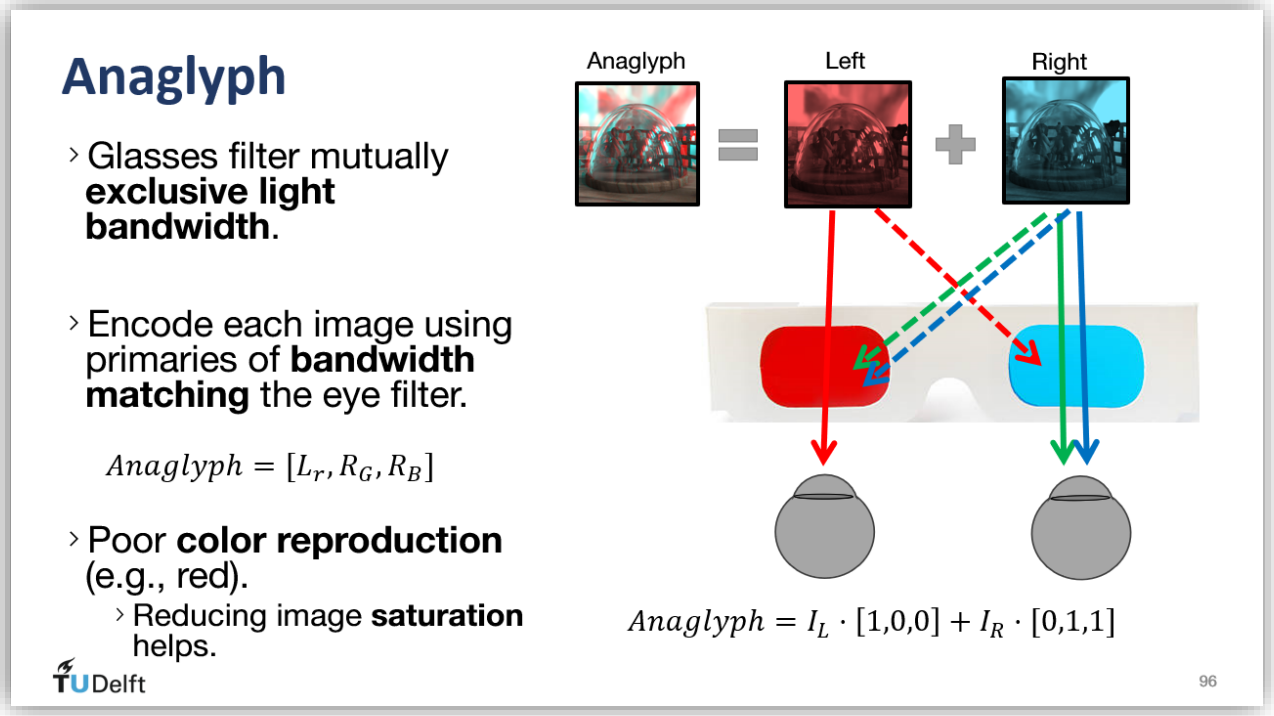
Left

Uncrossed

Crossed

5. Anaglyph

- Convert the left + right stereo pair into an **anaglyph image**.
  - See **Lecture 6** for details:



- Multiply **saturation** of each **input** pixel by the provided factor.
  - Convert to HSV color pace and back (methods provided).

5. Anaglyph

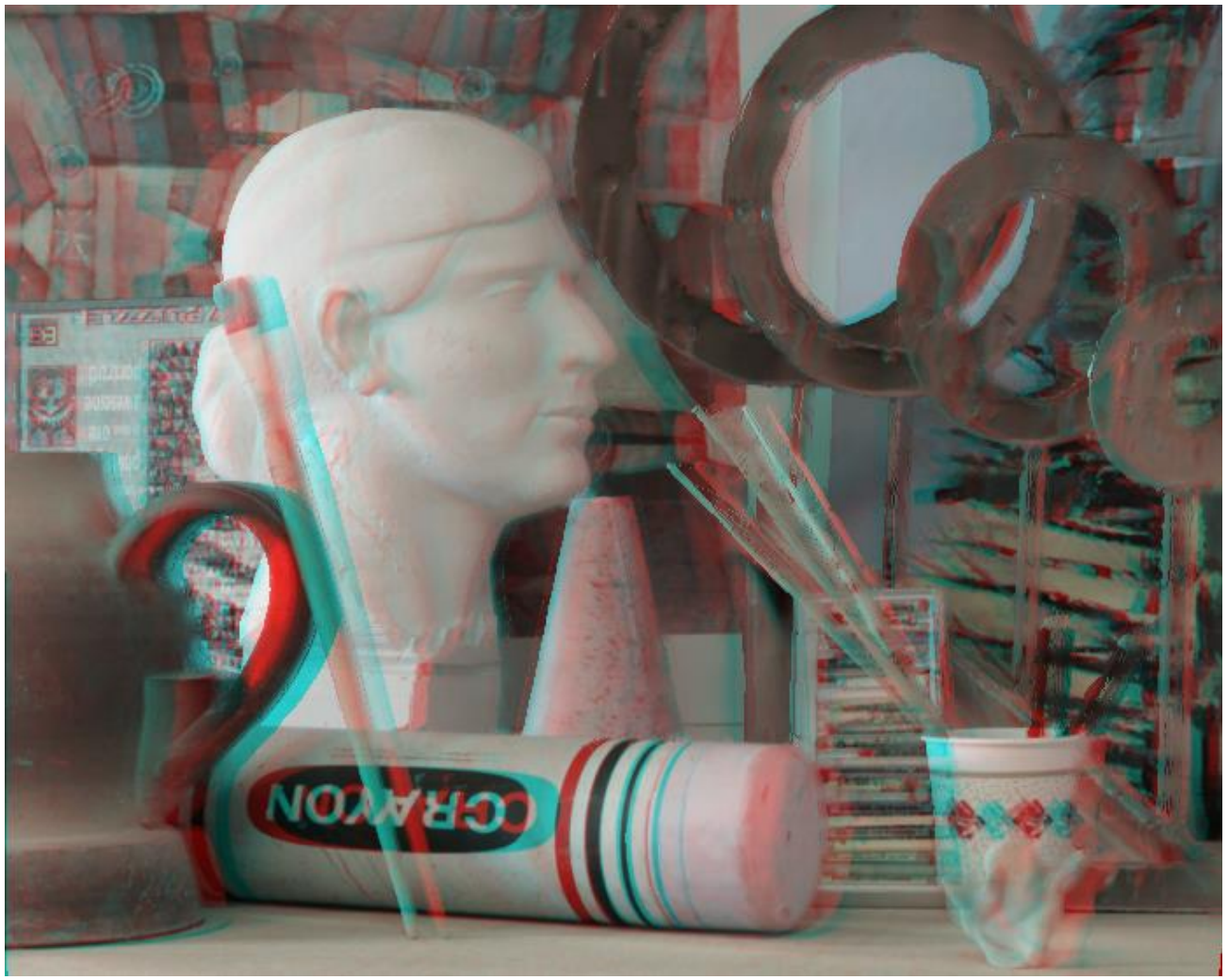
```
ImageRGB createAnaglyph(const ImageRGB& image_left, const ImageRGB& image_right, const float saturation)
{
    // An empty image for the resulting anaglyph.

    // Example: RGB->HSV->RGB should be approx identity.
    auto rgb_orig = glm::vec3(0.2, 0.6, 0.4);
    auto rgb_should_be_same = hsvToRgb(rgbToHsv(rgb_orig));
    // expect rgb == rgb_2 (up to numerical precision)

    //
    //    YOUR CODE GOES HERE
    //

    // Returns a single analgyph image.
    return anaglyph;
}
```

An example usage of RGB<->HSV conversion code provided to you.

5. Anaglyph

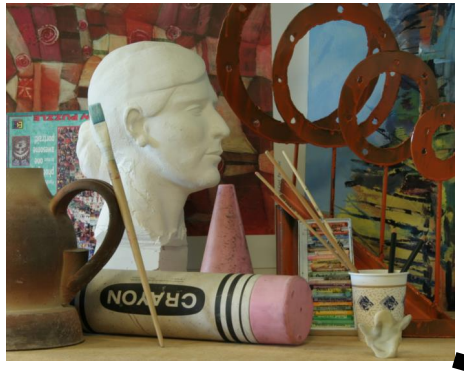
Left

Right



Anaglyph image

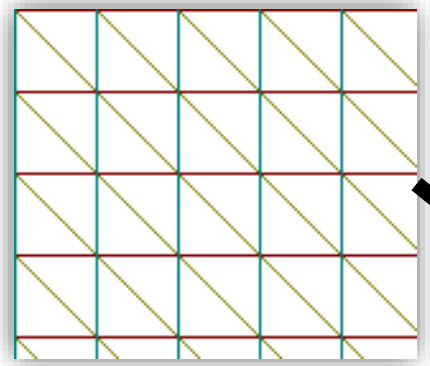
III) Backward warping



Input RGB

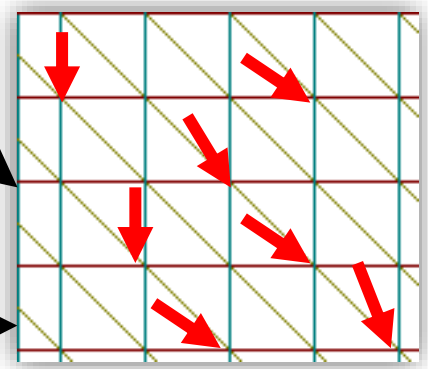
Normalized depth

Clean disparity



Pixel grid

Stereo pair



Warped grid

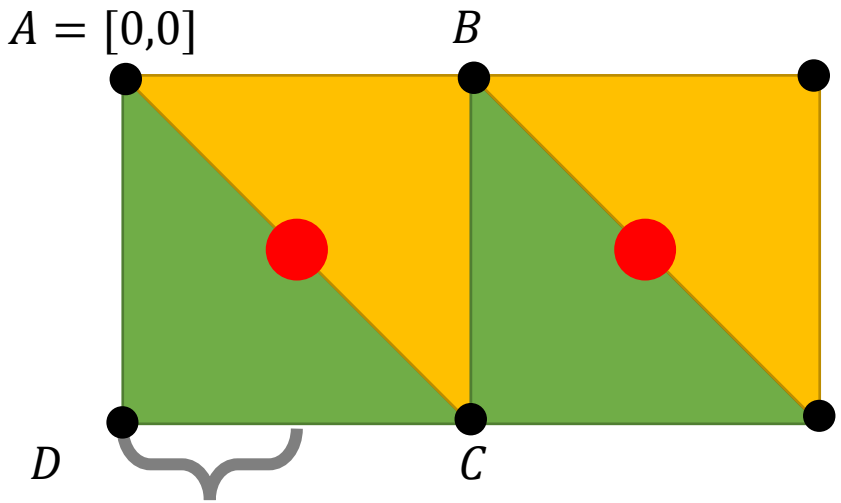
Anaglyph

6. Create a warping grid mesh

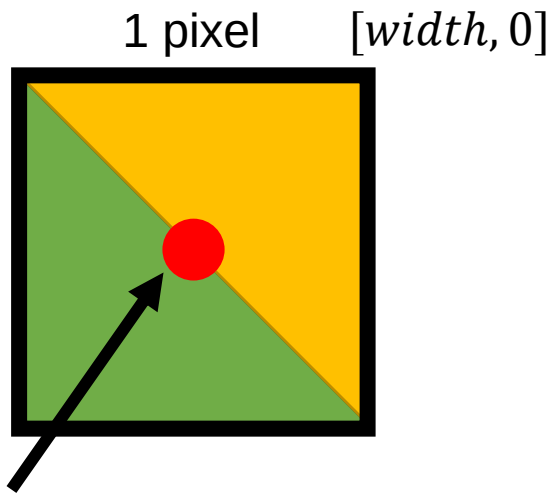
- **2** triangles per each pixel in the $w \times h$ pixel image
  - **Vertex buffer** = List of vertices (points) matching the pixel corners.
    - Shared between neighbors => $N_v = (w+1) \cdot (h+1)$
  - **Index buffer** = List of $N_{tri} = 2 \cdot w \cdot h$ triangles.
    - Triangle = 3x index to vertex buffer selecting the vertices in **clockwise** order.

- Vertex coordinates **go from [0,0] to [width, height]** where the extremes correspond to outer boundaries of the image.

6. Create a warping grid mesh



...



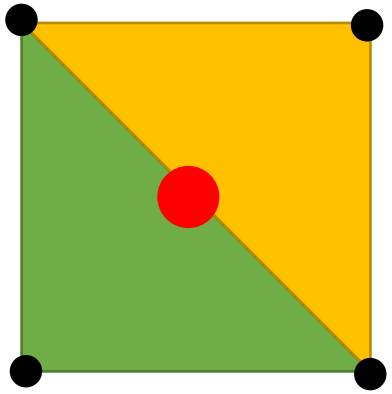
(Row by row)

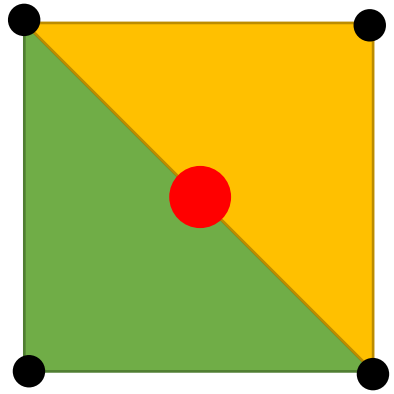
$$V_{buffer} = [A, B, \dots D, C, \dots]$$

$$I_{buffer} = [(A, B, C), (A, C, D), \dots]$$

(Clockwise, yellow first, row by row)

...

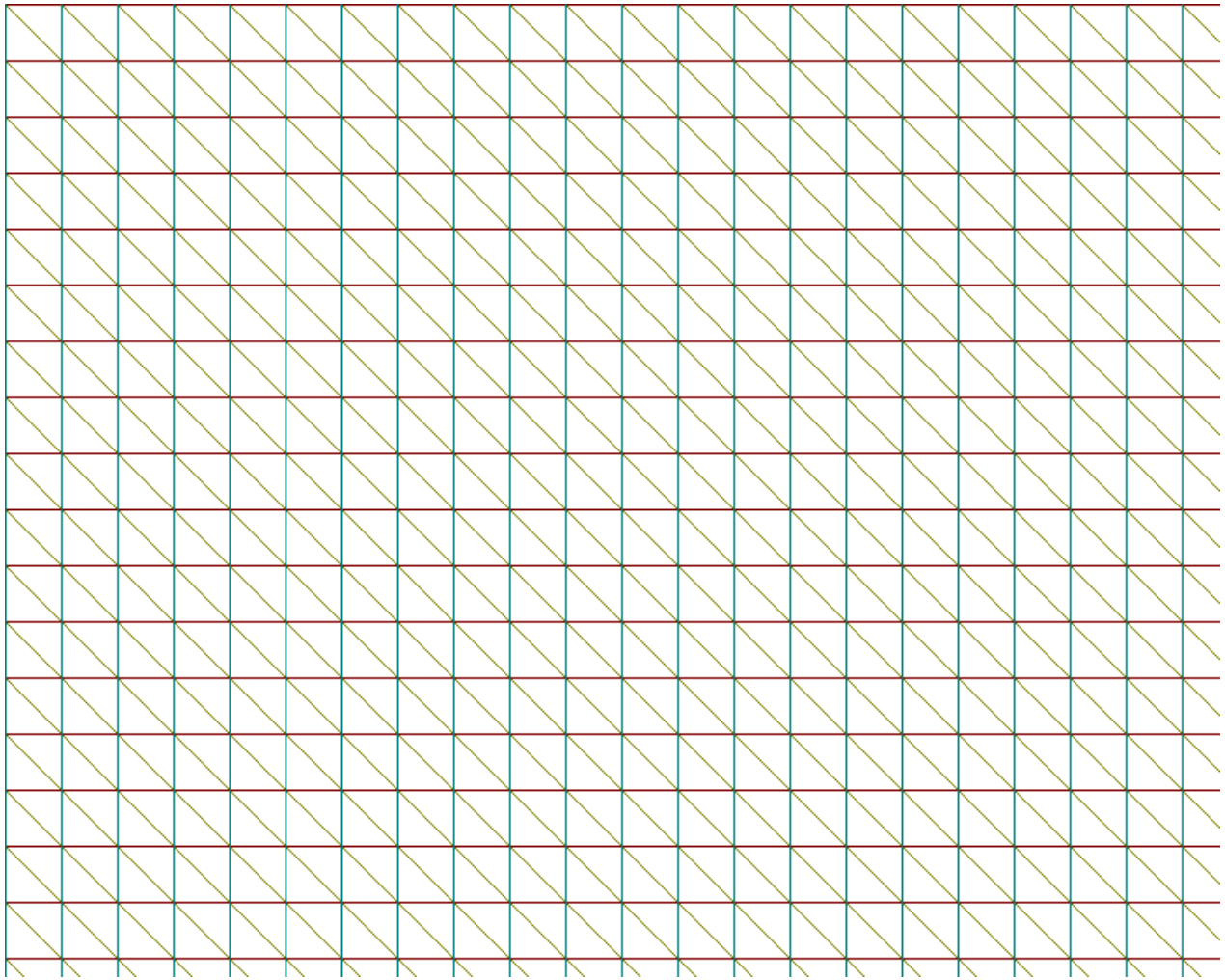




6. Create a warping grid mesh

```
Mesh createWarpingGrid(const int width, const int height) {
    // Build vertex buffer.
    auto num_vertices = (width + 1) * (height + 1);
    auto vertices = std::vector<glm::vec2>(num_vertices);

    //
    //    YOUR CODE GOES HERE
    //

    // Build index buffer.
    auto num_pixels = width * height;
    auto num_triangles = num_pixels * 2;
    auto triangles = std::vector<glm::ivec3>(num_triangles);

    //
    //    YOUR CODE GOES HERE
    //

    // Combine the vertex and index buffers into a mesh.
    return Mesh { std::move(vertices), std::move(triangles) };
}
```

6. Create a warping grid mesh

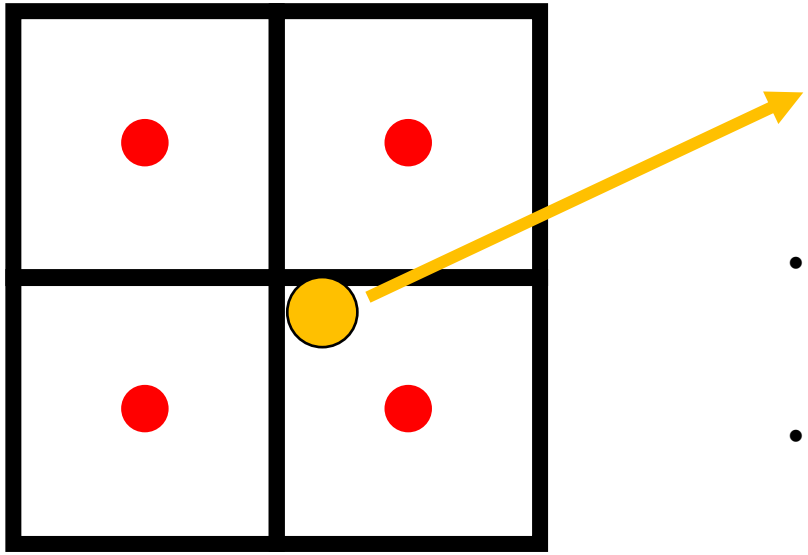


Left-top section of the grid

7. Warp the grid

- **Shift the vertices** in horizontal direction by values described in the disparity map (**normalized** and **scaled** by a provided factor).
  - **BUT** Keep the image **boundary vertices** attached to the boundary.

- Use **bilinear sampling** to read the disparity map.



2x2 pixel sub-image

$$f(x) = \sum_{i=0}^{4} w_i \cdot f(x_i)$$

- See **Lecture 6** for details of bilinear interpolation.
- Consider a general case (arbitrary α, β)!

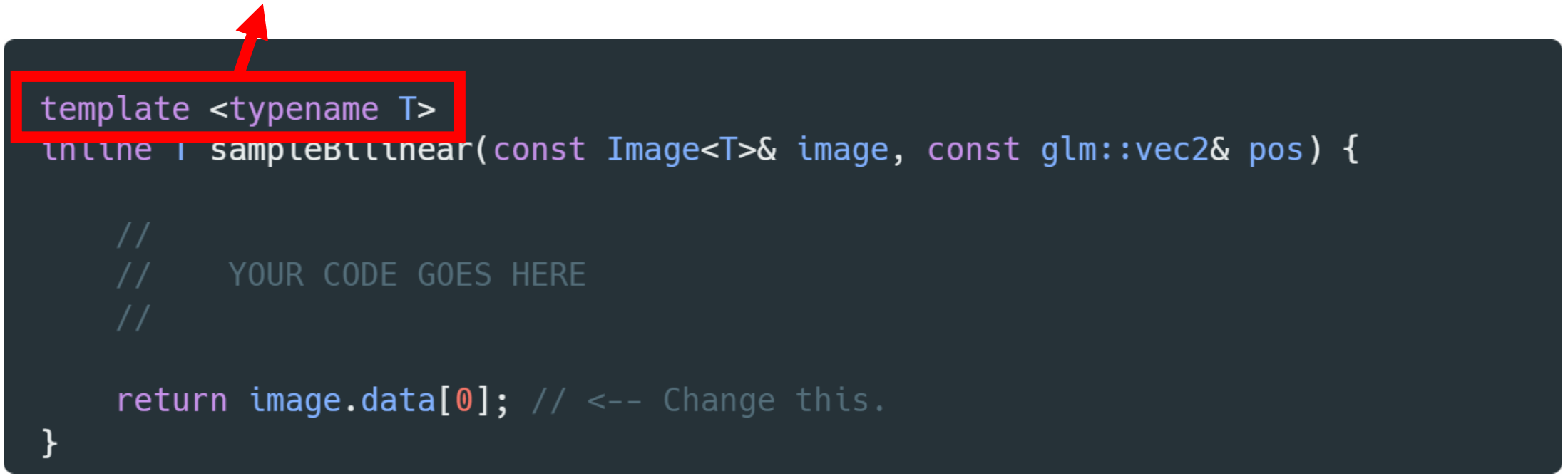
7. Warp the grid

This is reference to your method for you and to a reference implementation in our tests.

```
Mesh warpGrid(Mesh& grid, const ImageFloat& disparity, const float scaling_factor,
              const BilinearSamplerFloat& sampleBilinear)
{
    // Create a copy of the input mesh (all values are copied).
    auto new_grid = Mesh { grid.vertices, grid.triangles };
    const float EDGE_EPSILON = 1e-5f * disparity.width;

    // Here is an example use of the bilinear interpolation (using the provided function argument).
    auto interpolated_value = sampleBilinear(disparity, glm::vec2(1.0f, 1.0f));
    // Recommended test: For a 2x2 image it SHOULD return the mean of the 4 pixels.

    //
    //    YOUR CODE GOES HERE
    //

    return new_grid;
}
```

7. Warp the grid

Templated to work with both RGB and Float images.
Use glm:: functions to write code that works for both T = **float** and T = **glm::vec3**.

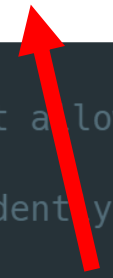


```
template <typename T>
inline T sampleBilinear(const Image<T>& image, const glm::vec2& pos) {

    //
    //    YOUR CODE GOES HERE
    //

    return image.data[0]; // <-- Change this.
}
```

7. Warp the grid

Your code (main.cpp):

This is how we connect them together.

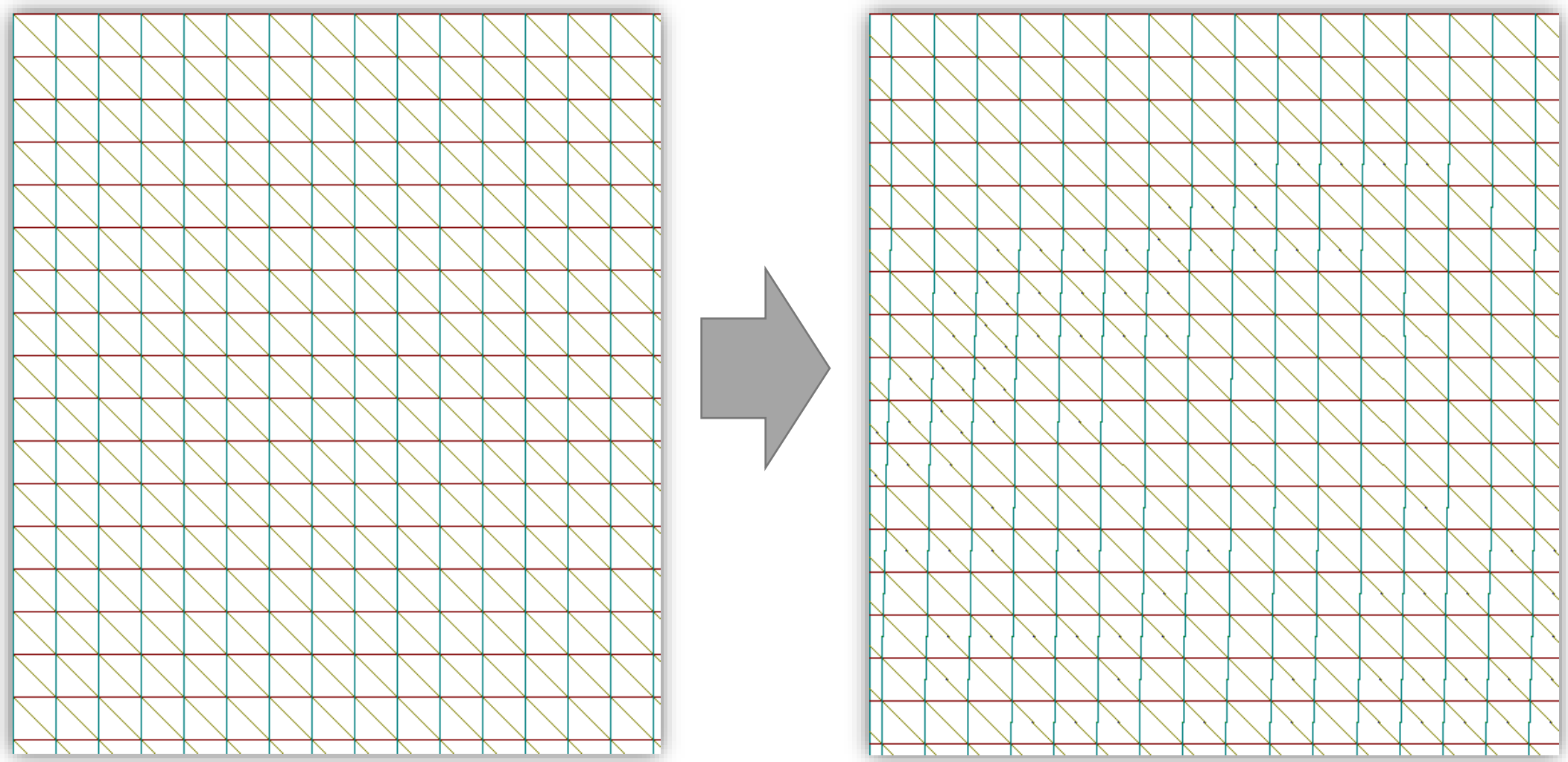


```
// Note that we are passing your sampleBilinear() function as an argument. That allows us to replace it with reference
// implementation during grading and therefore judge it and warpGrid() independently without propagating errors in one to the other.
auto dst_grid = warpGrid(src_grid, disparity_filtered, scene_params.warp_scale, sampleBilinear<float>);
```

Our grading tests:

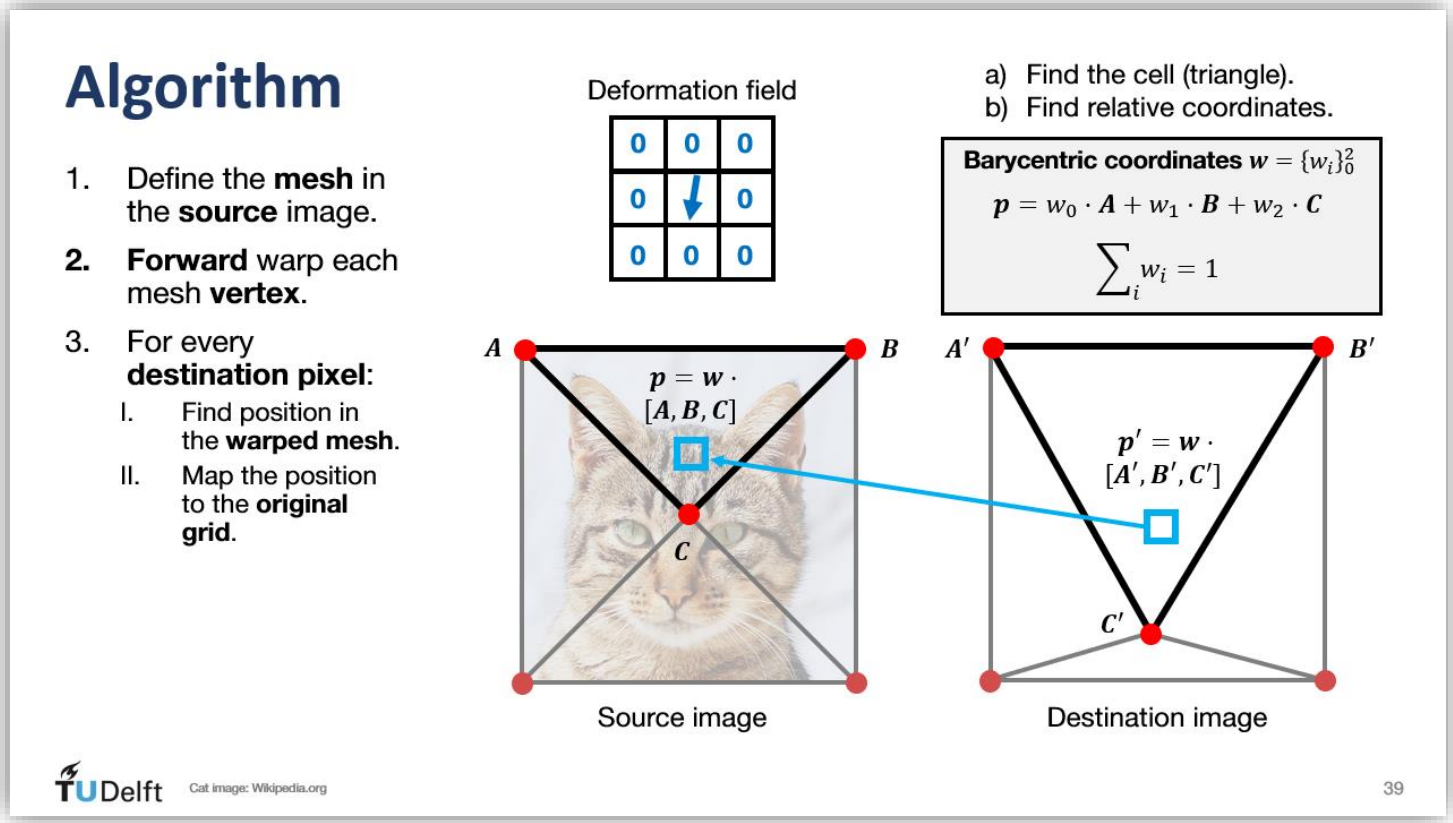
```
auto reference_grid = correct::warpGrid(grid, disparity, scene_params.warp_scale, correct::sampleBilinear<float>);
// vs.
auto user_grid = warpGrid(grid, disparity, scene_params.warp_scale, correct::sampleBilinear<float>);
```

7. Warp the grid



8. Mesh-based warping

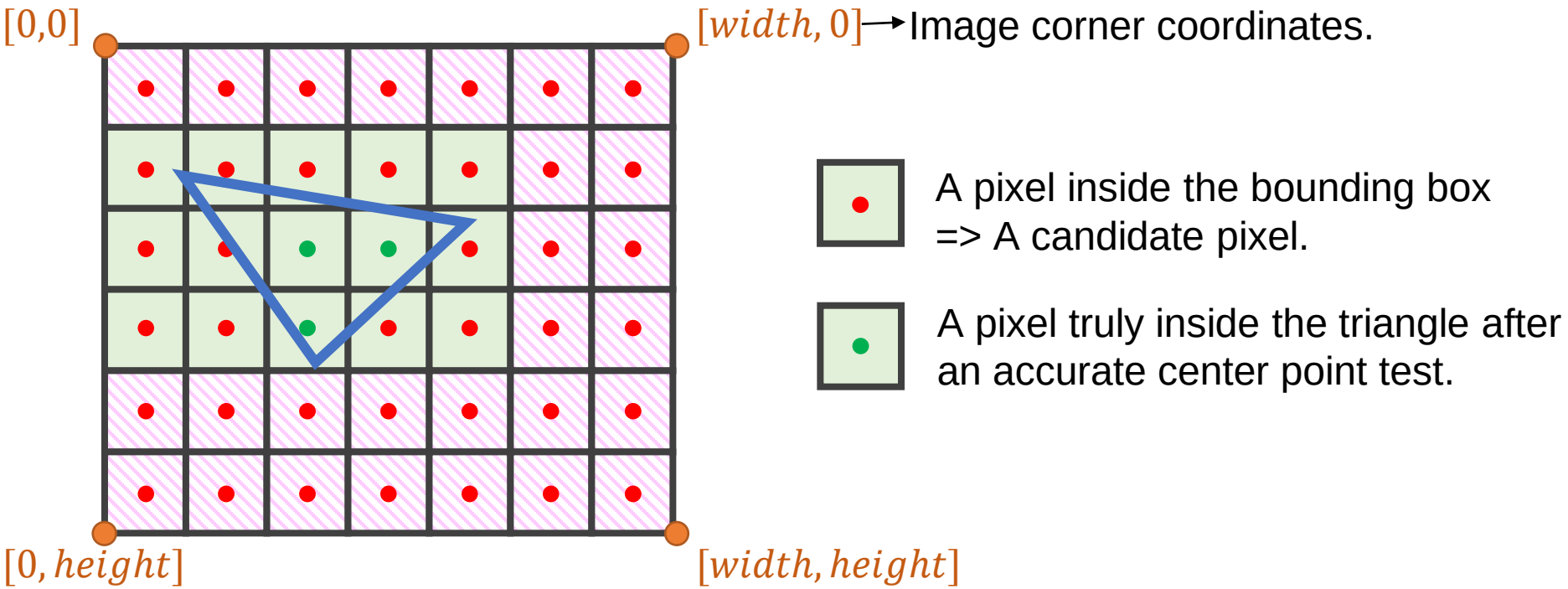
› Implement **mesh-based** DIBR warping method as described in **Lecture 6**:



8. Mesh-based warping

› **Algorithm** (a brief summary, refer to Lecture 6 for details)**:**

1. For every warped triangle, find pixels which **centers** are inside said triangles.
   - Use **bounding box** (min-max range) to restrict the candidates – **DO NOT test every pixel of the image with every triangle!**



8. Mesh-based warping

- **Algorithm** (a brief summary, refer to Lecture 6 for details)**:**

1. For every warped triangle, find pixels which **centers** are inside said triangles.
   - Use **bounding box** (min-max range) to restrict the candidates – **DO NOT test every pixel of the image with every triangle!**
2. Compute **barycentric coordinates** of the pixel in the **warped triangle**.
3. Use said coordinates to compute corresponding point in the corresponding **source triangle**.
4. **Bilinearly** sample the **source image and depth** at said point.
5. Apply the same **depth-test** logic as for the forward warping earlier.

8. Mesh-based warping

```cpp
ImageRGB backwardWarpImage(const ImageRGB& src_image, const ImageFloat& src_depth, const Mesh& src_grid, const Mesh& dst_grid,
    const BilinearSamplerFloat& sampleBilinear, const BilinearSamplerRGB& sampleBilinearRGB)
{
    // The dimensions of src image and depth match.
    assert(src_image.width == src_depth.width && src_image.height == src_depth.height);
    // We assume that both grids have the same size and also the same order (ie., there is 1:1 triangle pairing).
    // This implies that the content of index buffers of both meshes are exactly equal (we do not test it here).
    assert(src_grid.triangles.size() == dst_grid.triangles.size());

    // Create a new image and depth map for the output.
    auto dst_image = ImageRGB(src_image.width, src_image.height);
    auto dst_depth = ImageFloat(src_depth.width, src_depth.height);
    // Fill the destination depth map with a very large number.
    std::fill(dst_depth.data.begin(), dst_depth.data.end(), 1e20f);

    // Example of testing point [0.1, 0.2] is inside a triangle.
    bool is_point_inside = isPointInsideTriangle(glm::vec2(0.1, 0.2), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(0, 1));

    // Example of computing barycentric coordinates of a point [0.1, 0.2] inside a triangle.
    glm::vec3 bc = barycentricCoordinates(glm::vec2(0.1, 0.2), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(0, 1));

    //
    //    YOUR CODE GOES HERE
    //

    // Return the warped image.
    return dst_image;
}
```

For depth and color.

An example usage of methods provided to you.

9. Mesh-based warping

Original

Mesh-based Backward Warped

Comparison: Ground truth vs. Back-Warp

(animated)

Comparison: Ground truth vs. Back-Warp

Ground truth

Comparison: Ground truth vs. Back-Warp

Backward Warped

Comparison: Forward vs. Backward Warp
(animated)

Comparison: Forward vs. Backward Warp

Forward

Comparison: Forward vs. Backward Warp

Backward

Bwd Stereo – Left vs. Right (animated)

Bwd Stereo – Left vs. Right

Left

Bwd Stereo – Left vs. Right

Right

Bwd Stereo – Free viewing

Left

Right

Left

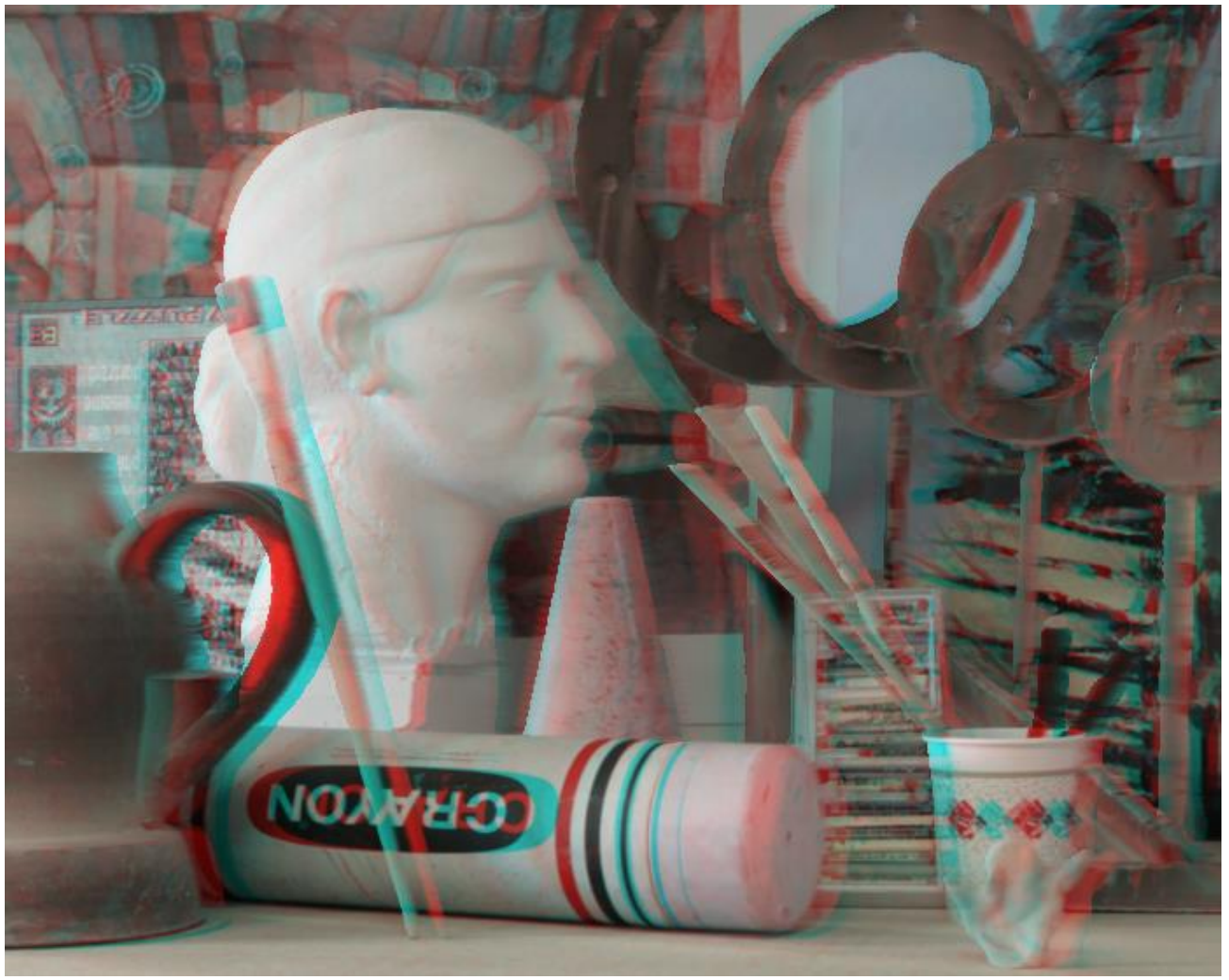
Uncrossed

Crossed

Bwd Stereo - Anaglyph

Left

Right



Anaglyph image

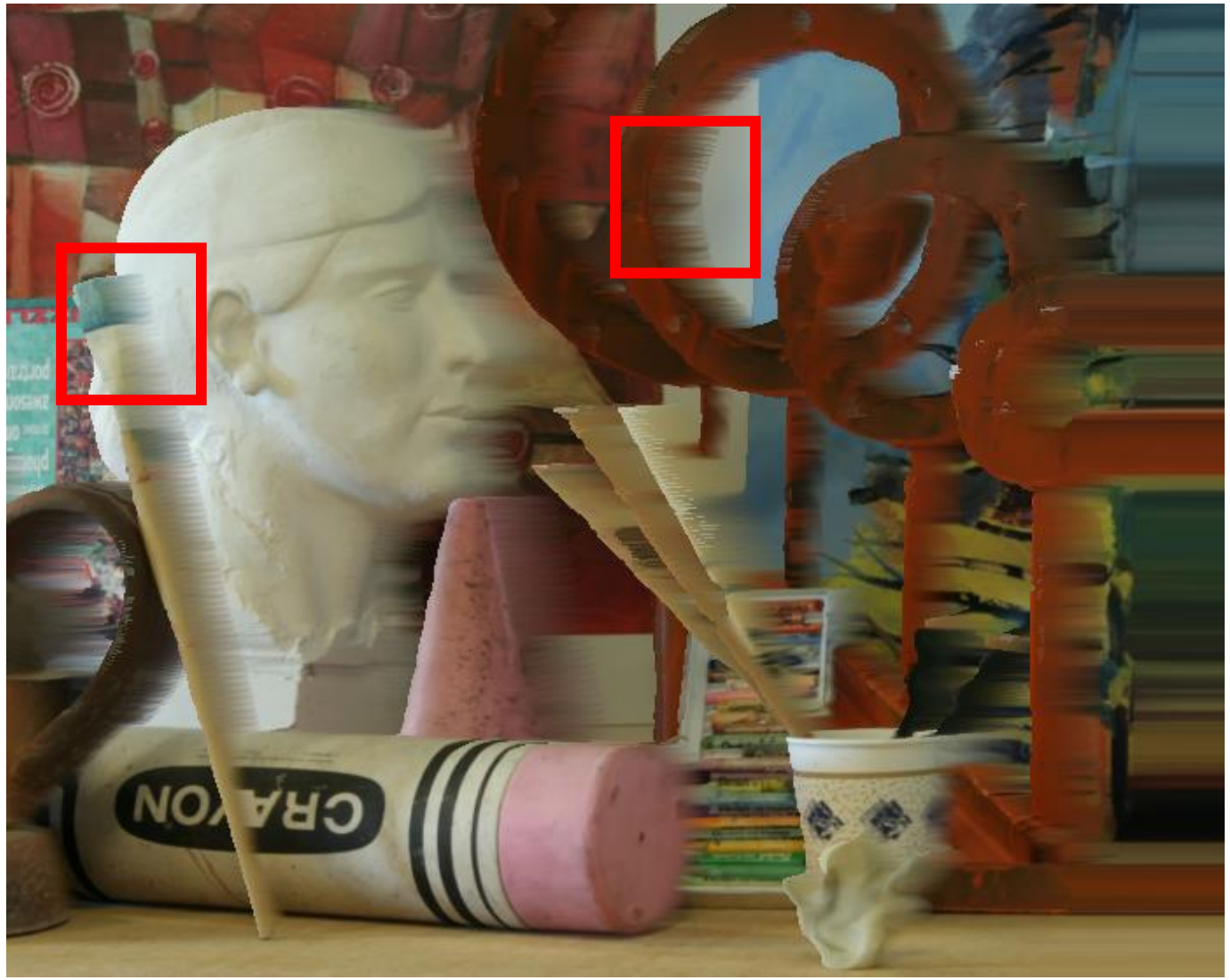
Limitations

- The **disoccluded** areas are filled with blend of foreground and background.

- Potential solutions **outside of the Assignment scope**:

  - **Snap the grid** to the front most object.

  - Detect disocclusions, remove, **inpaint**.



Backward warp

11. Image rotation

Challenge – Do without asking TAs!

- **Input**: Image, point and angle in degrees.
- **Output**: Image rotated counter-clockwise by the given **angle** around the given **point**.
- **Requirements:** Use the backward warping approach.
- **Notes**:
  - Output has the same shape as the input.
  - Ignore pixels that fall outside of the image.
  - Set pixels that become empty black (zero).

11. Rotate the warping grid

```
Mesh rotatedWarpGrid(Mesh& grid, const glm::vec2& center, const float& angle_deg)
{
    // Create a copy of the input mesh (all values are copied).
    auto new_grid = Mesh { grid.vertices, grid.triangles };

    const float DEGREE2RADIANS = 0.0174532925f;

    //
    //    YOUR CODE GOES HERE
    //

    return new_grid;
}
```

12. Image rotation

```
ImageRGB rotateImage(const ImageRGB& image, const Mesh& src_grid, const Mesh& dst_grid,
                     const BilinearSamplerFloat& sampleBilinear, const BilinearSamplerRGB& sampleBilinearRGB)
{
    //
    //    YOUR CODE GOES HERE
    //
    return ImageRGB(1, 1); // replace
}
```

12. Image rotation (angle = 45°)

Grading

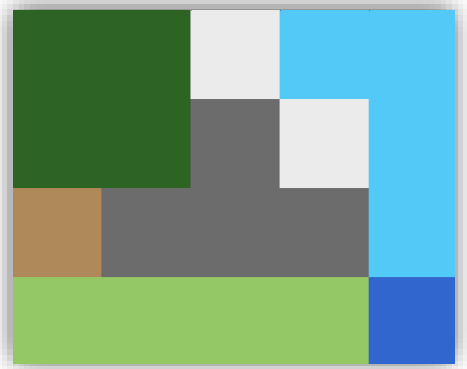
Step	Method	Maximum points
1	jointBilateralFilter(…)	3 + 2 points for OpenMP
2a	normalizeValidValues(…)	2
2b	disparityToNormalizedDepth(…)	1
2.1	forwardWarpImage(…)	3 + 2 points for OpenMP
2.2	inpaintHoles(…)	2
3	normalizedDepthToDisparity(…)	2
5	createAnaglyph(…)	1
6	createWarpingGrid(…)	3
7a	sampleBilinear(…)	3
7b	warpGrid(…)	1
8	backwardWarpImage(…)	5
11	rotatedWarpGrid(…)	3
12	rotateImage(…)	1
	Total	34

Partial points are awarded based on passed unit tests: $\text{grade} = 1 + \frac{\text{points}}{\text{max points}} * 9$

Inputs & Outputs

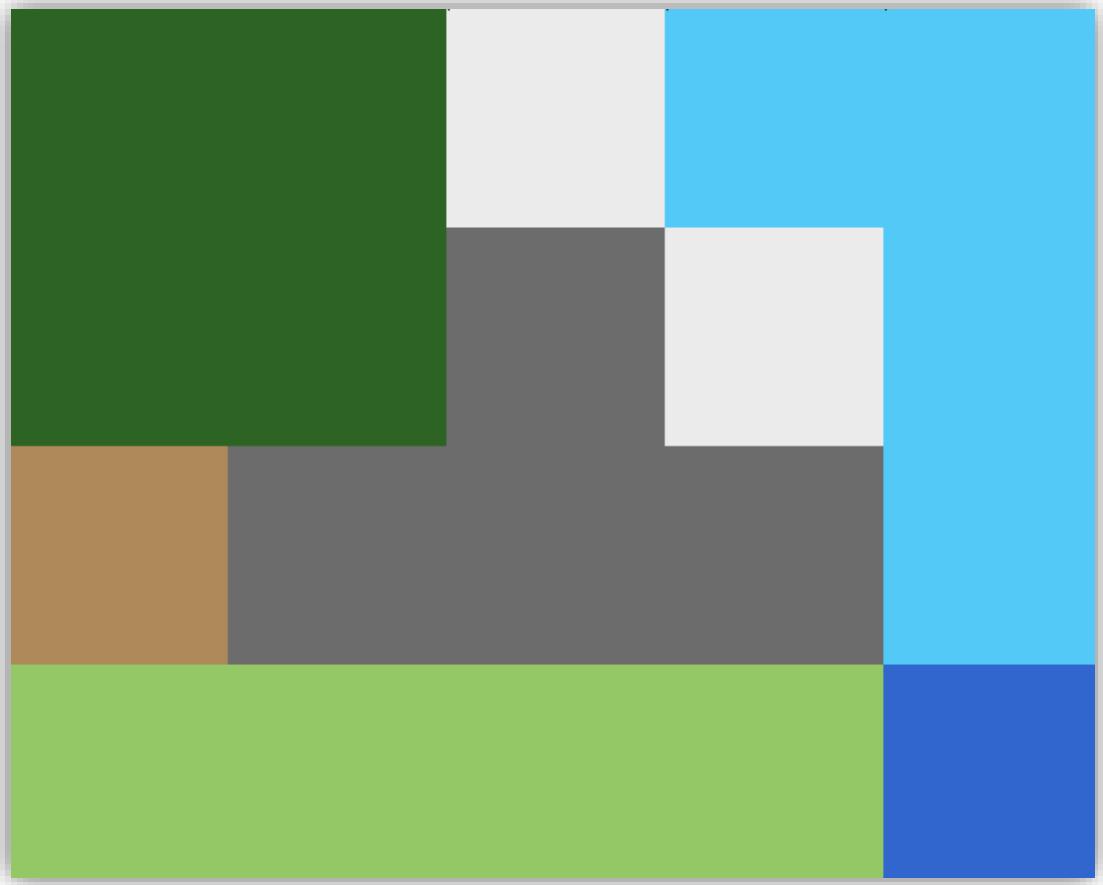
- **Two inputs** provided - (un)comment in `main.cpp` to switch.

```
// Change your inputs here!
const auto input_select = InputSelection::Mini;
//const auto input_select = InputSelection::Middlebury;
```

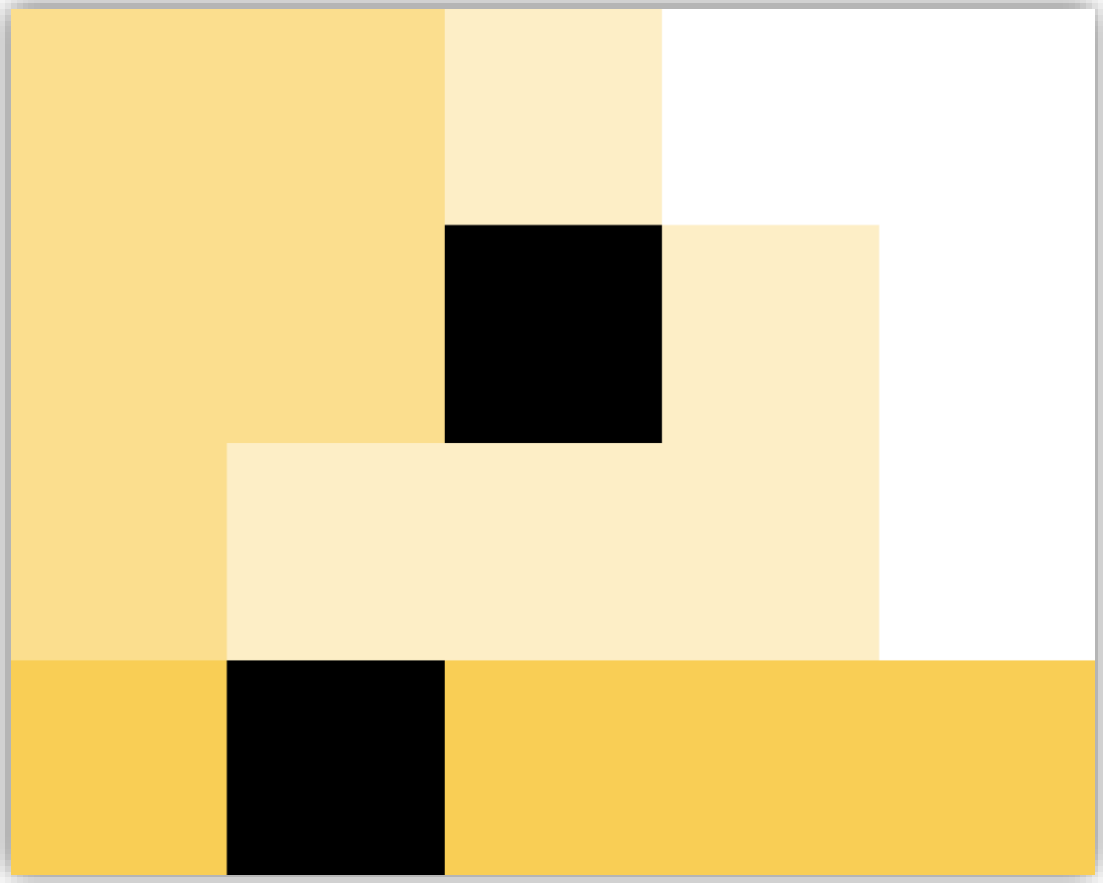


- No `expected-outputs` provided.
  - Use logical reasoning to determine whether your input is correct.

Mini test input

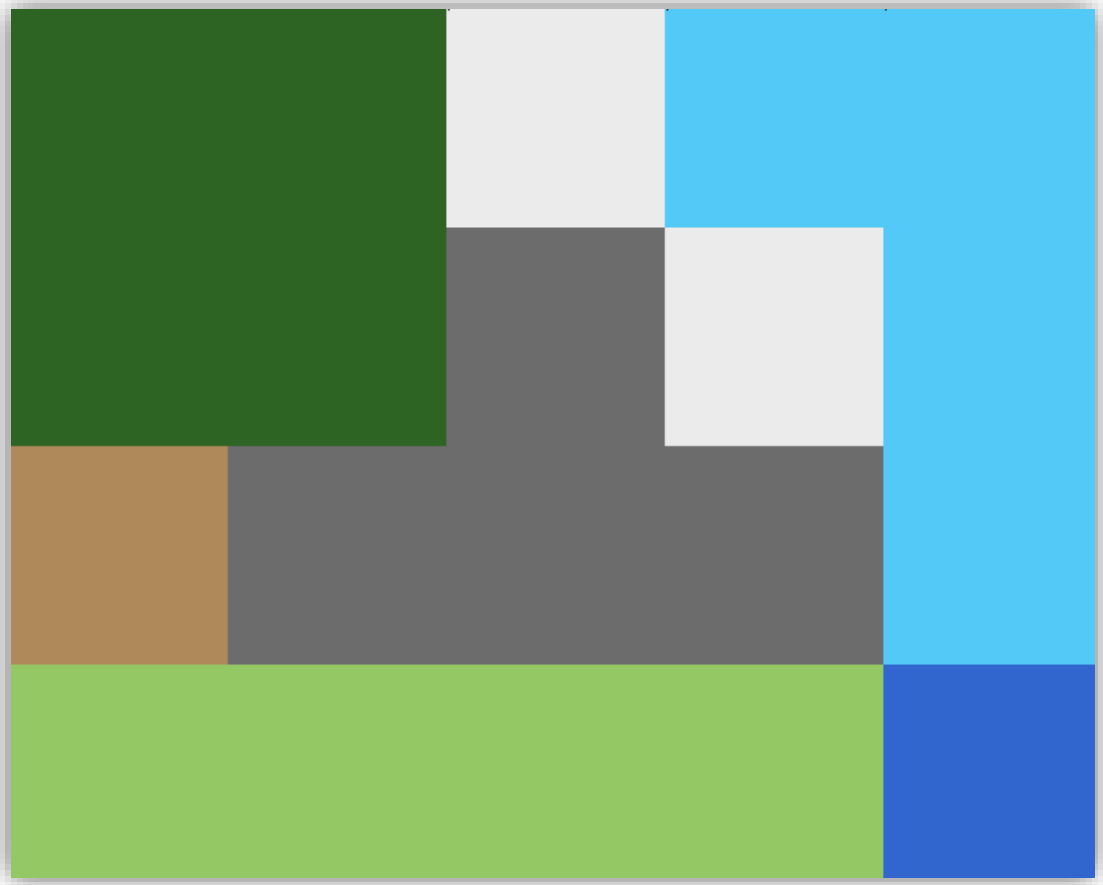


5x4 pixel image

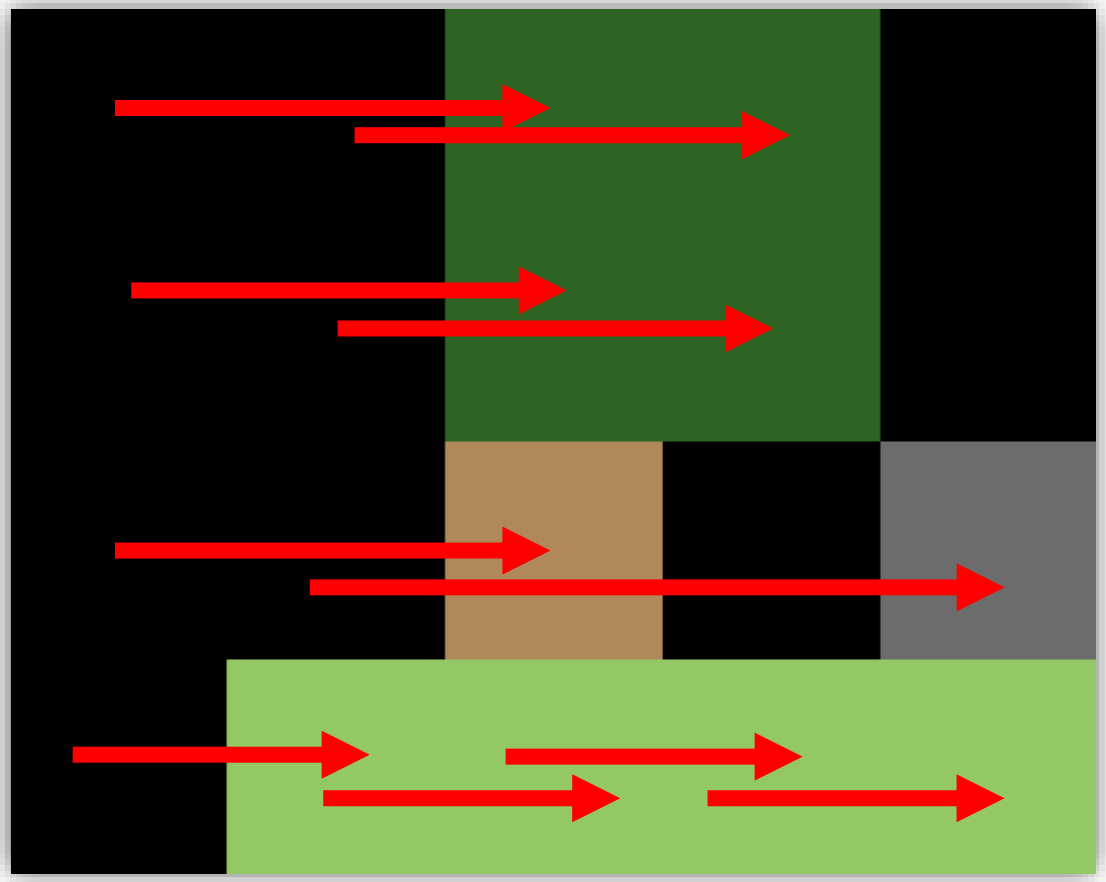


5x4 pixel disparity

Mini test example (Forward warp)

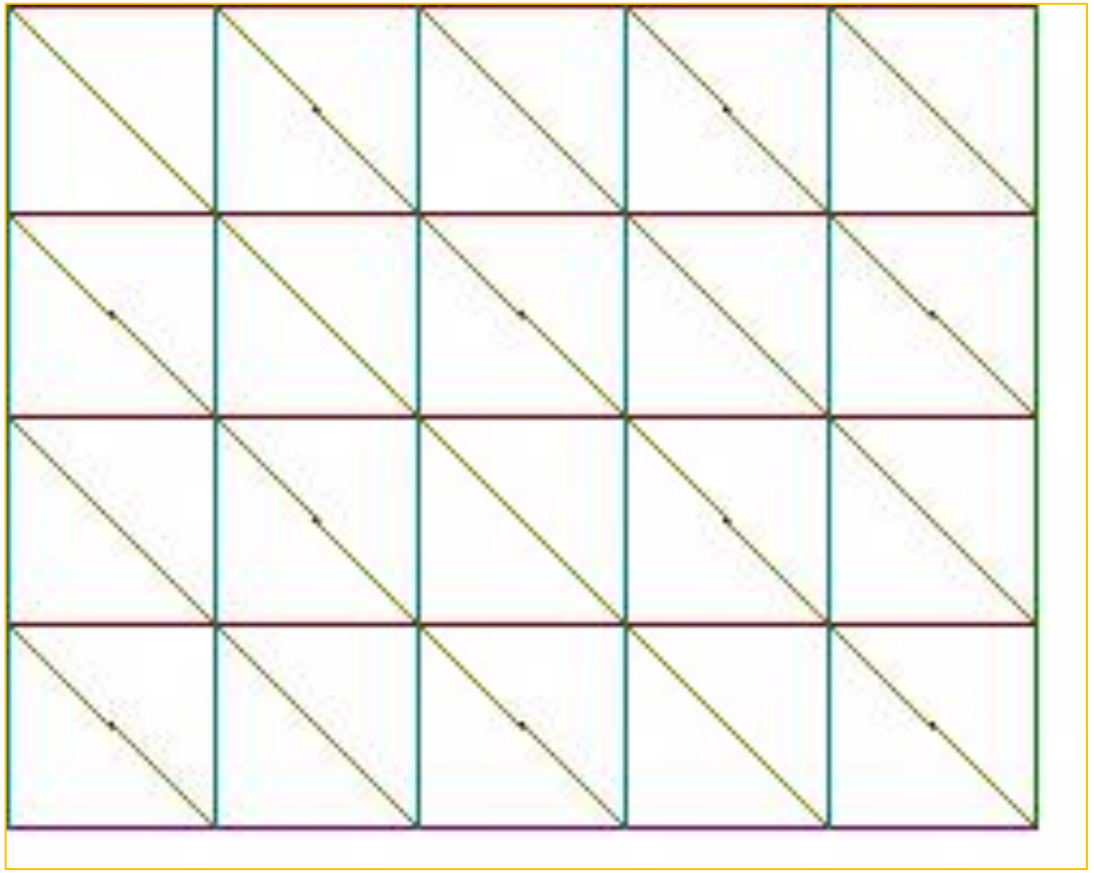


Input image

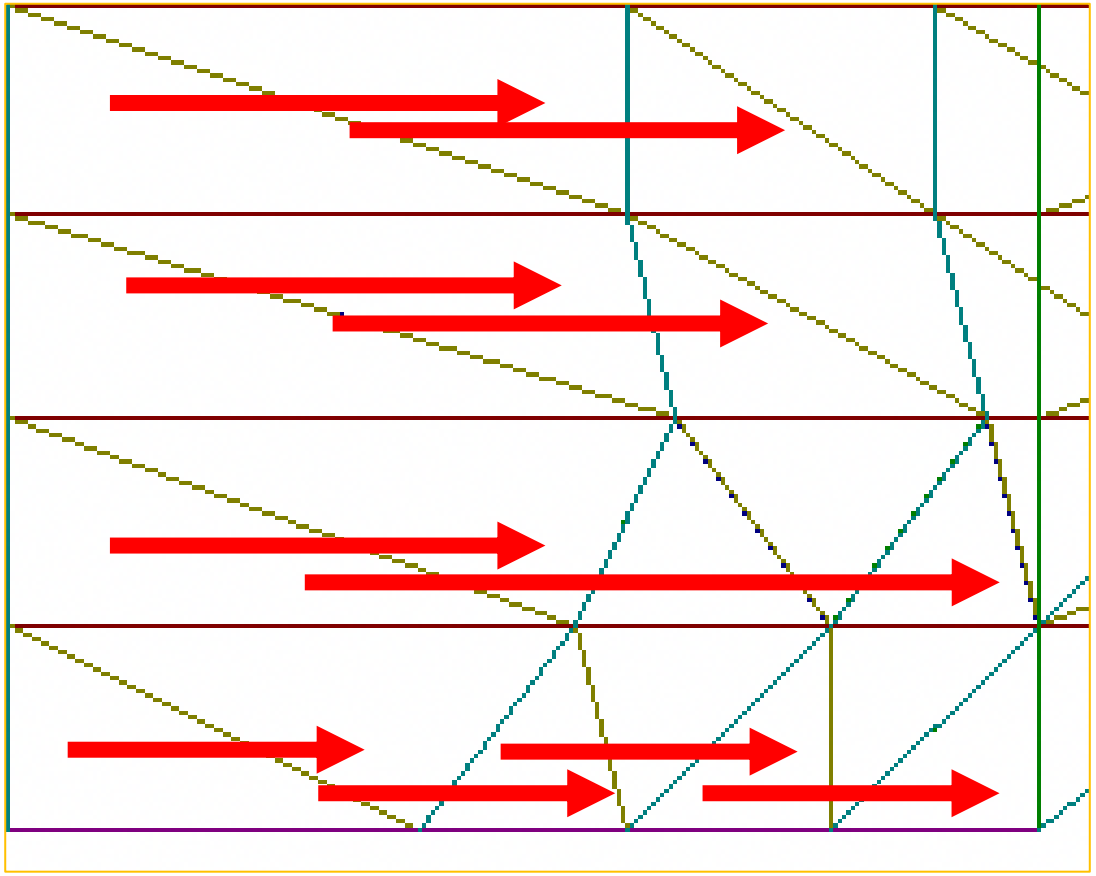


Forward warp

Mini test example (Warping grid)



Src grid



Dst grid

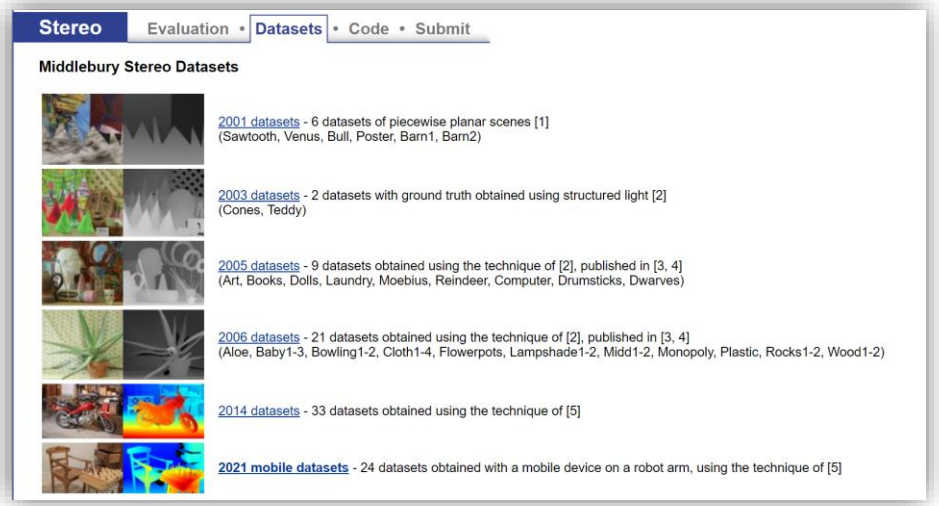
Additional RGBD Images

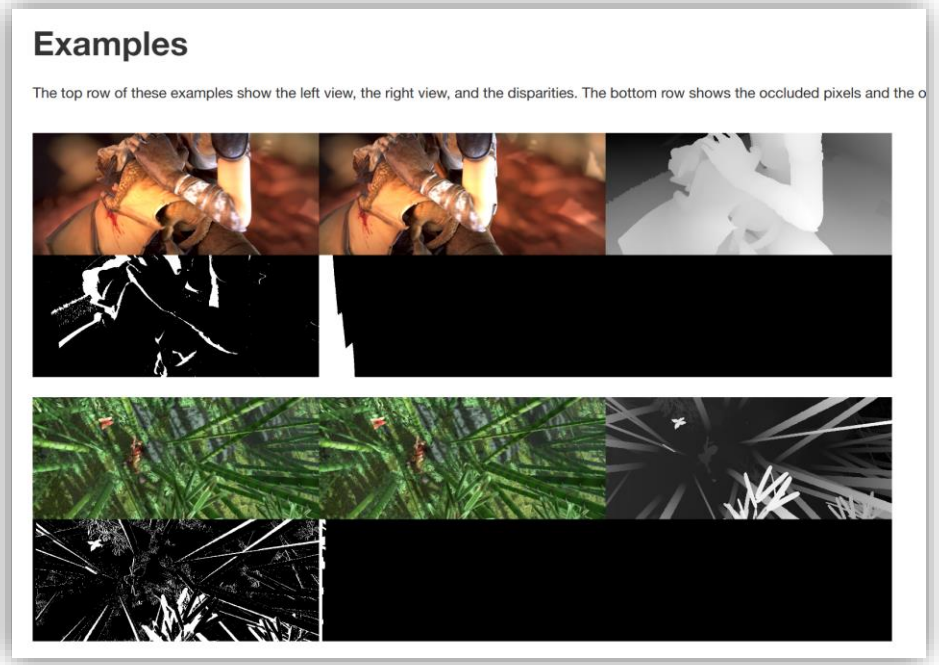
- More RGBD data provided in the **Middlebury Stereo Dataset**:
  - https://vision.middlebury.edu/stereo/data/
  - Contains also ground truth images.

Scharstein, Daniel, and Chris Pal. "Learning conditional random fields for stereo." CVPR 2007.

- Lot more data online..
  - E.g., the **Sintel** synthetic dataset: http://sintel.is.tue.mpg.de/stereo

Butler, Daniel J., et al. "A naturalistic open-source movie for optical flow evaluation." ECCV 2012.





Submission

- **Deadline**: Sunday Oct 29, 2023 (23:59 local time)

- Submit `your_code_here.h` to:
  Brightspace->Assignments->Assignment 3: Mesh-based Warping

- **Late submissions** can be submitted to “Assignment 3 – Late Submissions” with a penalty:

$$\text{adjusted grade} = \text{grade} - 1 - \left\lceil \frac{\text{minutes late}}{10} \right\rceil$$