Multi-Layer Depth of Field Rendering with Tiled Splatting

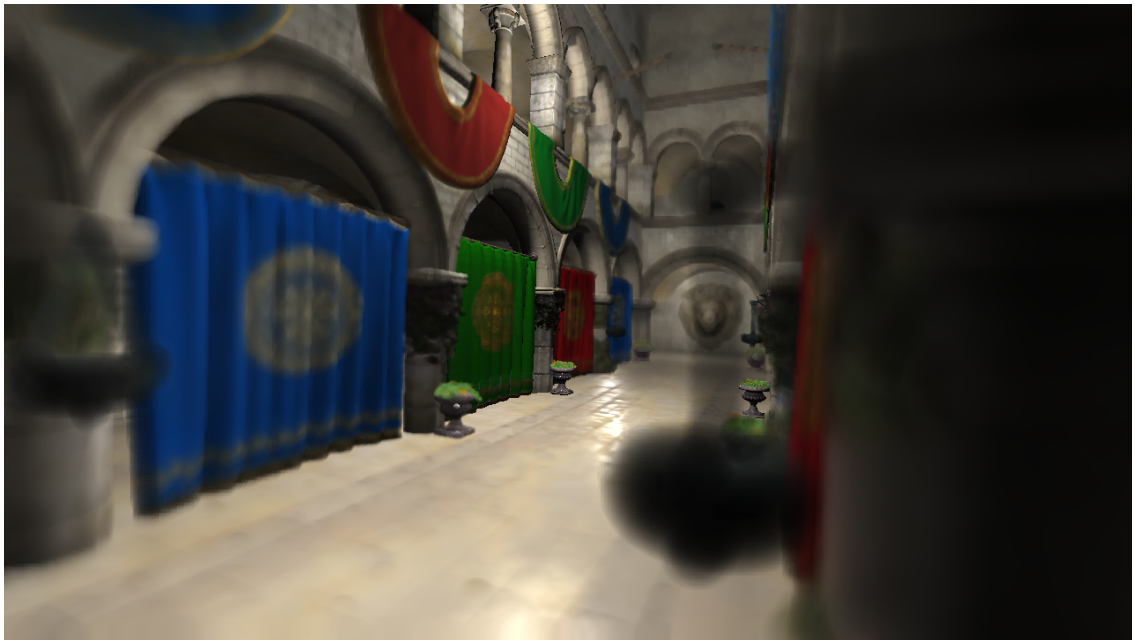
Supplemental Material

Linus Franke Nikolai Hofmann Marc Stamminger Kai Selgrad
Computer Graphics Group, University of Erlangen-Nuremberg

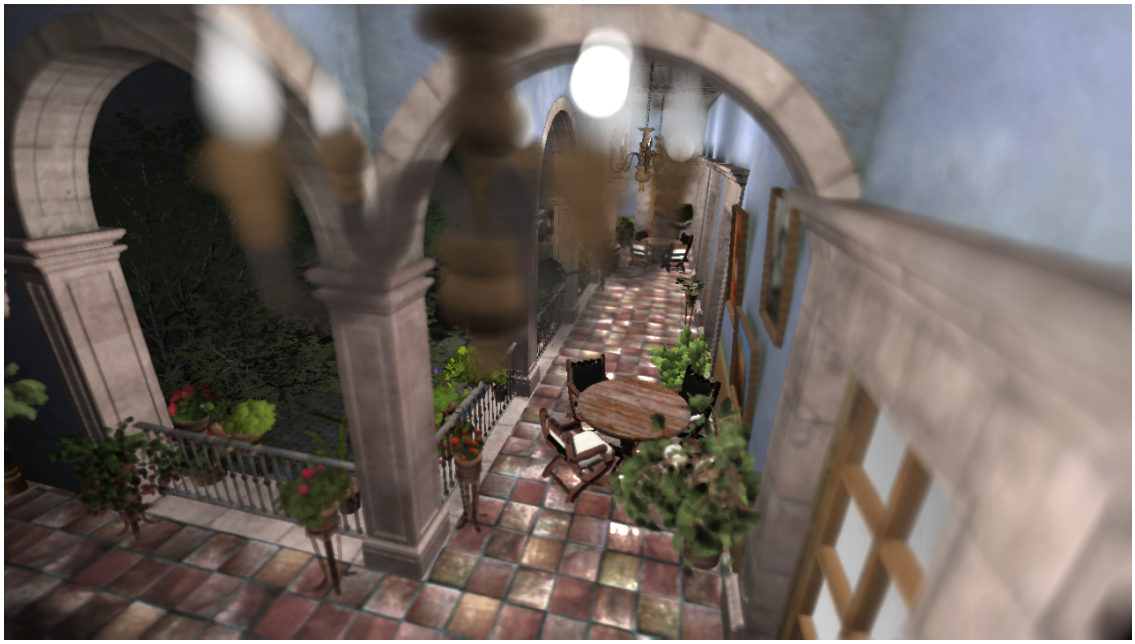
Contents

1 Larger Screenshots

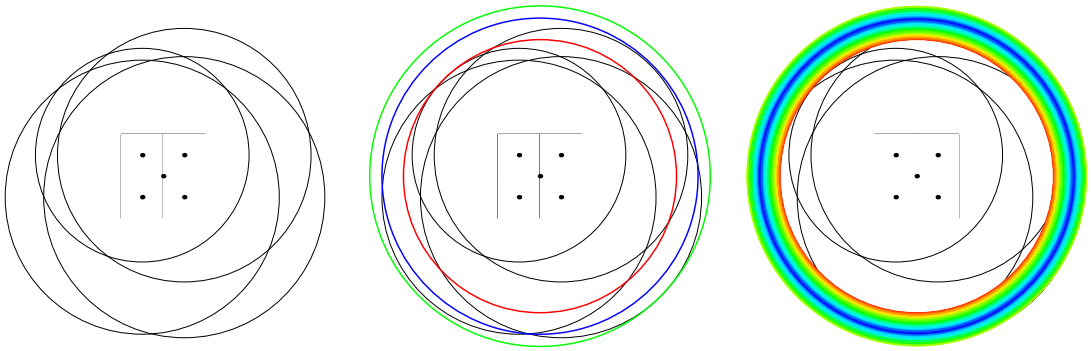
Times for Sponza refer to this view.



Times for San Miguel refer to these views.



2 Fragment Merging Error



On the left, four fragments with high variance in Coc-radii are shown. Possible radii for the merged fragment are shown in the middle. The Coc when taking the minimum of the merged radii is painted red, the maximum in green and the circular average (see paper) in blue. On the right, the resulting errors are encoded as a heatmap, showing the strength of the circular average and that the max-approximation is still quite robust in comparison to the the min-approximation.

3 Code

3.1 Multilayer Collection Pass

```
//GLSL 4.3 fragment shader
    uniform ivec2 screendim;
    uniform vec2 cam_near_far;
    uniform mat4 view, proj;
    uniform mat4 last_view, last_proj;
    uniform float aperture;
    uniform float npc;
    uniform float focus_plane;

    const mat3 inv_view = inverse(mat3(view));
    const mat3 normal_matrix = transpose(inv_view);
    const float inv_far = 1.f / (cam_near_far.y - cam_near_far.x);

    uniform layout(binding=22) sampler2D edges_back;
    uniform layout(binding = 1, r32i) restrict iimage2D counting_tex_out;

    in vec4 pos_wc;
    in vec3 norm_wc;

    out vec4 out_col;

    float umbraDistance(float lastDepth){
        float s = (1.0f/npc)*lastDepth*cam_near_far.x;
        float dis = (lastDepth*s)/(aperture-s);
        if(dis<=0.0) dis = 20000.0;
        dis = min(20000.0, dis);
        return lastDepth + dis;
    }
    void main() {
      if (mask() <= 0.2){
        discard; // alpha discard
      }

      // project to last frame
      vec4 tmp = last_proj * last_view * pos_wc;
      tmp /= tmp.w;
      vec2 last_frame = tmp.xy*0.5 + 0.5;

      //discard via depth discont
      float edge_bool = texelFetch(edges_back, ivec2(last_frame*screendim), 0).r;

      if(gl_Layer>=1 && edge_bool < 0.8){
        discard;
        return;
      }

      vec4 pos_ec = view * pos_wc;
      float depth = (-pos_ec.z) ;
      float lastDepth = linearize_depth(texelFetch(back_depth, ivec3(last_frame*screendim, gl_Layer-1),0).r);

      float umbra_dis = umbraDistance(lastDepth);

      // Layer discard based on umbra shadow of fragments
      if (gl_Layer >= 1){
        if(depth < umbra_dis){
          discard;
          return;
        }
      }

      //fragment stays
      imageAtomicMax(counting_tex_out, ivec2(gl_FragCoord.xy), gl_Layer);
      vec4 color_for_store = vec4(1,0,0,1);
      {
        vec3 norm_ec = normalize(normal_matrix * norm_wc);
        norm_ec = normalize(normal(norm_ec, pos_ec.xyz));
        vec4 spec_col = specular();
        float reflectivity = clamp((spec_col.r+spec_col.g+spec_col.b) / 3.f, 0.f, 1.f);
        vec4 col = diffuse();
        col.a = reflectivity;

        // shading
        color_for_store = vec4(pow(shade(pos_ec.xyz, norm_ec, col), vec3(1.f)), depth*inv_far);
      }
      out_col = color_for_store;
    }
```

3.2 Depth Discontinuity Mask Generation

```
//GLSL 4.3 compute shader
    uniform layout(binding = 0) sampler2DArray depth_texture;
    uniform layout(binding = 1, r8) restrict image2D edges_out;
    uniform layout(location = 0) ivec2 wh;
    uniform layout(location = 2) float eye_to_lens;
    uniform layout(location = 3) float max_coc;
    uniform layout(location = 4) vec2 linear_coc;
    uniform layout(location = 6) vec2 near_far;
    uniform layout(location = 8) float focus_depth;

#define BLOCK_SIZE 16
#define TILE_SPREAD 1

    shared float depths[(BLOCK_SIZE+2)*(BLOCK_SIZE+2)]; //16*16 blocks -> 18*18 pixels to look at
    shared bool needML_bool[BLOCK_SIZE*BLOCK_SIZE*9];

    int d_access(int x, int y){
      return ((x+1)+(y+1)*(BLOCK_SIZE+2));
    }
    int need_access(int x, int y){
      return ((x+BLOCK_SIZE)+(y+BLOCK_SIZE)*(BLOCK_SIZE*3));
    }
    void reset_need(int x, int y){
      needML_bool[(x+BLOCK_SIZE)+(y+BLOCK_SIZE)*(BLOCK_SIZE*3)] = false;
    }

    void set_need(int x, int y){
      needML_bool[(x+BLOCK_SIZE)+(y+BLOCK_SIZE)*(BLOCK_SIZE*3)] = true;
    }

    bool get_need(int x, int y){
      return needML_bool[(x+BLOCK_SIZE)+(y+BLOCK_SIZE)*(BLOCK_SIZE*3)];
    }

    float linearize_depth(float non_lin_depth){
      float z = 2.0 * non_lin_depth - 1.0;
      float z_lin = (2.0*near_far.y*near_far.x) /(near_far.y+near_far.x-z*(near_far.y-near_far.x));
      return z_lin;
    }

    void main() {
      ivec2 gid = ivec2(gl_GlobalInvocationID.xy);
      if (gid.x >= wh.x || gid.y >= wh.y) return;

      ivec2 tidInBlock = ivec2(gl_LocalInvocationID.xy);
      int tid = int(tidInBlock.x+tidInBlock.y*16);
      for(int i=-1; i<=1; ++i){
        for(int j=-1; j<=1; ++j){
          ivec2 index = tidInBlock + ivec2(j,i)*BLOCK_SIZE;
          reset_need(index.x,index.y);
        }
      }
      const float EPSILON = 20.0;
      const float factor = 1.0;

      depths[d_access(tidInBlock.x, tidInBlock.y)] = linearize_depth(texelFetch(depth_texture,ivec3(gid,0),0).r)*factor;

      //seam:
      ivec2 base = ivec2(gid/BLOCK_SIZE) * BLOCK_SIZE;
      int x_ind = 0;
      int y_ind = 0;
      //first warp only
      if(tid<32) {
        if(tid<16){ x_ind = -1; y_ind = tid;}
        if(tid>=16){x_ind = 16; y_ind = tid-16; }
      }else if(tid<64) {
        //second warp
        if(tid<48){ y_ind = -1;x_ind = tid-32;}
        if(tid>=48){y_ind = 16;x_ind = tid-48;}
      }else if(tid<68){
        // threads of the third warp
        if(tid==64){x_ind = -1;y_ind = -1;}
        if(tid==65){x_ind = -1;y_ind = 16;}
        if(tid==66){x_ind = 16;y_ind = -1;}
        if(tid==67){x_ind = 16;y_ind = 16;}
      }

      if(tid<68)
        depths[d_access(x_ind, y_ind)] =
              linearize_depth(texelFetch(depth_texture,ivec3(base.x+x_ind,base.y+y_ind,0),0).r)*factor;

      int xLow = -1;
      int xHigh = 1;
      int yLow = -1;
      int yHigh = 1;
      if(gid.x==0) xLow = 0;
      if(gid.y==0) yLow = 0;
      if(gid.x>=(wh.x-1)) xHigh = 0;
      if(gid.y>=(wh.y-1)) yHigh = 0;

      float refDepth = depths[d_access(tidInBlock.x,tidInBlock.y)];
      bool area = true;

      barrier();

      for(int i=yLow; i<=yHigh; i++){
        for(int j=xLow; j<=xHigh; j++){
          float depth = depths[d_access(tidInBlock.x+j,tidInBlock.y+i)];
          float depth_dis = depth-refDepth;
          if(abs(depth_dis)>EPSILON) area = false;
        }
      }
```

```
      if(!area)
      {

        //get coc
        float dist = refDepth-eye_to_lens;
        float coc = (linear_coc.x / dist + linear_coc.y);
        coc = min(abs(coc), max_coc);
        int cocRad = int(coc*0.5f)+1;
        //set area of coc to false
        xLow = -(min(gid.x,cocRad));
        xHigh = min(cocRad, wh.x-gid.x-1);
        yLow = -min(gid.y,cocRad);
        yHigh = min(cocRad, wh.y-gid.y-1);
        const float cocSq = (cocRad*cocRad);

        for(int i=yLow; i<=yHigh; i++){
          for(int j=xLow; j<=xHigh; j++){
//            if(fma(i,i,j*j)<cocSq) //better approximation, but much slower overall
            {
              ivec2 write_to_index = ivec2(gid.x+j,gid.y+i);
              set_need(write_to_index.x-base.x, write_to_index.y - base.y);

            }
          }
        }
      }
      barrier();

      for(int i=-1; i<=1; ++i){
        for(int j=-1; j<=1; ++j){
          ivec2 index = tidInBlock + ivec2(j,i)*ivec2(BLOCK_SIZE);
          ivec2 out_index = gid + ivec2(j,i)*ivec2(BLOCK_SIZE);
          if(get_need(index.x,index.y))
          {
            imageStore(edges_out, ivec2(out_index.x,out_index.y), vec4(1,0,0,0));

          }
        }
      }
    }
#undef BLOCK_SIZE
#undef TILE_SPREAD

  }
```

3.3 Merge 2x2

```
//CUDA 8.0 kernels, compute capability 6.1

__device__ static bool checkMerge2x2(float4* __restrict__ values, lensVariables lensVars, float* __restrict__ maxDep){
  const float COLOR_EPSILON = 0.05f;
  const float MERGE_EPSILON_BASE = 22000.f;
  const float MERGE_EPSILON = MERGE_EPSILON_BASE/max(2.0f,lensVars.aperture);
  const float LUMINANCE_EPSILON = 0.1f;
  const int DIM_X = 2, DIM_Y = 2, step = 2;

  bool luminance = true;
  for(int i=0; i<DIM_X*DIM_Y; i++){
    int otherColIndex = (i==(DIM_X*DIM_Y-1)) ? 0 : (i+1);

    if((abs(values[i].x*0.2126f + values[i].y*0.7152f + values[i].z*0.0722f)
        -(values[otherColIndex].x*0.2126f +values[otherColIndex].y*0.7152f + values[otherColIndex].z*0.0722f))
        > LUMINANCE_EPSILON)
    {
      luminance=false;
    }
  }

  bool cocMerge = false;
  float minDisToFocus=100000.f;
  for(int i=0; i<DIM_X*DIM_Y; i++){
    minDisToFocus = min(minDisToFocus, abs(abs(values[i].w)-lensVars.focus_plane));
  }

  float minDepth = 1000000.f;
  float maxDepth = 0.f;
  for(int i=0; i<DIM_X*DIM_Y;i++){
    minDepth = min(minDepth,abs(values[i].w));
    maxDepth = max(maxDepth,abs(values[i].w));
  }
  float maxDiffDepth = abs(maxDepth-minDepth);

  maxDiffDepth++; //little adjustment, div by zero
  float mergeFactor = (minDisToFocus*minDisToFocus)/(maxDiffDepth*step);
  if(mergeFactor > MERGE_EPSILON) cocMerge=true;

  maxDep[0] = maxDepth;
  if(cocMerge
      && luminance)
  {
    return true;
  } else{
      return false;
  }
}

#define BLOCK_DIM_X 20
#define BLOCK_DIM_Y 16

__global__ void kernelMerge2x2_ticket(int width2x2, int height2x2, int* __restrict__ scanned_counts_2x2, array2d<elemUint4>*
      __restrict__ out_2x2, elemUint4* __restrict__ data, unsigned int* __restrict__ counts2x2_actual, lensVariables lensVars,
      float npc, unsigned int* __restrict__ ticket_counter)
{

  //set the out array
  out_2x2->data = data;
  out_2x2->offsets = scanned_counts_2x2;
  out_2x2->width = width2x2;
  out_2x2->height = height2x2;
  out_2x2->actual_counts = counts2x2_actual;

  const unsigned int max_ticket = width2x2*height2x2;
  const int BATCH_SIZE = 1;
  unsigned int ticket = atomicAdd(ticket_counter,BATCH_SIZE);

  __shared__ float4 values[4*BLOCK_DIM_X*BLOCK_DIM_Y]; //register spill
  int tid = (threadIdx.y*BLOCK_DIM_X + threadIdx.x)*4;
  int counts[4];
  int elemIndex[4];

  while(ticket<max_ticket){

    int2 gid = make_int2(ticket%width2x2, ticket/width2x2);
    int2 outgid = make_int2(gid.x,gid.y);
    gid*=2;

    //+1 because layerindex is saved in counting tex
    counts[0] = tex2D(counting_tex_cuda,gid.x,gid.y)+1;
    counts[1] = tex2D(counting_tex_cuda,gid.x+1,gid.y)+1;
    counts[2] = tex2D(counting_tex_cuda,gid.x,gid.y+1)+1;
    counts[3] = tex2D(counting_tex_cuda,gid.x+1,gid.y+1)+1;

    values[tid+0] = tex3D(color_textures,gid.x,gid.y,0);
    values[tid+1] = tex3D(color_textures,gid.x+1,gid.y,0);
    values[tid+2] = tex3D(color_textures,gid.x,gid.y+1,0);
    values[tid+3] = tex3D(color_textures,gid.x+1,gid.y+1,0);

    elemIndex[0] = 1;
    elemIndex[1] = 1;
    elemIndex[2] = 1;
    elemIndex[3] = 1;

    int allCount = counts[0]+counts[1]+counts[2]+counts[3];

    //depth is in range [0,1] transform to eye space depth:
    values[tid+0].w *=(lensVars.cam_far-lensVars.cam_near);
    values[tid+1].w *=(lensVars.cam_far-lensVars.cam_near);
    values[tid+2].w *=(lensVars.cam_far-lensVars.cam_near);
    values[tid+3].w *=(lensVars.cam_far-lensVars.cam_near);

```

```
    //use for umbra discard
    float lastDepth = 0.f;
    float dis = 0.f;

    int out_index = 0;
    while(allCount>0){

      float maxDepth = 0.f;
      elemUint4 elem_out;

      //still 4 different elements left and mergeable?
      if((counts[0]>=elemIndex[0]) && (counts[1]>=elemIndex[1])
          && (counts[2]>=elemIndex[2]) && (counts[3]>=elemIndex[3])
          && checkMerge2x2(&values[tid], lensVars, &maxDepth))
      {
        //merge into elemUint4
        float3 merged_col =make_float3(0.f,0.f,0.f);
        for(int i=0; i<4; i++){
          float3 col = make_float3(values[tid+i].x,values[tid+i].y,values[tid+i].z);
          //sanity check against bad scene files
          if(isnan(col.x)
            || isnan(col.y)
            || isnan(col.z)){
            col = make_float3(0.f,0.f,0.f);
          }
          merged_col+= col;
        }
        merged_col.x*=0.25f;
        merged_col.y*=0.25f;
        merged_col.z*=0.25f;

        float newDepthToStore = maxDepth+0.01f;

        elem_out = elemUint4(__float2half_rn(merged_col.x),__float2half_rn(merged_col.y),__float2half_rn(merged_col.z)
            ,2,newDepthToStore,gid.x,gid.y);

        //umbra compute
        lastDepth =  maxDepth;
        float s = (1.0f/npc)*lastDepth * (1.5f) * lensVars.cam_near ;
        dis = (lastDepth*s)/(lensVars.aperture-s);
        if(dis<=0.0f) dis=20000.0f;
        dis= min(20000.0f,dis);

        //load next elems
        for(int i=0; i<4; i++){
          if(counts[i]>=elemIndex[i]){
            do{
              values[tid+i] = tex3D(color_textures ,gid.x+i%2,gid.y+i/2,elemIndex[i]);
              values[tid+i].w *=(lensVars.cam_far-lensVars.cam_near);

              ++elemIndex[i];
              --allCount;
            }while(((lastDepth+dis)>values[tid+i].w) && (lastDepth<values[tid+i].w ) && (counts[i]>=elemIndex[i]));
          }
        }
      } else {
        //remove first element
        int lowest = -1;
        get_minimum(counts, elemIndex ,&values[tid],lowest);

        float3 col = make_float3(values[tid+lowest].x,values[tid+lowest].y,values[tid+lowest].z);
        //sanity check against bad scene files
        if(isnan(col.x)
          || isnan(col.y)
          || isnan(col.z)){
          col = make_float3(0.f,0.f,0.f);
        }
        //put into elemUint4
        elem_out = elemUint4(__float2half_rn(col.x),__float2half_rn(col.y),__float2half_rn(col.z)
              ,1,values[tid+lowest].w,gid.x+lowest%2,gid.y+lowest/2);

        //get new
        values[tid+lowest] = tex3D(color_textures ,gid.x+lowest%2,gid.y+lowest/2,elemIndex[lowest]);
        values[tid+lowest].w *=(lensVars.cam_far-lensVars.cam_near);

        ++elemIndex[lowest];
        --allCount;
      }
      //write out
      out_2x2->put(outgid.x,outgid.y,out_index, elem_out);
      ++out_index;
    }
    out_2x2->setCount(outgid.x,outgid.y,out_index);

    //fetch next ticket
    ticket = atomicAdd(ticket_counter,BATCH_SIZE);
  }
}
```

3.4 Merge 4x4

```
//CUDA 8.0 kernels, compute capability 6.1

__device__ static bool checkMerge4x4(elemUint4* __restrict__ values, lensVariables lensVars, float* __restrict__ maxDep){
  const float COLOR_EPSILON = 0.05f;
  const float MERGE_EPSILON_BASE = 18000.f;
  float MERGE_EPSILON = MERGE_EPSILON_BASE/max(2.f,lensVars.aperture);
  const float LUMINANCE_EPSILON = 0.05f;
  const int DIM_X = 2, DIM_Y = 2, step = 4;

  bool luminance = true;
  bool cocMerge = false;

  float minDisToFocus=100000.f;
  for(int i=0; i<DIM_X*DIM_Y; i++){
    minDisToFocus = min(minDisToFocus, abs(abs(values[i].depth())-lensVars.focus_plane));
  }

  float minDepth = 1000000.f;
  float maxDepth = 0.f;

  for(int i=0; i<DIM_X*DIM_Y;i++){
    minDepth = min(minDepth,abs(values[i].depth()));
    maxDepth = max(maxDepth,abs(values[i].depth()));
  }
  float maxDiffDepth = abs(maxDepth-minDepth);

  for(int i=0; i<DIM_X*DIM_Y; i++){
    int otherColIndex = (i==(DIM_X*DIM_Y-1)) ? 0 : (i+1);

    float l1 = abs(__half2float(values[i].r())*0.2126f + __half2float(values[i].g())*0.7152f +
          __half2float(values[i].b())*0.0722f);
    float l2 = abs(__half2float(values[otherColIndex].r())*0.2126f +__half2float(values[otherColIndex].g())*0.7152f +
          __half2float(values[otherColIndex].b())*0.0722f);

    if(l1 - l2 > LUMINANCE_EPSILON){
      luminance=false;
    }
    if(l1>0.8) luminance=false;
  }

  maxDiffDepth++; //little adjustment, div by zero
  float mergeFactor = (minDisToFocus*minDisToFocus)/(maxDiffDepth*step);
  if(mergeFactor > MERGE_EPSILON) cocMerge=true;

  maxDep[0] = maxDepth;
  if(cocMerge && luminance){
    return true;
  } else{
    return false;
  }
}



__global__ void kernelMerge4x4_ticket(int width4x4, int height4x4, int* __restrict__ scanned_counts_4x4,
    array2d<elemUint4>* __restrict__ merged_2x2_in,array2d<elemUint4>* __restrict__ out_4x4, elemUint4* __restrict__  data,
    unsigned int* __restrict__ counts4x4_actual,  lensVariables lensVars, float npc, unsigned int* __restrict__ ticket_counter){

  //set the out array
  out_4x4->data = data;
  out_4x4->offsets = scanned_counts_4x4;
  out_4x4->width = width4x4;
  out_4x4->height = height4x4;
  out_4x4->actual_counts = counts4x4_actual;

  unsigned int max_ticket = width4x4*height4x4;
  const int BATCH_SIZE = 1;
  unsigned int ticket = atomicAdd(ticket_counter,BATCH_SIZE);

  __shared__ elemUint4 values[4*16*16]; //register spill
  int tid = (threadIdx.y*16 + threadIdx.x)*4;

  int counts[4];
  int elemIndex[4];

  while(ticket<max_ticket){

    int2 gid = make_int2(ticket%width4x4, ticket/width4x4);
    int2 outgid = make_int2(gid.x,gid.y);
    gid*=2;

    values[tid+0] = (*merged_2x2_in)(gid.x,gid.y,0);
    values[tid+1] = (*merged_2x2_in)(gid.x+1,gid.y,0);
    values[tid+2] = (*merged_2x2_in)(gid.x,gid.y+1,0);
    values[tid+3] = (*merged_2x2_in)(gid.x+1,gid.y+1,0);

    counts[0] = merged_2x2_in->getCount(gid.x,gid.y);
    counts[1] = merged_2x2_in->getCount(gid.x+1,gid.y);
    counts[2] = merged_2x2_in->getCount(gid.x,gid.y+1);
    counts[3] = merged_2x2_in->getCount(gid.x+1,gid.y+1);
    elemIndex[0] = 1;
    elemIndex[1] = 1;
    elemIndex[2] = 1;
    elemIndex[3] = 1;
    int allCount = counts[0]+counts[1]+counts[2]+counts[3];

        //umbra discard
    float lastDepth = 0.f;
    float s = 0.f;
    float dis = 0.f;

    int out_index = 0;

```

```
    while(allCount>0){

      float maxDepth = 0.f;
      //still 4 different elements left and only merged ones
      if((counts[0]>=elemIndex[0]) && (counts[1]>=elemIndex[1])
          && (counts[2]>=elemIndex[2]) && (counts[3]>=elemIndex[3])
          && values[tid+0].w()==2 && values[tid+1].w() == 2 && values[tid+2].w() == 2 && values[tid+3].w() == 2
          && checkMerge4x4(&values[tid], lensVars, &maxDepth))
      {

        //merge into elemUint4
        float3 merged_col =make_float3(0.f,0.f,0.f);
        for(int i=0; i<4; i++){
          merged_col+=make_float3(__half2float(values[tid+i].r()),
                          __half2float(values[tid+i].g()),__half2float(values[tid+i].b()));
        }
        merged_col.x*=0.25f;
        merged_col.y*=0.25f;
        merged_col.z*=0.25f;

        float newDepthToStore = maxDepth+0.01f;

        elemUint4 merged(__float2half_rn(merged_col.x),__float2half_rn(merged_col.y),__float2half_rn(merged_col.z)
            ,4,newDepthToStore,gid.x*2,gid.y*2);

        out_4x4->put(outgid.x,outgid.y,out_index, merged);
        ++out_index;

        //umbra compute
        lastDepth =  maxDepth;
        s = (1.0f/npc)*lastDepth * (1.5f) * lensVars.cam_near;
        dis = (lastDepth*s)/(lensVars.aperture-s);
        if(dis<=0.0f) dis=20000.0f;
        dis= min(20000.0f,dis);

        for(int i=0; i<4; i++){
          if(counts[i]>=elemIndex[i]){
            do{
              values[tid+i] = (*merged_2x2_in)(gid.x+i%2,gid.y+i/2,elemIndex[i]);
              ++elemIndex[i];
              --allCount;
            }while(((lastDepth+dis)>values[tid+i].depth()) && (lastDepth<values[tid+i].depth() ) && (counts[i]>=elemIndex[i]));
          }
        }
      } else {
        //find first element
        int lowest =get_minimum_elemUint4(counts, elemIndex ,&values[tid]);

        //write out
        out_4x4->put(outgid.x,outgid.y,out_index, values[tid+lowest]);
        ++out_index;

        //get new
        values[tid+lowest] = (*merged_2x2_in)(gid.x+lowest%2,gid.y+lowest/2,elemIndex[lowest]);
        ++elemIndex[lowest];
        --allCount;
      }
    }
    out_4x4->setCount(outgid.x,outgid.y,out_index);

    ticket = atomicAdd(ticket_counter,BATCH_SIZE);
  }
}
```

3.5 Splat

```
//CUDA 8.0 kernels, compute capability 6.1

//prepass for workqueue
__global__ static void kernelSplatListPrepass(int width4x4, int height4x4,
    array2d<elemUint4>* __restrict__ merged4x4,
    int* __restrict__ out,
    unsigned int* __restrict__ listOffset)
{
  int2 gid = make_int2(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y);
  if (gid.x >= width4x4 || gid.y >= height4x4) return;

  int count = merged4x4->getCount(gid.x,gid.y);
  int listOff=0;
  if(gid.x!=0||gid.y!=0)
    listOff = listOffset[(gid.x+gid.y*width4x4)-1];

  for(int i=0; i< count; ++i){
    int lane = gid.x+gid.y*merged4x4->width;
    int offset = i;
    if(lane!=0){
      offset += merged4x4->offsets[lane-1];
    }
    out[listOff+i] = offset;
  }
}


//actual splatting
__device__ static void checkAndSplat(int width4x4, int height4x4, array2d<elemUint4>*__restrict__ out_splatted, elemUint4 e,
      lensVariables lensVars, float npc)
{
  int2 TL_ID = make_int2(e.x()/TILE_SIZE,e.y()/TILE_SIZE);

  const int TILE_SIZE_FACTOR = TILE_SIZE/4;
  const int W_TILE_SIZE = (width4x4%TILE_SIZE_FACTOR)==0 ? width4x4/TILE_SIZE_FACTOR : (width4x4/TILE_SIZE_FACTOR)+1;
  const int H_TILE_SIZE = (height4x4%TILE_SIZE_FACTOR)==0 ? height4x4/TILE_SIZE_FACTOR : (height4x4/TILE_SIZE_FACTOR)+1;

  int low_x = ((TL_ID.x-TILE_SPREAD)<0) ? -TL_ID.x : -TILE_SPREAD;
  int high_x = (TL_ID.x>=(W_TILE_SIZE-TILE_SPREAD)) ? -(TL_ID.x-W_TILE_SIZE+1) : TILE_SPREAD;
  int low_y = (TL_ID.y-TILE_SPREAD<0) ? -TL_ID.y : -TILE_SPREAD;
  int high_y = (TL_ID.y>=(H_TILE_SIZE-TILE_SPREAD)) ? -(TL_ID.y-H_TILE_SIZE+1) : TILE_SPREAD;

  bool may[2*TILE_SPREAD+1][2*TILE_SPREAD+1];
  for(int i = 0; i<2*TILE_SPREAD+1; i++){
    for(int j=0; j<2*TILE_SPREAD+1; j++){
      may[i][j]=false;
    }
  }
  may[TILE_SPREAD][TILE_SPREAD]=true;

  float coc = getCocOfElem(e,lensVars);
  coc*=0.5f; //radius
  coc+=0.5f;
  coc = max(0.6f,coc);
  float elem_x = float(e.x())+float(e.w()-1.f)*0.5f;
  float elem_y = float(e.y())+float(e.w()-1.f)*0.5f;

  for(int i=0; i<TILE_SPREAD; ++i){
    int leftBorder = (TL_ID.x-TILE_SPREAD-1+i)*TILE_SIZE;
    int rightBorder = (TL_ID.x+1+i)*TILE_SIZE;
    int upperBorder = (TL_ID.y+1+i)*TILE_SIZE;
    int lowerBorder = (TL_ID.y-TILE_SPREAD-1)*TILE_SIZE;
    float disToLeftX = leftBorder - elem_x;
    float disToRightX = rightBorder - elem_x;
    float disToUpperY = upperBorder - elem_y;
    float disToLowerY = lowerBorder - elem_y;
    if(disToLeftX<coc) may[TILE_SPREAD][i] = true;
    if(disToRightX<coc) may[TILE_SPREAD][TILE_SPREAD+1+i] = true;
    if(disToUpperY<coc) may[TILE_SPREAD+1+i][TILE_SPREAD] = true;
    if(disToLowerY<coc) may[i][TILE_SPREAD] = true;
  }

  for(int i = 0; i<2*TILE_SPREAD+1; i++){
    for(int j=0; j<2*TILE_SPREAD+1; j++){
      if(i==TILE_SPREAD || j==TILE_SPREAD) continue;

      int add_x = 0;
      int add_y = 0;
      if(i>TILE_SPREAD) add_x=-1;
      if(j>TILE_SPREAD) add_y=-1;
      int point_x = (TL_ID.x-TILE_SPREAD+1+i+add_x)*TILE_SIZE;
      int point_y = (TL_ID.y-TILE_SPREAD+1+j+add_y)*TILE_SIZE;
      float disToX = point_x - elem_x;
      float disToY = point_y - elem_y;
      float cocSq = coc*coc;
      if((disToX*disToX+disToY*disToY)< cocSq) {
        may[j][i] = true;
      }
    }
  }

  for(int y = low_y; y<=high_y; ++y){
    for(int x = low_x; x<=high_x; ++x ){
            if(may[(y+TILE_SPREAD)][(x+TILE_SPREAD)]){
        int2 index = make_int2(TL_ID.x+x,TL_ID.y+y);
        int posInTileList = atomicAdd(&out_splatted->actual_counts[index.x + index.y*W_TILE_SIZE],1);
        out_splatted->put(index.x,index.y, posInTileList, e);

      }
    }
  }
}
```

```
//workqueue kernel for splat invoking
__global__ static void kernelSplattingWithTicketBatch(int width4x4, int height4x4,
    array2d<elemUint4>* __restrict__ out_splatted, int* __restrict__ scanned_counts, elemUint4* __restrict__ data,
    unsigned int* __restrict__ counts_actual, unsigned int* __restrict__ scanned_counts_of_4x4,
    array2d<elemUint4>* merged4x4, int* __restrict__ splat_list_pointer, unsigned int* __restrict__ ticket,
    lensVariables lensVars, float npc)
{

  const int BATCH_SIZE = 4;
  unsigned int max_elems = scanned_counts_of_4x4[width4x4*height4x4-1] - BATCH_SIZE;
  const int TILE_SIZE_FACTOR = TILE_SIZE/4;

  out_splatted->data = data;
  out_splatted->offsets = scanned_counts;
  out_splatted->width = (width4x4%TILE_SIZE_FACTOR)==0 ? width4x4/TILE_SIZE_FACTOR : (width4x4/TILE_SIZE_FACTOR)+1;
  out_splatted->height = (height4x4%TILE_SIZE_FACTOR)==0 ? height4x4/TILE_SIZE_FACTOR : (height4x4/TILE_SIZE_FACTOR)+1;
  out_splatted->actual_counts = counts_actual;

  unsigned int workIndex = atomicAdd(ticket,BATCH_SIZE);

  while(workIndex < max_elems){
    for(int i=0; i< BATCH_SIZE; ++i){
      elemUint4 e = *(merged4x4->data + splat_list_pointer[workIndex+i]);
      checkAndSplat(width4x4, height4x4, out_splatted, e, lensVars, npc);
    }
    workIndex = atomicAdd(ticket,BATCH_SIZE);
  }

  //last Elements
  while(workIndex < max_elems+BATCH_SIZE){
    elemUint4 e = *(merged4x4->data + splat_list_pointer[workIndex]);
    checkAndSplat(width4x4, height4x4, out_splatted, e, lensVars, npc);
    workIndex++;
  }
}
```

3.6 Sort

```
//CUDA 8.0 kernels, compute capability 6.1

//prepass to bin by sort amount
__global__ static void kernelSortingPrePass(int width_tilesize, int height_tilesize, array2d<elemUint4>* splatted, unsigned int
      *sort_prepass_counts, int *sorting_prepass_pointers)
{
  int2 gid = make_int2(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y);
  if (gid.x >= width_tilesize || gid.y >= height_tilesize) return;

  int N = width_tilesize*height_tilesize;
  int lane = gid.x+gid.y*splatted->width;
  int count = splatted->getCount(lane);
  int I = 0;
  if(count<256){
    I = 0;
  }else if(count<512){
    I = 1;
  }else if(count<1024){
    I = 2;
  }else if(count<2048){
    I = 3;
  }else if(count<4096){
    I = 4;
  }else if(count<8192){
    I = 5;
  }else {
    I = 6;
  }
  int index = atomicAdd(&sort_prepass_counts[I],1);
  sorting_prepass_pointers[I*N + index] = lane;
}

//actual bitonic sorting
template<unsigned int shared_mem_amount, unsigned int min_amount, unsigned int ticket_counter_offset>
__global__ static void kernelBitonicSorter_withTicket_balance(int width, int height, array2d<elemUint4>* __restrict__ splatted,
    array2d<elemUint4>* __restrict__ out_sorted, unsigned int*__restrict__ counts_sorted_actual, elemUint4* __restrict__ data,
    lensVariables lensVars, unsigned int* __restrict__ tc, unsigned int *sort_prepass_counts, int *sorting_prepass_pointers,
    unsigned int *tickets)
{
  int w = ((width%TILE_SIZE)==0?width/TILE_SIZE : (width/TILE_SIZE+1));
  int h = (height%TILE_SIZE)==0?height/TILE_SIZE : (height/TILE_SIZE+1);
  int N = w*h*ticket_counter_offset;

  out_sorted->data = data;
  out_sorted->offsets = splatted->offsets;
  out_sorted->width = w;
  out_sorted->height = h;
  out_sorted->actual_counts = counts_sorted_actual;
  const int blockSize = blockDim.x * blockDim.y;

  const int tidInBlock = threadIdx.y*blockDim.x + threadIdx.x;
  __shared__ sortElem sharedMem[shared_mem_amount];
  unsigned int* ticket_counter = tc+ticket_counter_offset;

  const unsigned int max_ticket = sort_prepass_counts[ticket_counter_offset];
  __shared__ unsigned int ticket_shared;
  unsigned int *ticket = &ticket_shared;

  unsigned int count;
  if(tidInBlock==0){
    *ticket = atomicAdd(ticket_counter,1);
  }
  __syncthreads();

  while(*ticket < max_ticket){

    int index_list_to_process = sorting_prepass_pointers[N+*ticket];
    count = splatted->getCount(index_list_to_process);
    elemUint4 *thisList = splatted->getListPointer(index_list_to_process);
    elemUint4 *out_list = out_sorted->getListPointer(index_list_to_process);
    int blockId = index_list_to_process;
    int loadIterations = count/blockSize;
    int loadLastElems = count % blockSize;

    //normal sorting
    int logN = ceil(log2(float(count)));
    int sortingAmount = 1<<logN;

    //each thread needs work
    sortingAmount = max(sortingAmount,blockSize);

    //load elements to shared mem
    for(int i=0; i< loadIterations; i++){
      elemUint4 e = thisList[i*blockSize + tidInBlock];
      float depth = e.depth();
      int depth_of = convert_depth_to_int(width, depth, e, blockId);
      sharedMem[i*blockSize + tidInBlock] =sortElem(i*blockSize + tidInBlock,depth_of);
    }

    //fill rest with dummies
    for(int i=loadIterations; i<sortingAmount/blockSize; i++){
      float max_depth = FLT_MAX;
      sharedMem[i*blockSize + tidInBlock] = sortElem(15000,max_depth);
    }

    if(tidInBlock < loadLastElems){
      elemUint4 e = thisList[loadIterations*blockSize+tidInBlock];
      float depth = e.depth();
      int depth_of = convert_depth_to_int(width, depth, e, blockId);
      sharedMem[loadIterations*blockSize+tidInBlock] = sortElem(loadIterations*blockSize+tidInBlock,depth_of);
    }
    __syncthreads();
```

```
103     //now ready to start sorting
104
105     int elementsToSortPerThread = ceil((float(sortingAmount)/float(blockSize)));
106     for(int i=0; i<logN; i++){
107       for(int j=0; j<=i; j++){
108         for(int k=0; k<elementsToSortPerThread; k++){
109
110           int compareWithElementOffset = 1;
111           compareWithElementOffset<<=(i-j);
112           int index = k*blockSize+tidInBlock;
113           bool up = ((index >> i) & 2) == 0;
114           int otherIndex = index | compareWithElementOffset;
115
116           if( ((sharedMem[index].depth() > sharedMem[otherIndex].depth()) &&  up) ||
117           ((sharedMem[index].depth() < sharedMem[otherIndex].depth()) && !up) )
118           {
119             sortElem tmp =sharedMem[index];
120             sharedMem[index] = sharedMem[otherIndex];
121             sharedMem[otherIndex] = tmp;
122           }
123         }
124         __syncthreads();
125       }
126     }
127
128     if(tidInBlock==0){
129       *ticket = atomicAdd(ticket_counter,1);
130     }
131     __threadfence();
132     __syncthreads();
133
134     //sorting is done, write back results
135     for(int i=0; i< loadIterations; i++){
136
137       int index_load = sharedMem[i*blockSize + tidInBlock].index();
138       assert(index_load!=15000);
139
140       elemUint4 cp = thisList[index_load];
141       precompute_for_blur(cp,lensVars);
142
143       out_list[i*blockSize + tidInBlock] = cp;
144     }
145
146     if(tidInBlock < loadLastElems){
147       int index_load = sharedMem[loadIterations*blockSize+tidInBlock].index();
148       assert(index_load!=15000);
149
150       elemUint4 cp = thisList[index_load];
151       precompute_for_blur(cp,lensVars);
152
153       out_list[loadIterations*blockSize+tidInBlock] = cp;
154     }
155     out_sorted->setCount(blockId, count);
156   }
157 }
158
159 //precomputation of alpha and coc squared, as those computations are costly in the apply step
160 __device__ __inline__ void precompute_for_blur(elemUint4 &elem, lensVariables lensVars){
161
162   float depth = elem.depth();
163   float coc = getCocOfElem(elem, lensVars);
164   coc*=0.5f; //radius
165
166   int merged = elem.w()*elem.w();
167   if(elem.w()>1.f ){
168     coc+= (elem.w()/2.f-0.5f)*1.4143f;
169   }
170
171   float cocSq = coc*coc;
172   float area = cocSq;
173   float alpha = 1.f/area;
174   float weight = 1.f;
175   for(int k=1; k<merged; k++){
176     float value = 1.0f;
177     for(int l=1; l<=k; l++){
178       value *= (1.f-alpha);
179     }
180     weight+=value;
181   }
182
183   weight*=alpha;
184   alpha = weight;
185   alpha = min(1.f,alpha);
186
187   if(elem.w()>1.f)
188     cocSq = (coc+elem.w()/4.f)*(coc+elem.w()/4.f);
189
190   elem.setNewXandY(float(elem.x())+(float(elem.w())-1.f)*0.5f,float(elem.y())+(float(elem.w())-1.f)*0.5f);
191   elem.setNewDepthCoc(cocSq);
192   elem.setNewW(alpha);
193 }
```

3.7 Apply

```
//CUDA 8.0 kernels, compute capability 6.1

//smaller version with 64 threads per block
__global__ static void kernelApplyBlur_Warpload_unroll_fma_bonus_small_ballot_break(int width, int height,
        array2d<elemUint4> *sorted_lists,
        lensVariables lensVars
        )
{
    const int tilewh = TILE_SIZE;
    int2 gid = make_int2(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y);
    if (gid.x >= width || gid.y >= height) return;

    float2 gidF = make_float2(float(gid.x),float(gid.y));

    int w = (width%tilewh)==0 ? width/tilewh : (width/tilewh)+1;
    int tileIndex = gid.x/tilewh + w* (gid.y/tilewh);
    int numPerTile = sorted_lists->getCount(tileIndex);
    elemUint4 *list = sorted_lists->getListPointer(tileIndex);

    float4 color = make_float4(0.f,0.f,0.f,1.f);
    int tid = threadIdx.y*blockDim.x + threadIdx.x;

    int warpId = tid/32;
    int threadIDInWarp = tid%32;
    __shared__ elemUint4 local_data[64];

    float alpha_dest = 1.0f;
    int iterationsWith32 = numPerTile/32;
    int lastIter32 = numPerTile%32;

    for(int i=0; i<iterationsWith32; i++){

        local_data[warpId*32+threadIDInWarp] = list[i*32+threadIDInWarp];

        if(alpha_dest>0.01f)
        {
            for(int j=0; j<32; j++){
                elemUint4 elem =local_data[warpId*32+j];
                float distY = elem.getPrecomputedYNew()-gidF.y;
                float distX = elem.getPrecomputedXNew()-gidF.x;
                float preFMAyy = distY*distY;
                float cocSq = elem.getPrecomputedCoC();
                float disToCenterSq = __fmaf_ru(distX,distX,preFMAyy);
                if(disToCenterSq <=cocSq)
                {
                    float preFMA = alpha_dest*elem.getPrecomputedW();
                    color.x = __fmaf_ru(preFMA,__half2float(elem.r()),color.x);
                    color.y = __fmaf_ru(preFMA,__half2float(elem.g()),color.y);
                    color.z = __fmaf_ru(preFMA,__half2float(elem.b()),color.z);

                    alpha_dest -= preFMA;
                }
            }
        }
        if(__all(alpha_dest<=0.01f)){
            surf2Dwrite(make_half4(color.x/(1.f-alpha_dest), color.y/(1.f-alpha_dest),
                  color.z/(1.f-alpha_dest),1.0f),hdr_image_surface_odp,gid.x*2*4,gid.y);
            return;
        }
    }

    if(threadIDInWarp < lastIter32){
        local_data[warpId*32+threadIDInWarp] = list[iterationsWith32*32+threadIDInWarp];
    }
    if(alpha_dest>0.01f)
    {
        for(int j=0; j<lastIter32; j++){
            elemUint4 elem =local_data[warpId*32+j];
            float distY = elem.getPrecomputedYNew()-gidF.y;
            float distX = elem.getPrecomputedXNew()-gidF.x;
            float preFMAyy = distY*distY;
            float cocSq = elem.getPrecomputedCoC();
            float disToCenterSq = __fmaf_ru(distX,distX,preFMAyy);
            if(disToCenterSq <=cocSq)
            {
                float preFMA = alpha_dest*elem.getPrecomputedW();
                color.x = __fmaf_ru(preFMA,__half2float(elem.r()),color.x);
                color.y = __fmaf_ru(preFMA,__half2float(elem.g()),color.y);
                color.z = __fmaf_ru(preFMA,__half2float(elem.b()),color.z);

                alpha_dest -= preFMA;
            }
        }

    }
    surf2Dwrite(make_half4(color.x/(1.f-alpha_dest), color.y/(1.f-alpha_dest),
          color.z/(1.f-alpha_dest),1.0f),hdr_image_surface_odp,gid.x*2*4,gid.y);
}
```

3.8 Data Structures

```
struct lensVariables{
  float aperture;
  float eye_to_lens;
  float S;  //lin_coc.x
  float B;  //lin_coc.y
  float max_coc;
  float cam_near;
  float cam_far;
  float focus_plane;
  float focal_length;

  void reset(float near_plane_correction, float maxCoc){
    eye_to_lens = lens.eye_to_lens;
    max_coc = maxCoc;
    aperture = lens.aperture;
    focal_length = 1.0f/((1.0f/lens.eye_to_lens)+(1.0f/lens.focus_distance));
    float f = focal_length;
    S = ((lens.aperture * f * lens.focus_distance) / (lens.focus_distance - f)) * near_plane_correction;
    B = (-(lens.aperture * f) / (lens.focus_distance - f)) * near_plane_correction;
    cam_near = camera_near(current_camera());
    cam_far = camera_far(current_camera());
    focus_plane = lens.focus_distance;
  }
};

class elemUint4{
  /*  Data in Format:
   *   16b 16b
   *  | r | g |
   *  | w | b | << w = weight
   *  | a | d | << a = alpha, d = depth //alpha not used, instead 32bit depth
   *  | x | y |
   */
public:
  uint4 internData;

  __device__ __host__ elemUint4(){}
  __device__ __host__ elemUint4(const elemUint4 &other): internData(other.internData){}
  __device__ elemUint4(unsigned short r,unsigned short g, unsigned short b, unsigned short w, float linear_depth, unsigned
        short x, unsigned short y, float alpha = 1.f)
  {
    internData.x = (uint)(( ((unsigned int)r)<<16 )| g);
    internData.y = (uint)(( ((unsigned int)w)<<16 ) |b);
    unsigned short depth_half_float = __half_as_ushort(__float2half(linear_depth/1000.f));
    unsigned short alpha_half_float = __half_as_ushort(__float2half(alpha));
    internData.z = __float_as_int(linear_depth);
    internData.w = (uint)((((unsigned int)x) << 16 )| y);
  }
  __device__ __forceinline__ unsigned short r(){
    return ushort(internData.x>>16);
  }
  __device__ __forceinline__ unsigned short g(){
    return ushort(internData.x );//& 0x0000ffff);
  }
  __device__ __forceinline__ unsigned short b(){
    return ushort(internData.y );//& 0x0000ffff);
  }
  __device__ __forceinline__ unsigned short w(){
    return ushort(internData.y>>16);
  }
  __device__ __forceinline__ float depth(){
    float depth = __int_as_float(internData.z);// & 0x0000ffff);
    return depth;
  }
  __device__ __forceinline__ float alpha(){
    ushort alpha = ushort(internData.z>>16);
    return __half2float(__ushort_as_half(alpha));
  }
  __device__ __forceinline__ unsigned short x(){
    return ushort(internData.w>>16);
  }
  __device__ __forceinline__ unsigned short y(){
    return ushort(internData.w & 0x0000ffff);
  }
  __device__ __forceinline__ void setNewW(float w){
    ushort it = __half_as_ushort(__float2half(w));
    ushort blue = ushort(internData.y & 0x0000ffff);
    internData.y = (uint)(( ((unsigned int)it)<<16 ) |blue);
  }
  __device__ __forceinline__ void setNewXandY(float x, float y){
    unsigned short nx = __half_as_ushort(__float2half(x));
    unsigned short ny = __half_as_ushort(__float2half(y));
    internData.w = (uint)(( ((unsigned int)nx)<<16 )| ny);
  }
  __device__ __forceinline__ float getPrecomputedXNew(){
    return __half2float(internData.w>>16);
  }
  __device__ __forceinline__ float getPrecomputedYNew(){
    return __half2float(internData.w);
  }
  __device__ __forceinline__ void setNewDepthCoc(float coc){
    unsigned short depth_half_float = __half_as_ushort(__float2half(coc));
    ushort alpha = ushort(internData.z>>16);
    internData.z = (uint)(( ((unsigned int)alpha)<<16 )| depth_half_float);
  }
  __device__ __forceinline__ float getPrecomputedCoC(){
    return __half2float(internData.z );
  }
  __device__ __forceinline__ float getPrecomputedW(){
    return __half2float(__ushort_as_half(internData.y>>16));
  }
};
```

```
struct sortElemINDEX16{
  int data;
  __device__ sortElemINDEX16(){}
  __device__ sortElemINDEX16(unsigned short index, unsigned int depth)
  {
    unsigned int i = index;
    data = (((unsigned int)( i))<<16) | (depth&0x000ffff);
  }
  __device__ unsigned int depth(){
    return ((unsigned int)(data&0x0000ffff));
  }
  __device__ unsigned short index(){
    return ushort((data>>16)&0x0000ffff);
  }
};

template<typename T>
struct array2d{
  T*  data;
  int*  offsets;
  int width, height;
  unsigned int*  actual_counts;

  __device__ __host__ array2d(){}
  __device__ __host__ array2d(int width, int height, T* d, int *off)
    : width(width),height(height),data(d),offsets(off){}

  __device__ __host__ T& access(int lane, int index){
    if(lane ==0) return data[index];
    int offset = offsets[lane-1];
    return data[offset+index];
  }

  __device__ T& access(int x, int y, int index){
    int lane = x+y*width;
    return access(lane,index);
  }

  __device__ T* getListPointer(int lane){
    if(lane ==0) return data;
    int offset = offsets[lane-1];
    return &data[offset];
  }

  __device__ T& operator()(int x, int y, int index){
    return access(x,y,index);
  }

  __device__ void put(int x, int y, int index, T elem){
    T *e = &access(x+y*width,index);
    *e = elem;
  }

  __device__ void setCount(int lane, int count){
    actual_counts[lane]=count;
  }

  __device__ void setCount(int x, int y, int count){
    int lane = x+y*width;
    setCount(lane,count);
  }

  __device__ int getCount(int lane){
    assert(lane>=0 && lane<width*height);
    return actual_counts[lane];
  }

  __device__ int getCount(int x,int y){
    int lane = x+y*width;
    return getCount(lane);
  }
};
```