JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 13 (2019) 349-353
Wysłane: 2019-09-14
Przyjęte: 2019-09-22

Badanie wzorca architektonicznego Entity-component-system zaprojektowanego z wykorzystaniem techniki Data-oriented design

Dawid Masiukiewicz*, Daniel Masiukiewicz*, Jakub Smołka
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Celem niniejszego artykułu jest prezentacja i przetestowanie architektury Entity-component-system zaprojektowanej w oparciu o dane. Rozwiązanie pozwala usprawnić proces tworzenia aplikacji jednocześnie zwiększając jej wydajność. Do badań przygotowana została aplikacja testowa w oparciu o autorskie rozwiązania. Na łamach artykułu przedstawiono porównanie badanych technik z programowaniem zorientowanym obiektowo.

Słowa kluczowe: tworzenie gier; DOD; ECS

*Autor do korespondencji.

Adresy e-mail: wismerchil@interia.eu, danielmz25@gmail.com

Research of an Entity-component-system architectural pattern designed with using of Data-oriented design technique

Dawid Masiukiewicz*, Daniel Masiukiewicz*, Jakub Smołka
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The purpose of this article is to present and evaluate Entity-component-system architecture designed based on data. The solution allows for improving application development process and increasing its efficiency. A test application was prepared for research using custom solutions. Evaluated techniques was compared with object-oriented programming in the article.

Keywords: game development; DOD; ECS

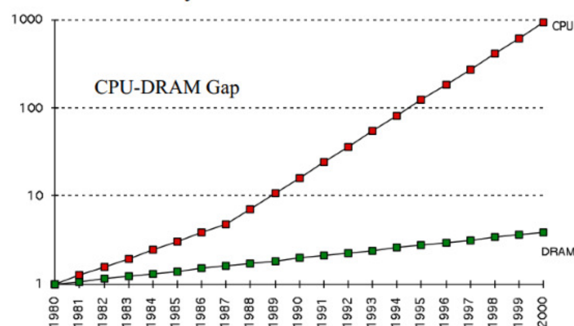
*Corresponding author.

E-mail addresses: wismerchil@interia.eu, danielmz25@gmail.com

1. Wstęp

Prężny rozwój gier komputerowych na przestrzeni ostatnich lat znacząco zwiększył zapotrzebowanie na nowe technologie wspomagające proces produkcji oraz wydajność aplikacji. Również technologia wewnątrz podzespołów komputera uległa znaczącym zmianom. Kolejne lata przyniosły znaczący wzrost mocy obliczeniowej procesora podczas gdy wydajność pamięci komputera wzrosła bardzo niewiele (Rys. 1). Programowanie równoległe może jeszcze zwiększyć narzut czasu dostępu do pamięci na wydajność programu. W wielu gałęziach programowania projektowanie oparte na danych (DOD) wykorzystywane jest od lat w celu zwiększenia wydajności. Dodatkowym problemem wiążącym się z rozwojem branży gier jest rosnący rozmiar nowych projektów zwiększający wymagania na nowe technologie przyśpieszające tworzenie gry, zapewniając dodatkowo większe bezpieczeństwo oraz zmniejszające koszty produkcji.

Celem niniejszego artykułu jest przedstawienie i przetestowanie rozwiązania Entity-component-system (ECS) opartego o DOD. Omówione zostaną podstawowe korzyści oraz ograniczenia płynące z wykorzystania podejścia, a także zostanie dokonane porównane go z aktualnie najpopularniejszym programowaniem zorientowanym obiektowo. Badania przedstawione w artykule oparte zostały o autorskie rozwiązania. Wykorzystany został język D będący statycznie typowanym językiem ogólnego przeznaczenia [1].



Rys. 1. Zestawienie wydajności procesora z wydajnością pamięci RAM w kolejnych latach [2]

2. Programowanie obiektowe

Programowanie obiektowe (ang. object-oriented programming, OOP) to paradygmat programowania, w którym aplikacje opisuje się za pomocą obiektów. Obiekt

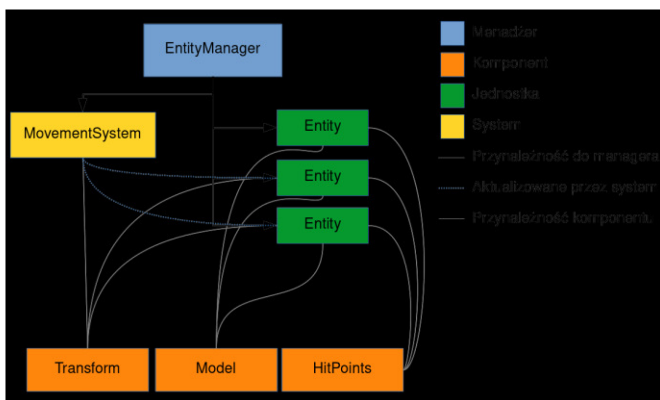
to połączenie danych oraz metod. Zazwyczaj obiekty tworzone są jako abstrakcja rzeczywistego obiektu klasyfikując dane oraz funkcje w sposób logiczny dla człowieka [3]. W parze z programowaniem obiektowym idzie projektowanie zorientowane obiektowo (ang. object-oriented design, OOD). Choć programowanie obiektowe jest jedynie wzorcem, nienarzucającym wykorzystywanych w programie technik, jest wysoce niekompatybilne z projektowaniem zorientowanym na dane.

3. Entity-component-system

ECS jest wzorcem architektonicznym używanym przede wszystkim przy tworzeniu gier komputerowych [4]. Jest zgodny z zasadą "kompozycja ponad dziedziczenie" minimalizując redundancję danych. Wzorzec składa się z trzech głównych elementów:

- Komponent (ang. component) – surowe dane opisujące pojedynczy aspekt obiektu oraz jego interakcję ze światem. Komponent implementowany jest zazwyczaj za pomocą struktur danych. Działa jak etykieta opisująca właściwość obiektu
- Jednostka (ang. entity) – obiekt ogólnego przeznaczenia. Zazwyczaj składa się jedynie z unikalnego identyfikatora, implementowanego jako liczba całkowita. Jednostka składa się z komponentów, które jednak nie muszą znajdować się w pamięci tam gdzie jednostka. Dostęp do komponentów odbywa się przy pomocy identyfikatora. Komponenty zawarte w jednostce określają jej typ
- System – zawiera logikę programu. Wykonuje akcje dla każdej jednostki posiadającej typ o tym samym aspekcie co system. Systemy działają niezależnie od siebie.

Wzorzec eliminuje problem programowania obiektowego związanego z głębokimi oraz szerokimi hierarchiami dziedziczenia. Zazwyczaj wszystkie kluczowe elementy tego rozwiązania spajane są za pomocą menadżera jednostek zarządzającego jednostkami oraz systemami. Na rysunku 2 przedstawiono przykładową strukturę aplikacji. System aktualizuje jedynie te jednostki, które posiadają komponenty przypisane również do systemu.



Rys. 2. Przykładowa struktura aplikacji wykorzystująca wzorzec ECS

Tworzenie aplikacji ogranicza się do dodawania nowych komponentów oraz programowania logiki systemów. ECS jest wysoce kompatybilny z projektowaniem opartym na danych.

4. Data oriented design

Projektowanie zorientowane na dane ma na celu zwiększyć wydajność programu poprzez zmniejszenie czasu oczekiwania na pamięć. Ponieważ procesor pobiera dane blokowo, gdzie rozmiar bloku to zazwyczaj 32B bądź 64B, wykorzystanie pojedynczej zmiennej spowoduje wczytanie redundantnych danych [5]. Czas oczekiwania od momentu wysłania informacji o zapotrzebowaniu na dane do pamięci RAM do ich dotarcia jest kilka rzędów wyższy niż standardowe operacje [6]. Celem zmniejszenia wyżej wymienionego problemu w procesorach montowana jest wielopoziomowa pamięć podręczna [7]. Każdy kolejny poziom cechuje się większym rozmiarem oraz większym czasem dostępu. Najniższy poziom posiada zazwyczaj jedynie kilkukrotnie dłuższy czas dostępu niż rejestry procesora. Ponieważ jednak programy wykorzystują ogromne ilości danych, prawdopodobieństwo, że potrzebne dane nie znajdują się na niższych poziomach pamięci podręcznej jest wysokie.

Procesory posiadają jeszcze jeden istotny mechanizm mający na celu zminimalizowanie wpływu narzutu dostępu do pamięci na aplikację. Technologia przewidywania kolejnych instrukcji programu pozwala wysyłać do pamięci żądanie jeszcze przed instrukcją, która będzie ich potrzebować. Aby jednak technika ta działała poprawnie wymaga nieskomplikowanego schematu wczytywania kolejnych danych, tak aby procesor mógł przewidzieć, które dane powinien wczytać. Zazwyczaj proces optymalizacji pod kątem tej funkcjonalności sprowadza się do umieszczania danych w pamięci ciągłej, gdzie procesor wykorzystuje wszystkie informacje jedna po drugiej [8].

Projektowanie zorientowane na dane opiera się więc głównie na dwóch założeniach.

1) Należy maksymalnie zmniejszyć redundancję danych tak, aby wszystkie dane wewnątrz bloku wczytanego przez procesor były wykorzystywane w aktualnych obliczeniach
2) Należy umieścić dane jedne obok drugich posortowane w kolejności w jakiej będą wykorzystane.

Pojedynczy blok danych

	Podejście obiektowe	Projektowanie zorientowane na dane
0x00-0x03		Obiekt A (pierwsze 4 bajty)
0x04-0x07	Obiekt A	Obiekt B (pierwsze 4 bajty)
0x08-9x0b		Obiekt C (pierwsze 4 bajty)
0x0c-0x0f		Obiekt D (pierwsze 4 bajty)
0x10-0x13		Obiekt E (pierwsze 4 bajty)
0x14-0x17	Obiekt B	Obiekt F (pierwsze 4 bajty)
0x18-9x1b		Obiekt G (pierwsze 4 bajty)
0x1c-0x1f		Obiekt H (pierwsze 4 bajty)
0x20-0x23		Obiekt I (pierwsze 4 bajty)
0x24-0x27	Obiekt C	Obiekt J (pierwsze 4 bajty)
0x28-9x2b		Obiekt A (pierwsze 4 bajty)
0x2c-0x2f		Obiekt B (pierwsze 4 bajty)
0x30-0x33		Obiekt C (pierwsze 4 bajty)
0x34-0x37	Obiekt D	Obiekt D (pierwsze 4 bajty)
0x38-9x3b		Obiekt E (pierwsze 4 bajty)
0x3c-0x3f		Obiekt F (pierwsze 4 bajty)
0x40-0x43		Obiekt G (pierwsze 4 bajty)
0x44-0x47	Obiekt E	Obiekt H (pierwsze 4 bajty)
0x48-9x4b		Obiekt I (pierwsze 4 bajty)
0x4c-0x4f		Obiekt J (pierwsze 4 bajty)

Aktualnie przetwarzane dane
Część obiektu, której aktualnie nie przetwarzamy

Rys. 3. Porównanie układu pamięci w programowaniu obiektowym oraz z użyciem techniki DOD

Spełnienie obu wyżej wymienionych założeń zapewni aplikacji najbardziej optymalne wykorzystanie pamięci. Rysunek 3 przedstawia porównanie układu pamięci rozwiązania DOD oraz programowania obiektowego.

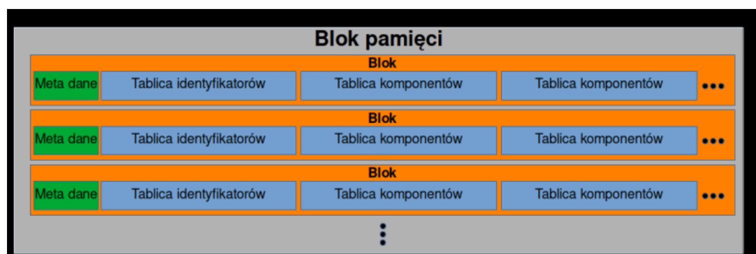
5. Aplikacja testowa

Na potrzeby testów przygotowana została aplikacja testowa oparta o autorskie rozwiązania, w skład których wchodzą silnik graficzny oraz biblioteka DECS realizująca wzorzec ECS. Oprogramowanie zawiera różne testy zaprojektowane w czterech wariantach:

- ECS – podejście oparte na wzorcu ECS. Posiada układ pamięci podobny jak przy programowaniu obiektowym grupując dane obiektów
- Liniowy ECS – rozwiązanie wykorzystujące wzorzec ECS z liniowym układem pamięci zapewniając, że pamięć identycznych komponentów zawsze grupowana jest w pamięci RAM minimalizując czas oczekiwania na dostęp do danych
- Programowanie obiektowe z dziedziczeniem - podejście dla każdego testu zawiera jedną klasę składającą się z danych oraz funkcji aktualizacji
- Programowanie obiektowe z grupowaniem obiektów – rozwiązanie, w którym pamięć obiektów przechowywana jest w ciągłej pamięci (jeden obiekt za drugim) dzięki czemu procesor odwołuje się do elementów pamięci znajdujących się blisko siebie.

Główną metodą omawianą w artykule jest ECS z liniowym układem pamięci. Schemat układu pamięci w tym rozwiązaniu przedstawia rysunek 4.

Wszystkie testy posiadają identyczną strukturę. Pierwszym etapem jest inicjalizacja aplikacji, podczas której alokowane są zasoby oraz tworzone obiekty. Każdy test poza trzecim, który tworzy pół miliona obiektów, wykonywany jest dla miliona obiektów. Kolejnym krokiem jest proces aktualizacji polegającej na wykonaniu pewnej grupy obliczeń na danych wszystkich utworzonych obiektów. Aktualizacja przeprowadzana jest sześćset razy i wyliczany jest średni czas przeprowadzenia jednego takiego procesu. Ostatnim etapem jest finalizacja, podczas której zwalniane są zasoby aplikacji.



Rys. 4. Schemat układu pamięci w rozwiązaniu ECS z liniowym układem pamięci

Pierwsze dwa testy zawierają prostą logikę aktualizacji, której jedynym elementem jest przemieszczanie obiektów. W każdym kroku symulacji obiekty przemieszczane są o stałą wartość. Test drugi zawiera większą ilość redundantnych danych wewnątrz obiektów. Przykłady 1 oraz 2 przedstawiają funkcje aktualizacji kolejno systemu architektury ECS oraz obiektu w podejściu obiektowym.

Przykład 1. Kod aktualizacji systemu w architekturze ECS

```
void update()
{
 position.y += -0.001 * app_state.delta_time;
 if(position.y < -10)
 {
  position.y = 10;
 }
}
```

Przykład 2. Kod aktualizacji obiektu

```
void update()
{
 position.y += -0.001 * app_state.delta_time;
 if(position.y < -10)
 {
  position.y = 10;
 }
}
```

Test trzeci różni się od dwóch poprzednich jedynie bardziej skomplikowanymi obliczeniami wewnątrz funkcji aktualizacji. Przemieszczanie obiektów wykorzystuje funkcje trygonometryczne oraz funkcje wygładzające.

Czwarty test nie zawiera funkcji aktualizacji, ale dużo zewnętrznych odwołań do danych obiektów. Celem testu jest zbadanie narzutu odwołania przy użyciu identyfikatora na wydajność aplikacji.

Ostatnie dwa testy przedstawiają różne rozwiązania identycznego problemu dla architektury ECS. Zdarza się tak, że pewien obiekt w trakcie działania aplikacji zmienia swoje zachowanie. W programowaniu obiektowym uzyskuje się to zazwyczaj instrukcją warunkową wewnątrz metody obiektu, bądź za pomocą listy obiektów podlegających pewnym obliczeniom. Architektura ECS pozwala dodatkowo zmieniać zachowanie obiektu poprzez dodanie bądź usunięcie komponentu do jednostki. Testy badają wydajność kolejno instrukcji warunkowej oraz komponentu etykiety do zmiany zachowania obiektu.

6. Porównanie wydajności

Testy przeprowadzone zostały na komputerze stacjonarnym o specyfikacji przedstawionej w tabeli 1.

Tabela 1. Specyfikacja sprzętu wykorzystywanego w testach

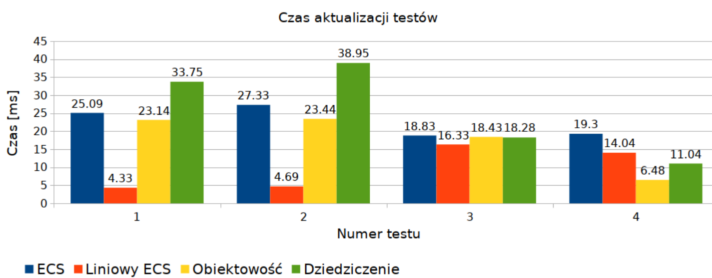
System operacyjny	Linux Arch
Wersja jądra	5.2.arch2
Procesor	AMD FX-8320e
Liczba rdzeni	8
Liczba wątków	8
Pamięć RAM	8GB DDR3 1600MHz

W tabeli 2 przedstawiony są wyniki testu pierwszego zawierające czas inicjalizacji programu, średni czas aktualizacji, czas zakończenia programu i zwalniania zasobów.

Tabela 2. Wyniki testu pierwszego dla wszystkich testowanych rozwiązań

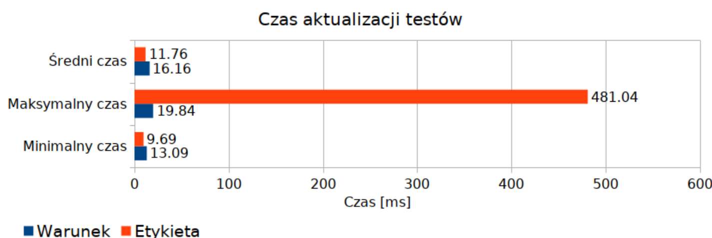
Rozwiązanie	Inicjalizacja [ms]	Aktualizacja [ms]	Zakończenie [ms]
ECS	173.34	25.09	7.68
Liniowy ECS	245.56	4.33	9.76
Programowanie obiektowe z grupowaniem obiektów	112.76	23.14	6.78
Programowanie obiektowe z dziedziczeniem	210.43	33.75	93

Rysunek 5 zawiera diagram czasu aktualizacji dla pierwszych czterech testów.



Rys. 5. Zestawienie czasu aktualizacji pierwszych czterech testów dla wszystkich testowanych rozwiązań

Na rysunku 6 ukazany jest wykres czasu trwania procesu aktualizacji dwóch ostatnich testów z uwzględnieniem najkrótszego oraz najdłuższego czasu aktualizacji.



Rys. 6. Porównanie czasów trwania etapu aktualizacji przy wykorzystaniu komponentu jako etykiety oraz instrukcji warunkowej wewnątrz logiki systemu

7. Podsumowanie

Czas finalizacji oraz inicjalizacji standardowego obiektowego podejścia obiektowego z dziedziczeniem jest wysoki, ponieważ pamięć każdego obiektu alokowana jest osobno. Czas inicjalizacji rozwiązania ECS z liniowym układem pamięci jest wysoki ze względu na konieczność kopiowania danych w różne miejsca w pamięci w celu zgrupowania komponentów.

Podejście ECS z liniowym układem pamięci pozwala na uzyskanie dużo wyższej wydajności. W pierwszych dwóch przypadkach gdy wąskim gardłem aplikacji był czas dostępu do pamięci wydajność tego rozwiązania była nawet 8x większa od standardowego podejścia obiektowego z dziedziczeniem. Wydajność architektury ECS w porównaniu do podejścia obiektowego z grupowaniem obiektów posiadającego identyczny układ pamięci jest podobna, a więc wpływ architektury na wydajność aplikacji jest względnie niewielki. W teście trzecim zauważyć można, że mimo dużo bardziej skomplikowanych obliczeń architektura ECS z liniowym układem pamięci jest nieco bardziej wydajna od pozostałych. Dostęp do pamięci przy użyciu identyfikatora jest nieco wolniejszy niż standardowy dostęp z wykorzystaniem adresu, jednak różnica nie jest tak duża. Najwydajniej prezentuje się rozwiązanie obiektowe z grupowaniem obiektów, jest to jednak rozwiązanie niepraktyczne, mające jedynie pokazać jaką wydajność można uzyskać poprzez idealne ułożenie pamięci zawierające jednocześnie redundantne dane.

Wykorzystanie komponentów jako etykiety warunkującej wykonanie obliczeń pozawala na uzyskanie niewielkiego wzrostu wydajności zakładając, że zmiana zachowania obiektu następuje względnie rzadko.

Przechodząc od wyników do porównania metod pod względem łatwości tworzenia aplikacji warto zauważyć, że architektura ECS stworzona została dla uproszczenia procesu tworzenia aplikacji. Pierwszą jej zaletą jest zmniejszenie redundancji kodu. W architekturze zorientowanej obiektowo klasy często składają się z częściowo identycznych danych oraz funkcji. Można oczywiście wykorzystać dziedziczenie, jednak dalej nie rozwiąże to problemu całkowicie. W architekturze ECS natomiast komponenty oraz systemy są współdzielone pomiędzy jednostkami w zależności od potrzeb. Każdy obiekt niezależnie od jego typu może uzyskać nowy komponent, automatycznie zmieniając jego zachowanie. Ponieważ logika przypisana jest do systemów oraz zawartych w jednostce komponentów, zamiast pisać kod dla każdego obiektu, tworzy się zachowania powiązane z komponentami. Podejście to znacząco zwiększa elastyczność projektowania aplikacji, pozwalając dodatkowo zmieniać zasadę działania programu nawet na zaawansowanym poziomie prac. Problemem projektowania zorientowanego obiektowo jest fakt, że gdy pewna funkcjonalność nie zostanie przemyślana na poziomie projektowania aplikacji, często dodanie jej wymaga przebudowy znacznej części programu. Oczywiście mowa tutaj to o rozwiązaniach opartych na rozbudowanych relacjach pomiędzy obiektami. Paradygmat programowania obiektowego pozwala na budowę dobrej architektury aplikacji, takiej, w której zmiany kodu nie wymuszają zmian całej jego struktury. Jednak tak jak projektowanie zorientowane obiektowo naturalnie prowadzi do redundancji oraz zależności pomiędzy elementami kodu, tak wzorzec ECS minimalizuje ten problem jednocześnie odseparowując funkcjonalności aplikacji. Zaletą architektury ECS jest także podział kodu oraz danych. Systemy nie posiadają informacji o typach obiektów jakie przetwarzają, ani o innych systemach. Dzięki temu podziałowi łatwo jest aplikację dzielić na wiele bibliotek, bądź zaprojektować rozwiązanie Hot-reload pozwalające na zmianę działania programu w czasie jego działania.

Do wad prezentowanej architektury należałoby zaliczyć przede wszystkim trudność zrozumienia metodyki programowania. OOD jest dużo bardziej przystępnym wzorcem projektowania aplikacji dla początkujących programistów. Dodatkowo dostęp do danych komponentów spoza funkcji systemu zawiera narzut obliczeniowy zmuszając programistę do minimalizacji zewnętrznych odwołań do obiektu. Ostatnią istotną wadą architektury ECS jest fakt, że rozwiązanie opiera się na grupowaniu danych w komponentach. Jeżeli aplikacja zawiera niewielką liczbę obiektów, dodatkowo niewspółdzielących funkcjonalności oraz danych, programowanie obiektowe czy inne podejścia mogą okazać się znacznie lepszym rozwiązaniem.

8. Wnioski

Prezentowana architektura jest doskonałym podejściem wykorzystywanym w procesie tworzenia gier. Wykorzystanie tego rozwiązania do innych dziedzin programowania niż tworzenie gier będzie prawdopodobnie złym pomysłem. Podejście pozwala jednak uzyskać znacznie wyższą wydajność oferując jednocześnie bardziej elastyczny proces tworzenia aplikacji oraz ułatwiając wprowadzanie wielu nowych technik. Ponieważ program posiada wszystkie dane systemów oraz komponentów, można w łatwy sposób zbudować algorytm automatycznego generowania zadań równoległych oraz ich zależności. Synchronizacja sieciowa może opierać się na zautomatyzowanej synchronizacji wybranych komponentów. Logika debugowania aplikacji może zawierać się w systemach, które włączane będą w zależności od potrzeb. Rozszerzanie funkcjonalności programu może odbywać się poprzez dodawanie dynamicznych bibliotek zawierających kod nowych systemów. Podział kodu na systemy pozwala w prosty sposób zmienić założenia projektowe na zaawansowanym poziomie prac.

Literatura

[1] Strona główna języka D https://dlang.org/ [10.09.2019]

[2] Opis działania pamięci podręcznej procesora https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips [10.09.2019]

[3] Opis programowania zorientowanego obiektowego https://en.wikipedia.org/wiki/Object-oriented_programming [10.09.2019]

[4] Opis wzorca Entity-component-system https://en.wikipedia.org/wiki/Entity_component_system [10.09.2019]

[5] Fabian R., Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules, DataOrientedDesign.com, 2018

[6] Albrecht T., Pitfalls of Object Oriented Programming, na: Proceedings of Game Connect: Asia Pacific (GCAP) 2009, Melbourne, Australia, 2009

[7] Kowarschik M., Weiß C., An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms, Springer-Verlag, Berlin, Heidelberg, 2003

[8] Nystrom R., Game Programming Patterns, gameprogrammingpatterns.com, 2014