Лекция

boost::geometry

what is Boost?

full name: Boost C++ Libraries

http://boost.org/

"The Boost C++ Libraries are a collection of free libraries that extend the functionality of C++"

»Wikipedia

Содержание

- overview
- design ideas
- features
- examples
- design rationale

Boost.Geometry

- **решает задачи вычислительной геометрии**
  - drawing, mapping, visualizations
  - matrix / vector calculations
- обобщённое программирование
  - абстрактные алгоритмы
  - никаких предположений о типах
- **C++ header-only** библиотека
  - Кроссплатформенность
  - Компактность
  - Эффективность
- Свободная лицензия
- Следует стандартам
  - **std::**
  - **boost::**
  - ISO / OGC

Boost.Geometry История

- 1995 - Geodan Geographic Library
- 2008 - 1st preview for Boost as Geometry Library
- 2009 - 4th preview for Boost as Generic Geometry Library (GGL)
- November 2009 - final review and acceptance to Boost collection as boost::geometry
- June 2011: Boost 1.47, first version including boost::geometry
- June 2012: Boost 1.50, many bugfixes

Boost.Review()

- *Review process:* November 5- 23, 2009
  Review manager: Hartmut Kaiser (Boost.Spirit)
  14 reviewers
- *Final report* on November 28, 2009:
  12 votes **Yes**,2 votes **No**
- *Conclusion:*
  «The design is very clear. I think it can serve as a standard example of how to cover a big non trivial problem domain using meta-programming, partial specialization and tag dispatch to make it uniformly accessible by a set of generic algorithms»

Boost.Geometry Команда

- Barend Gehrels at Geodan - lead developer and project manager
- Bruno Lalande - lead developer
- Mateusz Loskot - supporting developer
- Adam Wulkiewicz - Spatial_Index contributor

- GGL mailing list (http://lists.osgeo.org/mailman/listinfo/ggl) ~130 users
  - Boost mailing lists (http://lists.boost.org)
  - Stack Overflow
  - Tickets
- большое коммьюнити с несколькими деятками людей, обсуждающие идеи для boost::geometry

Boost.Geometry Сложности

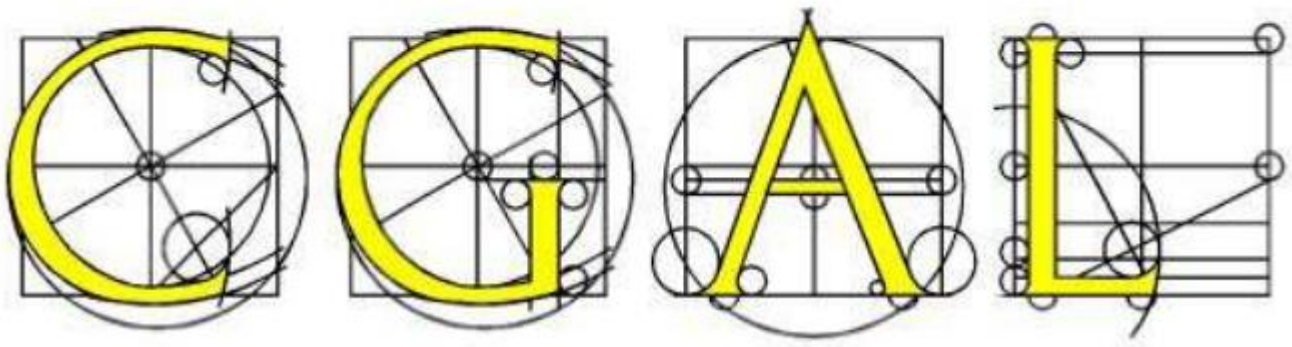
- Библиотека спроектирована и реализует:
  - generic
  - fast
  - robust
  - not specific to any domain
  - extensible
- *programming tool satisfying many with usable* “explosion of capabilities”
- низко-уровневая библиотека, настраиваемая стратегиями
  - max контроля,
  - max информации
  - max эффективности

Boost.Geometry Альтернативы

- **GEOS** (as in PostGreSQL / PostGIS)
- **CGAL**
- Microsoft .NET **Spatial** (as in SQL Server)
- **Boost.Polygon**
- **GPC**
- **Clipper**

"Make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications"

CGAL Project Proposal, 1996



Зачем нужны шаблоны?

конвертируем представления данных приложения в представления библиотеки

```
1.  struct GeoPoint {
2.      double latitude;
3.      double longitude;
4.  };

5.  struct Polyline {
6.      std::vector<GeoPoint> points;
7.      Polyline() = default;
8.      Polyline(const std::vector<GeoPoint> &points) : points(points) {}
9.  };

10. const geos::Coordinate pointToGeosCoordinate(const GeoPoint& point) {
11.     return geos::Coordinate(point.longitude, point.latitude);
12. }

13. geos::LineString* polylineToGeosLineString(const Polyline &polyline) {
14.     geos::CoordinateSequence s;
15.     for (const auto &p : polyline.points)
16.         s.add(pointToGeosCoordinate(p));
17.     return geos::geom::GeometryFactory::getDefaultInstance()->createLineString(&s);
18. }
```

Первый пример

длина отрезка и расстояние между точкой и отрезком

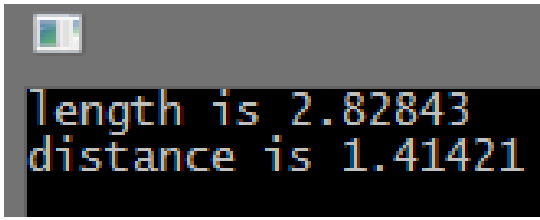
```
1.  #include <iostream>
2.  #include <boost/geometry.hpp>
```

```
3.  struct the_point_type { float x, y; };
4.  using Line = std::vector<the_point_type>;
```

```
1.  int main() {
2.      Line line;
3.      line.push_back(boost::geometry::make<the_point_type>(1, 2));
4.      line.push_back(boost::geometry::make<the_point_type>(3, 4));
5.      std::cout << "length is " << boost::geometry::length(line) << std::endl;
6.      int p[] = { 2, 5 };
7.      std::cout << "distance is " << boost::geometry::distance(line, p) << std::endl;
8.      return 0;
9.  }
```

Первый пример

длина отрезка и расстояние между точкой и отрезком




```
1.  #include <iostream>
2.  #include <boost/geometry.hpp>
3.  #include <boost/geometry/geometries/register/point.hpp>
4.  #include <boost/geometry/geometries/register/linestring.hpp>
5.  #include <boost/geometry/geometries/adapted/c_array.hpp>

6.  struct the_point_type { float x, y; };
7.  using Line = std::vector<the_point_type>;
8.  BOOST_GEOMETRY_REGISTER_POINT_2D(the_point_type, float, cs::cartesian, x, y)
9.  BOOST_GEOMETRY_REGISTER_C_ARRAY_CS(cs::cartesian)
10. BOOST_GEOMETRY_REGISTER_LINESTRING(Line)

11. int main() {
12.     Line line;
13.     line.push_back(boost::geometry::make<the_point_type>(1, 2));
14.     line.push_back(boost::geometry::make<the_point_type>(3, 4));
15.     std::cout << "length is " << boost::geometry::length(line) << std::endl;
16.     int p[] = { 2, 5 };
17.     std::cout << "distance is " << boost::geometry::distance(line, p) << std::endl;
18.     return 0;
19. }
```

Обобщённое программирование

шаблон (*template*)

+

конкретизация (*instantiation*)

+

компилятор (*compiler*)

=

итоговый исходный код
конкретной программы

Приёмы мета-программирования

- ***templates*** – *шаблоны* – обобщённая форма типов в языке программирования
- ***metafunctions*** – *мета-функции* – порождается для конкретного типа во время компиляции, техники выбора типа, скрывает (инкапсулирует) алгоритм вычислений
- ***traits*** – *характерные особенности* – дополнительная ассоциированная информация
- ***tag dispatching*** – *управление тэгами* – использует *traits* для различения зависящих от типов данных вызовов
- ***concepts*** – концепции – не (non-intrusive design_ - «создайте» свою собственную библиотеку типов и алгоритмов
- шаблон *стратегии* времени компиляции

Стратегии

шаблон (*template*)

+

особенности типов (*traits*)

+

конкретизация (*instantiation*)

=

выбор алгоритма

Концепции и модели

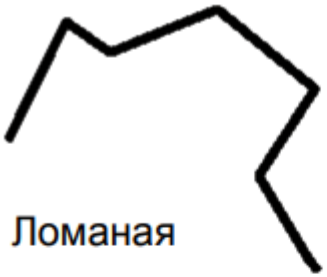
“**Концепция** – это набор требований из истинных (*valid*) выражений, связанных типов, инвариантов и обязательств по сложности выполнения”

“Тип, удовлетворяющий требованиям концепции называется **моделью**”

David Abrahams, Jeremy Siek

Геометрические примитивы
Basic concepts

Точка



Ломаная

Прямоугольник

Полигон

Полигон с дырками

Геометрические примитивы
Basic concepts

Основные строительные блоки
Basic concepts

- Point<>
- Segment<>
- LineString<>
- Ring<>
- Polygon<>
- Box<>

- MultiPoint<>
- MultiLineString<>
- MultiPolygon<>

- Variant

Абстрактные алгоритмы

- Пересечение двух примитивов
- Расстояние между примитивами
- Взаимное их расположение
- Площадь (объём)

Независимость алгоритмов

- `point<int, 1, cartesian>`
- `point<double, 2, spherical<radian>`

- **от размерности** пространства
- **от представления координат**
  - bg::cs::cartesian
  - bg::cs::geographic
  - bg::cs::polar
  - bg::cs::spherical
  - user-defined
- **от числового типа** и **точности** координатной системы (предпочитает боле точный тип)
  - *int* + *int* → *int*
  - *int* + *float* → *float*
  - *int* + *GMP* → *GMP*
  - *GMP* + *double* → *GMP*
- **Boost.Math.GMP** – is a high-precision floating point lib

Простой пример

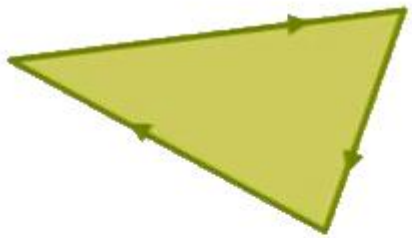
Hello World!

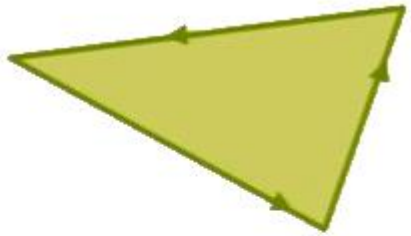
```
1.  #include <boost/geometry.hpp>
2.  #include <boost/geometry/geometries/geometries.hpp>
3.  #include <iostream>
4.  namespace bg = boost::geometry;
5.  int main()
6.  {
7.      using point = bg::model::point<double, 2, bg::cs::geographic<bg::degree>>;
8.      // Lodz > Brussels
9.      std::cout << bg::distance(point{ 19.454722, 51.776667 },
10.                               point{ 4.3500000, 50.833333 });
11.     return 0;
12. }
```

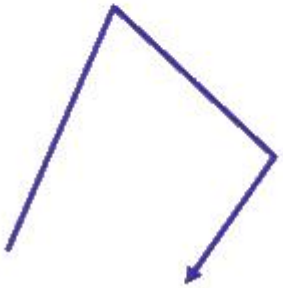
1056641.830203

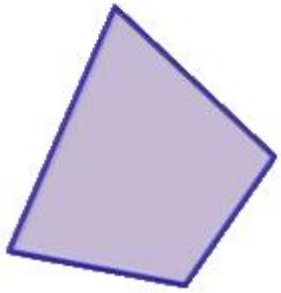
maps.google.com

Ориентация и замкнутость



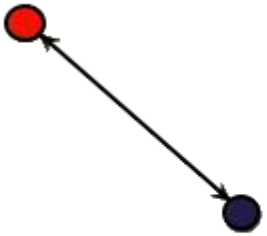


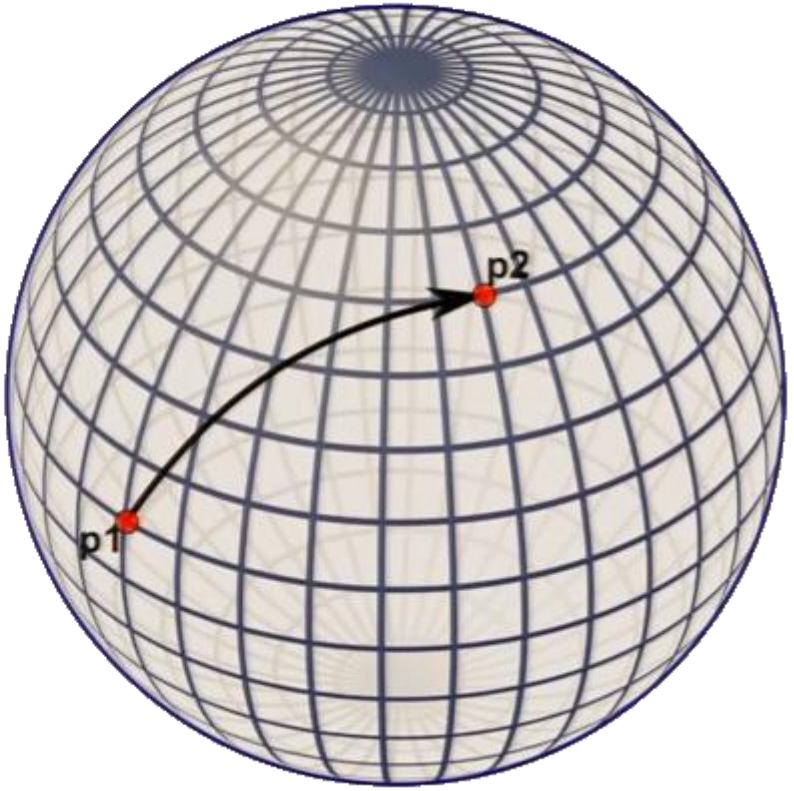




Обобщённые алгоритмы

- Generic distance

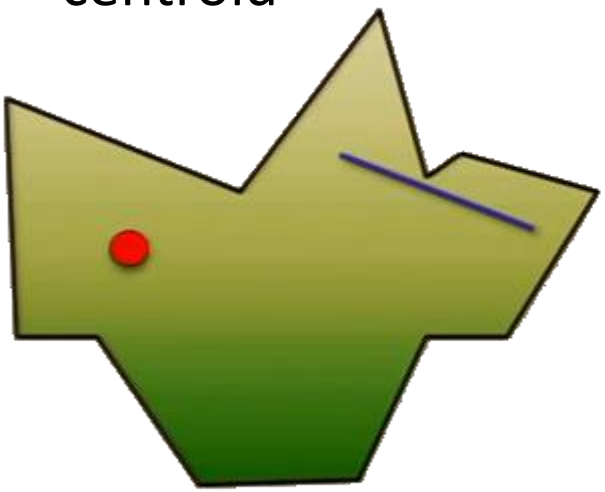


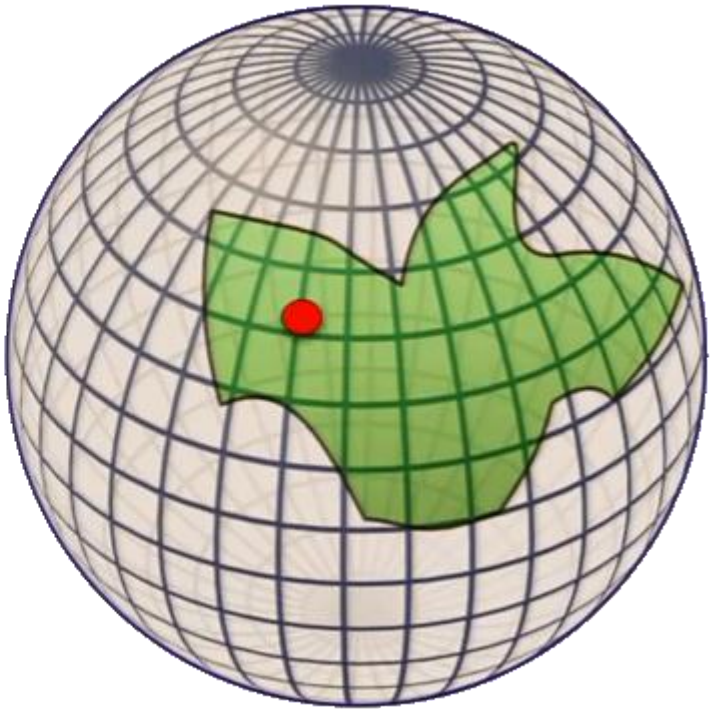


- Compare distances without SQRT

Обобщённые алгоритмы

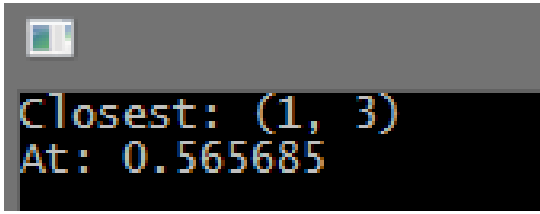
- Generic **within** algorithm
  - point_on_surface
  - point_on_border
  - centroid





Другой пример

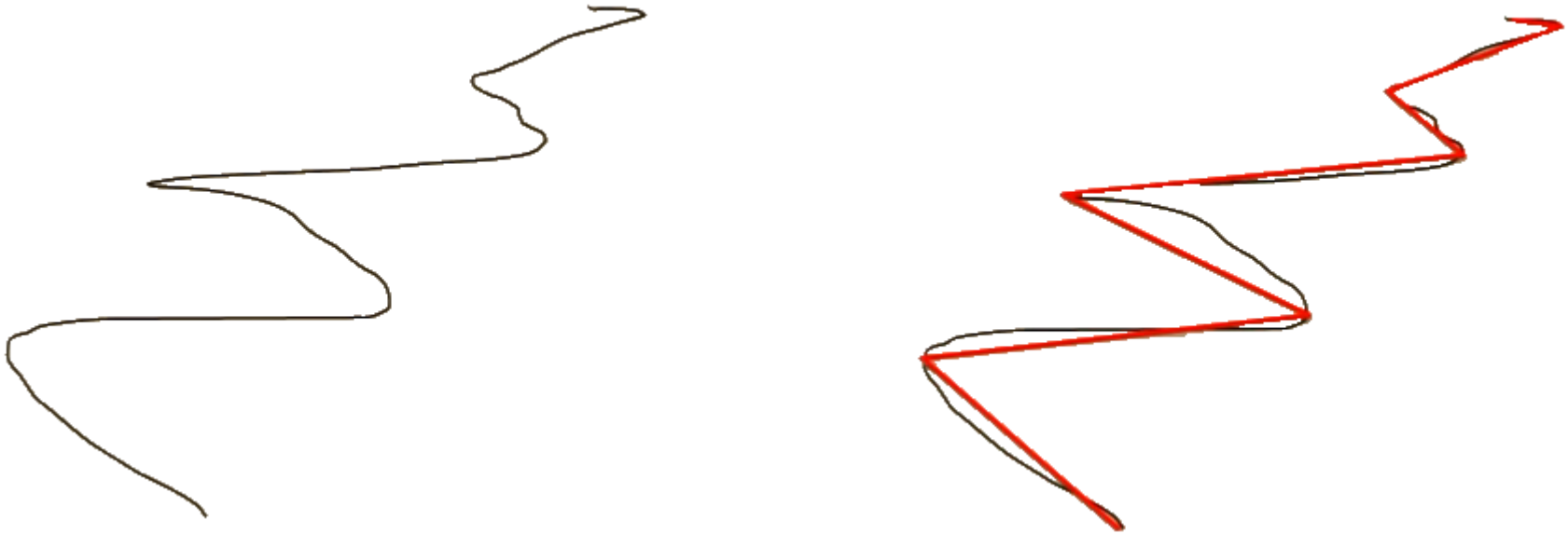
text format




```
1.  #include <boost/geometry.hpp>
2.  #include <boost/geometry/geometries/point_xy.hpp>
3.  #include <type_traits>
4.  #include <iostream>
5.  int main() {
6.      using point_type = boost::geometry::model::d2::point_xy<double>;
7.      point_type p{ 1.4, 2.6 };

8.      std::vector<point_type> v;
9.      for (double x = 0.0; x <= 4.0; ++x)
10.         for (double y = 0.0; y <= 4.0; ++y)
11.             v.push_back(point_type{ x, y });

12.     point_type min_p;
13.     double min_d = std::numeric_limits<double>::max();
14.     for (const auto &pv : v) {
15.         double d = boost::geometry::comparable_distance(p, pv);
16.         if (d < min_d) { min_d = d; min_p = pv; }
17.     }
18.     std::cout << "Closest: " << boost::geometry::dsv(min_p) << std::endl
19.               << "At: " << boost::geometry::distance(p, min_p) << std::endl;
20.     return 0;
21. }
```

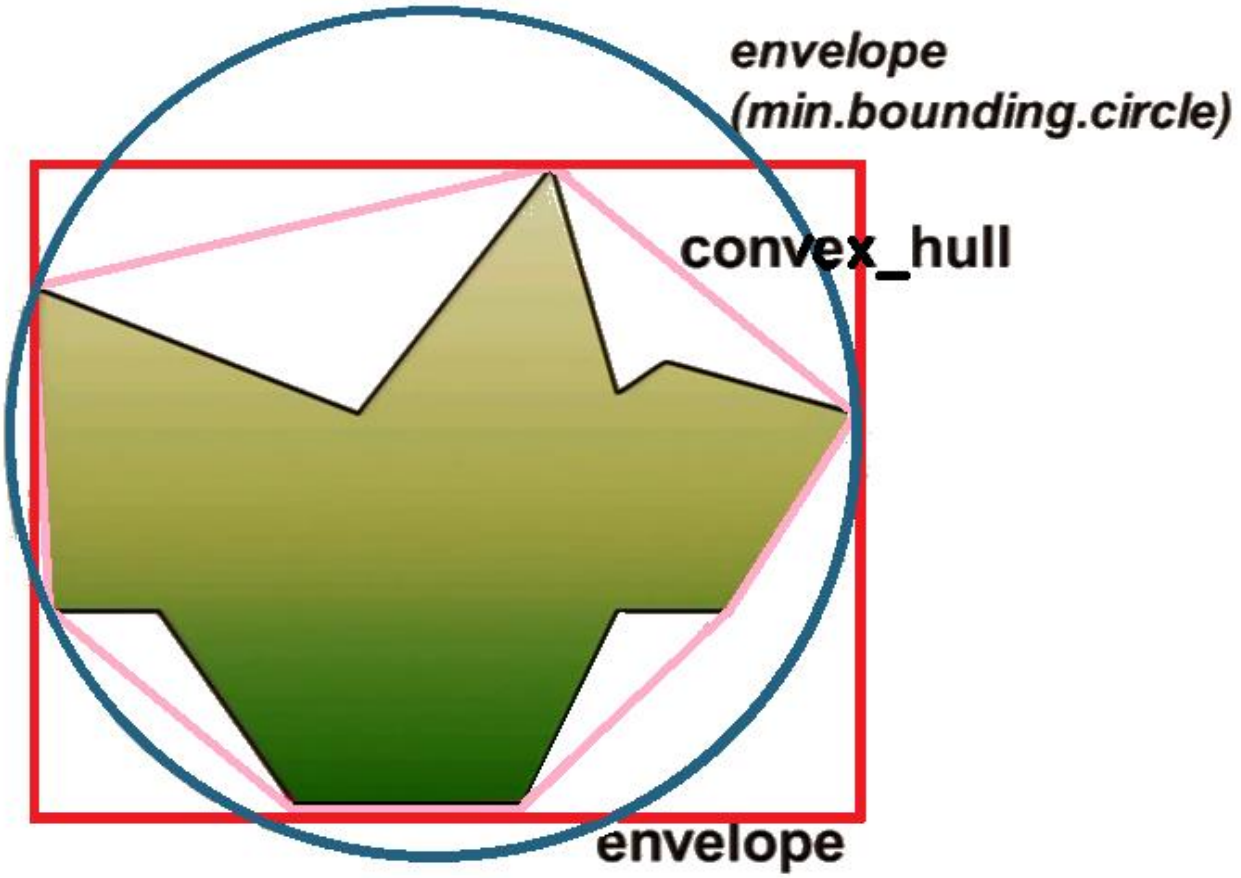
Обобщённые алгоритмы

- Generic **simplify** algorithm
  - Completely within (nothing on border)
  - Strategies to finetune calculation
  - Cartesian and Spherial



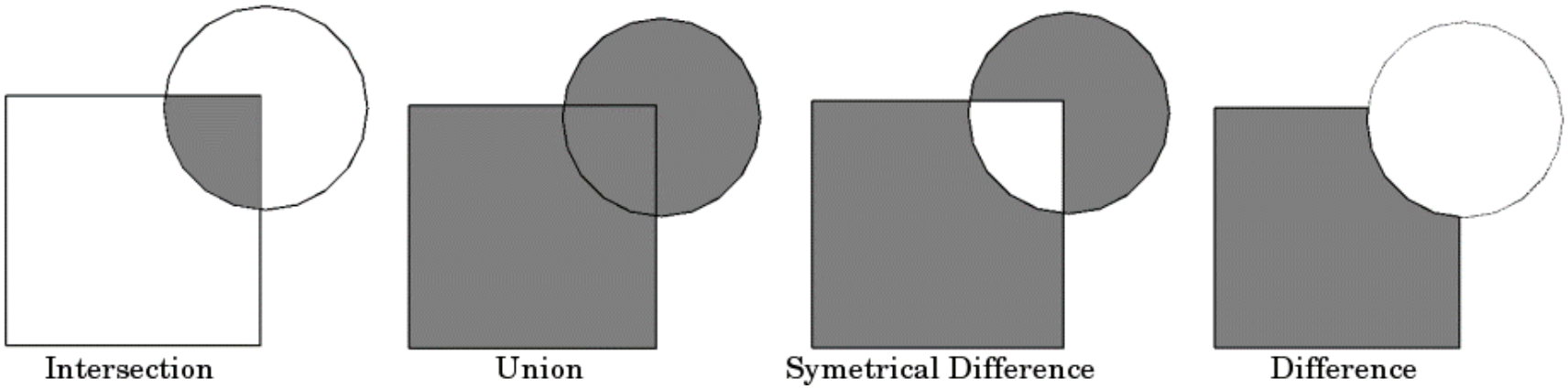
Обобщённые алгоритмы

- Generic **convex hull** algorithm

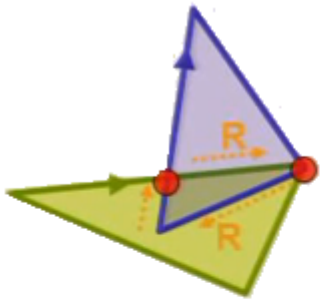


Обобщённые алгоритмы

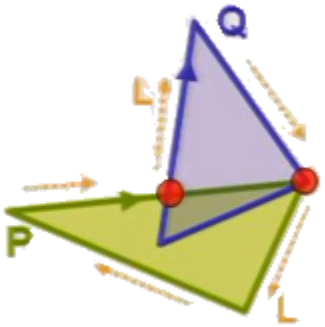
- Generic **overlay** algorithm
  - intersection
  - union
  - difference
  - Symmetric difference



Реализация перекрытия

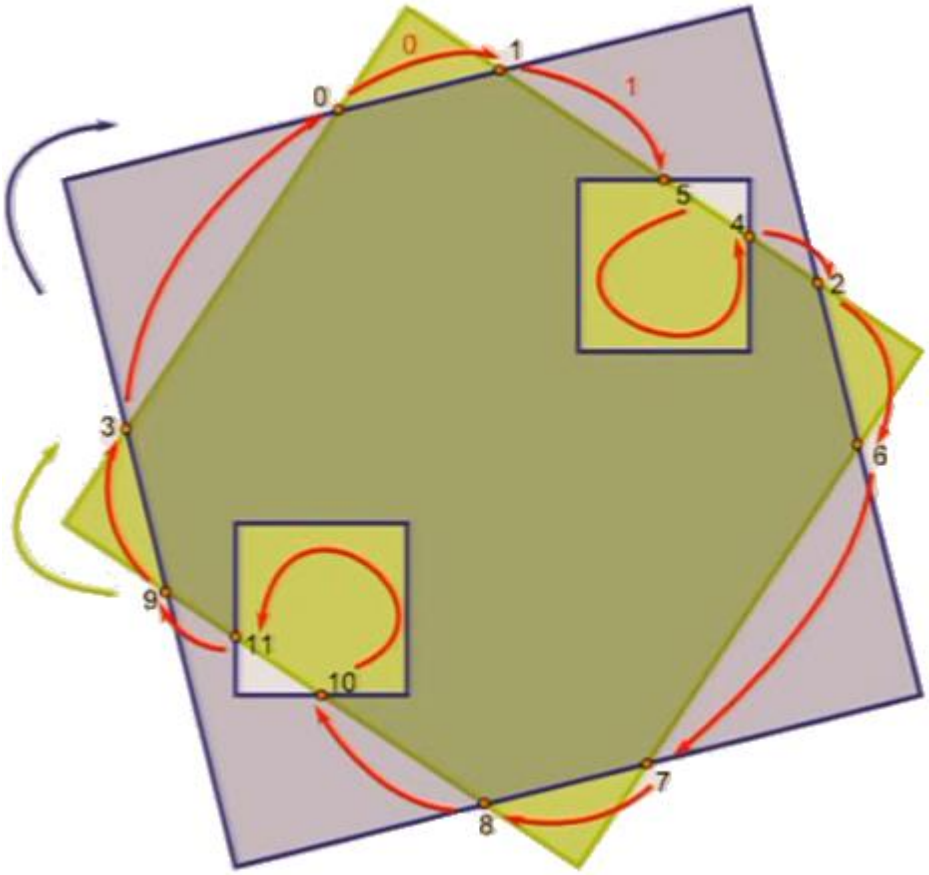


Intersection
Take the right turn everywhere

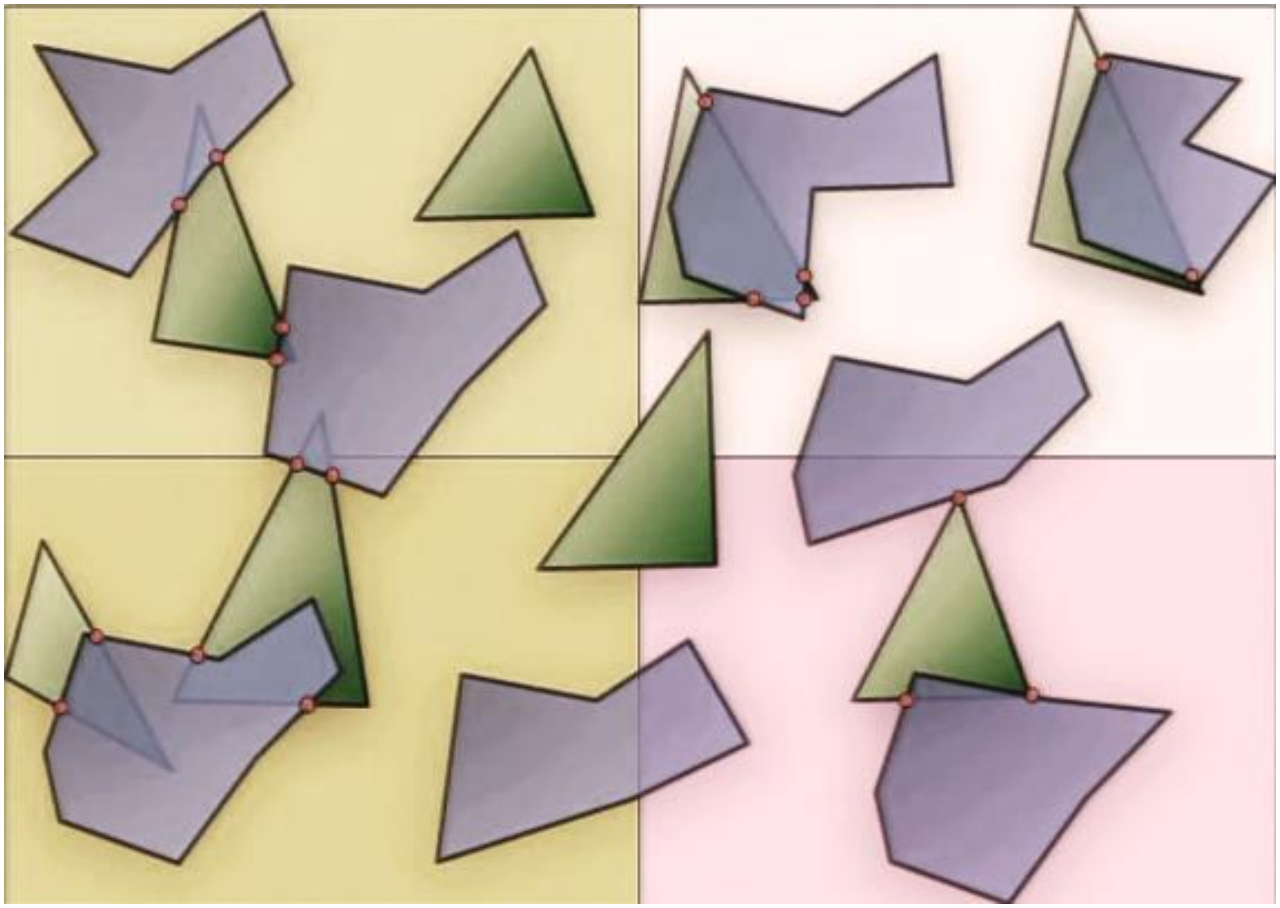


Union
Take the left turn everywhere

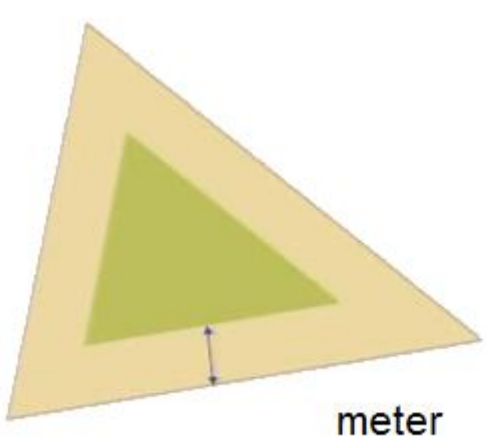
Реализация перекрытия



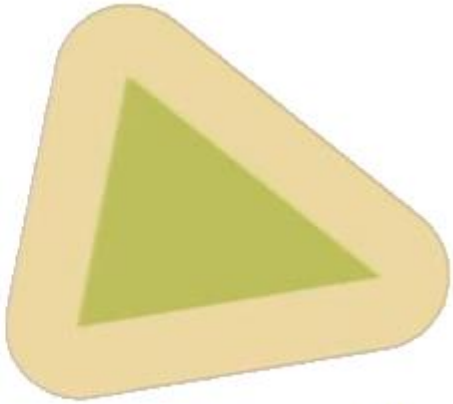
Оптимизация разделением вычислений на партиции



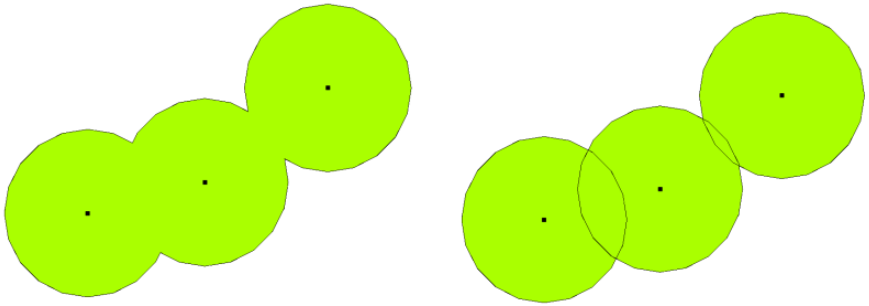
Буферная зона

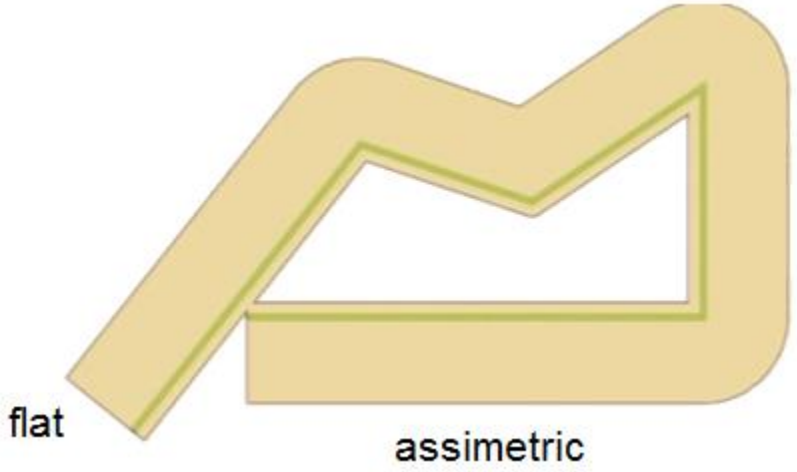


meter



round





flat

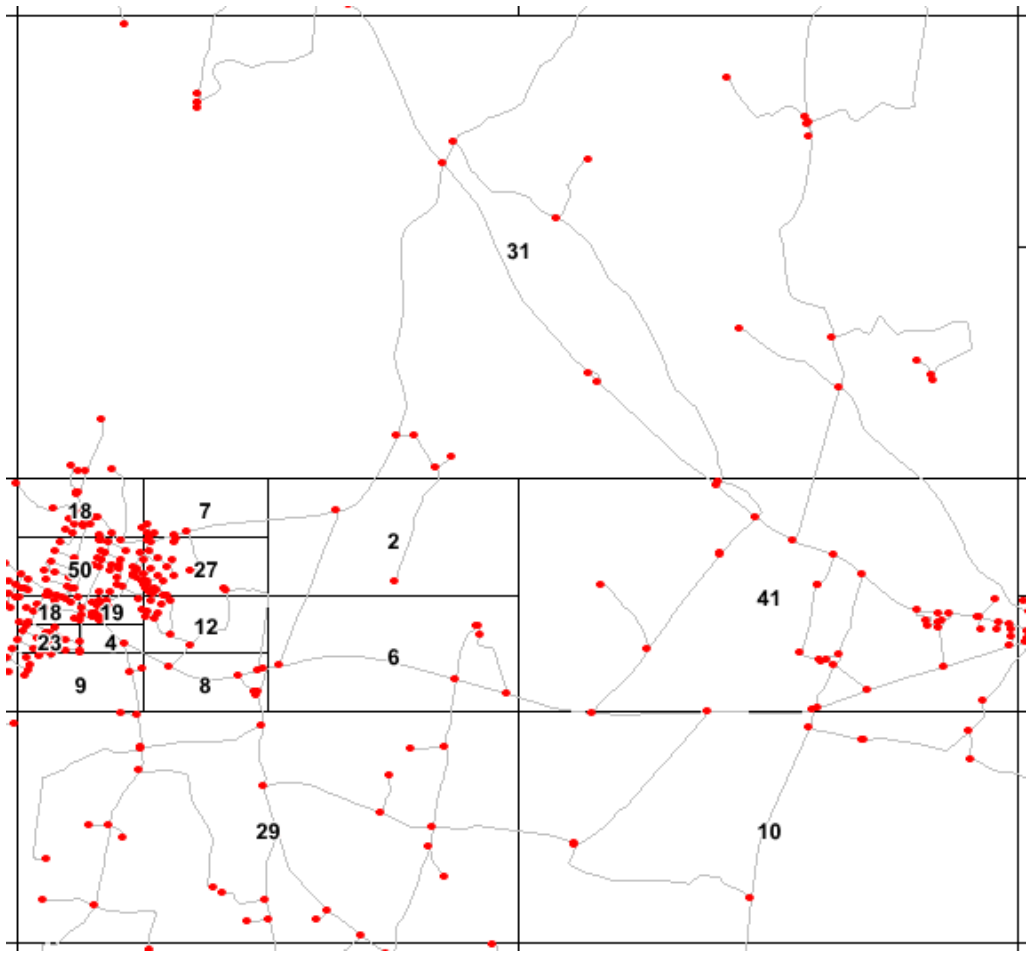
assimetric

Пространственный Поиск

rtree

- Started in GSoC
- Adam Wulkiewicz (revised)

- linear, quadratic or **r*-tree**
- bulk-loading
- user-defined value type
- various **spatial** and **knn** query
- stateful-allocator, move semantics, etc.



Построение интерфейсов
Building UI

Adaptors

- wxWidgets has “wxPoint”
- QT has “QPoint”

Dumping

- from strings (**WKT** – “Well-Known Text”, ISO/OGC)
- from file formats (**KML**/**SHP** etc) (not released)

Преобразование координат

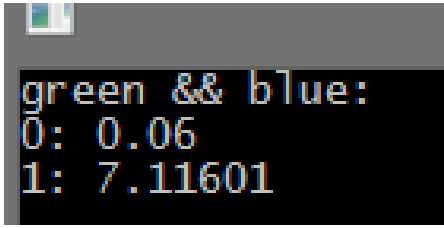
map_transformer strategy

```
1.  using map_transformer_type =
2.      boost::geometry::strategy::transform::map_transformer<wxPoint, 2, 2, true, true>;

3.  using inverse_transformer_type =
4.      boost::geometry::strategy::transform::inverse_transformer<wxPoint, 2, 2>;
```

- Spherical (degree) / Spherical (radian)
- Spherical / Cartesian (3D)
- Spherical (degree, with radius) / Spherical (radian, with radius)
- Spherical (with radius) / Cartesian (3D)

Полигоны

```
1.  #include <boost/geometry.hpp>
2.  #include <boost/geometry/geometries/linestring.hpp>
3.  #include <boost/geometry/geometries/point_xy.hpp>
4.  #include <boost/geometry/geometries/polygon.hpp>
5.  #include <iostream>
6.  #include <deque>
7.  int main() {
8.      namespace bm = boost::geometry::model;
9.      using polygon = bm::polygon<bm::d2::point_xy<double>>;
10.     try {
11.         polygon green, blue;
12.         boost::geometry::read_wkt(
13.             "POLYGON((2.0 1.3, 2.8 1.8, 9.0 0.8, 2.9 0.7, 2.0 1.3)"
14.                     "(4.0 2.0, 4.8 1.9, 4.4 2.2, 3.0 0.7, 4.0 2.0))", green);
15.         boost::geometry::read_wkt(
16.             "POLYGON((4.0 -0.5, 3.5 1.0, 4.0 -0.5, 9.0 0.8))", blue);
17.         std::deque<polygon> output;
18.         boost::geometry::intersection(green, blue, output);
19.         int i = 0;
20.         std::cout << "green && blue:" << std::endl;
21.         for (const polygon &p : output)
22.             std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
23.     }
24.     catch (const std::exception &e) { std::cout << e.what() << std::endl; }
25.     return 0;
26. }
```



Boost.Geometry
Производительность

- *“We are aware of the weaknesses of performance tests and that it is hard to design objective benchmarks”*
- *“There are so many differences in behaviour in all libraries under different circumstances, it appeared to be impossible or at least very difficult to compare libraries in one benchmark”*

http://trac.osgeo.org/ggl/wiki/Performance
http://svn.osgeo.org/osgeo/foss4g/benchmarking/geometry_libraries

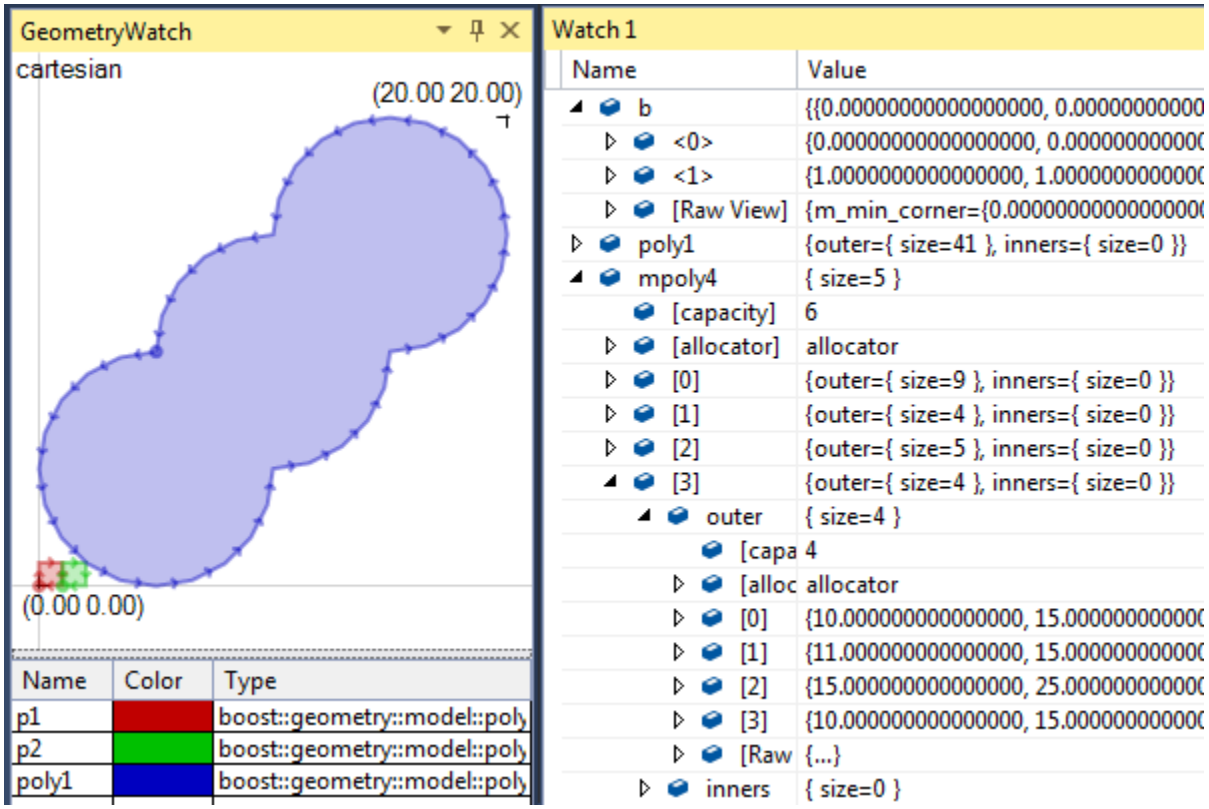
Производительность

- Microsoft MSVC preprocessor
  - _SECURE_SCL=0
  - _HAS_ITERATOR_DEBUGGING=0
- Use of **STLport** (popular open-source implementation), may result in significantly faster code than libs provided Microsoft (http://sourceforge.net/projects/stlport)

- Turn on compiler optimizations
- Compile in **release** mode

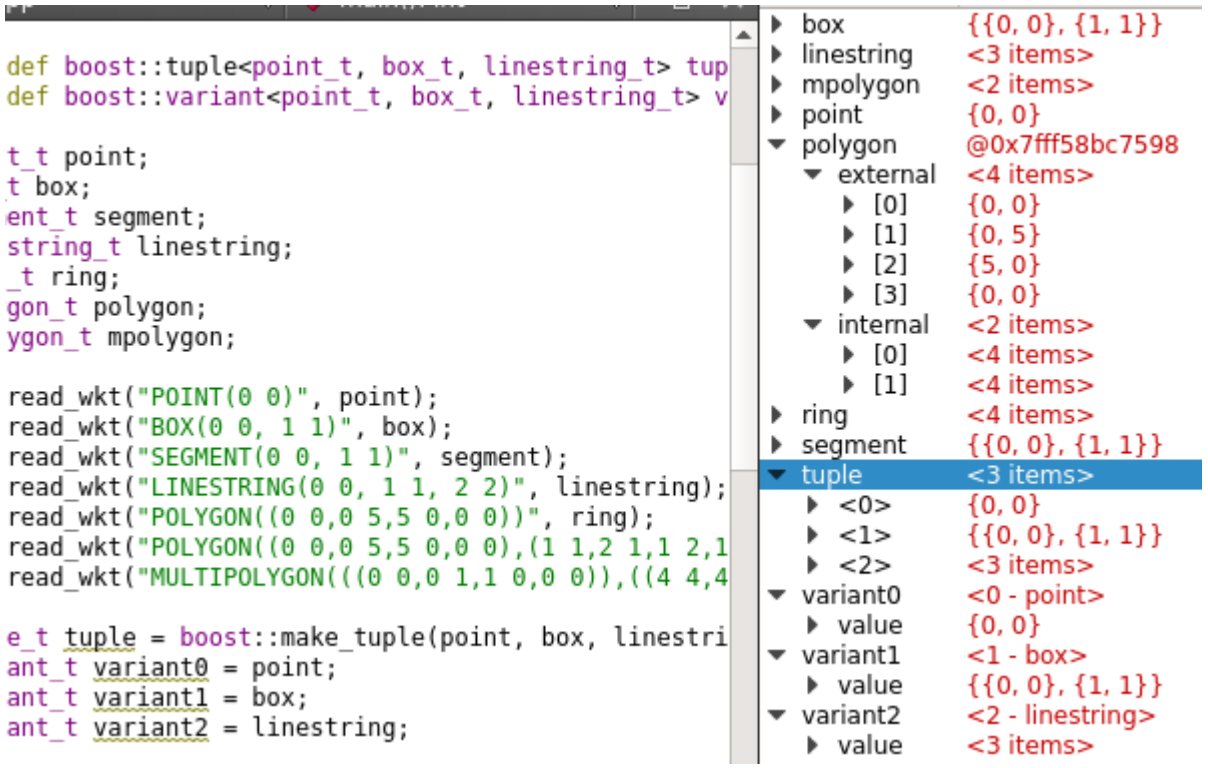
Graphical Debugging

- **MS VS** extension GraphicalDebugging
  - http://github.com/awulkiew/graphicaldebugging/



Graphical Debugging

- QtCreator Debugging Helpers
  - http://github.com/awulkiew/debugginghelpers



boost::geometry

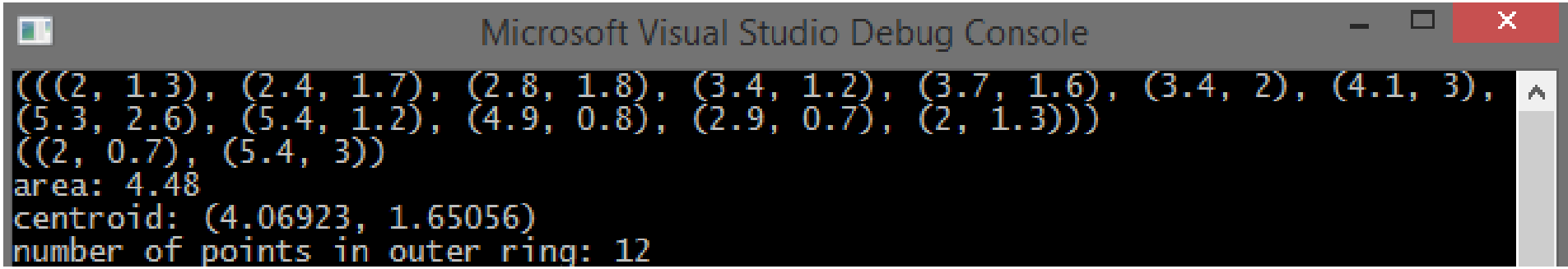
Polygon Example

```
1.  #include <algorithm> // for reverse, unique
2.  #include <iostream>
3.  #include <string>

4.  #include <boost/geometry/geometry.hpp>
5.  #include <boost/geometry/geometries/point_xy.hpp>
6.  #include <boost/geometry/geometries/polygon.hpp>
7.  #include <boost/geometry/geometries/adapted/c_array.hpp>
8.  #include <boost/geometry/multi/geometries/multi_polygon.hpp>

9.  BOOST_GEOMETRY_REGISTER_C_ARRAY_CS(cs::cartesian)

10. int main() {
11.     using namespace boost::geometry;
12.     using namespace std;

13.     typedef model::d2::point_xy<double> point_2d;
14.     typedef model::polygon<point_2d> polygon_2d;
15.     typedef model::box<point_2d> box_2d;
16.     // ...
17. }
```

boost::geometry

Polygon Example

```
10. int main() {
16.     // ...
17.     polygon_2d poly;
18.     {
19.         const double coor[][2] = {
20.             {2.0, 1.3}, {2.4, 1.7}, {2.8, 1.8}, {3.4, 1.2}, {3.7, 1.6},
21.             {3.4, 2.0}, {4.1, 3.0}, {5.3, 2.6}, {5.4, 1.2}, {4.9, 0.8}, {2.9, 0.7},
22.             {2.0, 1.3} }; // closing point is opening point
23.         assign_points(poly, coor);
24.     }
25.     correct(poly);
26.     box_2d b;
27.     envelope(poly, b);
28.     point_2d cent;
29.     centroid(poly, cent);

30.     cout << dsv(poly) << endl;
31.     cout << dsv(b) << endl;
32.     cout << "area: " << area(poly) << endl;
33.     cout << "centroid: " << dsv(cent) << endl;
34.     cout << "number of points in outer ring: " << poly.outer().size() << endl;
35.     // ...
36. }
```




```
10. int main() {
16.     // ...
17.     polygon_2d poly;
18.     {
19.         const double coor[][2] = {
20.             {2.0, 1.3}, {2.4, 1.7}, {2.8, 1.8}, {3.4, 1.2}, {3.7, 1.6},
21.             {3.4, 2.0}, {4.1, 3.0}, {5.3, 2.6}, {5.4, 1.2}, {4.9, 0.8}, {2.9, 0.7},
22.             {2.0, 1.3} }; // closing point is opening point
23.         assign_points(poly, coor);
24.     }
25.     correct(poly);
26.     box_2d b;
27.     envelope(poly, b);
28.     point_2d cent;
29.     centroid(poly, cent);

30.     cout << dsv(poly) << endl;
31.     cout << dsv(b) << endl;
32.     cout << "area: " << area(poly) << endl;
33.     cout << "centroid: " << dsv(cent) << endl;
34.     cout << "number of points in outer ring: " << poly.outer().size() << endl;
35.     // ...
36. }
```

boost::geometry

Polygon Example

```
10. int main() {
35.     // ...
36.     {
37.         poly.inners().resize(1);
38.         model::ring<point_2d>& inner = poly.inners().back();
39.         double coor[][2] = {{4., 2.}, {4.2, 1.4}, {4.8, 1.9}, {4.4, 2.2}, {4., 2.}};
40.         assign_points(inner, coor);
41.     }
42.     correct(poly);
43.     centroid(poly, cent);

44.     cout << "with inner ring:" << dsv(poly) << endl;
45.     cout << "new area of polygon: " << area(poly) << endl;
46.     cout << "new centroid: " << dsv(cent) << endl;

47.     // You can test whether points are within a polygon
48.     cout << "point in polygon:" << boolalpha
49.         << " p1: " << within(make<point_2d>(3.0, 2.0), poly)
50.         << " p2: " << within(make<point_2d>(3.7, 2.0), poly)
51.         << " p3: " << within(make<point_2d>(4.4, 2.0), poly) << endl;

52.     // ...
53. }
```

```
with inner ring:(((2, 1.3), (2.4, 1.7), (2.8, 1.8), (3.4, 1.2), (3.7, 1.6), (3.4
, 2), (4.1, 3), (5.3, 2.6), (5.4, 1.2), (4.9, 0.8), (2.9, 0.7), (2, 1.3)), ((4,
2), (4.2, 1.4), (4.8, 1.9), (4.4, 2.2), (4, 2)))
new area of polygon: 4.15
new centroid: (4.04663, 1.6349)
point in polygon: p1: false p2: true p3: false
```

Polygon Example

```
10. int main() {
35.     // ...
36.     {
37.         poly.inners().resize(1);
38.         model::ring<point_2d>& inner = poly.inners().back();
39.         double coor[][2] = {{4., 2.}, {4.2, 1.4}, {4.8, 1.9}, {4.4, 2.2}, {4., 2.}};
40.         assign_points(inner, coor);
41.     }
42.     correct(poly);
43.     centroid(poly, cent);

44.     cout << "with inner ring:" << dsv(poly) << endl;
45.     cout << "new area of polygon: " << area(poly) << endl;
46.     cout << "new centroid: " << dsv(cent) << endl;

47.     // You can test whether points are within a polygon
48.     cout << "point in polygon:" << boolalpha
49.         << " p1: " << within(make<point_2d>(3.0, 2.0), poly)
50.         << " p2: " << within(make<point_2d>(3.7, 2.0), poly)
51.         << " p3: " << within(make<point_2d>(4.4, 2.0), poly) << endl;

52.     // ...
53. }
```

boost::geometry

```
10. int main() {
53.     // ...
54.     box_2d cb(make<point_2d>(1.5, 1.5), make<point_2d>(4.5, 2.5));

55.     typedef vector<polygon_2d> polygon_list; polygon_list v;
56.     intersection(cb, poly, v);
57.     cout << "\nClipped output polygons" << endl;
58.     for (polygon_list::const_iterator it = v.begin(); it != v.end(); ++it)
59.         cout << dsv(*it) << endl;

60.     typedef model::multi_polygon<polygon_2d> polygon_set; polygon_set ps;
61.     union_(cb, poly, ps);
62.     polygon_2d hull;
63.     convex_hull(poly, hull);
64.     cout << "\nConvex hull:" << dsv(hull) << endl;

65.     typedef model::polygon<point_2d, true, true, deque, deque> deque_polygon;
66.     deque_polygon poly2;
67.     ring_type<deque_polygon>::type& ring = exterior_ring(poly2);

68.     append(ring, make<point_2d>(2.8, 1.9)); append(ring, make<point_2d>(2.9, 2.4));
69.     append(ring, make<point_2d>(3.3, 2.2)); append(ring, make<point_2d>(3.2, 1.8));
70.     append(ring, make<point_2d>(2.8, 1.9));
71.     cout << "\n" << dsv(poly2) << endl;
72.     return 0;
73. }
```

boost::geometry

```
10.  int main() {
53.      // ...
54.      box_2d cb(make<point_2d>(1.5, 1.5), make<point_2d>(4.5, 2.5));
```

```
Clipped output polygons
(((2.2, 1.5), (2.4, 1.7), (2.8, 1.8), (3.1, 1.5), (2.2, 1.5)))
(((3.625, 1.5), (3.7, 1.6), (3.4, 2), (3.75, 2.5), (4.5, 2.5), (4.5, 2.125), (4.
4, 2.2), (4, 2), (4.16667, 1.5), (3.625, 1.5)))
(((4.5, 1.65), (4.5, 1.5), (4.32, 1.5), (4.5, 1.65)))

Convex hull:(((2, 1.3), (2.4, 1.7), (4.1, 3), (5.3, 2.6), (5.4, 1.2), (4.9, 0.8)
, (2.9, 0.7), (2, 1.3)))

(((2.8, 1.9), (2.9, 2.4), (3.3, 2.2), (3.2, 1.8), (2.8, 1.9)))
```

```
63.      convex_hull(poly, hull);
64.      cout << "\nConvex hull:" << dsv(hull) << endl;

65.      typedef model::polygon<point_2d, true, true, deque, deque> deque_polygon;
66.      deque_polygon poly2;
67.      ring_type<deque_polygon>::type& ring = exterior_ring(poly2);

68.      append(ring, make<point_2d>(2.8, 1.9)); append(ring, make<point_2d>(2.9, 2.4));
69.      append(ring, make<point_2d>(3.3, 2.2)); append(ring, make<point_2d>(3.2, 1.8));
70.      append(ring, make<point_2d>(2.8, 1.9));
71.      cout << "\n" << dsv(poly2) << endl;
72.      return 0;
73.  }
```

boost::geometry

Polygon Example

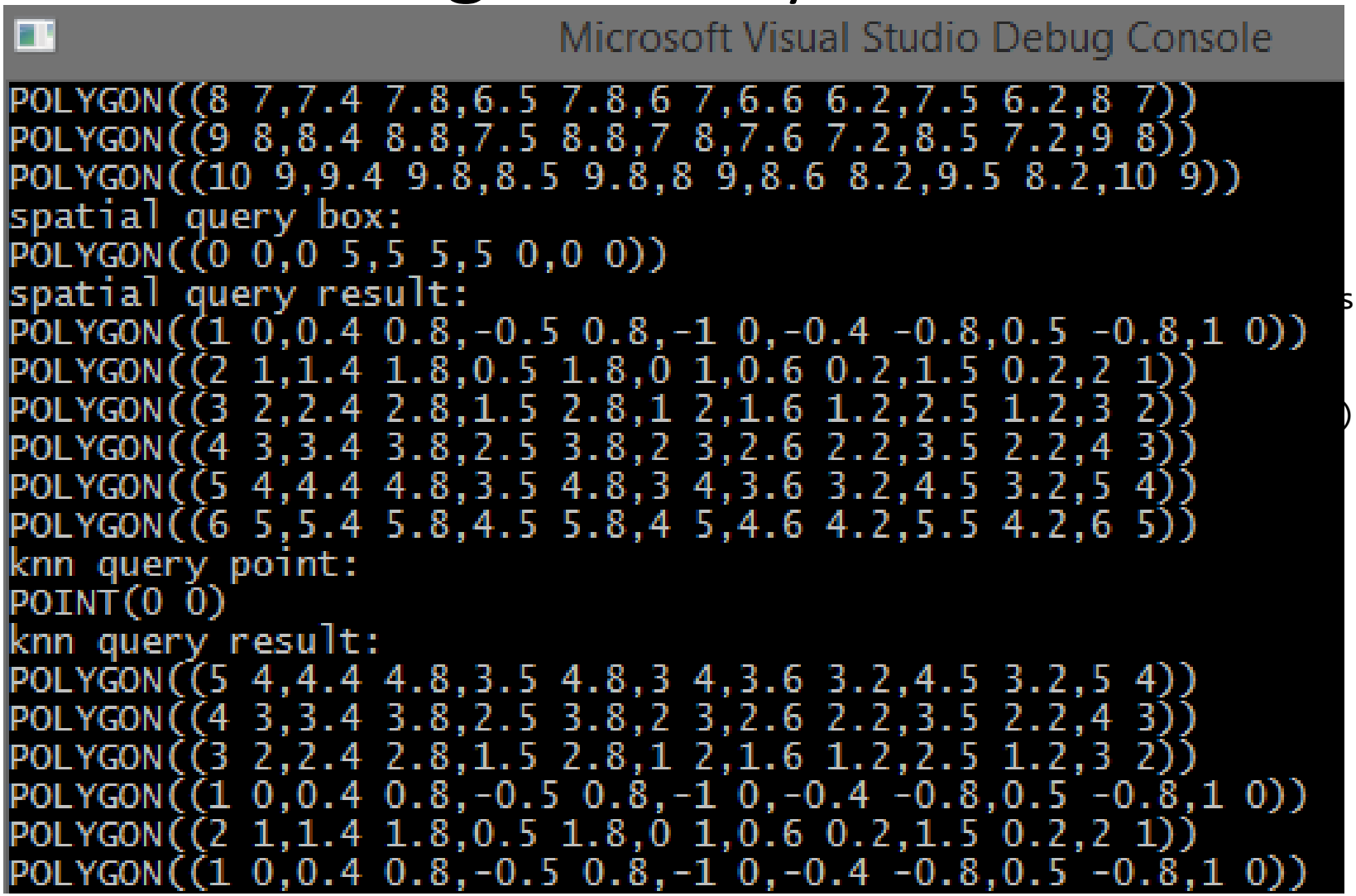
```
8.  #include <boost/geometry/index/rtree.hpp>
10. int main() {
11.     using namespace boost::geometry;
12.     using namespace boost::geometry::index;
13.     using my_point   = model::point<float, 2, cs::cartesian>;
14.     using my_box     = model::box<my_point> ;
15.     using my_polygon = model::polygon<my_point, false, false>; // ccw, open polygon
16.     using my_value   = std::pair<my_box, std::shared_ptr<my_polygon>>;

17.     rtree<my_value, linear<16, 4>> rtree{};
18.     std::cout << "filling index with polygons shared pointers:" << std::endl;
19.     for (unsigned i = 0; i < 10; ++i) {
20.         auto p = std::make_shared<my_polygon>();
21.         for (float a = 0; a < 6.28316f; a += 1.04720f) {
22.             float x = i + int(10 * std::cos(a))*0.1f;
23.             float y = i + int(10 * std::sin(a))*0.1f;
24.             p->outer().push_back(my_point(x, y));
25.         }
26.         std::cout << wkt<my_polygon>(*p) << std::endl;
27.         my_box b = return_envelope<my_box>(*p);
28.         rtree.insert(std::make_pair(b, p));
29.     }
30.     // ...
```

boost::geometry

Polygon Example

```
10. int main() {
22.     // ...
23.     my_box query_box(my_point(0, 0), my_point(5, 5));
24.     std::vector<my_value> result_s;
25.     rtree.query(index::intersects(query_box), std::back_inserter(result_s));

26.     std::vector<my_value> result_n;
27.     rtree.query(nearest(my_point(0, 0), 5), std::back_inserter(result_n));

28.     std::cout << "spatial query box:" << std::endl;
29.     std::cout << wkt<my_box>(query_box) << std::endl;
30.     std::cout << "spatial query result:" << std::endl;
31.     for (const auto& v : result_s)
32.         std::cout << wkt<my_polygon>(*v.second) << std::endl;

33.     std::cout << "knn query point:" << std::endl;
34.     std::cout << wkt<my_point>(my_point(0, 0)) << std::endl;
35.     std::cout << "knn query result:" << std::endl;
36.     for(const auto& v : result_n)
37.         std::cout << wkt<my_polygon>(*v.second) << std::endl;
38.     return 0;
39. }
```

boost::geometry




```
38.     return 0;
39. }
```

Пример из жизни

Умная точка

Point.h

```
1.   struct Point {
2.       using value_type = double;

3.       Point() : x_(0.), y_(0.) {}
4.       Point(Point::value_type x, Point::value_type y) : x_(x), y_(y) {}
5.       Point(const Point &p) = default;
6.       Point(Point&&) = default;
7.       const Point& operator=(const Point &) = default;
8.       const Point& operator=(Point&&) = default;

9.       void  rotate_radians(const Point &center, Point::value_type angle);
10.      void  rotate_degrees(const Point &center, Point::value_type angle)
11.      { rotate_radians(center, angle * value_type{M_PI} / value_type{180.}); }

12.      bool hit(const Point &p, Point::value_type eps) const;
13.      bool hit(const Point &p) const;

14.      Point::value_type angle(const Point &p) const;
15.      Point::value_type norm2() const { return sqrt(x * x + y * y); }

16.      Point orto() const { return Point{ -y,x }; }

16.      friend tostream& operator<<(tostream &s, const Point &p);
17.      friend tistream& operator>>(tistream &s, Point &p);
```

Умная точка

Point.h

```
16.     bool  operator<  (const Point &p) const;
17.     bool  operator>  (const Point &p) const;
18.     bool  operator<= (const Point &p) const;
19.     bool  operator>= (const Point &p) const;
20.     bool  operator!= (const Point &p) const;
21.     bool  operator== (const Point &p) const;

22.     Point operator- () const { return Point{ -x, -y }; }
23.     Point operator+ () const { return Point{ x, y }; }

24.     Point operator- (const Point &p) const;
25.     Point operator+ (const Point &p) const;
26.     Point operator* (const Point &p) const;
27.     Point operator/ (const Point &p) const;

28.     Point& operator+= (const Point &p);
29.     Point& operator-= (const Point &p);
30.     Point& operator*= (const Point &p);
31.     Point& operator/= (const Point &p);

32.     Point& operator+= (Point::value_type d);
33.     Point& operator-= (Point::value_type d);
34.     Point& operator*= (Point::value_type d);
35.     Point& operator/= (Point::value_type d);
```

Умная точка

Point.h

```
38.     /* Microsoft specific: C++ properties */
39.     __declspec(property(get = get_x, put = put_x)) Point::value_type x;
40.     Point::value_type get_x() const { return x_; }
41.     void put_x(Point::value_type x) { x_ = x; }

42.     __declspec(property(get = get_y, put = put_y)) Point::value_type y;
43.     Point::value_type get_y() const { return y_; }
44.     void put_y(Point::value_type y) { y_ = y; }

45.     operator std::pair<Point::value_type, Point::value_type>() const;
46.     operator boost::pair<Point::value_type, Point::value_type>() const;

47. private:
48.     Point::value_type x_, y_;
49. }; // end class Point
```

PointAdapter.h

для класса из Point.h

```
1.  #pragma once

2.  #include "Point.h"

3.  namespace boost {
4.  namespace geometry {
5.  namespace traits {

6.  template<> struct tag<Point> { typedef point_tag type; };

7.  template<> struct coordinate_type<Point> { typedef Point::value_type type; };

8.  template<> struct coordinate_system<Point> { typedef cs::cartesian type; };

9.  template<> struct dimension<Point> : boost::mpl::int_<2> {};

10. template<> struct access<Point, 0> {
11.     static Point::value_type get(const Point &p) { return p.get_x(); }
12.     static void set(Point& p, const Point::value_type& value) { p.put_x(value); }
13. };

14. template<> struct access<Point, 1> {
15.     static Point::value_type get(Point const& p) { return p.get_y(); }
16.     static void set(Point& p, Point::value_type const& value) { p.put_y(value); }
17. };

18. } // namespace traits
19. } // namespace geometry
20. } // namespace boost
```

Как это работает?

Как это работает?

есть какая-то точка, которую хотим использовать в нашей библиотеки геометрии

```
1.  #include <boost/geometry.hpp>
2.  #include <boost/mpl/int.hpp>
3.  #include <vector>
4.  #include <cmath>

5.  struct Point {
6.      double x, y;
7.  };

8.  struct Polyline {
9.      std::vector<Point> points;
10. };
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  inline
3.  auto distance(const Point& a, const Point& b) {
4.      double dx = a.x - b.x;
5.      double dy = a.y - b.y;
6.      return std::sqrt(dx * dx + dy * dy);
7.  }
8.  } // my_geometry
```

откажемся от типа и имён полей

Как это работает?

хотим не зависеть от типа и имён полей

```
1.  namespace my_geometry {
2.  namespace traits {

3.  template<typename P, int Dim> struct access {};

4.  template <> struct access<Point, 0> {
5.      static inline double get(const Point& p) { return p.x; }
6.      static inline void set(Point& p, const double& value) { p.x = value; }
7.  };

8.  template <> struct access<Point, 1> {
9.      static inline double get(const Point& p) { return p.y; }
10.     static inline void set(Point& p, const double& value) { p.y = value; }
11. };
12. } // traits

13. template <int Dim, typename P>
14. inline double get(const P& p) { return traits::access<P, Dim>::get(p); }

15. } // my_geometry
```

это уже похоже на концепцию

Как это работает?

Адаптация ломаной линии – Polyline

```
1.  #include <boost/range.hpp>
2.  namespace boost {
3.  template <> struct range_iterator<Polyline> {
4.      typedef std::vector<Point>::iterator type;
5.  };
6.  template <> struct range_const_iterator<Polyline> {
7.      typedef std::vector<Point>::const_iterator type;
8.  };
9.  } // end boost

10. inline std::vector<Point>::iterator range_begin(Polyline &polyline) {
11.     return polyline.points.begin();
12. }
13. inline std::vector<Point>::iterator range_end(Polyline &polyline) {
14.     return polyline.points.end();
15. }
16. inline std::vector<Point>::const_iterator range_begin(const Polyline& polyline) {
17.     return polyline.points.begin();
18. }
19. inline std::vector<Point>::const_iterator range_end(const Polyline& polyline) {
20.     return polyline.points.end();
21. }
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  inline
3.  auto distance(const Point& a, const Point& b) {
4.      double dx = a.x - b.x;
5.      double dy = a.y - b.y;
6.      return std::sqrt(dx * dx + dy * dy);
7.  }
8.  } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename P1, typename P2>
3.  auto distance(const P1& a, const P2& b) {
4.      double dx = get<0>(a) - get<0>(b);
5.      double dy = get<1>(a) - get<1>(b);
6.      return std::sqrt(dx * dx + dy * dy);
7.  }
8.  } // my_geometry
```

откажемся от размерности пространства и типа координат

Как это работает?

хотим не зависеть от размерности и типа координат

```
1.  namespace my_geometry {
2.  namespace traits {

3.  template <typename P> struct dimension {};
4.  template <typename P> struct coordinate_type {};
5.  template <typename P> struct coordinate_system {};

6.  template <> struct dimension<Point> : boost::mpl::int_<2> {};

7.  template <> struct coordinate_type<Point> { typedef double type; };

8.  template <> struct coordinate_system<Point> {
9.      typedef boost::geometry::cs::cartesian type;
10. }
11. } // traits

12. template <typename P> struct dimension : traits::dimension<P> {};
13. template <typename P> struct coordinate_type : traits::coordinate_type<P> {};
14. template <typename P> struct coordinate_system : traits::coordinate_system<P> {};

15. } // my_geometry
```

Как это работает?

хотим не зависеть от размерности и типа координат

```
1.  namespace my_geometry {

2.  template <typename P1, typename P2, int Dim>
3.  struct pythagoras {
4.      typedef typename select_most_precise<
5.          typename coordinate_type<P1>::type,
6.          typename coordinate_type<P2>::type>::type computation_type;

7.      static computation_type apply(const P1& a, const P2& b) {
8.          computation_type d = get<Dim - 1>(a) - get<Dim - 1>(b);
9.          return (d * d) + pythagoras<P1, P2, Dim - 1>::apply(a, b);
10.     }
11. };

12. template <typename P1, typename P2>
13. struct pythagoras<P1, P2, 0> {
14.     typedef typename select_most_precise<
15.         typename coordinate_type<P1>::type,
16.         typename coordinate_type<P2>::type>::type computation_type;

17.     static computation_type apply(const P1&, const P2&) { return 0; }
18. };
19. } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename P1, typename P2>
3.  auto distance(const P1& a, const P2& b) {
4.      double dx = get<0>(a) - get<0>(b);
5.      double dy = get<1>(a) - get<1>(b);
6.      return std::sqrt(dx * dx + dy * dy);
7.  }
8.  } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename P1, typename P2>
3.  auto distance(const P1& a, const P2& b) {
4.      static_assert(dimension<P1>::value == dimension<P2>::value);

5.      return std::sqrt(pythagoras<P1, P2, dimension<P1>::value>::apply(a, b));
6.  }} // my_geometry
```

разрешим использовать с геометриями разной природы

Как это работает?

напишем несколько функций одного алгоритма?

```
1.  template <typename P1, typename P2>
2.  auto distance_point_point(const P1& point1, const P2& point2) { /* ... */ }

3.  template <typename P, typename L>
4.  auto distance_point_linestring(const P& point, const L& linestring) { /* ... */ }

5.  template <typename L, typename P>
6.  auto distance_linestring_point(const L& linestring, const P& point) { /* ... */ }
```

не наш метод – одна distance для всего

Как это работает?

геометрии разной природы

```
1.  namespace my_geometry {

2.  struct point_tag {};
3.  struct linestring_tag {};

4.  namespace traits {

5.  template <typename G> struct tag {};

6.  template<> struct tag<Point> { typedef point_tag type; };
7.  template<> struct tag<Polyline> { typedef linestring_tag type; };

8.  } // traits
9.  } // my_geometry
```

Как это работает?

```
1.  namespace my_geometry {
2.  namespace dispatch {



3.  template <typename G1, typename G2>
4.  auto distance(const G1& g1, const G2& g2, point_tag, point_tag) { /* ... */ }



5.  template <typename G1, typename G2>
6.  auto distance(const G1& g1, const G2& g2, point_tag, linestring_tag) { /* ... */ }


7.  } // dispatch

8.  template <typename G1, typename G2>
9.  auto distance(const G1& g1, const G2& g2) {
10.     typename tag<G1>::type tag1;
11.     typename tag<G2>::type tag2;

12.     return dispatch::distance(g1, g2, tag1, tag2);
13. }
14. } // my_geometry
```

автоматический выбор геометрии

Как это работает?

```
1.  namespace my_geometry {
2.  namespace dispatch {

3.  template <typename Tag1, typename Tag2, typename G1, typename G2>
4.  struct distance {};

5.  template <typename P1, typename P2>
6.  struct distance<point_tag, point_tag, P1, P2> {
7.      static auto apply(const P1& a, const P2& b) { /* ... */ }
8.  };

9.  template <typename P1, typename P2>
10. struct distance<point_tag, linestring_tag, P1, P2> {
11.     static auto apply(const P1& a, const P2& b) { /* ... */ }
12. };
13. } // dispatch

14. template <typename G1, typename G2>
15. auto distance(const G1& g1, const G2& g2) {
16.     return dispatch::distance<
17.         typename tag<G1>::type,
18.         typename tag<G2>::type,
19.         G1, G2>::apply(g1, g2);
20. }
21. } // my_geometry
```

более эффективно

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename P1, typename P2>
3.  auto distance(const P1& a, const P2& b) {
4.      static_assert(dimension<P1>::value == dimension<P2>::value);

5.      return std::sqrt(pythagoras<P1, P2, dimension<P1>::value>::apply(a, b));
6.  }} // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename G1, typename G2>
3.  auto distance(const G1& g1, const G2& g2) {
4.      return dispatch::distance<
5.          typename tag<G1>::type,
6.          typename tag<G2>::type,
7.          G1, G2>::apply(g1, g2);
8.  }
9.  } // my_geometry
```

откажемся от конкретной координатной системы

Как это работает?

координатные системы

```
1.  namespace my_geometry {
2.  struct cartesian {};

3.  struct degree {};

4.  template<typename DegreeOrRadian>
5.  struct spherical { typedef DegreeOrRadian units; };

6.  // ...

7.  template <typename T1, typename T2, typename P1, typename P2, int Dim>
8.  struct strategy_distance { typedef void type; };

9.  template <typename P1, typename P2, int Dim>
10. struct strategy_distance<cartesian, cartesian, P1, P2, Dim> {
11.     typedef pythagoras<P1, P2, Dim> type;
12. };

13. template <typename P1, typename P2, int Dim>
14. struct strategy_distance<spherical<degree>, spherical<degree>, P1, P2, Dim> {
15.     typedef boost::geometry::strategy::distance::haversine<P1, P2> type;
16. };
17. } // my_geometry
```

Как это работает?

координатные системы

```
1.  namespace my_geometry {

2.  template <typename G1, typename G2, typename S>
3.  double distance(G1 const& g1, G2 const& g2, S const& strategy) {
4.      return dispatch::distance<
5.          typename tag<G1>::type,
6.          typename tag<G2>::type,
7.          G1, G2, S >::apply(g1, g2, strategy);
8.  }

9.  } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename G1, typename G2>
3.  auto distance(const G1& g1, const G2& g2) {
4.      return dispatch::distance<
5.          typename tag<G1>::type,
6.          typename tag<G2>::type, G1, G2>::apply(g1, g2);
7.  }
8.  } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename G1, typename G2>
3.  auto distance(const G1& g1, const G2& g2) {
4.      typedef typename strategy_distance<
5.          typename coordinate_system<G1>::type,
6.          typename coordinate_system<G2>::type,
7.          ???,
8.          dimension<G1>::value
9.          >::type strategy;


10.     return dispatch::distance<
11.         typename tag<G1>::type,
12.         typename tag<G2>::type, G1, G2, strategy>::apply(g1, g2, strategy{});
13. }
14. } // my_geometry
```

Как это работает?

определим тип точки для геометрий

```
1.  namespace my_geometry {
2.  namespace traits {
3.  template <typename Geometry> struct point_type {};
4.  } // traits

5.  namespace dispatch {
6.  template <typename Tag, typename Geometry> struct point_type {
7.      typedef typename traits::point_type<Geometry>::type type;
8.  };
9.  template <typename P> struct point_type<point_tag, P> { typedef P type; };

10. template <typename Linestring> struct point_type<linestring_tag, Linestring> {
11.     typedef typename boost::range_value<Linestring>::type type;
12. };
13. } // dispatch
14. } // my_geometry
```

Как это работает?

передаём для геометрии размерность, тип и систему координат её точки неё

```
1.  namespace my_geometry {
2.  namespace dispatch {
3.  template <typename GeometryTag, typename G> struct coordinate_type {
4.      typedef typename point_type<GeometryTag, G>::type point_type;
5.      typedef typename coordinate_type<point_tag, point_type>::type type;
6.  };
7.  template <typename Tag, typename G> struct dimension :
8.      dimension<point_tag, typename point_type<Tag, G>::type>
9.  {};
10. template <typename P> struct coordinate_type<point_tag, P> {
11.     typedef typename traits::coordinate_type<P>::type type;
12. };
13. template <typename P> struct dimension<point_tag, P> : traits::dimension<P>
14. {};
15. } // dispatch
16. // пересматриваем внешнюю мета-функцию, добавляем tag,
17. // продолжает зависеть только от типа геометрии
18. template <typename G> struct dimension :
19.     dispatch::dimension<typename tag<G>::type, G>
20. {};
21. template <typename G> struct coordinate_type :
22.     dispatch::coordinate_type<typename tag<G>::type, P>
23. {};
24. } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename G1, typename G2>
3.  auto distance(const G1& g1, const G2& g2) {
4.      typedef typename strategy_distance<
5.          typename coordinate_system<G1>::type,
6.          typename coordinate_system<G2>::type,
7.          ???,
8.          dimension<G1>::value
9.          >::type strategy;


10.     return dispatch::distance<
11.         typename tag<G1>::type,
12.         typename tag<G2>::type, G1, G2, strategy>::apply(g1, g2, strategy{});
13. }
14. } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  template <typename G1, typename G2>
3.  auto distance(const G1& g1, const G2& g2) {
4.      typedef typename strategy_distance<
5.          typename coordinate_system<G1>::type,
6.          typename coordinate_system<G2>::type,
7.          typename point_type<G1>::type,
8.          typename point_type<G2>::type,
9.          dimension<G1>::value
10.         >::type strategy;

11.     return dispatch::distance<
12.         typename tag<G1>::type,
13.         typename tag<G2>::type, G1, G2, strategy>::apply(g1, g2, strategy{});
14. }
15. } // my_geometry
```

определим реализации расстояния для всевозможных перестановок аргументов?

Как это работает?

нет – определим правильный порядок аргументов

```
1.  namespace my_geometry {
2.  namespace dispatch {

3.  template <typename Tag> struct geometry_id {};

4.  template <> struct geometry_id<point_tag> : boost::mpl::int_<1> {};

5.  template <> struct geometry_id<linestring_tag> : boost::mpl::int_<2> {};
6.  } // dispatch

7.  template <typename Geometry>
8.  struct geometry_id : dispatch::geometry_id<typename tag<Geometry>::type> {};

9.  template <typename G1, typename G2>
10. struct reverse_dispatch : boost::mpl::if_c<
11.     (geometry_id<G1>::value > geometry_id<G2>::value),
12.     std::true_type,
13.     std::false_type >
14. {};

15. template <typename Geometry>
16. struct reverse_dispatch<Geometry, Geometry> : std::false_type {};
17. } // my_geometry
```

Как это работает?

обратный порядок аргументов

```
1.  namespace my_geometry {
2.  namespace dispatch {

3.  template <typename Geometry1, typename Geometry2, typename Tag1, typename Tag2,
4.      bool Reverse = reverse_dispatch<Geometry1, Geometry2>::type::value>
5.  struct distance
6.  {};

7.  template <typename Geometry1, typename Geometry2, typename Tag1, typename Tag2>
8.  struct distance<Geometry1, Geometry2, Tag1, Tag2, true> :
9.      distance<Geometry2, Geometry1, Tag2, Tag1, false> {
10.     static auto apply(const Geometry1& g1, const Geometry2& g2) {
11.         return distance< // частичная специализация для каждой прямой
12.             Geometry2,
13.             Geometry1,
14.             Tag2, Tag1, // пары тэгов
15.             false >::apply(g2, g1, strategy);
16.     }
17. };

18. } // dispatch
19. } // my_geometry
```

Как это работает?

напишем абстрактный алгоритм

```
1.  namespace my_geometry {
2.  namespace dispatch {

3.  template <typename Point1, typename Point2>
4.  struct distance<Point1, Point2, point_tag, point_tag, false> {
5.      static auto apply(const Point1& point1,
6.                        const Point2& point2) { /* ... */ }
7.  };

8.  template <typename Point, typename Linestring>
9.  struct distance<Point, Linestring, point_tag, linestring_tag, false> {
10.     static auto apply(const Point& point,
11.                       const Linestring& linestring) { /* ... */ }
12. };

13. } // dispatch
14. } // my_geometry
```

Упражнение

- Написать rtree, которая в качестве ключа хранит круг (центр + радиус), написать поиск геометрий для данного случая по аналогии с поиском в прямоугольниках

СПАСИБО ЗА ВНИМАНИЕ!