ФАКУЛТЕТ ИНЖЕЊЕРСКИХ НАУКА

Универзитета у Крагујевцу

Назив студијског програма: **Машинско инжењерство**

Ниво студија: **Основне академске студије**

Модул: **Информатика у инжењерству**

Предмет: **Софтверски инжењеринг**

Број индекса: **715/2013**

ЂОРЂЕ С. РИСИМОВИЋ

ИЗРАДА АПЛИКАЦИЈЕ ЗА ПРЕПОЗНАВАЊЕ КОНТУРА И ВЕКТОРИЗАЦИЈУ

-ЗАВРШНИ РАД-

Комисија за преглед и одбрану:

1. др. Ненад Филиповић - ментор

2. _____ **Датум одбране:_____**

3. _____ **Оцена: _____**

Крагујевац, октобар 2014

У оквиру овог завршног рада кандидат треба да опише основне појмове из области препознавања контура објеката, њиховог детектовања и исцртавања у програмском језику C#. Практичан део рада односи се на креирање апликације која ће препознати контуру изабраног објекта, а затим исту конвертовати у вектор. На самом крају рада извршиће се учитавање конвертоване контуре и упоређивање њених карактеристика са оргиналном сликом.

Препоручена литература:

[1] Ненад Филиповић (2006), Објектно оријентисано програмирање, ФТН Чачак.

[2] Lindeberg (1998) "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision.

[3] H.G. Barrow and J.M. Tenenbaum (1981) "Interpreting line drawings as three-dimensional surfaces", Artificial Intelligence.

Крагујевац, 23. 10.2014 ментор:

Др Ненад Филиповић, ред.проф.

_____

Резиме:

Рад је настао као резултат интресовања за области рачунарске графике и објектно оријентисаног програмирања. Приказане су теоријске основе које описују основне појмове у обради слике и одређивању контура. Креирана је апликација која врши обраду података са слике бит по бит, препознаје граничне битове објекта на слици и од њих израђује контуру. У даљем току рада описан је поступак конверзије битмапиране контуре у вектор, да би на самом крају приказали карактеристике добијеног вектора.

САДРЖАЈ

1 УВОД

Завршни рад „ Израда апликације за препознавање контура и векторизацију“ настао је као резултат интресовања за области рачунарске графике и објектно оријентисаног програмирања. Упознавајући се са овим областима дошао сам до многобројних сазнања о особинама програмског језика *C#*, као и о могућностима практичне примене апликација које се у њему израђују. Кроз завршни рад биће изложен значај конвертовања битмапе у вектор, и њена практична примена.

Овај завршни рад се састоји из шест поглавља:

- У првом поглављу изложене су уводне напомене, дат је увод у тему завршног рада и објашњени су разлози његове израде.

- У другом поглављу изложени су основни појмови из области одређивања контура објеката, сврха одређивања контура као и основне методе.

- Треће поглавље пружа нам увид у могућности програмског језика *C#* у области проналажења и исцртавања контура, и упознаје нас са основама програмирања.

- Четврто поглавље приказује израду апликације корак по корак, уз објашњења о начину функционисања програма, као и о перформансама алгоритма.

- У петом поглављу направљена је паралела између полазне слике и контуре која је настала као резултат израде апликације.

- Шесто поглавље је закључак овог завршног рада, где су сумирана достигнућа овог рада.

2 ОСНОВНИ ПОЈМОВИ У ОДРЕЂИВАЊУ КОНТУРА ОБЈЕКАТА

2.1 Обрада слике

Када говоримо о науци сликања, обрада слике је било који облик обраде сигнала који је као улаз користио слику, као што је фотографија или видео. Излаз обраде слике могу бити слика, или скуп карактеристика или параметара који се односе на слику. Већина техника за обраду слике укључује третирање слике као дводимензионалног сигнала, чиме се примењују стандардне технике обраде.

Обрада слике обично се односи на дигиталну обраду, али оптичка и аналогна обрада слике такође су могуће. У блиској вези са обрадом слике су рачунарска графика и рачунарска визија. У компјутерској графици, слике су ручно израђене од физичких модела објеката, окружења, и осветљења, уместо да имају стечене особине (преко уређаја за снимање, као што су камере) од природних сцена, као и у већини анимираних филмова. *Computer vision*, са друге стране, често се сматра за обраду слика на високом нивоу код којих машина / софтвер дешифрује физичке садржај слике или низа слика (нпр. , видео или *3D full-body* магнетном резонанцом скенирања).

У савременим наукама и технологијама, слике добијају много шире домете због све већег значаја научне визуализације (неретко великих комплексних научних података). Примери укључују *microarray* податке у генетском истраживању, или *real-time multi-asset* профиле у финансијском пословању.

2.2 Одређивање контура

Одређивање контура је назив за скуп математичких метода које имају за циљ идентификовање тачака у дигиталним сликама на којој се осветљеност слике оштро мења или има дисконтинуитета. Тачке у којима се осветљеност оштро мења су обично организоване у скуп заобљених линија које најчешће називамо контуре (ивице). Исти проблем проналажења дисконтинуитета у 1*D* сигналу познат је као детекција корака. Проблем проналажења сигналног дисконтинуитета познат је као детекција промена.

Одређивање контура је основно средство у обради слике, машинској и компјутерској визији, посебно у областима детектовања и екстракције особина.[1]

Контура је скуп пиксела, чија се околина у разним нијансама сиве убрзано мења. Унутрашње карактеристике подручја чија се контура одређује су исте, док различите области имају различите карактеристике. Ивица је основна карактеристика слике. Постоји много информација о самој слици које можемо пронаћи на основу контуре. Циљ детекције ивица је да издвоји карактеристике дискретних делова на основу разлике карактеристика објекта на слици, а потом да се утврди подручје слике према затвореној контури. Детекција ивица је у широкој употреби у компјутерској графици, анализи слика, итд.

Одређивање контура се односи на процес идентификације и лоцирања оштрих дисконтинуитета на слици. Дисконтинуитети су нагле промене у интензитету пиксела које карактеришу границе објеката у сцени. Класичние методе одређивања контура укључују обраду слике са оператором (*2-D filter*), који је конструисан да буде осетљив на велике градијенте на слици, док би враћао вредности нуле у једноличним регионима. Постоји изузетно велики број доступних оператора, сваки дизајниран да буде осетљива на одређену врсту контуре. Променљиве укључене у избору оператора детекције контура укључују оријентација контуре, неуравнотеженост окружења и структуру контуре. Геометрија оператора одређује карактеристични правац на основу кога ће се контура одређивати.

2.2.1 Сврха одређивања контура

Сврха откривања оштрих промена у осветљености слике јесте заузимање важних догађаја и промена у особинама света. Може се рећи да под прилично општим претпоставкама за слику формације модела, дисконтинуитет у осветљености слике вероватно одговара: [2]

- прекиду у дубини,
- прекиду у површинским оријентацијама,
- променама у материјалним особинама и
- варијацијама у осветљењу сцене.

---

[1] Umbaugh, Scott E (2010). Digital image processing and analysis : human and computer vision applications with CVIPtools (2nd ed. ed.). Boca Raton, FL: CRC Press

[2] H.G. Barrow and J.M. Tenenbaum (1981) "Interpreting line drawings as three-dimensional surfaces", Artificial Intelligence, vol 17, issues 1-3, pages 75-116.

У идеалном случају, резултат примене детекције контура на слику може да доведе до низа повезаних кривих које указују на границе објеката, границе површинских ознака, као и криве које одговарају дисконтинуитету у површинској оријентацији. Такође, примењујући алгоритам за детекцију ивица на слици може се значајно смањити количина података који се обрађују, чиме се филтрирају информације које се могу посматрати као мање релевантне, уз очување важних структурних својства слике. Ако је процес детекције ивица успешан, накнадно тумачење информација садржаја у оригиналној слици може бити знатно олакшано. Међутим, није увек могуће добити идеалне контуре из снимака реалног живота који поседују одређену сложеност.

Контуре израђене на основу сложених слика често су ограничене фрагментацијом, што значи да контурне криве нису повезане. Разлог за то су недостатак сегмената контуре, као и лажне ивице које не одговарају стварним појавама на слици - чиме компликују касније тумачења података са слике.[3]

Детекција контура један је од основних појмова у обради слике, анализи слике, препознавању облика као и у визуелним компјутерским техникама.

2.2.2 Особине слике

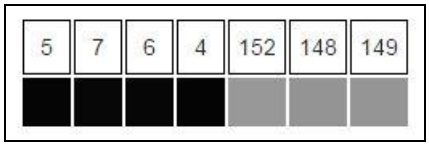
Контуре израђене на основу дводимензионалне слике на тродимензионалној сцени могу се класификовати као зависна гледишта или независна гледишта. Контура независног гледишта обично одражава особине тродимензионалних објеката, као што су површинске ознаке и површинские облици. Контуре зависног гледишта могу се променити како се гледиште мења, а обично одражава геометрију сцене, као што су предмети који затварају један другог.

Типична контура може на пример да буде граница између блока црвене и блока жуте боје. Насупрот томе линија може да буде мали број пиксела различите боје на иначе непроменљивој позадини. Стога, линуију може чинити обично једна ивица на свакој страни линије.

1[3] Lindeberg (1998) "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision, 30, 2, pages 117--154.

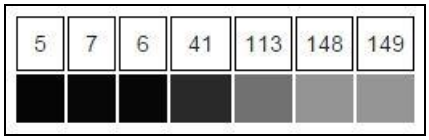
2.2.3 Зашто одређивање контура није једноставан посао?

Како би илустровно приказали зашто откривање контура није тривијалан задатак, размотрићемо проблем откривања контуре у наредном једнодимензионалном сигналу. Овде се интуитивно може рећи да контура треба да постоји између 4-ог и 5-ог пиксела (слика 1).



Слика 1. Једноставан једнодимензионални сигнал

Ако је разлика у интензитеу мања, и ако би интензитет разлике између суседних пиксела био већи, тада не би било лако рећи да ли би требало да постоји контура у одговарајућем региону. Штавише, могло би се рећи да је ово случај у коме постоји неколико ивица (слика 2).



Слика 2. Сигнал са малом разликом у интезитету између пиксела

Дакле, чврсто одердити конкретан праг као велику промену интензитета између два суседна пиксела и нагласити да између њих треба да постоји контура није увек једноставан посао[4]. Заиста, ово је један од разлога зашто детекција контуре може бити озбиљан проблем, осим ако су објекти у сцени посебно једноставни и услови осветљења могу бити добро контролисани.

2.3 Методе одређивања контура

Постоји много начина за детекцију контура, али већина њих се може груписати у две категорије, на основу претраживања базе и на основу проналажења нула у бази. Методе на основу претраге базе детектују контуре прво израчунавајући јачину ивице, обично користећи први извод величине градијена, а затим у потрази за смером крећу ка максимуму градијента

[4] T. Lindeberg (1998) "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision, 30, 2, pages 117--154.

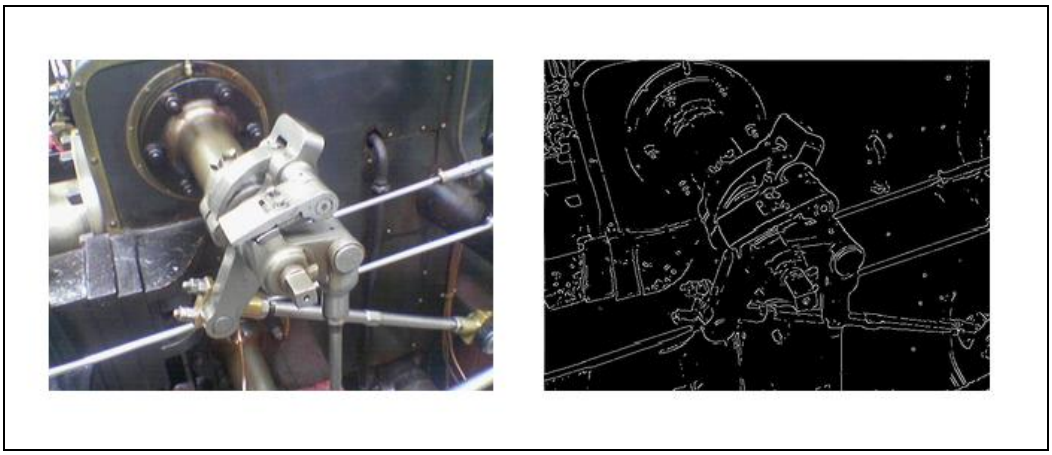
користећи израчунату локалну оријентацију контуре, обично правацем градијента. Методе засноване на проналажења нула у бази памте прелазе преко нула како би се пронашли ивице, обично користећи *Laplacian* или неки други нелинеарни диференцијални израз. Као предкорак у детекцији контура *Gaussian*-ов метод посветљивања скоро се увек примењује.[5]

2.3.1 Метода *John F. Canny* - ја

Алгоритам *John F. Canny*-ја познат је као оптимално решење за детектовање контура. *Canny*-јеве намере су биле да побољша програме за детекцију у време када је започињао свој рад. Био је веома успешан у остваривању свог циља. У свом раду, он је навео списак критеријума за побољшање тренутних метода детекције контура:

1. Први и најочигледнији критеријум је ниска стопа грешке. Важно је да се ивице које се налазе на слици не пропуштају, а да не буду обухваћени делови слике који не представљају контуру.
2. Други критеријум је да ивичне тачке буду добро локализоване. Другим речима, растојање између ивице пиксела које је пронађено од стране детектора и стварне ивице треба да буде сведено на минимум.
3. Трећи критеријум је да једна стварна ивица не треба да доведе до појаве више од једне контуре.

Иако је ова метода прилично стара, развијена 1986. год, постала је једна од стандардних метода детектовања контура и још увек се користи у истраживању (слика 3).

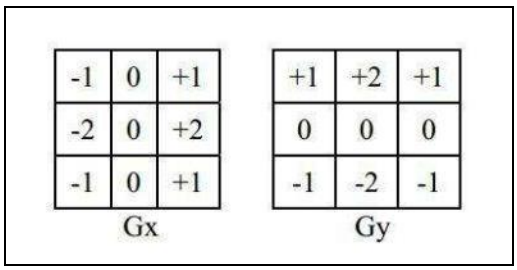


Слика 3. *Canny* метода одређивања контура примењена на фотографији парне машине у боји

[5] J. M. Park and Y. Lu (2008) "Edge detection in grayscale, color, and range images", in B. W. Wah (editor) Encyclopedia of Computer Science and Engineering, doi 10.1002/9780470050118.ecse603

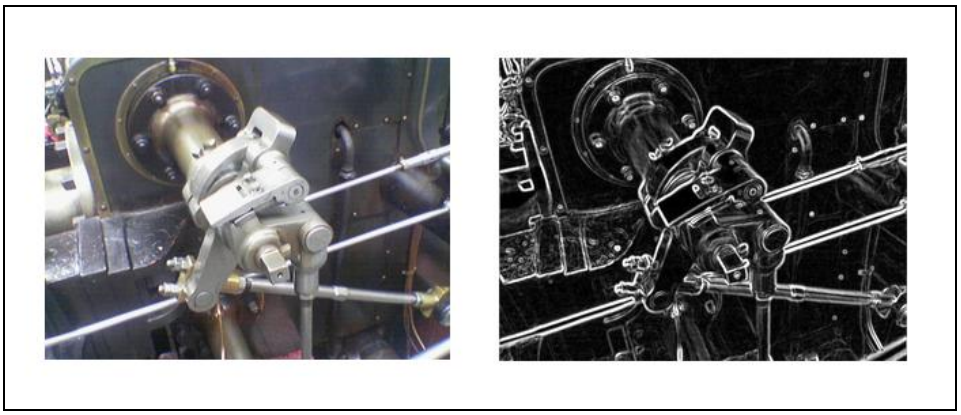
2.3.2 *Sobel* operator

Sobel оператор, понекад назива *Sobel* филтер, користи се у обради слике и компјутерској визији, посебно у области откривања контура, и стварању слике која наглашава ивице и прелазе. Овај алгоритам назван је по *Irwin Sobel*-у, који је представио идеју о "изотропном 3x3 градијентном оператору" у говору на *Stanford Artificial Intelligence Project*-у 1968.год.[6] Оператор користи две 3x3 матрице: једна одређује градијент у Gx смеру , док друга одређује градијент у Gy правцу (слика 4).



Слика 4. Матрица коју користи *Sobel* оператор

Технички, то је изолован диференцијални оператор, који прорачунава апроксимацију градијента према функцији интензитета слике (слика 5). У свакој тачки на слици, резултат *Sobel* оператора је или одговарајући вектор градијента или нормала овог вектора. *Sobel* оператор је заснован на умотавању слике са малим, одвојивим, целобројним филтером у хоризонталном и вертикалном правцу, зато је релативно једноставан у погледу прорачуна. С друге стране, производ градијенталне апроксимације је релативно груб, нарочито у високофреквентним варијацијама слике.



Слика 5. *Sobel* оператор примењена на фотографији парне машине у боји

[6] Irwin Sobel, 2014, History and Definition of the Sobel Operator

3 МОГУЋНОСТИ C# - а У ОБЛАСТИ ДЕТЕКЦИЈЕ И ИСЦРТАВАЊА КОНТУРА

Windows Forms интерфејси су засновани на концепту контрола које, између осталог, знају како да саме себе исцртају. Ако ваш интерфејс захтева цртање које је изван могућности контрола које су вам на располагању, биће потребно окренути се *.NET GDI namespaces*-у.

GDI (Graphical device Interface) пружа низ шаблона, координата и мерних алата крећући од једноставних који могу исцртати линије, до комплексних који могу испунити градијент. Предност ових особина је да се практично свака врста интерфејса може креирати коришћењем *GDI*-ја. Недостатак је што *GDI* нема сопствену меморију и потребно је писати код који је способан да омогући позивање целог *GDI* интерфејса у било које време. Добар део рада у *GDI*-ју такође ће укључивати писање прилагођеног *input* кода. Количина детаља који су укључени у руковање поновним исцртавањем и уносом значи да се мора посветити још више пажње у односу на логику из домена кода интерфејса. *C#* код комуницира са *GDI* помоћу објеката типа *Graphics*. Основни принцип је :

- Објектна промењљива је креирана да садржи референцу ка *Graphics* објекту.
- Објектна промењљива је постављена у валидан *Graphics* објекат.
- Да би цртали или приказали позивамо методе *Graphics* објекта.

GDI је библиотека која обезбеђује интерфејс за писање *Windows* и *Web* графичких апликација које су повезане са различитим графичким уређајима као што су принтери, монитори или фајлови. Реализована је као скуп *C++* класа смештених у библиотеци класа *Gdiplus. dll* која је уграђена у *Windows* 2007.

3.1 Исцртавање површина

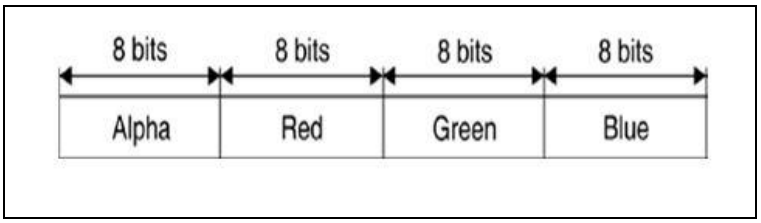
Свака апликација за цртање, без обзира на *OS*, састоји се из три компоненте:

- *canvas*: површина на којој се објекти исцртавају (површина за цртање),
- *brush* или *pen* објекти : представљају боју, текстуру и ширину објекта који се исцртава,
- *process* : описује како се објекат исцртава.

Свака површина за цртање има следеће карактеристике:

- *width* (ширину),
- *height* (висину),
- *resolution* (резолуцију изражену у *dotc per inch*),
- *color depth* (дубина боје – представља број бита искоришћених за репрезентацију сваког пиксела).

У *GDI* боја је представљена *Color* структуром која има елементе приказане на слици 6.



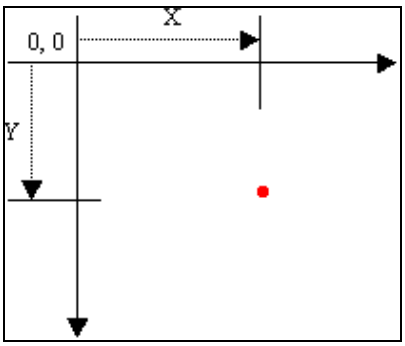
Слика 6. Елементи *Color* структуре

На овај начин је са 32 бита у потпуности предсатављен сваки пиксел, али да би видели правилно дефинисане боје у *GDI Color* стурктури, површина за цртање мора подржавати најмање 24-битни колор систем тј. за сваку *RGB* компоненту по 8 бита.

GDI подржава три типа површина за цртање:

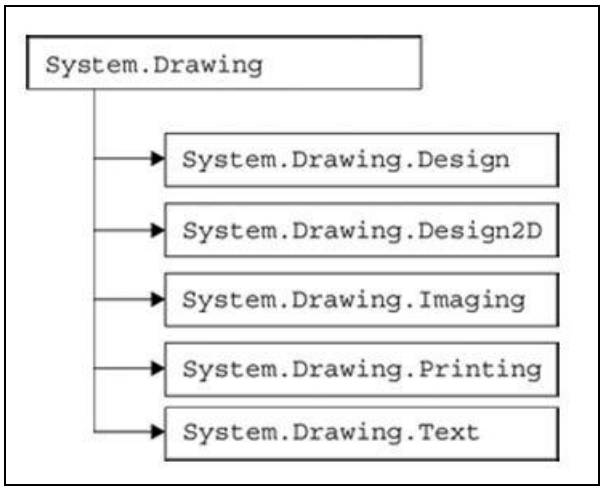
- форме као површине,
- принтери као површине,
- битмапе као површине.

GDI координатни систем приказан је на слици 7.



Слика 7. *GDI* координатни систем

Унутар *.NET Framework*-а , *GDI* класе су изложене у *nemespace System.Drawing* и његовеим под *nemespace*-овима. Ови именски простори уствари обезбјеђују класе којима можемо руковати а које су заправо класе *C# GDI* класа (слика 8).



Слика 8. *C# GDI* класа

Овај *namespace* обезбеђује *GDI* функционалност. Садржи неке од основних класа и структура које су коришћене у практичној реализацији завршног рада (табела 1).

Табела 1. *System.Drawing* основне класе које су коришћене за реализацију апликације

Класа	**Опис**
`Bitmap`	Енкапсулира битмапу, која представља слику (са својим особинама) сачувану у формату пиксела.
`Graphics`	Кључна класа које обухвата цртање површина. Међу многим другим особинама, графичка класа омогућава цртање и попуњавање графичких објеката.
`Image`	Пружа члановима могућност дефинисања величине, висине, ширине и формата слике. Класа *Image* такође пружа методе за стварање објеката слике из датотеке или *stream*-а, омогућава чување, ротирање и окретање слике. То је апстрактна основна класа, а њена функционалност се користи преко својих изведених класа: *Bitmap, Icon* и *Metafile*.
`Region`	Представља регион у *GDI*, који описује унутрашњост графичког облика.

Основна функционалност за рад са сликама одређена је класама из простора *System.Drawing*, док су напредне опције за рад са сликама одређене класама из простора *System.Drawing.Imaging*.

Да би апликација исправно радила тј. могла проступити класама из ових простора, ове именске просторе неопходно је импортовати у апликацију наредбом типа:

```
using System.Drawing;
```

3.2 Додавање објеката типа *Graphics*

У *Windows* апликацијама површина за цртање је форма. Свака форма има придружен *Graphics* објекат који обезбјеђује функције цртања.

Када год апликација треба нешто да нацрта она то реализује кроз овај *Graphics* објекат. У овом завршном раду, објекат типа *Graphics* преузима вредности из *pictureBox2*:

```
Graphics g = Graphics.FromImage(pictureBox2.Image);
```

Важно је напоменути следеће да када креирамо *Graphics* објекат коришћењем методе *CreateGraphics* морамо експлицитно позвати методу *Dispose()* како би ослободили ресурсе придружене објекту.

3.3 Структура *Color*

Color структура представља *GDI ARGB(alpha-red-green-blue*) боју. Ова структура имала је значајну улогу у програмском решењу практичног дела овог завршног рада. Због тога ћемо навести њене чланове (табела 2)[7]. У наредној линији кода приликом конвертовања објекта из боје у *Grayscale* можемо приметити да се нијанса сиве коју добијамо након конверзије узима из *Color* структуре *GDI ARGB*:

```
c = Color.FromArgb(r, g, b);
```

[7] Larry O'Brien and Bruce Eckel (2005). Thinking in C#, Prentice Hall Upper Saddle River, New Jersey

Tabela 2. *Color properties*

Property	**Опис**
`Red`, `Blue`, `Green`, `Aqua`, `Azure`, itd.	Прецизиран *color static property* за готово сваку боју.
`A`	Враћа алфа вредност компонената у структури *Color*.
`R`	Враћа вредност црвене компоненте у структури *Color*.
`G`	Враћа вредност зелене компоненте у структури *Color*.
`B`	Враћа вредност плаве компоненте у структури *Color*.
`IsEmpty`	Показује да ли је структура боје неиницијализована.
`IsKnownColor`	Показује да ли је боја задата.
`IsNamedColor`	Показује да ли је боја задата.
`IsSystemColor`	Показује да ли је боја системска.
`Name`	Враћа име боје.

3.4 Битмапе

GDI има могућност за приказ слике директно из фајла или за учитавање у *PictureBox*. Да би се приказала слика коју већ имамо у меморији, користићемо *Graphics.DrawImage( )*, који нам омогућава да прикажемо слику или део исте у оригиналној величини. На слици 9. приказан је један од начина учитавања слике:

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        Bitmap img = new Bitmap(pictureBox1.Image);
        Color c;
```

Слика 9. Учитавање битмапиране слике у *PictureBox*

4 ИЗРАДА АПЛИКАЦИЈЕ ЗА ПРЕПОЗНАВАЊЕ КОНТУРА И ВЕКТОРИЗАЦИЈУ

Пре почетка израде саме апликације потребно је замислити њен изглед и начин на који ће функционисати. Како би дошли до решења задатка, односно до креирања апликације која ће учитати једноставну битмапирану слику, пронаћи њене контуре, а затим те контуре конвертовати у вектор биће нам потребна само једна форма која ће у себи садржати две контроле типа *PictureBox*, и пет контрола тип *Button*. Замисао је да се кликом на прву команду типа *Button* у први *PictureBox* учита битмапирана слика. Кликом на друго командно дугме једноставан облик који се налази на слици потребно је конвертовати у црну боју док ће позадина бити бела, на тај начин приликом проналажења контуре програм може препознати разлику између црне и беле боје. Након што смо извршили конверзију објекта у црну боју потребно је извршити пролаз преко сваког бита по *X* и *Y* оси, када се дође до првог црног бита на објекту потребно је формирати матреицу која ће установити да се тај бити налази на ивици објекта и означити га као контуру. После процеса одређивања контуре приступићемо векторизацији. У компјутерској графици, вецторизација је процес конверзије растерске графике у векторску графику. На самом крају потребно је конвертовану контуру сачувати у неком од векторских формата.

4.1 Креирање апликације у програму *Visual C#*

Visual Studio је комплетан скуп развојних алата за изградњу *ASP.NET* (енг. *Active Server Pages. NET*) *Web* апликација, *XML Web* сервиса, десктоп апликација и мобилних апликација. Програмски језци укључени у *Visual* студио су: *Visual Basic, Visual C#* и *Visual C++*. Они користе исто интегрисано развојно окружење (*IDE*), алат који омогућава дељење и олакшава стварање мешовитих језичких решења. *Microsoft* обезбеђује бесплатно преузимање *„Express“* издања свог *Visual Studi*-а 2010 са својим компонентама.

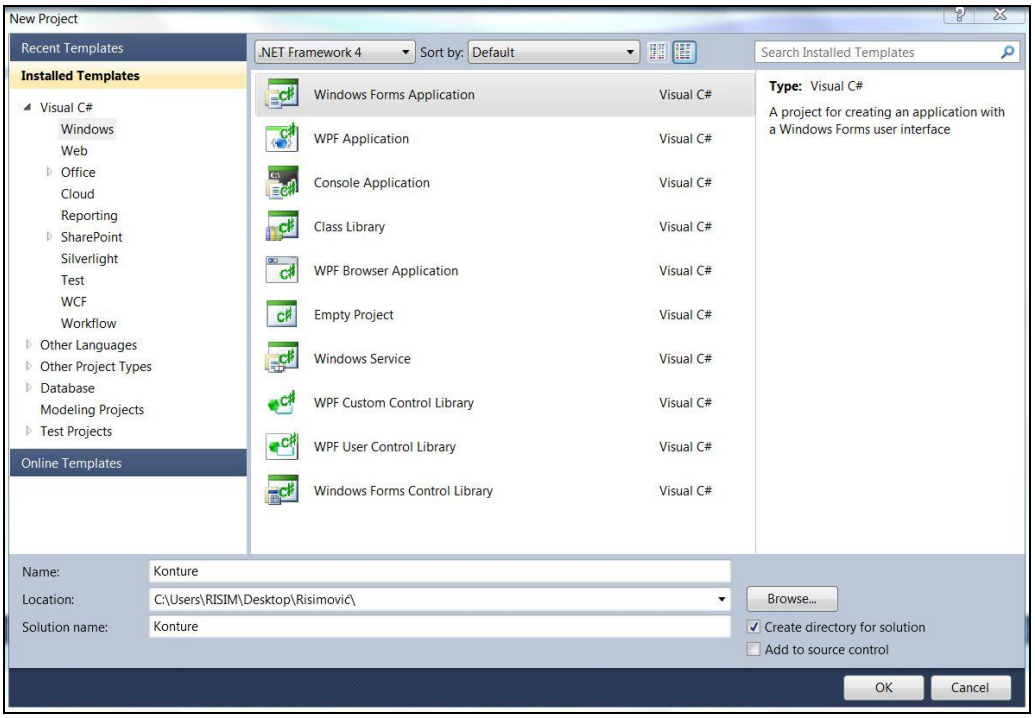
Да бисмо направили нову конзолну апликацију потребно је урадити следеће:

1. У мени ј „*File*“ потребно је изабрати *New Project*, отвара се оквир за дијалог New project (слика 10). Овај оквир за дијалог излистава шаблоне које можемо користити као полазну основу за прављење апликације.

2. У *Project types* окну, потребно је притиснути *Visual C#*. У окну *Templates*, изабраћемо икону *Windows Form Aplication*.

3. У поље Name потребно је уписати назив пројекта Контуре а затим потврдити на ОК.

4. Да би променили препоручену локацију пројекта потребно је одабрати *Tools* на менију, и селектовати *Options*. Затим је потребно изабрати *Projects and Solutions* и селектовати опцију *General*. У *Project location text box*, можемо унети жељену локацију за фајлове и пројекте (слика 10).

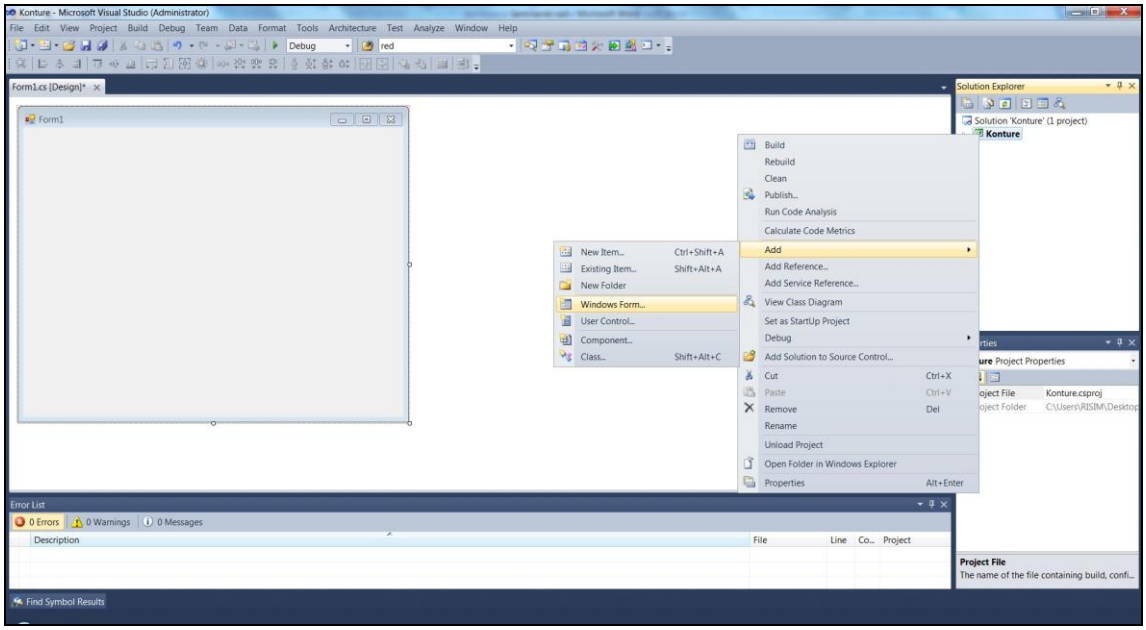


Слика 10. Креирање нове апликације

4.2 Додавање форме

Како нам је за реализацију задатка потребна само једна форма, утолико је и сам процес реализације једноставнији. Како у нову форму можемо сместити све контроле и остварити функционалност апликације, приликом употребе биће знатно допадљивија кориснику.

Нову форму можемо додати десним кликом на пројекат из *Solution Explorer*-а и одабиром опције *Add* (слика 11) , *Windows Form...* или када кликнемо на *Project, Add Windows Form...* тада ће нам се приказати прозор *Add New Item* – име пројекта, и једино што овде треба урадити јесте да се уверимо да је за категорију изабрано *Common Items* а од тема да је изабрано *Windows Form*. Када проверимо да је све уредно, такође је неопходно променити име (напомена: екстензија .cs мора остати на крају имена). Нову форму ћемо креирати кликом на дугме *Add*, чеме је направљена нова форма.



Слика 11. Додавање нове форме

4.3 Учитавање слике

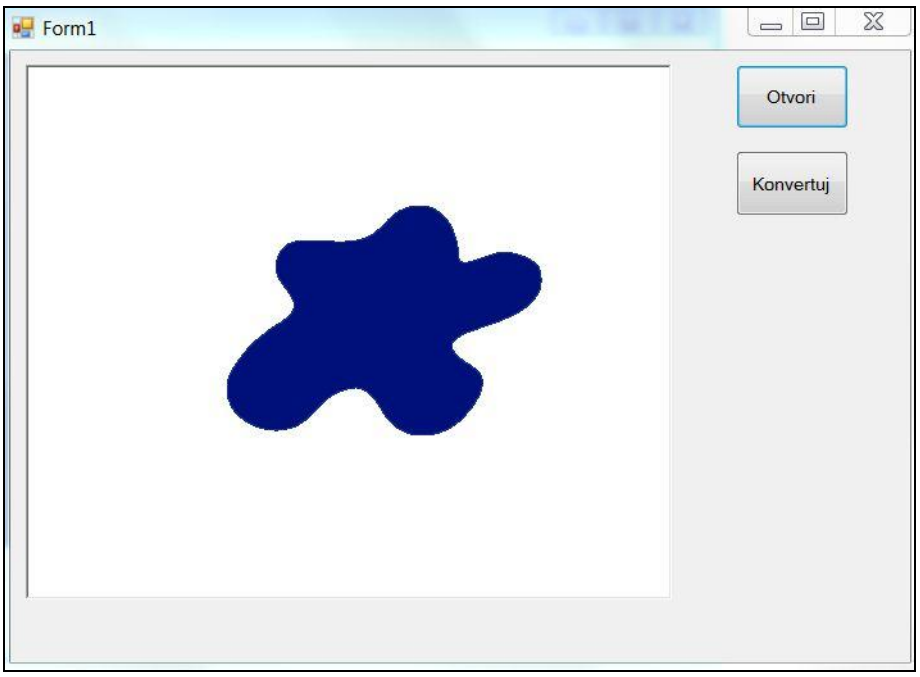
На самом почетку дизајнирања апликације додаћемо једну контролу *PictureBox* и једну контролу типа *Button.* За сада нам је циљ само да прикажемо слику на екрану. Са леве стране екрана из менија *Toolbox* изабраћемо жељене команде и превући их на површину форме. Пошто смо поставили команде на форму морамо им доделити функционалност, односно написати код на основу кога ће оне радити.

Двоструким кликом на командно дугме прелазимо у компајлер форме и то директно на део кода који се односи на командно дугме. Пошто још увек нисмо задали никакву функцију овој контроли њено тело је празно. Приступамо реализацији кода за ову команду, уносећи неопходне функције и промењљиве (слика 12).

```
private void button1_Click(object sender, EventArgs e)
{
        //Otvaramo dialog box
        OpenFileDialog openFileDialog = new OpenFileDialog();
        //Lokacija koja će se prikazati u dialog box-u
        openFileDialog.InitialDirectory = "c: Desktop\\";
        // Podešavamo filter,koji tip slika će moći da se prikaže u picturebox-u
        openFileDialog.Filter = "Bitmap files (*.bmp)|*.bmp|Jpeg files (*.jpg)|*.jpg|All
        valid files (*.bmp/*.jpg)|*.bmp/*.jpg";
        // Kada kliknemo na dugme OK ono će izvršiti ceo kod koji se nalazi ispod
        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
        // U PictureBox će se implementirati slika koju smo izabrali
        pictureBox1.ImageLocation = openFileDialog.FileName;
        }
}
```

Слика 12. Креирање кода за контролу *button1*

Првим линијама кода означили смо да кликом на командно дугме желимо да се отвори нови *dialog box*. За извршење овог задатка користили смо системску класу *OpenFileDialog* која допушта кориснику да отвори фајл. У следећој линији кода одредили смо локацију која ће се аутоматски приказати приликом отварања *dialog box*-а, у нашем случају то је *Desktop*. Затим одређујемо формат слика које ће нам бити видљиве у *dialog box*-у. Као дефолтну вредност поставићемо за нас битан *.bmp* формат, али оставићемо могућност приказа и *.jpg* формата. Битна ставка овог дела кода је поставка услова који ће уколико је изабрана слика исту приказати у *PictureBox*-у. Како би испитали да ли код врши своју функцију кликнућемо на дугме *Start debugging* на палети менија или на *F5* са тастатуре. Резултат је приказан на слици 13.



Слика 12. Приказ слике у програму

4.4 Конвертовање боја на слици у црну и белу

Како би могли одредити контуре на фотографији неопходно је све боје конвертовати у црну и белу, односно зависно од интезитета у различите нијансе сиве (*Grayscale*). Заправо, само облици без преливања боја у тамним тоновима приликом претварања имаће црну боју, док ће светлији тонови бити конвертовани у различите нијансе сиве.

Ову конверзију је неопходно извршити зато што компајлер познаје само 0 и 1, тако да белој боји можемо доделити вредност нуле, док ћемо црној доделити вредност јединице. На тај начин програм ће знати да направи разлику између објекта и позадине, и исцрта ивицу.

Метод који ћемо користит је најједноставнији за разумевање и имплементацију. На жалост, веома је спор (слика 13).

```
private void button2_Click(object sender, EventArgs e)
    {
       //izrada prazne bitmape iste veličine kao orginal
      Bitmap Image = new Bitmap(pictureBox1.Image);

       for (int i = 0; i < Image.Width; i++)
        {
          for (int x = 0; x < Image.Height; x++)
          {
          //uzimanje piksela iz orginalne slike
          Color oc = Image.GetPixel(i, x);
          // kreiranje grayscale verzije piksela
          int grayScale = (int)((oc.R * 0.3) + (oc.G * 0.59) + (oc.B * 0.11));
          //kreiranje objekta boja
          Color nc = Color.FromArgb(oc.A, grayScale, grayScale, grayScale);
          // podesavanje novih piksela na slici prema različitim nijansama sive
          Image.SetPixel(i, x, nc);
          }
        }
      pictureBox1.Image = Image;
    }
```

Слика 13. Код за конвертовање слике у *Grayscale*

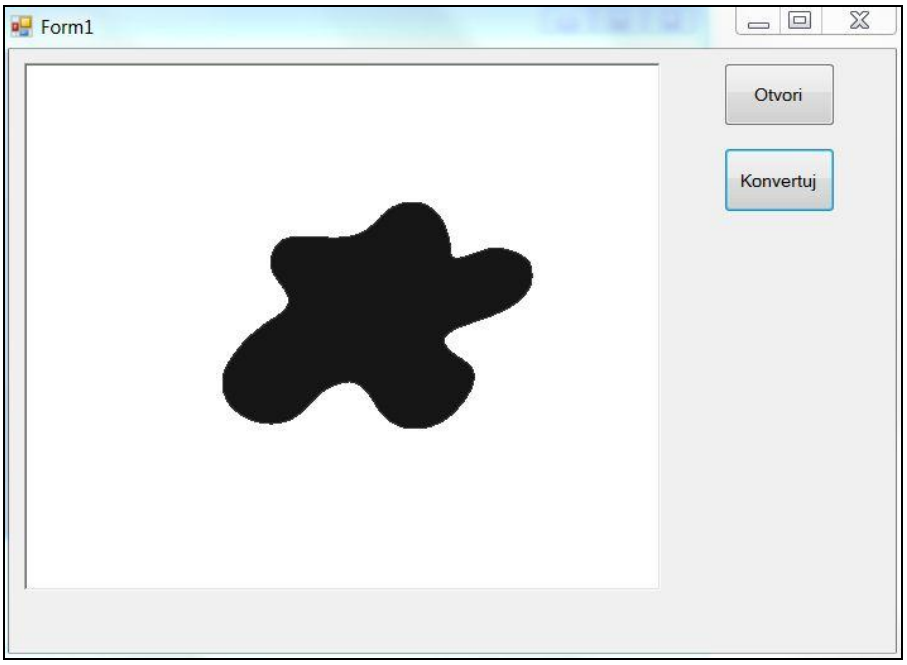
Приликом креирања *grayscale* верзије пиксела у коду смо користили бројеве 0.3, 0.59, и 0.11. У стварности, просечну боју смо могли добити сабирањем *R, G, B* и дељењем са три. На тај начин добили би прилично добру црно-белу слику. Међутим, у једном тренутку научници су схватили да ови бројеви више пријају осетљивости људског ока за сваку од ових боја.

4.4.1 Перформансе алгоритма

Ова метода ће радити одлично за мале слике или када стварно није битно колико је времена потребно да се слика конвертујете у црно и бело. Ако је потребно да се конверзија брзо уради, онда је боље потражити друге методе.

Карактеристика овог алгоритма је да проверава сваки пиксел у оригиналној слици и поставља исти пиксел у новој *grayscale* верзији битмапе. Неко се може запитати зашто је овај процес спор? Ако је слика 2048 x 2048, овај код ће позвати *GetPixel* и *SetPixel* преко 4 милиона пута. Те функције нису најефикаснији начин да се преузму подаци пиксела са слике.

Да би извршили тестирање исписаног кода потребно је покренути програм, учитати слику,а затим кликнути на дугме конвертуј, чиме добијамо резултат приказан на слици 14.



Слика 14. Конвертована слика

4.5 Проналажење контуре

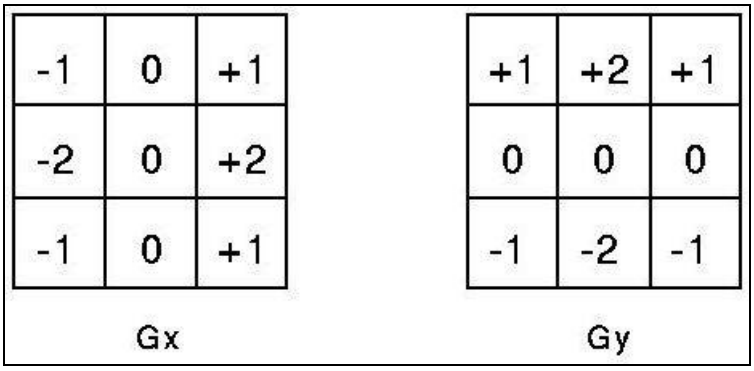
Детекција ивица је термин који се користи у обради слике заснован на алгоритмима који се баве променама у светлини, односно боји . Филтери за детекцију контуре раде на принципу контраста слике. Ово се може урадити на велики број различитих начина, спирални филтери примењују негативну тежину на једној ивици, и позитивну на другој. Ово поступак има ефекат мреже тежећи према нули ако су вредности исте, и тренд раста, ако контраст постоји.

У овом раду као сонова за одређивање контура послужио је *Sobel* оператор. Ова метода није брза, али је једноставна и одлична за разумевање. Након проналажења контуру ћемо приказати у новом *PictureBox*-у.

На основу анализе, теорија може да се пренесе на две димензије докле год постоји прецизна апроксимација за израчунавање деривата дводимензионалне слике. *Sobel* оператер обавља функцију мерења *2-D* просторног градијент на слици.

Обично се користи да пронађе приближно апсолутну величину градијента на свакој тачки на улазу сивих тонова слике. *Sobel edge detector* користи пар 3x3 спиралних маски, једну која процењује градијент у к-правцу (колоне) и друга која процењује градијент у и-правцу (редови).

Спирална маска је обично много мања од стварне слике. Као резултат тога, маска клизи преко слике, манипулишући граничним пикселима истовремено. Стварне *Sobel* маске су приказане на слици 15.



Слика 15. *Sobel* матрице

Ова језгра су дизајнирана да максимално одговарају ивицама који се налазе вертикално и хоризонтално у односу на мрежу пиксела, по једно језгро за сваку од две нормалне оријентације. Језгра се могу применити одвојено на улазној слици, како би произвела одвојена мерења компоненте градијента у свакој оријентацији (*Gx* и *Gy*). Они се могу комбиновати заједно како би се пронашла апсолутна величина градијента у свакој тачки и оријентацији градијента . Величина градијента се добија из обрасца:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

4.5.1 Креирање кода

Да бисмо пронашли релативне промене у нивоу интензитета, алгоритам обрађује слику један по један пиксел. Он тражи промене у интензитету са леве и десне стране одабраног пиксела, чува податке и затим проверава промене у интензитету изнад и испод тренутно одабраног пиксела, и њих чува такође. Сам процес се одвија за једну димензију у датом тренутку за сваки пиксел (хоризонталну а потом вертикалну), а затим комбинује резултат у дводимензиони пиксел, излазну слику.

Када се сваки пиксел прорачуна, прелази се на прорачун интензитета суседних пиксела, а затим се одузима од суседног пиксела на супротној страни. Добијени збир свих пиксела је релативна промена у интензитету на тој локацији. Ако су супротни суседни пиксели исте боје, када се један одузима од другог резултат ће бити нула (црно).

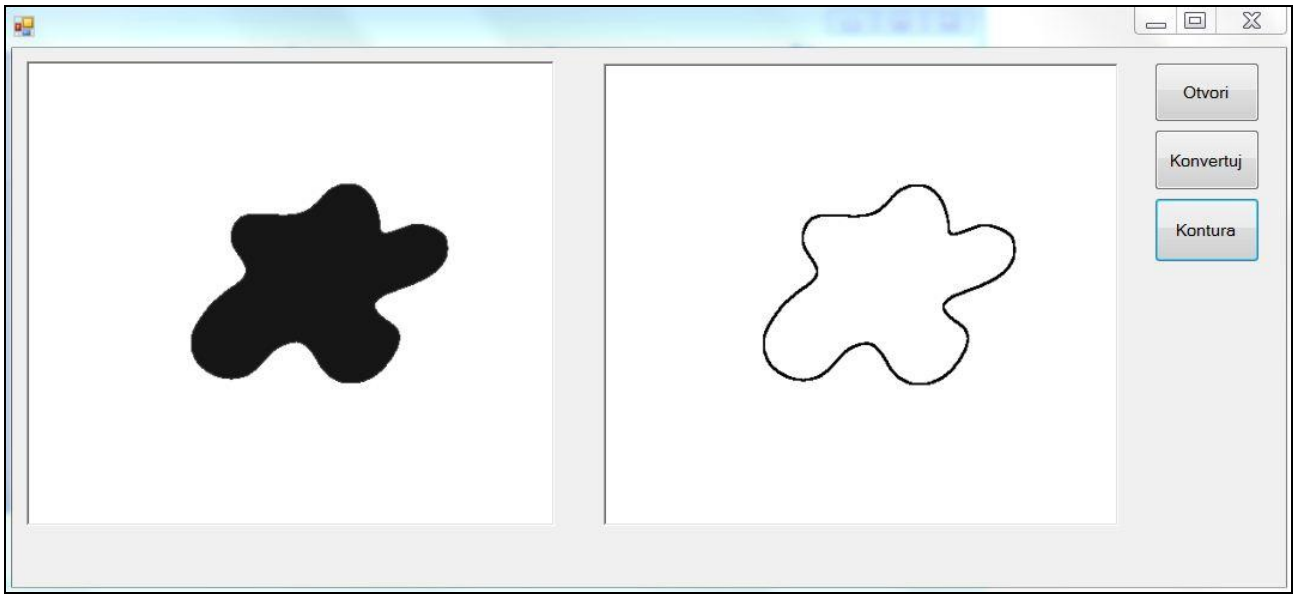
Ако су два суседна пиксела радикално различитих нивоа интензитета излаз ће бити већи од нуле. Механизам који прорачунава просек пиксела је матрица, једна за вертикалне и једна за хоризонталне пикселе (слика 16).

```
private void button3_Click(object sender, EventArgs e)
    {
      pictureBox2.Image = kontura(pictureBox1.Image);

    }
    public Image kontura(Image im)
    {
              //matrica po Gx
              int[,] gx = new int[,] { { -1, 0, 1 }, { -2, 0, 2 }, { -1, 0, 1 } };
              //matrica po Gy
              int[,] gy = new int[,] { { 1, 2, 1 }, { 0, 0, 0 }, { -1, -2, -1 } };

              Bitmap b = (Bitmap)im;
              Bitmap b1 = new Bitmap(im);
       //petlja koja prelazi preko piksela po visini
      for (int i = 1; i < b.Height - 1; i++)
      {
       // petlja koja prelazi preko piksela po vširini
        for (int j = 1; j < b.Width - 1; j++)
        {
          float new_x = 0, new_y = 0;
          float c;
          //petlja za spiralnu matricu
          for (int hw = -1; hw < 2; hw++)
          {
            for (int wi = -1; wi < 2; wi++)
            {
              //obrada inteziteta piksela
              c = (b.GetPixel(j + wi, i + hw).B + b.GetPixel(j + wi, i + hw).R
              + b.GetPixel(j + wi, i + hw).G) / 3;
              new_x += gx[hw + 1, wi + 1] * c;
              new_y += gy[hw + 1, wi + 1] * c;
            }
          }
          //ukoliko se piksel nalazi na ivici dodelićemu se crna boja
          if (new_x * new_x + new_y * new_y > 128 * 128)
            b1.SetPixel(j, i, Color.Black);
          else
            b1.SetPixel(j, i, Color.White);
        }
      }
      return (Image)b1;
    }
```

Слика 16. Алгоритам за одређивање контуре

За вертикалну *y* маску строго хоризонтални елементи су нула, а супротно важи за *x* маску. Елементи сваке матрице су помножени нивоима интензитета граничних пиксела, а затим се резултати сабирају. Пошто су наспрамне стране у супротним знацима, збир је промена у интензитету коју смо тражили. Последњи корак је додавање апсолутне вредност хоризонталне и вертикалне разлике у интензитету. Овај процес је заправо груба апроксимација математичког градијента слике.

Како би проверили резултат кода који смо написали, покренућемо *debugger*, учитати слику, конвертовати је у *grayscale*, а затим кликом на дугме Контура у *PictureBox*-у добићемо контуру објекта који се налази на слици (слика 17).



Слика 17. Исцртана контура у *PictureBox2*

4.5.2 Брзина извршења алгоритма

Приликом покретања програма, желео сам да задржим детаље о раду података са слике на минимуму, па сам креирао алгоритам за рад са веома спорим *GDI* битмап објектима користећи *GetPixel* и *SetPixel* методе.

Овај процес обично се извршава за око 3 K пиксела / секунди, што звучи брзо, али заправо је потребно око 160 секунди да се обради слика од 800 x 600 пиксела (480 K пиксела). Тако да је за комерцијалну употребу готово потпуно бескористан. Иако је веома спор, овај метод функционише веома добро и за мене као почетника испуњава све захтеве, такође може бити веома користан као основа за тестирање нових приступа.

4.6 Векторизација контуре

Векторизација или растер-вектор конверзије је без сумње централни део обраде у графици. Бави се конвертовањем обрађене слике у векторски облик који је погодан за даљу обрада и анализа. Многи алгоритми за векторизацију су развијен од како је компјутерска технологија доступна за ову сврху, такође је порастао и број софтверских производа који су се нашли на тржишту. Како су многи софтверски пакети доступни, проблем растер-вектор конверзије се сматра решен. Ипак, и даље постоје проблеми прецизности, робусности и стабилност процеса векторизације. Векторизација се може поделити на векторизацију на ниском нивоу, и векторизацију високом нивоу. Нижи ниво се бави препознавањем основних графичких објеката, укључујући праве линије, лукове, кругове и криве. Високи ниво обухвата структурне анализе. На пример, *Optical Character Recognition (OCR)* и графичко препознавање објеката су области у којима је потребан високи ниво векторизације.

Већина алгоритама за векторизацију су развијени тако да буду свестрани за многе врсте инжењерских цртежа, као што су мапе, механички цртежа, и електронски цртежи. Многи од њих дају добре резултате ако су графички објекти изоловани, али су склон грешкама ако се објекти секу или додирују. Типичан проблем је да ће се линија која се обрађује зауставити када се уочи прелаз на другу линију, тако да ће линија бити призната као неколико линија раздвојених укрштањем. Да би реконструисали оригинални линију, потребно је да постпроцесор анализира прелазе и повеже колинеарне линије. Ако дебљина оригиналне линије није усклађена због деградације, тада је компликовано реконструисати одговарајућу линију.

У поређењу са растерским сликама, векторске слике су економична алтернатива. Вектор фајлови имају следеће погодности у мапираним апликацијама:

- потребно је много мање сладишних капацитета,
- могу бити смањени на било коју величину без губитка резолуције,
- могу бити модификоване са *CAD/GIS* програмима.

Ово је важан аспект, јер можемо додати нови пут или мост на мапи без пројектовања од нуле.

Због све веће употребе компјутерске технологије за сврхе мапирања постоји потреба за дигитализацијом постојећих папирних мапа. Векторизација је алтернатива ручној дигитализацији, што је дуготрајан метод. Мапе имају тачну скалу, и објекти на њој имају одређене величине и тачно дефинисане локације. Стога је тачност векторизацију од кључног значаја за мапирање.

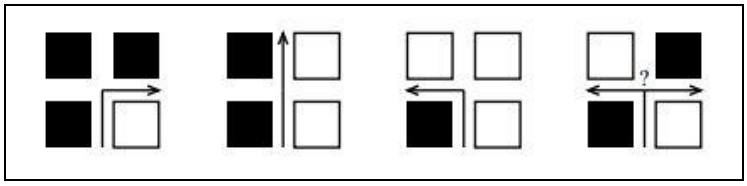
4.6.1 Опис алгоритма за векторизацију

Код који је приказан у раду заснован је *Potrace* алгоритму за креирање вектора. Алгоритам који ћемо објаснити трансформише битмапу у векторску линију у неколико корака. У првом кораку, битмапа се разграђује у неколико путања, које формирају границе између црних и белих области. У другом кораку, свака стаза ограничена је оптималним бројем полигона. У трећем кораку, сваки полигон се трансформише у глатку контуру. Коначно, излаз се генерише у захтеваном формату.

4.6.2 Креирање путање вектора *(Paths)*

На самом почетку можемо замислити на нашој растерској контури такав координатни систем где углови сваког пиксела имају целе координате. Даље знамо да је боја позадине слике бела, а боја контуре црна. Делови координатне равни који леже ван битмапиране контуре биће испуњени белим пикселима. Сада је потребно конструисати директан графикон из наше битмапе. Нека *p* буде тачка целе координате, таква тачка се налази поред четири пиксела. Ту тачку назваћемо *vertex* уколик четири пиксела нису исте боје. Ако су *v* и *w* темена, кажемо да постоји ивица од *v* до *w,* уколико је растојање између *v* и *w* 1, права линија која се протеже од *v* до *w* одваја црни пиксел из беле боје, тако да је црни пиксел на њеној левој страни и бели пиксел је на њеној десној страни док путујете у правцу од *v* до *w.*

Potrace користи следећу једноставну методу како би разлажио битмапу у стазе. Почиње одабиром пар суседних пиксела различите боје. Ово се може постићи, на пример, избором са леве стране црног пиксела у неком реду. Два одабрана пиксела сусрећу се на ивици. Ми ћемо оријентисати ову ивицу, тако да је црни пиксел на њеној левој страни, а бели пиксел је на десној. Ова ивица дефинише стазу дужине један. Након тога ћемо наставити да продужујемо ову стазу на такав начин да свака нова ивица има црни пиксел на левој и бели пиксел на десној страни, у односу на правац путање. Другим речима, крећемо се дуж ивице између пиксела, и сваки пут смо погодили угао, или идемо право или скрећемо лево или десно, у зависности од боје околних пиксела као што је приказано на слици 18.



Слика 18. Проналазак путање

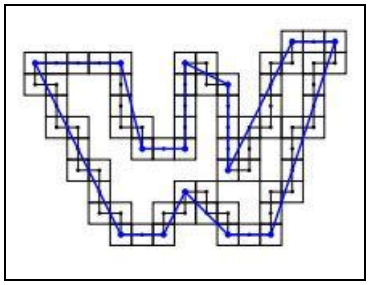
Овај процес ће се наставити све док се не вратимо до тачке од које смо почели, у том тренутку смо дефинисали затворену контуру.

Сваки пут када пронађемо затворену контуру, уклонићемо је из графикона окретањем боје пиксела у своју унутрашњост. На овај начин дефинисаћемо нову битмапу, на коју смо применили алгоритам рекурзивно све док више није преостао ни један црни пиксел. Резултат је скуп затворених путања које се даље могу обрађивати (слика 19) .

```
    private void draw()
    {
      if (ListOfCurveArray == null) return;
      Graphics g = Graphics.FromImage(pictureBox1.Image);
      GraphicsPath gp = new GraphicsPath();
      for (int i = 0; i < ListOfCurveArray.Count; i++)
      {
        ArrayList CurveArray = (ArrayList)ListOfCurveArray[i];
        GraphicsPath Contour=null;
        GraphicsPath Hole = null;
        GraphicsPath Current=null;
      for (int j = 0; j < CurveArray.Count; j++)
        {
          if (j == 0)
          {
            Contour = new GraphicsPath();
            Current = Contour;
          }
          else
          {

            Hole = new GraphicsPath();
            Current = Hole;
           }
          Potrace.Curve[] Curves = (Potrace.Curve[])CurveArray[j];
          float factor = 1;
          if (radioButton1.Checked)
            factor = (trackBar1.Value + 1);
          for (int k = 0; k < Curves.Length; k++)
          {
            if (Curves[k].Kind == Potrace.CurveKind.Bezier)
              Current.AddBezier((float)Curves[k].A.x * factor,
     (float)Curves[k].A.y * factor, (float)Curves[k].ControlPointA.x * factor,
     (float)Curves[k].ControlPointA.y * factor,
    (float)Curves[k].ControlPointB.x * factor, (float)Curves[k].ControlPointB.y * factor,
     (float)Curves[k].B.x * factor, (float)Curves[k].B.y * factor);
            else
    Current.AddLine((float)Curves[k].A.x * factor, (float)Curves[k].A.y * factor,
     (float)Curves[k].B.x * factor, (float)Curves[k].B.y * factor);
          }
          if (j > 0) Contour.AddPath(Hole, false);
        }
        gp.AddPath(Contour, false);
      }
```

Слика 18. Алгоритам за конвертовање битова

Код који је приказан на предходној слици извршиће оптималну векторизацију битова креирањем најсличније путање оргиналном објекту (слика 19).



Слика 19. Оптимална путања по полигону

Излаз горе наведеног кода је оптималан полигон који је приближан овој путањи. Путања није равна ако се сва четири смера јављају на њеном путу. Јасно је дефинисанио да ако је путања равна, онда су равне све њене подпутање.

Да би се израчунало да ли је дата путања равна или не, ми користимо чињеницу да је равност одређена са три особине. Ова опсервација доводи до алгоритма за тестирање равности који је веома сложен, зато се може приступити једноставном тестирању које се врши по три особине (*i, j, k*).

У реализовању алгоритма користимо оптимизацију која нам омогућава да пронађемо све равне подпутање датог затвореног пута дужине *n* у времену *O* у најгорем случају. Укратко, трик је да се израчуна, за сваки пар (*i, j*), ограничење на положај. Ако је *i* фиксно а *j* се повећава, довољно је проверити ограничење једном за сваки j. Штавише, ограничење се састоје од највише две неједнакости и може да се ажурира и проверава у сваком тренутку.

Завршна фаза алгоритма трансформише полигоне добијене у глатке векторске контуре. У прелиминарном кораку потребно је подесити положај темена полигона да одговара оригиналној битмапи што је најбоље могуће. У главном кораку потребно је израчунати углове кривих на основу дужине суседних линијских сегмената и углова између њих.

Функцију *draw* позваћемо у структури *refreshPicture* која нам освежава слику у *PictureBox2*, и уместо битмапиране контуре приказује нам векторску контуру (слика 20).

```
private void refreshPicture()
    {
      if (Matrix == null) return;
      {
        pictureBox2.Width = B.Width;
        pictureBox2.Height = B.Height;
         pictureBox2.Image = B;
      }

      draw();
    }
```

Слика 20. Позив функције *draw*

Затим структуру *refreshPicture* позивамо у телу комадног дугмета *button4*, чиме ће се извршити како исцртавање контура тако и освежавање слике (слика 21).

```
      private void button4_Click(object sender, EventArgs e)
    {
      //formiranje liste sa novom konturom
      ListOfCurveArray = new ArrayList();
      //optimizovanje putanje, zamenjivanjem sekvenci Bezijerovih segmenata
      //jednodelnim segmentima kada je to moguće.
      Potrace.curveoptimizing = checkBox4.Checked;
      Potrace.potrace_trace(Matrix, ListOfCurveArray);
      refreshMatrix();
    }
```

Слика 21. Телу комадног дугмета *button4*

4.6.3 Чување контуре у векторском формату

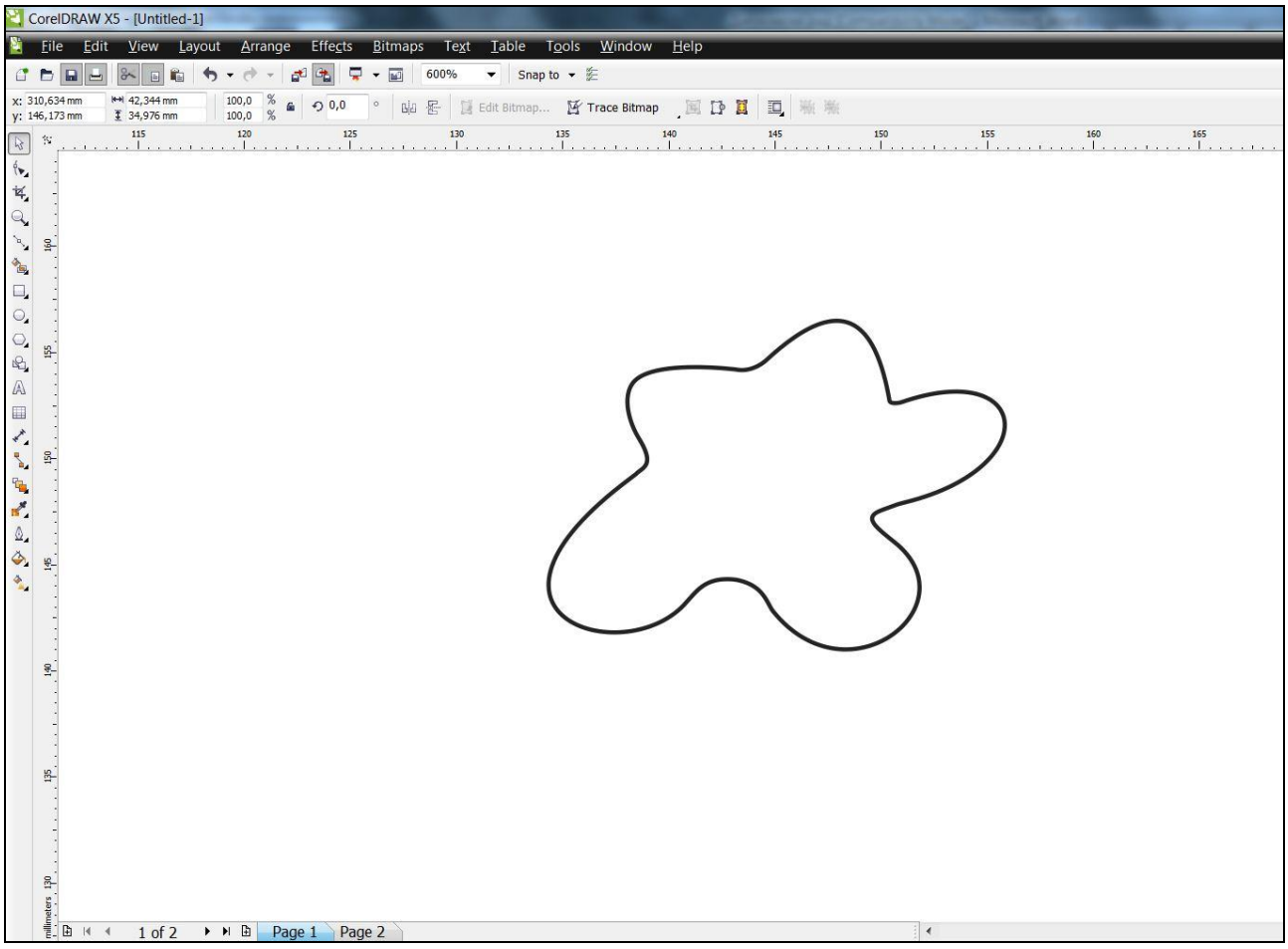
Како би комплетирали процес конвертовања контуре у вектор потребно је добијени објекат сачувати у неком од векторских формата. У нашем случају одлучили смо се за *SVG* формат који је у широкој употреби и може га отворити готово сваки векторски програм .

```
private void button5_Click(object sender, EventArgs e)
    {
      if (saveFileDialog1.ShowDialog() == DialogResult.OK)
      {
        string s = Potrace.Export2SVG(ListOfCurveArray, Bitmap.Width, Bitmap.Height);
        System.IO.StreamWriter FS = new
        System.IO.StreamWriter(saveFileDialog1.FileName);
        FS.Write(s);
        FS.Close();
      }
    }
```

Слика 22. Телу комадног дугмета *button4*

5 УЧИТАВАЊЕ КОНТУРЕ У *CORELDRAW*

Како би били сигурни да је апликација извршила потребну конверзију отвориће мо контуру коју смо претходно сачували у програму *CorelDraw*. Уколико *Corel* не препозна сачувану слику то значило да је у процесу конверзије или чувања слике дошло до грешке. Међутим учитавање слике је протекло без проблема, контура коју смо конвертовали приказала се на радној површини без грешака (слика 23).

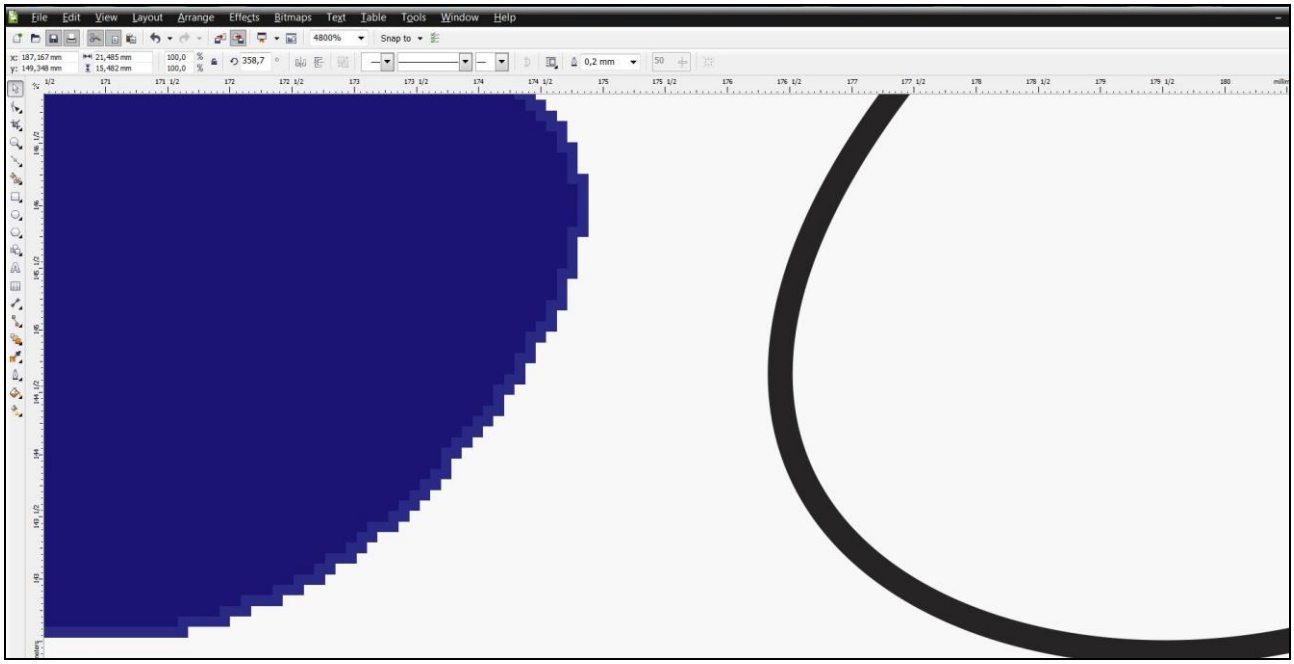


Слика 23. Конвертована контура приказана у програму *CorelDraw*

Иако се приликом одређивања контуре и векторизације увек губи одређени део оргиналности облика, можемо рећи да је контура настала у приказаној апликацији веома веродостојна и да је испунила очекивања.

5.1 Карактеристике оргиналне слике и векторске контуре

Да би добили праве резултате процеса одређивања контуре на одређеном објекту и векторизације, најбоље је добијену контуру упоредити са оргиналном сликом. На самом почетку напоменућемо да је величина оргиналне битмапране слике 404 *KB*, док је величина векторизоване слике 1,44 *KB*. Одмах можемо уочити да је количина меморије коју конвертована слика заузима на рачунару готово 300. пута мања, што је огромна уштеда поготово ако се ради о већим и сложенијим сликама. Међутим праву проверу представља квалитет исцртаних линија. Након што смо у *CorelDraw* учитали и битмапирани оргинал, добили смо запањујуће резултате (слика 24).



Слика 24. Квалитет ивица оргиналне битмапиране слике и конвертоване онтуре

На први поглед обе слике су изгледале слично јер су веома мале, међутим приликом значакног зумирања можемо увидети да на оргиналној слици готово можемо избројати битове од којих је састављена, док је векторизована контура савршених ивица чак и при значајном зумирању. Поред значајне разлике у квалитету и величини коју заузимају, векторизована слика је знатно лакша за обраду и дорађивање што је веома битно. Овим резултатима апликација је показала и своју практичну примену и употребљивост.

6 ЗАКЉУЧАК

Конверзија података из битмапе у вектор је кључна функција укључена у комерцијалне софтверске пакете. Међутим, исцртавање контура одређеног објекта и њихова векторизација тренутно представља процес који може одузети доста времена, што указује на потребу побољшања ефикасности. Овим радом представљена је апликација која одређује контуре битмапираних објекта на слици и врши њихово конвертовање у векторски облик. Битмапа представља слику као мрежу црних или белих пиксела. Вектор приказује слику путем алгебарских описа својих контура, чију типичну форму можемо видети у облику *Bezier* криве. Предност представљања слике као векторске контуре је то што може бити смањена на било коју величину без губитка квалитета.

Јасно је да ниједан алгоритам за конвертовање не може бити савршен у апсолутном смислу, јер генерално гледајући, постоји много линија које могу довести до исте битмапе. С друге стране, од многих могућих контура које би могле представити облик датог битмапираног објекта, неке су очигледније, или естетски прихватљивије од других. Сваки добар алгоритам за препознавање контура мора да обавља више функција. Две од тих функција су одређивање најверодостојније криве која одговара задатом облику, као и одређивање углова. Између ова два циља мора постојати компромис. Ако се превише углова открије, излаз ће изгледати као полигон и више неће бити гладак. Ако се открије само неколико углова, излаз ће изгледати глатко, али сувише заобљено.

У овом раду описан је настанак алгоритма који је једноставан, поуздан, и тежи да произведе одличне резултате. Једини недостатак је нешто веће време обраде података, што је нормално јер су коришћени базни алгоритми најједноставнији за разумевање.

7 ЛИТЕРАТУРА

[1] Ненад Филиповић (2006), Објектно оријентисано програмирање, ФТН Чачак.

[2] Lindeberg (1998) "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision.

[3] H.G. Barrow and J.M. Tenenbaum (1981) "Interpreting line drawings as three-dimensional surfaces", Artificial Intelligence.

[4] J. M. Park and Y. Lu (2008) "Edge detection in grayscale, color, and range images", in B. W. Wah (editor) Encyclopedia of Computer Science and Engineering .

[5] Irwin Sobel, 2014, History and Definition of the Sobel Operator.

[6] Larry O'Brien and Bruce Eckel (2005). Thinking in C#, Prentice Hall Upper Saddle River, New Jersey.

[7] http://stackoverflow.com

[8] http://www.codeproject.com

[9] http://msdn.microsoft.com

[10] http://www.youtube.com/user/ProgramiranjeDJB