```
class CV_EXPORTS_W SeamFinder
{
public:
    CV_WRAP    virtual ~SeamFinder() {}
    enum { NO, VORONOI_SEAM, DP_SEAM };
    /** @brief Estimates seams.

    @param src Source images
    @param corners Source image top-left corners
    @param masks Source image masks to update
     */
    CV_WRAP virtual void find(const std::vector<UMat> &src, const std::vector<Point> &corners,
                              CV_IN_OUT std::vector<UMat> &masks) = 0;
    CV_WRAP static Ptr<SeamFinder> createDefault(int type);
};
/** @brief Stub seam estimator which does nothing.
 */
class CV_EXPORTS_W NoSeamFinder : public SeamFinder
{
public:
    CV_WRAP void find(const std::vector<UMat>&, const std::vector<Point>&, CV_IN_OUT
std::vector<UMat>&) CV_OVERRIDE {}
};
/** @brief Base class for all minimum graph-cut-based seam estimators.
 */
class CV_EXPORTS GraphCutSeamFinderBase
{
public:
    enum CostType { COST_COLOR, COST_COLOR_GRAD };
};
/** @brief Minimum graph cut-based seam estimator. See details in @cite V03 .
 */
class CV_EXPORTS_W GraphCutSeamFinder : public GraphCutSeamFinderBase, public SeamFinder
{
public:
    GraphCutSeamFinder(int cost_type = COST_COLOR_GRAD, float terminal_cost = 10000.f,
                       float bad_region_penalty = 1000.f);
    CV_WRAP GraphCutSeamFinder(String cost_type,float terminal_cost = 10000.f,
        float bad_region_penalty = 1000.f);

    ~GraphCutSeamFinder();
```

```
        CV_WRAP void find(const std::vector<UMat> &src, const std::vector<Point> &corners,
                          std::vector<UMat> &masks) CV_OVERRIDE;

private:
        // To avoid GCGraph dependency
        class Impl;
        Ptr<PairwiseSeamFinder> impl_;
};
/** @brief Base class for all pairwise seam estimators.
  */
class CV_EXPORTS_W PairwiseSeamFinder : public SeamFinder
{
public:
        CV_WRAP virtual void find(const std::vector<UMat> &src, const std::vector<Point> &corners,
                                  CV_IN_OUT std::vector<UMat> &masks) CV_OVERRIDE;

protected:
        void run();
        /** @brief Resolves masks intersection of two specified images in the given ROI.

        @param first First image index
        @param second Second image index
        @param roi Region of interest
          */
        virtual void findInPair(size_t first, size_t second, Rect roi) = 0;

        std::vector<UMat> images_;
        std::vector<Size> sizes_;
        std::vector<Point> corners_;
        std::vector<UMat> masks_;
};
Ptr<SeamFinder> SeamFinder::createDefault(int type)
{
        if (type == NO)
                return makePtr<NoSeamFinder>();
        if (type == VORONOI_SEAM)
                return makePtr<VoronoiSeamFinder>();
        if (type == DP_SEAM)
                return makePtr<DpSeamFinder>();
        CV_Error(Error::StsBadArg, "unsupported seam finder method");
```

```
}
void PairwiseSeamFinder::find(const std::vector<UMat> &src, const std::vector<Point> &corners,
                              std::vector<UMat> &masks)
{
    LOGLN("Finding seams...");
    if (src.size() == 0)
        return;

#if ENABLE_LOG
    int64 t = getTickCount();
#endif

    images_ = src;
    sizes_.resize(src.size());
    for (size_t i = 0; i < src.size(); ++i)
        sizes_[i] = src[i].size();
    corners_ = corners;
    masks_ = masks;
    run();

    LOGLN("Finding seams, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
}
void PairwiseSeamFinder::run()
{
    for (size_t i = 0; i < sizes_.size() - 1; ++i)
    {
        for (size_t j = i + 1; j < sizes_.size(); ++j)
        {
            Rect roi;
            if (overlapRoi(corners_[i], corners_[j], sizes_[i], sizes_[j], roi))
                findInPair(i, j, roi);
        }
    }
}
template <typename T>
float diffL2Square3(const Mat &image1, int y1, int x1, const Mat &image2, int y2, int x2)
{
    const T *r1 = image1.ptr<T>(y1);
    const T *r2 = image2.ptr<T>(y2);
    return static_cast<float>(sqr(r1[3*x1] - r2[3*x2]) + sqr(r1[3*x1+1] - r2[3*x2+1]) +
                              sqr(r1[3*x1+2] - r2[3*x2+2]));
```

```
}
template <typename T>
float diffL2Square4(const Mat &image1, int y1, int x1, const Mat &image2, int y2, int x2)
{
    const T *r1 = image1.ptr<T>(y1);
    const T *r2 = image2.ptr<T>(y2);
    return static_cast<float>(sqr(r1[4*x1] - r2[4*x2]) + sqr(r1[4*x1+1] - r2[4*x2+1]) +
                              sqr(r1[4*x1+2] - r2[4*x2+2]));
}
class GraphCutSeamFinder::Impl CV_FINAL : public PairwiseSeamFinder
{
public:
    Impl(int cost_type, float terminal_cost, float bad_region_penalty)
        : cost_type_(cost_type), terminal_cost_(terminal_cost),
bad_region_penalty_(bad_region_penalty) {}

    ~Impl() {}

    void find(const std::vector<UMat> &src, const std::vector<Point> &corners, std::vector<UMat>
&masks) CV_OVERRIDE;
    void findInPair(size_t first, size_t second, Rect roi) CV_OVERRIDE;

private:
    void setGraphWeightsColor(const Mat &img1, const Mat &img2,
                              const Mat &mask1, const Mat &mask2, GCGraph<float>
&graph);
    void setGraphWeightsColorGrad(const Mat &img1, const Mat &img2, const Mat &dx1, const
Mat &dx2,
                                  const Mat &dy1, const Mat &dy2, const Mat &mask1, const
Mat &mask2,
                                  GCGraph<float> &graph);

    std::vector<Mat> dx_, dy_;
    int cost_type_;
    float terminal_cost_;
    float bad_region_penalty_;
};
void GraphCutSeamFinder::Impl::find(const std::vector<UMat> &src, const std::vector<Point>
&corners,
                                    std::vector<UMat> &masks)
{
```

```
    // Compute gradients
    dx_.resize(src.size());
    dy_.resize(src.size());
    Mat dx, dy;
    for (size_t i = 0; i < src.size(); ++i)
    {
        CV_Assert(src[i].channels() == 3);
        Sobel(src[i], dx, CV_32F, 1, 0);
        Sobel(src[i], dy, CV_32F, 0, 1);
        dx_[i].create(src[i].size(), CV_32F);
        dy_[i].create(src[i].size(), CV_32F);
        for (int y = 0; y < src[i].rows; ++y)
        {
            const Point3f* dx_row = dx.ptr<Point3f>(y);
            const Point3f* dy_row = dy.ptr<Point3f>(y);
            float* dx_row_ = dx_[i].ptr<float>(y);
            float* dy_row_ = dy_[i].ptr<float>(y);
            for (int x = 0; x < src[i].cols; ++x)
            {
                dx_row_[x] = normL2(dx_row[x]);
                dy_row_[x] = normL2(dy_row[x]);
            }
        }
    }
    PairwiseSeamFinder::find(src, corners, masks);
}
void GraphCutSeamFinder::Impl::setGraphWeightsColor(const Mat &img1, const Mat &img2,
                                                    const Mat &mask1, const Mat
&mask2, GCGraph<float> &graph)
{
    const Size img_size = img1.size();

    // Set terminal weights
    for (int y = 0; y < img_size.height; ++y)
    {
        for (int x = 0; x < img_size.width; ++x)
        {
            int v = graph.addVtx();
            graph.addTermWeights(v, mask1.at<uchar>(y, x) ? terminal_cost_ : 0.f,
                                    mask2.at<uchar>(y, x) ? terminal_cost_ : 0.f);
        }
```

```
    }

    // Set regular edge weights
    const float weight_eps = 1.f;
    for (int y = 0; y < img_size.height; ++y)
    {
        for (int x = 0; x < img_size.width; ++x)
        {
            int v = y * img_size.width + x;
            if (x < img_size.width - 1)
            {
                float weight = normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
                               normL2(img1.at<Point3f>(y, x + 1), img2.at<Point3f>(y, x + 1)) +
                               weight_eps;
                if (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y, x + 1) ||
                    !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y, x + 1))
                    weight += bad_region_penalty_;
                graph.addEdges(v, v + 1, weight, weight);
            }
            if (y < img_size.height - 1)
            {
                float weight = normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
                               normL2(img1.at<Point3f>(y + 1, x), img2.at<Point3f>(y + 1, x))
+
                               weight_eps;
                if (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y + 1, x) ||
                    !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y + 1, x))
                    weight += bad_region_penalty_;
                graph.addEdges(v, v + img_size.width, weight, weight);
            }
        }
    }
}
void GraphCutSeamFinder::Impl::setGraphWeightsColorGrad(
        const Mat &img1, const Mat &img2, const Mat &dx1, const Mat &dx2,
        const Mat &dy1, const Mat &dy2, const Mat &mask1, const Mat &mask2,
        GCGraph<float> &graph)
{
    const Size img_size = img1.size();

    // Set terminal weights
```

```
    for (int y = 0; y < img_size.height; ++y)
    {
        for (int x = 0; x < img_size.width; ++x)
        {
            int v = graph.addVtx();
            graph.addTermWeights(v, mask1.at<uchar>(y, x) ? terminal_cost_ : 0.f,
                                    mask2.at<uchar>(y, x) ? terminal_cost_ : 0.f);
        }
    }

    // Set regular edge weights
    const float weight_eps = 1.f;
    for (int y = 0; y < img_size.height; ++y)
    {
        for (int x = 0; x < img_size.width; ++x)
        {
            int v = y * img_size.width + x;
            if (x < img_size.width - 1)
            {
                float grad = dx1.at<float>(y, x) + dx1.at<float>(y, x + 1) +
                             dx2.at<float>(y, x) + dx2.at<float>(y, x + 1) + weight_eps;
                float weight = (normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
                                normL2(img1.at<Point3f>(y, x + 1), img2.at<Point3f>(y, x + 1)))
/ grad +
                               weight_eps;
                if (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y, x + 1) ||
                    !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y, x + 1))
                    weight += bad_region_penalty_;
                graph.addEdges(v, v + 1, weight, weight);
            }
            if (y < img_size.height - 1)
            {
                float grad = dy1.at<float>(y, x) + dy1.at<float>(y + 1, x) +
                             dy2.at<float>(y, x) + dy2.at<float>(y + 1, x) + weight_eps;
                float weight = (normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
                                normL2(img1.at<Point3f>(y + 1, x), img2.at<Point3f>(y + 1, x)))
/ grad +
                               weight_eps;
                if (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y + 1, x) ||
                    !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y + 1, x))
                    weight += bad_region_penalty_;
```

```
                graph.addEdges(v, v + img_size.width, weight, weight);
            }
        }
    }
}
void GraphCutSeamFinder::Impl::findInPair(size_t first, size_t second, Rect roi)
{
    Mat img1 = images_[first].getMat(ACCESS_READ), img2 = images_[second].getMat(ACCESS_READ);
    Mat dx1 = dx_[first], dx2 = dx_[second];
    Mat dy1 = dy_[first], dy2 = dy_[second];
    Mat mask1 = masks_[first].getMat(ACCESS_RW), mask2 = masks_[second].getMat(ACCESS_RW);
    Point tl1 = corners_[first], tl2 = corners_[second];

    const int gap = 10;
    Mat subimg1(roi.height + 2 * gap, roi.width + 2 * gap, CV_32FC3);
    Mat subimg2(roi.height + 2 * gap, roi.width + 2 * gap, CV_32FC3);
    Mat submask1(roi.height + 2 * gap, roi.width + 2 * gap, CV_8U);
    Mat submask2(roi.height + 2 * gap, roi.width + 2 * gap, CV_8U);
    Mat subdx1(roi.height + 2 * gap, roi.width + 2 * gap, CV_32F);
    Mat subdy1(roi.height + 2 * gap, roi.width + 2 * gap, CV_32F);
    Mat subdx2(roi.height + 2 * gap, roi.width + 2 * gap, CV_32F);
    Mat subdy2(roi.height + 2 * gap, roi.width + 2 * gap, CV_32F);

    // Cut subimages and submasks with some gap
    for (int y = -gap; y < roi.height + gap; ++y)
    {
        for (int x = -gap; x < roi.width + gap; ++x)
        {
            int y1 = roi.y - tl1.y + y;
            int x1 = roi.x - tl1.x + x;
            if (y1 >= 0 && x1 >= 0 && y1 < img1.rows && x1 < img1.cols)
            {
                subimg1.at<Point3f>(y + gap, x + gap) = img1.at<Point3f>(y1, x1);
                submask1.at<uchar>(y + gap, x + gap) = mask1.at<uchar>(y1, x1);
                subdx1.at<float>(y + gap, x + gap) = dx1.at<float>(y1, x1);
                subdy1.at<float>(y + gap, x + gap) = dy1.at<float>(y1, x1);
            }
            else
            {
                subimg1.at<Point3f>(y + gap, x + gap) = Point3f(0, 0, 0);
```

```
                submask1.at<uchar>(y + gap, x + gap) = 0;
                subdx1.at<float>(y + gap, x + gap) = 0.f;
                subdy1.at<float>(y + gap, x + gap) = 0.f;
            }

            int y2 = roi.y - tl2.y + y;
            int x2 = roi.x - tl2.x + x;
            if (y2 >= 0 && x2 >= 0 && y2 < img2.rows && x2 < img2.cols)
            {
                subimg2.at<Point3f>(y + gap, x + gap) = img2.at<Point3f>(y2, x2);
                submask2.at<uchar>(y + gap, x + gap) = mask2.at<uchar>(y2, x2);
                subdx2.at<float>(y + gap, x + gap) = dx2.at<float>(y2, x2);
                subdy2.at<float>(y + gap, x + gap) = dy2.at<float>(y2, x2);
            }
            else
            {
                subimg2.at<Point3f>(y + gap, x + gap) = Point3f(0, 0, 0);
                submask2.at<uchar>(y + gap, x + gap) = 0;
                subdx2.at<float>(y + gap, x + gap) = 0.f;
                subdy2.at<float>(y + gap, x + gap) = 0.f;
            }
        }
    }

    const int vertex_count = (roi.height + 2 * gap) * (roi.width + 2 * gap);
    const int edge_count = (roi.height - 1 + 2 * gap) * (roi.width + 2 * gap) +
                           (roi.width - 1 + 2 * gap) * (roi.height + 2 * gap);
    GCGraph<float> graph(vertex_count, edge_count);

    switch (cost_type_)
    {
    case GraphCutSeamFinder::COST_COLOR:
        setGraphWeightsColor(subimg1, subimg2, submask1, submask2, graph);
        break;
    case GraphCutSeamFinder::COST_COLOR_GRAD:
        setGraphWeightsColorGrad(subimg1, subimg2, subdx1, subdx2, subdy1, subdy2,
                                 submask1, submask2, graph);
        break;
    default:
        CV_Error(Error::StsBadArg, "unsupported pixel similarity measure");
    }
```

```
        graph.maxFlow();

        for (int y = 0; y < roi.height; ++y)
        {
            for (int x = 0; x < roi.width; ++x)
            {
                if (graph.inSourceSegment((y + gap) * (roi.width + 2 * gap) + x + gap))
                {
                    if (mask1.at<uchar>(roi.y - tl1.y + y, roi.x - tl1.x + x))
                        mask2.at<uchar>(roi.y - tl2.y + y, roi.x - tl2.x + x) = 0;
                }
                else
                {
                    if (mask2.at<uchar>(roi.y - tl2.y + y, roi.x - tl2.x + x))
                        mask1.at<uchar>(roi.y - tl1.y + y, roi.x - tl1.x + x) = 0;
                }
            }
        }
    }
    GraphCutSeamFinder::GraphCutSeamFinder(String cost_type, float terminal_cost, float
    bad_region_penalty)
    {
        CostType t;
        if (cost_type == "COST_COLOR")
            t = COST_COLOR;
        else if (cost_type == "COST_COLOR_GRAD")
            t = COST_COLOR_GRAD;
        else
            CV_Error(Error::StsBadFunc, "Unknown cost type function");
        impl_ = new Impl(t, terminal_cost, bad_region_penalty);
    }
    GraphCutSeamFinder::GraphCutSeamFinder(int cost_type, float terminal_cost, float
    bad_region_penalty)
        : impl_(new Impl(cost_type, terminal_cost, bad_region_penalty)) {}
    void GraphCutSeamFinder::find(const std::vector<UMat> &src, const std::vector<Point> &corners,
                                  std::vector<UMat> &masks)
    {
        impl_->find(src, corners, masks);
    }
```