# DisCODe: komponentowa struktura ramowa do przetwarzania danych sensorycznych

**Tomasz Kornuta, Maciej Stefańczyk**
Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska

**Streszczenie:** Aby działać w nieuporządkowanym i dynamicznie zmieniającym się środowisku, roboty muszą być wyposażone w rozmaite sensory, a ich systemy sterowania muszą być w stanie przetworzyć odbierane dane sensoryczne możliwie najszybciej w celu odpowiedniego zareagowania na zachodzące wydarzenia. Celem poniższej pracy było opracowanie oprogramowania umożliwiającego tworzenie podsystemów sensorycznych, mogących stanowić ekwiwalent dowolnych zmysłów zwierząt. Przeprowadzone testy wykazały, że wprowadzane narzuty komunikacyjne są minimalne, a rozwiązanie nadaje się do implementacji podsystemów sensorycznych działających w czasie rzeczywistym.

**Słowa kluczowe:** układy sterowania robotów, podsystemy sensoryczne, przetwarzanie danych sensorycznych, struktury ramowe

Pozbawione zmysłów roboty mogą szybko i sprawnie wykonywać przydzielone im prace w dobrze ustrukturyzowanym, deterministycznym środowisku, jakim przykładowo jest fabryka. W przypadku dynamicznie zmieniającego się otoczenia niezbędne jest wyposażenie robota w zmysły umożliwiające rozpoznanie znajdujących się w pobliżu obiektów czy też zrozumienie zachodzących dookoła zdarzeń. Na zmysły robotów składają się sensory (układy złożone z czujników dostarczających informacji o pojawieniu się określonego bodźca oraz przetworników odpowiedzialnych za przekształcenie rejestrowanej wielkości fizycznej na inną, użyteczną dla dalszego przetwarzania, np. napięcie elektryczne) oraz oprogramowanie, które przetwarza (agreguje) odebrane z nich odczyty do postaci użytecznej w sterowaniu. Ważne jest, aby przetworzenie to było realizowane w możliwie jak najkrótszym czasie, dzięki czemu robot będzie mógł odpowiednio szybko reagować na zachodzące w jego otoczeniu zdarzenia. Znając te fakty oraz mając na uwadze rozmaitość sensorów oraz sposobów przetwarzania odbieranych z nich danych, problem implementacji, testowania oraz integracji podsystemów sensorycznych w sterownikach robotycznych okazuje się być nietrywialny. W latach osiemdziesiątych XX w. sformułowano tezę zwaną Paradoksem Moravca (Moravec, 1988), twierdzącą, iż wbrew tradycyjnym przeświadczeniom wysokopoziomowe rozumowanie wymaga niewielkiej mocy obliczeniowej, natomiast to niskopoziomowa percepcja (a więc podsystemy sensoryczne) oraz zdolności motoryczne (podsystemy motoryczne) wymagają dużej mocy obliczeniowej. Celem niniejszego artykułu jest zaprezentowanie narzędzia umożliwiającego tworzenie różnorodnych podsystemów sensorycznych.

Spośród pięciu zmysłów człowieka (wzrok, słuch, węch, smak i dotyk) niewątpliwie ten pierwszy jest najbardziej uniwersalny (a przez to również najbardziej skomplikowany). Z tego powodu wykorzystanie informacji wizyjnej odebranej z kamery jako odpowiednika zmysłu wzroku leżało od dawna w sferze zainteresowań robotyki. W kamery wyposażone były m.in. pierwsze eksperymentalne roboty mobilne *Stanford Cart* (1960–1980), *Shakey the Robot* (1966–1972) czy *CMU Rover* (Moravec, 1983). Należy zauważyć, iż robotyka, sztuczna inteligencja oraz wizja komputerowa nie tylko nie rozwijały się niezależnie, ale często stymulowały siebie nawzajem. Przykładowo, właśnie na potrzeby robotyki (konkretnie mobilnej) Moravec (1979) stworzył techniki ekstrakcji punktów charakterystycznych. Ponieważ do dzisiejszego dnia podsystemy wizyjne są najbardziej wymagającymi, a jednocześnie rokującymi największe nadzieje na rozwiązanie problemów percepcji na potrzeby sterowania czy nawigacji, dlatego w poniższej pracy zostały one wybrane jako przykłady zastosowania opracowanego rozwiązania.

## 1. Analiza wymagań

W ciągu ostatniej dekady można zaobserwować rozwój oprogramowania służącego do implementacji sterowników robotów. Korzenie tej problematyki sięgają jednak wczesnych lat siedemdziesiątych. Analizując rozwój technik i metod programowania robotów można zauważyć, iż uwaga początkowo skoncentrowana była na tworzeniu specjalizowanych języków, np. język WAVE (Paul, 1977). Z powodu małej elastyczności tego podejścia (niezbędne zmiany kompilatora, parsera etc. przy każdorazowej potrzebie dodania nowego rozkazu) uwaga została następnie przeniesiona na biblioteki modułów dla języków ogólnego przeznaczenia – przykładowe biblioteki to PASRO (od ang. *Pascal for Robots*) (Blume oraz Jakob, 1985) dla języka Pascal czy RCCL (ang. *Robot Control C Library*) (Hayward oraz Paul, 1986) dla języka C. Ze względu na pojawiające się nowe typy robotów i sensorów, a także coraz bardziej złożone aplikacje stopniowo wagi zaczęły nabierać takie czynniki jak wygoda korzystania z danej biblioteki czy powtórne wykorzystanie kodu. Z tego też powodu biblioteki, bazując na rozwiązaniach zaczerpniętych z inżynierii oprogramowania, stopniowo zaczęły ewoluować w programowe struktury ramowe, np. Orca (Brooks et al., 2007). W odróżnieniu od bibliotek (które są zestawem funkcji) struktury ramowe składają się z dwóch zasadniczych elementów: bibliotek modułów (funkcji tworzonych przez użytkowników na potrzeby

konkretnych robotów, aplikacji etc.) oraz ich właściwego trzonu (niezmiennej, niezależnej od zadania czy sprzętu części, zawierającej np. mechanizmy komunikacji pomiędzy modułami czy zarządzania konfiguracją). Dodatkowo struktury ramowe narzucają również wzorce zarówno tworzenia modułów użytkowników, jak ich wykorzystania i łączenia. Wraz ze strukturami ramowymi zaczęto dostarczać również zestawy narzędzi (ang. *toolchains*) użytecznych podczas implementacji nowych modułów czy całych sterowników – w poniższym przeglądzie rozwiązania takie nazwane zostały systemami. Należy tutaj podkreślić, iż mimo istnienia rozmaitych narzędzi do implementacji sterowników robotów, na specjalistycznym oprogramowaniu do projektowania i implementacji podsystemów sensorycznych robotów uwaga została skupiona stosunkowo niedawno.

W strukturze ramowej Player (Gerkey et al., 2001) układy sterowania robotów konstruowane są z tzw. sterowników (ang. *driver*), komunikujących się ze sobą poprzez predefiniowane, osobno kompilowane interfejsy. Projekt ten przez wiele lat dominował na polu robotyki mobilnej, głównie za sprawą dużej liczby sterowników oraz dwóch pobocznych projektów: Stage (symulatora 2D) oraz Gazebo (symulatora 3D), umożliwiających rozwijanie oraz testowanie aplikacji w środowisku symulacyjnym. Z kolei w zadaniach sterowania manipulatorami doskonale sprawdza się OROCOS (ang. *Open RObot COntrol Software*) (Bruyninckx, 2003), głównie dzięki swoim bibliotekom: RTT (ang. *Real-Time Toolkit*, służąca do symulacji i sterowania w czasie rzeczywistym) oraz KDL (ang. *Kinematics and Dynamics Library*, odpowiedzialna za rozwiązywanie zagadnień związanych z kinematyką i dynamiką manipulatorów). W OROCOS aplikacje składane są z niezależnych modułów zwanych komponentami. Struktura ramowa YARP (ang. *Yet Another Robot Platform*) (Metta et al., 2006) jest z kolei zestawem bibliotek, protokołów i narzędzi umożliwiających prostą dekompozycję systemu na niezależnie działające moduły, przy czym nie narzuca ona żadnych ram ani wzorców co do łączenia tych modułów. Analogiczne podejście (tzn. brak określonej z góry struktury) stosowane jest w jednym z najnowszych systemów do tworzenia rozproszonych, wieloprocesowych sterowników robotycznych: systemie ROS (ang. *Robot Operating System*) (Quigley et al., 2009). W ROS każdy element sterownika działa jako niezależny proces w sieci (węzeł, ang. *node*), a system dostarcza mechanizmów przesyłania komunikatów, zarządzania pakietami oraz monitorowania i wizualizacji stanu poszczególnych węzłów, kanałów komunikacyjnych, jak i całej aplikacji. Z kolei w programowej strukturze ramowej MRROC++ (Zieliński, 1999) postawiono na sztywną, z góry ustaloną, hierarchiczną strukturę współpracujących ze sobą procesów, podzielonych na warstwy – wyróżniono procesy zależne od sprzętu oraz zależne od zadania. Wydzielenie warstw charakteryzuje również CLARAty (ang. *Coupled-Layer Architecture for Robotic Autonomy*) (Nesnas, 2007), modułową strukturę ramową rozwijaną przez NASA dla celów sterowania robotami w przestrzeni kosmicznej – w CLARAty również wydzielono dwie warstwy (decyzyjną oraz funkcjonalną), jednak w odróżnieniu od MRROC++ pracujące w nich moduły nie muszą tworzyć określonej struktury.

Mimo różnorodności rozwiązań opracowanych w ramach tych kilku przytoczonych przykładów można wśród nich znaleźć cechy wspólne. Jedną z nich jest podejście modułowe, objawiające się w dekompozycji systemu sterowania na mniejsze, współpracujące ze sobą elementy. Niezależnie czy są to sterowniki w Player, moduły w CLARAty, procesy w ROS, czy komponenty w OROCOS, dekompozycja zwiększa ich wielokrotne użycie (gdyż ten sam element wykorzystany może być w różnych aplikacjach), a przez to również odporność tych systemów jako całości (bloki te są lepiej przetestowane, przy czym warte podkreślenia jest, iż moduły te często mogą być testowane niezależnie) oraz przyśpiesza czas projektowania i implementacji nowych aplikacji. Podejście to, mimo wymienionych zalet, posiada również wady, wśród których jedną z największych jest integracja. Przykładowo w ROS, mimo całej jego elastyczności oraz różnorodnych mechanizmów komunikacji, integracja podukładów sensorycznych (a w szczególności podsystemów wizyjnych) okazała się dla wielu użytkowników problematyczna. Z tego powodu autorzy niedawno rozpoczęli pracę nad nowym projektem Ecto (Willow Garage, 2011), gdzie uwaga została skupiona na wspieraniu organizacji obliczeń i przepływu danych w ramach jednego procesu – zauważono bowiem, że programiści, aby zmniejszyć narzuty komunikacyjne, implementowali podsystemy percepcji (a szczególnie percepcji wizyjnej) w ramach jednego procesu, czego już ROS nie wspierał.

Następnym aspektem jest konfiguracja systemu, a więc umożliwienie zmiany wartości parametrów niezbędnych do właściwej pracy poszczególnych elementów układu. W strukturach Player, OROCOS, ROS oraz MRROC++ istnieją mechanizmy wczytywania ustawień z plików konfiguracyjnych, jednak w przypadku układów wizyjnych podczas testowania z reguły potrzebne jest ich ręczne dostrajanie (zmiana progu podczas binaryzacji, ręczna selekcja wymaganych cech etc.). Był to jeden z powodów wydzielenia z MRROC++ projektu FraDIA (ang. *Framework for Digital Image Analysis*) (Kornuta, 2010), wyspecjalizowanej programowej struktury ramowej do implementacji podsystemów wizyjnych. W strukturze tej integralną częścią komponentu (wydzielono dwie klasy komponentów: źródła obrazów oraz zadania wizyjne) jest panel interfejsu graficznego. Mimo to ręczne oprogramowanie interfejsów (które dominuje w większości wspomnianych rozwiązań) jest uciążliwe – brakuje generycznych (a więc opartych o ideę szablonu) rozwiązań, które na podstawie właściwości (parametrów) danego komponentu potrafiłyby automatycznie wygenerować odpowiednie elementy interfejsu oraz mechanizmy wiążące obsługę zdarzeń ze zmianą tych właściwości.

Dodatkowo bardzo ważnym aspektem, na który często nie jest zwracana uwaga, jest organizacja samego projektu, a więc sposób zarządzania modułami oraz właściwym jego trzonem. W przypadku tak dużych projektów, jakimi są systemy robotyczne, odpowiednia organizacja niezbędna jest do zwiększenia liczby użytkowników i deweloperów. Brak niezależności pomiędzy trzonem a modułami oraz zadaniami powoduje bowiem znaczne problemy z wykorzystaniem danego narzędzia w innym laboratorium – każde laboratorium dysponuje z reguły unikalnym zestawem sprzętowym.

Dekompozycja samego repozytorium, gdzie przechowywane są kody źródłowe, związana jest więc ze skalowalnością projektu. Przykładowo, trzon struktury ramowej Player przechowywany był we wspólnym repozytorium plików razem ze wszystkimi sterownikami. Dlatego też jego autorzy, rozpoczynając pracę nad system ROS, postanowili rozdzielić te repozytoria, przy czym poszli o krok dalej od twórców OROCOS (gdzie wydzielono trzon, a komponenty przechowywane są w jednym repozytorium OCL [ang. *OROCOS Component Library*]) i zaproponowali rozproszoną organizację repozytoriów.

Na podstawie przeprowadzonej analizy można wyróżnić cechy, jakie powinno posiadać oprogramowanie do projektowania oraz implementacji podsystemów sensorycznych:

– dekompozycja zadań na małe, niezależne moduły,
– generyczne mechanizmy komunikacji pomiędzy nimi,
– mechanizmy i narzędzia ułatwiające składanie modułów w konkretne układy percepcji,
– trzon narzędzia niezależny od modułów oraz aplikacji,
– mechanizmy zarządzania konfiguracją i parametrami poszczególnych modułów,
– automatycznie generowane interfejsy graficzne do dostrajania parametrów.

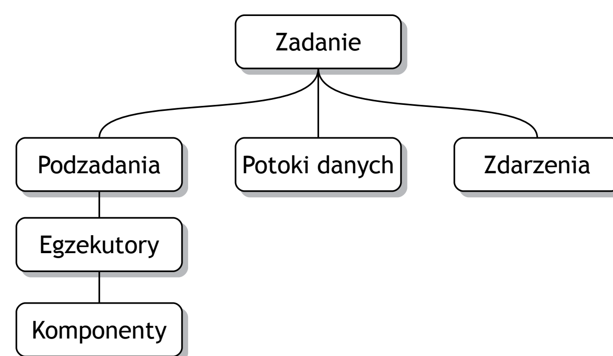
## 2. System DisCODe

Wyróżnione w wyniku przeprowadzonej analizy cechy posłużyły jako wytyczne podczas projektowania systemu DisCODe (od ang. *Distributed Component Oriented Data Processing*). Trzon systemu został napisany w języku C++, a implementacja została oparta na trzech paradygmatach: programowaniu zorientowanym komponentowo, programowaniu refleksyjnym oraz programowaniu generycznym.

Programowanie zorientowane komponentowo (ang. *Component Oriented Programing*) to technologia rozszerzająca programowanie obiektowe, dziedzicząca z niego takie cechy jak enkapsulację oraz dziedziczenie. Szyperski et al. (2002) definiuje komponent jako niezależnie wytworzony moduł programistyczny, którego wewnętrzne zależności oraz interfejsy zewnętrzne są wyraźnie określone, a który może być stosowany niezależnie od innych komponentów i reszty systemu. Warunkiem niezależności komponentów jest ich luźne powiązanie (ang. *loose coupling*), co oznacza, iż w żadnym z nich nie jest zakodowana informacja dotycząca komponentów, które w przyszłości będą z nim współpracowały, a właściwe kanały komunikacji tworzone są dynamicznie podczas startu aplikacji. W celu opracowania ogólnego mechanizmu zarządzania dowolną liczbą komponentów, każdy z nich kompilowany jest do osobnej biblioteki dynamicznej. Podczas uruchamiania konkretnego zadania DisCODe wczytuje komponenty z nim związane oraz dynamicznie tworzy kanały komunikacyjne pomiędzy nimi. Tutaj wykorzystano programowanie refleksyjne (Sobel oraz Friedman, 1996), którego idea opiera się na dynamicznym korzystaniu ze struktur danych, które nie muszą być jawnie określone w momencie implementacji oprogramowania, a które w czasie działania mogą dynamicznie zmieniać swoje właściwości i zachowanie. Z kolei programowanie generyczne (ang. *generic programming*) (Alexandrescu, 2001) pozwala na pisanie kodu programu bez wcześniejszej znajomości typów danych, na których kod ten będzie pracował. Oznacza to, iż na etapie implementacji opracowywane są szablony (ang. *templates*), na podstawie których kompilator na początku kompilacji generuje odpowiednie kody właściwe. Opracowano dwa generyczne mechanizmy komunikacji: potoki danych (ang. *data streams*), do przekazywania przetwarzanych danych pomiędzy komponentami oraz zdarzenia (ang. *events*), służące do przesyłania danych sterujących pracą poszczególnych komponentów.

Kombinacja tych paradygmatów dała teoretyczne podstawy do stworzenia mechanizmów pozwalających na dekompozycję dowolnego procesu percepcji na acykliczny, skierowany graf niezależnych, ale potrafiących ze sobą współpracować komponentów.

### 2.1. Dekompozycja zadania percepcji



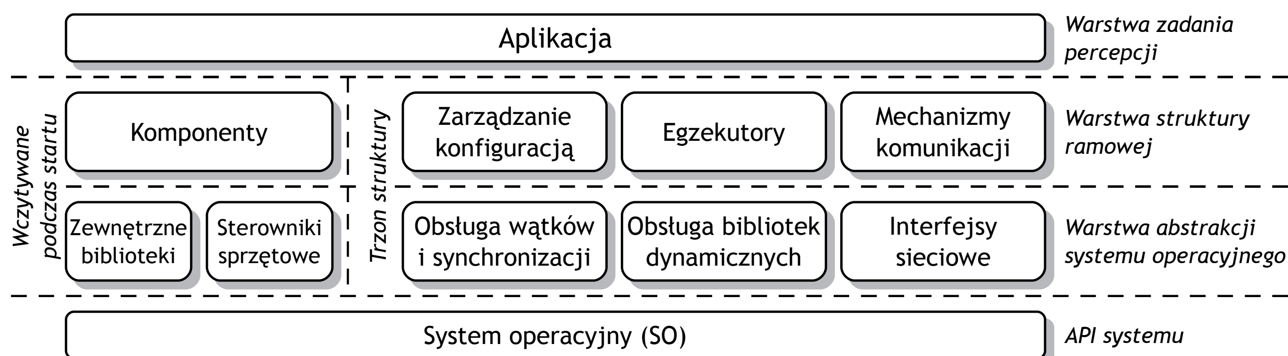

**Rys. 1.** Dekompozycja zadania percepcji
**Fig. 1.** Decomposition of the perception task

W DisCODe dany proces percepcji jest reprezentowany przez zadanie o strukturze pokazanej na rys. 1. Podstawowymi elementami zadań percepcji są komponenty, komunikujące się ze sobą za pomocą potoków danych oraz zdarzeń. Komponenty organizowane są w wątki zarządzające ich pracą zwane egzekutorami.Umożliwia to przyspieszenie działania całego procesu percepcji poprzez zrównoleglenie obliczeń. Kolejność obsługi komponentów może być ustalona bądź odgórnie (na poziomie pliku konfiguracyjnego), bądź dynamicznie na podstawie zdarzeń przychodzących do komponentów. Egzekutory z kolei przypisane są konkretnym podzadaniom. Podział na podzadania jest istotny z punktu widzenia pracy systemu jako podsystemu sensorycznego układu sterowania robota – dzięki niemu można zaplanować wiele różnych potoków przetwarzania, a w danym momencie uruchamiać tylko te, które są potrzebne. Zatrzymanie podzadania powoduje wstrzymanie wszystkich jego egzekutorów, a więc całkowicie odciąża system.

Każde zadanie przechowywane jest w postaci pliku XML (o strukturze odpowiadającej rys. 1), wczytywanego przez system podczas startu. W zależności od zadania system wczytuje biblioteki dynamiczne zawierające odpowiednie komponenty, uruchamia egzekutory, podzadzania oraz tworzy połączenia komunikacyjne (łączy potoki danych oraz tworzy mechanizmy obsługi zdarzeń).

### 2.2. Architektura systemu

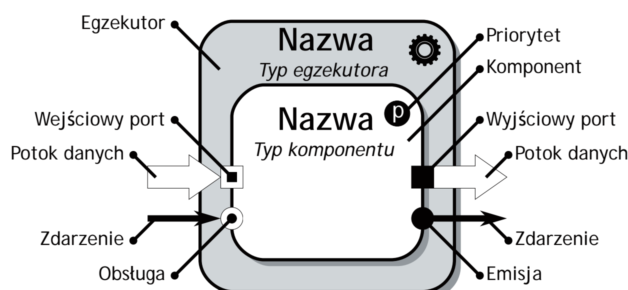
Dotychczas przedstawiono system z punktu widzenia osoby wykorzystującej dostarczone komponenty i zadania. Drugą




**Rys. 2.** Podział DisCODe na warstwy
**Fig. 2.** DisCODe layers decomposition

grupą użytkowników są programiści trzonu systemu – z ich punktu widzenia ważna jest odpowiednia architektura, jasny podział na moduły odpowiedzialne za poszczególne fragmenty systemu, a także łatwość utrzymania i rozwoju kodu. Na rys. 2 przedstawiono podział systemu na warstwy o coraz wyższym stopniu abstrakcji. Na dole znajduje się warstwa systemu operacyjnego i sterowników urządzeń (faktycznie niebędąca częścią systemu, została jednak pokazana dla zachowania porządku). Ponad nią jest warstwa abstrahująca różne aspekty systemu operacyjnego. Obecnie DisCODe działa zarówno pod kontrolą systemów z rodziny Linux, jak i Windows. W kolejnej warstwie znajdują się właściwe moduły zarządzania wątkami i komponentami, a także same komponenty, wczytywane dynamicznie podczas startu aplikacji. Na samej górze znajduje się warstwa zadania, uruchamiająca właściwe zadanie percepcji.

## 2.3. Komponent




**Rys. 3.** Graficzna reprezentacja elementów zadania
**Fig. 3.** Graphical representation of task elements

Jak już zostało zaznaczone wcześniej, poszczególne kroki zadania percepcji implementowane są w postaci oddzielnych komponentów, uruchamianych w obrębie różnych egzekutorów oraz komunikujących się ze sobą poprzez potoki danych oraz zdarzenia. Na rys. 3 zaprezentowano opracowaną dla tych elementów notację graficzną.

**Potoki danych** służą do przesyłania przetwarzanych danych pomiędzy komponentami. Wyróżniono dwa rodzaje portów danych: **porty wejściowe**, przez które komponent otrzymuje dane, oraz **porty wyjściowe**, przez które dane są wysyłane. Potok danych łączy jeden port wyjściowy z dowolną liczbą portów wejściowych, a każdy komponent może mieć dowolną liczbę obu typów portów.

**Zdarzenia** służą do przesyłania komunikatów sterujących pracą (aktywacją) komponentów. Przykładowym komunikatem tego typu jest informacja, iż komponent zakończył przetwarzanie danych oraz zapisał wynik do jednego z jego portów wyjściowych. Może wtedy dokonać **emisji** informującego o tym zdarzenia, które odebrane zostanie przez wszystkie komponenty, które potrafią je **obsłużyć**. Kolejność obsługi komponentów pracujących w ramach jednego egzekutora wynika z ich **priorytetów**.

### 2.3.1. Kategorie komponentów

Wyróżniono cztery kategorie komponentów:

– Źródło (ang. *Source*): komponenty odpowiedzialne za pobieranie danych z urządzeń zewnętrznych (takich jak kamery czy mikrofony) lub plików (filmy, zdjęcia) i udostępnianie ich innym komponentom. Zazwyczaj źródło posiada jeden port wyjściowy.

– Procesor (ang. *Processor*): ich zadaniem jest przetwarzanie danych otrzymanych od innych komponentów i udostępnianie wyników na zewnątrz (poprzez porty wyjściowe). Procesory muszą zawierać co najmniej po jednym wejściowym i wyjściowym porcie danych.

– Odpływ (ang. *Sink*): mogą być wykorzystane do zapisu wyników działania procesu przetwarzania do plików (np. pojedynczych obrazów lub ich sekwencji w przypadku obrazów odbieranych z kamery) lub wyświetlania rezultatów na ekranie. Posiadają jedynie porty wejściowe.

– Pośrednik (ang. *Proxy*): ich zadaniem jest komunikacja z zewnętrznymi systemami. W szczególności, gdy DisCODe pełni rolę podsystemu sensorycznego, pośredniki odbierają polecenia oraz odsyłają wyniki działania zadania percepcji do właściwego sterownika robota, opartego np. na MRROC++ czy ROS.

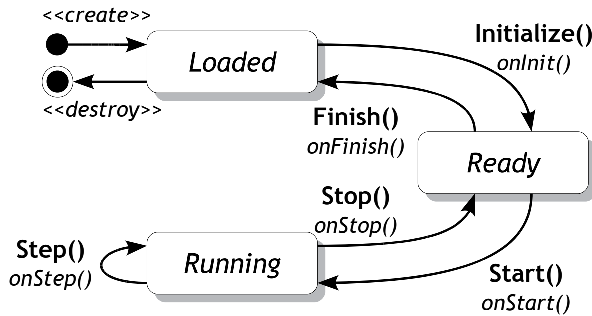
### 2.3.2. Stany komponentu

Na rys. 4 pokazany został diagram stanów komponentu z zaznaczonymi metodami odpowiedzialnych za przechodzenie pomiędzy stanami. Komponent może znajdować się w następujących stanach:

– Wczytany (ang. *Loaded*): komponent załadowany do systemu, ale jeszcze nie zainicjowany.

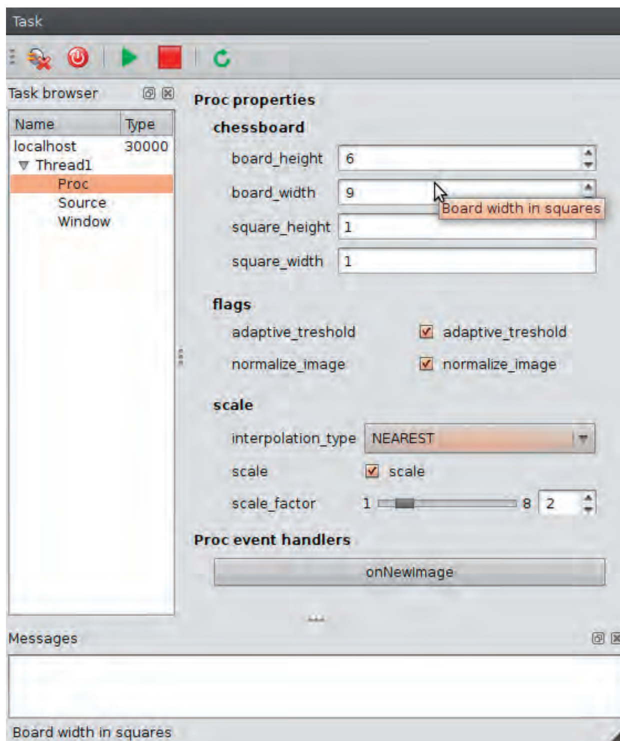
– Gotowy do pracy (ang. *Ready*): komponent zainicjowany (zmienne zostały zainicjalizowane, utworzono porty oraz podpięto funkcje obsługi sygnałów), gotowy do pracy.

– Pracujący (ang. *Running*): komponent w trakcie pracy (przetwarzający dane, obsługujący zdarzenia etc.).




**Rys. 4.** Diagram stanów komponentu
**Fig. 4.** Component state diagram

Na rys. 4 metody publiczne, dostępne dla egzekutorów, zostały pogrubione. Są one odpowiedzialne za wywołanie odpowiadających im prywatnych metod (oznaczonych kursywą), których ciało musi być zdefiniowane podczas implementacji danego komponentu. Jeżeli wykonanie prywatnej metody zakończy się poprawnie, publiczna metoda zmienia stan komponentu na wskazany strzałką.




**Rys. 5.** Przykład wygenerowanego panelu interfejsu graficznego
**Fig. 5.** An example of generated GUI panel

### 2.3.3. Właściwości komponentów

Poza mechanizmami komunikacji równie ważna jest konfiguracja poszczególnych komponentów, a więc ustawienie parametrów (zwanych właściwościami, ang. *properties*) niezbędnych do ich poprawnej pracy. Zaproponowany w DisCODe mechanizm umożliwia ustawianie właściwości zarówno przed uruchomieniem aplikacji (wartości mogą być zapisane w plikach XML z zadaniami), jak również w czasie jej działania. W tym celu uruchamiana jest oddzielna aplikacja interfejsu graficznego, oparta o strukturę ramową Qt (Blanchette oraz Summerfield, 2008). Aplikacja ta po podłączeniu się do aktualnie działającego zadania percepcji odczytuje jego konfigurację oraz na podstawie właściwości wszystkich komponentów generuje automatycznie okna i kontrolki umożliwiające ich zmianę. Okno interfejsu wygenerowane dla przykładowego zadania percepcji pokazane zostało na rys. 5. Warte podkreślenia jest, iż poza różnymi typami kontrolek, generowanymi w zależności od typu danej właściwości, tworzone są wszystkie mechanizmy związane z ze zmianą ich wartości, a sam użytkownik nie musi pisać żadnego kodu.

## 2.4. Egzekutory

Każdy egzekutor pracuje w ramach danego zadania jako oddzielny wątek i jest odpowiedzialny za aktywację należących do niego komponentów: wywoływanie metod związanych z ich stanem, przekazywanie zdarzeń etc. **Priorytet** komponentu (rys. 3) jest związany z kolejnością, w jakiej komponenty mają być obsługiwane. Komponent o najniższym numerze staje się **głównym komponentem** danego egzekutora (a więc będzie obsługiwany jako pierwszy).

### 2.4.1. Stany egzekutora

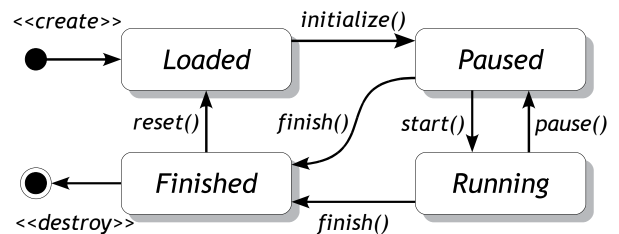
Rys. 6 przestawia diagram stanów egzekutora. Jego stany zdefiniowane są następująco:

– Utworzony (ang. *Loaded*): egzekutor aktywny, aczkolwiek jego komponenty nie zostały jeszcze zainicjalizowane.

– Zatrzymany (ang. *Paused*): komponenty zostały zainicjalizowane, egzekutor gotowy do pracy.

– Pracujący (ang. *Running*): egzekutor oraz jego komponenty pracują.

– Zakończony (ang. *Finished*): egzekutor zakończył pracę.




**Rys. 6.** Diagram stanów egzekutora
**Fig. 6.** Executor state diagram

### 2.4.2. Tryby pracy egzekutorów

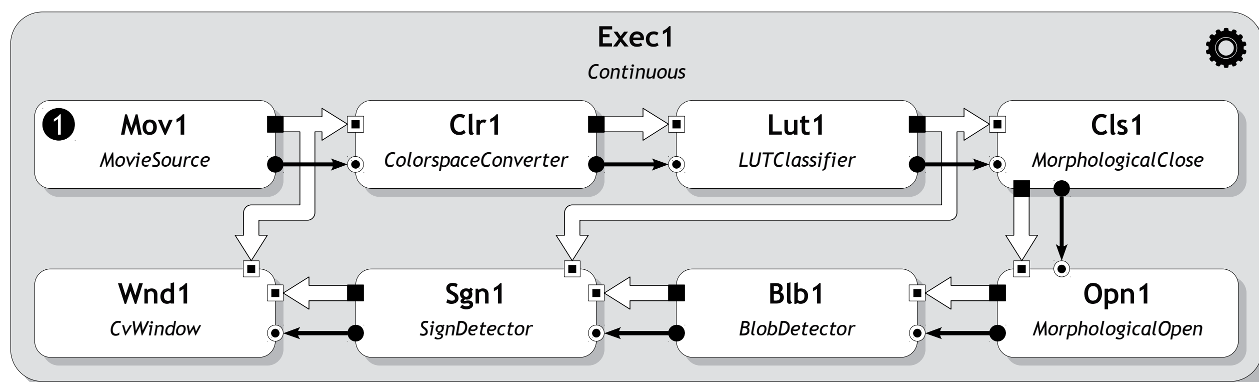
Zaimplementowano trzy tryby pracy egzekutorów:

– Ciągły (ang. *Continuous*): Aktywuje jego główny komponent od razu po zakończeniu obsługi zdarzeń należących do niego komponentów. Wszystkie komponenty, poza głównym, aktywowane są jedynie poprzez obsługę otrzymanych zdarzeń.

– Periodyczny (ang. *Periodic*): Główny komponent aktywowany jest okresowo (co dany kwant czasu), pozostałe komponenty działają w trybie obsługi zdarzeń.

– Pasywny (ang. *Passive*): W tym trybie egzekutor aktywuje komponenty tylko w przypadku otrzymania zdarzenia.

Pasywny egzekutor powinien być wykorzystany w zadaniach, które zawierają składniki mogące się aktywować




**Rys. 7.** Potok przetwarzania zadania testowego
**Fig. 7.** Test task execution flow

samodzielnie, np. do obsługi karty akwizycji obrazów z kamery, która o tym, że odebrała nową klatkę z kamery, informuje wysyłając przerwanie sprzętowe.

### 2.5. Organizacja systemu

Poza repozytorium przechowującym trzon struktury, w systemie DisCODe wyróżniono oddzielne repozytoria dla bibliotek komponentów (ang. *DisCODe Component Library*, DCL), w których przechowywane są zestawy komponentów związanych z różnymi sensorami lub zadaniami. Przykładowo, w CameraUniCap przechowywane są komponenty związane z kamerami analogowymi, w CameraGigE z kamerami cyfrowymi, Audio zawiera komponenty użyteczne przy przetwarzaniu sygnałów dźwiękowych, a CvBasic oraz CvBlobs zawierają zestawy podstawowych komponentów do przetwarzania obrazów z biblioteki OpenCV (Bradski oraz Kaehler, 2008). Oprócz samych komponentów w DCL zawarte mogą być także specyfikacje zadań percepcji (pliki XML), bądź dodatkowe biblioteki niezbędne do działania komponentów. Warte podkreślenia jest, iż wraz z trzonem DisCODe dostarczane są skrypty zarówno do tworzenia nowych repozytoriów dla bibliotek, jak i do generowania szkieletów komponentów.

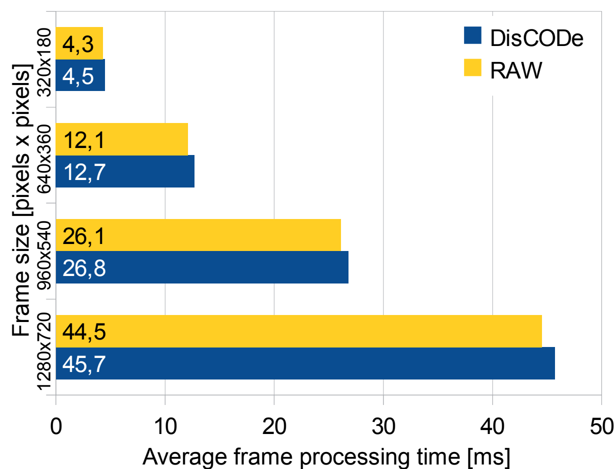
## 3. Zadanie testowe

W celu przetestowania poprawności działania systemu, zbadania opóźnień wprowadzanych przez mechanizmy komunikacji oraz pokazania jego elastyczności przygotowane zostało zadanie testowe. Jego działanie polegało na oznaczaniu w sekwencji wideo znaków zakazu postoju i zatrzymywania się. Procedura testowa polegała na porównaniu szybkości działania tego zadania zaimplementowanego w postaci kilku komponentów DisCODe z implementacją monolityczną (tzw. RAW), gdzie wszystkie funkcje realizowane były w głównej pętli programu.

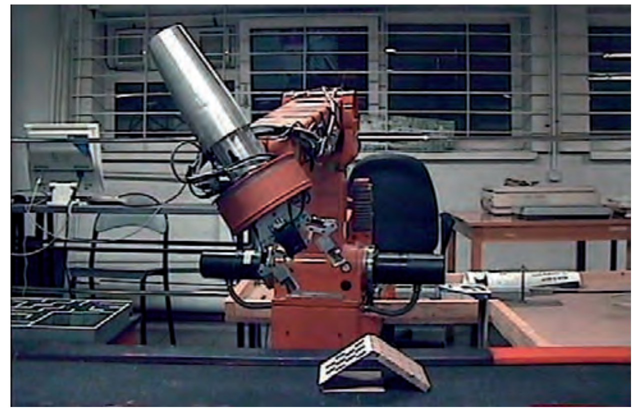
Rys. 7 przedstawia potok przetwarzania danych w zadaniu. Wyróżnione zostało w nim osiem różnych komponentów. Komponent **Mov1** odpowiedzialny jest za pobranie ramki z sekwencji wideo, **Clr1** za konwersję przestrzeni barw z RGB na HSV, a **Lut1** za klasyfikację kolorów (wyróżniono kolory czerwony i niebieski). Dwa kolejne komponenty **Cls1** i **Opn1** dokonują operacji morfologicznych w celu usunięcia szumów. Następnie tak przetworzony obraz trafia na wejście komponentu **Blb1** realizującego segmentację obrazu, z którego dane, już w postaci listy segmentów, trafiają do komponentu **Sgn1** dokonującego właściwej ich analizy. Komponent ten na wejściu otrzymuje także obraz wynikowy z klasyfikatora kolorów, ponieważ określenie, czy dane segmenty są znakiem czy nie, wymaga analizy zarówno ich kształtu, jak i koloru. Ostatni komponent **Wnd1** odpowiedzialny jest za wyświetlenie obrazu wejściowego wraz z nałożonymi na niego kształtami opisującymi wykryte znaki na ekranie (przykładowa ramka przedstawiona jest na rys. 8).

**Rys. 8.** Rezultat działania zadania testowego z zaznaczonymi wykrytymi znakami
**Fig. 8.** Test task result with detected signs marked

Kryterium przy porównaniu przedstawionego systemu ze specjalizowaną aplikacją wykonującą analogiczne zadanie był czas potrzebny na przetworzenie pojedynczej ramki obrazu. W ramach porównania wykonano serię pomiarów, podając na wejście ten sam film w czterech różnych rozdzielczościach, od $320 \times 180$ do $1280 \times 720$ pikseli. Pomiary zostały przeprowadzone na komputerze z AMD Phenom II X3 710, z trzyrdzeniowym procesorem o częstotliwości taktowania 2,6 GHz, 4 GB pamięci RAM, działającym pod kontrolą systemu Ubuntu 10.04. Na rys. 9 przedstawiono średnie czasy przetwarzania dla każdej z przetestowanych rozdzielczości obrazu w obu wersjach implementacyjnych (DisCODe oraz RAW). Na wykresach widać, iż narzut komunikacyjny struktury ramowej jest najwyższy w przypadku małych obrazów i maleje wraz z wielkością obrazu (od 5 do około 2,6 procent), co można uznać za dopuszczalne.




**Rys. 9.** Średni czas przetwarzania ramki obrazu
**Fig. 9.** Average frame processing time



**Rys. 10.** Manipulator IRp-6 nad szachownicą podczas serwowizji
**Fig. 10.** The IRp-6 manipulator above the chessboard during visual servoing

## 4. Serwowizja

Sterownik robota wykorzystujący sprzężenie wizyjne wymaga dwóch głównych modułów: podsystemu sterowania, odpowiedzialnego za realizację danego zadania, generację trajektorii etc. oraz podsystemu wizyjnego, zawierającego algorytmy przetwarzania i analizy obrazów, którego celem jest wyodrębnienie i agregacja danych do postaci użytecznej w sterowaniu. W serwomechanizmach wizyjnych (VS, od ang. *visual servoing*) (Chaumette oraz Hutchinson, 2008) pozycja danego obiektu, obliczana przez podsystem wizyjny, jest bezpośrednio wykorzystywana przez podsystem sterowania do generowania odpowiedniego ruchu robota, istnieje więc wizyjna pętla sprzężenia zwrotnego. Staniak oraz Zieliński (2010) zaproponowali szereg kryteriów klasyfikacji serwomechanizmów wizyjnych, wśród których najważniejsze jest związane z wzajemną relacją pomiędzy robotem a kamerą: wyróżniono mobilne kamery (EIH, od ang. *Eye-In-Hand*), zamontowane bezpośrednio na manipulatorze (lub robocie mobilnym) oraz kamery nieruchome (SAC, od ang. *Stand-Alone-Camera*), mogące obserwować zarówno poruszający się obiekt, jak i robota. System DisCODe, pełniąc rolę podsystemu wizyjnego sterownika opartego o programową strukturę ramową MRROC++ (Zieliński, 1999), został zweryfikowany w obu przypadkach (rys. 10). Warte podkreślenia jest, iż w eksperymentach wykorzystano nie tylko różne kamery (analogową EIH zintegrowaną z chwytakiem manipulatora oraz cyfrową SAC zawieszoną nad sceną), ale również stosowano zamiennie różne obiekty zainteresowania (widoczną na rys. 10 małą szachownicę oraz niebieską piłeczkę), co wymagało innych zadań percepcji.

## 5. Podsumowanie

W artykule dokonano przeglądu rozwoju oprogramowania służącego do implementacji sterowników robotów. W wyniku przeprowadzonej analizy opracowano listę wymagań, jakie powinno spełniać narzędzie do tworzenia podsystemów sensorycznych. Następnie omówiono system DisCODe oraz zaprezentowano testy wydajnościowe, potwierdzające poprawność implementacji. Przytoczono również aplikację robotyczną, w której informacja wizyjna wykorzystana została w pętli sprzężenia zwrotnego do sterowania rzeczywistym manipulatorem. Ponieważ wizja jest jednym z najbardziej złożonych zmysłów robotów, poprawne działanie rozwiązania weryfikuje jego przydatność w zastosowaniach czasu rzeczywistego.

Zaprezentowane narzędzie jest z powodzeniem wykorzystywane w różnorodnych aplikacjach, m.in. do rozpoznawania i lokalizacji elementów przez robota konstruującego przestrzenne budowle z klocków czy rozpoznawania układów dłoni na cele proaktywnej, multimodalnej komunikacji człowieka z robotem (Kornuta oraz Zieliński, 2011).

Warto również wspomnieć, że system DisCODe został z powodzeniem wykorzystany jako narzędzie dydaktyczne na przedmiocie *Computer Vision* w ramach EMARO (European Master on Advanced Robotics), zintegrowanych studiów uzupełniających, prowadzonych przez trzy europejskie i trzy azjatyckie instytucje. Przedstawione narzędzie umożliwiło studentom praktyczne zapoznanie się z różnorodnymi algorytmami i problemami wizji komputerowej, od kalibracji kamery, przez przetwarzanie obrazu i ekstrakcję cech, aż po klasyfikację wzorców (rozpoznawanie ręcznego pisma).

Plany rozwoju DisCODe obejmują zastosowanie metod i narzędzi inżynierii opartej na modelach (ang. *Model Driven Engineering*, MDE) (Schmidt, 2006) do opracowania narzędzia umożliwiającego graficzne projektowanie potoków przetwarzania danych.

## Bibliografia


Alexandrescu A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.

Blanchette J., Summerfield M.: *C++ GUI Programming with Qt 4*, Prentice Hall, 2008.

Blume C., Jakob W.: *PASRO: Pascal for Robots*, Springer-Verlag, 1985.

Bradski G., Kaehler A.: *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly, 2008.

Brooks A., Kaupp T., Makarenko A., Williams S., Orebäck A.: *Orca: A Component Model and Repository*, 231–251. Springer, 2007.

Bruyninckx H.: *The real-time motion control core of the OROCOS project*, Proceedings of the International Conference on Robotics and Automation (ICRA), 2766–2771, 2003.

Chaumette F.,Hutchinson S.: *The Handbook of Robotics*, chapter Visual Servoing and Visual Tracking, 563–583. Springer, 2008.

Gerkey B. P., Vaughan R. T., Støy K., Howard A., Sukhatme G. S., Mataric M. J.: *Most Valuable Player: A Robot Device Server for Distributed Control*, Proceedings of the International Conference on Intelligent Robots and Systems (IROS), 1226–1231, 2001.

Hayward V., Paul R. P.: *Robot manipulator control under unix RCCL: A robot control C library*, "International Journal of Robotics Research", 5(4):94–111, 1986.

Kornuta T.: *Application of the FraDIA vision framework for robotic purposes*, Proceedings of the International Conference on Computer Vision and Graphics (ICCVG), vol. 6375 "Lecture Notes in Computer Science", 65–72. Springer Berlin/Heidelberg, 2010.

Kornuta T., Zieliński C.: *Behavior-based control system of a robot actively recognizing hand postures*, 15th IEEE International Conference on Advanced Robotics (ICAR), 265–270, 2011.

Metta G., Fitzpatrick P., Natale L.: *YARP: Yet Another Robot Platform*, "International Journal on Advanced Robotics Systems", 3(1):43–48, 2006.

Moravec H.: *Visual mapping by a robot rover*, Proceedings of the 6th International Joint Conference on Artificial Intelligence, 599–601, 1979.

Moravec H.: *The Stanford Cart and the CMU Rover*, Proceedings of the IEEE, 71(7):872–884, 1983.

Moravec H.: *Mind Children: The Future of Robot and Human Intelligence*, Harvard University Press, 1988.

Nesnas I.: *The CLARAty project: Coping with hardware and software heterogenity*, Brugali D., edytor, *Software Engineering for Experimental Robotics*, vol. 30 *Springer Tracts in Advanced Robotics*, 31–70. Springer, 2007.

Paul R.: *WAVE – a model based language for manipulator control*, "The Industrial Robot", 10–17, 1977.

Quigley M., Gerkey B., Conley K., Faust J., Foote T., Leibs J., Berger E., Wheeler R., Ng A. Y.: *ROS: an open-source Robot Operating System*, Proceedings of the International Conference on Robotics and Automation (ICRA), 2009.

Schmidt D.: *Model-driven engineering*, IEEE Computer, 39(2):25–31, 2006.

Sobel J. M., Friedman D. P.: *An introduction to reflection-oriented programming*, 1996.

Staniak M., Zieliński C.: *Structures of visual servos*, "Robotics and Autonomous Systems", 58(8):940–954, 2010.

Szyperski C., Gruntz D., Murer S.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Professional, 2nd edition, 2002.

Willow Garage. *Ecto – a framework for perception*, 2011, `http://ecto.willowgarage.com/`.

Zieliński C.: *The MRROC++ System*, First Workshop on Robot Motion and Control, RoMoCo, 147–152, 1999.

## DisCODe: a component framework for sensory data processing

**Abstract:** In order to cope with an unstructured and dynamically changing environment service robots must be equipped with many diverse sensors and their control systems must be able to process the perceived information about their surroundings in real-time in order to react appropriately to the occurring events. The paper seeks to develop universal software facilitating the development of any sensor subsystems, being equivalent of diverse animal or robot senses. Performed tests proved that the introduced communication overhead can be neglected in comparison to the profits resulting from task decomposition into many simple components, as well as the overall usefulness of the created solution in real-time applications.

**Keywords:** robot control systems, sensory data processing, programming frameworks

**mgr inż. Tomasz Kornuta**

Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W 2003 roku uzyskał tytuł inżyniera, w 2005 r. tytuł magistra inżyniera. Od 2008 r. pracuje na etacie asystenta w Instytucie Automatyki i Informatyki Stosowanej (IAiIS); od 2009 r. pełni funkcję Kierownika Laboratorium Podstaw Robotyki. Od 2005 r. w ramach doktoratu prowadzi badania związane z projektowaniem systemów robotycznych wykorzystujących paradygmat aktywnego czucia do analizy otoczenia. Jego główne zainteresowania naukowe obejmują wykorzystanie informacji wizyjnej w robotyce.

*e-mail: tkornuta@ia.pw.edu.pl*

**mgr inż. Maciej Stefańczyk**

Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W 2010 r. uzyskał tytuł inżyniera, w 2011 r. tytuł magistra inżyniera, oba z wyróżnieniem. W 2011 r. rozpoczął prace nad doktoratem dotyczącym zastosowania aktywnej wizji wraz z systemami opartymi na bazie wiedzy w systemie sterowania robotów. Główne zainteresowania naukowe obejmują zastosowanie informacji wizyjnej, zarówno w robotyce, jak i w systemach rozrywki komputerowej.

*e-mail: stefanczyk.maciek@gmail.com*