

```
1  /*****
2  Copyright (c) 2018 Anonymous author(s) of
3
4  Foveated Real-Time Path Tracing in Visual-Polar Space
5
6  *****/
7
8  // In this file value has _uv ending if it is range is 0 ... 1 and
9  // _coords ending if it is range 0 ... resolution
10
11 #ifndef VISUAL_POLAR_CL
12 #define VISUAL_POLAR_CL
13
14 // PI was sometimes defined already and sometimes not
15 // -> own PI removed missign and double definition errors.
16 #define VISUAL_POLAR_PI 3.14159265358979323846f
17
18 #define BMFR_BLOCK_OFFSETS_COUNT 16
19 __constant int2 BMFR_BLOCK_OFFSETS[BMFR_BLOCK_OFFSETS_COUNT] = {
20     (int2) {-14, -14},
21     (int2) { 4, -6},
22     (int2) {-8, 14},
23     (int2) { 8, 0},
24     (int2) {-10, -8},
25     (int2) { 2, 12},
26     (int2) { 12, -12},
27     (int2) {-10, 0},
28     (int2) { 12, 14},
29     (int2) {-8, -16},
30     (int2) {6, 6},
31     (int2) {-2, -2},
32     (int2) {6, -14},
33     (int2) {-16, 12},
34     (int2) {14, -4},
35     (int2) {-6, 4}
36 };
37
38 inline float compute_l(
39     const float2 cartesian_gaze_uv,
40     const int2 cartesian_resolution) {
41
42     const float2 maximums = (float2) {
43         fmax(cartesian_gaze_uv.x, 1.f - cartesian_gaze_uv.x),
44         fmax(cartesian_gaze_uv.y, 1.f - cartesian_gaze_uv.y)
45     };
46
47     return length(maximums * convert_float2(cartesian_resolution));
48 }
49
50 inline int2 compute_visual_polar_resolution(
51     const int2 cartesian_resolution,
52     const float4 visual_polar_parameters) {
53     return convert_int2_rte(convert_float2(cartesian_resolution) / visual_polar_parameters.x);
54 }
55
56 // This function tells the height of the triangular visual-polar image based on the rho.
57 // By using this function we can remove triangle shaped area from the visual-polar image.
58 inline float visual_polar_height_uv_staircase(
59     float rho_uv,
60     float4 visual_polar_parameters,
61     bool bmfr, //latter parameters not used if this is false
62     int2 resolution,
63     int frame
64 ) {
65
66     float uv;
67
68     const float block_size = 32.f;
69     const int offset = 17 - BMFR_BLOCK_OFFSETS[frame % BMFR_BLOCK_OFFSETS_COUNT].x;
70     if (bmfr) {
71         rho_uv = ceil((rho_uv * resolution.x + offset) / block_size) * (block_size / resolution.x);
72     }
73
74     // visual acuity is done by both scaling and cutting
75     float hori_scaler = visual_polar_parameters.z;
76     float fovea_fovea_limit = 0.0965f * 2.36f;
77     uv = rho_uv / (hori_scaler * fovea_fovea_limit);
78
79     if (bmfr) {
80         uv = ceil(uv * resolution.y / block_size) * (block_size / resolution.y);
81     }
82
83     float c = 32.f / 1440.f;
84     return clamp(uv, c, 1.f);
85 }
86
87 inline float visual_polar_height_uv(
88     float rho_uv,
89     float4 visual_polar_parameters
90 ) {
91     return visual_polar_height_uv_staircase(
```

```

92     rho_uv, visual_polar_parameters,
93     false, (int2) { 0, 0 }, 0);
94 }
95
96 inline int visual_polar_height_staircase(
97     int rho,
98     int2 visual_polar_resolution,
99     float4 visual_polar_parameters,
100    bool bmfr, //latter parameters not used if this is false
101    int frame
102 ) {
103     return convert_int_rte(visual_polar_height_uv_staircase(
104         rho / convert_float(visual_polar_resolution.x),
105         visual_polar_parameters, bmfr, visual_polar_resolution, frame
106     ) * visual_polar_resolution.y);
107 }
108
109 inline int visual_polar_height(
110     int rho,
111     int2 visual_polar_resolution,
112     float4 visual_polar_parameters
113 ) {
114     return visual_polar_height_staircase(rho, visual_polar_resolution,
115         visual_polar_parameters, false, 0);
116 }
117
118 // This returns values needed for the sampling triangular visual-polar image
119 int2 bilinear_sampling_visual_polar_staircase( // returns first sample location in pixels
120     int2* out_offsets, // returns the offset of each sample in pixels compared to first sample
121     float* out_weights, // returns weight of each sample
122     const float2 sample_uv, // wanted sample location in visual polar space 0 ... 1
123     const float2 mipmap_resolution, // resolution of the used mipmap level (can be same as the next parameter)
124     const float2 visual_polar_resolution, // resolution of the whole visual polar space.
125     const float4 visual_polar_parameters,
126     bool bmfr,
127     int frame_number
128 ) {
129
130     float prev_frame_pixel_f_x = sample_uv.x * visual_polar_resolution.x;
131     int prev_frame_pixel_x = convert_int_rtn(prev_frame_pixel_f_x);
132
133     // In triangular height of different pixel columns can be different.
134     float left_column_height = visual_polar_height_staircase(prev_frame_pixel_x + 0,
135         convert_int2(visual_polar_resolution), visual_polar_parameters, bmfr, frame_number);
136     float right_column_height = visual_polar_height_staircase(prev_frame_pixel_x + 1,
137         convert_int2(visual_polar_resolution), visual_polar_parameters, bmfr, frame_number);
138
139     prev_frame_pixel_f_x = sample_uv.x * mipmap_resolution.x;
140
141     float prev_frame_pixel_f_left_y = sample_uv.y * convert_float(left_column_height) *
142         (mipmap_resolution.y / visual_polar_resolution.y);
143     float prev_frame_pixel_f_right_y = sample_uv.y * convert_float(right_column_height) *
144         (mipmap_resolution.y / visual_polar_resolution.y);
145
146     prev_frame_pixel_x = convert_int_rtn(prev_frame_pixel_f_x);
147     int prev_frame_pixel_left_y = convert_int_rtn(prev_frame_pixel_f_left_y);
148     int prev_frame_pixel_right_y = convert_int_rtn(prev_frame_pixel_f_right_y);
149
150     out_offsets[0] = (int2) { 0, 0 };
151     out_offsets[1] = (int2) { 1, prev_frame_pixel_right_y - prev_frame_pixel_left_y };
152     out_offsets[2] = (int2) { 0, 1 };
153     out_offsets[3] = (int2) { 1, prev_frame_pixel_right_y - prev_frame_pixel_left_y + 1 };
154
155     float prev_pixel_fract_x = prev_frame_pixel_f_x - convert_float(prev_frame_pixel_x);
156     float prev_pixel_fract_left_y = prev_frame_pixel_f_left_y - convert_float(prev_frame_pixel_left_y);
157     float prev_pixel_fract_right_y = prev_frame_pixel_f_right_y - convert_float(prev_frame_pixel_right_y);
158
159     out_weights[0] = (1.f - prev_pixel_fract_x) * (1.f - prev_pixel_fract_left_y);
160     out_weights[1] = prev_pixel_fract_x * (1.f - prev_pixel_fract_right_y);
161     out_weights[2] = (1.f - prev_pixel_fract_x) * prev_pixel_fract_left_y;
162     out_weights[3] = prev_pixel_fract_x * prev_pixel_fract_right_y;
163
164     return (int2) { prev_frame_pixel_x, prev_frame_pixel_left_y };
165 }
166
167 int2 bilinear_sampling_visual_polar( // returns first sample location in pixels
168     int2* out_offsets, // returns the offset of each sample in pixels compared to first sample
169     float* out_weights, // returns weight of each sample
170     const float2 sample_uv, // wanted sample location in visual polar space 0 ... 1
171     const float2 mipmap_resolution, // resolution of the used mipmap level (can be same as the next parameter)
172     const float2 visual_polar_resolution, // resolution of the whole visual polar space.
173     const float4 visual_polar_parameters
174 ) {
175     return bilinear_sampling_visual_polar_staircase(out_offsets, out_weights, sample_uv,
176         mipmap_resolution, visual_polar_resolution, visual_polar_parameters, false, 0);
177 }
178
179
180 inline int2 compute_cartesian_resolution(
181     const int2 visual_polar_resolution,
182     const float4 visual_polar_parameters) {

```

```

183     return convert_int2_rte(convert_float2(visual_polar_resolution) * visual_polar_parameters.x);
184 }
185
186 float2 cartesian_to_visual_polar(           // return value 0 ... 1
187     const float2 cartesian_uv,             // 0 ... 1
188     const int2 visual_polar_resolution,    // in pixels
189     const float4 visual_polar_parameters, // .x = sigma and .y = alpha
190     const float2 cartesian_gaze_uv        // 0 ... 1
191 ) {
192     const int2 cartesian_resolution = compute_cartesian_resolution(
193         visual_polar_resolution, visual_polar_parameters);
194
195     float2 coords = (cartesian_uv - cartesian_gaze_uv)
196         * convert_float2(cartesian_resolution);
197     float rho;
198     float hori_scaler = visual_polar_parameters.z;
199     float vert_scaler = visual_polar_parameters.w;
200     float distance_to_gaze_uv = length(coords) / compute_L(cartesian_gaze_uv, cartesian_resolution);
201
202     float inverse_limit = fabs(visual_polar_parameters.y);
203     bool show_fovea = 0.f > visual_polar_parameters.y;
204     float fove0_fovea_limit = 0.0965f * 2.36f;
205
206     float x = (1.f / vert_scaler) * distance_to_gaze_uv;
207     if (distance_to_gaze_uv < inverse_limit)
208     {
209         if (show_fovea)
210         {
211             rho = 0.f;
212         }
213         else
214         {
215             rho = (fove0_fovea_limit / inverse_limit) * distance_to_gaze_uv * hori_scaler;
216         }
217     }
218     else
219     {
220         rho = hori_scaler * -0.135763314321855f;
221         float m = x;
222         rho += hori_scaler * 3.825446322211148f * m;
223         m *= x;
224         rho += hori_scaler * -9.907847134876771f * m;
225         m *= x;
226         rho += hori_scaler * 21.875941577232023f * m;
227         m *= x;
228         rho += hori_scaler * -33.788446055810360f * m;
229         m *= x;
230         rho += hori_scaler * 33.004110465035240f * m;
231         m *= x;
232         rho += hori_scaler * -18.071744663268458f * m;
233         m *= x;
234         rho += hori_scaler * 4.198584275009534f * m;
235     }
236
237
238     float adder = VISUAL_POLAR_PI;
239     if (coords.x >= 0) {
240         adder = coords.y <= 0.f ? 2.f * VISUAL_POLAR_PI : 0.f;
241     }
242     const float phi = (atan(coords.y / coords.x) + adder) / (2.f * VISUAL_POLAR_PI);
243     return (float2) { rho, phi };
244 }
245
246 float2 visual_polar_to_cartesion(           // returns value 0 ... 1
247     const float2 visual_polar_uv,         // 0 ... 1
248     const int2 visual_polar_resolution,    // In pixels
249     const float4 visual_polar_parameters, // .x = sigma and .y = alpha
250     const float2 cartesian_gaze_uv        // 0 ... 1
251 ) {
252 }
253 {
254     const int2 cartesian_resolution = compute_cartesian_resolution(
255         visual_polar_resolution, visual_polar_parameters);
256
257     float2 screen_coords;
258     float hori_scaler = visual_polar_parameters.z;
259     float vert_scaler = visual_polar_parameters.w;
260     float inverse_limit = fabs(visual_polar_parameters.y);
261
262
263     float fove0_fovea_limit = 0.0965f * 2.36f;
264     float limit = hori_scaler * fove0_fovea_limit;
265
266
267     float rho = (1.f / hori_scaler) * visual_polar_uv.x;
268
269     if (visual_polar_uv.x < limit)
270     {
271         screen_coords = (inverse_limit / fove0_fovea_limit) * rho;
272     }
273     else

```

```
274     {
275         float u = vert_scaler * 0.0404f;
276         float m = rho;
277         u += vert_scaler * 0.2984f * m;
278         m *= rho;
279         u += vert_scaler * 0.3451f * m;
280         m *= rho;
281         u += vert_scaler * 0.0021f * m;
282         m *= rho;
283         u += vert_scaler * 0.3136f * m;
284         screen_coords = u;
285     }
286
287     screen_coords *= compute_L(cartesian_gaze_uv, cartesian_resolution);
288
289     const float B = 2.f * VISUAL_POLAR_PI;
290     const float C = B * visual_polar_uv.y;
291     screen_coords.x *= cos(C);
292     screen_coords.y *= sin(C);
293
294     float2 screen_uv = screen_coords / convert_float2(cartesian_resolution)
295         + cartesian_gaze_uv;
296     return screen_uv;
297 }
298
299
300 #endif // VISUAL_POLAR_CL
```