ZED Scene Segmentation

In cadrul proiectului final de la scoala de vara, am realizat implementarea a doua tehnici de segmentare a imaginilor. Prima tehnica a fost segmentarea iterativa fara puncte germen, unde am folosit metoda de parcurgere scan-line pentru identificarea regiunilor. Am integrat si un pas suplimentar de region merging, care a implicat stocarea regiunilor vecine si unirea acestora pe baza similaritatilor, pentru a imbunatati coeziunea segmentarii. A doua tehnica a fost segmentarea bazata pe muchii, care a inclus obtinerea muchiilor, binarizarea, dilatarea si umplerea regiunilor folosind algoritmul BFS.

De asemenea, am etichetat regiunile in functie de caracteristicile geometrice ale acestora. Am identificat si etichetat podeaua, peretii, tavanul si obiectele generice pe baza regulilor stabilite: regiunile orizontale de la distanta mare au fost etichetate ca podea, regiunile perpendiculare pe podea ca pereti, regiunile de sus cu normala specifica ca tavan, iar regiunile neetichetate anterior ca obiecte generice.

Planar Segmentation

```
enum LabelType {
    FLOOR,
    WALL,
    CEILING,
    GENERIC_OBJECT,
    UNLABELED
};

struct Region {
    int n = 1, n_valid = 1;
    uchar mean_intensity, mean_depth;
    cv::Vec4f mean_normal, mean_position;
    std::unordered_set<ushort> neighbors;
    LabelType label = UNLABELED;
    Region(uchar intensity, uchar depth, const cv::Vec4f& normal, const cv::Vec4f& position);
    void add_pixel(uchar intensity, uchar depth, const cv::Vec4f& normal, const cv::Vec4f& position);
    void merge_region(const Region& region);
    float distance();
};
```

```cpp
Region::Region(uchar intensity, uchar depth, const cv::Vec4f& normal, const cv::Vec4f& position) :mean_intensity(intensity), mean_depth(depth), mean_normal(normal), mean_position(position)
    if (isnan(normal[0])) {
        mean_depth = 0;
        mean_normal = cv::Vec4f(0, 0, 0, 1);
        mean_position = cv::Vec4f(0, 0, 0, 1);
        n_valid = 0;
    }
}
void Region::add_pixel(uchar intensity, uchar depth, const cv::Vec4f& normal, const cv::Vec4f& position) {
    mean_intensity = (n * mean_intensity + intensity) / (n + 1);
    if (!isnan(normal[0])) {
        mean_depth = (n_valid * mean_depth + depth) / (n_valid + 1);
        mean_normal = (n_valid * mean_normal + normal) / (n_valid + 1);
        mean_position = (n_valid * mean_position + position) / (n_valid + 1);
        n_valid++;
    }
    n++;
}
void Region::merge_region(const Region& region) {
    mean_intensity = (n * mean_intensity + region.n * region.mean_intensity) / (n + region.n);
    mean_depth = (n_valid * mean_depth + region.n_valid * region.mean_depth) / (n_valid + region.n_valid);
    mean_normal = (n_valid * mean_normal + region.n_valid * region.mean_normal) / (n_valid + region.n_valid);
    mean_position = (n_valid * mean_position + region.n_valid * region.mean_position) / (n_valid + region.n_valid);
    neighbors.insert(region.neighbors.begin(), region.neighbors.end());
    n += region.n;
    n_valid += region.n_valid;
}

float Region::distance() {
    glm::vec3 normal(mean_normal[0], mean_normal[1], mean_normal[2]),
        position(mean_position[0], mean_position[1], mean_position[2]);
    return abs(glm::dot(normal, position)) / glm::length(normal);
}
```

Structura Region reprezinta o regiune in imagine, stocand informatii despre intensitatea medie, adancimea, norma si pozitia medie a pixelilor din regiune. Metodele add_pixel si merge_region actualizeaza aceste medii pe baza noilor pixeli si regiuni, respectiv, asigurand corectitudinea statisticilor in cadrul fuziunii.

```cpp
void Filter::planarSegmentation(uchar* grayscaleData, cv::Vec4b* depthData, cv::Vec4f* normalMeasureData, cv::Vec4f* pointCloudData, ushort* regionsData, int width, int height) {
    int offset = 0, offset_neighbor, offset_min, n = 0;
    float cost, cost_normal, cost_depth, cost_min;
    regions.clear();
    for (int y = 0;y < height;y++)
    {
        for (int x = 0;x < width;x++)
        {
            cv::Vec4f& normal = normalMeasureData[offset];
            offset_min = -1;
            cost_min = 350;
            for (int k = -1;k <= 0;k++) {
                for (int l = -1;l <= 1;l++) {
                    offset_neighbor = offset + k * width + l;
                    if ((y + k >= 0 && y + k < height && x + l >= 0 && x + l < width) && offset_neighbor < offset) {
                        Region& region = regions[regionsData[offset_neighbor]];
                        if (isnan(normal[0]) || isnan(normalMeasureData[offset_neighbor][0])) {
                            cost_normal = 0;
                            cost_depth = 0;
                        }
                        else {
                            cost_normal = abs(1 - glm::dot(glm::vec3(normal[0], normal[1], normal[2]), glm::vec3(region.mean_normal[0], region.mean_normal[1], region.mean_normal[2])));
                            cost_depth = abs(depthData[offset][0] - depthData[offset_neighbor][0]);
                        }
                        cost = abs(grayscaleData[offset] - region.mean_intensity) + 1000 * cost_normal + 50 * cost_depth;
                        if (cost < cost_min) {
                            offset_min = offset_neighbor;
                            cost_min = cost;
                        }
                    }
                }
            }
            if (offset_min != -1) {
                regions[regionsData[offset_min]].add_pixel(grayscaleData[offset], depthData[offset][0], normal, pointCloudData[offset]);
                regionsData[offset] = regionsData[offset_min];
            }
            else {
                regions.push_back(Region(grayscaleData[offset], depthData[offset][0], normal, pointCloudData[offset]));
                regionsData[offset] = n;
                n++;
            }
            offset++;
        }
    }
    offset = 0;
    for (int y = 0;y < height;y++)
    {
        for (int x = 0;x < width;x++)
        {
            for (int k = -1;k <= 1;k++) {
                for (int l = -1;l <= 1;l++) {
                    offset_neighbor = offset + k * width + l;
                    if ((y + k >= 0 && y + k < height && x + l >= 0 && x + l < width) && regionsData[offset_neighbor] != regionsData[offset]) {
                        regions[regionsData[offset]].neighbors.insert(regionsData[offset_neighbor]);
                    }
                }
            }
            offset++;
        }
    }
}
```

Functia “planarSegmentation” implementeaza segmentarea planara a imaginii, alocand fiecare pixel intr-o regiune specifica pe baza unor costuri calculate. In prima parte, pentru fiecare pixel, se calculeaza costul de asociare cu vecinii sai, folosind informatii de intensitate a griurilor, adancime si normale.

Costurile sunt ponderate astfel incat diferentele de normala si adancime au un impact semnificativ asupra deciziei de alocare a pixelilor in regiuni. Daca pixelul nu se potriveste cu niciuna dintre regiunile existente, este creata o noua regiune.

In a doua parte, se identifica regiunile vecine, actualizand lista de vecini pentru fiecare regiune. Acest proces ajuta la construirea unei retele de regiuni interconectate, facilitand analiza ulterioara.

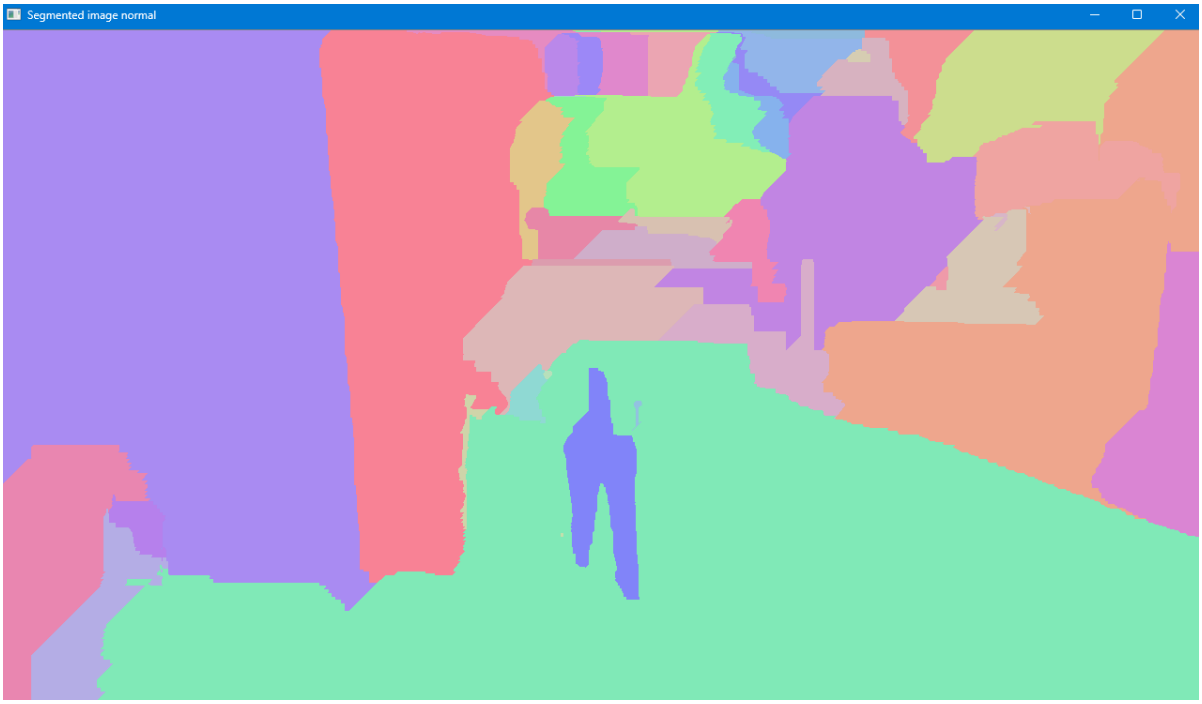

```
void Filter::regionMerging(ushort* regionsData, int width, int height) {
    float cost, cost_distance;
    std::unordered_map<ushort, ushort> ids;
    std::unordered_set<ushort> merged_neighbors;
    glm::vec3 normal, normal_neighbor;
    for (ushort i = 0;i < regions.size();i++) {
        ids[i] = i;
    }
    for (ushort i = 0;i < ids.size();i++) {
        normal = glm::vec3(regions[ids[i]].mean_normal[0], regions[ids[i]].mean_normal[1], regions[ids[i]].mean_normal[2]);
        merged_neighbors.clear();
        for (ushort neighbor : regions[ids[i]].neighbors) {
            if (ids[neighbor] != ids[i]) {
                normal_neighbor = glm::vec3(regions[ids[neighbor]].mean_normal[0], regions[ids[neighbor]].mean_normal[1], regions[ids[neighbor]].mean_normal[2]);
                if (glm::length(normal) < 1e-2 || glm::length(normal_neighbor) < 1e-2) {
                    cost_distance = 0;
                }
                else {
                    cost_distance = abs(regions[ids[i]].distance() - regions[ids[neighbor]].distance());
                }
                cost = cost_distance + 4000 * abs(1 - glm::dot(normal, normal_neighbor));
                if (cost < 1400) {
                    merged_neighbors.insert(neighbor);
                }
            }
        }
        for (ushort neighbor : merged_neighbors) {
            if (ids[neighbor] != ids[i]) {
                regions[ids[i]].merge_region(regions[ids[neighbor]]);
                regions.erase(regions.begin() + ids[neighbor]);
                for (ushort j = 0;j < ids.size();j++) {
                    if (ids[j] == ids[neighbor] && j != neighbor) {
                        ids[j] = ids[i];
                    }
                }
                for (ushort j = 0;j < ids.size();j++) {
                    if (ids[j] > ids[neighbor]) {
                        ids[j]--;
                    }
                }
                ids[neighbor] = ids[i];
            }
        }
    }
    for (int offset = 0;offset < width * height;offset++) {
        regionsData[offset] = ids[regionsData[offset]];
    }
}
```

Functia “regionMerging” se ocupa cu unificarea regiunilor adiacente bazate pe similaritatea normelor si distantelor minime pana la camera ale regiunilor. In prima parte, functia initializeaza un map care asociaza fiecare regiune cu un identificator propriu. Apoi, pentru fiecare regiune, se calculeaza costurile de fuziune cu vecinii sai, avand in vedere diferentele dintre normele regiunilor si distantele minime pana la camera. Daca costul total al fuziunii este sub un anumit prag, regiunile sunt unite, iar identificatorii acestora sunt actualizati.

Procesul continua pana cand toate posibilele fuziuni sunt efectuate. Acest algoritm ajuta la reducerea numarului de regiuni si la crearea unor regiuni mai omogene si reprezentative.

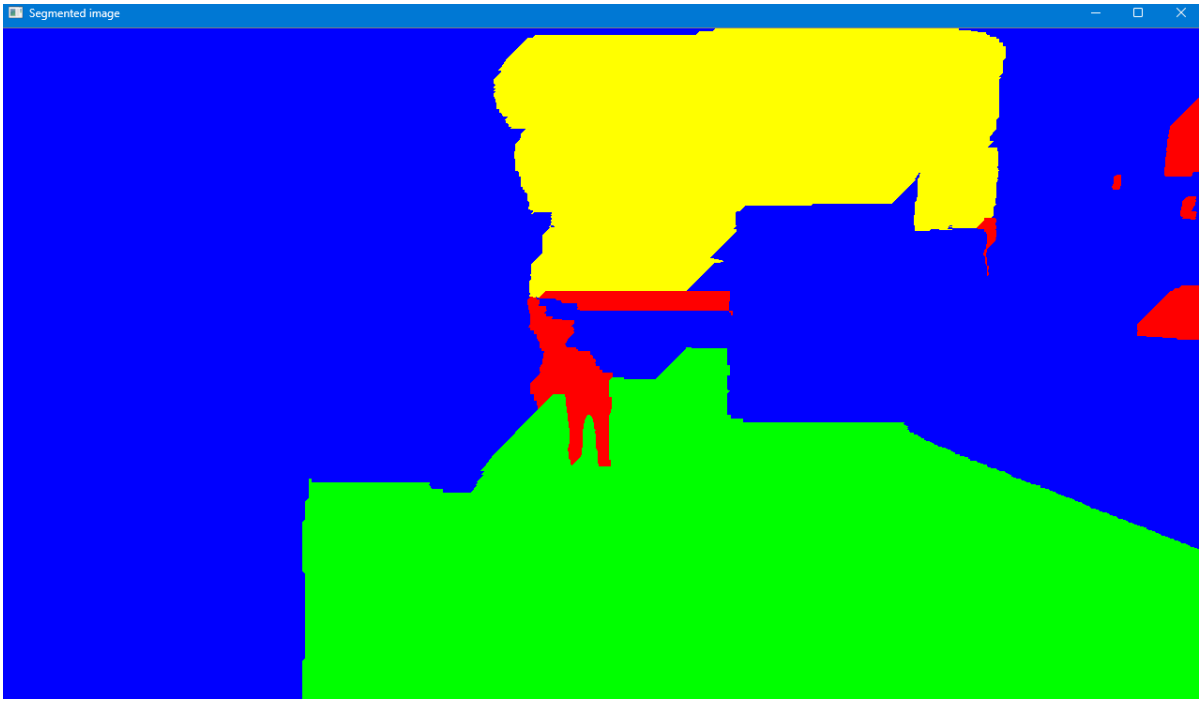

```
void Filter::regionsToPropertyImages(ushort* regionsData, uchar* segmentedImageGrayscaleData, cv::Vec4b* segmentedImageDepthData, cv::Vec4b* segmentedImageNormalData, int width, int height) {
    for (int offset = 0;offset < width * height;offset++) {
        Region& region = regions[regionsData[offset]];
        segmentedImageGrayscaleData[offset] = region.mean_intensity;
        segmentedImageDepthData[offset] = cv::Vec4b(region.mean_depth, region.mean_depth, region.mean_depth, 255);
        segmentedImageNormalData[offset] = cv::Vec4b((abs(region.mean_normal[2]) + 1) / 2 * 255, (abs(region.mean_normal[1]) + 1) / 2 * 255, (abs(region.mean_normal[0]) + 1) / 2 * 255, 255);
    }
}
```

Functia “regionsToPropertyImages” mapeaza caracteristicile medii ale fiecarei regiuni (intensitate, adancime si normala) in imagini separate pentru a vizualiza aceste proprietati.




```
void Filter::labelRegions(ushort* regionsData, cv::Vec4b* labeledImageData, int width, int height) {
    const cv::Vec4b labelColors[4] = { cv::Vec4b(0, 255, 0, 255),cv::Vec4b(255, 0, 0, 255),cv::Vec4b(0, 255, 255, 255),cv::Vec4b(0, 0, 255, 255) };
    for (Region& region : regions) {
        glm::vec3 normal(region.mean_normal[0], region.mean_normal[1], region.mean_normal[2]);
        if (glm::dot(normal, glm::vec3(0, -1, 0)) > 0.7 && region.distance() > 2000) {
            region.label = FLOOR;
        }
        else if (abs(glm::dot(normal, glm::vec3(0, -1, 0))) < 0.5 && region.n > 5000) {
            region.label = WALL;
        }
        else if (glm::dot(normal, glm::vec3(0, 1, 0)) > 0.3) {
            region.label = CEILING;
        }
        else {
            region.label = GENERIC_OBJECT;
        }
    }
    for (int offset = 0;offset < width * height;offset++) {
        LabelType& label = regions[regionsData[offset]].label;
        if (label != UNLABELED) {
            labeledImageData[offset] = labelColors[label];
        }
    }
}
```

Functia “labelRegions” eticheteaza regiunile in imagine pe baza normalelor si distantelor, clasificandu-le ca podea, perete, tavan sau obiect generic. Apoi, coloreaza imaginea de iesire folosind culorile corespunzatoare fiecarei etichete. Acest proces faciliteaza vizualizarea si analiza segmentarii scenei.



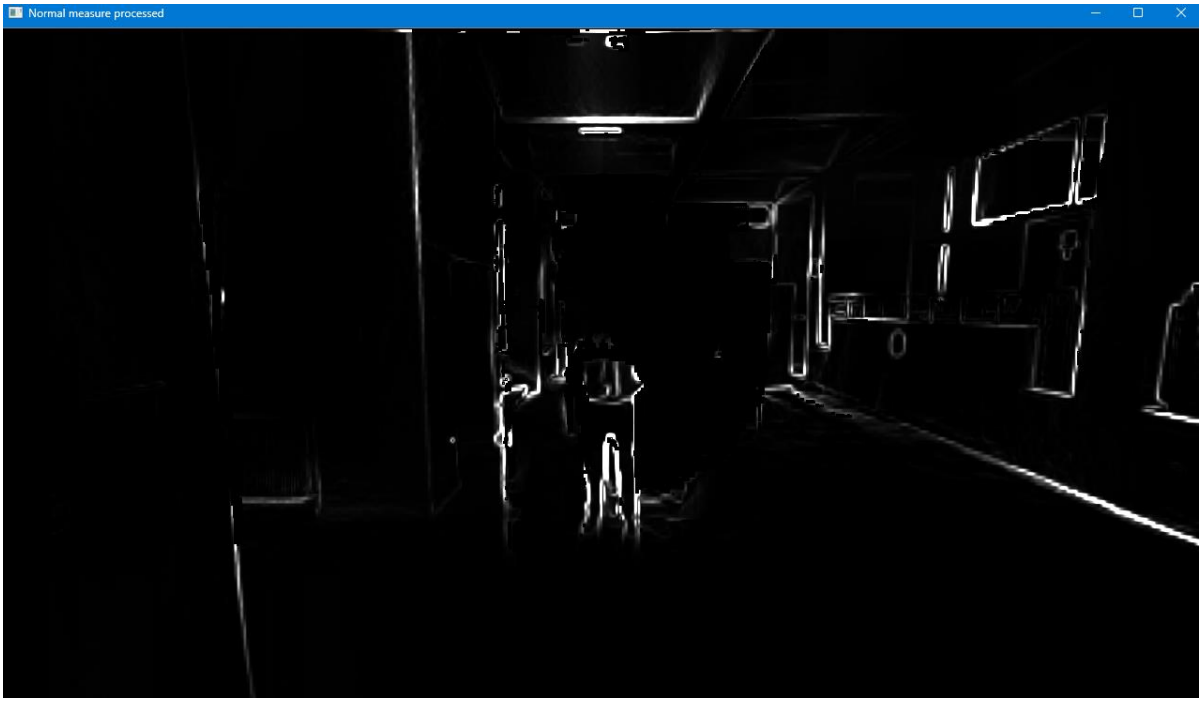
Edge Segmentation

```
void Filter::filterGrayscaleSobel(uchar* grayscaleData, uchar* grayscaleProcessedData, int width, int height) {
    int offset, offset_neighbor;
    int sobelKernelX[3][3] = { {1,0,-1},{2,0,-2},{1,0,-1} }, sobelKernelY[3][3] = { {1,2,1},{0,0,0},{-1,-2,-1} };
    for (int y = 1; y < height - 1; y++)
    {
        for (int x = 1; x < width - 1; x++)
        {
            float grayX = 0, grayY = 0;
            for (int k = -1; k <= 1; k++)
            {
                for (int l = -1; l <= 1; l++)
                {
                    offset_neighbor = (y + k) * width + (x + l);
                    grayX += (float)grayscaleData[offset_neighbor] * sobelKernelX[k + 1][l + 1];
                    grayY += (float)grayscaleData[offset_neighbor] * sobelKernelY[k + 1][l + 1];
                }
            }

            offset = y * width + x;
            grayscaleProcessedData[offset] = (uchar)sqrt(grayX * grayX + grayY * grayY);
        }
    }
}
```

Functia “filterGrayscaleSobel” aplica filtrele Sobel pentru detectarea marginilor intr-o imagine in nuante de gri. Utilizeaza doua kerneluri Sobel, unul pentru directia orizontala si altul pentru cea verticala, pentru a calcula gradientul imaginii in ambele directii. Rezultatul este o imagine care evidentiaza marginile prin intensificarea diferentelor de intensitate.

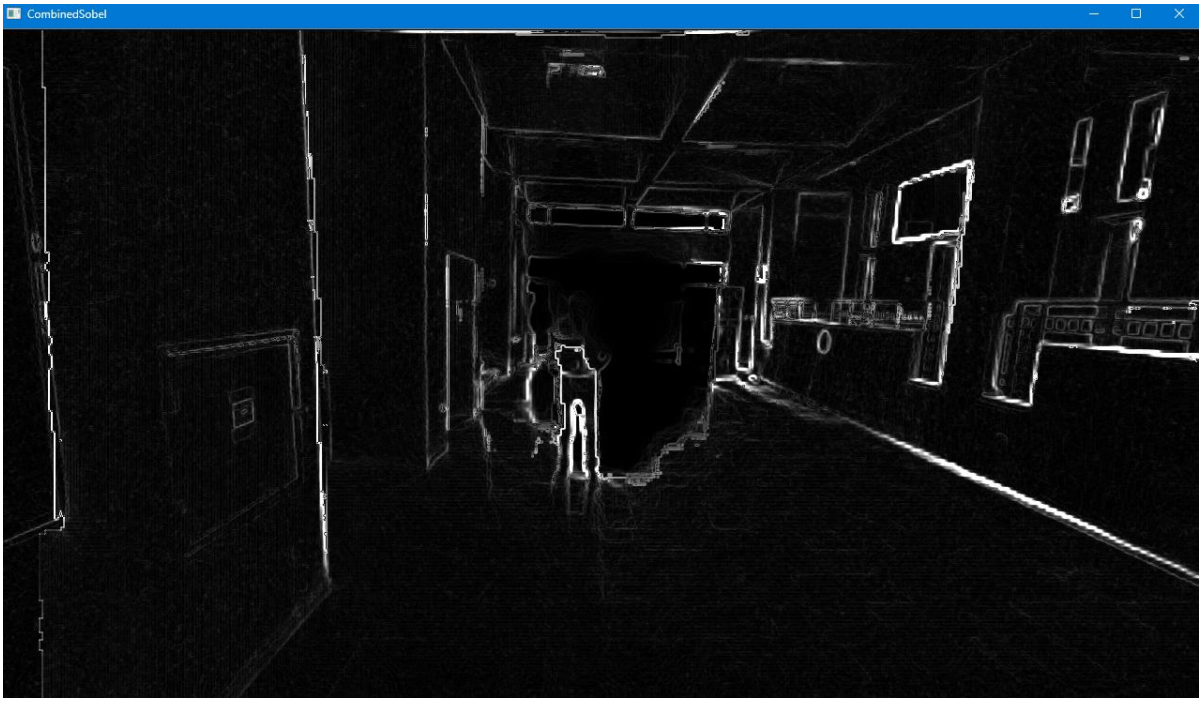

```cpp
void Filter::filterNormalSobel(cv::Vec4f* normalMeasureData, cv::Vec4b* normalMeasureProcessedData, int width, int height) {
    cv::Vec4f normal_left, normal_right, normal_up, normal_down;

    int offset;
    int sobelKernel[3] = { 1,2,1 };
    for (int y = 1; y < height - 1; y++)
    {
        for (int x = 1; x < width - 1; x++)
        {
            float grayX = 0, grayY = 0;
            uchar gray;
            offset = y * width + x;
            for (int k = -1; k <= 1; k++)
            {
                normal_left = normalMeasureData[offset + k * width - 1];
                normal_right = normalMeasureData[offset + k * width + 1];
                normal_up = normalMeasureData[offset + k - width];
                normal_down = normalMeasureData[offset + k + width];
                grayX += (1 - glm::dot(glm::vec3(normal_left[0], normal_left[1], normal_left[2]), glm::vec3(normal_right[0], normal_right[1], normal_right[2]))) * sobelKernel[k + 1];
                grayY += (1 - glm::dot(glm::vec3(normal_up[0], normal_up[1], normal_up[2]), glm::vec3(normal_down[0], normal_down[1], normal_down[2]))) * sobelKernel[k + 1];
            }
            if (isnan(grayX) || isnan(grayY)) {
                gray = 0;
            }
            else {
                gray = (uchar)std::min(255.0f, 2550 * sqrt(grayX * grayX + grayY * grayY));
            }
            normalMeasureProcessedData[offset] = cv::Vec4b(gray, gray, gray, 255);
        }
    }
}
```

Functia “filterNormalSobel” aplica un filtru Sobel pe normalele de suprafata pentru a detecta marginile bazate pe schimbarile dintre vectorii normali. Foloseste doua kernel-uri Sobel pentru a calcula gradientul normalelor in directiile X si Y.




```
void Filter::filterDepthSobel(cv::Vec4b* depthData, cv::Vec4b* depthProcessedData, int width, int height) {
    int offset, offset_neighbor;
    int sobelKernelX[3][3] = { {1,0,-1},{2,0,-2},{1,0,-1} }, sobelKernelY[3][3] = { {1,2,1},{0,0,0},{-1,-2,-1} };
    for (int y = 1; y < height - 1; y++)
    {
        for (int x = 1; x < width - 1; x++)
        {
            float grayX = 0, grayY = 0;
            uchar gray;
            for (int k = -1; k <= 1; k++)
            {
                for (int l = -1; l <= 1; l++)
                {
                    offset_neighbor = (y + k) * width + (x + l);
                    grayX += (float)depthData[offset_neighbor][0] * sobelKernelX[k + 1][l + 1];
                    grayY += (float)depthData[offset_neighbor][0] * sobelKernelY[k + 1][l + 1];
                }
            }

            offset = y * width + x;
            gray = (uchar)sqrt(grayX * grayX + grayY * grayY);
            depthProcessedData[offset] = cv::Vec4b(gray, gray, gray, depthData[offset][3]);
        }
    }
}
void Filter::filterCombinedSobel(uchar* grayscaleProcessedData, cv::Vec4b* depthProcessedData, cv::Vec4b* normalMeasureProcessedData, uchar* combinedSobelImageData, int width, int height) {
    for (int offset = 0;offset < width * height;offset++) {
        combinedSobelImageData[offset] = (uchar)std::min(255.0, 0.4 * grayscaleProcessedData[offset] + depthProcessedData[offset][0] + normalMeasureProcessedData[offset][0]);
    }
}
```

Functia “filterDepthSobel” aplica kernel-uri Sobel pentru a detecta marginile in datele de adancime, calculand gradientul in directiile X si Y pentru a produce o imagine bazata pe magnitudinea schimbarilor de adancime, iar functia “filterCombinedSobel” foloseste rezultatele acestei filtrari, impreuna cu datele procesate de la imagini in tonuri de gri si normale, pentru a genera o imagine combinata. Aceasta imagine combinata reflecta intensitatea marginilor din toate sursele, imbunatatind analiza detaliilor geometrice si contururilor prin fuzionarea informatiilor din diferite tipuri de imagini.

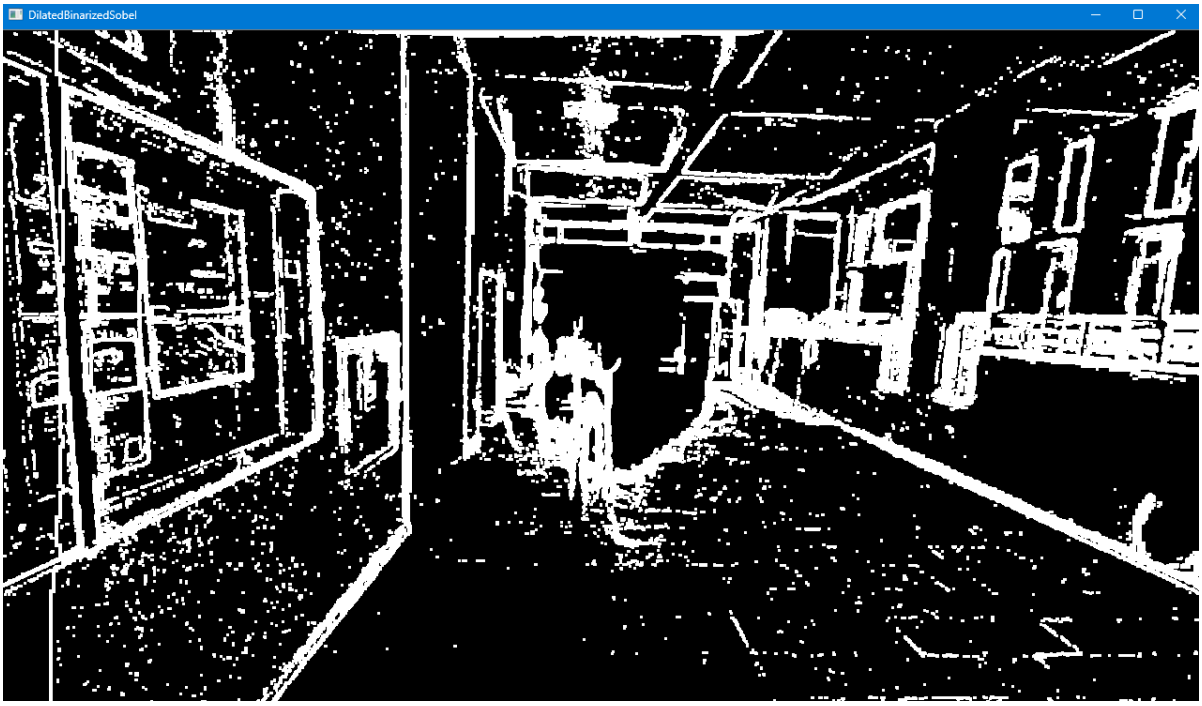



```
void Filter::filterBinarization(uchar* sobelImageData, uchar* binarizedSobelImageData, int width, int height) {
    for (int offset = 0;offset < width * height;offset++) {
        if (sobelImageData[offset] < 25) {
            binarizedSobelImageData[offset] = 0;
        }
        else {
            binarizedSobelImageData[offset] = 255;
        }
    }
}
```

Functia “filterBinarization” transforma imaginea Sobel intr-o imagine binara. Aceasta proceseaza fiecare pixel din imaginea Sobel si, pe baza unui prag fixat (25), seteaza valoarea pixelilor sub acest prag la 0 (negru) si valoarea pixelilor deasupra pragului la 255 (alb). Acest proces evidentiaza marginile detectate prin Sobel, simplificand imaginea pentru analize ulterioare.

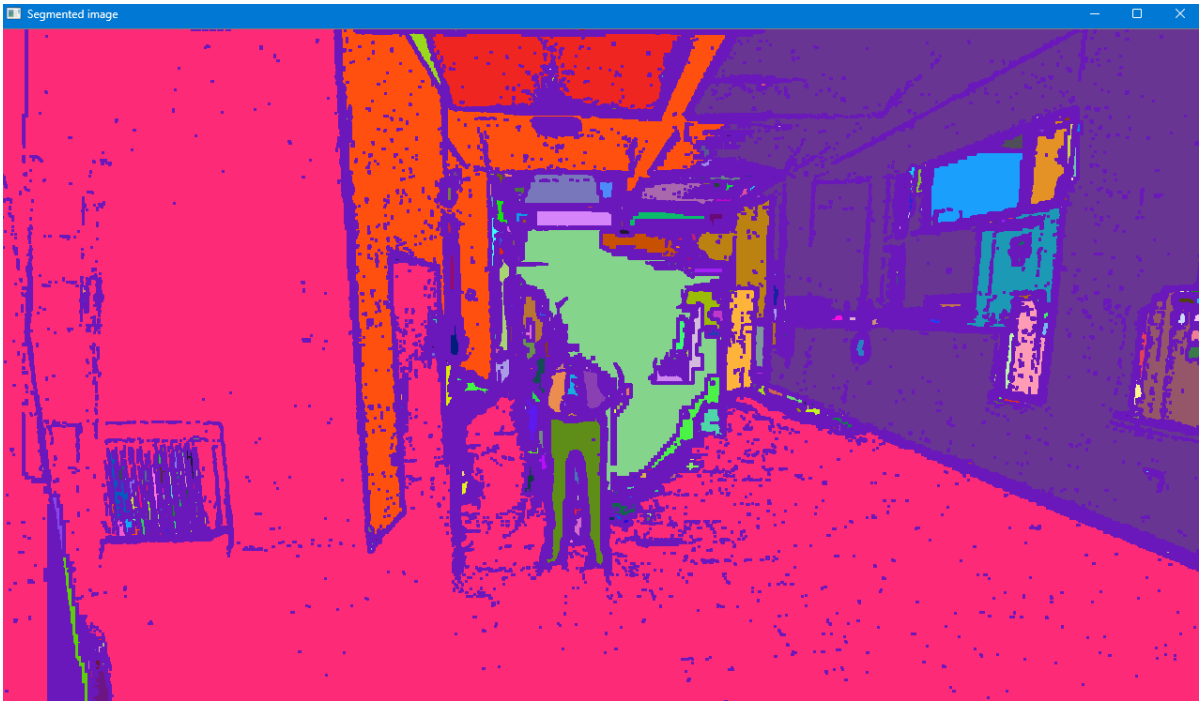

```
void Filter::filterDilation(uchar* binarizedSobelImageData, uchar* dilatedBinarizedSobelImageData, int width, int height) {
    int offset = 0, offset_neighbor;
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            if (binarizedSobelImageData[offset] == 0) {
                dilatedBinarizedSobelImageData[offset] = 0;
                for (int k = -1; k <= 1; k++)
                {
                    for (int l = -1; l <= 1; l++)
                    {
                        if (y + k >= 0 && y + k < height && x + l >= 0 && x + l < width) {
                            offset_neighbor = offset + k * width + l;
                            if (binarizedSobelImageData[offset_neighbor] != 0) {
                                dilatedBinarizedSobelImageData[offset] = 255;
                                goto exit;
                            }
                        }
                    }
                }
            }
            else {
                dilatedBinarizedSobelImageData[offset] = 255;
            }
        exit:
            offset++;
        }
    }
}
```

Functia “filterDilation” aplica o dilatare morfologica asupra imaginii binarizate. In aceasta functie, fiecare pixel din imaginea binarizata este verificat: daca pixelul este negru (0), se examineaza vecinii sai pentru a vedea daca unul dintre acestia este alb (255). Daca exista un vecin alb, pixelul curent este setat la alb, altfel ramane negru. Pixelii care sunt deja albi raman albi. Acest proces extinde zonele albe si poate ajuta la umplerea lacunelor din margini sau obiecte.




```
void Filter::edgeSegmentation(uchar* edgeImageData, ushort* regionsData, int width, int height) {
    std::queue<int> region;
    int offset, offset_neighbor, n = 1;
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            offset = y * width + x;
            if (edgeImageData[offset] != 255 && regionsData[offset] == 0) {
                region.push(offset);
                regionsData[offset] = n;
                while (!region.empty()) {
                    offset = region.front();
                    for (int k = -1;k <= 1;k++) {
                        for (int l = -1;l <= 1;l++) {
                            offset_neighbor = offset + k * width + l;
                            if ((offset / width + k >= 0 && offset / width + k < height && offset % width + l >= 0 && offset % width + l < width) &&
                                edgeImageData[offset_neighbor] != 255 && regionsData[offset_neighbor] == 0) {
                                region.push(offset_neighbor);
                                regionsData[offset_neighbor] = n;
                            }
                        }
                    }
                    region.pop();
                }
                n++;
            }
        }
    }
}
```

Functia “edgeSegmentation” aplica o tehnica de segmentare bazata pe cautarea in latime pentru a identifica regiunile in imagine. In aceasta functie, se parcurge imaginea si, pentru fiecare pixel care nu a fost deja marcat, se initiaza o cautare in latime. Pixelii sunt adaugati intr-o coada si verificati pentru a gasi vecinii care sunt si ei parte a regiunii si nu au fost inca marcati. Fiecare pixel identificat intr-o regiune este marcat cu un identificator de regiune unic. Acest proces continua pana cand toti pixelii dintr-o regiune sunt marcati, si apoi se trece la urmatoarea regiune neexplorata.




```
void Filter::regionsToRandomColorImage(ushort* regionsData, cv::Vec4b* segmentedImageRandomColorData, int width, int height) {
    int n = *std::max_element(regionsData, regionsData + width * height) + 1;
    static std::vector<cv::Vec4b> colors;
    if (colors.size() < n) {
        int n_old = colors.size();
        colors.resize(n);
        srand(time(0));
        for (int i = n_old; i < n; i++) {
            colors[i] = cv::Vec4b(rand() % 256, rand() % 256, rand() % 256, 255);
        }
    }
    for (int offset = 0;offset < width * height;offset++) {
        segmentedImageRandomColorData[offset] = colors[regionsData[offset]];
    }
}
```

Functia “regionsToRandomColorImage” aloca culori aleatorii fiecarei regiuni detectate intr-o imagine. Se determina numarul total de regiuni si se genereaza culori unice pentru fiecare regiune, stocandu-le intr-un vector. Apoi, pentru fiecare pixel din imagine, se aplica culoarea corespunzatoare regiunii sale, creand astfel o imagine segmentata colorata aleatoriu.