

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего
профессионального образования

«Волгоградский государственный технический университет»

Факультет Электроники и вычислительной техники
наименование факультета

Кафедра Электронно-вычислительные машины и системы
наименование кафедры

ОТЧЕТ

О учебной практике на кафедре ЭВМиС
вид практики наименование места прохождения практики

Руководитель практики _____ Ф.И.О.
Должность подпись

Студент гр. Титов А.К. _____ Ф.И.О.
Подпись

Студент гр. Марков А.Е. _____ Ф.И.О.
Подпись

Отчет защищен с оценкой _____
Дата Подпись
руководителя
практики

Волгоград 2015

Оглавление

1 Введение	4
2 Основная часть	5
2.1 Теоретические сведения	5
2.2 Построение облака точек с использованием OpenCV.....	8
2.2.1 Оптический поток и метод triangulate.....	8
2.2.1.1 Алгоритм построения облака точек	8
2.2.1.2 Оптический поток	10
2.2.1.2.1 Особенности реализации.....	11
2.2.1.3 Триангуляция точек для создания облака точек.....	11
2.2.1.3.1 Особенности реализации.....	12
2.2.1.3.1.1 1 способ получения матриц проекции:	12
2.2.1.3.1.2 2 способ получения матриц проекции:	12
2.2.1.3.1.2.3 Построение облака точек	13
2.2.1.4 Результаты	13
2.2.2 Стереο калибровка и Block Matching	15
2.2.2.1 Общий принцип.....	15
2.2.2.2 Калибровка камер.....	16
2.2.2.2.1 Особенности реализации.....	19
2.2.2.3 Алгоритм построения облака точек	20
2.2.2.4 Построение карты различий	20
2.2.2.4.1 Особенности реализации.....	24
2.2.2.5.1 Особенности реализации.....	29
3 Заключение	33
4 Список использованной литературы.....	35

Приложение А: функция калибровки	36
Приложение В: функция построения облака точек.....	39
Приложение С: функция усреднения карты различий.....	41
Приложение D: функция разделения объектов.....	42

1 Введение

Задание на учебную практику заключается в разработке программы, формирующую облако точек по набору фотографий объекта.

Задачи:

1. Изучить

- а) а) Основы работы с библиотекой компьютерного зрения OpenCV
- б) б) Изучить способы формирования облака точек по нескольким изображениям, полученных с разных ракурсов
- в) в) Изучить основные возможности библиотеки обработки изображений Aforge.NET

2. Разработать программу, формирующую облако точек по набору фотографий объекта. Оптимизировать процесс съемки для получения наиболее качественного облака точек.

3. Произвести анализ полученных результатов, сделать выводы о возможности практического применения разработанной программы.

Для решения поставленной задачи используется библиотека компьютерного зрения OpenCV (Open Source Computer Vision Library, библиотека компьютерного зрения с открытым исходным кодом) — библиотека алгоритмов компьютерного зрения, обработки изображений и численных алгоритмов общего назначения с открытым кодом.

Программа разрабатывается на языке C++.

2 Основная часть

2.1 Теоретические сведения

Для построения облака точек, необходимо два и более снимков объекта с разных ракурсов. В основе этого процесса лежит проективная геометрия – раздел геометрии, изучающий проективные плоскости и пространства. С использованием аппарата проективной геометрии можно осуществлять проецирование точки в трехмерном пространстве на плоскость, а также осуществлять обратный процесс, при наличии двух и более проекций точки.

Точка в проективной геометрии задается с использованием однородных координат: (x, y, z, w) . Переход от однородных координат к декартовым осуществляется следующим образом: $(x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$.

Важную роль в проективной геометрии играет модель камеры:

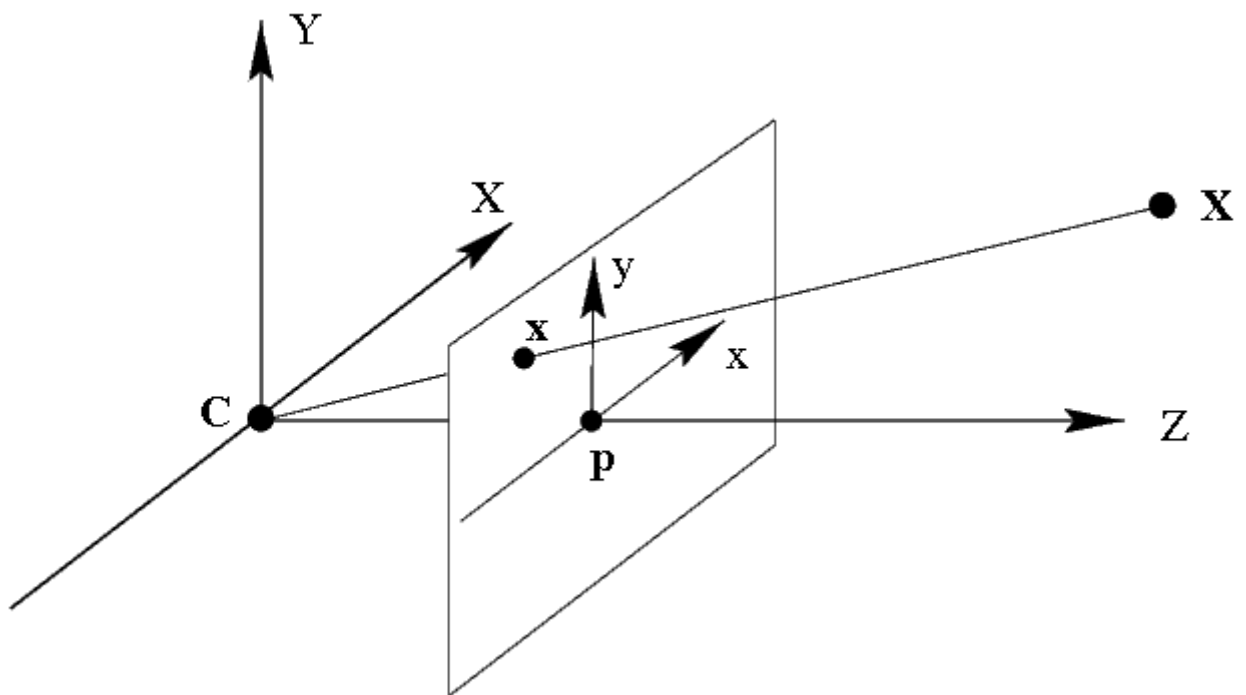


Рисунок 2.1 – схема камеры

На данной схеме:

- X – точка в трехмерном пространстве
- x – проекция точки

- C_p – главная ось камеры [1]

Координаты проекции в СК камеры определяется следующим образом:

$x = PX$, где P – проективная матрица (матрица проекции)

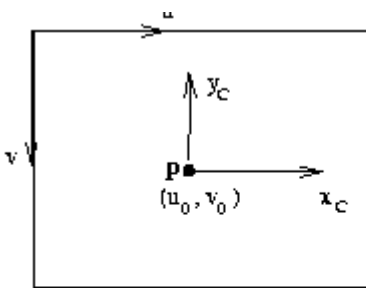
Проективная матрица описывает камеру и определяется:

- Внутренними параметрами камеры
- Внешними параметрами камеры

Внутренние параметры камеры определяются ее фокусным расстоянием и центром (он может быть смещен).

$$\begin{aligned} k_u x_c &= u - u_0 \\ k_v y_c &= v_0 - v \end{aligned}$$

where the units of k are [pixels/length].



$$x_i = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f k_u & 0 & u_0 \\ 0 & -f k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ f \end{bmatrix} = C \begin{bmatrix} x_c \\ y_c \\ f \end{bmatrix}$$

Рисунок 2.2 – внутренние параметры камеры

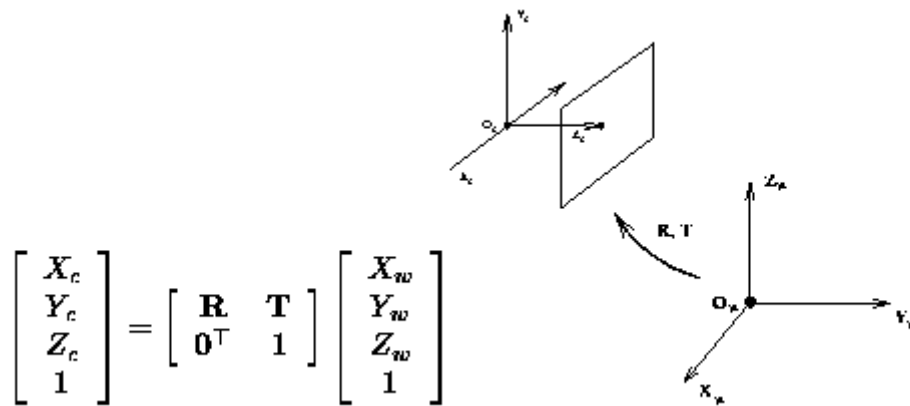
Рисунок

Матрица C – матрица камеры размерностью 3×3 :

$$C = \begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

- α_u, α_v – описывают фокусное расстояние камеры
- u_0, v_0 – описывают положение центра (см. рисунок). В случае идеальной камеры $\alpha_u = \alpha_v = f, v_0 = u_0 = 0$

Внешние параметры определяют положение и поворот самой камеры в трёхмерном пространстве:



Таким образом, проецирование точки в трехмерном пространстве на плоскость камеры осуществляется следующим образом:

$$\mathbf{x} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{C} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$= \mathbf{C} [\mathbf{R} | \mathbf{T}] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Где матрица проекции: $P = C[R|T]$ [2]

Помимо этого, реальная камера имеет оптические искажения, которые не зависят от рассматриваемой сцены и описываются параметрами дисторсии:

$$\begin{aligned} x'' &= x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2 (r^2 + 2x'^2) \\ y'' &= y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1 (r^2 + 2y'^2) + 2p_2 x' y' \end{aligned}$$

k_1, k_2, p_1, p_2, k_3 – коэффициенты дисторсии. [1]

2.2 Построение облака точек с использованием OpenCV

В ходе работы нами были рассмотрены два способа для построения облака точек по нескольким, а именно по двум, изображениям.

Оба этих подхода осуществляются по следующему общему алгоритму:

1. Для точек на одном изображении находятся соответствующие точки на втором изображении
2. По полученным смещениям пар точек с использованием матриц камер строится облако точек

2.2.1 Оптический поток и метод triangulate

Одним из рассмотренных нами подходов заключался в нахождении вышеупомянутых соответствий с использованием оптического потока.

2.2.1.1 Алгоритм построения облака точек

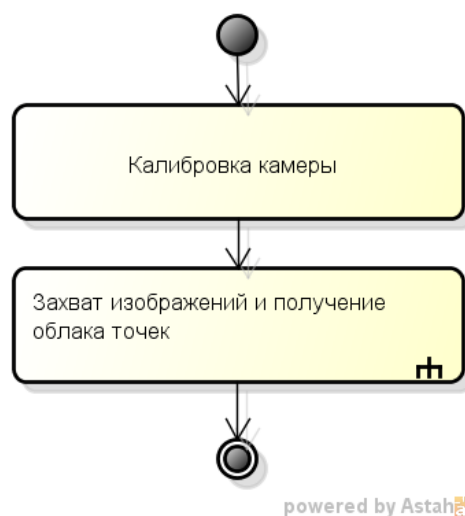


Рисунок 2.3 – общий алгоритм построения облака точек

Калибровка камеры позволяет определить внешние и внутренние параметры камеры. Она должна выполняться один раз для камеры.

Захват изображений и получение облака точек выполняется каждый раз при создании облака точек рассматриваемого объекта.

Алгоритм построения облака точек:

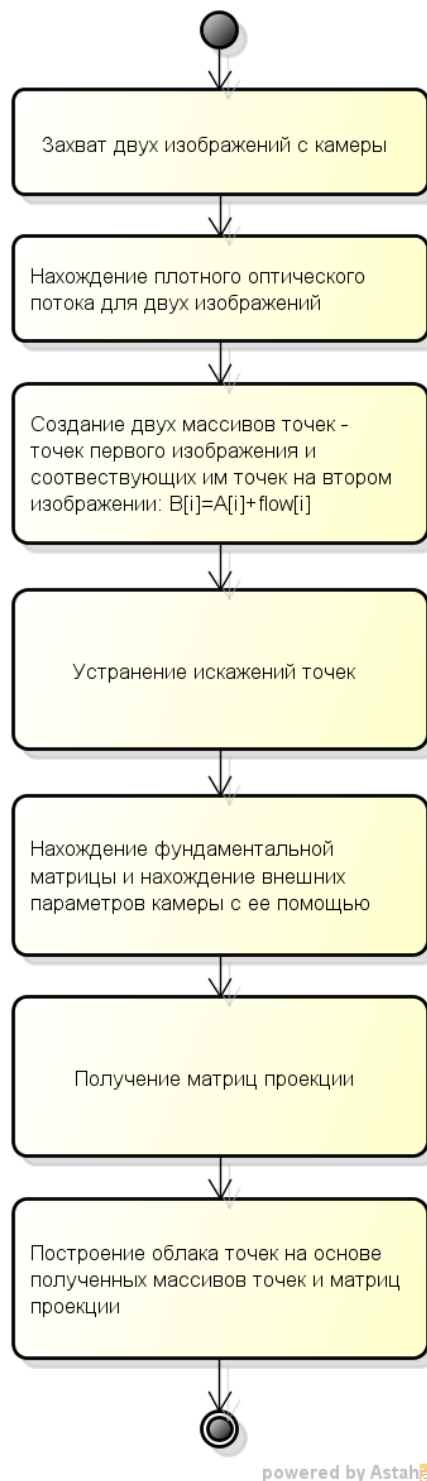


Рисунок 2.4 – алгоритм построения облака точек

2.2.1.2 Оптический поток

Оптический поток (ОП) – изображение видимого движения, представляющее собой сдвиг каждой точки между двумя изображениями. По сути, он представляет собой поле скоростей.

Для каждой точки изображения $I_1(x, y)$ находится такой сдвиг (dx, dy) , чтобы исходной точке соответствовала точка на втором изображении $I_2(x + dx, y + dy)$.

В OpenCV оптический поток можно рассчитать разными алгоритмами. В данном алгоритме использовался алгоритмом Farneback, который рассчитывает т.н. плотный оптический поток – для каждой точки изображения.

Прототип функции calcOpticalFlowFarneback[5]:

```
void calcOpticalFlowFarneback(InputArray prev, InputArray next, InputOutputArray  
flow, double pyr_scale, int levels, int winsize, int iterations, int poly_n, double  
poly_sigma, int flags)
```

Пример работы алгоритма Farneback:

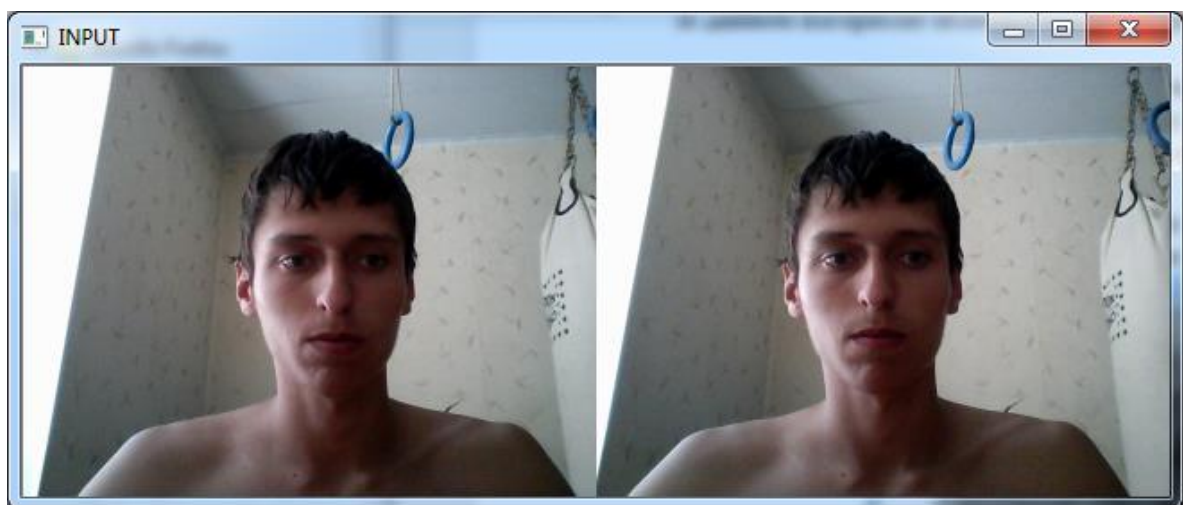


Рисунок 2.5 – входные данные для тестирования оптического потока

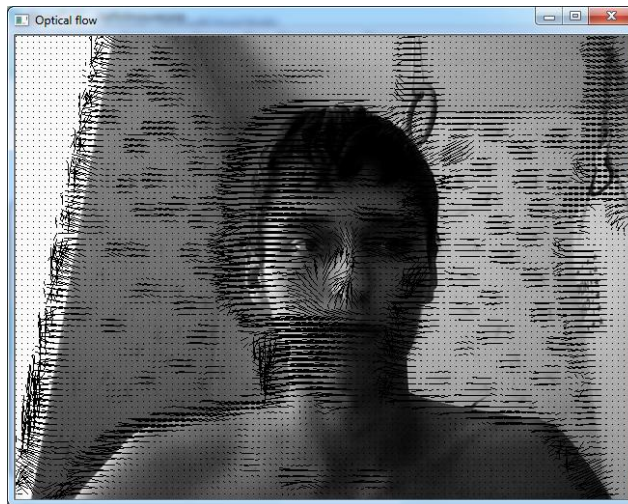


Рисунок 2.6 – визуализация оптического потока

2.2.1.2.1 Особенности реализации

Получив оптический поток, мы можем составить 2 массива:

- Массив точек на первом изображении
- Массив соответствующих точек на втором изображении

Код, обрабатывающий оптический поток и формирующий массивы точек

```
vector<Point2f> left_points, right_points;
for (int y = 0; y < img1.rows; y+=6) {
    for (int x = 0; x < img1.cols; x+=6) {
        //Поток - расстояние между точками на левом и правом изображениях
        Point2f flow = flow_mat.at<Point2f>(y, x);

        //Если оптический поток слишком мал, пренебрегаем им
        if (fabs(flow.x) < 0.1 && fabs(flow.y) < 0.1)
            continue;

        left_points.push_back(Point2f(x, y));
        right_points.push_back(Point2f(x + flow.x, y + flow.y));
    }
}
```

Этот код с шагом в 6 по x и y отбрасывает точки с малым оптическим потоком. Он затрагивает не все точки (выбранных точек достаточно).

2.2.1.3 Триангуляция точек для создания облака точек

Функция `triangulatePoints` библиотеки OpenCV восстанавливает 3D точки при помощи триангуляции.

Прототип функции `triangulatePoints[3]`

```
void triangulatePoints(InputArray projMatr1, InputArray projMatr2, InputArray  
projPoints1, InputArray projPoints2, OutputArray points4D)
```

Помимо массивов соответствующих друг другу точек с первого и второго изображений, она требует также матрицы проекций первой и второй камеры.

2.2.1.3.1 Особенности реализации

2.2.1.3.1.1 1 способ получения матриц проекции:

Получить эти матрицы из функции `stereoRectify`. Этот подход будет наиболее правильным. Но функция `stereoRectify` требует данных, которые нужно получить от функции `stereoCalibrate`, что означает предварительную калибровку 2 камер.

2.2.1.3.1.2 2 способ получения матриц проекции:

Получение матриц проекции при помощи `findFundamentalMat[6]`:

```
/* Try to find essential matrix from the points */  
Mat fundamental = findFundamentalMat(left_points, right_points, cv::FM_RANSAC, 3.0,  
0.99);  
Mat essential = cam_matrix.t() * fundamental * cam_matrix;  
  
/* Find the projection matrix between those two images */  
SVD svd(essential);[x3  
static const Mat W = (Mat_<double>(3, 3) <<  
    0, -1, 0,  
    1, 0, 0,  
    0, 0, 1);  
  
static const Mat W_inv = W.inv();  
  
Mat_<double> R1 = svd.u * W * svd.vt;  
Mat_<double> T1 = svd.u.col(2);  
  
Mat_<double> R2 = svd.u * W_inv * svd.vt;  
Mat_<double> T2 = -svd.u.col(2);  
  
static const Mat P1 = Mat::eye(3, 4, CV_64FC1);  
Mat P2 = (Mat_<double>(3, 4) <<  
    R1(0, 0), R1(0, 1), R1(0, 2), T1(0),  
    R1(1, 0), R1(1, 1), R1(1, 2), T1(1),  
    R1(2, 0), R1(2, 1), R1(2, 2), T1(2));
```

2.2.1.3.1.2.3 Построение облака точек

На выходе мы получаем массив точек в общих координатах (x, y, z, w) . Остается только перевести их в Евклидово пространство путем деления каждой координаты на w .

Получаем $(x/w, y/w, z/w, w/w)$. Сохраняем первые три координаты для каждой точки и получаем таким образом облако точек.

Полный код построения оптического потока см. в приложении Е.

2.2.1.4 Результаты

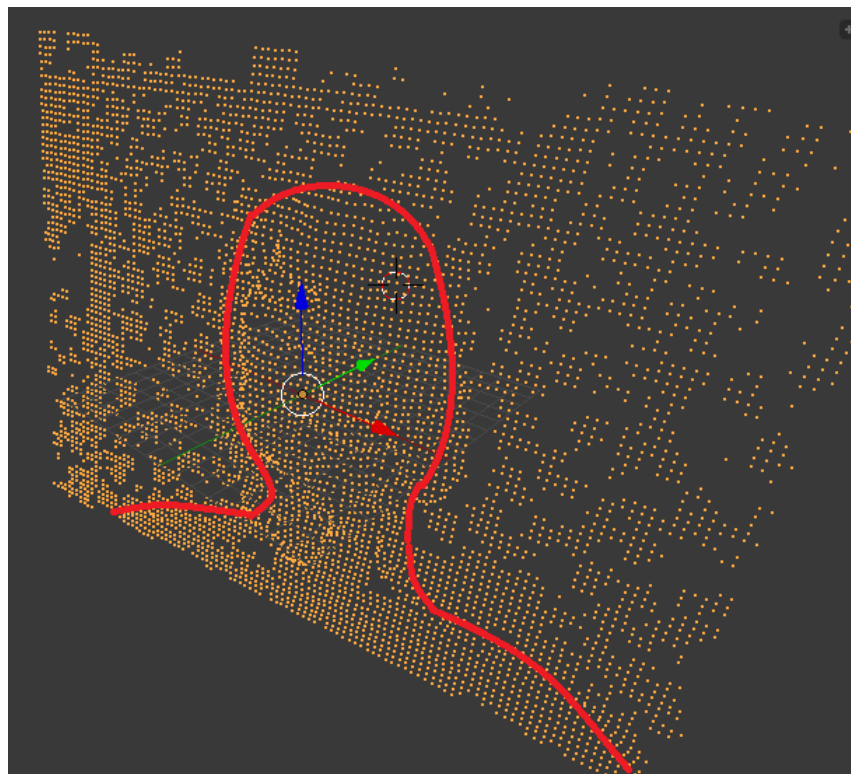


Рисунок 2.7 – результат построения облака точек

Недостатки:

- Слабая детализация
- Нечеткое облако (объекты получаются смазанными)
- Малая глубина облака

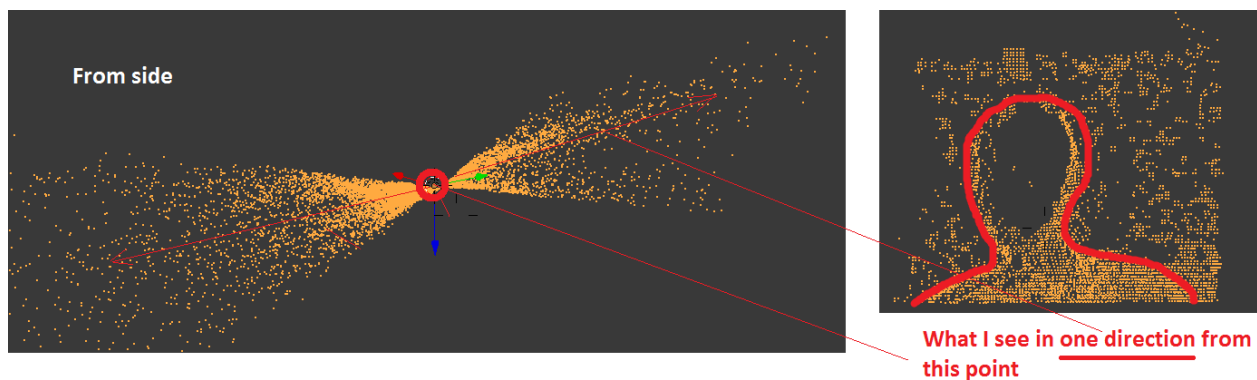


Рисунок 2.8 – результат построения облака точек

Это два тетраэдра, расположенные друг напротив друга.

Если переместиться в центр облака (на картинке ниже отвечен красным кружком) и посмотреть в направлении основания одного из тетраэдров, то можно будет увидеть объекты с фотографии.

Нами не удалось выяснить причины появления такого варианта облака точек.

2.2.2 Стерео калибровка и Block Matching

2.2.2.1 Общий принцип

Следующий вариант заключается в применении алгоритмов Block Matching для нахождения соответствий между точками для построения карты различий (distority map), которая используется методом `reprojectImageTo3d` для построения облака точек. [3]

Алгоритм построения облака точек:

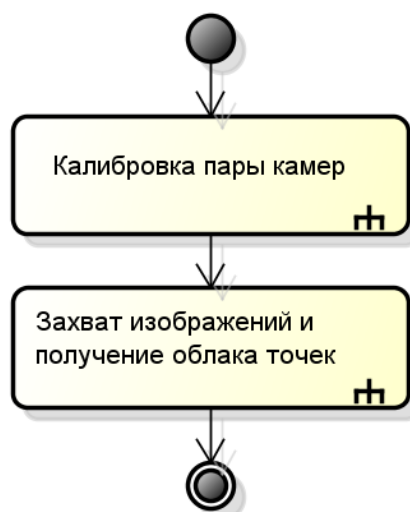


Рисунок 2.9 – общий алгоритм построения облака точек

Библиотека OpenCV имеет два алгоритма: Block Matching, который реализуется классом `StereoBM` и Semi-Global Block Matching, который реализуется классом `StereoSGBM`. Второй выдает лучшие результаты, но первый более производительный. [3]

Для работы этих алгоритмов, изображения должны быть выравнены таким образом, чтобы соответствующие точки двух изображений лежали на одной линии. Для выравнивания и функции `reprojectImageTo3d` требуются внешние и внутренние параметры камер. Они определяются в результате калибровки.

Данный метод был реализован в виде класса `StereoVision`, который предоставляет методы для калибровки камер, нахождения оптического потока и сохранения настроек.

2.2.2.2 Калибровка камер

Алгоритм калибровки:

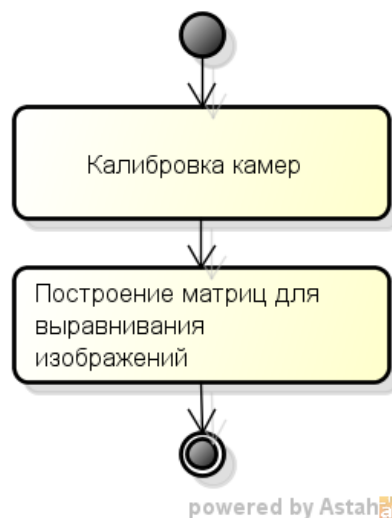


Рисунок 2.10 – алгоритм стерео калибровки

В результате первого действия определяются внутренние и внешние параметры камер. Данные параметры могут быть сохранены в файл для дальнейшего использования. Второй этап калибровки заключается в расчете матриц, необходимых для трансформации изображений с целью устранения искажений и выравнивания. Данные матрицы имеют размер, равный разрешению изображения, поэтому они не сохраняются в файл, а вычисляются заново каждый раз после его открытия, а затем используются в течении сеанса работы с программой.

Библиотека OpenCV имеет функцию, которая определяет внешние и внутренние параметры двух камер — `stereoCalibrate`. [3] На вход этой функции подаются координаты опорных точек на изображениях с обеих камер, а так же координаты этих точек в трехмерном пространстве. Классическим решением является использование шахматной доски — углы между квадратами легко определяются и используются в качестве опорных точек. В качестве реальных координат этих точек применяются координаты $(x, y, 0)$, x — индекс точки по горизонтали, y — индекс точки по вертикали, иногда $(x * a, y * a, 0)$, a — физический размер квадрата шахматной доски. В разрабатываемой программе мы используем на данный момент первый вариант.

A photograph of a person holding a board with a checkerboard pattern. Overlaid on the board is a sequence of colored dots (red, orange, yellow, green, cyan, blue) connected by lines, illustrating a sequence of points or a path.

В противном случае возможно появление сильных искажений, см. рисунок 2.12.



Рисунок 2.12 – искажения при неудачной калибровке

Зная внешние и внутренние параметры камер можно выровнять исходные изображения, видно, что одни и те же точки не лежат на одной линии.



Рисунок 2.13 – исходная пара изображений

Выравненные изображения, видно, что одни и те же точки лежат на одной линии:



Рисунок 2.14 – выровненная пара изображений

2.2.2.2.1 Особенности реализации

За калибровку отвечает метод StereoVision::Calibrate ((полный код функции см. в приложении А):

```
/* Калибровка стерео-камеры (пары камер)
 * param[in] left - изображение с левой камеры (CV_8UC1 - серое)
 * param[in] right - изображение с правой камеры
 * param[in] patternSize - число углов шахматной доски
 (число квадратов на стороне - 1)
 * result - успех калибровки
 */
bool Calibrate(const vector<Mat>& left, const vector<Mat>& right, cv::Size
patternSize);
```

На вход функции подается два вектора изображений – с левой и правой камер. Для всех изображений осуществляется поиск углов при помощи функции OpenCV findChessboardCorners [3]:

```
bool isFoundLeft = cv::findChessboardCorners(left[i], patternSize, imagePointsLeftSingle,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);

bool isFoundRight = cv::findChessboardCorners(right[i], patternSize, imagePointsRightSingle,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);
```

Найденные координаты точек и реальные координаты точек добавляются в вектора:

```
imagePointsLeft.push_back(imagePointsLeftSingle);
imagePointsRight.push_back(imagePointsRightSingle);
objectPoints.push_back(objectPointsSingle);
```

Полученные данные передаются в функцию stereoCalibrate для получения внутренних и внешних параметров камеры:

```
stereoCalibrate(objectPoints, imagePointsLeft, imagePointsRight,
CM1, D1, CM2, D2, imSize, R, T, E, F, CV_CALIB_SAME_FOCAL_LENGTH |
CV_CALIB_ZERO_TANGENT_DIST,
cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5));
```

Полученные внешние и внутренние параметры камер передаются в функцию stereoRectify для получения матриц проекции:

```
cv::stereoRectify(CM1, D1, CM2, D2, imSize, R, T, R1, R2, P1, P2, Q);
```

Полученные данные сохраняются в объекте класса StereoCalibData, который служит для удобного хранения всех параметров калибровки и предоставляет методы для сохранения их в файл и загрузки из файла.

Для выравнивания изображения требуется пара матриц, вычисляемых функцией OpenCV initUndistortRectifyMap. Вычисление этих матриц осуществляется после

осуществления калибровки или после загрузки сохраненных результатов калибровки в конструкторе класса StereoVision.

2.2.2.3 Алгоритм построения облака точек

При наличии откалиброванных камер, построение облака точек по двум изображениям осуществляется по следующему алгоритму:



Рисунок 2.15 – алгоритм построения облака точек

2.2.2.4 Построение карты различий

Полученные выравненные изображения подаются на вход алгоритма Block Matching, который строит карту различий – черно-белое изображение, где яркость обозначает, как сильно отличаются соответствующие точки на двух изображениях, и, соответственно, расстояние до этой точки. Формально, это не является картой глубины, но сильно на нее похоже. Качество результата сильно зависит от настроек алгоритма.

Примеры карт различий:

1. StereoBM

Без настройки алгоритм выдает неверное и крайне шумное изображение

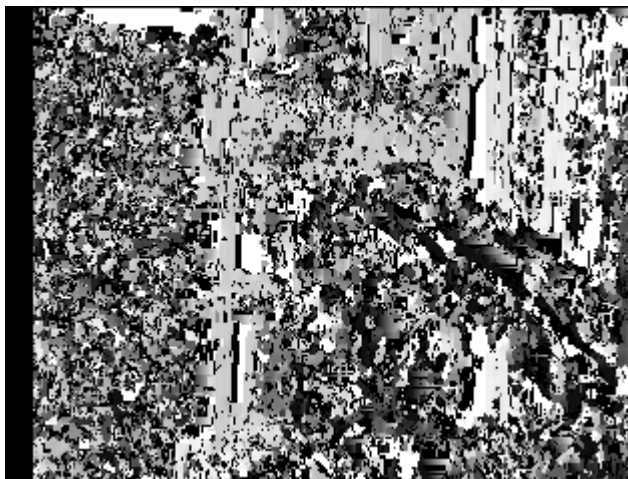


Рисунок 2.16 – карта различий

2. StereoBM

Небольшое снижение шумов позволяет получить карту смещения

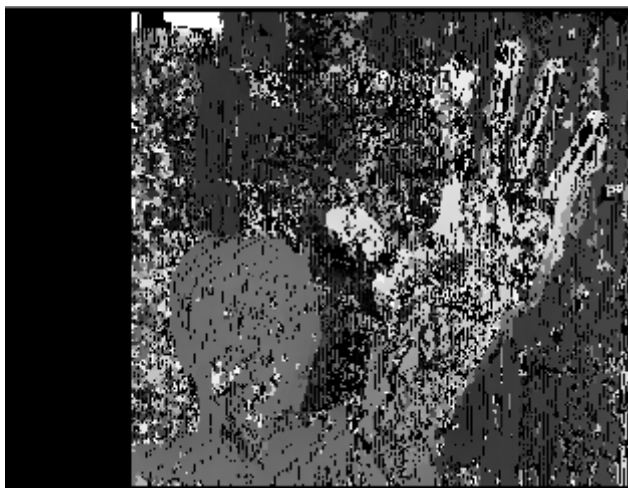


Рисунок 2.17 – карта различий

Облако точек крайне зашумлено, но детализировано.

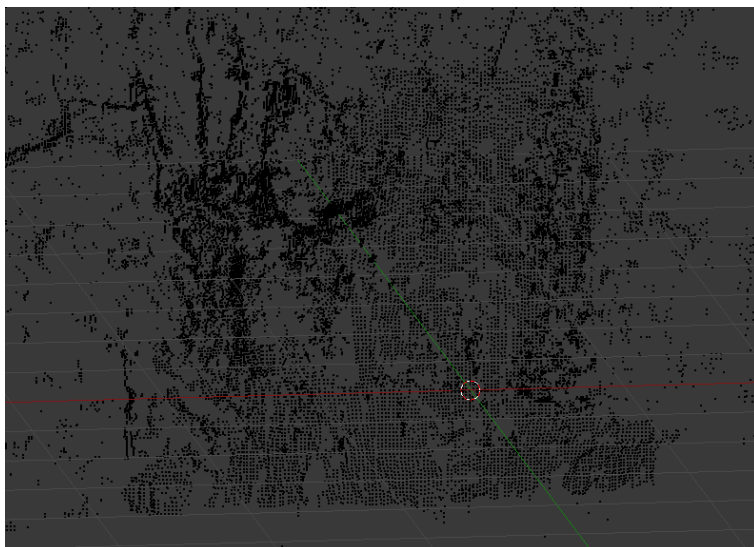


Рисунок 2.17 – пример облака точек

3. StereoBM

Уменьшение шумов приводит к размытию

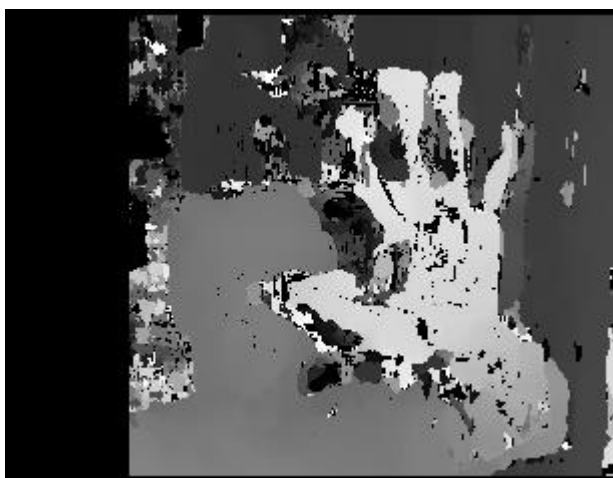


Рисунок 2.18 – карта различий

Облако точек менее шумное, но возможно детализованное, объемные неровности сглаживаются

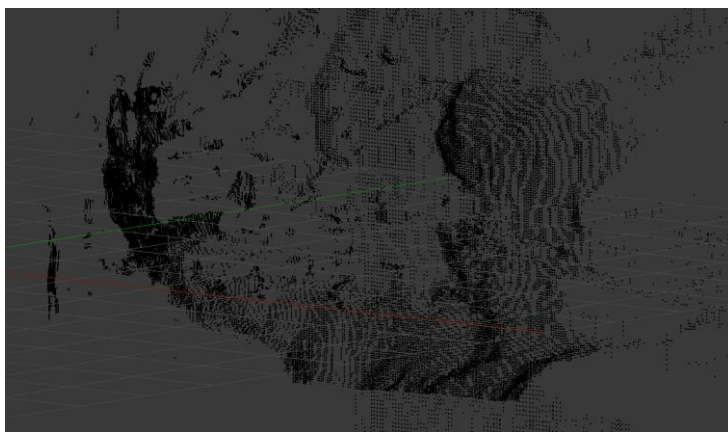


Рисунок 2.19 – пример облака точек

4. StereoSGBM

Качественная карта различий, небольшое количество шумов в пространстве, но шумные поверхности объектов.



Рисунок 2.20 – карта различий

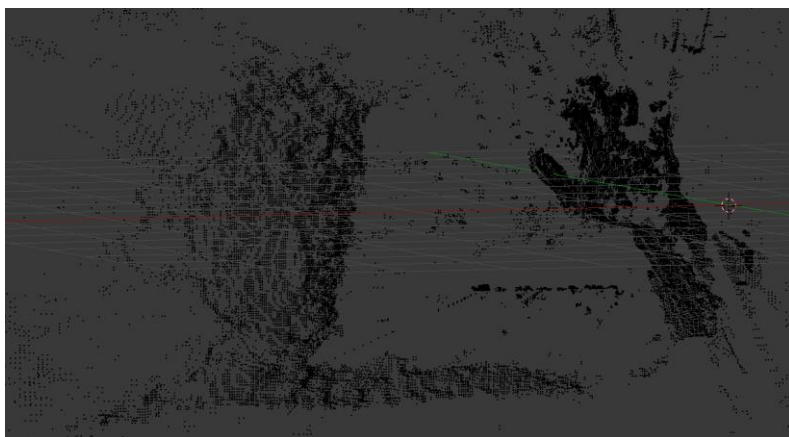


Рисунок 2.21 – пример облака точек

Для уменьшения шумов и повышения точности построения поверхности объектов, мы пытались усреднять карты различий, полученных на основе серии пар снимков, снятых через короткий промежуток времени, но данный метод не привел к заметному улучшению результатов.

2.2.2.4.1 Особенности реализации

За построение облака точек отвечает метод StereoVision::CalculatePointCloud (полный код функции см. в приложении В):

```
/* Строит облако точек по парам изображений
 * param[in] left - вектор изображений с левой камеры
 * param[in] right - вектор изображений с правой камеры
 * param[in] noDisparityOut - флаг, указывающий, что не надо копировать карту различий в
                        disparityResult
 * param[out] disparityResult - результирующая карта различий
 * param[in] - disparityOnly - генерировать только карту различий без облака точек
 * result - указатель на облако точек
 */
PointCloudStorage* _calculatePointCloud(const vector<Mat> & left,
                                       const vector<Mat> & right,
                                       bool noDisparityOut,
                                       cv::Mat& disparityResult,
                                       bool disparityOnly) const;
```

Этот метод позволяет как строить облако точек, так и строить лишь карту различий, без построения облака точек, что применимо для вывода карты различий в реальном времени.

Для удобства использования, данный приватный метод имеет несколько публичных оберток:

- Без параметра disparityResult (noDisparityOutput = true) – не выводит карту различий
- С параметров disparityResult (noDisparityOutput = false) – выводит карту различий
- Два аналогичных метода, принимающие на вход не vector<Mat> для левого и правого изображения, а просто два изображения типа Mat

Каждая пара изображений приводится в требуемый формат, выравнивается и используется для создания карты различий.

Перевод в одноканальное в оттенках серого, если исходное изображение было цветное:

```
if (left_image.channels() == 3)
    cv::cvtColor(left, leftGrey, CV_RGB2GRAY);
else
    leftGrey = left_image;
```



```

if (right_image.channels() == 3)
    cv::cvtColor(right, rightGrey, CV_RGB2GRAY);
else
    rightGrey = right_image;

```

Выравнивание изображений с использованием сгенерированных ранее матриц:

```

cv::remap(leftGrey, leftRemaped, calibData.LeftMapX, calibData.LeftMapY, cv::INTER_LINEAR,
cv::BORDER_CONSTANT, cv::Scalar());
cv::remap(rightGrey, rightRemaped, calibData.RightMapX, calibData.RightMapY, cv::INTER_LINEAR,
cv::BORDER_CONSTANT, cv::Scalar());

```

Построение карты различий и нормализация карты, чтобы привести ее к стандартному диапазону яркостей 0 – 255:

```

stereoMatcher->compute(rightRemaped, leftRemaped, disparity);
cv::normalize(disparity, normalDisparity, 0, 255, CV_MINMAX, CV_8U);

```

Полученный набор карт различий усредняется при помощи функции `average_disparity` (см. приложение C):

```

cv::Mat normal_disparity = StaticHelpers::average_disparity(disparities);

```

Полученная усредненная карта различий используется функцией `reprojectImageTo3D` [3], чтобы построить облако точек.

```

cv::reprojectImageTo3D(normal_disparity, cloud, calibData.Q, true);
return new PointCloudStorage(cloud.clone());

```

Класс `PointCloudStorage` является хранилищем облака точек и имеет методы для сохранения облака точек в файл и фильтрации шумов.

2.2.2.5 Фильтрация шумов

Для получения качественного результата, необходимо устранить посторонние объекты, которые получаются в облаке точек из-за шумов на карте различий.

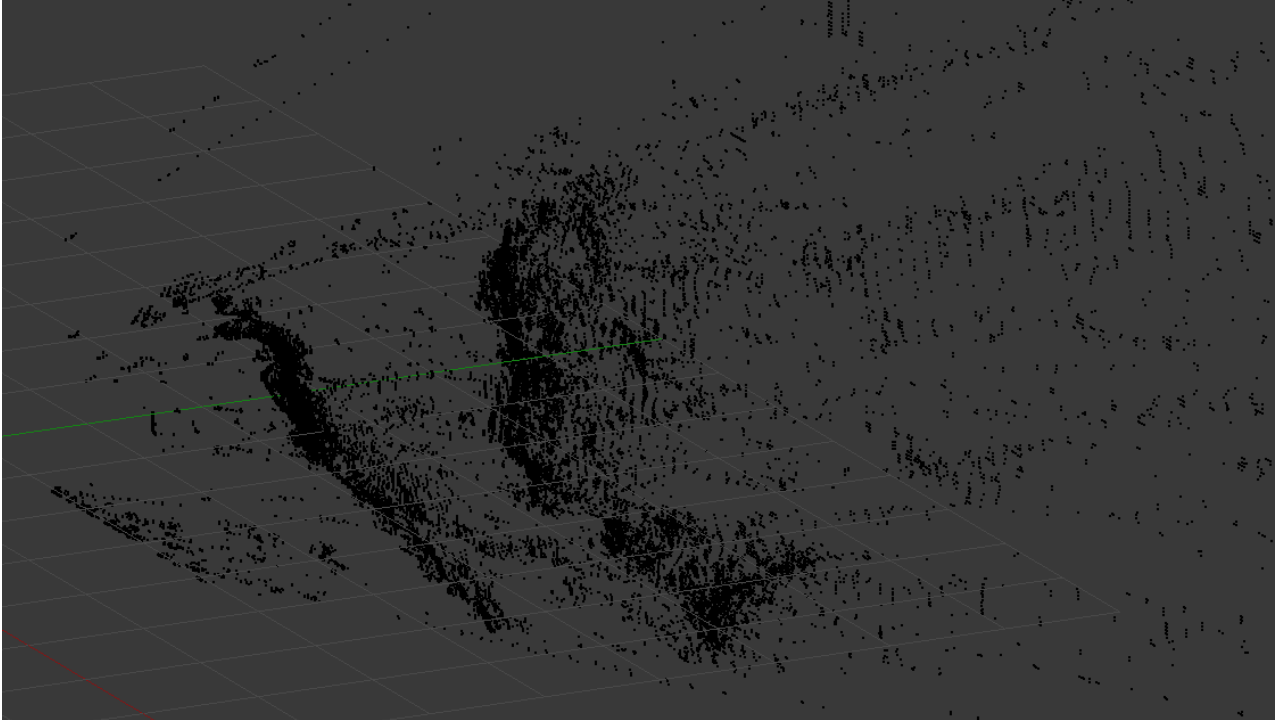


Рисунок 2.22 – шумы на облаке точек

На изображении облака точек видно, что шумы, в основном, представляют небольшие скопления точек в пространстве. На этом основан разработанный нами алгоритм устранения шумов:



Рисунок 2.23 – алгоритм удаления шумов

Разделение облака точек на объекты осуществляется путем объединения групп точек, расстояния между соседними членами которых не велико и не превосходит задаваемый пользователем предел. Таким образом, скопления точек будут выделены в отдельные объекты. Затем, маленькие объекты будут удалены.

В основе алгоритма разделения облака точек на объекты лежит следующее: облако точек, получаемое из функции `reprojectImageTo3D` представляет собой матрицу `cv::Mat`, с размерами, равными разрешению карты смещений, элементами которой являются точки с тремя координатами. Таким образом, соседние элементы этой матрицы будут являться точками с близкими по значениями координатами x и y (координата z – глубина – может существенно различаться). Таким образом, чтобы получить соседнюю точку, достаточно взять соседнюю ячейку матрицы.

Данный алгоритм позволяет разбить исходное облако точек на иерархическую систему объектов:

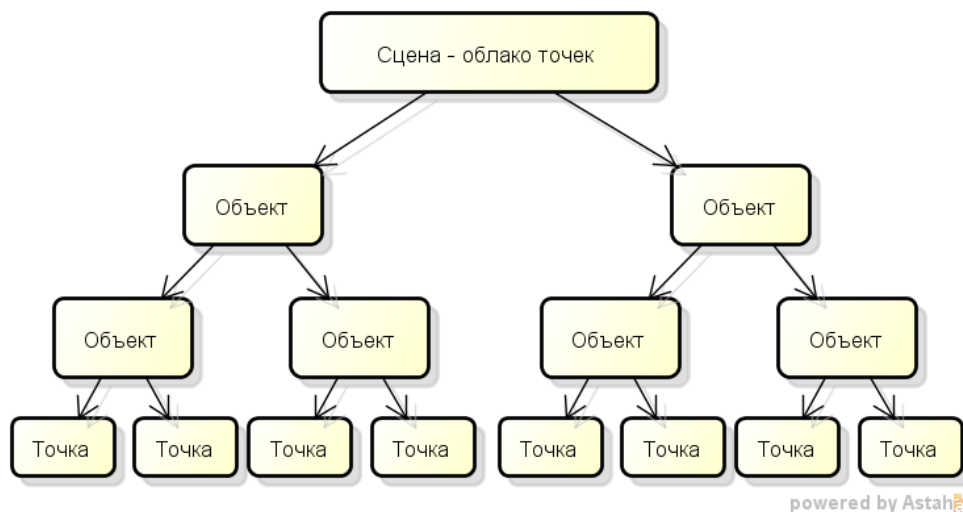


Рисунок 2.24 – иерархическая структура сцены

Для удобного просмотра разбиения на объекты и отладки был создан небольшой 3D просмотрщик на OpenGL с использованием библиотеки `freeglut`[4]

Результаты работы алгоритма разбиения и фильтрации, где разные цвета обозначают разные объекты.

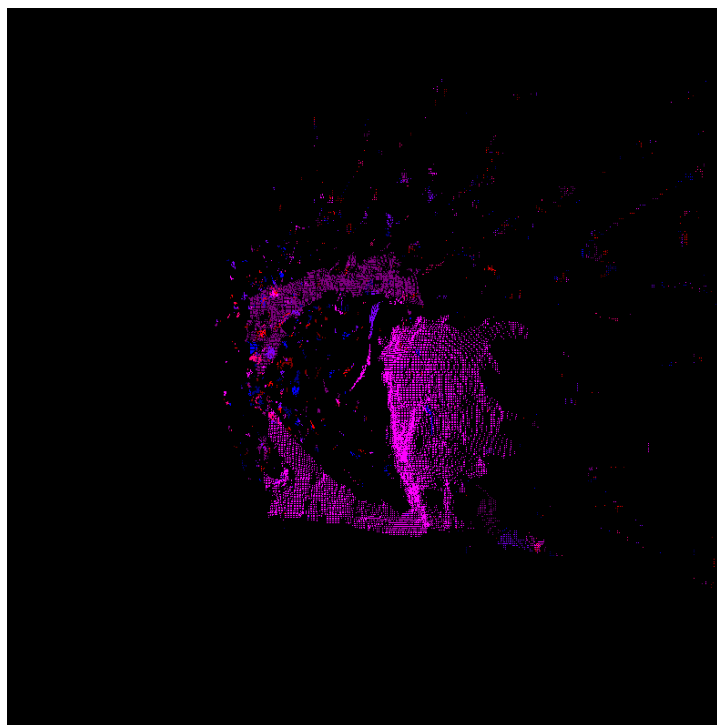


Рисунок 2.25 – облако точек, разделенное на объекты

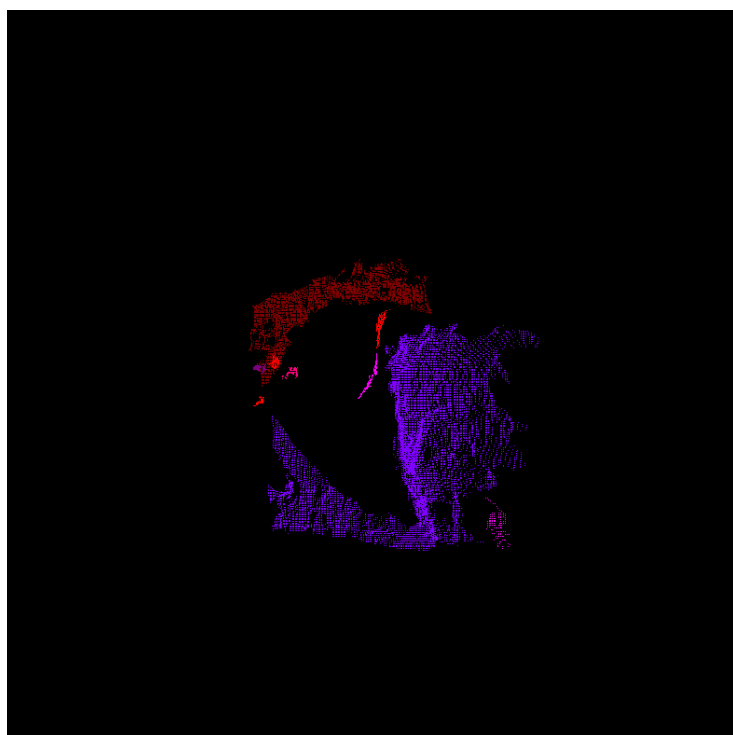


Рисунок 2.26 – облако точек с убраным шумом

На втором изображении видно, что количество шумов сильно уменьшилось.

2.2.2.5.1 Особенности реализации

Для реализации этой структуры данных используются следующие классы:

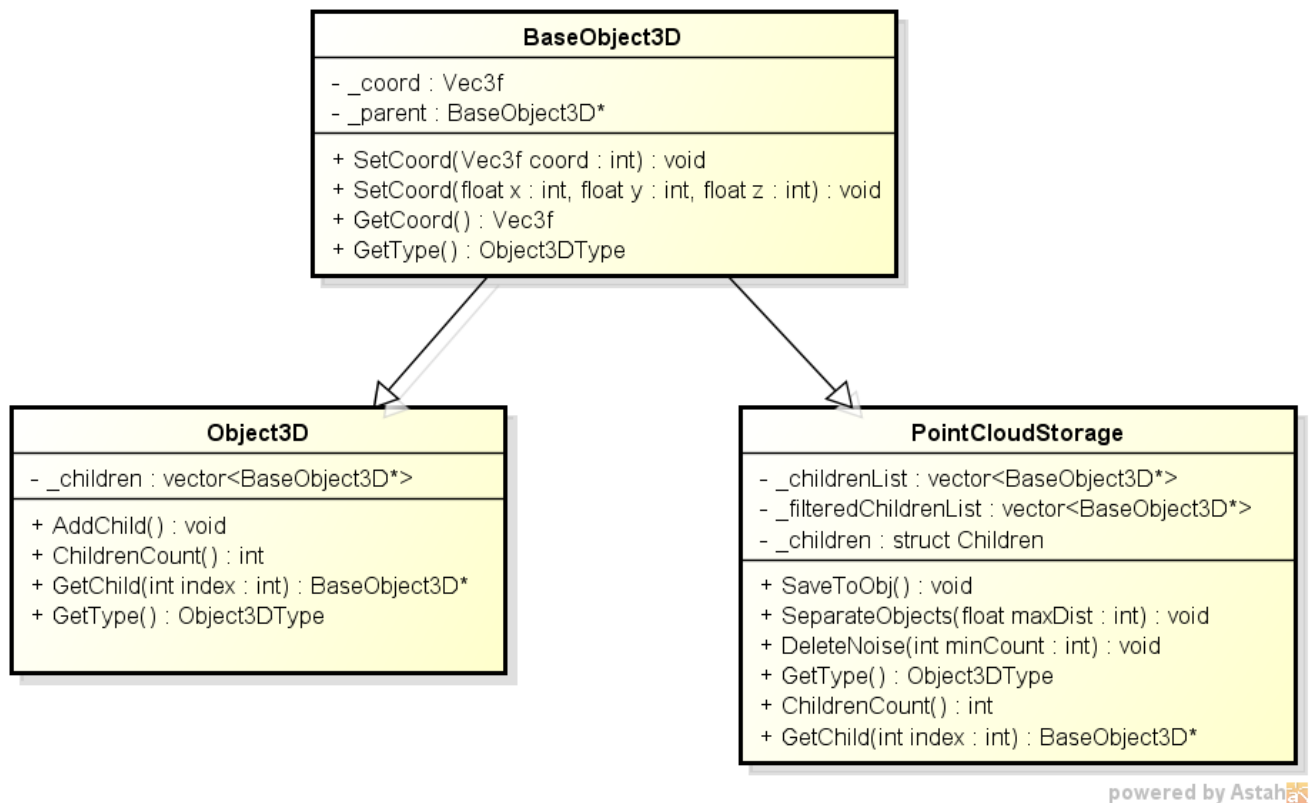
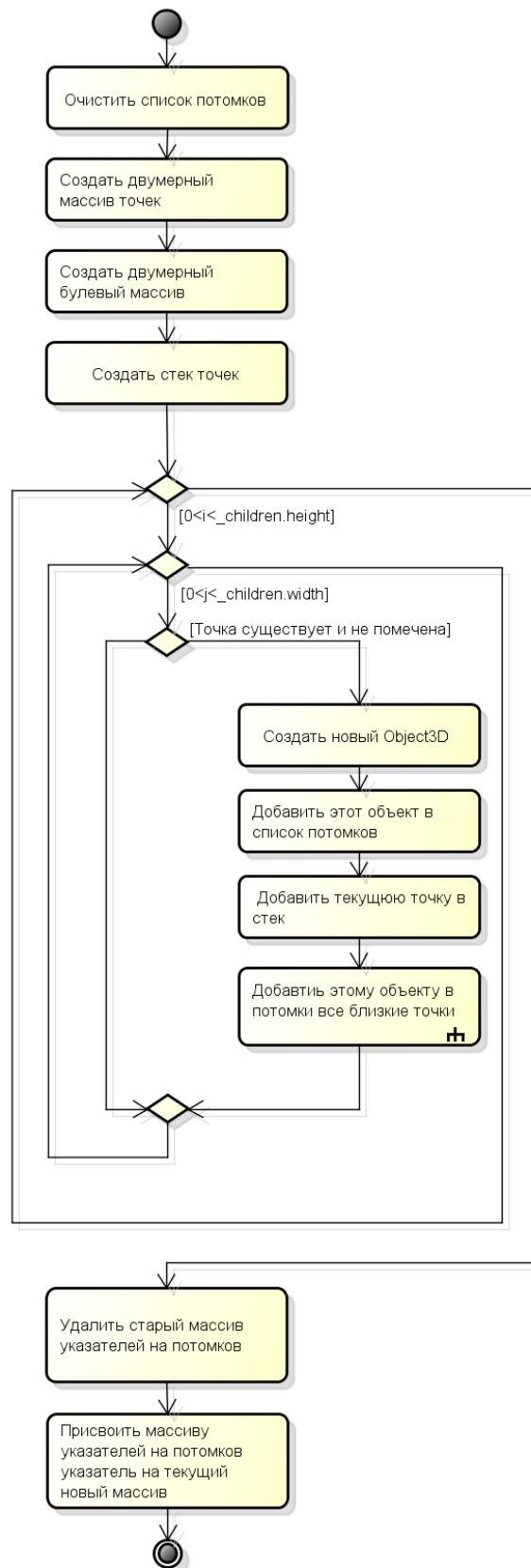


Рисунок 2.27 – диаграмма классов 3D объектов

- BaseObject3D – базовый класс для иерархии объектов, представляет собой точку
 - _coord – координаты объекта
 - _parent – указатель на родительский объект
 - SetCoord – задает координату
 - GetCoord – возвращает координату
 - GetType – возвращает тип объекта Object3DType::TYPE_POINT
- Object3D – представляет собой объект – промежуточный уровень иерархии объектов
 - _children – вектор указателей на потомков
 - AddChild – добавляет потомка в вектор и устанавливает себя в качестве значения _parent у потомка

- o ChildrenCount – возвращает число потомков
 - o GetChild – возвращает потомка под заданным индексом
 - o GetType – возвращает тип объекта Object3DType::TYPE_OBJECT
- PointCloudStorage – представляет собой хранилище облака точек – верхний уровень иерархии объектов
 - o _childrenList – список потомков
 - o _filteredChildrenList – список потомков, где нет шума
 - o _children – структура, состоящая из двумерного массива указателей на потомков и его размеров
 - o SaveToObj – сохраняет облако точек в файл типа .obj
 - o SeparateObjects – разделяет потомков на объекты
 - o DeleteNoise – удаляет маленьких потомков
 - o GetType – возвращает тип объекта Object3DType::TYPE_SCENE
 - o ChildrenCount – возвращает список потомков
 - o GetChild – возвращает потомка под заданным индексом

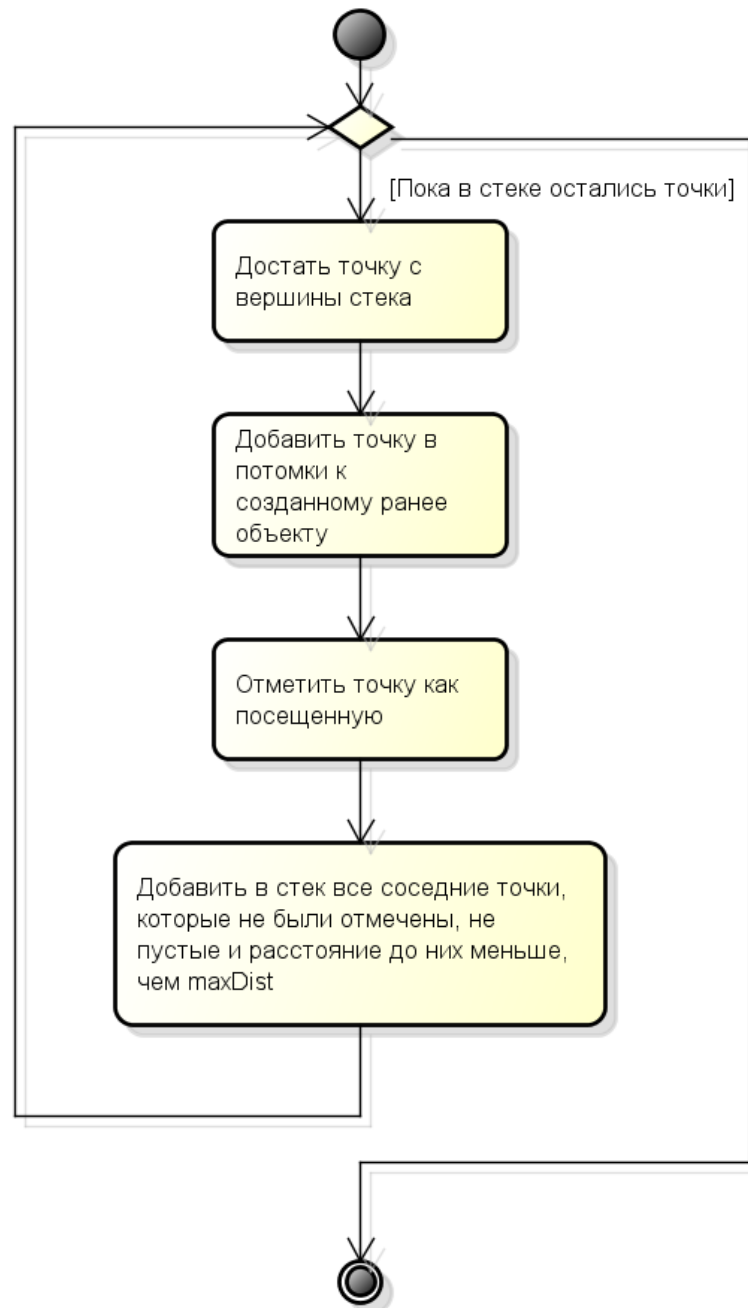
Алгоритм разделения объектов:



powered by Astah

Рисунок 2.28 – алгоритм разделения облака точек на объекты

Алгоритм добавления всех близких точек:



powered by Astah

Рисунок 2.29 – алгоритм поиска близких точек

Полный код функции разделения объектов см. в приложении D

3 Заключение

Перед нами была поставлена задача формирования 3Д модели по набору изображений. Для работы в данном направлении была выбрана библиотека OpenCV. Было найдено 2 способа решить задачу построения облака точек при помощи этой библиотеки. Первый заключается в нахождении оптического потока по 2 изображениям и последующем вызове функции triangulatePoints. Второй заключается в нахождении карты различий при помощи StereoBM/SGBM алгоритмов и вызове функции reprojectImageTo3D, которая использует карту различий для построения облака точек.

Второй способ (reprojectImageTo3D) позволяет получить более качественные облака точек, чем первый. Однако он требует и большего количества действий по сравнению с первым вариантом (triangulatePoints).

Сравнение способов:

1)reprojectImageTo3D

- +Построение карты различий как один из результатов
- +Качественное облако точек
- Более сложный процесс, чем во втором методе.

2)triangulatePoints

- +Относительно прост
- +Возможно использование лишь одной камеры
- Низкая точность
- Слабо детализированное облако точек (слабо выражена трехмерность)
- Иногда получаются совершенно неадекватные результаты (причина не выяснена)

Первый способ был выбран в качестве основного и доработан. Добавлена обработка карты различий и фильтрация полученного облака точек. А также реализовано выделение объектов в облаке точек.

4 Список использованной литературы

1. Основы стереозрения // Хабрахабр URL: <http://habrahabr.ru/post/130300/>
2. 3x4 Projection Matrix // INFORMATICS HOMEPAGES SERVER URL: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/EPSRC_SSAZ/norde3.html
3. Camera Calibration and 3D Reconstruction // OpenCV 2.4.11.0 documentation URL: http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
4. The Open-Source OpenGL Utility Toolkit (freeglut 3.0.0) Application Programming Interface // Freeglut URL: <http://freeglut.sourceforge.net/docs/api.php>
5. Motion Analysis and Object Tracking // OpenCV 2.4.11.0 documentation URL: http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.html
6. STRUCTURE FROM MOTION USING FARNEBACK'S OPTICAL FLOW PART 2 // URL: <http://subokita.com/2014/03/26/structure-from-motion-using-farnebacks-optical-flow-part-2/>

Приложение А: функция калибровки

```
bool StereoVision::Calibrate(const std::vector<cv::Mat>& left, const std::vector<cv::Mat>&
right, cv::Size patternSize)
{
    /* Алгоритм стерео-калибровки:
    1. Находим на них шахматную доску и определяем углы
    2. Создаем реальные координаты углов шахматной доски:
    считаем, что начальный угол имеет координаты (0,0,0), а остальные
    смещаются от него на заданное расстояние - размер клетки
    3. Выполняем stereoCalibrate, чтобы получить
    матрицы камер, дисторсию и матрицы перехода между камерами (R, T)
    4. Выполняем stereoRectify
    5. Находим матрицы устранения искажений
    */
    const int size = 1;

    cv::Size imSize;

    // Точки на изображении
    //Вектор найденных на изображении точек
    std::vector<cv::Point2f> imagePointsLeftSingle, imagePointsRightSingle;
    //Вектор векторов точек - по вектору на изображение (у нас по одному на камеру)
    std::vector<std::vector<cv::Point2f>> imagePointsLeft, imagePointsRight;

    // Реальные точки
    //Реальные координаты точек
    std::vector<cv::Point3f> objectPointsSingle;
    //Вектор векторов координат - по вектору на изображение
    std::vector<std::vector<cv::Point3f>> objectPoints;

    //Выходные параметры stereoCalibrate
    //CameraMatrix указывает как перейти от 3D точек в мировой СК в 2D точкам в изображении
    cv::Mat CM1(3, 3, CV_64FC1), CM2(3, 3, CV_64FC1);
    cv::Mat D1, D2; //Коэффициенты дисторсии
    cv::Mat R, //Матрица поворота между СК первой и второй камер
    T, //Вектор перехода между СК первой и второй камер
    E, //Существенная матрица (Устанавливает соотношения между
    //точками изображения????) (тут что-то связано с
    //эпиполярными линиями)
    F; //Фундаментальная матрица

    //Выходные параметры для stereoRectify
    cv::Mat R1, R2, //Матрицы поворота для исправления (3x3)
    P1, P2; //Проективная матрица в исправленной системе координат (3x4)
    //Первые 3 столбца этих матриц - новые матрицы камер
    cv::Mat Q; //Матрица перевода смещения в глубину

    //2. ----- Строим реальные координаты точек -----

    /* Здесь мы задаем "реальные" координаты углов шахматной доски. Т.к реальные координаты
    мы не знаем, то
    просто принимаем их такими:
    (0,0) (0,1) ...
    (1,0) (1,1) ...
    */

    for (int j = 0; j<patternSize.height*patternSize.width; j++)
        objectPointsSingle.push_back(cv::Point3f(j / patternSize.width,
```

```

j%patternSize.width, 0.0f));

//1. ----- Ищем шахматные доски на изображении -----

for (int i = 0; i < left.size(); i++)
{
    //Документация: http://goo.gl/kV8Ms1 (Используется минифицированные ссылки, потому что
    //исходная содержит пробелы)
    bool isFoundLeft = cv::findChessboardCorners(left[i], patternSize,
                                                imagePointsLeftSingle,
                                                CV_CALIB_CB_ADAPTIVE_THRESH |
                                                CV_CALIB_CB_FILTER_QUADS);

    bool isFoundRight = cv::findChessboardCorners(right[i], patternSize,
                                                  imagePointsRightSingle,
                                                  CV_CALIB_CB_ADAPTIVE_THRESH |
                                                  CV_CALIB_CB_FILTER_QUADS);

    imSize = left[i].size();

    if (isFoundLeft && isFoundRight)
    {
        std::cout << i<<" : success\n";

        //Уточняем углы (назначение параметров не известно)
        //Документация: http://goo.gl/7BjZKd
        cornerSubPix(left[i], imagePointsLeftSingle, cv::Size(11, 11),
                    cv::Size(-1, -1),
                    cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30,
                    0.1));
        cornerSubPix(right[i], imagePointsRightSingle, cv::Size(11, 11),
                    cv::Size(-1, -1),
                    cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30,
                    0.1));

        //Добавляем в вектор векторов
        imagePointsLeft.push_back(imagePointsLeftSingle);
        imagePointsRight.push_back(imagePointsRightSingle);
        objectPoints.push_back(objectPointsSingle);
    } }

    //Проверка, что углы были найдены
    if (objectPoints.size() == 0) return false;

//3. ----- Стерео калибровка -----
//Документация: http://goo.gl/mKCH63
stereoCalibrate(objectPoints, imagePointsLeft, imagePointsRight,
                CM1, D1, CM2, D2, imSize, R, T, E, F, CV_CALIB_SAME_FOCAL_LENGTH |
                CV_CALIB_ZERO_TANGENT_DIST,
                cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5));

cv::stereoRectify(CM1, D1, CM2, D2, imSize, R, T, R1, R2, P1, P2, Q);

// ----- Сохранение результатов -----

calibData.ImageSize = imSize;

```

```
calibData.LeftCameraMatrix = CM1.clone();
calibData.RightCameraMatrix = CM2.clone();
calibData.LeftCameraDistortions = D1.clone();
calibData.RightCameraDistortions = D2.clone();
calibData.LeftCameraRectifiedProjection = P1.clone();
calibData.RightCameraRectifiedProjection = P2.clone();
calibData.LeftCameraRot = R1.clone();
calibData.RightCameraRot = R2.clone();
calibData.Q = Q;

_createUndistortRectifyMaps(calibData);

return true;
}
```

Приложение В: функция построения облака точек

```
PointCloudStorage* StereoVision::_calculatePointCloud(const std::vector<cv::Mat> & left, const
std::vector<cv::Mat> & right, bool noDisparityOut, cv::Mat& disparityResult, bool
disparityOnly) const
{
    std::vector<cv::Mat> disparities;
    int pairs_count = left.size(); //Число пар изображений

    //Для каждого находим disparity
    for (int pair_index = 0; pair_index < pairs_count; pair_index++)
    {
        cv::Mat left_image = left[pair_index],
            right_image = right[pair_index];

        cv::Mat leftRemaped, rightRemaped;
        cv::Mat leftGrey, rightGrey;
        cv::Mat disparity, normalDisparity;

        //Цветное изображение обесцвечиваем
        if (left_image.channels() == 3)
            cv::cvtColor(left, leftGrey, CV_RGB2GRAY);
        else
            leftGrey = left_image;

        if (right_image.channels() == 3)
            cv::cvtColor(right, rightGrey, CV_RGB2GRAY);
        else
            rightGrey = right_image;

        //Выпрямляем
        cv::remap(leftGrey, leftRemaped,
            calibData.LeftMapX, calibData.LeftMapY, cv::INTER_LINEAR,
            cv::BORDER_CONSTANT, cv::Scalar());
        cv::remap(rightGrey, rightRemaped,
            calibData.RightMapX, calibData.RightMapY, cv::INTER_LINEAR,
            cv::BORDER_CONSTANT, cv::Scalar());

        //Строим карту различий
        //Надо передавать изображения наоборот. ХЗ зачем
        stereoMatcher->compute(rightRemaped, leftRemaped, disparity);
        cv::normalize(disparity, normalDisparity, 0, 255, CV_MINMAX, CV_8U);

        disparities.push_back(normalDisparity.clone());
    }

    //Усредняем все карты неровностей
    cv::Mat normal_disparity = StaticHelpers::average_disparity(disparities);
    cv::Mat cloud;

    if (!noDisparityOut)
    {
        disparityResult = normal_disparity.clone();
    }

    if (!disparityOnly)
    {

```

```
        //Создаем облако точек
        cv::reprojectImageTo3D(normal_disparity, cloud, calibData.Q, true);
        return new PointCloudStorage(cloud.clone());
    }
    else
    {
        //Не создаем облако точек
        return NULL;
    }
}
```


Приложение С: функция усреднения карты различий

```
cv::Mat StereoVision::average_disparity(std::vector<cv::Mat>& disparities)
{
    cv::Mat example_mat = disparities[0];
    cv::Mat average_disparity(example_mat);    //Среднее арифметическое всех элементов
    disparities

    int disparities_count = disparities.size();    //Число всех элементов disparities

    //Размеры каждой из матриц disparities
    int cols = example_mat.cols;
    int rows = example_mat.rows;

    cv::Mat d = disparities[0];
    auto a = d.type();

    //Среднее значение элемента матрицы с координатами x, y
    unsigned int average_element_value;

    //Усреднение по каждой точке каждого элемента disparity
    for (int x = 0; x < cols; x++)
    {
        for (int y = 0; y < rows; y++)
        {
            //Среднее арифметическое для всех элементов с координатами x, y
            average_element_value = 0;
            for (int i = 0; i < disparities_count; i++)
                average_element_value += disparities[i].at<unsigned char>(y, x);

            average_disparity.at<unsigned char>(y, x) = average_element_value /
                                                         (float)disparities_count;
        }
    }

    return average_disparity;
}
```

Приложение D: функция разделения объектов

```
void PointCloudStorage::SeparateObjects(float maxDist)
{
    _childrenList.clear();

    //1. Выделяем новый массив под указатели на потомков
    BaseObject3D*** tempArr = new BaseObject3D**[_children.height];
    for (int i = 0; i < _children.height; i++)
    {
        tempArr[i] = new BaseObject3D*[_children.width];
        for (int j = 0; j < _children.width; j++)
            tempArr[i][j] = NULL;
    }

    //2. Создаем матрицу размером с childrenArray, показывающую, посетили ли
    // мы уже ячейку
    bool** visited = new bool*[_children.height];
    for (int i = 0; i < _children.height; i++)
    {
        visited[i] = new bool[_children.width];
        for (int j = 0; j < _children.width; j++)
        {
            visited[i][j] = false;
        }
    }

    //Очередь индексов точек
    std::stack<Point> points;

    //3. Выполняем разбивку
    for (int i = 0; i < _children.height; i++)
    {
        for (int j = 0; j < _children.width; j++)
        {
            //Берем очередного потомка
            BaseObject3D* child = _children.childrenArray[i][j];
            if ((child != NULL) && !visited[i][j])
            {
                //Если он существует, то находим все близкие объекты и добавляем их
                //В новый объект-группу
                Object3D* newParent = new Object3D(this);
                _childrenList.push_back(newParent);
                points.push(Point(j, i));

                while (points.size() > 0)
                {
                    //Берем последнюю точку
                    Point currentPoint = points.top();
                    points.pop();

                    //Добавляем текущий объект в потомки новому объекту-группе
                    BaseObject3D* currentObject =
                        _children.childrenArray[currentPoint.Y][currentPoint.X];

                    newParent->AddChild(currentObject);
                    visited[currentPoint.Y][currentPoint.X] = true;
                }
            }
        }
    }
}
```

```

        //Добавляем в очередь всех близких соседей этого объекта
        for (int y = currentPoint.Y - 1; y <= currentPoint.Y + 1; y++)
        {
            for (int x = currentPoint.X - 1; x <= currentPoint.X + 1; x++)
            {
                if ((y>0) && (x > 0) && (y < _children.height - 1) &&
                    (x < _children.width - 1) && !visited[y][x])
                {
                    //std::cout << y << " "<<x << " yes\n";
                    BaseObject3D* object = _children.childrenArray[y][x];
                    if ((object != NULL) && isDistLess(*object, *currentObject, maxDist))
                    {
                        points.push(Point(x, y));
                    }
                }
            }
        }
    }
}

}

//Удаляем створый childrenArray
for (int i = 0; i < _children.height; i++)
    delete[] _children.childrenArray[i];
delete[] _children.childrenArray;

//Удаляем visited
for (int i = 0; i < _children.height; i++)
    delete[] visited[i];
delete[] visited;

_children.childrenArray = tempArr;
cv::imshow("wtf", img);

DeleteNoise(minCount);
}

```

Приложение Е: построение облака точек с использованием ОПТИЧЕСКОГО ПОТОКА

```
Mat sfm(Mat& img1, Mat& img2, OpticalFlowMatrices & of_matr)
{
    Mat gray1, gray2;
    cvtColor(img1, gray1, CV_BGR2GRAY);
    cvtColor(img2, gray2, CV_BGR2GRAY);

    Mat flow_mat;
    //Считает оптический поток
    //http://docs.opencv.org/master/dc/d6b/group__video__track.html#ga5d10ebbd59fe09c5f650289ec0ece5af
    calcOpticalFlowFarneback(gray1, gray2, flow_mat, 0.5, 3, 12, 3, 5, 1.2, 0);
    showOpticalFlow(flow_mat, gray1); //Отрисовать оптический поток

    //выбираем точки
    vector<Point2f> left_points, right_points;
    for (int y = 0; y < img1.rows; y += 6) {
        for (int x = 0; x < img1.cols; x += 6) {
            //Поток - расстояние между точками на левом и правом изображениях
            Point2f flow = flow_mat.at<Point2f>(y, x);

            //Если оптический поток слишком мал, пренебрегаем им
            if (fabs(flow.x) < 0.1 && fabs(flow.y) < 0.1)
                continue;

            left_points.push_back(Point2f(x, y));
            right_points.push_back(Point2f(x + flow.x, y + flow.y));
        }
    }

    //Исправляем изображения в зависимости от параметров камеры (intrinsic params & dist coefficients)
    undistortPoints(left_points, left_points, of_matr.CM1, of_matr.D1);
    undistortPoints(right_points, right_points, of_matr.CM2, of_matr.D2);

    Mat point_cloud;
    triangulatePoints(of_matr.P1, of_matr.P2, left_points, right_points, point_cloud);
    from_homogenous(point_cloud);

    return point_cloud;
}
```