8.3 2D Discrete Fourier Transformation (30 + 5 + 15 = 50 Points)

Fourier transformations can be used to transform periodical signals to the frequency domain where the frequency distribution of the signal can be seen easily.
The Discrete Fourier Transformation (DFT) can be used to transform images (which are discretized 2D functions) to Fourier space. DFTed images reveal the distributions and directions of the primary frequencies in images and are therefore an important tool in image analysis.

The task of this exercise is to implement the DFT and the Inverse Discrete Fourier Transformation (IDFT) as parts of a pipeline which transforms an image from image space to Fourier space and back to the (hopefully) original image. (See Figure 3)

a) Implement the 2D Discrete Fourier Transformation:

```
Texture DiscreteFourierTransform::dft(const Texture& input, bool inverse=false)
```

It is given by

$$X_{u,v} = \frac{1}{\sqrt{W \cdot H}} \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} p_{x,y} \cdot e^{-i2\pi \frac{u}{W} x} \cdot e^{-i2\pi \frac{v}{H} y}. \tag{1}$$

X are complex Fourier coefficients, p are image pixel values and W and H are the image width and height. (x, y) is an image position while (u, v) indexes one Fourier coefficient.
Use `std::complex<float>` to represent complex numbers. Use small images for testing.

b) Implement the inverse 2D DFT. It is given by

$$p_{x,y} = \frac{1}{\sqrt{W \cdot H}} \sum_{u=0}^{W-1} \sum_{v=0}^{H-1} X_{u,v} \cdot e^{i2\pi \frac{u}{W} x} \cdot e^{i2\pi \frac{v}{H} y}. \tag{2}$$

As you can see, only the sign of the exponent changes. Adjust your algorithm to behave accordingly if `inverse` is `true`. Now, when you run the inverse DFT on the result of the DFT, you should get the original image back.

c) Separate the computation of $X_{u,v}$ into a loop over u and another one over v. You still have to loop over all x and y. This should reduce the complexity of your algorithm down from $O(n^4)$ to $O(n^3)$.

In the end, you should get the same results, but the computation should be noticeably faster.

Another (optional!) way to make things faster is using OpenMP for easy multi-threading. Simply put the following line of code in front of your outer `for`-loop(s):

```
#pragma omp parallel for
```

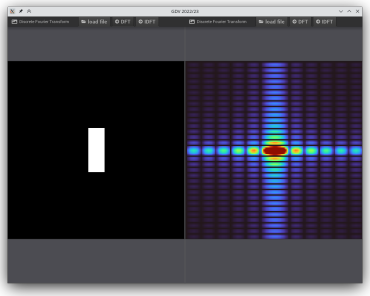

Figure 3: The discrete Fourier transformation of a rectangle. For the display, the Fourier transformed image is shifted by half its width and height. Also, the false color display shows the amplitude of the Fourier coefficients in logarithmic scale for better visibility.

8.4 Supersampling (15 + 15 = 30 Points)

A pixel actually corresponds to a square area. To calculate its value means to integrate over this area. It is not possible to solve these integrals analytically, so they have to be approximated numerically. This is done by computing weighted means of samples of the integrand. The actual image is reconstructed by sampling the integrand on the selected points.
The placement of the sample points greatly influences the result. The approximation error may result in random noise or in aliasing artifacts in the image.

For this exercise, implement the sampling strategies and test them with the given test functions.

Note: To show the GUI for this exercise, you have to disable the DFT GUI in the `main.cpp` file.

a) Implement the following test functions (see Figure 4) in

```
float SamplingPatterns::eval(Point2D pos) const
```

$$\text{Campbell-Robson:} \quad (x, y) \to \frac{1}{2}\left(1 + (1-y)^3 \cdot \sin(2\pi x e^{10x})\right) \tag{3}$$

$$\text{Smooth Zone Plate:} \quad (x, y) \to \frac{1}{2}\left(1 + \sin(1600 \cdot (x^2 + y^2))\right) \tag{4}$$

$$\text{Siemens Star:} \quad (x, y) \to \frac{1}{2}\left(1 + \sin\left(60 \cdot 4\pi \cdot \arctan\frac{x}{y}\right)\right) \tag{5}$$

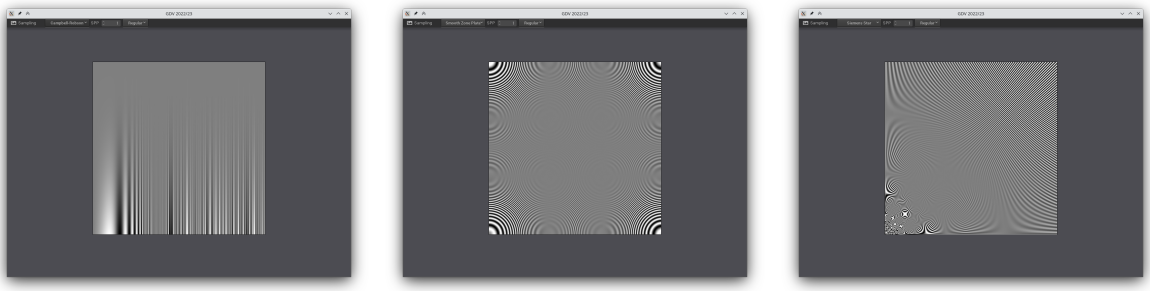

Figure 4: The three test functions regularly sampled with 1 sample per pixel at 512x512 pixels resolution. There are still some severe aliasing artifacts visible here. (Your PDF-viewer might add even more aliasing.) Here, the view shows the texture upside-down, such that $(0, 0)$ is in the bottom-left corner.

b) Place samples according to the following strategies in

```
std::vector<Point2D> SamplingPatterns::generateSamplePosition(uint32_t
    numSamplesPerDim) const
```

Here, $m =$ `numSamplesPerDim`, i.e. $n =$ `numSamplesPerDim*numSamplesPerDim`.

- **Regular Sampling**: The pixel is subdivided into $n = m \times m$ equally sized regions, which are sampled in the middle:

$$(x, y) = \left(\frac{i + \frac{1}{2}}{m}, \frac{j + \frac{1}{2}}{m}\right)_{i,j=0}^{m-1}$$

- **Random Sampling**: The pixel is sampled by n randomly placed samples:

$$(x, y) = (\xi_{i,1}, \xi_{i,2})_{i=0}^{n-1}$$

where $\xi_i \in_r [0, 1)$.

- **Stratified Sampling**: This is a combination of regular and random sampling. One sample is randomly placed in each of the $n = m \times m$ regions with $\xi_i, \xi_j \in_r [0, 1)$:

$$(x, y) = \left(\frac{i + \xi_i}{m}, \frac{j + \xi_j}{m} \right)_{i,j=0}^{m-1}$$

You can use `static Point2D Sampler::randomSquare()` to produce a pair of random numbers or `static float Sampler::randomFloat()` to generate a single random number.