Санкт-Петербургский Государственный Университет
Математико-механический университет
Кафедра системного программирования

Визуальное программирование при помощи мыши

Курсовая работа студентки 345 группы
Осечкиной Марии Сергеевны

Научный руководитель ст. преп. Ю. В. Литвинов

Санкт-Петербург
2010

Оглавление

Введение..3
Существующие решения...5
Создание списка идеальных жестов...7
Алгоритм распознавания жестов...9
Фильтрация пути..10
Генерация ключа по пути мыши...11
Расстояние Левенштейна...11
Алгоритмы создания строки..12
Алгоритм распознавания жестов без генерации строки-ключа.........................14
Связи..16
Заключение..17
Список литературы...18

Введение

В условиях современного роста компьютерных технологий всё актуальнее становится проблема автоматизации разработки различных систем. CASE-системы (Computer Aided Software Engineering) – средства разработки программных систем. Быстрый рост мощности вычислительной техники, всё возрастающий спрос на программную продукцию ведет к стремительному развитию универсальных и удобных средств разработки. CASE-системы движутся в направлении автоматизации разработки, оптимизации и автоматической генерации кода.

На кафедре системного программирования математико-механического факультета СПбГУ разработана одна из таких CASE-систем — QReal. В QReal реализуется генерация графических редакторов для специализированных визуальных языков.

QReal предоставляет список объектов, которые можно с помощью операции drag-and-drop перетаскивать на рабочее поле. Хочется максимально ускорить добавление объектов на рабочее поле с помощью подручных средств — клавиатуры и мыши. Таким образом, в концепцию QReal естественным образом вписываются мышиные жесты — способ управления программами в компьютере при помощи движений мыши, образующих команды. То есть в идеале вообще не придется утруждать себя обращениям к списку объектов и перетаскиванием их на рабочее поле. Достаточно просто изобразить жест, который создаст нужную фигуру.

Итак, задача состоит в том, чтобы реализовать в CASE-системе QReal поддержку мышиных жестов, удовлетворяющих следующим требованиям:

1) Расширяемость — при добавлении новых объектов должно быть просто добавить соответствующий жест. Ещё лучше, чтобы жест генерировался сам, а разработчик при необходимости мог его просто подкорректировать.

2) Легкость рисования. Нелогично заставлять пользователя тратить время на рисование сложных фигур, будет проще перетащить нужный объект из предоставленного списка на рабочее поле. Поэтому должно быть просто рисовать жесты.

3) Разнообразие жестов. Некоторые утилиты в ответ на жест пользователя предлагают список наиболее подходящих объектов. Для экономии времени желательно, чтобы любому жесту соответствовал максимум один объект. Таким образом, у разных объектов жесты должны отличаться как можно сильнее.

В общем случае алгоритм распознавания жестов работает следующим образом: есть список идеальных жестов, с каждым из которых сравнивается жест, изображенный пользователем. Пользователь рисует фигуру, удерживая нажатой кнопку мыши. Если программа распознает фигуру, найдя похожую в списке идеальных жестов, то выполняется ассоциированная с ней команда. В нашем случае генерируется объект из загруженного редактора.

Существующие решения

1. Браузеры Opera, Mozilla Firefox.
   В Opera введены простейшие жесты, вызывающие наиболее часто применяемые команды. К примеру, чтобы вернуться к предыдущей странице, можно нажать кнопку «Back» в панели браузера, а можно выполнить движение мышью влево при зажатой правой кнопке. Чтобы перейти к следующей странице (кнопка «Forward»), нужно двигать мышь вправо, для обновления страницы (кнопка «Reload» – вверх-вниз, чтобы открыть новую вкладку (File - New Tab) – вниз. Ясно, что при таких абстрактных жестах от алгоритма распознавания не требуется большая точность — достаточно просто разбить жест на направления и выделить 1-3 вектора наибольшей длины.

2. Некоторые игры: в игре Myth жесты указывали бойцам куда двигаться, а в Arx Fatalis и Black & White вызывали колдовские чары.
   Так как в QReal редакторы генерируются, необходимо, чтобы и жесты не приходилось писать вручную. В играх такие тонкости, очевидно, излишни — набор жестов там статичен.

3. Утилиты Sensiva, StrokeIt, Mojo Mouse Gesture, которые вводят мышиные жесты в любую программу.
   Недостаток состоит в том, что добавляется ограниченное число заранее определенных команд: сохранение (Ctrl + S), распечатка (Ctrl + P), копирование (Ctrl + C), вставка (Ctrl + V). В QReal необходимо реализовать на порядок больше различных жестов. И если в вышеперечисленных утилитах можно обойтись перебором всевозможных путей мыши, по которым вызывается та или иная команда, то в QReal перебирать придется слишком много вариантов. Ограничиться простейшими жестами (вверх, вниз, вправо, влево) невозможно, так как объектов слишком много и может возникнуть конфликт между двумя командами, у которых будет один и тот же жест. Некоторые программы в таких случаях по жесту предоставляют список команд, из которых пользователь может выбрать. Но если список будет слишком большим или будет вызываться при каждом жесте мышью, то смысл введения жестов пропадает, так как QReal уже предоставляет пользователю список объектов, которые с помощью операции drag-and-drop можно перетаскивать на рабочее поле.

4. Visual Paradigm.
   В Visual Paradigm мышиные жесты подразделяются на 3 типа — жесты, порождающие объект, вызывающие команду или создающие связь между объектами. Жесты, порождающие объект и вызывающие команды, задаются набором направлений (вверх, вниз, вправо, влево).

Мышиных жестов в Visual Paradigm гораздо больше, чем в вышеописанных утилитах. Но, как и в первых трех случаях, они жестко зашиты в код.

Создание списка идеальных жестов

Прежде чем заниматься алгоритмом распознавания, необходимо определить какой жест соответствует тому или иному объекту. Нужно создать список идеальных жестов. Идеальные жесты – эталон, с которым сравниваются нарисованные пользователем жесты. Каждому объекту соответствует максимум один идеальный жест. Было бы вполне естественно, если б жест визуально был похож на генерируемый объект. Идеальные жесты будем строить, исходя из этого соображения.

Для каждого объекта в QReal есть графическое представление, описание которого хранится в xml-файле. Описание представляет собой набор отрезков, каждый из которых определяется координатами начала и конца, и набор окружностей, которые определяются координатами диаметра. Окружность при равномерном движении мыши можно представить как правильный многоугольник. Для каждого объекта можно выделить несколько вершин (концы всех отрезков и точки на окружности) и ребер (отрезки), по которым строится граф, соответствующий объекту. А идеальный граф – эйлеров путь в этом графе, при условии что таковой имеется. Эйлеров путь — путь, проходящий по всем ребрам графа и притом только по одному разу. Если эйлеров путь не существует, добавим несколько ребер, чтобы получить эйлеров путь. Добавлять будем ребра, связывающие два подграфа, в которых уже существует эйлеров путь. Концы добавленного ребра должны совпадать с концами эйлеровых путей.

Получим некоторые точки на пути мыши при равномерном движении в предположении, что рука пользователя не дрожала и он изобразил ровно то, что хотел. Но чтобы получить путь мыши целиком, а не просто граф, нужно разбить длинные отрезки на более короткие. Исходя из предположения, что мышь двигалась равномерно, чем длиннее отрезок, тем больше на нем должно быть точек пути мыши. После разбиения получим идеальный путь (жест).

Для построения идеального пути по описанию графического представления объекта был создан инструмент, позволяющий генерировать идеальные жесты. После того, как жесты сгенерированы, их можно отредактировать: иногда эйлеров путь может получиться излишне сложным, особенно если изначально он не существовал, и пришлось добавлять новые ребра. Кроме того, графическое отображение не уникально для каждого объекта. То есть, могут существовать 2 различных объекта, для которых сгенерированы одинаковые жесты. Для этого предусмотрено «вращение» пути.

При вращении изменяется направление жеста, то есть если при генерации обход графа осуществлялся по часовой стрелке, то после вращения рисовать

жест надо будет против часовой стрелки. И наоборот.

Кроме того, для изменения идеального пути предусмотрен метод «увеличение». Увеличение необходимо, если изображение жеста, описанное в xml-файле, слишком мало. Иногда получается, что после создания идеального пути некоторые алгоритмы не смогут корректно его обработать, приняв за случайное дрожание руки. Это происходит из-за того, что графическое отображение объекта слишком мало и не проходит ограничения на минимальный размер.

Эти два преобразования встречаются чаще других, легко программируются, в то время как делать их вручную довольно долго.

Кроме того, жест можно отредактировать вручную. Отметим, что для редактирования предоставляется идеальный граф, а не путь.

После того, как идеальный путь мыши сгенерирован и отредактирован, его можно сравнивать с жестами пользователя.

Алгоритм распознавания жестов

Жесту соответствует путь мыши — список точек. Желательно иметь дело с более простыми объектами, чем списки точек. К примеру, при переносе траектории мыши должен генерироваться один и тот же объект, но координаты точек в списке сильно меняются. Для обработки этой ситуации можно переносить жест в начало координат, но тогда возникает проблема масштабирования — подобным жестам разного размера соответствует один и тот же объект. Можно приводить жест к заранее заданному размеру, но это может повлиять на распознавание объекта.

Таким образом, даже после переноса и масштабирования довольно сложно понять, на какой из идеальных жестов больше похож нарисованный путь. Некоторые существующие реализации предлагают следующее решение проблемы: сопоставим каждому жесту ключ-строку и сравнивать будем не пути, а строки. Сравнивать можно, находя расстояния Левенштейна между рабочим ключом (строкой, сгенерированной по жесту пользователя) и каждым из идеальных ключей. Идеальные ключи – строки, построенные по идеальным жестам. Найдем минимальное расстояние Левенштейна — найдем и соответствующий объект.

В конечном итоге задача разбивается на следующие этапы:

1) Определение пути мыши — создание списка точек.

   Программа получает сигнал о том, что кнопку мыши нажали или отпустили или мышь двигают, зажав кнопку. У сигнала есть метод, возвращающий координаты курсора мыши. Заносим эти координаты в список точек траектории мыши.

2) Фильтрация пути [4].

   На этом шаге сглаживается путь мыши. Этот шаг необходим, так как различное оборудование дает разную точность при получении позиции мыши. Кроме того, если учитывать все дрожания руки, придется усложнять следующий шаг – сопоставление объекта: каждому объекту будет соответствовать слишком много разных вариантов пути мыши.

3) Преобразование пути в строку.

   Списку точек сопоставляется строка. Существуют различные алгоритмы генерации строки по пути мыши, которые будут рассмотрены чуть позже.

4) Выбор объекта из списка идеальных ключей с помощью алгоритма Левенштейна.

   Расстояние Левенштейна (дистанция редактирования) между строками — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой для превращения одной строки в другую.

   Находим идеальный ключ, расстояние между которым и сгенерированным по нарисованному жесту наименьшее. Генерируем объект, соответствующий этому ключу, если ключ не сильно отличается от идеального.

Фильтрация пути

1. Получение текущих координат мыши
2. Вычисление параметров движения мыши
3. Получение предыдущих координат мыши
4. Вычисление исправленных координат мыши

Заметим, что если пользователь двигает мышь медленно, он хочет, чтобы путь указателя мыши на дисплее непосредственно соответствовал пути мыши. И наоборот, если скорость мыши достаточно велика, значит пользователю важнее конечная точка, в которой должен оказаться указатель, нежели пройденный путь. Таким образом, при быстром движении мыши полученный путь надо «сглаживать» сильнее, чем при медленном.
Путь мыши корректируется по следующей формуле:

$$\vec{corrected} = b * \vec{current} + (1-b) * \vec{previous}$$

Где $b = a * \exp(-c * s)$, $0 < a \le 1$, $c > 0$ - константы, s — скорость мыши

Требуется, чтобы после фильтрации по возможности сглаживались отрезки от дрожания руки, но при этом те углы, которые пользователь рисовал умышленно, должны остаться углами, а не переходить в дугу (квадрат, к примеру, должен остаться квадратом, а не преобразоваться в круг). Исходя из этих соображений константы c и a были выбраны эмпирически:

$c = 0.0275$; $a = 1$;

Скорость мыши вычисляется исходя из предположения, что оборудование считывает координаты равномерно:

$s = length(Points(i+1) - Points(i))/t;$

где $Points$ — список точек, $length$ — длина вектора, а t – константа времени, которая входит в подобранное c.

Обычно фильтрация проводится уже на уровне драйверов. Но для алгоритмов распознавания мышиных жестов требуется более точная фильтрация.

Генерация ключа по пути мыши

Желательно, чтобы подобным кривым соответствовали одинаковые ключи — размер жеста не должен кардинально влиять на полученную строку. Иначе придется масштабировать путь — приводить к заранее определенному размеру. В этом случае возникает проблема с кривыми, которые не подобны, но соответствуют одному и тому же объекту: при уменьшении фигуры важные звенья ломанной могут перейти в слишком маленькие, которые при фильтрации должны исчезнуть.

Пример масштабирования. По различным прямоугольникам программа должна вернуть один и тот же объект. Если все фигуры сводятся к размеру $n*n$, а мы рисуем прямоугольник $n*20\text{n}$, то при уменьшении фигуры получится $0.05\text{n}*n$. При таких параметрах нужно выбрать большое n, чтобы 0.05n как-то повлияло на построение ключа, иначе большинство алгоритмов разпознают меньшую сторону прямоугольника как случайное движение руки. В итоге жест сведется к двум разнонаправленным векторам. Если же n взять слишком большим, то после масштабирования случайное дрожание руки может перейти в длинные отрезки, что повлечет за собой неверную работу остатка алгоритма (распознавания жеста).

Можно выбрать алгоритм генерации ключа, чувствительный к малейшим изменениям движения и учитывающий короткие отрезки в пути мыши. Но тогда, несмотря на фильтрацию, одному и тому же объекту будет соответствовать очень большое число возможных жестов, некоторые из которых будут мало похожи друг на друга. А дальнейшая часть алгоритма предполагает, что жесты, генерирующие один и тот же объект, мало различаются.

Расстояние Левенштейна

Расстояние Левенштейна (дистанция редактирования) между строками — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой для превращения одной строки в другую. [6]

Расстояние Левенштейна обладает некоторыми недостатками:

1. При перестановке местами слов или частей слов получаются большие расстояния.

2. Расстояния между совершенно разными короткими словами оказываются небольшими, а расстояния между похожими длинными словами оказываются значительными.

Если первый недостаток критичен скорее при исправлении ошибок в слове и не важен в нашем случае, то из-за пункта 2 сложному жесту может быть сопоставлен неверный объект. Для того, чтобы избежать неверной интерпретации жеста, будем сравнивать не само расстояние, а соотношение: $d(S1, S2)/min(S1, S2)$ (*).

Найдем расстояние Левенштейна между ключом, сгенерированным по нарисованному жесту, и каждым из «идеальных» ключей в списке. Выберем «идеальный» ключ, которому соответсвует минимальное отношение (*). Выбранному идеальному ключу сопоставлен объект, причем единственный. Этот объект и надо генерировать.

Алгоритмы создания строки

1) Наиболее очевидный вариант — рассматривать направления как символы алфавита. То есть вводим 2^n направлений, каждому из которых соответствует угол на плоскости и свой символ в алфавите, путь мыши представляем как последовательность направлений (направление определяется принадлежностью одному из углов). Сначала составляем строку из символов, определенных направлениями векторов в пути мыши, потом удаляем повторяющиеся, подряд идущие символы, получаем конечную строку. Недостаток этого алгоритма состоит в том, что при недостаточной фильтрации легко перепутать объекты. Длины отрезков в пути мыши не учитываются, то есть даже небольшое дрожание руки, породившее короткий отрезок в неправильном направлении, может привести к ложному распознаванию жеста.

2) Можно не удалять одинаковые подряд идущие символы, но тогда жест придется масштабировать, иначе между подобными фигурами разного размера будет слишком большое расстояние. А как было показано выше, масштабирование может испортить фигуру.

3) Хотелось бы как-то учитывать длины отрезков, но только тех, которые были удалены окончательно (то есть удалена часть списка, где вектора относятся к одному направлению, при этом предыдущий вектор и

последующий принадлежат другим направлениям). Для этого можно ввести вес при нахождении расстояния Левенштейна — $dist$.

1. Каждому вектору на пути мыши сопоставим символ алфавита (начало алгоритма неизменно).

2. Введем список весов weights, длина которого $PointsNum-1$, где $PointsNum$ — количество точек в пути. То есть каждому вектору (а значит и символу) соответствует свой вес, изначально равный 1.

3. Запустим алгоритм Левенштейна. При удалении символа, у которого есть такой же соседний с индексом i, не будем увеличивать расстояние, но изменим список весов $weights[i]=weights[i]+1$ и удалим из него вес, соответствующий удаленному элементу.

4. При удалении i-ого символа, не равного ни одному из соседних, изменим расстояние: $dist=weights[i]+dist$. Соответствующий вес удалим.

5. При добавлении к нашей строке символа, равного одному из соседних, расстояние не увеличивается. Весу нового символа присваиваем 1.

6. Будем исходить из предположения, что пользователь не пропускает элементов (то есть векторов) жеста, и при добавлении нового элемента, не равного соседним, будем сильно увеличивать расстояние ($dist=k+dist$).

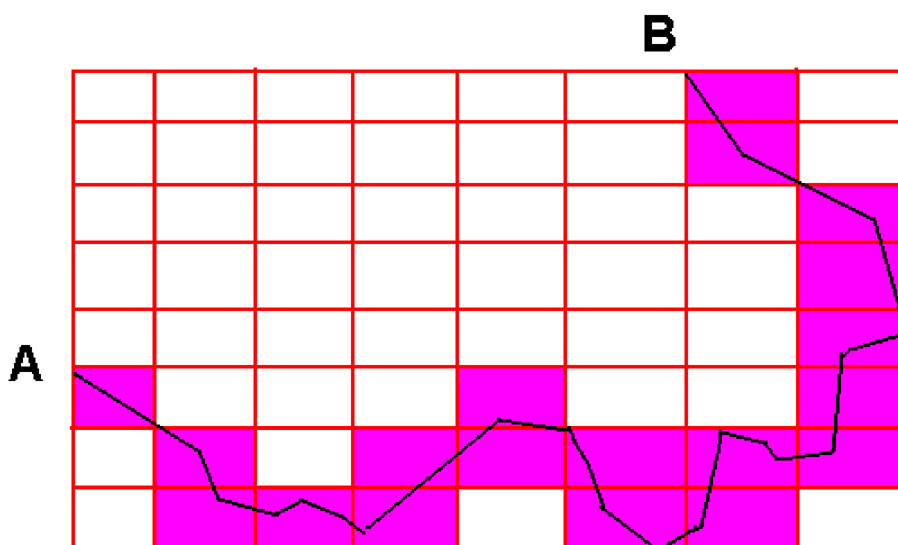
Если представлять жест в виде простейших отрезков, где никакие 2 соседние не параллельны , то сгенерируется объект, для которого нужно удалить отрезки минимальной суммарной длины. Во всех трех случаях можно ввести другие веса: чем дальше удаленное направление от соседнего, тем сильнее меняем расстояние.

4) Следующий алгоритм предлагает рассматривать не принадлежность направлению, а принадлежность прямоугольнику. Выделяется прямоугольник, целиком содержащий путь мыши (его стороны соответствуют границам жеста). Этот прямоугольник делится на некоторое количество более мелких равных между собой, в нашем случае на 64 прямоугольника. Каждая сторона делится горизонтальными и вертикальными прямыми на 8 частей (см. рис. 1). Каждому маленькому прямоугольнику соответствует символ алфавита. Строим строку по жесту, то есть добавляем символ, соответствующий

тому прямоугольнику, которому принадлежит точка пути. Удаляем подряд идущие одинаковые символы из строки, тем самым получаем конечный ключ.

Этот алгоритм хорош тем, что при его использовании не надо масштабировать жест. Подобным фигурам действительно соответствуют одни и те же ключи.

Рисунок 1



Алгоритм распознавания жестов без генерации строки-ключа

При создании QReal использовался инструментарий Qt, где есть реализация поддержки мышиных жестов. Рассмотрим алгоритм распознавания мышиных жестов, предложенный разработчиками Qt. [3]

Шаги алгоритма:

1) Фильтрация пути мыши.

2) Сопоставление объекта из списка. Предполагается, что объект в этом алгоритме — это ломаная (в нашем случае идеальный граф.)

   2.1) Приближение направлениями.
   При этом определяется, что именно пользователь имел в виду.

Необходимо приблизить каждый сегмент ломаной одним из заранее определенных базовых направлений. Как правило, их 4 – вверх, вниз, вправо, влево. К примеру, можно присвоить нулю меньшую из компонент (x, y) каждого вектора.

Для большей точности количество базовых направлений можно увеличить, при этом удобно брать число, кратное 4, чтобы 4 базовых направления оставались неизменными, и каждое было биссектрисой двух соседних. При увеличении числа направлений к жестам можно добавлять сложные многоугольники, стороны которых не обязательно параллельны осям координат. В ином случае бессмысленно генерировать идеальный жест с направлениями, отличными от 4 данных, так как алгоритм все равно никогда не распознает объект, соответствующий этому жесту.

2.2) Упрощение списка направлений.
Упрощение заключается в том, чтобы найти подряд идущие сонаправленные вектора и объединить их в один вектор, просуммировав длину.

2.3) Сопоставление и удаление.
Эта часть алгоритма является, пожалуй, наиболее сложной, так как несмотря на фильтрацию, останутся мелкие сбои вдоль длинных отрезков.
На этом шаге списку направлений сопоставляется команда. Если для списка нет команды, мы удаляем кратчайший отрезок и повторяем попытку. Алгоритм заканчивается, если удалось получить команду, или большая часть первоначального списка направлений была удалена.

Заметим, что этот алгоритм аналогичен алгоритму 3 из предыдущего пункта. Если в данном случае критерием является количество удаленных отрезков, то в алгоритме 3 критерием является их суммарная длина. Недостаток этого алгоритма состоит в том, что изменение оставшихся отрезков после удаления самого короткого довольно нетривиально. Чтобы путь не потерял связность, удалить отрезок, не изменив концы соседних, невозможно. Чтобы отрезки сохранили свое направление, придется менять не только соседние (на рисунке 2 показана часть пути мыши. Удаляем самый короткий отрезок 2, после этого переносим отрезок 3, тем самым меняем координаты начала отрезка 4. Путь преобразуется в траекторию, приведенную на рисунке 3). Причем, при увеличении числа направлений перестраивать путь будет все сложнее. Поэтому был выбран алгоритм с генерацией строки.

Рисунок 2

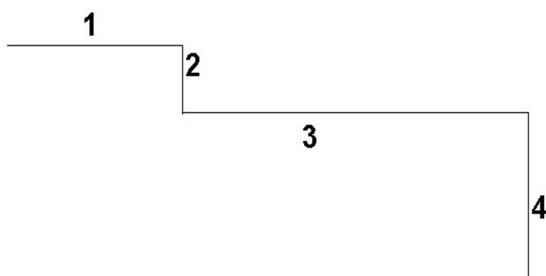
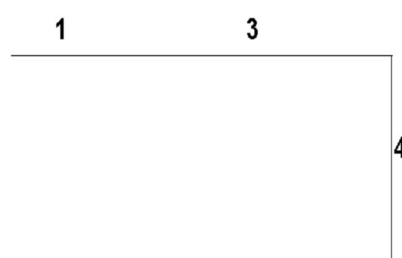


Рисунок 3



Связи

У некоторых объектов в QReal можно изменять ширину и высоту, но их нельзя поворачивать. Все предыдущие рассуждения, касающиеся генерации жеста, похожего на объект, и схожести строк, были применимы именно к таким объектам. Но помимо всего прочего, в QReal есть связи, графическое отображение которых представляет собой вектор в произвольном направлении, а иногда произвольную кривую. Поиск произвольного вектора, похожего на изображенный жест, методом генерации ключа или разбиением на направления затруднителен. Гораздо проще проверить заранее, не является ли требуемый объект связью. Это можно сделать, определив, принадлежат ли координаты конца и начала жеста каким-либо объектам. Если принадлежат, то наш жест — это связь между этими объектами. А если не принадлежат, то придется искать объект по списку идеальных путей.

Заключение

В результате работы был создан инструмент для распознавания мышиных жестов. Для этого по описанию графического представления объектов было сгенерировано описание идеальных жестов. Объект представлен в виде графа, в нем можно найти эйлеров путь, который при необходимости редактируется. Найденный эйлеров путь — идеальный жест для соответствующего объекта. Сгенерированные идеальные жесты можно добавить в xml-описания редакторов. Написана программа, визуализирующая созданные жесты: по списку точек изображается как должен выглядеть мышиный жест для соответствующего объекта.

Рассмотрено несколько алгоритмов распознавания жестов — сравнения нарисованного пользователем жеста и каждого из идеальных жестов. Выбран наиболее подходящий алгоритм — генерация ключа-строки, выбор ближайшего ключа из списка идеальных (ближайшего, в смысле расстояния Левенштейна). Ключ-строка генерируется с помощью алгоритма разбиения прямоугольника на более мелкие. В ходе работы над курсовой оказалось, что при использовании этого алгоритма жесты распознаются точнее, чем при использовании алгоритма разбиения пути по направлениям.

Для большей точности при работе алгоритма распознавания реализовано сглаживание траектории мыши. Эмпирически подобраны константы для алгоритма фильтрации.

При помощи созданного инструмента можно убедиться, что генерируемые жесты действительно распознаются посредством выбранного алгоритма.

Список литературы

1) Daniele Mancini, C# application to create and recognize mouse gestures (.Net) // The code project. URL. http://www.codeproject.com/KB/recipes/cmgblade.aspx (Дата обращения: 19.10.2009)

2) Didier Brun, Mouse gesture recognition // ByteArray.org Actionscript 3 Experiments. URL. http://www.bytearray.org/?p=91 (Дата обращения: 19.10.2009)

3) Johan Thelin, Recognizing mouse gestures // Qt online reference documentation. URL http://doc.trolltech.com/qq/qq18-mousegestures.html (Дата обращения: 21.10.2009)

4) Xuejiang Cheng, Self-adjusting digital filter for smoothing computer mouse movement // Freepatentsonline. URL http://www.freepatentsonline.com/5661502.pdf (Дата обращения: 14.02.2010)

5) Терехов А. Н., Брыксин Т. А., Литвинов Ю. В. и др., Архитектура среды визуального моделирования QReal, // Системное программирование, вып. 4, сб. статей / под ред. А. Н. Терехова, Д. Ю. Булычева, СПб., 2009, С. 171-196.

6) Расстояние Левенштейна // Википедия Свободная энциклопедия. URL. http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_%D0%9B%D0%B5%D0%B2%D0%B5%D0%BD%D1%88%D1%82%D0%B5%D0%B9%D0%BD%D0%B0 (Дата обращения: 14.02.2010)