Marcin Nowak
Poznań, 3 kwietnia 2007

Raport – Rozpoznawanie Obrazów

1. Cel projektu.

1.1. Opis ogólny

Celem projektu jest wykrycie poruszających się obiektów, oraz estymacja ich kształtu oraz ruch z wejściowego strumienia wideo, przy czym w celu przyspieszenia obliczeń, możliwie duża ich część ma być wykonywana przez akcelerator graficzny komputera, co może pozwolić na obliczenia wykonywane w czasie rzeczywistym.

1.2. Kryteria oceny

Użyty algorytm pozwala na wyliczenie mapy głębi widzianej sceny. Taką mape można w prosty sposób zwizualizować poprzez mapowanie głębi w kolor. W ten sposób można ocenić, czy kształt widzianych obiektów odowiada tym ze świata rzeczywistego. Innym kryterium oceny algorymu będzie jego wydajność, gdyż jednym z celów jest osiągnięcie estymacji w czasie rzeczywisty.

1.3. Dane wejściowe

Dane wejściowe to dowolny strumień wideo pochodzący np z pliku, lub kamery. Docelowy rozmiar strumienia to 640x480@30FPS, jednak zakłada się możliwość redukcji liczby klatek na sekundę, lub rozdzielczości, tak by wybrana implementacja działała w czasie rzeczywistym.

2. Próba osadzienia projektu w w dziedzinie rozpoznawania obrazów.

- Brak uczenia się, projekt działa w oparciu jedynie o sąsiednie klatki
- Brak wyliczania konkretnych cech
- Podejście bezpośrednie

3. Proponowane podejście

Pozyskanie mapy głębi, oraz wyliczenie wektora ruchy dla każdego piksela będzie składało się z kilku etapów:

- Przefiltrowanie każdej klatki przy użyciu filtru Gauss'a
- Przefiltrowanie strumienia w dziedzinie czasu przy użyciu filtru Gauss'a
- Wyliczenie *przepływu optycznego* (*Optic flow*)
- Wykrycie poruszających się obiektów metodą odejmowania obrazów
- Wyliczenie mapy głębi, oraz wektorów przesunięcia dla wszystkich pikseli
- Wizualizacja wyniku

4. Przewidywane problemy

4.1. Trudności z „debugowaniem" kodu wykonywanego na GPU

- Brak sensownych pomysłów na rozwiązanie tego problemu

4.2. Opóźnienie uzyskanego wyniu związane z filtrowaniem stumienia w dziedzinie czasu.

- Użycie małego jądra dla filtra Gauss'a
- Filtrowanie tylko „w tył" (użycie tylko historycznych klatek)
- Filtorwanie asymetryczne (użycie mniejszej liczby klatek „w przód" niż „w tył")

5. Użyte technologie

Biblioteki

- OpenCV lub DSVideoLib by pozyskać strumień wideo
- GLUT, Cg, CgGL, by przeprowadzić obliczenia na GPU

5.1. Interfejs

- Brak

6. Implementacja

Uwagi ogólne

Projekt został napisany w języku C++, w środowisku Visual Studio 2005.
Wszystkie obrazy, zarówno te pobrane z kamery, oraz te wyliczone są pamiętane bezpośrednio w pamięci karty graficznej w postaci tekstur. Aby zobaczyć wynik obliczeń należy wyrenderować odpowiednią teksturę do głównego okna programu, lub odczytać wartości z bufora ramki przy pomocy funkcji glReadPixels(). Wszystkie zaimplementowane obliczenia są wykonywane na GPU. Program główny zajmuje się przygotowaniem obrazów (tj. wczytaniem obrazu z kamery, ustawieniem odpowiedniej jednostki tekstury w OpenGL/Cg), ustawieniem docelowego miejsca renderingu, oraz ustawieniem parametrów renderingu. Wszystkie złożone obliczeniowo algorytmy zaimplementowano bezpośrednio na GPU

Cały kod znajduję się praktycznie w dwóch plikach.

- Main.cpp – Kod głównego programu
- postEffect.cgfx – plik zawierający podprogramy dla GPU

Najważniejsze funkcje głównego programu to:

- display() – tu następuje pobranie obrazu z kamery, wywołanie pod procedur wykonywanych na GPU, oraz wizualizacja wyników
- InitGL() – ustawia parametry OpenGL, tworzy tekstury o odpowiednich rozmiarach i formatach
- InitPostEffect() – wczytuje programy dla GPU, oraz inicjalizuje bibliotekę CgToolkit
- processKeys() – przetwarza wejście od użytkownika
- CreateTextureRect() – tworzy prostokątne tekstury trzymające wyniki obliczeń
- RenderFullScreenQuad() – renderuje prostokąt, którego wypełnianie powoduje wywołanie obliczeń na GPU
- Render2() – renderuje wybraną technikę do wybranej tekstury wynikowej. Wspiera rendering wieloprzebiegowy (rendering technik z plików CgFx o ilości przejść większej niż jeden)

Opis algorytmu wyliczającego *Optical Flow*

Do wyliczenia *Optical Flow* użyto iteracyjne, piramidalnej wersji algorytmu Lucas-Kanade [81], zaproponowaną przez pana Bouguet na potrzeby biblioteki OpenCV. Poniżej podano krótkie wyprowadzenie wzorów użytych w tej metodzie.

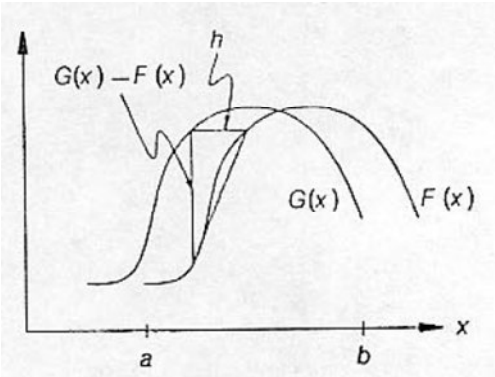


Figure 2: Two curves to be matched

Problem: dla danych funkcji *F(x)*, oraz *G(x)* znajdź takie przesunięcie *h*, dla którego:

$$E = \sum_x (F(x+h) - F(x))^2 = 0,$$

Gdzie sumowanie następuje po punktach będących sąsiedztwem punktu, dla którego szukamy *h*.
Z powyższego obrazka widzimy, że:

$$F'(x) \approx \frac{F(x+h) - F(x)}{h} = \frac{G(x) - F(x)}{h}$$

$$h \approx \frac{G(x) - F(x)}{F'(x)}$$

Wyliczając pochodną, oraz rozwiązując względem *h* otrzymujemy:

$$E = \sum_x (F(x+h) - F(x))^2$$

$$0 = \frac{\partial E}{\partial h} \approx \frac{\partial}{\partial h} \sum_x [F(x) + hF'(x) - G(x)]^2 = 2\sum_x [F(x) + hF'(x) - G(x)]^2$$

$$h \approx \frac{\sum_x F'(x)[G(x) - F(x)]}{\sum_x F'(x)^2}$$

Otrzymane w ten sposób *h* jest pewnym przybliżeniem szukanego przesunięcia. Możliwe jest przeprowadzenie wielu iteracji w celu polepszenia rozwiązania:

$$h_{k+1} = h_k + \frac{\sum_x w(x)F'(x+h_k)[G(x) - F(x+h_k)]}{\sum_x w(x)F'(x+h_k)^2}$$

W naszym przypadku należy wyliczyć identyczną estyma tę, tyle, że dla funkcji dwu wymiarowej:

$$h = \left[\sum_x \left(\frac{\partial F(x)}{\partial \underline{x}}\right)^T [G(x) - F(x)]\right] \left[\sum_x \left(\frac{\partial F(x)}{\partial \underline{x}}\right)^T \left(\frac{\partial F(x)}{\partial \underline{x}}\right)\right]^{-1}$$

$$h_{k+1} = h_k + \left[\sum_x \left(\frac{\partial F(x+h_k)}{\partial \underline{x}}\right)^T [G(x) - F(x+h_k)]\right] \left[\sum_x \left(\frac{\partial F(x+h_k)}{\partial \underline{x}}\right)^T \left(\frac{\partial F(x+h_k)}{\partial \underline{x}}\right)\right]^{-1}$$

Gdy wprowadzimy następujące oznaczenia:

$$\nabla I = \begin{bmatrix} F_x \\ F_y \end{bmatrix} \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} \end{bmatrix}^T, \quad \delta I(x) = G(x) - F(x)$$

$$G = \sum_x \begin{bmatrix} F_x^{\,2} & F_x F_y \\ F_x F_y & F_y^{\,2} \end{bmatrix}, \quad \underline{b} = \sum_x \begin{bmatrix} F_x \delta I(x) \\ F_y \delta I(x) \end{bmatrix}$$

To problem estymacji *h* można zapisać następująco:

$$h \approx G^{-1} \underline{b}$$

W praktyce wektor ∇I jest wyliczane z drugiego obrazu (G(x)). Można tego bezkarnie dokonać przy założeniu, że przesunięcie między dwoma obrazami jest bardzo małe, co jest w praktyce spełnione przy wersji piramidalnej algorytmu. Dzięki temu macierz G można wyliczyć tylko raz dla danego poziomu piramidy.

Cały algorytm można zapisać następująco:

Wylicz piramidy od L_0 do L_k, gdzie L_0 to obraz orginalny

Dla każdego poziomu piramidy:

-wylicz gradienty ∇I

-oblicz macierz G

-użyj podwojonej estymacji z poprzedniego poziomu piramidy

-**Dla każdej iteracji**

-Oblicz wektor b

-Rozwiąż układ $h \approx G^{-1} \underline{b}$

Koniec pętli po iteracjach

Koniec pętli po piramidach

Liczba iteracji oraz piramid została domyślnie ustalona na 5. Większa liczba piramid nie ma sensu, ponieważ w tym przypadku rozdzielczość obrazu zostanie zredukowana z 640x480 do 40x30 pikseli.

W dalszej części przedstawiono szczegóły implementacyjne powyższej metody.

Filtrowanie obrazu

Filtrowanie obrazu w przypadku algorytmów wyliczających *Optical Flow* jest często procedurą niezbędną, ponieważ są one bardzo czułe na szumy oraz zmiany jasności. W celu eliminacji szumów przefiltrowano obraz w dziedzinie współrzędnych X, Y oraz dziedzinie czasu T.
Do każdego z tych filarowań użyto filtru konwolucyjnego o rozmiarze 5 elementów, o rozkładzie Gauss'a: `gauss5 = {0.0625, 0.25, 0.375, 0.25, 0.0625}`.
Kod CgFx realizujący filtrowanie w dziedzinie X i Y wygląda następująco:

```
float gauss2[] = {0.006134, 0.04908, 0.171779, 0.343558, 0.429448};
float GaussX(float2 uv : TEXCOORD0) : COLOR0
{
  float4 avg = float4(0,0,0,1);
  for (int x=-2; x<=2; x++)
   avg.xyz += texRECT(screen,uv+float2(x,0)).zyx *  gauss5[x+2];

   float3 colorMask = float3(0.299, 0.587, 0.114);
   return  dot(colorMask,avg.xyz).x; //redukcja koloru
}

float GaussY(float2 uv : TEXCOORD0) : COLOR0
{
  float avg=0;
  for (int y=-2; y<=2; y++)
   avg += texRECT(screen,uv+float2(0,y)).x *  gauss5[y+2];
   return avg;
}


technique GaussXY
{
   pass Xpass
   {
          FragmentProgram = compile fp40 GaussX();
   }


   pass Ypass
   {
          FragmentProgram = compile fp40 GaussY();
   }

}
```

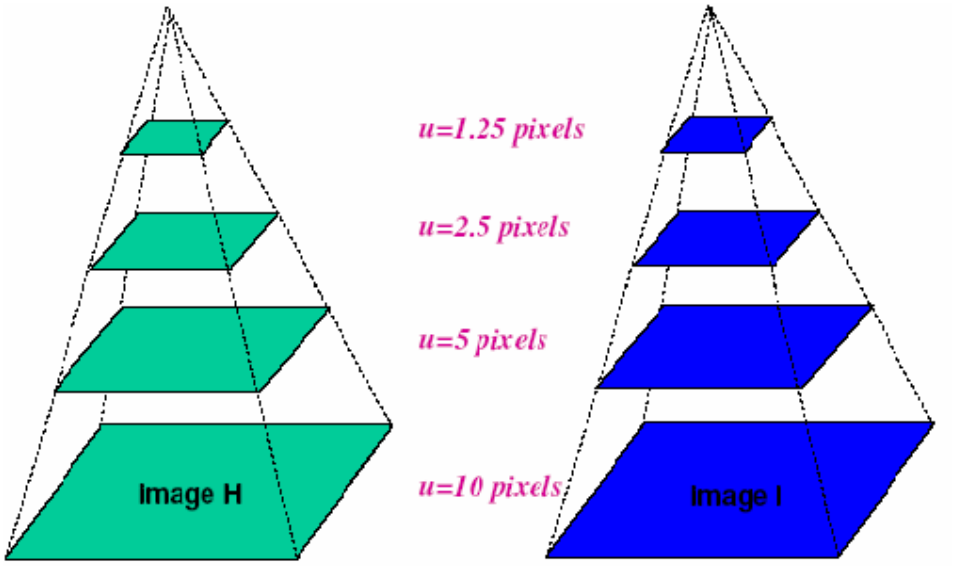
Przy okazji filtrowania obrazu następuje również redukcja kolorów z trzech kanałów RGB, do jednego kanału intensywności. Dokonuje tego poniży kawałek kodu:

```
float3 colorMask = float3(0.299, 0.587, 0.114);
   return  dot(colorMask,avg.xyz).x; //redukcja koloru
```

W analogiczny sposób dokonywane jest filtrowanie w dziedzinie czasu.

Wyliczanie piramid.

Do wyliczenia *Optical Flow* zaimplementowano piramidalną, iteracyjną wersję algorytmu Lucas-Kanade [81], zaproponowaną przez pana Bouguet na potrzeby biblioteki OpenCV. Główna różnica w tym przypadku polega na tym, że jest to wersja gęsta (*dense*), wyliczana na dla każdego piksela obrazu.
Wersja ta wymaga wyliczenia serii obrazów o coraz niższej rozdzielczości zwanych reprezentacją piramidową.



Rys.1 Reprezentacja piramidowa obrazów

Kolejne poziomy piramidy wyliczane są rekurencyjnie z niższych poziomów poprzez uśrednianie 4 sąsiednich próbek. Poziom 0, to obraz oryginalny, poziom 1, to obraz o rozmiarze zredukowanym o połowę w osi X i Y.
Implementacja polega na renderingu prostokąta o rozmiarze odpowiadającym rozmiarowi danego poziomu piramidy. Poniżej przedstawiono kod odpowiedniego programy dla GPU:

```
float4 Resample(float2 uv : TEXCOORD0) : COLOR0
{
   const float2 offset = float2(0.5,0.5);
   uv-=offset;
   uv*=2.0;
   uv+=offset;
   float4 result = float4(0,0,0,0);
   result.xyz += texRECT(screen,uv+float2(0,0)).xyz;
   result.xyz += texRECT(screen,uv+float2(1,0)).xyz;
   result.xyz += texRECT(screen,uv+float2(0,1)).xyz;
   result.xyz += texRECT(screen,uv+float2(1,1)).xyz;
   result.xyz /= 4.0;
   return result;
}

technique Rescale{
   pass p0{
          FragmentProgram = compile fp40 Resample();
   }
}
```

Wyliczenie wektora gradientu.

Do wyliczenia operatora gradientu użyto operatora Sobela. Jego wyliczenie rozbito na dwie iteracje. Poniżej przedstawiono program dla GPU, który w dwóch iteracjach oblicza wektor gradientu. Wynik obliczeń pierwszej iteracji przekazywany jest do drugiej iteracji poprzez zmienną *screen.*

```
float4 SobelGrad1It(float2 uv : TEXCOORD0) : COLOR0
{
   float4 result = float4(0,0,0,1);
   float3 conv = float3(1,2,1);

   float3 xVals;
   float3 yVals;
   //pobranie odpowiednich punktow z obrazu drugiego
   xVals.x = texRECT(Img2,uv+float2(0,-1)).x;
   xVals.y = texRECT(Img2,uv+float2(0,0)).x;
   xVals.z = texRECT(Img2,uv+float2(0,1)).x;

   yVals.x = texRECT(Img2,uv+float2(-1,0)).x;
   yVals.y = xVals.y;
   yVals.z = texRECT(Img2,uv+float2(1,0)).x;


   result.x = dot(conv,xVals); //iloczyn wektorowy
   result.y = dot(conv,yVals); //iloczyn wektorowy
   result.w = texRECT(Img2,uv).x;
   return result;
}

float4 SobelGrad2(float2 uv : TEXCOORD0) : COLOR0
{
   float4 result = float4(0,0,0,1);
   float3 conv = float3(1,0,-1);

   float3 xVals;
   float3 yVals;
   //pobieranie wynikow wyliczonych w ostatniej klatce obrazu
   float4 middle = texRECT(screen,uv+float2(0,0)).xyzw;

   xVals.x = texRECT(screen,uv+float2(-1,0)).x;
   xVals.y = middle.x;
   xVals.z = texRECT(screen,uv+float2(1,0)).x;

   yVals.x = texRECT(screen,uv+float2(0,-1)).y;
   yVals.y = middle.y;
   yVals.z = texRECT(screen,uv+float2(0,1)).y;

   result.x = dot(conv,xVals); //iloczyn wektorowy
   result.y = dot(conv,yVals); //iloczyn wektorowy
   result.zw = middle.zw; //iloczyn wektorowy
   return result;
}
```

```
technique Grad{
   pass p1
   {
         FragmentProgram = compile fp40 SobelGrad1();
   }
   pass p2
   {
               FragmentProgram = compile fp40 SobelGrad2();
   }
   }
```

Wyliczenie macierzy G.

Wyliczenie macierzy G sprowadza się do zsumowania wartości gradientu wyliczonego wcześniej dla całego sąsiedztwa punktu. Suma ta jest sumą warzoną z wagami o rozkładzie Gauss'a zdefiniowanymi w taki sposób, by punkt środkowy miał największą wagę. Po wyliczeniu macierzy można wyliczyć jej wyznacznik oraz wartości własne. Gdy wartość wyznacznika lub wartości własnych jest zbyt niska, to program podstawia 0 pod zwracaną wartość wyznacznika. Wyjściem tego programu są wartości sum częściowych F_x^2, F_y^2, oraz F_xF_y, jak również wartość wyznacznika macierzy G.

```
float4 CumulativeGrads1(float2 uv: TEXCOORD) : COLOR0
{
   float4 result = float4(0,0,0,0);
   float4 gradients;
   float2 offset;

   for (int y=-2; y<=2; y++)
         for (int x=-2; x<=2; x++)
         {
           offset = uv + float2(x,y);// + optFlow;
           //gradients.xyzw = bilinearTex(gradTex,offset).xyzw;
           gradients.xyzw = texRECT(gradTex,offset).xyzw;
           result.xyz+=(gradients.xxy * gradients.xyy) * gauss5[y+2]
*gauss5[x+2] ;
         }
   //oblicznie wyznacznika G
   result.w = result.y * result.y - result.x * result.z;
   //wartości wlasne G
   float ab,c2,del;
   ab = result.x+result.z;//dx2 + dy2
   c2 = result.y*result.y;
   del = sqrt(ab*ab-4*(ab-c2));
   float l1,l2;
   l1 = (-ab-del)*0.5;
   l2 = l1+del;
   if (abs(l1) > detThreshold && abs(l2) > detThreshold && abs(result.w)
> 0.000003)
         result.w = 1.0 / result.w;
   else
         result.w = 0.0;

   return result;
}
```

Wartość `detThreshold` to wartość progowa poniżej której uznaje się, że wartość wyznacznika macierzy G jest równa 0. Parametr ten może być zmieniany podczas działania programu.

Wyliczenie wektora b.

Ten krok algorytmu realizowany jest przez kolejny program GPU. Jest on wykonywany dla każdej iteracji programu dla każdego poziomu piramidy. Wyliczenie wektora b jest również najkosztowniejszym krokiem całych obliczeń, ponieważ wymaga dwukrotnie więcej pobrań tekstury niż przy wyliczeniu macierzy G.
Dzięki wcześniej wyliczonym wartościom wyznacznika oraz wartości własnych macierzy G możliwe jest zaprzestanie obliczeń w tych punktach, dla których wartość wyznacznika wynosi 0. Do tego celu można użyć np. sprzętowego bufora głębi, który jest implementowany we wszystkich współczesnych kartach graficznych. Próby wykorzystania bufora nie powiodły się jednak, najprawdopodobniej przez błąd w implementacji.

Ze względu na dużą złożoność obliczeniową tego kroku wymyślono kolejną optymalizację. Polega ona na sumowaniu wartości najpierw po kolumnach, a następnie po wierszach. Innymi słowy dokonano pewnej dekompozycji podobnie jak dla operatora Sobela. Ze względu na fakt, że takie sumowanie w sąsiedztwie jednego punku wpływa na wszystkich jego sąsiadów w jego wierszu można się spodziewać pewnych dodatkowych niedokładności, jednak ze względu na niewielki rozmiar okna, oraz na dużą lokalność obliczeń (w algorytmie LK zakłada się, że punkt porusza się tak samo jak jego niewielkie sąsiedztwo) dodatkowy błąd nie jest duży.
Dla okna o rozmiarze *n*, pierwotna wersja wymaga *O(n^2)* operacji, wersja uproszczona wymaga ich już tylko *O(n)*.
Kod tego podprogramu wygląda bardzo podobnie do tego zaprezentowanego wcześniej, dlatego go nie zamieszczono.

Rozwiązanie układu – obliczenie wektora przesunięcia.

Jest to ostatni krok algorytmu. Rozwiązuje układ równań metodą wyznacznikową. Zwraca nowo wyliczone przesunięcie zsumowane z poprzednim wynikiem. Gdy wcześniej wyliczona wartość wyznacznika wyniesie 0, to program zwróci przesunięcie wyliczone we wcześniejszej iteracji. Przed zwróceniem wyniku program sprawdza, czy popełniony w ostatniej iteracji błąd jest mniejszy od pewnego progu ustalanego w programie. Popełniony błąd z ostatniej iteracji to po prostu suma pochodnych po czasie, których wartości wraz z kolejnymi iteracjami powinny się zmniejszać o ile wyliczenia są zbieżne. Pozwala to kontrolować sytuacje, w których z jakiegoś powodu algorytm nie potrafi wyliczyć poprawnego przesunięcia. Powodów takiego problemu może być kilka:

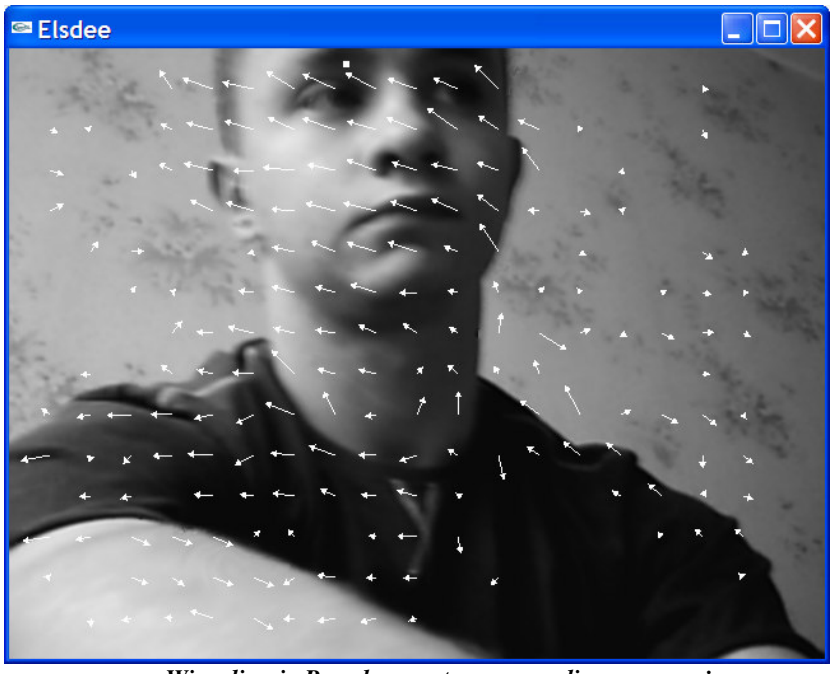
- Zasłonięcie śledzonego punku
- Zmiana średniej jasności sceny
- Duże przesunięcie
- Problem apertury
- Otoczenie punktu jest zbyt mało zróżnicowane (niskie wartości własne G)

Poniżej przedstawiono program GPU realizujący przedstawiony krok.

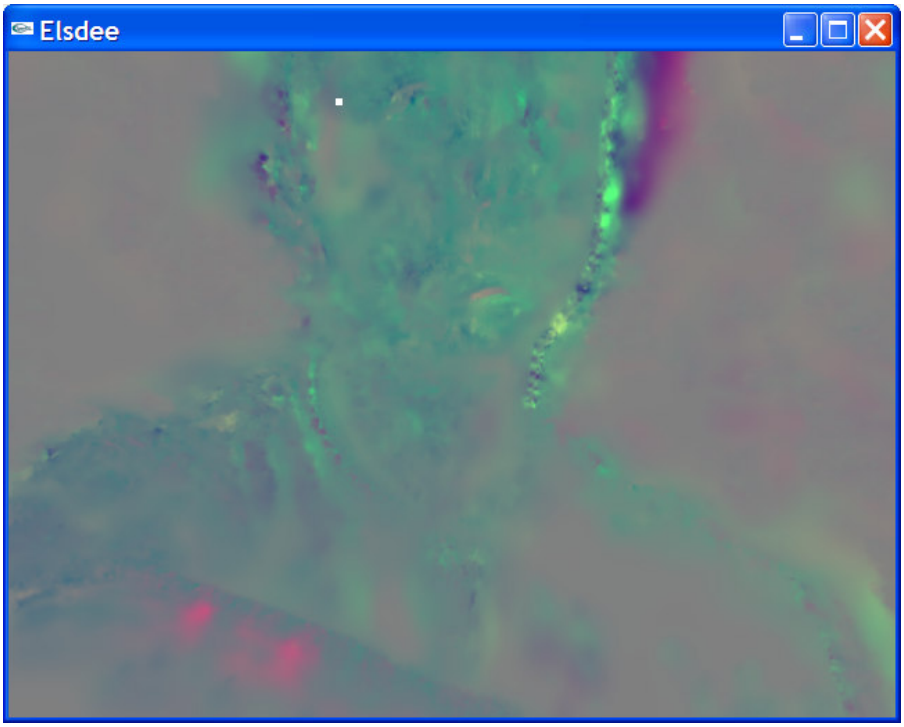
```
float4 opticalFlowIt(float2 uv : TEXCOORD0) : COLOR0
{
	grads cumulative; //struktura przechowujaca wektor b I macierz G
	cumulative.a.xyzw = texRECT(cumGrad1Tex,uv).xyzw; //pobranie G
	float4 resultR = float4(0,0,1.0,0.0);
	float2 optFlow;

	optFlow = GetOptFlow(uv); //opbranie ostatniego opticalFlow

	if (cumulative.a.w == 0.0)//gdy wyznacznik =0, zwroc ostatni wynik
		return float4(optFlow,0,0);
	cumulative.b = texRECT(cumGrad2Tex,uv).xyz; //pobranie b

	if (cumulative.a.w != 0) //non singular solution
	{
		float deltaX;
		float deltaY;
	  deltaX = -(cumulative.b.y * cumulative.a.y - cumulative.b.x *
cumulative.a.z);
	  deltaY = -(cumulative.a.y * cumulative.b.x - cumulative.a.x *
cumulative.b.y);
	  resultR.x = deltaX * cumulative.a.w;
	  resultR.y = deltaY * cumulative.a.w;
	}
	if (cumulative.b.z < lkThreshold)
		return resultR+float4(optFlow,0,0);
	return float4(0,0,0,0);
}
```

7. Wyniki

Przedstawiony algorytm działa w czasie rzeczywistym przy ponad 20 klatkach/s w rozdzielczości VGA. Poniżej zrzut ekranu wizualizujący otrzymany *przepływ optyczny* jako wektory przesunięć poszczególnych pikseli.



Wizualizacja Przepływu optycznego wyliczanego na żywo.



Wizualizacja 2 – Kolory oznaczają kierunek ruchu. Kanał R – Tx, Kanał G – Ty

Otrzymane rezultaty są bardzo zadowalające. Jakość wyliczeń zależy w dużym stopniu od jakości oświetlenia, ustawień kamery (kontrast, jasność) jak również samej sceny. Scena powinna być dostatecznie zróżnicowana.
Drugi rodzaj wizualizacji pozwala ocenić błedy obliczeń. Na powyższym zrzucie widzimy pewną aure (szegółnie po prawej stronie, w górnej częsci obrazu) otaczającą poruszające się obiekty. Błąd tego typu powstaj ze względu na implementację piramidalną - ruch jest propagowany z wyższych poziomów piramidy na piksele dla których nie da się wyliczyć odpowiedniej korekty ze względu na przysłonięcie pikseli (tj. zmianę w ich sąsiedztwie) oraz problem apartury. Kontrola błędu (sumy kwadratów) poprzez modyfikację progu odcięcia mogłaby pomóc, w przypadku, gdy tło na którym porusza się obiekt jest dostatecznie zróżnicowane, gdy tło jest jednolite, to wyliczony błąd będzie bliski zeru, co powoduje, że usunięcie powstałej aury nie jest możliwe.