```
/*********************************************************************
Copyright (c) 2018 Anonymous author(s) of

Foveated Real-Time Path Tracing in Visual-Polar Space

*********************************************************************/

// In this file value has _uv ending if it is range is 0 ... 1 and
// _coords ending if it is range 0 ... resolution

#ifndef VISUAL_POLAR_CL
#define VISUAL_POLAR_CL

// PI was sometimes defined already and sometimes not
// -> own PI removed missign and double definition errors.
#define VISUAL_POLAR_PI 3.14159265358979323846f

#define BMFR_BLOCK_OFFSETS_COUNT 16
__constant int2 BMFR_BLOCK_OFFSETS[BMFR_BLOCK_OFFSETS_COUNT] = {
    (int2) {-14, -14},
    (int2) { 4,  -6},
    (int2) {-8,  14},
    (int2) { 8,  0},
    (int2) {-10, -8},
    (int2) { 2,  12},
    (int2) { 12, -12},
    (int2) {-10, 0},
    (int2) { 12, 14},
    (int2) {-8, -16},
    (int2) {6,   6},
    (int2) {-2,  -2},
    (int2) {6, -14},
    (int2) {-16,  12},
    (int2) {14,  -4},
    (int2) {-6,   4}
};

inline float compute_L(
    const float2 cartesian_gaze_uv,
    const int2 cartesian_resolution) {

    const float2 maximums = (float2) {
        fmax(cartesian_gaze_uv.x, 1.f - cartesian_gaze_uv.x),
            fmax(cartesian_gaze_uv.y, 1.f - cartesian_gaze_uv.y)
    };

    return length(maximums * convert_float2(cartesian_resolution));
}

inline int2 compute_visual_polar_resolution(
    const int2 cartesian_resolution,
    const float4 visual_polar_parameters) {
    return convert_int2_rte(convert_float2(cartesian_resolution) / visual_polar_parameters.x);
}

// This function tells the height of the triangular visual-polar image based on the rho.
// By using this function we can remove triangle shaped area from the visual-polar image.
inline float visual_polar_height_uv_staircase(
    float rho_uv,
    float4 visual_polar_parameters,
    bool bmfr, //latter parameters not used if this is false
    int2 resolution,
    int frame
) {

    float uv;

    const float block_size = 32.f;
    const int offset = 17 - BMFR_BLOCK_OFFSETS[frame % BMFR_BLOCK_OFFSETS_COUNT].x;
    if (bmfr) {
        rho_uv = ceil((rho_uv * resolution.x + offset) / block_size) * (block_size / resolution.x);
    }

    // visual acuity is done by both scaling and cutting
    float hori_scaler = visual_polar_parameters.z;
    float fove0_fovea_limit = 0.0965f * 2.36f;
    uv = rho_uv / (hori_scaler * fove0_fovea_limit);

    if (bmfr) {
        uv = ceil(uv * resolution.y / block_size) * (block_size / resolution.y);
    }

    float c = 32.f / 1440.f;
    return clamp(uv, c, 1.f);
}

inline float visual_polar_height_uv(
    float rho_uv,
    float4 visual_polar_parameters
) {
    return visual_polar_height_uv_staircase(
```

```
        rho_uv, visual_polar_parameters,
        false, (int2) { 0, 0 }, 0);
}

inline int visual_polar_height_staircase(
    int rho,
    int2 visual_polar_resolution,
    float4 visual_polar_parameters,
    bool bmfr, //latter parameters not used if this is false
    int frame
) {
    return convert_int_rte(visual_polar_height_uv_staircase(
        rho / convert_float(visual_polar_resolution.x),
        visual_polar_parameters, bmfr, visual_polar_resolution, frame
    ) * visual_polar_resolution.y);
}

inline int visual_polar_height(
    int rho,
    int2 visual_polar_resolution,
    float4 visual_polar_parameters
) {
    return visual_polar_height_staircase(rho, visual_polar_resolution,
        visual_polar_parameters, false, 0);
}

// This returns values needed for the sampling triangular visual-polar image
int2 bilinear_sampling_visual_polar_staircase( // returns first sample location in pixels
    int2* out_offsets,                         // returns the offset of each sample in pixels compared to first sample
    float* out_weights,                        // returns weight of each sample
    const float2 sample_uv,                    // wanted sample location in visual polar space 0 ... 1
    const float2 mipmap_resolution,            // resolution of the used mipmap level (can be same as the next parameter)
    const float2 visual_polar_resolution,      // resolution of the whole visual polar space.
    const float4 visual_polar_parameters,
    bool bmfr,
    int frame_number
) {

    float prev_frame_pixel_f_x = sample_uv.x * visual_polar_resolution.x;
    int prev_frame_pixel_x = convert_int_rtn(prev_frame_pixel_f_x);

    // In triangular height of different pixel columns can be different.
    float left_column_height = visual_polar_height_staircase(prev_frame_pixel_x + 0,
        convert_int2(visual_polar_resolution), visual_polar_parameters, bmfr, frame_number);
    float right_column_height = visual_polar_height_staircase(prev_frame_pixel_x + 1,
        convert_int2(visual_polar_resolution), visual_polar_parameters, bmfr, frame_number);

    prev_frame_pixel_f_x = sample_uv.x * mipmap_resolution.x;

    float prev_frame_pixel_f_left_y = sample_uv.y * convert_float(left_column_height) *
        (mipmap_resolution.y / visual_polar_resolution.y);
    float prev_frame_pixel_f_right_y = sample_uv.y * convert_float(right_column_height) *
        (mipmap_resolution.y / visual_polar_resolution.y);

    prev_frame_pixel_x = convert_int_rtn(prev_frame_pixel_f_x);
    int prev_frame_pixel_left_y = convert_int_rtn(prev_frame_pixel_f_left_y);
    int prev_frame_pixel_right_y = convert_int_rtn(prev_frame_pixel_f_right_y);

    out_offsets[0] = (int2) { 0, 0 };
    out_offsets[1] = (int2) { 1, prev_frame_pixel_right_y - prev_frame_pixel_left_y };
    out_offsets[2] = (int2) { 0, 1 };
    out_offsets[3] = (int2) { 1, prev_frame_pixel_right_y - prev_frame_pixel_left_y + 1 };

    float prev_pixel_fract_x = prev_frame_pixel_f_x - convert_float(prev_frame_pixel_x);
    float prev_pixel_fract_left_y = prev_frame_pixel_f_left_y - convert_float(prev_frame_pixel_left_y);
    float prev_pixel_fract_right_y = prev_frame_pixel_f_right_y - convert_float(prev_frame_pixel_right_y);

    out_weights[0] = (1.f - prev_pixel_fract_x) * (1.f - prev_pixel_fract_left_y);
    out_weights[1] = prev_pixel_fract_x         * (1.f - prev_pixel_fract_right_y);
    out_weights[2] = (1.f - prev_pixel_fract_x) * prev_pixel_fract_left_y;
    out_weights[3] = prev_pixel_fract_x         * prev_pixel_fract_right_y;

    return (int2) { prev_frame_pixel_x, prev_frame_pixel_left_y };
}

int2 bilinear_sampling_visual_polar(       // returns first sample location in pixels
    int2* out_offsets,                     // returns the offset of each sample in pixels compared to first sample
    float* out_weights,                    // returns weight of each sample
    const float2 sample_uv,                // wanted sample location in visual polar space 0 ... 1
    const float2 mipmap_resolution,        // resolution of the used mipmap level (can be same as the next parameter)
    const float2 visual_polar_resolution,  // resolution of the whole visual polar space.
    const float4 visual_polar_parameters
) {
    return bilinear_sampling_visual_polar_staircase(out_offsets, out_weights, sample_uv,
        mipmap_resolution, visual_polar_resolution, visual_polar_parameters, false, 0);
}


inline int2 compute_cartesian_resolution(
    const int2 visual_polar_resolution,
    const float4 visual_polar_parameters) {
```

```
    return convert_int2_rte(convert_float2(visual_polar_resolution) * visual_polar_parameters.x);
}

float2 cartesian_to_visual_polar(         // return value 0 ... 1
    const float2 cartesian_uv,            // 0 ... 1
    const int2 visual_polar_resolution,   // in pixels
    const float4 visual_polar_parameters, // .x = sigma and .y = alpha
    const float2 cartesian_gaze_uv        // 0 ... 1
) {
    const int2 cartesian_resolution = compute_cartesian_resolution(
        visual_polar_resolution, visual_polar_parameters);

    float2 coords = (cartesian_uv - cartesian_gaze_uv)
        * convert_float2(cartesian_resolution);
    float rho;
    float hori_scaler = visual_polar_parameters.z;
    float vert_scaler = visual_polar_parameters.w;
    float distance_to_gaze_uv = length(coords) / compute_L(cartesian_gaze_uv, cartesian_resolution);

    float inverse_limit = fabs(visual_polar_parameters.y);
    bool show_fovea = 0.f > visual_polar_parameters.y;
    float fove0_fovea_limit = 0.0965f * 2.36f;

    float x = (1.f / vert_scaler) * distance_to_gaze_uv;
    if (distance_to_gaze_uv < inverse_limit)
    {
        if (show_fovea)
        {
            rho = 0.f;
        }
        else
        {
            rho = (fove0_fovea_limit / inverse_limit) * distance_to_gaze_uv * hori_scaler;
        }
    }
    else
    {
        rho = hori_scaler * -0.135763314321855f;
        float m = x;
        rho += hori_scaler * 3.825446322211148f * m;
        m *= x;
        rho += hori_scaler * -9.907847134876771f * m;
        m *= x;
        rho += hori_scaler * 21.875941577232023f * m;
        m *= x;
        rho += hori_scaler * -33.788446055810360f * m;
        m *= x;
        rho += hori_scaler * 33.004110465035240f * m;
        m *= x;
        rho += hori_scaler * -18.071744663268458f * m;
        m *= x;
        rho += hori_scaler * 4.198584275009534f * m;
    }


    float adder = VISUAL_POLAR_PI;
    if (coords.x >= 0) {
        adder = coords.y <= 0.f ? 2.f * VISUAL_POLAR_PI : 0.f;
    }
    const float phi = (atan(coords.y / coords.x) + adder) / (2.f * VISUAL_POLAR_PI);
    return (float2) { rho, phi };

}

float2 visual_polar_to_cartesion(         // returns value 0 ... 1
    const float2 visual_polar_uv,         // 0 ... 1
    const int2 visual_polar_resolution,   // In pixels
    const float4 visual_polar_parameters, // .x = sigma and .y = alpha
    const float2 cartesian_gaze_uv        // 0 ... 1
)
{
    const int2 cartesian_resolution = compute_cartesian_resolution(
        visual_polar_resolution, visual_polar_parameters);

    float2 screen_coords;
    float hori_scaler = visual_polar_parameters.z;
    float vert_scaler = visual_polar_parameters.w;
    float inverse_limit = fabs(visual_polar_parameters.y);


    float fove0_fovea_limit = 0.0965f * 2.36f;
    float limit = hori_scaler * fove0_fovea_limit;


    float rho = (1.f / hori_scaler) * visual_polar_uv.x;

    if (visual_polar_uv.x < limit)
    {
        screen_coords = (inverse_limit / fove0_fovea_limit) * rho;
    }
    else
```

```
    {
        float u = vert_scaler * 0.0404f;
        float m = rho;
        u += vert_scaler * 0.2984f * m;
        m *= rho;
        u += vert_scaler * 0.3451f * m;
        m *= rho;
        u += vert_scaler * 0.0021f * m;
        m *= rho;
        u += vert_scaler * 0.3136f * m;
        screen_coords = u;
    }

    screen_coords *= compute_L(cartesian_gaze_uv, cartesian_resolution);

    const float B = 2.f * VISUAL_POLAR_PI;
    const float C = B * visual_polar_uv.y;
    screen_coords.x *= cos(C);
    screen_coords.y *= sin(C);

    float2 screen_uv = screen_coords / convert_float2(cartesian_resolution)
        + cartesian_gaze_uv;
    return screen_uv;
}


#endif // VISUAL_POLAR_CL
```