УДК 004.02

**С. И. Сметанин,** аспирант, **В. А. Игнатюк,** д-р физ.-мат. наук, проф., e-mail: victor.ignatyuk@vvsu.ru
**А. А. Евстифеев,** аспирант
Владивостокский государственный университет экономики и сервиса, г. Владивосток

# Способ реализации программной веб-части системы спутникового мониторинга

*Предлагается подход к созданию программной веб-части системы спутникового мониторинга с использованием БД PostgreSQL, Javasript и C#. Обосновывается необходимость разработки собственного решения в условиях различной картографической информации и информационных данных. Предлагаемый концептуальный подход к созданию системы мониторинга, опирающейся на веб-технологии, позволит существенно упростить разработку подобных систем, направленных на решение задач любого возможного типа.*

***Ключевые слова:*** *система спутникового мониторинга, картографический движок, структура базы данных, JavaScript и C#, сервер, клиентское приложение*

## Введение

На данный момент существует множество предложений по продаже мобильных устройств, предназначенных для контроля движущихся объектов или трекеров. В большинстве из них есть функция передачи информации по GPRS на любой заданный веб-адрес через определенный интервал времени.

Системы спутникового мониторинга должны решать следующие задачи:

1) выбор карты и ее отображение;

2) отображение информации об объектах мониторинга (скорость, угол поворота, расход топлива);

3) отображение полигонов, линий, точек;

4) отображение информации, связанной с полигонами, линиями, точками (всплывающие подсказки);

5) математические действия (подсчет пройденного пути, площади полигона, принадлежности точки полигону).

Большинство систем мониторинга используют карты Google Maps и OpenStreetMap, но поставленные перед системой задачи могут быть различными. Иногда требуется использовать картографическую информацию из собственных источников, и готовые фреймворки, рассчитанные на перечисленные выше сервисы, приходится перерабатывать под требования разрабатываемой системы. Это не всегда возможно и актуально в условиях поставленной задачи.

В данной работе представлена структура авторской системы мониторинга, по образцу которой можно создать универсальную систему, работающую с данными различных типов и категорий. Она основана на синтезе трех составляющих:

1) базы данных PostgreSQL;

2) ASP.NET (Active Server Pages) — технологии создания веб-приложений и веб-сервисов от компании Майкрософт;

3) собственного фреймворка, реализованного на языке JavaScript.

В работе описаны основные части системы и способы их реализации, а также возможные проблемы, с которыми разработчик может столкнуться при создании собственной системы спутникового мониторинга, и способы их решения.

## Картографическая информация и учет ее в базе данных

Система спутникового мониторинга должна основываться на данных векторных карт, чтобы использовать и хранить информацию для интерактивного взаимодействия с географическими объектами: зданиями и помещениями, изображенными в качестве полигонов; дорогами, показанными как линии; отметками карты (например, отметки видеокамер, телефонов, магазинов), представленные точками.

При разработке данной системы мониторинга была использована БД PostgreSQL. На специфические типы данных этой БД приведены подробные объяснения.

Поскольку у любой карты есть различные уровни масштабирования, и, следовательно, различные объекты, отображаемые на разных уровнях, поэтому необходимо создать либо отдельные таблицы для

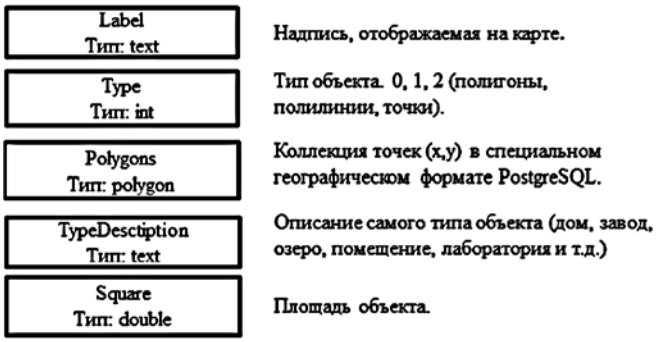


**Рис. 1. Список столбцов таблицы MapObjects**

каждого из уровней, либо общую таблицу и представления (View), разделяемые по параметру $K$ — коэффициенту масштабирования.

Структура созданных таблиц, каждая из которых имеет идентификатор MapObjects ($K = 1...n$), представлена на рис. 1.

Все объекты необходимо хранить в виде полигонов, поскольку это облегчает дальнейшие математические преобразования и действия с ними. Если в формате карт есть такие объекты, как линии или точки, то их следует преобразовать в полигоны по следующему алгоритму.

Если объект является точкой, то вокруг него описывается прямоугольник, где $n$ — ширина точки:

```
PointF[] points = new PointF[ ] {
    new PointF(point.X – n, point.Y – n),
    new PointF(point.X + n, point.Y – n),
    new PointF(point.X + n, point.Y + n),
    new PointF(point.X – n, point.Y + n),
    new PointF(point.X – n, point.Y – n) };
```

Если объект является линией, то прямоугольник образуется с помощью последовательного обхода всей линии точками — от начала к концу, и от конца к началу. При этом значение координаты X, по сравнению с базовой точкой линии не изменяется, а координата Y смещается на ±n. Кроме того, необходимо учитывать угол поворота от текущей точки к следующей точке в массиве, чтобы полностью повторить кривую линии новой фигурой — полигоном.

```
for (int i = Ø; i < points.Length; i++)
{ //движемся от начала к концу
    PointF p1 = new pointF(points[i].X, points[i].Y – n);
    if (i != points.Length – 1) // если это не последняя точка, то найти угол между этой и следующей точкой в массиве
        angle = findAngle(points[i], points[i + 1]);
    p1 = rotatePoint(points[i].X, points[i].Y, p1, angle); // повернуть созданную точку на заданный угол и добавить ее в новый массив точек полигона
    p.Add(p1);
}
for (int i = points.Length – 1; i != –1; i--)
{ //движемся от конца к началу. Действия повторяются.
    PoiintF p1 = new PointF(points[i].X, points[i].Y + n);
    if (i !: Ø)
        angle = findAngle(points[i – 1], points[i]);
    p1 = rotatePoint(points[i].X, points[i].Y, p1, angle);
    p.Add(p1);
}
pointF p2 = new PointF(points[Ø].X, points[Ø].Y – n);
p2 = rotatePoint(points[Ø].X, points[Ø].Y, p2, angle);
p.Add(p2);
```

Функция findAngle основана на нахождении угла между двумя векторами: вектора, образованного точками, переданными в функцию, и единичного вектора с координатами (1, 0). Функция rotatePoint вращает точку создаваемого полигона вокруг первоначальной точки линии с учетом вычисленного угла. Она выглядит следующим образом:

```
private static float findAngle(PointF p1, PointF p2)
{
    System.Windows.Vector vector1 = new System.Windows.Vector(p2.X – p1.X, p2.Y – p1.Y);
    System.Windows.Vector vector2 = new System.Windows.Vector(1, Ø);
    float labelAngle = (float)System.Windows.Vector.AngleBetween(vector2, vector1);
    if (labelAngle >= 180/2)
        labelAngle –= 180;
    if (labelAngle < –180/2)
        labelAngle += 180;
    return labelAngle;
}
```

Впоследствии вхождение курсора пользователя в выбранный полигон может быть вычислено запросом к базе данных PostgreSQL. Важно отметить, что данная БД поддерживает функции работы с географическими типами [1], поэтому ее использование предпочтительно при построении собственной системы мониторинга.

Стандартный запрос такого вида представлен ниже, где point — параметр с географическим типом точки, передаваемый при выполнении хранимой процедуры:

```
SELECT Label, Type, TypeDescription,
    Polygons FROM MapObjectsK =
= 256 WHERE (select :point @ Polygons);
```

Специальный символ @ задает функцию Contains, при этом проверяется, входит ли указанная точка в границы полигона. Это может использоваться для различных способов взаимодействия с объектами: их передвижения, изменения различных характеристик или вызова динамической подсказки. Иногда возникает необходимость также обозначить область, доступную для взаимодействия с объектами карты. Это может быть как вся карта целиком, так и некоторый отдельный ее участок, чтобы не вызывать обработчиков клика пользователя при выходе за границы рабочей области карты.

В разработанной системе мониторинга такая таблица называется MapLimits и включает столбцы xMin, xMax, yMin, yMax, K. Таким образом, при обработке события клика мыши, в первую очередь проверяется, входят ли координаты курсора в установленную область карты, после чего запускается то событие, которое и должно было быть применено изначально.

### Рисование карт и взаимодействие с ними

При создании собственных карт разработчик может использовать различные форматы векторной графики. В разработанной системе мониторинга, в качестве карт использовался так называемый "польский формат карт" — свободно распространяемый формат векторной графики, предоставляющий широкие возможности для прорисовки местности [2].

Несмотря на использование векторного формата, ввиду вопросов, связанных с чрезмерной нагрузкой на сервер, был применен "тайловый подход". Тайлинг — метод создания больших изображений из маленьких фрагментов (тайлов) одинаковых размеров, подобно картине из мозаики.

Вся карта разбивается на строки и столбцы. Допустим, целиком карта занимает 1024 пикселя по ширине, и столько же по высоте. Если принять размер тайла равным 256 пикселей, то карту можно условно разбить на 4 строки, каждая из которых содержит 4 столбца. Пример такого подхода демонстрируется на рис. 2.

Каждый из тайлов заранее отрисовывается и сохраняется в папке на сервере, таким образом, по запросу пользователя достаточно только передать уже существующий тайл, а не рисовать все изображение заново.

Координаты карты также могут быть представлены в разных датумах: например, в датуме WGS-84. Следовательно, для того чтобы прорисовать тайл, необходимо сопоставить его координаты (широту и долготу в случае WGS-84) с пикселями карты. Это делается с помощью афинного преобразования [3].

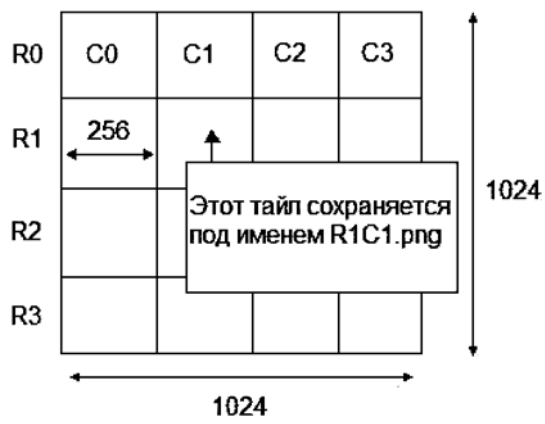


**Рис. 2. Разметка карты на тайлы**

Взаимосвязь пикселей экрана и географических координат можно представить в виде системы уравнений с неизвестными коэффициентами:

$$x' = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2; \quad (1)$$

$$y' = b_0 + b_1 x + b_2 y + b_3 x^2 + b_4 xy + b_5 y^2, \quad (2)$$

где $x$, $y$ — координаты в исходной системе координат; $x'$, $y'$ — координаты в конечной системе координат (пиксели); $a_0$—$a_5$, $b_0$—$b_5$ — коэффициенты (неизвестны).

Точная привязка проводится по шести точкам, таким образом, требуется шесть пар таких уравнений для нахождения соответствующих коэффициентов. Данные системы уравнений можно решить с использованием уже готовых библиотек. Например, в рассматриваемой работе это реализовано с помощью библиотеки ALGLIB, которая реализует метод LU-декомпозиции на языке C#. Она доступна для скачивания по приведенной ссылке [4].

После предварительного создания карты и ее разбиения на тайлы необходимо средство для вывода этих изображений на экран и взаимодействия с ними. Для этих целей автором статьи был разработан собственный картографический движок, созданный на основе JavaScript, без использования каких-либо дополнительных библиотек или фреймворков. В процессе разработки был использован только плагин jQuery для облегчения работы с объектной моделью документа (DOM).

Отрисовка тайлов происходит с использованием HTML 5 Canvas. Реализация движка — модульная, и состоит из двух основных частей: lottaMap.Map и lottaMap.Objects. Неполное ее описание приведено в работе [5].

Для начала работы необходимо в *html*-разметке указать элемент <canvas id="canvas"></canvas>.

### Внутренняя механика

После инициализации объекта необходимо рассчитать и сформировать тайлы, полностью покрывающие выбранный элемент canvas. Для этого необходимо знать его ширину и высоту, после чего определяем число тайлов. К ним добавляем тайлы буфера — дополнительные, хранящиеся в памяти изображения за пределами видимой области карты, необходимые для быстрого отображения карты при ее перемещении пользователем.

К объекту lottaMap.Map относится подмножество других объектов, каждый из которых отвечает за те или иные действия пользователя. К перемещению карты относится объект draggableMap — он проверяет, передвинул ли пользователь карту на определенное расстояние, после чего меняет значения глобальных переменных mapX и mapY, отвечающих за координаты верхнего левого угла карты. После этого запускается функция отрисовки карты по заданным координатам.

```
dragMap = function () {
    // перетащить карту
    var dX = startPoint.x – endPoint.x,
        dY = startPoint.y – endPoint.y;
    mapX –= dX;
    mapY –= dY;
    tiles.tilesMX += dX; // смещение тайлов по x. y
    tiles.tilesMY += dY;
    ctx.translate(dX, dY); // контекст основного канваса
    if (useObjects)
            ctxBackground.translate(dX, dY) // канвас заднего фона, для взаимодействия с объектами
    lottaMap.Map.drawMap(); // вызывается рисование карты
    tiles.unvisibleTilesCollector(); // запускается "сборщик мусора"
};
```

За различные действия с тайлами, а именно за их загрузку, формирование и удаление лишних тайлов, отвечает объект tiles. Поскольку все изображения содержатся в массиве, определенном в данном объекте, то от тех тайлов, которые не попадают в область видимости карты, необходимо избавиться, чтобы они не расходовали дополнительную память. Это делается с помощью "сборщика мусора". Он рассчитывает границы текущей области, попадающей в пределы элемента canvas. Получившийся таким образом прямоугольник проверяется на пересечение с прямоугольником каждого элемента из массива тайлов tilesStore — если пересечения не происходит, то данный тайл необходимо удалить.

```
unvisibleTilesCollector: function () {
    var collectorLimit = 4; // количество тайлов за границей видимой карты, которые будут учитываться сборщиком
    // сборщик мусора (тайлов, вышедших за границы зоны просмотра)
    var borderRightWidth = mapX + canvas.width + collectorLimit * tileSize, // правая координата X
    borderRightHeight = mapY + canvas.height + collectorLimit * tileSize, // правая нижняя координата Y
    borderLeftWidth = mapX – collectorLimit * tileSize, // левая координата X
    borderLeftHeight = mapY – collectorLimit * tileSize, // левая нижняя координата Y
    tile, // переменная, использующаяся для получения названия тайлов из их массива
        obj1 = [borderLeftWidth, borderLeftHeight, borderRightWidth, borderRightHeight],
        obj2 = [ ],
        xStart, xEnd, yStart, yEnd;
    for (tile in tilesStore) { // tilesStore – массив тайлов
        if (tilesStore.hasOwnProperty(tile)) {
            xStart = tilesStore[tile].x; // координаты прямоугольника самого тайла
            xEnd = tilesStore[tile].x + tilesSize;
            yStart = tilesStore[tile].y;
            yEnd = tilesStore[tile].y + tilesSize;
            obj2 = [xStart, yStart, xEnd, yEnd];
            if ( !(((obj2[0] <= obj1[0] && obj1[0] <= obj2[2])
                || (obj1[0] ⩽ obj2[0] && obj2[0] <= obj1[2]))
            && ((obj2[1] <= obj1[1] && obj1[1] <= obj2[3])
                || (obj1[1] ⩽ obj2[1] && obj2[1] <= obj1[3])))) {
                delete tilesStore[tile]; // если тайл не попадает в проверяемую область, удалить его
            }
        }
    }
}
```

### Масштабирование

После того как срабатывает событие "mousewheel" мыши, выполняются следующие основные действия.

Сначала вычисляется новый уровень карты — это происходит за счет переменной delta из события прокрутки мыши. В зависимости от того, в какую сторону крутилось колесо мыши, ее значение будет либо положительным, либо отрицательным. Использовать ее очень удобно, поскольку достаточно просто отнять от текущей переменной уровня значение delta.

Затем происходит вычисление коэффициента масштабирования, как отношения между значениями старого и нового уровней. Это используется для вычисления "шагов" — определения, сколько раз в цикле нужно проводить операцию увеличения или отдаления тайла для создания плавного визуального эффекта.

```
if (levels.scales[current] > levels.scales[oldL]) // если новый коэффициент масштабирования больше старого, то рассчитываем разницу между ними как новый уровень / старый уровень, иначе наоборот
    diff = levels.scales[currentL] / levels.scales[oldL];
else
    diff = levels.scales[oldL] / levels.scales[currentL];
steps = Math.abs(Math.round(Math.log(diff) / Math.log(zoom))); // вычисляем количество шагов. Zoom = 2, если колесо крутят вверх, 0,5 — если вниз.
```

До того как начать операцию изменения масштаба, старые тайлы копируются в буфер.

```
tilesCopyStore = [ ];
for (var tile in tilesStore) {
    tilesCopyStore[tile] = tilesStore[tile];
}
```

После чего средствами canvas происходит увеличение или отдаление тайла из копии, сделанной на предыдущем шаге. Для этого вызывается функция масштабирования тайла, причем вызывается то число раз, которое было определено переменной steps, через последовательно инкрементируемые задержки времени i * 80.

```
for (var i = 0; i < steps; i++) {
    setTimeout (function () {
        tiles.scaleTiles(zoom, x, y);
    }, i * 80};
}
```

Известны координаты места в пикселях, над которым было прокручено колесо мыши. Эти координаты переводятся в координаты карты, после чего умножаются на коэффициент zoom. Необходимо также учитывать смещение тайлов при масштабировании canvas — за эти значения отвечают переменные this.tilesMX и this.tilesMY. Кроме того, на данном шаге задается значение для масштабирования скопированных тайлов текущего уровня scaleT, также использующихся для создания необходимого визуального эффекта:

```
clearTimeout(timeout); // масштабирование происходит в несколько шагов, по таймеру
ScaleT *= zoom;
center.oldCenter.x = x – this.tilesMX;
center.oldCenter.y = y – this.tilesMY;
center.newCenter.x = Math.round(center.oldCenter.x * zoom);
center.newCenter.y = Math.round(center.oldCenter.y * zoom);
```

Значение точки прокрутки колесика фиксируется, после чего сравнивается со значением этой точки с учетом ее масштабирования. Карта смещается на разницу между ними:

```
    ctx.translate(–Math.round(center.newCenter.x – center.oldCenter.x), – Math.round(center.newCenter.y – center.oldCenter.y)); // смещение карты
    if (useObjects) // смещение буфера на ту же величину, если используются объекты (подробнее ниже)
    ctxBackground.translate(–Math.round(center.newCenter.x – center.oldCenter.x), – Math.round(center.newCenter.y – center.oldCenter.y));
    this.tilesMX += – Math.round(center.newCenter.x – center.oldCenter.x); // вычисления нового значения переменной для смещения тайлов
    this.tilesMY += – Math.round(center.newCenter.y – center.oldCenter.y);
    mapX –= – Math.round(center.newCenter.x – center.oldCenter.x); // те же самые действия необходимо осуществить для глобальных переменных mapX и mapY, показывающих координаты верхнего левого угла экрана
    mapY –= – Math.round(center.newCenter.y – center.oldCenter.y);
    drawCopy();
```

Функция drawCopy вызывает прорисовку в цикле снятой копии тайлов, помноженной на коэффициент масштабирования:

```
    ctx.drawImage(thisTile.img, Math.ceil(thisTile.x * scaleT), Math.ceil(thisTile.y * scaleT), Math.ceil(tileSize * scaleT), Math.ceil(tileSize * scaleT));
```

После этого запускается таймер для начала рисования тайлов нового уровня. Задержка в рисовании установлена на 90 мс, и если все шаги, использующиеся для плавного увеличения копии, еще не выполнены, то это событие отменяется, как видно из кода выше. Таким образом, до того как весь процесс визуального масштабирования тайлов не завершится, событие таймера по рисованию новых тайлов не будет задействовано:

```
timeout = setTimeout(function () {
    tiles.clearTiles();
    lottaMap.Map.drawMap(true);
}, 90)
```

### Взаимодействие с объектами карты

Объект lottaMap.Objects отвечает за взаимодействие с различными структурами на карте, начиная от подсказок касательно зданий, дорог и прочих векторных объектов, до управления непосредственно данными трека.

Поскольку canvas не поддерживает векторные данные, определение объекта, с которым взаимодействует пользователь, должно быть реализовано самостоятельно.

Один подход заключается в использовании математических формул, например формулы окружности, для того чтобы вызывать событие клика мышью по точке как результат проверки вхождения координат мыши в окружность. Но достаточно сложные фигуры, такие как многоугольники различной формы, тяжело описать подобным образом.

Подход, который использовали авторы статьи, связан с применением второго, буферного canvas'а, который остается скрытым для пользователя и хранится лишь в памяти браузера. Каждый объект отрисовывается на нем своим собственным, уникальным цветом, а ключ RGB, отвечающий за данный цвет, хранится в массиве. Таким образом, общее число возможных ключей $255 \times 255 \times 255 = 16\,777\,216$, чего более чем достаточно в большинстве случаев.

Поскольку сам фон прозрачен, считывание значений цветности пикселя в месте клика мыши укажет, какой объект использовался на данный момент.

Необходимо создать элемент canvas и получить с него контекст:

```
    var canvasBackground = document.createElement ('canvas'), // канвас для проверки клика по точкам трекам
    ctxBackground = canvasBackground.getContext('2d');
```

Важно отметить, что над этим элементом также необходимо проводить операции передвижения и масштабирования для того чтобы синхронизировать видимый и фоновый объекты canvas между собой.

Объекты, с которыми будет проводиться взаимодействие, необходимо предварительно нарисовать на фоновом canvas'е. Если это фигура, заданная определенными координатами, то сделать это не представляет сложности. Проблема возникает, когда необходимо использовать готовые изображения, например значки транспортных средств. В этом случае необходимо проанализировать каждый пиксель изображения и закрасить непрозрачные пиксели выбранным цветом, соответствующим индивидуальному ключу объекта. Это можно сделать с помощью метода putImageData, принадлежащего контексту элемента canvas, что накладывает следую-

щее ограничение: для каждого нового изображения необходимо создавать свой собственный фоновый canvas. Тем не менее, его не требуется хранить постоянно — после отрисовки изображения его можно сохранить в файл или память браузера с помощью метода canvas'a — toDataURL("image/png"). Метод контекста canvas'a, getImageData позволяет получить по соответствующим координатам *x* и *y* прямоугольник, ширину и высоту которого необходимо установить равными единице. Этот объект содержит информацию о цветности пикселей экрана (0 — красный, 1 — зеленый, 2 — голубой, 3 — альфа-канал). Проверка альфа-канала позволяет установить, содержится ли по выбранным координатам пиксель закрашиваемого изображения, в этом случае его значение будет больше нуля.

Чтобы создать новый пиксель вместо предыдущего с необходимым и установленным цветом, требуется использовать метод createImageData. Так же, как и в методе getImageData, ширину и высоту области данных изображения следует установить равными единице. Объект, созданный этим методом, содержит в себе массив data, в элементы которого [0]...[3] необходимо занести значения цвета RGB-ключа, установив значение альфа-канала в состояние "непрозрачный" — оно должно быть равно 255. Метод putImageData помещает данный объект в фоновый canvas, созданный на первом шаге.

Код, приведенный ниже, демонстрирует попиксельный анализ изображения и закрашивание непрозрачных пикселей RGB-цветом индивидуального ключа объекта:

```
var imageCanvas = document.createElement('canvas'), // создается фоновый canvas
imageCtx = imageCanvas.getContext('20'); // его контекст
imageCanvas.width = img.width; // размеры canvas'a устанавливаются равными размерам изображения
imageCanvas.height = img.height;
imageCtx.drawImage(img, 0, 0); // отрисовка изображения на временном фоновом canvas'e
for (var h = 0; h < img.height; h++) { // перебор пикселей в цикле
    for (var w = 0; w < img.width; w++) {
        if (image Ctx.getImageData(w, h, 1, 1).data[3] > 0) { // проверка альфа-канала
            var imageData=imageCtx.createImageData(1, 1);
            var d = imageData.data;
            d[0] = this.rgb[0];
            d[1] = this.rgb[1];
            d[2] = this.rgb[2];
            d[3] = 255;
            imageCtx.putImageData(imageData, w, h);
        }
    }
}
this.backgroundImage = new Image();
this.backgroundImage.src = imageCanvas.toDataURL("image/png"); // преобразование canvas'a в объект Image
```

Аналогичным образом происходит проверка RGB-ключа при клике мышью по видимому canvas'у. По этим координатам проверяется значение пиксельных данных фонового элемента canvas, после чего найденный RGB-ключ преобразуется в строку и ищется в массиве ключей всех объектов.

```
var isOpaque = pixel[3] === 255; // если пиксель непрозрачен, то читаем его параметры цветности
if (isOpaque) {
    var r = pixel[0],
        g = pixel[1],
        b = pixel[2],
        id = r.toString() + g.toString() + b.toString(); // формируем ключ
    return keys[id];
}
```

Таким образом, возможно реализовать полноценное взаимодействие с объектами без использования векторной графики.

Необходимо подчеркнуть, что все основные действия с интерфейсом пользователя происходят с использованием JavaScript. Сервер занимается только получением необходимых данных из БД и выполняет такие второстепенные задачи, как перекрашивание изображения транспортного средства и сохранение его в папке на сервере.

Тем не менее, чтобы сервер мог осуществить запрос к базе данных, необходимо уведомить его о данной задаче. Наиболее простой способ организовать такое взаимодействие — это AJAX (от англ. *Asynchronous Javascript and XML* — асинхронный JavaScript и XML). Во фреймворке jQuery уже есть объекты для использования AJAX [6], при его применении реализовать процесс взаимодействия с сервером не представляет сложности:

```
$.ajax({
    type: "POST", // POST или GET сообщение
    url: "AjaxData.aspx", // страница, которая примет сообщение
    data: str, // данные
    success: function (msg) { // функция, которая вызывается при условном приеме ответных данных с сервера
    }
});
```

### Передача данных от сервера к клиенту на примере динамических изображений

В разработанном авторами картографическом движке использованы различные типы объектов — как географические объекты векторной карты, так и объекты, использующиеся для навигации по треку (тип транспортного средства, начало—конец трека, метка "остановки" объекта мониторинга).

В то время как объекты первого типа уже отрисованы на тайлах карты и занесены в базу данных, объектам второго типа требуются готовые изображения, которые необходимо предварительно передать

| | |
|---|---|
| ObjectType<br>Тип: text | Тип объекта. Автомобиль, грузовой автомобиль, пешеход и т.д. |
| ImagePath<br>Тип: text | Путь к изображению в «не активированном» виде – при отсутствии наведения на него курсора мыши пользователя |
| ActiveImagePath<br>Тип: text | Путь к изображению в «активированном» виде |
| DefaultColor<br>Тип: text | Основной цвет изображения по умолчанию, который будет использоваться при поиске пикселей для перекрашивания |

**Рис. 3. Список столбцов таблицы ObjectImages**

**Рис. 4. Изображение для таблицы ObjectImages**

клиентскому приложению системы мониторинга — в данном случае, браузеру пользователя.

Рассмотрим следующий пример: пользователю доступна возможность изменения цвета трека наблюдаемого им подвижного объекта. Но будет более наглядно, если и сам трек, и изображение, отвечающее за данный объект, можно перекрашивать в зависимости от выбранного пользователем цвета.

Подобные типы объектов хранятся в таблице ObjectImages. Объекты, изображения которых не изменяются со временем, хранятся в таблице MarkImages.

Структура таблицы ObjectImages продемонстрирована на рис. 3.

Таблица MarkImages отличается отсутствием в ней столбца DefaultColor.

Поскольку изображения таблицы ObjectImages являются динамически изменяемыми, сами изображения также должны создаваться в ходе работы приложения.

Изображения для данных объектов подбираются таким образом, чтобы они имели максимально однородный цвет. За счет этого упрощается процесс поиска пикселей нужного цвета для алгоритма перекрашивания базового изображения.

Пример такого изображения продемонстрирован на рис. 4.

При смене цвета трека, меняется и цвет значка, отображающего текущий объект:

```
bitm = new System.Drawing.Bitmap(serverPath + defaultImagePath); // получаем базовое изображение, которое будет перекрашено
for (int y = 0; y < bitm.Height; y++)
{
    for (int x = 0; x < bitm.Width; x++)
    { // последовательно обходим изображение по пикселям
        Color rgb = bitm.GetPixel(x, y);
        if (rgb/A > 0 && rgb.R == defaultColor.R && rgb.G == defaultColor.G && rgb.B == defaultColor.B) // проверяем пиксели на соответствие стандартному цвету изображения, которые требуется закрасить новым цветом
        {
            bitm.SetPixel(x, y, newColor); // перекрашиваем пиксель
        }
    }
}
```

Новое изображение сохраняется по указанному пути с генерированным временным именем. При следующей смене цвета трека это изображение должно быть удалено и заменено новым.

Для каждого пользователя генерируются таблицы userInfo (рис. 5) и userData.

Колонки LastRepaintingType, LastImageName, LastAImageName и LastRepaintingColor из таблицы userInfo (рис. 5), применяются для условия проверки изменения изображения. Само условие выглядит следующим образом:

```
bool needToPaint = (LastRepaintingType == String.Empty || LastRepaintingColor == String.Empty);
needToPaint = needToPaint || ((LastRepaintingColor != String.Empty ? ColorTranslator.FromHtml("#" + 1 LastRepaintingColor).Name !+ Color.Name : true) || TransportType != LastRepaintingType);
```

Таким образом, если значения LastRepaintingType и LastRepaintingColor пусты — это значит, что изображение загружается в первый раз, и должно быть перекрашено под установленный пользователем цвет. Если же эти значения уже заполнены, то проверяется, равен ли последний сохраненный цвет LastRepaintingColor — новому цвету Color (если они равны, то необходимость в перерисовке изображения отсутствует), после чего проверяется соответствие столбцов TransportType и LastRepaintingType, поскольку тип объекта мог измениться за прошедшее время. В этом случае его изображение также необходимо изменить.

Данные мониторинга хранятся в таблице userData. Она включает в себя такие столбцы, как:

1) Point, тип point — точки ($x$, $y$), в координатах карты (пикселях экрана);

| | | | |
|---|---|---|---|
| Name<br>Тип: text | Имя объекта мониторинга | LastImageName<br>Тип: text | Имя последнего использованного изображения объекта |
| Color<br>Тип: text | Цвет трека (и иконки объекта) | | |
| TransportType<br>Тип: text | Текущий тип объекта выбранного объекта (может меняться) – автомобиль, грузовой транспорт и т.д. | LastAImageName<br>Тип: text | Имя последнего активного изображения объекта |
| LastRepaintingType<br>Тип: text | Последний используемый тип объекта. | LastRepaintingColor<br>Тип: text | Последний сохранённый цвет объекта |

**Рис. 5. Структура таблицы userInfo**

2) GPSpoint, тип point — эти же точки, но в географических координатах (широта и долгота);

3) Date, тип timestamp — дата и время получения данных;

4) Name — имя объекта мониторинга.

Остальные параметры вторичны и используются для информационных отчетов. В зависимости от поставленных задач, в данную таблицу могут быть добавлены дополнительные столбцы, например, для данных скорости, уровня топлива или высоты.

## Заключение

В данной статье авторами была рассмотрена реализация следующих основных частей системы спутникового мониторинга.

1. Предпочтительная структура базы данных.

2. Взаимодействие с векторными объектами на карте.

3. Реализация картографического движка на языке JavaScript.

4. Взаимодействие с сервером для получения необходимой информации по треку.

Результатом проделанной работы является программа для системы спутникового мониторинга, способная использовать карты польского формата, адекватно отображать объекты мониторинга и демонстрировать графики различных параметров за указанный промежуток времени — уровень топлива, скорость движения, показания различных датчиков. Хотя в подобных системах часто используют готовые решения, но они оказываются совершенно бесполезными, когда ситуация требует использовать свои собственные картографические данные или применять различные параметры, которые не предусмотрены в разработанной системе мониторинга.

Информация, указанная в данной статье, послужит для применения в разработке систем мониторинга различной направленности: мониторинга транспорта, физических лиц, событий (пример — уведомление о возгорании в помещении). Приведенные алгоритмы позволяют быстро создать свою собственную систему, не требуя использовать готовые фреймворки или приложения в рамках поставленной задачи.

### Список литературы


1. **Уорсли Дж., Дрейк Дж.** PostgreSQL. Для профессионалов. СПб.: Питер, 2003. 613 с.

2. **cGPSmapper** User Manual. URL: http://cgpsmapper.com/download/cGPSmapper-UsrMan-v02.5.pdf (дата обращения 9.08.2014).

3. **Полиномиальные** преобразования — математика. URL: http://gis-lab.info/qa/polynom-calc.html (дата обращения 9.08.2014).

4. **ALGLIB** Free Edition. URL: http://www.alglib.net/download.php (дата обращения 9.08.2014).

5. **Сметанин С. И., Савченко И. С.** Разработка системы мониторинга, использующей веб-интерфейс на базе GPS/ГЛОНАСС // Материал XIV международной научно-практической конференции "Интеллектуальный потенциал вузов — на развитие Дальневосточного региона России и стран АТР". Владивосток, 2012. С. 82—85.

6. **Бибо Б., Кац И.** jQuery. Подробное руководство по продвинутому JavaScript. М.: Символ-Плюс, 2009. 376 с.



**S. I. Smetanin,** Graduate Student, **V. A. Ignatyuk,** Dr. Sci Sciences, prof., e-mail; victor.ignatyuk@vvsu.ru,
**A. A. Evstifeev,** Graduate Student
Department of Electronics, Vladivostok State University of Economics and Service, Vladivostok


# Implementation of the Software Part of the System of Satellite Monitoring


*The text shows an approach to the creation of the software part of the satellite monitoring system. Also, it talks about the necessity to develop their own solutions to a wider variety of use cartographic information and other data information. The proposed conceptual approach to the creation of a monitoring system, which is based on web technology, will greatly simplify the development of such systems, aimed at addressing the problems of any possible type. In particular, realization of the main functions which any cartographical cursor — scalings and movements of the card has to realize is given in this article. Also the example ot comparison of pixels of the card to geographical coordinates of the objects which are stored in a database that allows to connect them in program functions is given. Code examples are given. The code is given in the C# language as server part and JavaScript as client part.*



***Keywords:*** *satellite monitoring system, engine mapping, database design, JavaScript, C#, default server, the client application*


### References


1. **Worsley J. C., Drake J. D.** *Practical PostgreSQL. O'Reilly*. Sebastopol, CA, 2002. 613 p.

2. **cGPSmapper** User Manual. URL: http://cgpsmapper.com/download/cGPSmapper-UsrMan-v02.5.pdf. 2014.

3. **Polinomial'nye** *preobrazovaniya — matematika*. URL: http://gis-lab.info/qa/polynom-calc.html (accessed 9.08.2014).

4. **ALGLIB** Free Edition. URL: http://www.alglib.net/download.php (accessed 9.08.2014).

5. **Smetanin S. I., Savchenko I. S.** Razrabotka sistemy monitoringa, ispol'zuyushchei veb-interfeis na baze GPS/GLONASS. *Materialy XIV mezhdunarodnoj nauchno-prakticheskoi konferentsii: "Intellektual'nyi potentsial vuzov — na razvitie Dal'nevostochnogo regiona Rossii i stran ATR"*. Vladivostok, 2012. P. 82—85 (in Russian).

6. **Bibeault B., Katz Y.** *jQuery in Action, Manning*. Greenwich, CT. 2008. 376 p.