KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Dainius Silvanavičius

Vaizdo pasukimo algoritmų tyrimas

Magistro darbas

Darbo vadovas
doc. dr. Armantas Ostreika

KAUNAS 2011

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Vaizdo pasukimo algoritmų tyrimas

Magistro darbas

Atliko .....
(parašas)
Dainius Silvanavičius, IFM-9/1

2011m. 05

Vadovas .....
(parašas)
doc. dr. Armantas Ostreika

2011m. 05

Recenzentas .....
(parašas)
doc. dr. Regina Misevičienė

2011m. 05

KAUNAS, 2011

Santrauka

Šiame darbe tiriamas skaitmeninių vaizdų pasukimo algoritmų efektyvumas, kokybės ir darbo laiko atžvilgiu. Darbe pateikiama populiariausių vaizdo pasukimo metodų apžvalga.

Pateikiamos eksperimento atlikimo schemos. Darbe išsamiai komentuojami atliktų eksperimentų su skaitmeniniais vaizdais rezultatai. Remiantis gautais rezultatais, formuluojamos išvados apie vaizdų pasukimo efektyvumą naudojant skirtingus algoritmus, pateikiamos argumentuotos išvados apie optimalaus algoritmo parinkimą. Be to nurodomos tolimesnių tyrimų kryptys bei pateikiama su darbu susijusi literatūra.

Summary

This Master degree paper analyses image rotation algorithms, their efficiency, in quality and work time. Several popular image rotation methods (both lossless and lossy) are reviewed.

Schemes for the experimental examination are prepared. This paper comprehensively describes experiments with images and the results of the experiments. Conclusions about the efficiency of image rotation are based on the results of the experiments. Guidelines for further research are also discussed.

Terminai ir santrumpos

AVG – aritmetinis vidurkis.

C++ – objektinio programavimo kalba

Directx – aplikacijų programavimo sąsajų rinkinys, skirtas multimedijos užduočių tvarkymui.

Interpoliacija – tai procesas apjungti atskirus taškus taip, kad galima būtų gauti tinkamus duomenų įverčius tarp duotų taškų

Qt – tarpplatforminė aplikacijų programavimo sąsaja ir vartotojo sąsajos karkasas skirtas C++ programavimo kalbai.

USC-SIPI – skaitmeninių vaizdų duomenų bazė. Jos tikslas – remti skaitmenių vaizdų apdorojimo tyrimus, vaizdų analizę.

VKP – vidutinė kvadratinė paklaida.

Turinys

Paveikslų sąrašas

Lentelių sąrašas

1. Įvadas

Skaitmeninė fotografija išsprendė vaizdų išsaugojimo bei atkūrimo (iš dalies perdavimo) problemas, bet žmonėms iškilo poreikis analizuoti, modifikuoti ir kitaip apdoroti išsaugotus vaizdus. Tas, praktiškai, buvo sunkiai įgyvendinama operuojant analoginiais vaizdais. Vaizdų apdorojimas tapo įmanomu tiktai atsiradus kompiuteriams. Kita vertus norint apdoroti vaizdus kompiuteriu, pastarieji, akivaizdu, turi būti ne tik diskretizuojami, bet ir kvantuojami (fiksuojamos šviesos intensyvumo reikšmės). Paprasčiausias tokio pertvarkio rezultatas – plokščias dvimatis skaitmeninis paveikslėlis, paprastai tapatinamas su dvimačiu taškų masyvu, kuriame užrašomi užfiksuoti vaizdo elementų (pikselių) šviesos intensyvumai. [1]

Vaizdo pasukimo operacijos yra vienas iš svarbiausių vaizdo modifikavimo tipų. Vaizdų pasukimai yra naudojami daugelyje skirting disciplinų: medicina, skaitmeninė fotografija, kompiuteinė grafika.

Netikslumai vaizdų pasukime gali sukelti subtilias problemas. Pačiame paprasčiausiame lygyje, keleto pasukimų suma, kuri sudaro pilną 2π pasukimą, gali nesugrąžinti vaizdo į pradinę padėtį. Tai gali turėti nepageidaujamą efekta vaizdų apdorojime. Pavyzdžiui medicinoje, informacijos praradimas vaizdo sukimo metu, gali pradanginti daug mažų, bet labia svarbių anatominių savybių.

Netikslumai sukant vaizdus atsiranda dėl skaitmeninio vaizdo savybių. Skaitmeninis vaizdas yra diskretizuotas realaus pasaulio vaizdas. Jo informacija saugoma dvimačiame taškų masyve. Norint pasukti tokį masyvą kuriuo nors kampu, reikia perkelti visas jo reikšmes į naujas koordinates. Kadangi vaizdas nėra tolydus, o diskretizuotas, jį pasukus kitu kampu, daug taškų persidengia vienas su kitu. Norint gauti geros kokybės pasuktą vaizdą, reikia gerai parinkti atitinkamų tašku vietas naujame vaizde. Šiuo metu yra daug vaizdo pasukimo algoritmų, kurie kiekvienas kitaip sprendžia šią problemą. [2]

Šio tiriamojo darbo tikslai yra šie:

- susipažinti su skaitmeninių vaizdų pasukimo algoritmo koriniais principais,
- eksperimentiškai ištirti populiariausius vaizdų pasukimo algoritmus,
- remiantis eksperimentų rezultatais, pasiūlyti algoritmo pakeitimą, kuris pagerintų jo rezultatų kokybę.

2. Skaitmeninių vaizdų pasukimo algoritmų analizė

2.1 Skaitmeninių vaizdų erdvė

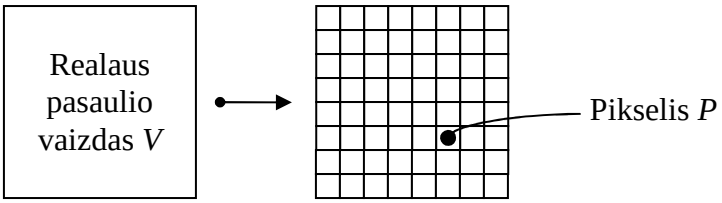
Realaus pasaulio vaizdai gali būti nagrinėjami dviem aspektais: sandaros ir tuo, kaip žmogus juos mato (priima). Vaizdai yra sudaryti iš realaus pasaulio objektų išspinduliuotos (ar atspindėtos) šviesos spindulių, kurie projektuojami į akies tinklainę. Būtent tuo ir remiamasi, sudarinėjant matematinį realaus pasaulio vaizdų modelį, o realaus pasaulio vaizdas siejamas su šviesos spindulių projekcija į ribotą fizinių matmenų stačiakampią plokštumos sritį. [3]

Tariama, jog R – realaus pasaulio vaizdų aibė, o $V \in R$ – šios aibės elementas (realaus pasaulio vaizdas). V būdingos šios savybės:

1. Kiekvienas realaus pasaulio vaizdas $V \in R$ turi fizinius matmenis (euklidinės erdvės „gabalėlį“, kurį toliau vadinsime „atrama“) – $I = \{(x, y) \in R^2 | a \le x \le b, c \le y \le d$. Kiekvienas „atramos“ taškas sutampa su vieninteliu realaus pasaulio vaizdo V tašku, ir atvirkščiai. Atstumas tarp bet kurių dviejų vaizdo „atramos“ taškų (x_1, y_1) ir (x_2, y_2) yra matuojamas euklidine metrika $d = d(x, y) = \sqrt{(x_1 - x_2) + (y_1 - y_2)^2}$.

2. Vaizdas $V \in \Re$ turi spalvines (chromatines) charakteristikas, kurios nusako šviesos dažnius (spalvas) ir intensyvumus, susijusius su vaizdo V poaibiais (fragmentais, taškais). Šias chromatines charakteristikas galima modeliuoti panaudojant realiąsias funkcijas. Chromatinės vaizdo charakteristikos dažniausiai aprašomos panaudojant realiųjų skaičių intervalą I=[0;255]. Spalvotą vaizdą galima modeliuoti kaip susidedantį iš keleto spalvinių komponenčių, kai atskirai fiksuojamas kiekvienos spalvinės komponentės intensyvumas.

3. Realaus pasaulio vaizdas nėra susietas su jokia jo detalizacijos schema, t.y. vaizdui V galima parinkti bet kokį detalizacijos lygį, priskiriant fiksuotiems vaizdo fragmentams (pikseliams) skaitines reikšmes, nusakančias jų spalvas ir intensyvumus, t.y. chromatines charakteristikas. 2.1 pav vaizduoja detalizacijos schema.



2.1 pav. Vaizdo detalizacijos schema

4. Aibė $\Re$ yra uždara „iškirpimo“ operacijos atžvilgiu (2 pav.). Aibė $\Re$ yra uždara izotropinio ištempimo atžvilgiu. Aibė $\Re$ yra uždara atspindžio (kurios nors iš „atramos“ briaunų atžvilgiu) bei posūkio operacijų požiūriais. Paminėtos savybės leidžia teigti, jog realaus pasaulio vaizdų aibė $\Re$ yra neišsemiama, t.y. koks mažas bebūtų realaus pasaulio vaizdas, jis kaupia savyje begalinę informaciją, ir tuo pačiu yra pilnateisis vaizdų aibės $\Re$ narys.

5. Baigtiniai pasirinkto detalizacijos lygio vaizdo modeliai gaunami fiksuojant chromatines charakteristikas tik tam tikruose vaizdo taškuose (pikseliuose). Konkretaus vaizdo elemento šviesos intensyvumo reikšmė gali būti apskaičiuojama taip:

$$f(P) = \frac{\iint\limits_{P} f(x,y)\,dx\,dy}{\iint\limits_{P} dx\,dy}$$

Čia: P – pikselio užimama atramos sritis ir f(x,y) – šviesos intensyvumą taške (x,y) nusakanti funkcija. Norint gauti skaitmeninį vaizdą, reikia ne tik diskretizuoti jį, bet ir kvantuoti šviesos intensyvumo lygius. Kvantavimas atliekamas, atsižvelgiant į tai, jog žmogaus akis logaritmiškai reaguoja į šviesos intensyvumo didėjimą (mažėjima); be to, žmogaus akis fiksuoja tik 1 proc. šviesos intensyvumo pokyčius. Taigi jei I_0 – pradinis intensyvumo lygis, tuomet: $I_1 = 1{,}01\ I_0$, $I_2 = (1{,}01)_2\ I_0$, ..., $I_n = (1{,}01)_n\ I_0$.

Atlikus vienoje ar kitoje erdvėje diskretizavimą bei kvantavimą, jau galima kalbėti apie apibendrintą skaitmeninių vaizdų aibę:

$$S^d(n) = \{ [X(m)] \mid m = (m_1, \ldots, m_d)\ \ I^d \};$$

čia: $I = \{0,1,...,N-1\}$, $N = 2^n$, $n \in N$, $X(m) \in \{0,1,...,2^p - 1\}$ ir nusako m-ojo vaizdo elemento (pikselio) šviesos intensyvumą; skaičius *p* nurodo pikselio reikšmėms koduoti skirtą bitų kiekį (kai $p=1$, turime dvispalvį – juodai baltą vaizdą; kai $p>2$ turime pilką (nespalvotą) vaizdą su 2^P intensyvumo lygiais); parametras *n* charakterizuoja vaizdo detalizacijos lygį; *d* – parametras nusakantis vaizdo dimensiškumą ($d \in \{1, 2,3\}$). Dimensiškumo įvedimas leidžia praplėsti skaitmeninio vaizdo sąvoką, įvedant vienmatę ir trimatę vaizdų erdves. Vienmačio skaitmeninio vaizdo pavyzdžiu gali būti diskretizuotas garsas, EKG signalai, o trimačio vaizdo – video sekos.

Skaitmeninių vaizdų aibę papildome, įvesdami atstumo (metrikos) tarp bet kurių dviejų šios aibės elementų sąvoką:

$$\delta = \delta(X_1, X_2) = \sqrt{\frac{1}{N^d} \sum_{m \in I^d} (X_1(m) - X_2(m))^2} ;$$

čia $[X_1(m)]$ ir $[X_2(m)]$ yra du skaitmeniniai vaizdai. Ši metrika vadinama – vidutinė kvadratinė paklaida.

2.2 Vaizdų pasukimo algoritmu apžvalga

2.2.1 Vaizdų pasukimo algoritmai be informacijos praradimo.

Vaizdą sukant 90, 180 arba 270 laipsnių kampu, interpoliacija nėra būtina. Šiuos pasukimus lengva įvykdyti perkeliant atitinkamų pikselių koordinates - pasirinkti pradinius taškus, kurie atitinka kiekvieną paskirties vietos pikselį ir nustatyti paskirties vietos pikselių vertes. Ortogonalūs pasukimai gali būti patikrinti atliekant juos nuosekliai keturis kartus (du kartus 180 laipsnių pasukimui) ir lyginant rezultatus su pradiniu vaizdu. Šis palyginimas yra lengvai atliekamas naudojant XOR tarp rezultatų ir pradinių vaizdų, ir patikrinant, ar rezultatas yra "nulis", ty, nėra jokių skirtingų elementų. [4]

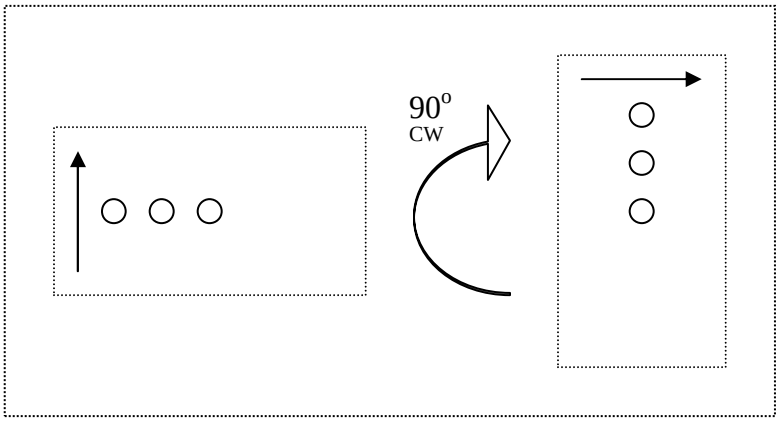
- Pasukimas 180 laipsnių kampu.

Tai yra konceptualiai labai paprasta: kiekviena vaizdo pikselių eilutė yra apverčiama ir sukeičiama su jai atitinkama apversta eilute kitoje vaizdo pusėje, eilutės yra sukeičiamos apie vaizdo centrinę eilutę.

- Pasukimas 90 laipsnių kampu.

90 laipsnių pasukimas gali būti arba pagal laikrodžio rodyklę (CW) arba prieš laikrodžio rodyklę (CCW). 90 laipsnių pasukimas, kaip 180 pasukimas, yra iš esmės paprastas: nuskaityti taškų šaltinius ir nukopijuoti juos į paskirties koordinates.

2.2 paveikslėlyje rodyklės rodo nuskaitomos eilutes skenavimo kryptį pradiniame ir rezultato vaizduose.



2.2 pav. Pasukimo 90° kampu algoritmo taikymo pavyzdys

2.2.2 Tipinis vaizdo pasukimo algoritmas

Sukimosi operatorius atlieka geometrinę transformaciją, kurios perkelia pradinio paveikslėlio pikselių pozicijas (x_1, y_1) į pasukto paveikslėlio pozicijas (x_2, y_2), pasukant paveikslėlį per nurodytą kampą θ apie tašką O. Daugeliu atvejų, gautos pozicijos (x_2, y_2), kurios nepatenka į vaizdą yra ignoruojamos. [4,5]

Sukimosi operatorius atlieka transformaciją pagal šias funkcijas:

$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta)*(y_1 - y_0) + x_0$

$y_2 = \sin(\theta) * (x_1 - x_0) - \cos(\theta)*(y_1 - y_0) + y_0$

kur (x_0, y_0) yra sukimosi centro koordinatės ir θ yra posūkio kampas, kuris laikrodžio rodyklės kryptimi turi teigiamas reikšmes. Pasukimo metu gaunamos pikselių pozicijos (x_2, y_2), kurios neatitinka vaizdo ribų, kurios apibrėžtos pirminio vaizdo dimensijomis. Tokiais atvejais pozicijos, kurios buvo priskirtos ne vaizdui yra ignoruojamos. Pikselių pozicijos, iš kurių vaizdas buvo pasukamas, dažniausiai yra užpildomos juodais pikseliais. [6,7]

Sukimosi algoritmas gali gauti koordinates (x_2, y_2), kurios nėra sveikieji skaičiai. Siekiant sukurti pikselių intensyvumą kiekvienai pozicijai, naudojami skirtingi interpoliavimo būdai. Du dažniausiai naudojami metodai yra:

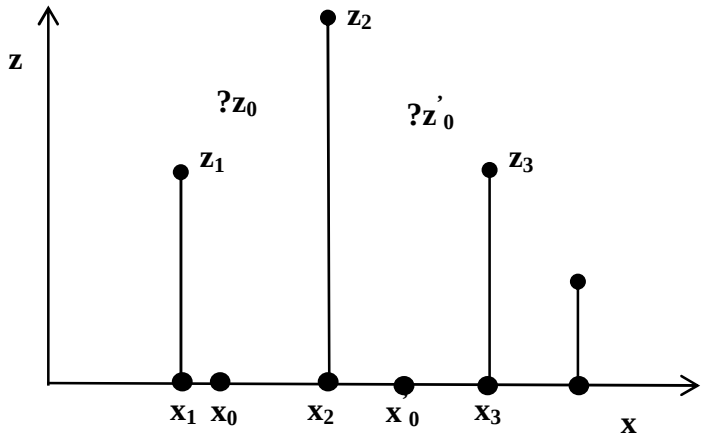
- Kiekvienam realaus skaičiaus pozicijoje (x_2, y_2) esančiam pikseliui priskirti artimiausio sveiko skaičiaus pozicijos reikšmę.
- Apskaičiuoti intensyvumo lygį kiekvienoje realioje pikselių pozicijoje pagal svertinį vidurkį *n* artimiausių sveikų skaičių reikšmių. Koeficientas yra proporcingas atstumui arba pikselių persidengimui artimose projekcijose.

Antrasis metodas duoda geresius rezultatus, bet padidėja algoritmo skaičiavimo laikas. [8]

2.2.3 Interpoliacija

Interpoliacija – tai procesas apjungti atskirus taškus taip, kad gauti tinkamus duomenų įverčius tarp duotų taškų.[9]

Interpoliavimo principas trumpai gali būti nusakomas 2.3 pav. pateikiamu pavyzdžiu. Logiška manyti, kadangi taškas x'_0 yra atstumo tarp x_2 ir x_3 viduryje, tai ir jo z'_0 reikšmė bus $½(z_2+z_3)$, o z_0 bus artimesnė z_1 nei z_2, kadangi x_0 arčiau x_1 nei x_2.



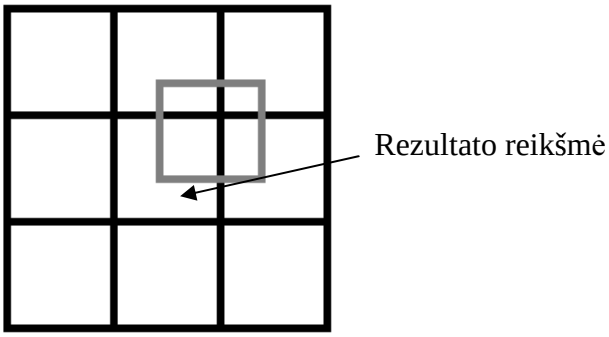
2.3 pav. Interpoliavimo uždavinys

Nežinomos taško reikšmės suradimui dažniausia naudojamos tik tam tikru požiūriu artimiausių taškų reikšmės.

2.2.3.1 Artimiausio kaimyno (Nearest Neighbor) interpoliacija

Šis interpoliavimo algoritmas yra pats paprasčiausias ir reikalaujantis mažiausiai skaičiavimo laiko. Artimiausio kaimyno algoritmas paprasčiausiai paima artimiausio taško reikšmę ir nekreipia dėmesio į kitas šalia esančias reikšmes. Šis algoritmas yra labai paprastai realizuojamas ir dažniausiai naudojamas realaus laiko trimačiuose atvaizdavimuose parenkant spalvas tekstūruotams paviršiams.

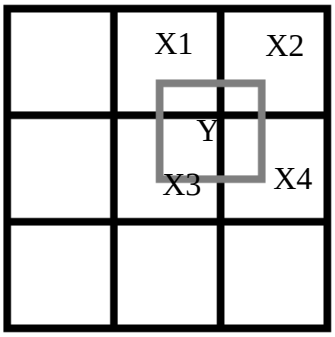
2.4 pav. vaizduojama interpoliacijos algoritmo iliustracija. Pilkai pavaizduotas ieškomas pikselis uždėtas ant pradinio vaizdo.



2.4 pav. Artimiausio kaimyno interpoliacija

2.2.3.2 Tiesinė (Bilinear) interpoliacija

Tiesinio interpoliavimo algoritmas paima 2x2 kaimyninių pikselių reikšmes, kurios yra aplink nežinomą pikselį. Tuomet jis paima svertinį vidurkį iš šių 4 taškų gaudamas galutinę interpoliuotą reikšmę. Rezultate gaunami daug sklandžiau atrodantys vaizdai, nei naudojant artimiausio kaimyno algoritmą.



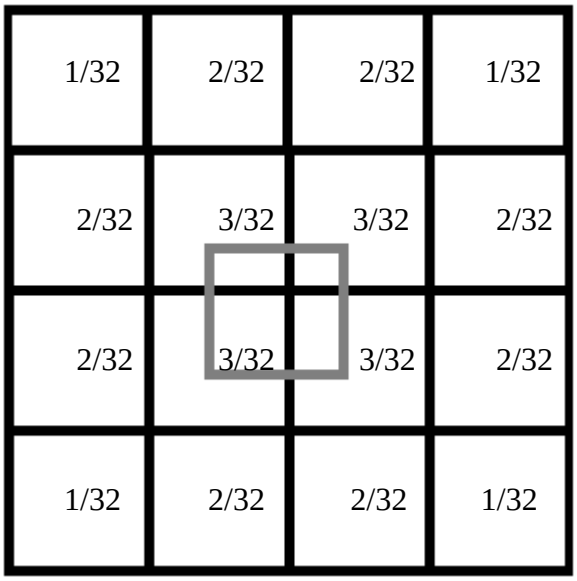
Y = ¼*X1+¼*X2+¼*X3+¼*X4

2.5 pav. Tiesinė interpoliacija

2.5 pav. vaizduojama interpoliacijos algoritmo iliustracija. Pilkai pavaizduotas ieškomas pikselis uždėtas ant pradinio vaizdo. Keturiems persidengiantiems pikseliams suteikiami vienodi svoriai.

2.2.3.3 Bikubinė (Bicubic) interpoliacija

Bikubinės interpoliacijos algoritmas paeina žingsniu į priekį už tiesinės interpoliacijos algoritmą, ir ima 4x4 kaimyninių pikselių reikšmes – iš viso 16 pikselių. Kadangi šios reikšmės yra ne vienodai nutolusios nuo nežinomo pikselio, tai artimesniems pikseliams yra suteikiamas didesnis svoris nei tolimesniems.



2.6 pav. Bikubinė interpoliacija (pikselių svoriai)

2.6 pav. Pavaizduotas bikubinėje interpoliacijoje naudojamas plotas su kiekvienam pikseliui priskirtu svoriu. Šiame pavyzdyje pavaizduotas pats paprasčiausias svorių paskirstymas. Realizuojant algoritmą, svoriai gali būti paskirti kiti.

2.2.3 Vaizdo pasukimo algoritmai su vienos dimensijos transformacijomis

Catmull ir Smith [9] vaizdo pasukimą su šlyties ir dydžio keitimo deformacijomis aprašė taip:

$$R(\Theta) = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} = \begin{bmatrix} \cos\Theta & \sin\Theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\tan\Theta & \frac{1}{\cos\Theta} \end{bmatrix}. \text{ [10, 11]}$$

Pirmoji matrica aprašo horizontalias šlyties ir dydžio keitimo operacijas, o antroji vertikalias. Transformacijas aprašant formulėmis, horizontali transformacija gaunama:

$x_2 = x_1 * \cos(\theta) + y_1 * \sin(\theta)$

$y_2 = y_1$

Vaizdas yra pastumiamas dydžiu $y_1*\sin(\theta)$ ir sumažinamas dydžiu $x_1*\cos(\theta)$. Taigi ši formulė yra sudaryta iš šlyties ir dydžio keitimo transformacijų tik horizontalia kryptimi.

Taip pat antroji transformacija aprašoma taip:

$x_2 = x_1$

$y_2 = y1 / \cos(\theta) – x_1 * \tan(\theta)$

Šį kartą transformacija atliekama tik vertikalia kryptimi, bet skirtumas tas, kad vaizdas yra padidinamas vertikaliai pagal dydį $y1 / \cos(\theta)$.

Šis dydžio keitimo būdas sukti vaizdams turi du trūkumus. Pirma, dydžio keitimo operacija reikalauja daugiau skaičiavimų, nes vaizdui turi būti ne tik pritaikyta šlyties deformacija, bet ir dydžio keitimo. Antra, nors vaizdo dydis nesikeičia sukimo metu, pirma dydžio keitimo operacija sumažina vaizdo dydį horizontalia kryptimi ir sukelia glodinimo problemų.

Vaizdo pasukimo matematika naudojant tik horizontalią ir vertikalią šlyties deformaciją yra paprasta. Horizontalioje šlytyje deformacijoje pastumiama eilutė pikselių horizontaliai atstumu, kuris yra proporcingas vertikaliam atstumui nuo kurio nors atskaitos taško. Paeiliui padarant tris šlyties deformacijas, galima puikiai pasukti bet kokį vaizdą bet kokiu kampu. [12,13]

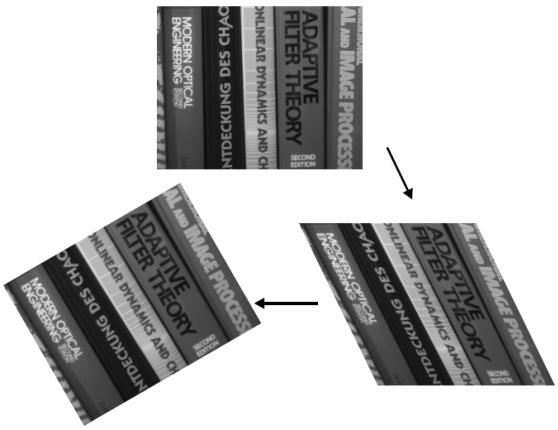
Tipinis vaizdo pasukimas iš taškų (x_1, y_1) į (x_2, y_2) kampu θ yra padaromas padauginant taškų vektorių iš pasukimo matricos:

$$R(\Theta) = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix}$$

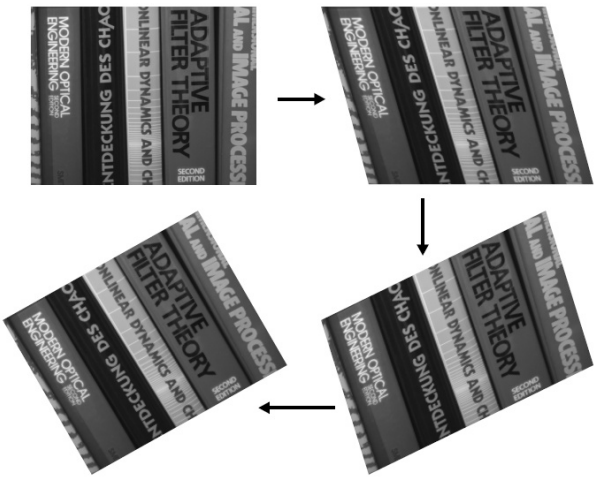
Vaizdo pasukimą su šlyties deformacija galima aprašyti taip:

$$R(\Theta) = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} = \begin{bmatrix} 1 & -\tan\frac{\Theta}{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\Theta & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\frac{\Theta}{2} \\ 0 & 1 \end{bmatrix}. \text{ [14]}$$

Šie algoritmai yra taikomi vaizdo sukimui, kai sukimo kampas yra $|\theta| < 45^o$. Dirbant su didesniais sukimo kampais turi buti pritaikyti 90^o arba 180^o pasukimai, kol pagaliau sukimo kampas lieka mažesnis nei 45^o. [15, 16, 17]



2.7 pav. Vaizdo pasukimas su šlyties ir dydžio keitimo deformacijomis.

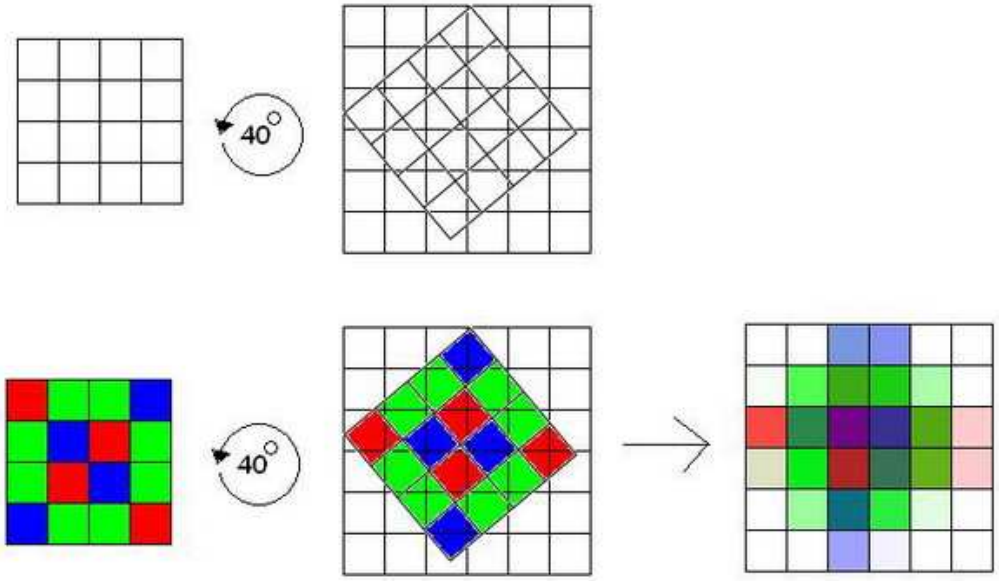


2.8 pav. Vaizdo pasukimas su šlyties deformacija.

2.2.4 Vaizdų pasukimas su geometrinėmis operacijomis

Straipsnyje [18] aprašomas algoritmas, kuris remiasi pikselių verčių apskaičiavimu remiantis geometrinėmis operacijomis. Šis algoritmas perdengia pradinų vaizdą and galutinio vaizdo kuriuo nors kampu. Persidengiančių pikselių vertėms pagal persidengimo plotą suteikiami svertiniai koeficientai. Rezultato pikselio vertė yra visų per tą pikselį persidengiančių pradinio vaizdo pikselių aritmetinis vidurkis.

2.9 paveikslėlis demonstruoja algoritmo veikimą. Kairėje yra pradinis vaizdas. Viduryje, pradinis vaizdas yra pasuktas 40 laipsnių kampu ir užkeltas ant galutinio vaizdo. Tuomet algoritmas randa persidengiantį plotą kiekvienam pradinio vaizdo pikseliui į kiekvieną galutinio vaizdo pikselį. Jis naudoja šias ploto vertes pasverti, kiek spalvų informacijos iš kiekvieno pradinio vaizdo pikselio priklauso kiekvienam paskirties taškui.



2.9 pav. Vaizdo pasukimas su šlyties deformacija.

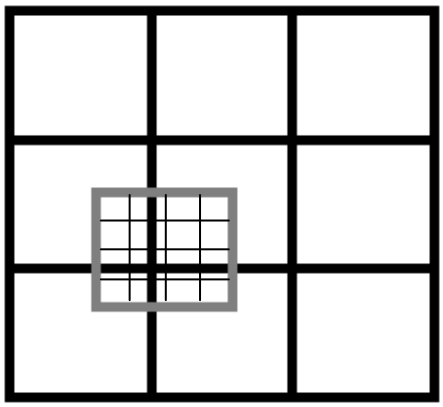
Šio algoitmo didelis trūkumas yra tai, jog reikia atlikti labai daug geometrinių operacijų, kas labai padidina algoritmo darbo laiką.

2.2.4 Siūlomas algoritmas

Kadangi pasukimo algoritmas su geometrinėmis operacijomis dirbdamas atlieka daug geometrių, tai jo darbo laikas yra didelis, kas trukdo šio algoritmo pritaikymui darbo srityse, kur reikalingas greitas užduočių sprendimas. Šiame darbe siūlomas naujas pikselių persidengenčio ploto skaičiavimo algoritmas.

Persidengiančių pikselių vertės skaičiuojamos taip: interpoliacijos metu ieškomas pikselis padalijamas į N x N subpikselių. Kiekvieno subpikselio vertė sudaro vienodą kiekį pačio pikselio vertės. Skaičiuojant subpikselio įvertį, naudojama artimiausio kaimyno interpoliaciją. Yra eksperimentiškai įrodyta, jog artimiausio kaimyno interpoliacijos algoritmas nėra labai sudėtingas ir turi labai mažą darbo laiką, todėl jis yra tinkamas šiam atvejui, kai reikia jį naudoti daug kartų. Radus visus subpikselių įverčius yra gaunamas jų aritmetinis vidurkis. Jei N = 1, tai algoritmo veikimas yra beveik identiškas su tipinio algoritmo su artimiausio kaimyno interpoliacija veikimu.

Galima teigti, jog pasirinkus vis didesnę N reikšmę, skaičiavimų tikslumas vis didės. Kuo pasirenkama didesnė N reikšmė, tuo tiksliau dirba algoritmas, bet padidėja atliekmų operacijų skaičius, kas padidina darbo laiką. Ši hipotezė bus ištirta tolimesniuose eksperimentuose.



2.10 pav. Ieškomas pikselis padalijamas į 4x4 subpikselius

Pikslelių įverčiai paskaičiuojami pagal formulę:

$$P(u,v) = \frac{1}{\mathrm{n}\cdot\mathrm{n}}\sum_{\mathrm{i}=0,\mathrm{j}=0}^{\mathrm{n}-1,\mathrm{n}-1} \mathrm{X}[\mathrm{u}+\frac{\mathrm{i}}{\mathrm{n}},\mathrm{v}+\frac{\mathrm{j}}{\mathrm{n}}]$$

čia:P(u,v) – pikselio vertė koordinatėse (u,v), X[u,v] – vaizdas, n – pikselių padalijimo koeficientas.

3. Eksperimentų planavimas

3.1 Įrankių pasirinkimo analizė

Iš didelės programavimo kalbos įvairovės buvo pasirinkta viena iš populiariausių kalbų, žinoma kaip objektinio programavimo kalba - C++. Šios programavimo kalbos pagrindai yra įgyti studijavimo laikotarpiu, tai svariausia priežastis, kodėl būtent ši kalba naudojama darbe. Jai yra sukurta nemažai įvairių bibliotekų, palengvinančių programų kūrimą.

Išsirinkus programavimo kalbą, sekantis tikslas - išsirinkti įrankį, kuriuo bus kuriama vartotojo grafinė sąsaja. Vartotojo grafinei aplinkai kurti skirto įrankio pasirinkimui buvo iškelti keli reikalavimai pagal kuriuos ir buvo pasirinktas įrankis:

1. Turi turėti komponentą, kuris palengvintų darbą su skaitmeniniais vaizdais.
2. Nesudėtingas objektų kūrimas.
3. Turi būti nemokamas.

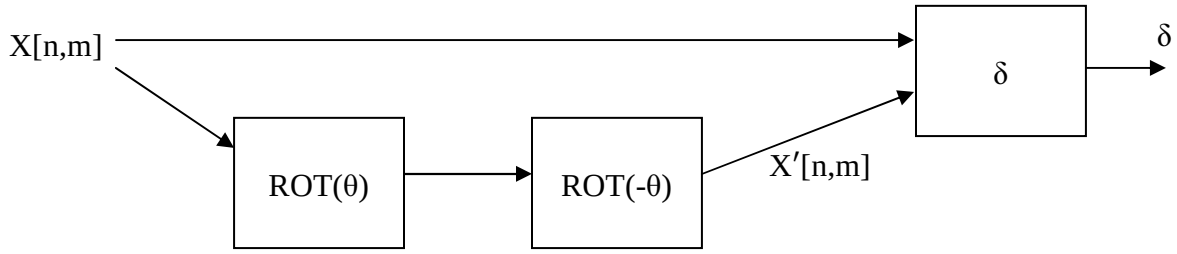
Išanalizavus iškeltus reikalavimus buvo nuspręsta naudoti QT programų kūrimo įrankį. Jis atitiko visus keliamus reikalavimus. QT karkase yra realizuotos klasės padedančios apdoroti skaitmeninius vaizdus, o tai pateisina keliamo pagrindinio tikslo reikalavimus. QT karkasas yra atviro kodo programa, t.y. ją galima parsisiųsti nemokamai.

Pasirinkimo alternatyvos: C++ Builder, Microsoft Visual C++ ir integruotos directx bibliotekos.

3.2 Eksperimentų struktūrinės schemos

Šiame darbe siekiama eksperimentiškai ištirti, kaip kokybiškai veikia skirtingi vaizdų pasukimo algoritmai. Tuo tikslu atliekamos trys eksperimentų serijos: viena serija tirti vaizdo informacijos praradimą panaudojus algoritmą vieną kartą, antra serija tirti informacijos praradimą naudojant algoritmą keletą kartų, ir trečia – tirti laiko sanaudas, per kurias algoritmas atlieka darbą.

Atliekant pirmąją eksperimentų seriją vaizdas pasukamas tam tikru kampu, ir vėl atsukamas į pradinę padėtį. Gautas vaizdas apkarpomas, panaikinant atsiradusias nereikalingas išorinias pikselių vertes. Gauti vaizdo įverčiai lyginami su pradiniu vaizdu vidutinės kvadratinės paklaidos prasme. Eksperimento atlikimo schema pateikta tolimesniame paveikslėlyje. Čia: δ – vidutinė kvadratinė paklaida, ROT(θ) – vaizdo pasukimo algoritmas, θ – pasukimo kampas, X[n,m] – pradinis vaizdas.

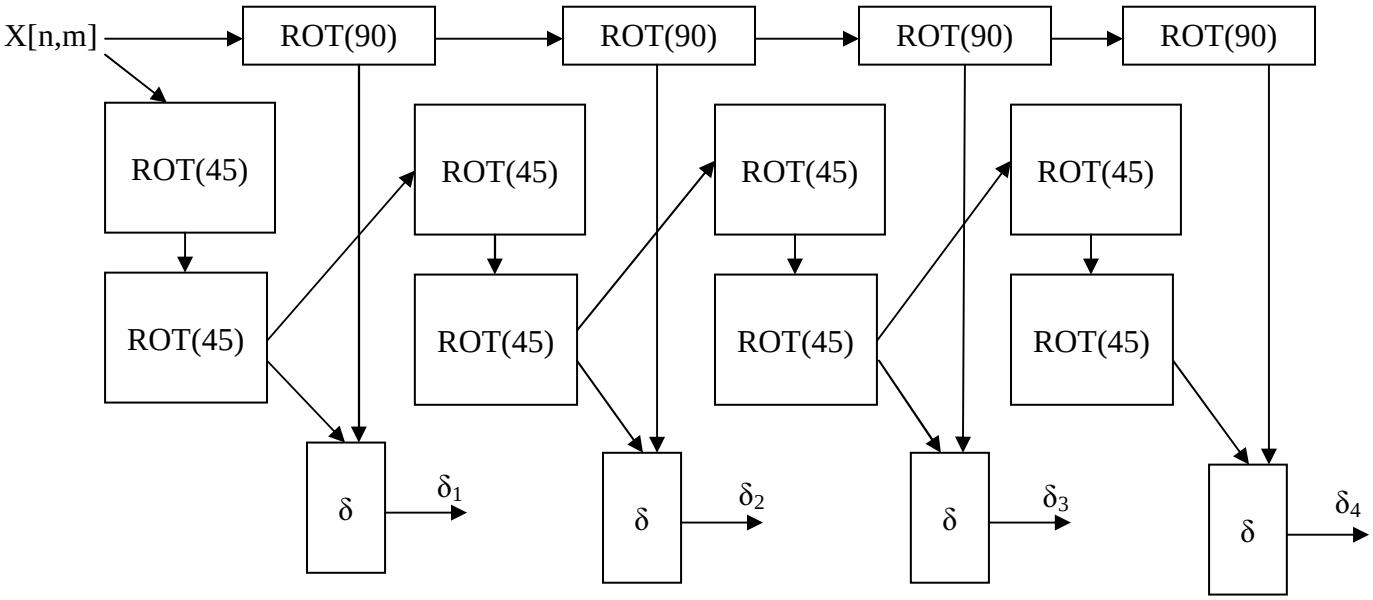


3.1 pav. Eksperimento atlikimo schema (pirmas eksperimentas).

Vidutinė kvadratinė paklaida apskaičiuojama pagal formulę:

$$\boldsymbol{\delta = \sqrt{\frac{\sum_{i=0,j=0}^{n,m}(X[i,j] - X'[i,j])^2}{n \cdot m}}}$$

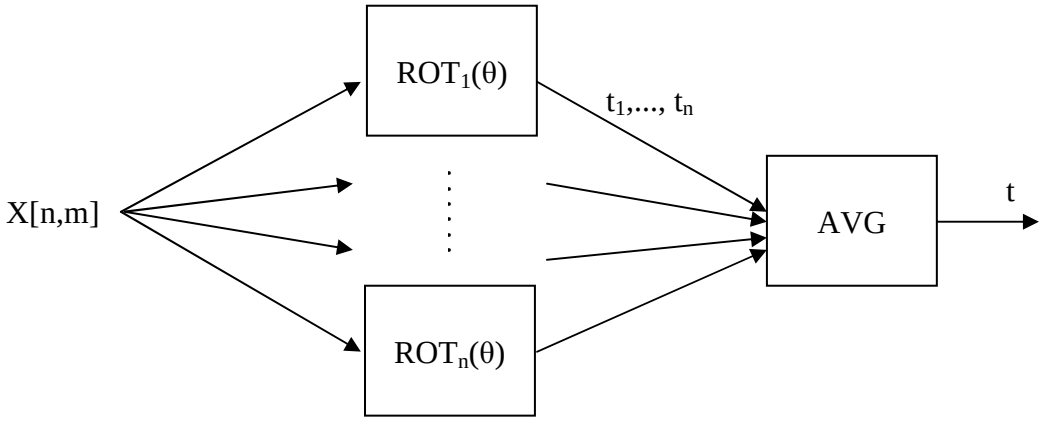
Panašiai atliekamas ir antras eksperimentas. Šį kartą tiriamas vaizdo informacijos praradimas naudojant pasukimo algoritmą keletą kartų. Vidutinė kvadratinė paklaida tikrinama po dviejų, keturių, šešių ir aštuonių pasukimų. Pasukto vaizdo įverčiai lyginami su pradiniu vaizdu ir su vaizdu pasuktu be informacijos praradimo 90^o kampu. Eksperimento atlikimo schema pateikta tolimesniame paveikslėlyje.



3.2 pav. Eksperimento atlikimo schema (antras eksperimentas).

Trečiuoju eksperimentu tiriamas algoritmų darbo laikas. Pradinis vaizdas pasukamas n kartų. Pamatuojamas kiekvieno pasukimo atlikimo laikas ir suskaičiuojamas rezultatų vidurkis. Eksperimento atlikimo schema pateikta tolimesniame paveikslėlyje. Čia: AVG – vidurkio skaičiavimas, t – laikas milisekundėmis. Galutinis laikas t apskaičiuojamas pagal formulę:

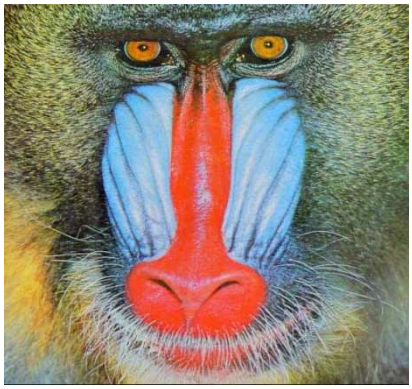
$$t = \frac{1}{n} \cdot \sum_{i=0}^{n} t_i$$



3.3 pav. Eksperimento atlikimo schema (trečias eksperimentas).

3.3 Tyrimuose naudojami duomenys

Dvimačiai skaitmeniniai vaizdai – tai skaitmeninės nuotraukos, brėžiniai ir kiti dvimačiai grafiniai vaizdai. Eksperimentiniam tyrimui parinktos nuotraukos iš USC-SIPI vaizdų duomenų bazės. Tyrimui panaudoti šeši vaizdai (dimensijos: 512x512):



(a)

(d)

(b)

(e)

(c)

(f)

3.4 pav. Eksperimentui panaudoti dvimačiai vaizdai (a) – (f): „Img01“ – „Img06“

4. Eksperimentinis vaizdo pasukimo algoritmų tyrimas

4.1 Informacijos praradimo tyrimas

Pradiniai skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.1 pav. pateiktą eksperimento schemą. Naudojamas 15, 30 ir 45 laipsnių pasukimas, gautas rezultato vaizdas pasukamas į pradinę padėtį ir palyginamas su pradiniu vazidu. Iš visų trijų pasukimų rezultatų išvedamas aritmetinis vidurkis, taip gaunant algoritmo įverčius. Eksperimento rezultatai pateikti tolimesnėse lentelėse.

Algoritmai:

1 Paprastas algoritmas su artimiausio kaimyno interpoliacija
2 Paprastas algoritmas su tiesine interpoliacija
3 Paprastas algoritmas su bikubine interpoliacija
4 Pasukimas su vienmatėmis šlyties transformacijomis
5 Vaizdo pasukimas su geometrinėmis operacijomis

4.1 lentelė. Vidutinės kvadratinės paklaidos priklausomybė nuo algoritmo sukant 15^okampu

Pav.\ Alg.	1	2	3	4	5
Img01	28.013	15.552	20.030	30.400	10.216
Img02	13.905	9.804	12.872	16.795	5.412
Img03	13.852	10.771	13.675	16.762	5.34
Img04	11.734	8.613	10.872	14.321	5.235
Img05	13.008	10.303	13.440	17.846	6.388
Img06	13.140	10.427	13.091	16.207	5.362
Vidurkis:	**15.609**	**10.912**	**13.997**	**18.722**	**6.326**

Lentelėje 4.1 surašyti visų penkių algoritmų rezultatai sukant vaizdus 15 laipsnių kampu. Geriausi rezultatai gaunami naudojant vaizdo pasukimą su geometrinėmis operacijomis. Truputį blogesni – naudojant tipinį algoritmą su bikubine interpoliacija. Naudojant vaizdo pasukimą su geometrinėmis operacijomis, gaunami 43% geresni rezultatai nei naudojant tipinį algoritmą su bikubine interpoliacija. Patys blogiausi rezultatai gaunami naudojant pasukimą su vienmatėmis šlyties transformacijomis. Šio algoritmo rezultatai beveik tris kartus blogesni už geriausius rezultatus.

4.2 lentelė. Vidutinės kvadratinės paklaidos priklausomybė nuo algoritmo sukant 30^okampu

Pav.\ Alg.	1	2	3	4	5
Img01	26.833	15.514	18.850	24.694	11.295
Img02	14.169	10.201	11.799	13.213	5.876
Img03	14.399	10.791	12.201	13.883	5.562
Img04	13.323	8.866	10.136	13.065	5.229
Img05	17.237	10.633	12.393	17.093	6.142
Img06	14.200	9.694	11.470	13.461	5.459
Vidurkis:	**16.694**	**10.950**	**12.808**	**15.902**	**6.594**

Lentelėje 4.2 surašyti visų penkių algoritmų rezultatai sukant vaizdus 30 laipsnių kampu. Geriausi rezultatai gaunami naudojant vaizdo pasukimą su geometrinėmis operacijomis. Truputį blogesni – naudojant tipinį algoritmą su bikubine interpoliacija. Patys blogiausi rezultatai gaunami naudojant pasukimą su vienmatėmis šlyties transformacijomis. Visų algoritmų rezultatų kokybė blogesnė, nei buvo sukant 15 laipsnių kampu.

4.3 lentelė. Vidutinės kvadratinės paklaidos priklausomybė nuo algoritmo sukant 45^okampu

Pav.\ Alg.	1	2	3	4	5
Img01	28.502	19.063	22.138	30.880	10.741
Img02	14.350	12.419	14.412	17.341	6.827
Img03	14.113	13.743	15.801	17.648	6.378
Img04	12.199	10.503	12.282	15.172	5.855
Img05	13.768	12.342	15.046	19.488	7.282
Img06	13.459	12.474	14.397	16.954	5.818
Vidurkis:	**16.065**	**13.424**	**15.679**	**19.581**	**7.150**

Lentelėje 4.3 surašyti visų penkių algoritmų rezultatai sukant vaizdus 45 laipsnių kampu. Geriausi rezultatai gaunami naudojant vaizdo pasukimą su geometrinėmis operacijomis. Truputį blogesni – naudojant tipinį algoritmą su bikubine interpoliacija. Patys blogiausi rezultatai gaunami naudojant pasukimą su vienmatėmis šlyties transformacijomis. Beveik visų algoritmų rezultatų kokybė blogesnė, nei buvo sukant 15 ir 30 laipsnių kampais. Tipinio algoritmo su artimiausio kaimyno interpoliacija rezultatų kokybės įvertis truputį padidėjo nei sukant 30 laipsnių kampu.

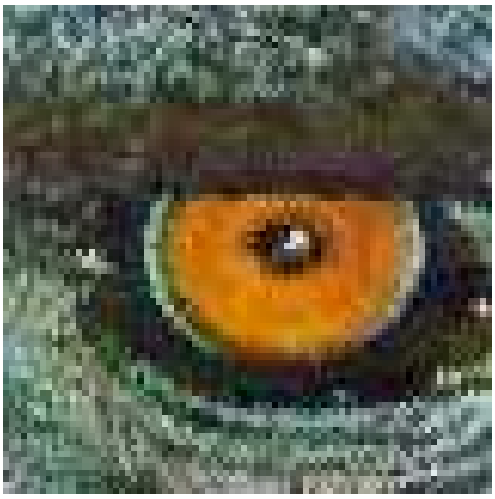
4.4 lentelė. Vidutinės kvadratinės paklaidos priklausomybė nuo algoritmo

Pav.\ Alg.	1	2	3	4	5
Img01	27.783	16.710	20.339	28.658	10.751
Img02	14.141	10.808	13.028	15.783	6.038
Img03	14.121	11.768	13.892	16.098	5.760
Img04	12.419	9.327	11.097	14.186	5.440
Img05	14.671	11.093	13.626	18.142	6.604
Img06	13.600	10.865	12.986	15.541	5.546
Vidurkis:	**16.122**	**11.762**	**14.161**	**18.068**	**6.690**

Lentelėje 4.4 surašyti visų penkių algoritmų rezultatų, sukant 15, 30 ir 45 laipsnių kampais aritmetiniai vidurkiai. Galutiniai įverčių vidurkiai toliau naudojami kaip tyrimo rezultatų įvertinimas.

Palyginus gautų rezultatų vidutines kvadratines paklaidas δ, galima teigti, jog efektyviausias algoritmas, sukant vaizdą vieną kartą yra pasukimas su geometrinėmis operacijomis. Šiuo algoritmu gauti rezultatai yra beveik 50% geresni nei kitais algoritmais gauti rezultatai. Mažiausiai efektyvus (prarandama daugiausiai vaizdo informacijos) – tipinis pasukimas su artimiausio kaimyno interpoliacija ir pasukimas su vienmatėmis šlyties transformacijomis.

(a)



(b)

(c)

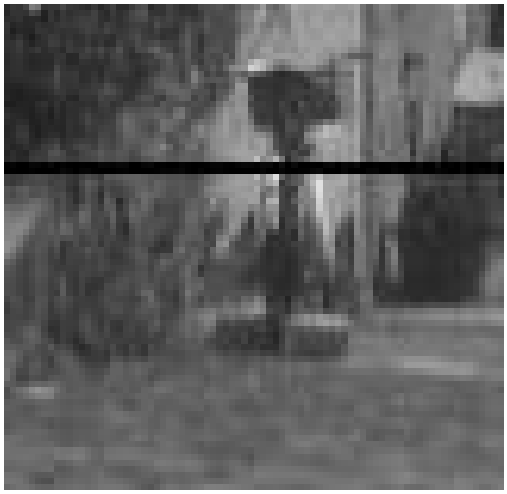
(e)

(d)

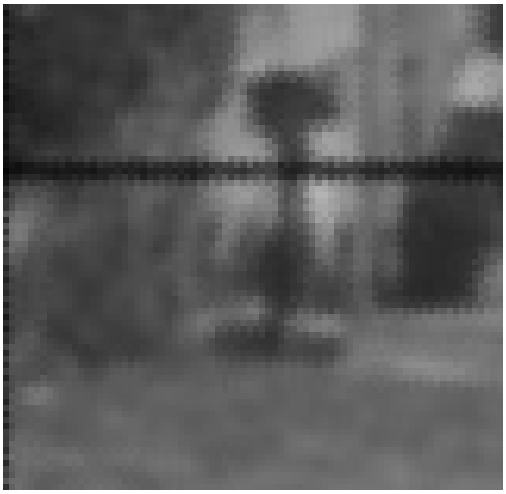
(f)

4.1 pav. Pirmojo eksperimento Img01 paveikslėlio rezultato fragmentai sukant 30^{o} kampu. (a) – pradinis vaizdas, (b – f) – skirtinigų algoritmų rezultatai.

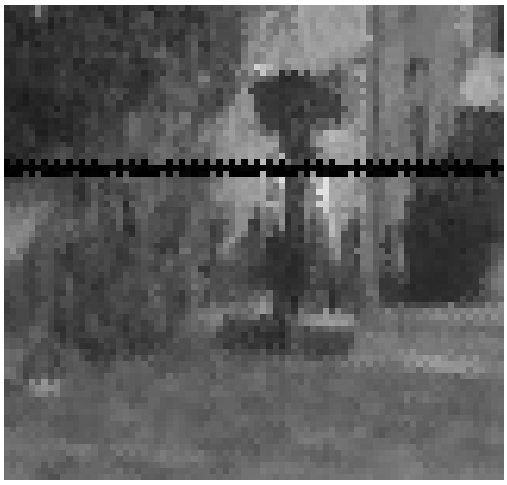
4.1 pav vaizduojami pirmojo tyrimo rezultatai, tiriant visus penkis pagrindinius algoritmus. Naudojamas vaizdas „Img01“. Iš (b) ir (e) vaizdų fragmentų matoma, jog tipinio su artimiausio kaimyno interpoliacija ir pasukimo su vienmatėmis šlyties deformacijomis algoritmų atvejais būdinga ryški pikselizacija didelio kontrasto zonose. Tipinių algoritmų su tiesine ir bikubine interpoliacijų atvejais vaizdams būdingas išsiliejimo efektas. Vaizdo pasukimo su geometrinėmis operacijomis algoritmo rezultate gautas vaizdas yra labai panašus į pradinį, bet yra matomas labai nežymus išsiliejimo efektas.



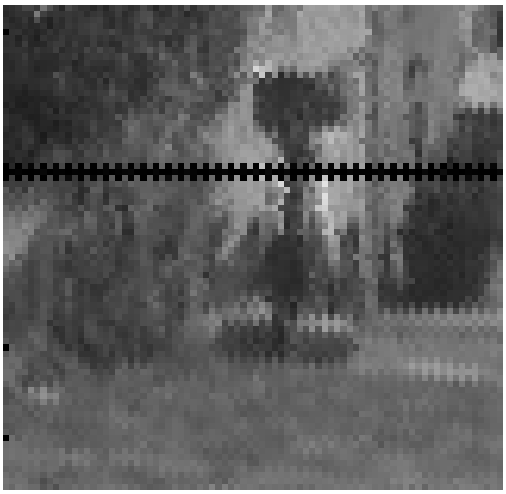
(a)



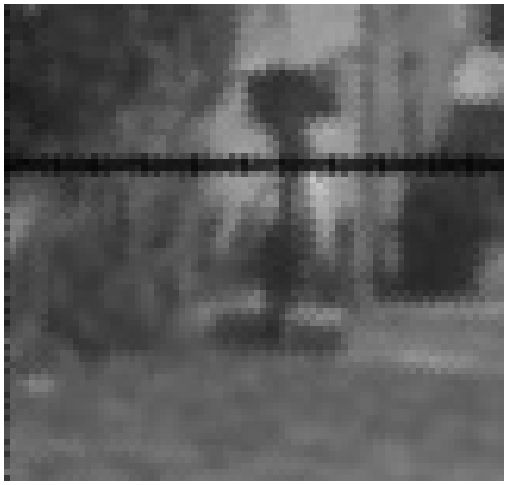
(d)



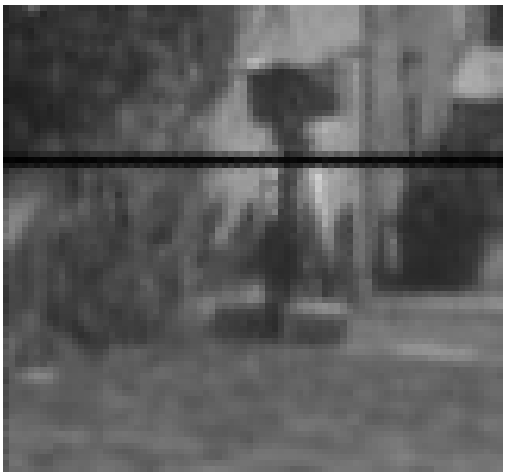
(b)



(e)



(c)



(f)

4.2 pav. Pirmojo eksperimento Img06 paveikslėlio rezultato fragmentai sukant 30º kampu. (a) – pradinis vaizdas, (b – f) – skirtinigų algoritmų rezultatai.

4.2 pav vaizduojami pirmojo tyrimo rezultatai, tiriant visus penkis pagrindinius algoritmus. Iš (b) ir (e) vaizdų fragmentų matoma, jog tipinio su artimiausio kaimyno interpoliacija ir pasukimo su vienmatėmis šlyties deformacijomis algoritmų atvejais būdinga ryški pikselizacija didelio kontrasto zonose. Tipinių algoritmų su tiesine ir bikubine interpoliacijų atvejais vaizdams būdingas išsiliejimo efektas. Vaizdo pasukimo su geometrinėmis operacijomis algoritmo rezultate gautas vaizdas yra labai panašus į pradinį, bet yra matomas labai nežymus išsiliejimo efektas.

4.2 Informacijos praradimo tyrimas siūlomame algoritme

Skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.1 pav. pateiktą eksperimento schemą. Naudojamas 30 laipsnių pasukimas ir gautas rezultato vaizdas pasukamas į pradinę padėtį. Rezultatas yra „apkarpomas“ ir palyginamas su pradiniu vazidu. Eksperimentas buvo papildytas galimybe pasirinkti pikselių suskaidymo dydį N. Šio eksperimento metu naudojamos N reikšmės, keičiamos nuo 1 iki 8. Eksperimento rezultatai pateikti tolimesnėje lentelėje.

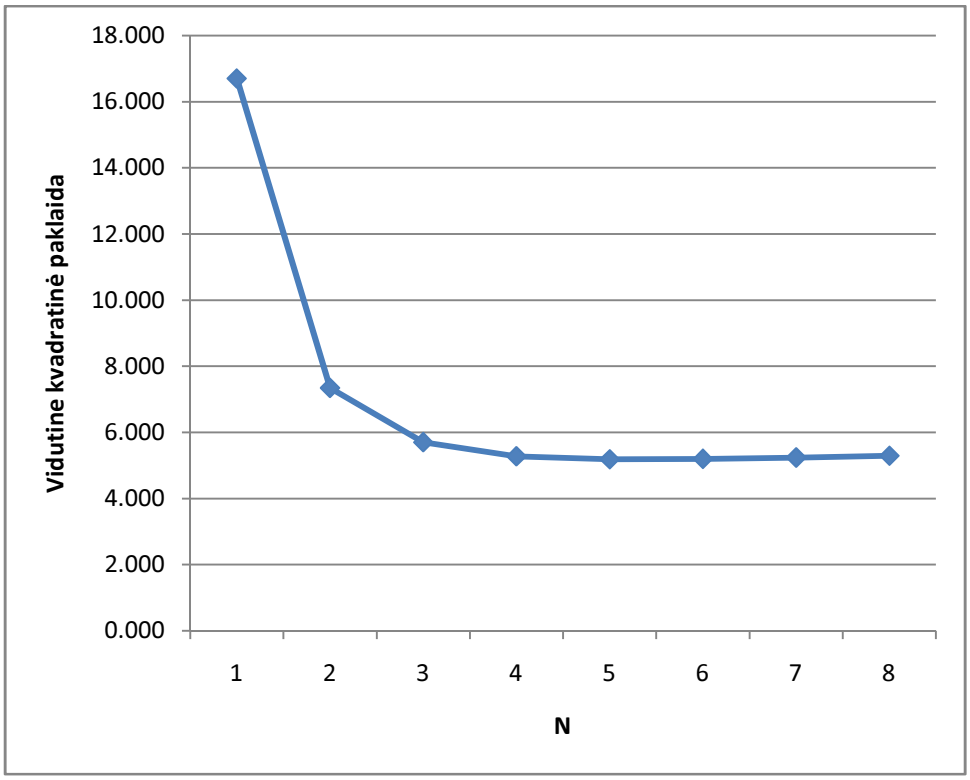
4.5 lentelė. Vidutinės kvadratinės paklaidos priklausomybė nuo N

Pav.\N	1	2	3	4	5	6	7	8
Img01	26.833	12.498	10.131	9.480	9.294	9.255	9.264	9.304
Img02	14.169	6.046	4.548	4.106	3.989	3.968	3.991	4.020
Img03	14.399	6.192	4.631	4.212	4.107	4.111	4.125	4.182
Img04	13.323	5.931	4.612	4.305	4.273	4.304	4.360	4.415
Img05	17.237	7.245	5.493	5.121	5.109	5.190	5.294	5.389
Img06	14.200	6.138	4.792	4.428	4.356	4.366	4.406	4.449
Vidurkis:	**16.694**	**7.341**	**5.701**	**5.275**	**5.188**	**5.199**	**5.240**	**5.293**

Lentelėje 4.5 surašyti siūlomo algoritmo tyrimo rezultatai sukant vaizdus 30 laipsnių kampu. Tyrimas darytas keičiant pkselių suskaidymo koeficientą N keičiant nuo 1 iki 9. Kai N yra lygus 1, rezultatai gaunami labai panašūs į pirmajame eksperimente gautus rezultatus, dirbant su tipiniu algoritmu ir artimiausio kaimyno interpoliacija. N padidinus iki 2, daugiau ne 50% padidėja rezultatų kokybė. N padidinus nuo 2 iki 3, kokybė padidėja tik 22%. Didinant N, kokybė įvertis mažėja ir nusistovi apie vieną reikšmę.

Hipotezė, jog vis didinant N algoritmas tiksliau dirba, nepasitvirtino. Didinant N vidutinė kvadratinė paklaida pamažu nusistovi ties viena reikšme. Iš tyrimų rezultatų matoma, jog norint gauti geriausią rezultatų kokybę, N koeficientas turi būti lygus 4.

4.3 pav. vaizduojamas vidutinės kvadratinės paklaidos priklausomybės nuo N grafikas. Jo pagalba galima gerai matyti vidutinės kvadratinės paklaidos kaitą.



4.3 pav. Vidutinės kvadratinės paklaidos priklausomybė nuo N.

(a)

(b)

(c)

(f)

(d)

(g)

(e)

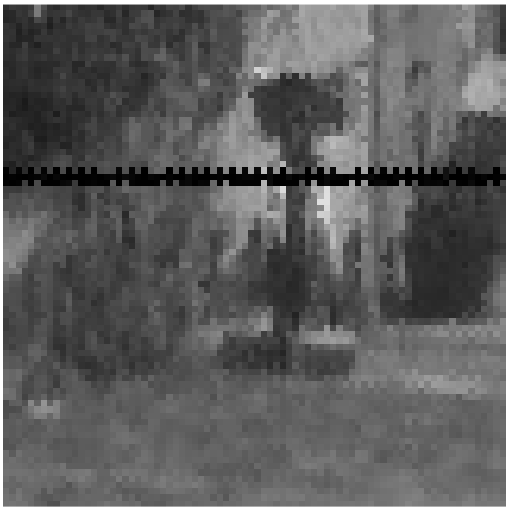
(h)

(i)

4.4 pav. Img01 fragmentai su skirtingomis N reikšmėmis sukant 30^{o} kampu. (a) – pradinio vaizdo fragmentas, (b – i) - rezultato fragmentai kai N keičiasi nuo 1 iki 5.

4.4 pav. vaizduojami tyrimo rezultatai naudojant siūlomą algoritmą, pasukant vaizdus 30 laipsnių kampu ir priskiriant skirtingus N koeficientus. (b) fragmente matomas žymus pikselizacijos efektas. Didinant N rekšmę, pikselizacijos efektas visiškai išnyksta, bet atsiranda nežymus išsiliejimo efektas. (c-i) vaizdų fragmentuose nėra akimi įžiūrimų skirtumų.

(a)



(b)

(c)

(f)

(d)

(g)

(e)

(h)

(i)

4.5 pav. Img06 fragmentai su skirtingomis N reikšmėmis sukant 30^o kampu. (a) – pradinio vaizdo fragmentas, (b – i) - rezultato fragmentai kai N keičiasi nuo 1 iki 5.

4.5 pav. vaizduojami tyrimo rezultatai naudojant siūlomą algoritmą, pasukant vaizdus 30 laipsnių kampu ir priskiriant skirtingus N koeficientus. (b) fragmente matomas žymus pikselizacijos efektas. Didinant N rekšmę, pikselizacijos efektas pamažu išnyksta, bet atsiranda nežymus išsiliejimo efektas. (g-i) vaizdų fragmentuose nėra akimi įžiūrimų skirtumų.

4.3 Informacijos praradimo per kelis pasukimus tyrimas

Pradiniai skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.2 pav. pateiktą eksperimento schemą. Vaizdai sukami 45^o kampu ir kas antras pasukimas skaičiuojama vidutinė kvadratinė paklaida. Iš viso naudojami aštuoni pasukimai. Eksperimento rezultatai pateikti tolimesnėse lentelėje.

4.6 lentelė. Tipinis algoritmas su artimiausio kaimyno interpoliacija

Pav.\Aps.	2	4	6	8
Img01	12.664	15.188	16.576	17.722
Img02	6.092	7.599	8.440	9.013
Img03	6.423	7.562	7.872	8.593
Img04	6.296	7.356	8.107	8.701
Img05	7.896	9.356	9.983	10.911
Img06	6.318	7.552	8.390	9.056
Vidurkis:	**7.615**	**9.102**	**9.895**	**10.666**

4.6 lentelėje surašyti tyrimo rezultatai naudojant tipinį algortima su artimiausio kaimyno interpoliacija. Vidutinės kvadratinės paklaidos įverčiai gana žymiai pablogėja vaizdą pasukus 4 kartus. Dvigubai mažiau kokybė pablogėja pasukus kitus du pasukimus.

4.7 lentelė. Tipinis algoritmas su tiesine interpoliacija

Pav.\Aps.	2	4	6	8
Img01	18.597	21.106	21.736	22.357
Img02	11.250	15.550	16.535	15.477
Img03	13.123	16.631	17.761	17.842
Img04	10.646	12.913	13.599	14.085
Img05	12.621	15.665	16.300	16.985
Img06	10.694	13.931	16.082	16.698
Vidurkis:	**12.822**	**15.966**	**17.002**	**17.241**

4.7 lentelėje surašyti tyrimo rezultatai naudojant tipinį algortima su tiesine interpoliacija. Vidutinės kvadratinės paklaidos įverčiai gana žymiai pablogėja vaizdą pasukus 4 ir 6 kartus. Labai mažai kokybė pablogėja pasukus kitus du pasukimus.

4.8 lentelė. Tipinis algoritmas su bikubine interpoliacija

Pav.\Aps.	2	4	6	8
Img01	18.494	21.631	23.361	24.639
Img02	10.429	14.810	17.444	18.250
Img03	12.274	16.308	18.898	20.362
Img04	10.624	12.862	14.454	15.689
Img05	12.651	15.714	17.606	19.162
Img06	10.580	13.616	16.227	18.160
Vidurkis:	**12.509**	**15.823**	**17.998**	**19.377**

4.8 lentelėje surašyti tyrimo rezultatai naudojant tipinį algortima su bikubine interpoliacija. Vidutinės kvadratinės paklaidos įverčiai keičiasi gana tolygiai, lyginant su kitų algoritmų rezultatais.

4.9 lentelė. Pasukimas su vienmatėmis šlyties transformacijomis

Pav.\Aps.	2	4	6	8
Img01	26.368	31.314	31.109	23.953
Img02	14.128	18.129	18.127	15.024
Img03	15.722	19.821	19.330	15.745
Img04	14.635	16.852	15.347	12.957
Img05	18.445	21.579	19.032	16.019
Img06	14.235	17.101	16.268	14.051
Vidurkis:	**17.255**	**20.799**	**19.869**	**16.292**

4.9 lentelėje surašyti tyrimo rezultatai naudojant pasukimą su vienmatėmis šlyties transformacijomis. Vidutinės kvadratinės paklaidos įverčiai labai įvairiai keičiasi. Po 4 pasukimų, kokybė žymiai pablogėja, bet dar po 2 pasukimų, kokybės įvertis pamažu pradeda mažėti. Po paskutinių 2 pasukimų, kai vaizdas atsukamas į pradinę padėtį, vidutinė kvadratinė paklaida dar sumažėja.

4.10 lentelė. Pasukimas su geometrinėmis operacijomis

Pav.\Aps.	2	4	6	8
Img01	12.111	15.222	16.189	17.281
Img02	6.815	8.365	8.686	10.355
Img03	6.485	8.834	9.242	10.46
Img04	5.579	7.937	8.569	9.105
Img05	6.536	9.574	10.321	10.967
Img06	6.749	8.854	9.008	9.786
Vidurkis:	**7.379**	**9.798**	**10.336**	**11.326**

4.10 lentelėje surašyti tyrimo rezultatai naudojant pasukimą su geometrinėmis operacijomis. Vidutinės kvadratinės paklaidos įverčiai gana tolygiai didėja visų pasukimų metu.

4.11 lentelė. VKP priklausomybė nuo algoritmo ir pasukimų skaičiaus

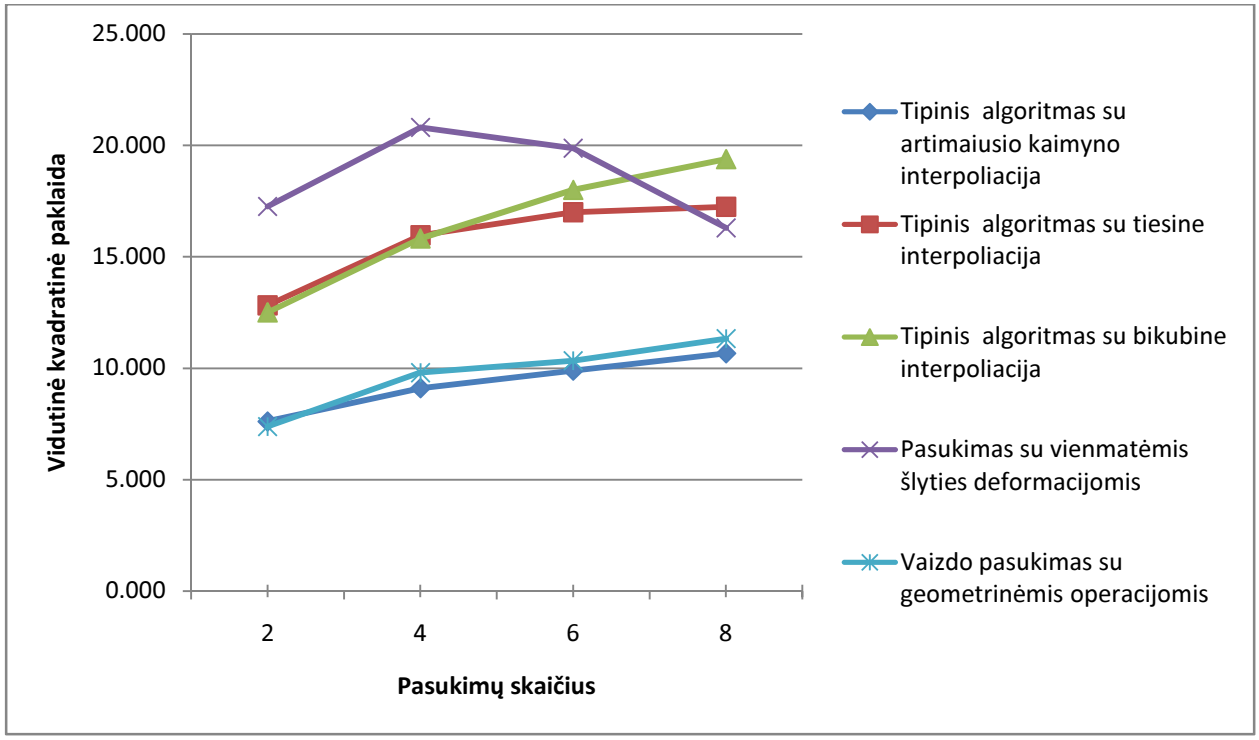
Alg.\Aps.	2	4	6	8
1	7.615	9.102	9.895	10.666
2	12.822	15.966	17.002	17.241
3	12.509	15.823	17.998	19.377
4	17.255	20.799	19.869	16.292
5	7.379	9.798	10.336	11.326

Algoritmai:

1 Paprastas algoritmas su artimiausio kaimyno interpoliacija
2 Paprastas algoritmas su tiesine interpoliacija
3 Paprastas algoritmas su bikubine interpoliacija
4 Pasukimas su vienmatėmis šlyties transformacijomis
5 Vaizdo pasukimas su geometrinėmis operacijomis

Lentelėje 4.4 surašyti visų penkių algoritmų rezultatų, sukant 30 laipsnių kampais aritmetiniai vidurkiai. Galutiniai įverčių vidurkiai toliau naudojami kaip tyrimo rezultatų įvertinimas.

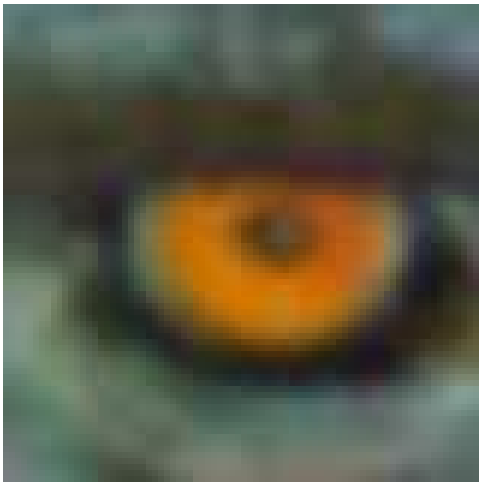
Palyginus gautų rezultatų vidutines kvadratines paklaidas δ, galima teigti, jog efektyviausi algoritmai, sukant vaizdą keletą kartų yra tipinis pasukimas su artimiausio kaimyno interpoliacija i pasukimas su geometrinėmis operacijomis.. Mažiausiai efektyvūs (prarandama daugiausiai vaizdo informacijos) – tipinis pasukimas su bikubine interpoliacija ir pasukimas su vienmatėmis šlyties transformacijomis.



4.6 pav. Antrojo eksperimento rezultatai. VKP priklausomybė nuo pasukimų skaičiaus.

4.6 pav. pavaizduotas vidutinės kvadratinės paklaidos priklausomybės nuo pasukimų skaičiaus grafikas. Patys efektyviausi algoritmai yra tipinis algoritmas su artimiausio kaimyno interpoliacija ir pasukimas su geometrinėmis operacijomis. Šių dviejų algoritmų rezultatų įverčiai yra labai artimi vienas kito.

(a)



(d)

(b)

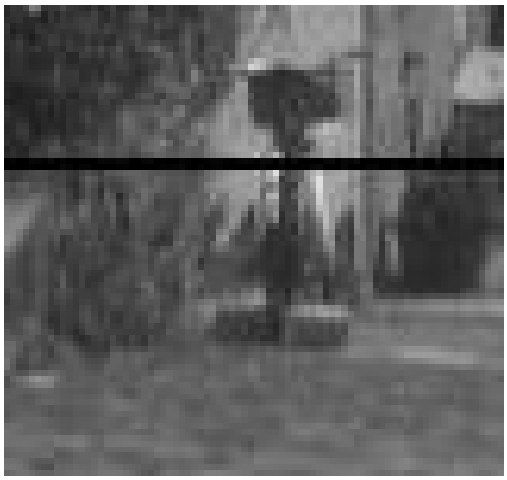
(e)

(c)

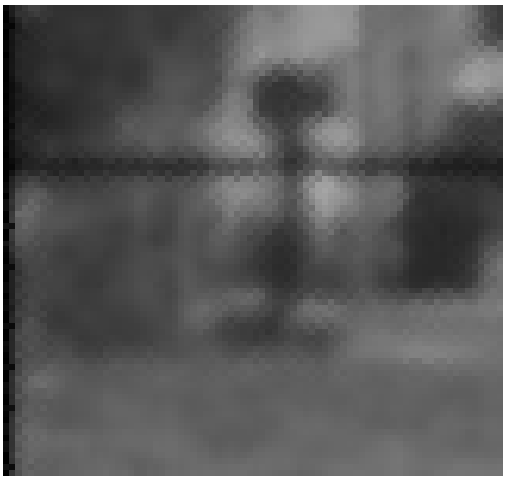
(f)

4.7 pav. Antrojo eksperimento Img01 paveikslėlio rezultato fragmentai. (a) – pradinis vaizdas, (b – f) – skirtinigų algoritmų rezultatai.

4.7 pav. vaziduojami tyrimo rezultatų „Img06“ vaizdo fragmentai. Paveikslėlyje pavaizduoti fragmentai yra iš rezultatų, gautų panaudojus 8 pasukimus. Iš rezultato vaizdų fragmentų matoma, jog rezultatams būdingos tokios pačios savybės, kokio buvo būdingos ir pirmojo eksperimento rezultatams. Tipinio su artimiausio kaimyno interpoliacija ir pasukimo su vienmatėmis šlyties deformacijomis algoritmų atvejais būdinga ryški pikselizacija didelio kontrasto zonose. Tipinių algoritmų su tiesine ir bikubine interpoliacijomis vaizdams būdingas išsiliejimo efektas. Taip pat pasukimo su geometrinėmis operacijomis vaizdui būdingas silpnas efektas.

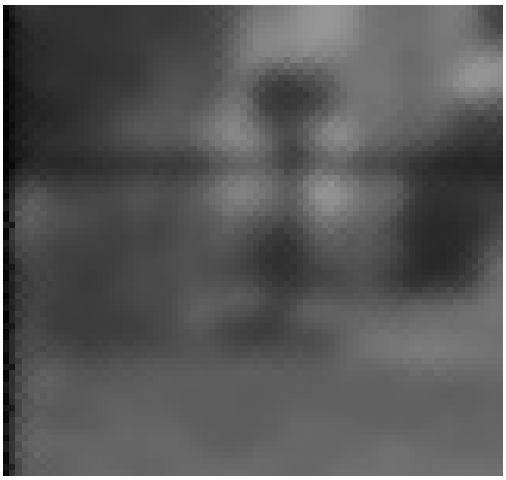


(a)



(c)

(b)



(d)

(e)

(f)

4.8 pav. Antrojo eksperimento Img06 paveikslėlio rezultato fragmentai. (a) – pradinis vaizdas, (b – f) – skirtinigų algoritmų rezultatai.

4.8 pav. vaziduojami tyrimo rezultatų „Img06“ vaizdo fragmentai. Paveikslėlyje pavaizduoti fragmentai yra iš rezultatų, gautų panaudojus 8 pasukimus. Iš rezultato vaizdų fragmentų matoma, jog rezultatams būdingos tokios pačios savybės, kokio buvo būdingos ir pirmojo eksperimento rezultatams. Tipinio su artimiausio kaimyno interpoliacija ir pasukimo su vienmatėmis šlyties deformacijomis algoritmų atvejais būdinga ryški pikselizacija didelio kontrasto zonose. Tipinių algoritmų su tiesine ir bikubine interpoliacijomis bei pasukimo su geometrinėmis operacijomis vaizdams būdingas išsiliejimo efektas.

4.4 Informacijos praradimo per kelis pasukimus siūlomame algoritme tyrimas

Pradiniai skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.2 pav. pateiktą eksperimento schemą. Vaizdai sukami 45^{o} kampu ir kas antras pasukimas skaičiuojama vidutinė kvadratinė paklaida. Šis eksperimentas atliekamas, kai pikselio suskaidymo dydis N yra keičiamas nuo 1 iki 5. Praeito eksperimento metu nustatyta, jog nėra prasmės atlikti eksperimento su didesniais N, nes vidutinės kvadratinės paklaidos įvertis nusistovi apie vieną reikšmę, kai N yra daugiau už 4. Šio eksperimento rezultatai pateikti tolimesnėse lentelėse.

4.12 lentelė. VKP priklausomybė nuo pasukimų skaičiaus, kai N = 1

Pav.\Aps.	2	4	6	8
Img01	12.664	15.188	16.576	17.722
Img02	6.092	7.599	8.440	9.013
Img03	6.423	7.562	7.872	8.593
Img04	6.296	7.356	8.107	8.701
Img05	7.896	9.356	9.983	10.911
Img06	6.318	7.552	8.390	9.056
Vidurkis:	**7.615**	**9.102**	**9.895**	**10.666**

4.12 lentelėje surašyti tyrimo rezultatai naudojant siūlomą algoritmą su pikselių suskaidymo koeficientu 1. Vidutinės kvadratinės paklaidos įverčiai sparčiai keičiasi po 4 pasukimų, truputį lėčiau po 6 ir 8 pasukimų.

4.13 lentelė. VKP priklausomybė nuo pasukimų skaičiaus, kai N = 2

Pav.\Aps.	2	4	6	8
Img01	11.278	15.968	16.471	17.086
Img02	5.553	7.947	8.196	8.477
Img03	6.094	8.700	8.895	9.206
Img04	5.986	7.326	7.666	8.127
Img05	9.822	10.895	10.264	10.132
Img06	5.729	7.702	8.046	8.475
Vidurkis:	**7.411**	**9.756**	**9.923**	**10.250**

4.13 lentelėje surašyti tyrimo rezultatai naudojant siūlomą algoritmą su pikselių suskaidymo koeficientu 2. Vidutinės kvadratinės paklaidos įverčiai sparčiai keičiasi po 4 pasukimų, daug lėčiau po 6 ir 8 pasukimų.

4.14 lentelė. VKP priklausomybė nuo pasukimų skaičiaus, kai N = 3

Pav.\Aps.	2	4	6	8
Img01	13.630	17.359	17.802	18.946
Img02	7.019	8.748	9.254	10.151
Img03	7.429	9.286	9.663	10.346
Img04	7.504	8.447	8.444	8.725
Img05	9.822	10.132	10.264	10.895
Img06	7.265	8.780	8.943	9.470
Vidurkis:	**8.778**	**10.459**	**10.728**	**11.422**

4.14 lentelėje surašyti tyrimo rezultatai naudojant siūlomą algoritmą su pikselių suskaidymo koeficientu 3. Vidutinės kvadratinės paklaidos įverčiai sparčiai keičiasi po 4 pasukimų, truputį lėčiau po 6 ir 8 pasukimų.

4.15 lentelė. VKP priklausomybė nuo pasukimų skaičiaus, kai N = 4

Pav.\Aps.	2	4	6	8
Img01	14.951	17.455	18.632	20.580
Img02	7.828	8.806	9.884	11.188
Img03	8.439	9.622	10.572	11.750
Img04	8.291	8.513	8.824	9.438
Img05	10.928	10.187	10.721	11.848
Img06	8.043	8.827	9.365	10.291
Vidurkis:	**9.747**	**10.568**	**11.333**	**12.516**

4.15 lentelėje surašyti tyrimo rezultatai naudojant siūlomą algoritmą su pikselių suskaidymo koeficientu 4. Vidutinės kvadratinės paklaidos įverčiai tolygiai keičiasi vykdant visus pasukimus.

4.16 lentelė. VKP priklausomybė nuo pasukimų skaičiaus, kai N = 5

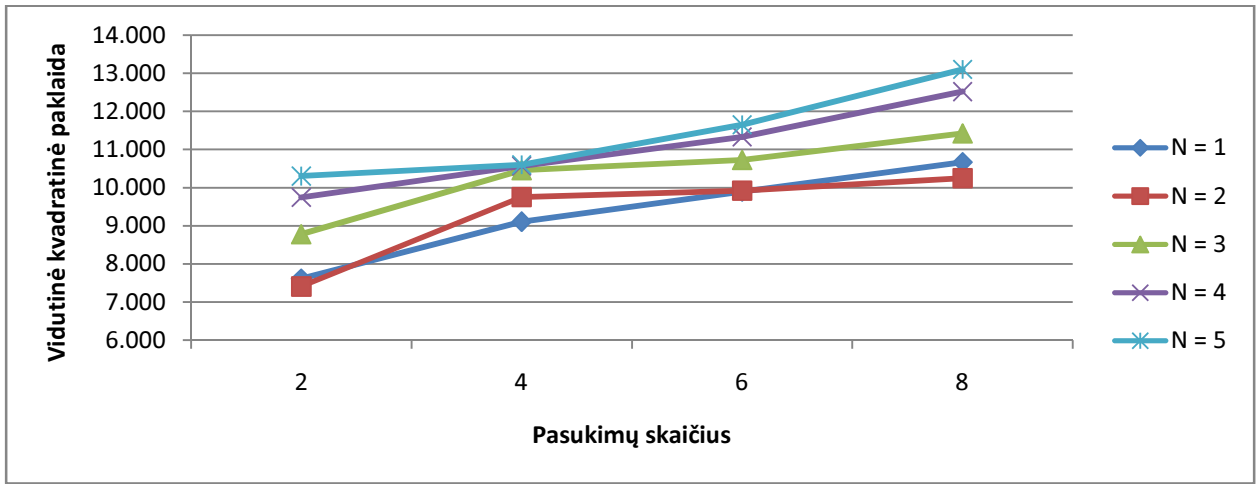
Pav.\Aps.	2	4	6	8
Img01	15.737	17.470	19.023	21.416
Img02	8.303	8.861	10.216	11.852
Img03	8.923	9.670	10.911	12.302
Img04	8.756	8.547	9.072	9.850
Img05	11.534	10.211	11.015	12.364
Img06	8.557	8.849	9.653	10.806
Vidurkis:	**10.302**	**10.601**	**11.648**	**13.098**

4.16 lentelėje surašyti tyrimo rezultatai naudojant siūlomą algoritmą su pikselių suskaidymo koeficientu 5. Vidutinės kvadratinės paklaidos letai keičiasi per pirmus 4 pasukimus, trigubai greičiau po 6 ir 8 pasukimų.

4.17 lentelė. VKP priklausomybė nuo N ir pasukimų skaičiaus

N\Aps.	2	4	6	8
1	7.615	9.102	9.895	10.666
2	7.411	9.756	9.923	10.250
3	8.778	10.459	10.728	11.422
4	9.747	10.568	11.333	12.516
5	10.302	10.601	11.648	13.098

Lentelėje 4.17 surašyti visų rezultatų, su skirtingomis N reikšmėmis, aritmetiniai vidurkiai. Galutiniai įverčių vidurkiai toliau naudojami kaip tyrimo rezultatų įvertinimas.

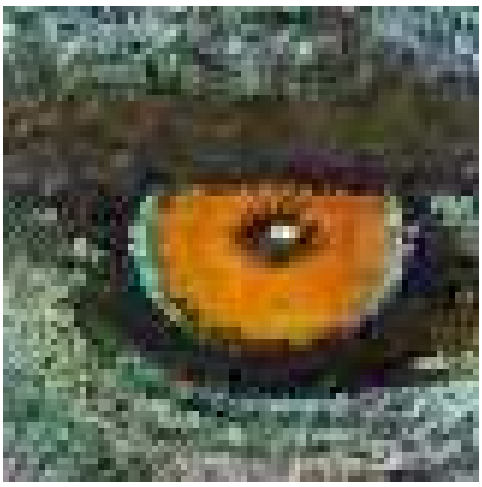


4.9 pav. VKP priklausomybė nuo pasukimų skaičiaus.

4.9 pav. vaizduojamas vidutinės kvadratinės paklaidos priklausomybės nuo pasukimų skaičiaus ir pikselių suskaidymo koeficiento N grafikas. Mažiausia vidutinė kvadratinė paklaida gaunama priskiriant koeficientui N reikėmę 2. Blogiausia – priskiriant 5.

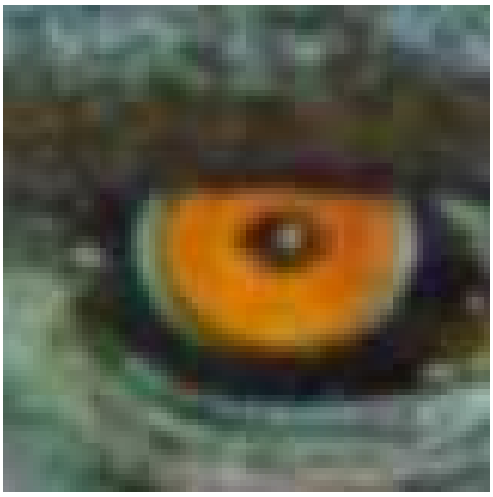
Lyginant praeito eksperimento rezultatus su šiais rezultatais, galima teigti, jog algoritmas geriau išsaugo vaizdo informaciją, jei yra didesnė N reikšmė, bet naudojant algoritmą viekartiniam vaizdo pasukimui. Pasukant vaizdą kelis kartus, jo informacija geriau išsaugoma, kai yra mažesnė N reikšmė.

(a)



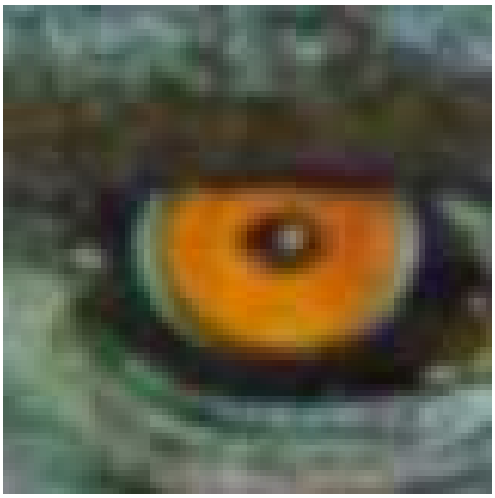
(b)

(c)



(e)

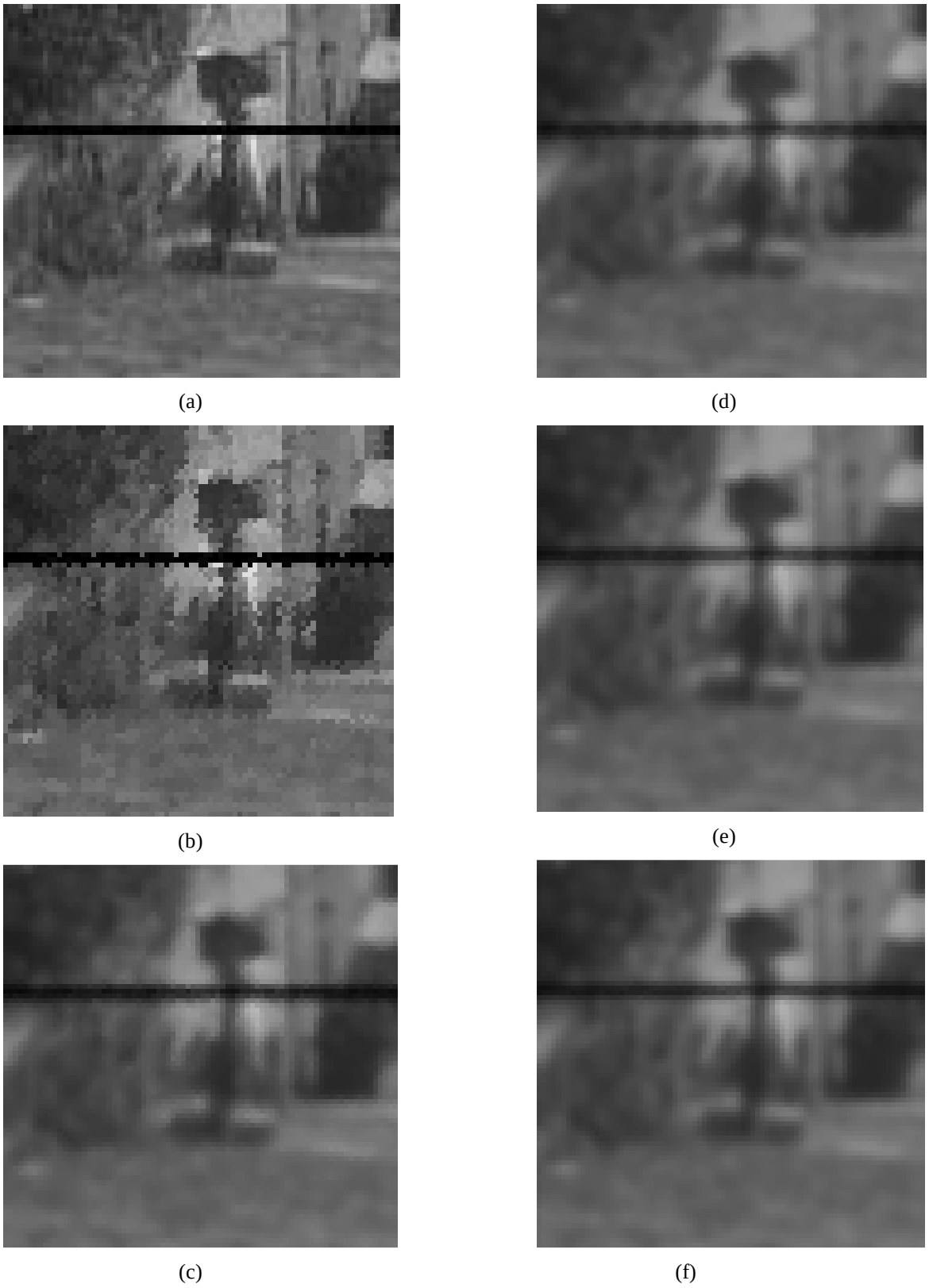
(d)



(f)

4.10 pav. Img01 paveikslėlio rezultato fragmentai. (a) – pradinis vaizdas, (b – f) – vaido fragmentai po 8 pasukimų su skirtingais N.

4.10 paveikslėlyje vaizduojami „Img01“ vaizdo rezultatų fragmentai, kai vaizdas buvo pasuktas 8 kartus. (b) vaizdo fragmente matoma ryški pikselizacija. Kai N yra didesnis už 1, pikselizacija išnyksta, bet atsiranda išsiliejimo efektas. Taip pat, kai N yra lygus 2, 3, 4 ir 5, vaizdo fragmentai vienas nuo kito mažai skiriasi.



(a) (d)

(b) (e)

(c) (f)

4.11 pav. Img06 paveikslėlio rezultato fragmentai. (a) – pradinis vaizdas, (b – f) – vaido fragmentai po 8 pasukimų su skirtingais N.

4.11 paveikslėlyje vaizduojami „Img06“ vaizdo rezultatų fragmentai, kai vaizdas buvo pasuktas 8 kartus. (b) vaizdo fragmente matoma ryški pikselizacija. Kai N yra didesnis už 1, pikselizacija pamažu išnyksta, bet atsiranda išsiliejimo efektas. Taip pat, kai N yra lygus 3, 4 ir 5, vaizdo fragmentai vienas nuo kito mažai skiriasi.

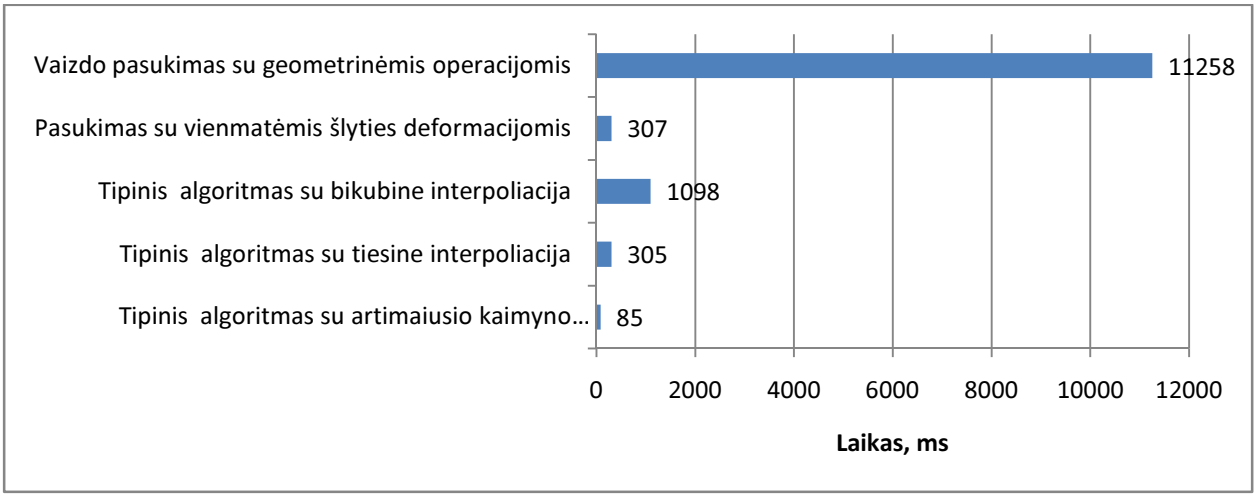
4.5 Algoritmų darbo laiko tyrimas

Pradiniai skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.3 pav. pateiktą eksperimento schemą. Vaizdai sukami 30^o kampu, vykdant 12 pasukimų. Kiekvieno pasukimo darbo laikas yra fiksuojamas ir iš jų suskaičiuojamas aritmetinis vidurkis. Šio eksperimento rezultatai pateikti tolimesnėje lentelėje.

4.18 lentelė. Vidutinis algoritmų darbo laikas

Algoritmas	Laikas, ms
Tipinis algoritmas su artimaiusio kaimyno interpoliacija	85
Tipinis algoritmas su tiesine interpoliacija	305
Tipinis algoritmas su bikubine interpoliacija	1098
Pasukimas su vienmatėmis šlyties deformacijomis	307
Vaizdo pasukimas su geometrinėmis operacijomis	11258

4.18 lentelėje surašyti penkių algortitmų darbo laiko tyrimo rezultatai.



4.12 pav. Vidutinis algoritmų darbo laikas.

4.12 pav. vaizduoja algortmų darbo laiko grafiką. Iš šio grafiko aiškiai matyti, jog greičiausias yra tipinis algoritmas su artimaiusio kaimyno interpoliacija, o lėčiausias – pasukimas

su geometrinėmis operacijomis. Pasukimo algoritmo su geometrinėmis operacijomis atžvilgiu, visi kiti algoritmai yra daugiau nei 90% greitesni.

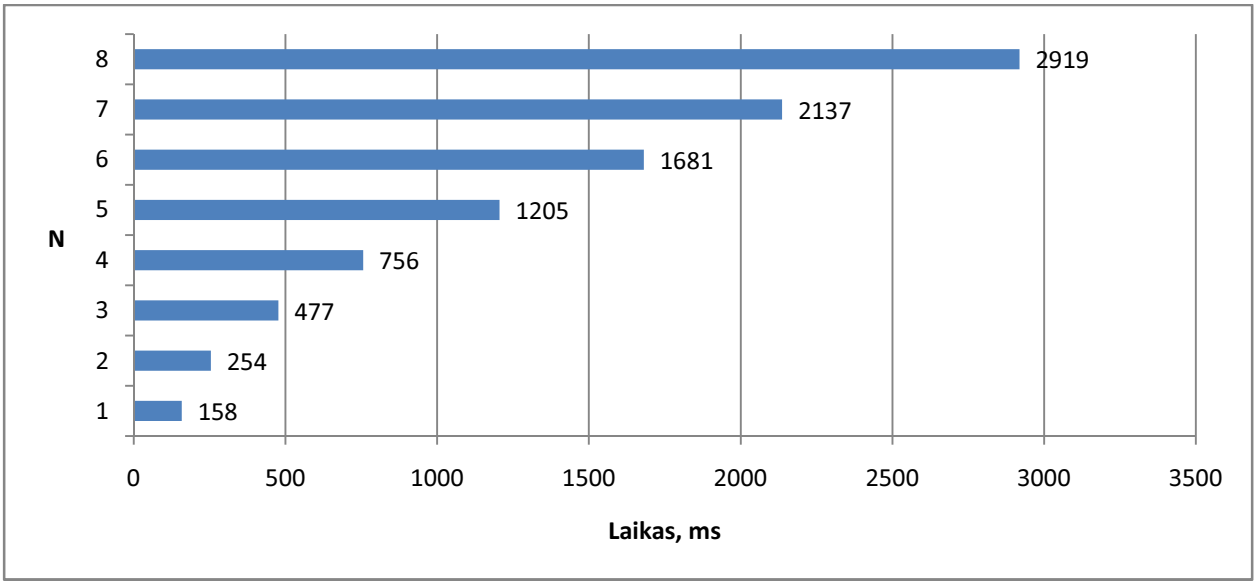
4.6 Siūlomo algoritmo darbo laiko tyrimas

Pradiniai skaitmeniniai vaizdai buvo panaudoti eksperimente pagal 3.3 pav. pateiktą eksperimento schemą. Vaizdai sukami 30^o kampu, vykdant 12 pasukimų. Kiekvieno pasukimo darbo laikas yra fiksuojamas ir iš jų suskaičiuojamas aritmetinis vidurkis. Eksperimentas vykdomas naudojant algoritma su skirtingomis N reikšmėmis. Eksperimento rezultatai pateikti tolimesnėje lentelėje.

4.19lentelė. Vidutinis algoritmo darbo laikas su skirtingais N

N	1	2	3	4	5	6	7	8
Laikas	158	254	477	756	1205	1681	2137	2919

4.19 lentelėje surašyti algortitmo darbo laiko tyrimo rezultatai. Tyrimas atliktas aūtuonis kartus, naudojant vis kitas pikselių padalijimo koeficiento N reikšmes.



4.13 pav. Algoritmo darbo laiko priklausomybė nuo N.

4.13 pav. Vaizduojama algoritmo darbo laiko priklausomybė nuo pikselių padalijimo reikšmės N. Iš gautų eksperinmento rezultatų galima teigti, jog algoritmo darbo laikas turi kvadratinę priklausomybę nuo N.

Remiantis 4.5 ir 4.18 lentelių duomenimis galima suskaičiuoti kiek gaunama kokybės įverčio darbo laiko atžvilgiu, didinant koeficientą N. Tolimesnėje lentelėje parodoma kiek gaunama vidutinės kvadratinės paklaidos įverčio ir darbo laiko padidėjimo, didinant N reikšmę.

4.20 lentelė. Gaunama VKP laiko atžvilgiu

N	2	3	4	5
VKP	9.352	1.640	0.426	0.087
t	96	223	279	449
VKP/t	0.0974	0.0074	0.0015	0.0002

4.20 lentelėje aprašoma gaunama vidutinės kvadratinės paklaidos vertė lako atžvilgiu, didinant koeficientą N.

Remiantis 4.20 lentelės duomenimis, galimą teigti, jog siūlomas algoritmas efektyviausiai dirbs, kai pikselių suskaidymo koeficientas N bus lygus 3. Toliau didinti N nėra tikslinga, nes žymiai išauga laiko sąnaudos, gaunant mažą kokybės padidėjimą.

4.7 Eksperimentų rezultatų įvertinimas

Atlikus eksperimentus gaunamos išvados apie algoritmus:

1. Palyginus gautų rezultatų vidutines kvadratines paklaidas δ, galima teigti, jog efektyviausias algoritmas, sukant vaizdą vieną kartą yra pasukimas su geometrinėmis operacijomis.
2. Mažiausiai efektyvus (prarandama daugiausiai vaizdo informacijos) – tipinis pasukimas su artimiausio kaimyno interpoliacija.
3. Efektyviausias algoritmas, sukant vaizdą keletą kartų yra tipinis pasukimas su artimiausio kaimyno interpoliacija. Mažiausiai efektyvus (prarandama daugiausiai vaizdo informacijos) – tipinis pasukimas su bikubine interpoliacija.
4. Iš gautų eksperinmento rezultatų galima teigti, jog greičiausias yra tipinis algoritmas su artimaiusio kaimyno interpoliacija, o lėčiausias – pasukimas su geometrinėmis operacijomis.
5. Imant visų eksperimentų rezultatų išvadas, geriausiai veikiantys algoritmai yra tipiniai pasukimai su artimiausio kaimyno ir tiesine interpoliacijomis.
6. Siūlomas algoritmas laiko ir kokybės atžvilgiu geriausiai veikia, kai pikselių suskaidymo koeficientas N yra lygus 3.

5. Išvados

1. Apžvelgti ir eksperimentiškai ištirti populiariausi vaizdo pasukimo algoritmai: tipinis pasukimas su trimis interpoliacijų tipais, pasukimas su vienmatėmis šlyties deformacijomis ir pasukimas su geometrinėmis operacijomis.
2. Tyrimų rezultatai parodė, jog kokybiškiausiai (mažiausia vidutinė kvadratinė paklaida), bet lėčiausiai, dirba pasukimas su geometrinėmis operacijomis.
3. Remiantis literatūra ir tyrimų rezultatais, realizuotas vaizdo pasukimo algoritmas, kuris veikia pikselių padalijimo į subpikselius pagrindu.
4. Hipotezė, jog vis didinant N algoritmas tiksliau dirbs, nepasitvirtino. Didinant N vidutinė kvadratinė paklaida pamažu nusistovi ties viena reikšme.
5. Iš tyrimų rezultatų matoma, jog norint gauti geriausią rezultatų kokybę, N reikšmė turi būti 3.

6. Literatūros sąrašas

1. Frizot M. The new history of photography. Köln: Könenmann Verlagsgesellschaft mbH, 1998.
2. Gilles Aubert and Pierre Kornprobst, Mathematical Problems inImage Processing – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://www-sop.inria.fr/members/Pierre.Kornprobst/tmp/extraits.pdf>
3. Valantinas, Jonas, *Diskrečiosios transformacijos*. Kaunas : Technologija, 2008.
4. Image Processing Learning Resources – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://homepages.inf.ed.ac.uk/rbf/HIPR2/>
5. Image Rotation in matlab, Rein van den Boomgaard – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://staff.science.uva.nl/~rein/multimedia/labRotation.pdf>
6. Richard Alan Peters, EECE\CS 253 Image Processing, Vanderbilt University School of Engineering, 2007
7. Comp.Graphics.Algorithms, Joseph O'Rourke – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://www.exaflop.org/docs/cgafaq/>
8. Ihtisham Kabir, High Performance Computer Imaging, Manning Publications, 1996
9. Linear and cubic interpolation – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://polymathprogrammer.com/2008/09/29/linear-and-cubic-interpolation/>
10. E. Bourennane, C. Milan, M. Paindavoine and S. Bouchoux, *Real Time Image Rotation Using Dynamic Reconfiguration*, Laboratory LE21, University of Burgundy, France, 2002
11. B. Chen and A. Kaufman, *"Two-pass image and volume rotation," IEEE Workshop on Volume Graphics*, 2001.
12. Bernd Jähne, *Practical Handbook on Image Processing for Scientific Applications,* CRC-Press, 1997
13. ANDREW S. GLASSNER *Graphics Gem*. Elsevier (USA), 1990.
14. *Laurent Condat, Dimitri Van De Ville,* FULLY REVERSIBLE IMAGE ROTATION BY 1-D FILTERING
15. "High Accuracy Rotation of Images," in *Computer Vision, Graphics and Image Processing*, Vol. 54, No. 4, July 1992, pp. 340-344.

16. Charles B. OwenFillia Makedon. High Quality Alias Free Image Rotation. *30th Asilomar Conference on Signals, Systems, and Computers*. Pacific Grove, California, 1996.
17. Bernd Jähne, IMAGE PROCESSING for SCIENTIFIC and TECHNICAL APPLICATIONS. University of Heidelberg, 2004.
18. Anti Aliased Image Rotation – [žiūrėta 2011-04-22]. Prieiga per internetą: <http://www.codeproject.com/KB/graphics/aarot.aspx>

7. Priedai

7.1 Vaizdo pasukimo algoritmų kodas

Tipinis vaizdo pasukimo algoritmas su artimiausio kaimyno interpoliacija.

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
#define max(a, b)  (((a) > (b)) ? (a) : (b))

QImage SimpleRotation::rotate(QImage image, int angle)
{
    float radians=(2*3.1416*angle)/360;
    float cosine=(float)cos(radians);
    float sine=(float)sin(radians);
    float Point1x=(-image.height()*sine);
    float Point1y=(image.height()*cosine);
    float Point2x=(image.width()*cosine-image.height()*sine);
    float Point2y=(image.height()*cosine+image.width()*sine);
    float Point3x=(image.width()*cosine);
    float Point3y=(image.width()*sine);
    float minx=min(0,min(Point1x,min(Point2x,Point3x)));
    float miny=min(0,min(Point1y,min(Point2y,Point3y)));
    float maxx=max(0,max(Point1x,max(Point2x,Point3x)));
    float maxy=max(0,max(Point1y,max(Point2y,Point3y)));
    int DestBitmapWidth=(int)ceil(fabs(maxx)-minx);
    int DestBitmapHeight=(int)ceil(fabs(maxy)-miny);
    QImage result(DestBitmapWidth,DestBitmapHeight,QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
      for(int y=0; y<DestBitmapHeight; y++)
      {
        int SrcBitmapx=(int)((x+minx)*cosine+(y+miny)*sine);
        int SrcBitmapy=(int)((y+miny)*cosine-(x+minx)*sine);
        if(SrcBitmapx >= 0 && SrcBitmapx < image.width() && SrcBitmapy >=0 &&
SrcBitmapy < image.height())
        {
            result.setPixel(x, y, image.pixel(SrcBitmapx, SrcBitmapy));
        }
      }
    }
    return result;
}
```

Tipinis vaizdo pasukimo algoritmas su tiesine interpoliacija (interpoliacijos kodas).

```
QRgb SimpleBilinear::interpolate(QImage image, float x, float y)
{
    int x1 = floor(x), x2 = ceil(x);
    int y1 = floor(y), y2 = ceil(y);
    QColor p1 = image.pixel(x1, y1);
    QColor p2 = image.pixel(x2, y1);
    QColor p3 = image.pixel(x1, y2);
    QColor p4 = image.pixel(x2, y2);
    int pRed = (p1.red() + p2.red() + p3.red() + p4.red())/4;
    int pGreen = (p1.green() + p2.green() + p3.green() + p4.green())/4;
    int pBlue = (p1.blue() + p2.blue() + p3.blue() + p4.blue())/4;
```

```
    return qRgb(pRed, pGreen, pBlue);
}
```

Tipinis vaizdo pasukimo algoritmas su bikubine interpoliacija (interpoliacijos kodas).

```
QRgb SimpleBicubic::interpolate(QImage image, float x, float y)
{
    int x1 = floor(x), x2 = ceil(x);
    int y1 = floor(y), y2 = ceil(y);
    // 3/32
    QColor p11 = image.pixel(x1, y1);
    QColor p12 = image.pixel(x2, y1);
    QColor p13 = image.pixel(x1, y2);
    QColor p14 = image.pixel(x2, y2);
    QColor p21, p22, p23, p24, p25, p26, p27, p28;
    QColor p31, p32, p33, p34;
    // 2/32
    if(x1-1 < 0)//left
    {
        p21 = image.pixel(x1, y1);
        p22 = image.pixel(x2, y1);
    }else
    {
        p21 = image.pixel(x1-1, y1);
        p22 = image.pixel(x2-1, y1);
    }
    if(x1+1 > image.width()-2)//right
    {
        p23 = image.pixel(x1, y1);
        p24 = image.pixel(x2, y1);
    }else
    {
        p23 = image.pixel(x1+1, y1);
        p24 = image.pixel(x2+1, y1);
    }
    if(y1-1 < 0)//top
    {
        p25 = image.pixel(x1, y1);
        p26 = image.pixel(x2, y1);
    }else
    {
        p25 = image.pixel(x1, y1-1);
        p26 = image.pixel(x2, y1-1);
    }
    if(y1+1 > image.height()-1)//bottom
    {
        p27 = image.pixel(x1, y1);
        p28 = image.pixel(x2, y1);
    }else
    {
        p27 = image.pixel(x1, y1+1);
        p28 = image.pixel(x2, y1+1);
    }
    // 1/32
    if(x-1 < 0 || y-1 < 0)
```

```
        p31 = image.pixel(x1, y1);
    else
        p31 = image.pixel(x1-1, y1-1);
    if(x-1 < 0 || y+1 > image.height()-1)
        p32 = image.pixel(x1, y1);
    else
        p32 = image.pixel(x1-1, y1+1);
    if(x+1 > image.width()-1 || y-1 < 0)
        p33 = image.pixel(x1, y1);
    else
        p33 = image.pixel(x1+1, y1-1);
    if(x+1 > image.width()-1 || y+1 > image.height()-1)
        p34 = image.pixel(x1, y1);
    else
        p34 = image.pixel(x1+1, y1+1);
    int pRed =((p11.red() + p12.red() + p13.red() + p14.red())*3+
               (p21.red() + p22.red() + p23.red() + p24.red() + p25.red() +
p26.red() + p27.red() + p28.red())*2+
               (p31.red() + p32.red() + p33.red() + p34.red()))/32;
    int pGreen =((p11.green() + p12.green() + p13.green() + p14.green())*3+
                 (p21.green() + p22.green() + p23.green() + p24.green() +
p25.green() + p26.green() + p27.green() + p28.green())*2+
                 (p31.green() + p32.green() + p33.green() + p34.green()))/32;
    int pBlue =((p11.blue() + p12.blue() + p13.blue() + p14.blue())*3+
                (p21.blue() + p22.blue() + p23.blue() + p24.blue() +
p25.blue() + p26.blue() + p27.blue() + p28.blue())*2+
                (p31.blue() + p32.blue() + p33.blue() + p34.blue()))/32;
    return qRgb(pRed, pGreen, pBlue);
}
```

Vaizdo pasukimo algoritmas su šlyties deformacija.

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
#define max(a, b)  (((a) > (b)) ? (a) : (b))

QImage ShearRotation::rotate(QImage image, int angle)
{
    float radians=(2*3.1416*angle)/360;
    QImage result = xshear(yshear(xshear(image, radians), radians),radians);
    return result;
}
QImage ShearRotation::xshear(QImage image, float angle)
{
    float tang = tan(angle/2);
    float Point2x=(image.width());
    float Point3x=(-image.height()*tang);
    float Point4x=(image.width()-image.height()*tang);
    int minx = min(0, min(Point2x, min(Point3x, Point4x)));
    int maxx = max(0, max(Point2x, max(Point3x, Point4x)));
    int DestBitmapWidth = (int)ceil(fabs(maxx)-minx);
    int DestBitmapHeight = image.height();
    QImage result(DestBitmapWidth, DestBitmapHeight, QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
      for(int y=0; y<DestBitmapHeight; y++)
      {
        int SrcBitmapx=(int)((x+minx) + tang*(y));
```

```
        if(SrcBitmapx >= 0 && SrcBitmapx < image.width())
        {
            result.setPixel(x, y, image.pixel(SrcBitmapx, y));
        }
      }
    }
    return result;
}
QImage ShearRotation::yshear(QImage image, float angle)
{
    float sine = sin(angle);
    float Point3y=(image.height());
    float Point2y=(image.width()*sine);
    float Point4y=(image.height()+image.width()*sine);
    int minx = min(0, min(Point2y, min(Point3y, Point4y)));
    int maxx = max(0, max(Point2y, max(Point3y, Point4y)));
    int DestBitmapHeight = (int)ceil(fabs(maxx)-minx);
    int DestBitmapWidth = image.width();
    QImage result(DestBitmapWidth, DestBitmapHeight, QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
      for(int y=0; y<DestBitmapHeight; y++)
      {
        int SrcBitmapy=(int)((y+minx) - sine*(x));
        if(SrcBitmapy >= 0 && SrcBitmapy < image.height())
        {
            result.setPixel(x, y, image.pixel(x, SrcBitmapy));
        }
      }
    }
    return result;
}
```

Vaizdo pasukimo algoritmai be informacijos praradimo (90^o ir 180^o)

```
QImage rotate90::rotate(QImage image)
{
    int DestBitmapWidth=image.height();
    int DestBitmapHeight=image.width();
    QImage result(DestBitmapWidth, DestBitmapHeight, QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
      for(int y=0; y<DestBitmapHeight; y++)
      {
        result.setPixel(x, y, image.pixel(y, DestBitmapHeight - x - 1));
      }
    }
    return result;
}
QImage rotate90::flip(QImage image)
{
    int DestBitmapWidth=image.height();
    int DestBitmapHeight=image.width();
    QImage result(DestBitmapWidth, DestBitmapHeight, QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
```

```
      for(int y=0; y<DestBitmapHeight; y++)
      {
        result.setPixel(x, y, image.pixel(DestBitmapWidth - x - 1,
DestBitmapHeight - y - 1));
      }
    }
    return result;
}
```

Vaizdo pasukimo algoritmas su pikselių suskaldymu.

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
#define max(a, b)  (((a) > (b)) ? (a) : (b))

QImage brotation::rotate(QImage image, int angle, int blocks)
{
    this->block  = blocks;
    float radians=(2*3.1416*angle)/360;
    float cosine=(float)cos(radians); this->cosines = cosine;
    float sine=(float)sin(radians); this->sines = sine;
    float Point1x=(-image.height()*sine);
    float Point1y=(image.height()*cosine);
    float Point2x=(image.width()*cosine-image.height()*sine);
    float Point2y=(image.height()*cosine+image.width()*sine);
    float Point3x=(image.width()*cosine);
    float Point3y=(image.width()*sine);
    float minx=min(0,min(Point1x,min(Point2x,Point3x)));
    float miny=min(0,min(Point1y,min(Point2y,Point3y)));
    float maxx=max(0,max(Point1x,max(Point2x,Point3x)));
    float maxy=max(0,max(Point1y,max(Point2y,Point3y)));
    this->mx = minx; this->my = miny;
    int DestBitmapWidth=(int)ceil(fabs(maxx)-minx);
    int DestBitmapHeight=(int)ceil(fabs(maxy)-miny);
    QImage result(DestBitmapWidth,DestBitmapHeight,QImage::Format_RGB32);
    for(int x=0; x<DestBitmapWidth ;x++)
    {
      for(int y=0; y<DestBitmapHeight; y++)
      {
        int SrcBitmapx=(int)((x+minx)*cosine+(y+miny)*sine);
        int SrcBitmapy=(int)((y+miny)*cosine-(x+minx)*sine);
        if(SrcBitmapx >= 0 && SrcBitmapx < image.width() && SrcBitmapy >=0 &&
SrcBitmapy < image.height())
        {
            result.setPixel(x, y, interpolate(image, x, y));
        }
      }
    }
    return result;
}
QRgb brotation::interpolate(QImage image, float x, float y)
{
    int pRed = 0; int pGreen = 0; int pBlue = 0;
    int h = this->block;
    for(int i = 0; i < this->block; i++)
    {
        for(int j = 0; j < this->block; j++)
```

```
        {
            float xi = x + ((float)i / (float)h);
            float yi = y + ((float)j / (float)h);
            int SrcBitmapx=(int)((xi + this->mx) * this->cosines + (yi +
this->my) * this->sines);
            int SrcBitmapy=(int)((yi + this->my) * this->cosines - (xi +
this->mx) * this->sines);
            if(SrcBitmapx >= image.width()) SrcBitmapx = image.width() - 1;
            if(SrcBitmapy >= image.height()) SrcBitmapy = image.height() - 1;
            if(SrcBitmapx < 0) SrcBitmapx = 0;
            if(SrcBitmapy < 0) SrcBitmapy = 0;
            QColor pixel = image.pixel(SrcBitmapx, SrcBitmapy);
            pRed = pRed + pixel.red();
            pGreen = pGreen + pixel.green();
            pBlue = pBlue + pixel.blue();
        }
    }
    pRed = pRed / (h * h);
    pGreen = pGreen / (h * h);
    pBlue = pBlue / (h * h);
    return qRgb(pRed, pGreen, pBlue);
}
```

Vaizdo pasukimas su geometrinėmis operacijomis.

```
#define aar_min(a, b) (((a) < (b))?(a):(b))
#define aar_max(a, b) (((a) > (b))?(a):(b))
#define aar_abs(a) (((a) < 0)?(-(a)):(a))
typedef bool CALLBACK (*aar_callback) (double);
inline int aar_roundup(double a) {if (aar_abs(a - (int)(a + 5e-10)) < 1e-9)
return (int)(a + 5e-10); else return (int)(a + 1);}
inline int aar_round(double a) {return (int)(a + 0.5);}
inline BYTE aar_byterange(double a) {int b = aar_round(a); if (b <= 0) return
0; else if (b >= 255) return 255; else return (BYTE)b;}
struct aar_pnt
{
        double x,y;
        aar_pnt(){}
        aar_pnt(double x,double y):x(x),y(y){}
};
double aar_area(const std::vector <aar_pnt> & p)
{
    double ret = 0.0;
    //Loop through each triangle with respect to p[0] and add the cross
multiplication
    for (int i = 1; i + 1 < p.size(); i++)
        ret += (p[i].x - p[0].x) * (p[i + 1].y - p[0].y) - (p[i + 1].x -
p[0].x) * (p[i].y - p[0].y);
    //Take the absolute value over 2
    return aar_abs(ret) / 2.0;
}
std::vector <aar_pnt> aar_ConvexHull(const std::vector <aar_pnt> & p)
{
    //Check for a polygon of size greater than three (less than that nothing
needs to be done)
    if (p.size() <= 3) return p;
```

```
    //Find the left-most index
    int leftmost = 0;
    for (int i = 1; i < p.size(); i++)
    {
        if (p[i].x < p[leftmost].x)
            leftmost = i;
        else if (p[i].x == p[leftmost].x && p[i].y < p[leftmost].y)
            leftmost = i;
    }
    std::vector <bool> inhull(p.size(), false);
    std::vector <aar_pnt> ret;
    int lastpoint = leftmost;
    //loop until you get back to leftmost
    do
    {
        int selectedpoint = -1;
        for (int i = 0; i < p.size(); i++)
        {
            //ignore the lastpoint and points already in the hull
            if (i == lastpoint) continue;
            if (inhull[i]) continue;

            if (selectedpoint == -1)
            {
                //if no point is yet selected, select this one
                selectedpoint = i;
            }
            else if ((p[i].x - p[lastpoint].x) * (p[selectedpoint].y -
p[lastpoint].y) - (p[selectedpoint].x - p[lastpoint].x) * (p[i].y -
p[lastpoint].y) <= 0)
            {
                //if the cross multiplication of the selected point and point
i in reference to lastpoint is <= 0 than select it
                selectedpoint = i;
            }
        }
        //Add selected point
        lastpoint = selectedpoint;
        inhull[lastpoint] = true;
        ret.push_back(p[lastpoint]);
    } while (lastpoint != leftmost);
    return ret;
}
bool aar_isinsquare(aar_pnt r, aar_pnt c, double coss, double sins)
{
    //Offset r
    r.x -= c.x;
    r.y -= c.y;
    //rotate r
    aar_pnt nr;
    nr.x = r.x * coss - r.y * sins;
    nr.y = r.y * coss + r.x * sins;
    //Find if the rotated polygon is within the square of size 1 centerd on
the origin
    nr.x = aar_abs(nr.x);
    nr.y = aar_abs(nr.y);
    return (nr.x < 0.5 && nr.y < 0.5);
```

```
}
double aar_pixoverlap(std::vector <aar_pnt> p, double lx, double ly, double
coss, double sins)
{
    //Offset the polygon by lx, ly so that the destination square is at (0,
0)
    //At the same time find the center of the source's polygon (after offset)
    aar_pnt c(0, 0);
    for (int i = 0; i < 4; i++)
    {
        p[i].x -= lx;
        p[i].y -= ly;
        c.x += p[i].x / 4.0;
        c.y += p[i].y / 4.0;
    }
    //Search for source points within the destination square
    std::vector <aar_pnt> np;
    for (int i = 0; i < 4; i++)
        if (p[i].x >= 0 && p[i].x <= 1 && p[i].y >= 0 && p[i].y <= 1)
            np.push_back(p[i]);
    //Search for destination points within the source square
    int dx[] = {0, 1, 1, 0};
    int dy[] = {0, 0, 1, 1};
    for (int i = 0; i < 4; i++)
        if (aar_isinsquare(aar_pnt(dx[i], dy[i]), c, coss, sins))
            np.push_back(aar_pnt(dx[i], dy[i]));
    double z;
    //Search for line intersections
    for (int i = 0; i < 4; i++)
    {
        int j = (i + 1) % 4;
        double minx = aar_min(p[i].x, p[j].x);
        double miny = aar_min(p[i].y, p[j].y);
        double maxx = aar_max(p[i].x, p[j].x);
        double maxy = aar_max(p[i].y, p[j].y);
        if (minx < 0.0 && 0.0 < maxx)
        {//Cross left
            z = p[i].y + -p[i].x * (p[i].y - p[j].y) / (p[i].x - p[j].x);
            if (z >= 0 && z <= 1)
                np.push_back(aar_pnt(0.0, z));
        }
        else if (minx < 1.0 && 1.0 < maxx)
        {//Cross right
            z = p[i].y + (1 - p[i].x) * (p[i].y - p[j].y) / (p[i].x -
p[j].x);
            if (z >= 0 && z <= 1)
                np.push_back(aar_pnt(1.0, z));
        }
        if (miny < 0 && 0 < maxy)
        {//Cross bottom
            z = p[i].x + -p[i].y * (p[i].x - p[j].x) / (p[i].y - p[j].y);
            if (z >= 0 && z <= 1)
                np.push_back(aar_pnt(z, 0.0));
        }
        else if (miny < 1 && 1 < maxy)
        {//Cross top
```

```
            z = p[i].x + (1 - p[i].y) * (p[i].x - p[j].x) / (p[i].y -
p[j].y);
            if (z >= 0 && z <= 1)
                np.push_back(aar_pnt(z, 1.0));
        }
    }
    //Sort the points and return the area
    return aar_area(aar_ConvexHull(np));
}
struct aar_dblrgbquad
{
        double red, green, blue, alpha;
};
QImage antialiasedrotate(QImage srcbmp, double Rotation)
{
    //Get Rotation between [0, 360)
    int mult = (int)Rotation / 360;
    if (Rotation >= 0)
        Rotation = Rotation - 360.0 * mult;
    else
        Rotation = Rotation - 360.0 * (mult - 1);
    //Calculate the cos and sin values that will be used throughout the
program
    double coss = aar_cos(Rotation);
    double sins = aar_sin(Rotation);
    //Calculate some index values so that values can easily be looked up
    int indminx = ((int)Rotation / 90 + 0) % 4;
    int indminy = ((int)Rotation / 90 + 1) % 4;
    int indmaxx = ((int)Rotation / 90 + 2) % 4;
    int indmaxy = ((int)Rotation / 90 + 3) % 4;
    //Calculate the sources x and y offset
    double srcxres = (double)srcbmp.width() / 2.0;
    double srcyres = (double)srcbmp.height() / 2.0;
    //Calculate the x and y offset of the rotated image (half the width and
height of the rotated image)
    int mx[] = {-1, 1, 1, -1};
    int my[] = {-1, -1, 1, 1};
    double xres = mx[indmaxx] * srcxres * coss - my[indmaxx] * srcyres *
sins;
    double yres = mx[indmaxy] * srcxres * sins + my[indmaxy] * srcyres *
coss;
    //Get the width and height of the image
    int width = aar_roundup(xres * 2);
    int height = aar_roundup(yres * 2);
    QImage result(width,height,QImage::Format_RGB32);
    double xtrans;
    double ytrans;
    for (int x = 0; x < srcbmp.width(); x++)
    {
        for (int y = 0; y < srcbmp.height(); y++)
        {
            //Construct the source pixel's rotated polygon
            std::vector<aar_pnt> p;
            xtrans = (double)x - srcxres;
            ytrans = (double)y - srcyres;
            p.push_back(aar_pnt( xtrans * coss - ytrans * sins + xres, xtrans
* sins + ytrans * coss + yres));
```

```
            p.push_back(aar_pnt( (xtrans + 1) * coss - ytrans * sins + xres,
(xtrans + 1) * sins + ytrans * coss + yres));
            p.push_back(aar_pnt( (xtrans + 1) * coss - (ytrans + 1) * sins +
xres, (xtrans + 1) * sins + (ytrans + 1) * coss + yres));
            p.push_back(aar_pnt( xtrans * coss - (ytrans + 1) * sins + xres,
xtrans * sins + (ytrans + 1) * coss + yres));

            //Find the scan area on the destination's pixels
            int mindx = (int)p[indminx].x;
            int mindy = (int)p[indminy].y;
            int maxdx = aar_roundup(p[indmaxx].x);
            int maxdy = aar_roundup(p[indmaxy].y);

            for (int xx = mindx; xx < maxdx; xx++)
            {
                for (int yy = mindy; yy < maxdy; yy++)
                {
                    double dbloverlap = aar_pixoverlap(p, xx, yy, coss, -
sins);
                    if (dbloverlap)
                    {
                        QColor pixel = srcbmp.pixel(x, y);
                        QColor target = result.pixel(xx,yy);
                        int pRed = aar_round(((double)pixel.red() *
dbloverlap + target.red()));
                        int pGreen = aar_round(((double)pixel.green() *
dbloverlap + target.green()));
                        int pBlue = aar_round(((double)pixel.blue() *
dbloverlap + target.blue()));
                        result.setPixel(xx, yy, qRgb(pRed, pGreen, pBlue));
                    }
                }
            }

        }
    }
    return result;
}
```