Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka

Praca dyplomowa - inżynierska

NARZĘDZIE DO PROJEKTOWANIA ZACHOWAŃ JEDNOSTEK STEROWANYCH PRZEZ KOMPUTER, DZIAŁAJĄCYCH Z PACZKĄ UNITY ENTITIES

Kamil Socha

słowa kluczowe:
Unity Entities
Gry komputerowe
Jednostki sterowane komputerowo

krótkie streszczenie:

W pracy przedstawiono projekt i implementację narzędzia do projektowania jednostek sterowanych przez komputer. Dodatkowo przedstawiono paradygmat projektowania zorientowanego na dane i wzorzec architektoniczny ECS, które są realizowane przez paczkę Unity Entities, na której oparte są zasady projektowania i generowania kodu przez narzędzie.

opiekun pracy dyplomowej	.....
	Tytuł/stopień naukowy/imię i nazwisko

Ostateczna ocena za pracę dyplomową

Przewodniczący Komisji egzaminu dyplomowego	.....	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

a) kategorii A (akta wieczyste)

b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)

* *niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław, 2020

Streszczenie

Rynek gier komputerowych rozwija się bardzo dynamicznie. Gry komputerowe stają się coraz bardziej zaawansowane i większe zarówno pod względem zawartości, długości rozgrywki, czasu życia produktu oraz używanych systemów i mechanizmów. Ze względu na wymagania stawiane nowoczesnym grom komputerowym, ważne jest jak najskuteczniejsze wykorzystanie dostępnych zasobów platformy uruchomieniowej (komputera, urządzenia, konsoli do gier lub serwera gry), gdzie najskuteczniejsze nie oznacza maksymalnej utylizacji sprzętu, lecz minimalną jego utylizację, przy równoczesnym osiągnięciu wszystkich mechanik i efektów wizualnych zakładanych przez twórców. Praca przybliża paradygmat projektowania zorientowanego na dane, wzorzec ECS, temat programowania wizualnego, generacji kodu i sztucznej inteligencji w grach. Następnie przedstawione są artefakty analizy wymagań projektu, po których objaśniono jak realizowany był projekt i zaprezentowane zostały wybrane artefakty procesu implementacyjnego. Na koniec pracy przedstawiono wnioski wyciągnięte z realizacji projektu.

Abstract

The video games market is developing very dynamically. Video games are becoming more and more advanced and larger in terms of content, length of the game, product life, and the systems, and mechanisms used. Due to the requirements for modern video games, it is required to use the available resources of the running platform (computer, device, game console or game server) as efficiently as possible. Where the most effective does not mean the maximum disposal of equipment, but the minimum disposal of it, while achieving all the mechanics and visual effects assumed by the creators. The work introduces the data-oriented design paradigm, the ECS pattern, the topic of visual programming, code generation and artificial intelligence in games. Then, results of the project requirements analysis are presented, followed by an explanation of how the project was implemented and the presentation of selected artifacts of the implementation process. At the end of the work, conclusions drawn from the project are presented.

Spis treści

1. Wstęp do pracy 1
   1.1. Motywacja 1
   1.2. Geneza pracy 1
   1.3. Cel pracy 1
   1.4. Zakres pracy 1
2. Wstęp teoretyczny 2
   2.1. Projektowanie zorientowane na dane 2
      2.1.1. Wprowadzenie 2
      2.1.2. Pryncypia 3
      2.1.3. Zastosowanie 3
      2.1.4. Wydajność 4
   2.2. Badania wydajnościowe 7
   2.3. Porównanie z popularnymi paradygmatami 10
      2.3.1. Porównanie z programowaniem obiektowym 10
      2.3.2. Realizacja zasad SOLID 11
      2.3.3. Porównanie z programowaniem funkcyjnym 11
      2.3.4. Porównanie z programowaniem proceduralnym 12
   2.4. Wzorzec ECS 12
      2.4.1. Rodzaje implementacji 14
      2.4.2. Użyteczność przy trenowaniu sztucznej inteligencji 15
   2.5. Programowanie wizualne 15
   2.6. Generacja kodu 16
   2.7. Systemy sztucznej inteligencji w grach 17
   2.8. Techniki implementacji sztucznej inteligencji w grach 17
      2.8.1. Maszyna stanów skończonych 17
      2.8.2. Hierarchiczna maszyna stanów 18
      2.8.3. Drzewa zachowań 19
      2.8.4. Planowanie akcji zorientowanych na cel 20
3. Analiza wymagań projektu 21
   3.1. Presony 21
      3.1.1. Krzysztof Abacki: 21
      3.1.2. Alex Baldwin: 21
      3.1.3. Jackob Brugière: 21
   3.2. Historyjki użytkownika 22
   3.3. Wymagania 22
      3.3.1. Milestone 1: 23

3.3.2. Milestone 2: ... 23
3.3.3. Milestone 3: ... 23
4. Projekt ... 24
4.1. Konkurencyjne rozwiązania ... 24
4.1.1. Visual Paradigm ... 24
4.1.2. Unreal Engine Blueprints ... 24
4.1.3. Unity Bolt ... 25
4.1.4. DOTS Visual Scripting ... 25
4.1.5. ECS_FSM ... 26
4.2. Proces realizacji projektu ... 26
4.3. Narzędzia ... 27
4.3.1. Unity ... 27
4.3.2. DOTS ... 27
4.3.3. C# Job System ... 28
4.3.4. Entitas ... 29
4.3.5. C# ... 31
4.3.6. xNode ... 31
4.3.7. ClickUp ... 31
4.3.8. Visual Studio Enterprise 2019 ... 31
4.3.9. Git i GitHub ... 31
4.4. Testy ... 31
5. Prezentacja projektu ... 33
5.1. Instalacja ... 33
5.2. Użycie ... 34
5.3. Przykłady ... 35
5.3.1. Przykład 1 ... 36
5.3.2. Przykład 2 ... 36
5.4. Inne opcje ... 37
6. Implementacja ... 38
6.1. Zmiany w xNode ... 38
6.2. Strukturyzacja plików z kodem ... 38
6.3. Węzły, dane węzłów i wizualizacja ... 39
6.3.1. Dostosowane rysowanie: ... 39
6.3.2. Funkcjonalności: ... 41
6.4. Walidacja ustawień systemów ... 42
6.5. Pasek narzędzi ... 43
6.6. Generacja kodu ... 44

6.6.1. Generacja systemów: 45
6.6.2. Generacja komponentów: 47
6.7. Kreator komponentów 47
6.8. Wczytywanie systemu 49
6.9. Filtr komponentów współdzielonych 49
6.10. Rozmiar komponentów 50
6.11. Pomniejsze artefakty 52
Zakończenie 58
Bibliografia 59
Spis rysunków 63
Spis listingów 65
Spis tabel 66

1. Wstęp do pracy

1.1. Motywacja

Motywacją wybrania takiego tematu jest chęć autora, aby zwrócić uwagę na problemy wydajności w grach komputerowych. Gorsza wydajność to nie tylko większe zużycie energii, ale i sprzętu gracza. Dodatkowo, słaba optymalizacja uniemożliwia zabawę graczom z mniej zasobnymi portfelami. Kolejnym czynnikiem, który miał wpływ na wybór takiego tematu, była technika sztucznej inteligencji, zwana uczeniem przez wzmocnienie (reinforcement learning), która w ostatnich latach zyskuje coraz większą atencję. W technice tej bardzo wartościowym jest prędkość i zrównoleglenie symulacji światów, w których uczone są modele.

1.2. Geneza pracy

Projekt rozpoczął się jako projekt implementacji jednostek półautomatycznych w strategicznej grze czasu rzeczywistego. Bardzo ambitne przedsięwzięcie, pomimo początkowo dobrego startu, zostało zahamowane przez niedojrzałość wykorzystywanych narzędzi. Paczka Entities zawierała dużo błędów i publiczny interfejs programistyczny, który nie był stabilny., Aby osiągać zadany efekt należało stosować 'sztuczki'. Każda kolejna wersja tej paczki, wprowadzała niszczące zmiany (breaking changes) - niekiedy psując API, a niekiedy psując zastosowane 'sztuczki' bez naprawy pierwotnych błędów, co prowadziło do niemożności uzyskania potrzebnych efektów. W związku z powyższym postanowiono zrezygnować z implementacji gry, a skupić się jedynie na głównej części związanej z narzędziem do implementacji jednostek sterowanych przez komputer (NPC - non-player character).

1.3. Cel pracy

Celem pracy jest dostarczenie narzędzia do wizualnego projektowania i programowania (w ograniczonej formie) systemów sztucznej inteligencji, w szczególności jednostek sterowanych przez komputer (NPC). Narzędzie ma ułatwić pracę (i współpracę) projektantom i programistom gier komputerowych, którzy chcą tworzyć wydajne i dzięki temu bardziej ekologiczne gry komputerowe. Wydajność zostanie osiągnięta dzięki wykorzystaniu paradygmatu projektowania zorientowanego na dane i wzorca architektonicznego ECS realizowanego przez paczkę Entities.

1.4. Zakres pracy

Każdy projekt informatyczny wzoruje się pewną nieskończonością, w każdym projekcie można coś poprawić i/lub wprowadzić nową funkcjonalność. Projekt opisywany przez tą pracę nie jest wyjątkiem i można by definiować wiele dodatkowych funkcjonalności. Z tego powodu na potrzeby pracy założono, że projekt będzie realizował jedynie trzy pierwsze kamienie milowe, a nawet cztery, licząc proces analizy projektu (więcej w podrozdziale 4.2 Proces realizacji projektu).

2. Wstęp teoretyczny

2.1. Projektowanie zorientowane na dane

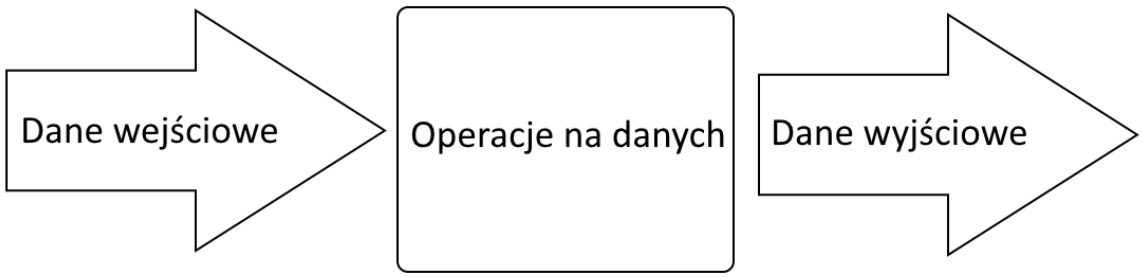
2.1.1. Wprowadzenie

Projektowanie zorientowane na dane (DOD - Data oriented design) jest opisywane na wiele sposobów:

a) Paradygmat.
b) Sposób myślenia i rozwiązywania problemów.
c) Metoda optymalizacji i/lub uproszczania kodu.

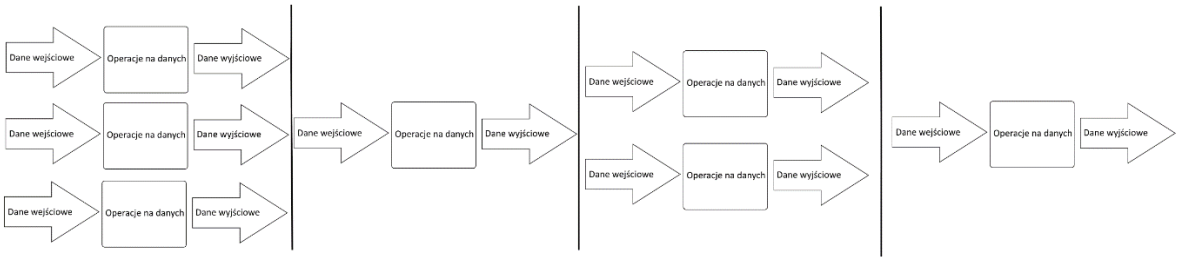
Na potrzeby uproszenia i ujednolicenia terminów w tej pracy przyjęto, iż jest to pełnoprawny paradygmat programowania. Jednak niezależnie od tego, do której klasy rozwiązań zostanie zakwalifikowany, należy poznać podstawowe założenia projektowania zorientowanego na dane. Podstawą tutaj jest założenie, że patrząc na każdy program z odległości można zobaczyć trzy komponenty, tak jak przedstawiono to na Rysunek 2.1:

a) Dane wejściowe
b) Operacja na danych
c) Dane wyjściowe



Rysunek 2.1 Uproszczony schemat programu w założeniach programowania zorientowanego na dane

Przybliżając widok można zobaczyć więcej takich bloków trójek, gdzie dane wyjściowe jednego bloku to dane wejściowe do innego, tak jak przedstawia to Rysunek 2.2.



Rysunek 2.2 Przybliżony, uproszczony schemat programu w założeniach programowania zorientowanego na dane

W bardzo podobny sposób można zobrazować podstawy paradygmatu funkcyjnego. Jednak w programowaniu funkcyjnym największy nacisk jest kładziony na blok 'Operacja na danych', który realizowany jest przez czyste funkcje (często, prymitywne składane ze sobą, aby realizować zaawansowane funkcjonalności). W DOD nacisk jest kładziony na dane wejściowe i dane wyjściowe [1].

Można zauważyć duże podobieństwo tego sposobu wyrażenia programu do systemu bazodanowego i jest to poprawne spostrzeżenie. Projektowanie zorientowane na dane czerpie garściami z osiągnięć informatyki na tym polu nauki [1].

2.1.2. Pryncypia

Pryncypia tego paradygmatu są następujące [1] [2] [3]:

a) Poznaj dane (i domenę) - aby zastosować odpowiednie rozwiązanie lub algorytm, należy wiedzieć z jakimi danymi mamy do czynienia, ile tych danych jest, jak najlepiej ułożyć je w pamięci, czy na pewno wymagana jest posiadana postać danych, jak kształtuje się domena, która zawiera problematykę (dane zawarte w samej domenie). Rozwiązanie pasujące do danych, które często się zmieniają, prawdopodobnie będzie nienajlepszym rozwiązaniem, kiedy dane okażą się dużo bardziej statyczne.

b) Poznaj platformę – błędnym założeniem jest założenie, że pisany kod działa w obrębie jakiejś abstrakcji, a nie na rzeczywistej maszynie. Przyjmując takie błędne pewniki wykorzystujemy notację wielkiego O, nie biorąc pod uwagę możliwości i niedoskonałości platformy uruchomieniowej. Jednak dopiero biorąc pod uwagę rzeczywiste dane i rzeczywistą platformę na jakiej wykonywany będzie kod, można stwierdzić czy O(n) jest lepszym rozwiązaniem czy O(n^2). Często dostęp do pamięci i rozgałęzienia przepływu kodu są większymi „złodziejami" wydajności niż teoretyczna (np. notacja wielkiego O) i praktyczna (np. wyliczenie obydwu wartości rozgałęzienia i matematyczne wyliczenie wartości rozgałęzienia) większa liczba operacji. Platforma uruchomieniowa jest często pomijalnym aspektem tworzenia oprogramowania, lecz jak pokazują badania, nawet różne konfiguracje tej samej platformy mają znaczący wpływ na wydajność [4].

c) Rozwiązuj prawdziwe problemy - ostatnimi laty obserwuje się dużą popularność budowania systemów informatycznych poprzez nabudowywanie warstw abstrakcji, chcąc osiągnąć dany efekt, koder nie szuka rozwiązania bezpośrednio problemu, tylko skupia się na wykorzystaniu różnych abstrakcji, które rozwiązują problemy na przestrzeni własnej głębokości abstrakcji. Jednak jedynie na najniższej warstwie leży esencjonalny problem, reszta problemów na wyższych warstwach to problemy przypadkowe [5]. Przez to powstają takie rozwiązania jak FizzBuzzEnterpriseEdition [6].

2.1.3. Zastosowanie

W świecie programowania i informatyki, rzadko kiedy istnieje możliwość aby udzielić łatwej i szybkiej odpowiedzi na pytanie. Większość pytań kończy się klasycznym sformułowaniem: "To zależy". Taka sama odpowiedź jest odpowiednią na bardzo ogólne pytanie: "Czy w projekcie informatycznym powinno się zastosowywać paradygmat projektowania zorientowanego na dane?".

Patrząc z inżynieryjnej strony można stwierdzić, że ten paradygmat zawsze nadaje się do użycia, daje znakomite możliwości wykorzystania platformy uruchomieniowej i jest bardzo mocnym i potężnym narzędziem. Pozwala z efektywnością bliską maksymalnej wykorzystać zasoby sprzętowe, zapewnia dużą testowalność, łatwość konserwacji (maintainability), zabezpieczenie na przyszłość (futureproofing), pozwala ograniczyć zależności i znacznie eliminuje złożoność przypadkową produkowanego rozwiązania. Lista korzyści jest bardzo duża i dla firmy lub instytucji, gdzie właśnie to są najważniejsze czynniki i główne wartości wytwarzanego oprogramowania, warto wybrać ten paradygmat jako główny czynnik kształtujący architekturę i wybierane rozwiązania problemów.

Jednak w obecnym świecie cenione są tanie i szybkie rozwiązania, które dopiero w razie potrzeby analizuje się, refaktoruje i optymalizuje. Oczywiście takie podejście ma swoje mocne i słabe strony, to co jest cenione w takim podejściu to zwinność. Szybko można wytworzyć prototyp, który spełnia podstawowe założenia, i dzięki któremu można szybko testować funkcjonalności. Nie trzeba tracić czasu na analizę danych, na których będzie rozwiązanie operowało, jak i platformy, na której będzie uruchamiane [1] [2]. Kolejną wadą jest to, że podejście DOD wymaga pracowników z wyższymi

kwalifikacjami. Podejście programowania obiektowego jest zdecydowanie bardziej odporne na brak doświadczenia i wiedzy pracowników. Bez przeszkód może funkcjonować zespół złożony z jednego doświadczonego programisty i kilku młodych. Realizacja podobnego zespołu wytwarzającego kod w podejściu zorientowanym na dane będzie co najmniej mniej podstawna, ze względu na obniżoną wydajność produkcyjną. Również na rynku pracy zdecydowanie łatwiej znaleźć pracowników operujących w paradygmatach takich jak OOP i/lub FP. Mała popularność tego paradygmatu oznacza trudność w rozbudowie zespołów.

Biorąc pod uwagę powyższe spostrzeżenia, można wyznaczyć kilka pytań. W momencie, kiedy większość odpowiedzi jest twierdząca, warto przyjrzeć się projektowaniu zorientowanemu na dane jako przyszłości wydajnego i łatwego w utrzymaniu oprogramowania, które firma ma zamiar tworzyć i rozwijać:

- Czy wydajność tworzonego rozwiązania jest głównym lub jednym z głównych czynników świadczących o jego jakości?
- Czy projekt jest dobrze określony już na początku, możemy zastosować metodologię zarządzania projektem zbliżoną bardziej do metodologii waterfall niż agile? Lub, czy przy zmianach wymagań można pozwolić zespołowi na ponowną ewaluację dostarczonego rozwiązania i w razie potrzeby zastąpienia go zupełnie nowym?
- Czy zespół projektowy składa się w większości z doświadczonych pracowników i/lub w razie potrzeby mamy dostęp do tego typu pracowników?

2.1.4. Wydajność

Programowanie/projektowanie zorientowanie na dane opiera się na kilku fundamentalnych założeniach i spostrzeżeniach. Jednym z pierwszych obserwacji jest to, że dane, na których programy wykonują transformacje, bardzo rzadko występują pojedynczo [1], zazwyczaj występuje więcej niż jedna animacja szkieletowa, więcej niż jedna siatka obiektu i więcej niż jeden przycisk.

Następnym spostrzeżeniem jest to, że struktury lub obiekty, na których zazwyczaj wykonywane są operacje i transformacje, zawierają wiele niepotrzebnego szumu. Szumu, czyli danych, które nie są potrzebne przy wybranej operacji, nie mniej obecnych przy pobieraniu właściwych atrybutów. Podczas sprawdzania czy wskaźnik myszki znajduje się w obrębie prostokąta przycisku, nie potrzeba wiedzieć jakim kolorem przycisk ten będzie rysowany, czy też jaki audio klip ma zostać zagrany w momencie kliknięcia. Tak samo, podczas obliczania pozycji kości w animacji, nie występuje potrzeba, aby wiedzieć, ile zdrowia i amunicji posiada postać, na której awatarze wykonywania jest animacja.

Powyższe spostrzeżenia prowadzą do zjawiska znanego jako 'luka pomiędzy prędkością działania procesora, a prędkością działania pamięci RAM', szerzej opisana w pracy o tym samym tytule [7] i pokrewnych [3] [8]. Zjawisko to wskazuje na rosnącą lukę pomiędzy tym, jak szybko procesor potrafi wykonywać obliczenia, a tym jak szybko dane mogą być do niego dostarczone z pamięci RAM. Prędkości te różnią się paroma rzędami wielkości [9] [10]. Rozwiązaniem tego problemu jest użycie pamięci podręcznych l1 i l2, do których procesor ma dostęp niemal bezpośredni i rzędy wielkości szybszy niż do pamięci RAM. Aby maksymalnie wykorzystać ten mechanizm, dane, na których wykonywane będą transformacje, muszą być odpowiednio ustawione w pamięci i jak najbardziej treściwe, czyli pozbawione szumów. Tematyka mechanizmów pamięci podręcznych l1 i l2 nie jest celem tej pracy, więcej informacji na ten temat można znaleźć w publikacji [9]. Jednak należy wspomnieć, że pobierając z obiektu (lub struktury) jedno lub dwa pola, zostanie załadowana do pamięci podręcznej duża ilość danych, które nie są potrzebne w tym momencie, wyrzucając dane, które mogą być niezbędne za chwilę, w następnych liniach programu. Jako że jest to stosunkowo dobrze znany problem

(zwłaszcza przy programowaniu gier komputerowych i/lub wysokowydajnych systemów), to została wymyślona technika programowania wspierająca maksymalne wykorzystanie linii pamięci podręcznej.

Mowa tutaj o technice zwanej „Structure of Arrays – SoA” (struktura tablic) [3] [8] [11]. Technika ta polega na deklarowaniu struktury danych z tablicowymi atrybutami i zainicjalizowanie jeden raz. Jest to przeciwieństwo powszechnego podejścia zwanego „Array of Structures – AoS” (tablica struktur) gdzie struktury są definiowane z pojedynczymi polami i następnie instancjonowane x razy (tablica struktur). Różnicę w realizacji tych technik obrazuje Listing 2.1.

```
1  public class Ball_AoS{
2      public string Name;
3      public float2 Position;
4      public float2 Velocity;
5      public float4 Color;
       ...
6      public bool IsGrounded;

7      public void UpdatePosition(float deltaTime){
8          Position += Velocity * deltaTime;
9      }
10 }
11 public class BallsManager_AoS{
12     const int BALLS_COUNT = 10000;
13     public Ball_AoS[] Balls = new Ball_AoS[BALLS_COUNT];
14     public void UpdatePosition(float deltaTime){
15         foreach(var ball in Balls){
16             ball.UpdatePosition(deltaTime);
17         }
18     }
19 }

20 public class Ball_SoA{
21     public string[] Names;
22     public float2[] Positions;
23     public float2[] Velocities;
24     public float4[] Colors;
       ...
25     public bool[] IsGrounded;

26     public Ball_SoA(int ballsCount){
27         Names = new string[ballsCount];
28         Positions = new float2[ballsCount];
29         Velocities = new float2[ballsCount];
30         Colors = new float4[ballsCount];
31         IsGrounded = new bool[ballsCount];
32     }
33 }

34 public class BallsManager_SoA{
35     const int BALLS_COUNT = 10000;
36     public Ball_SoA Balls = new Ball_SoA(BALLS_COUNT);
```

```
37      public void UpdatePosition(float deltaTime){
38          for(int i = 0; i < BALLS_COUNT; i++){
39              Balls.Positions[i] += Balls.Velocities[i] * deltaTime;
40          }
41      }
42  }
```

Listing 2.1 Przykładowa realizacja klasy piłka i logiki przesuwania jej w technice AoS i SoA

Przy omawianiu wydajności w kontekście programowania zorientowanego na dane, nie można także pominąć założenia, że wykonawca systemu zna dane, na których chce operować. Dodatkowo, tak samo jak znajomością danych, powinien wykazywać się on odpowiednią znajomością domeny, którą wspomniane dane reprezentują. Przy takim założeniu można zasięgnąć do technik i wiedzy z zakresu technik bazodanowych, dzięki czemu możliwe jest przetransformowanie danych tak, aby były uszeregowane w tabelarycznej formie, znane z relacyjnych baz danych. Te wszystkie operacje pozwalają zastosować na nich techniki takie jak normalizacja tabel [1]. Wraz z takim spojrzeniu na dane, można dokonać wielu optymalizacji pamięci i operacji, a pisany kod aplikacji staje się łatwiejszy w utrzymaniu. To są oczywiste korzyści płynące z wykorzystania dorobku na polu baz danych. Jednak na początku została również wspomniana o wiedza domenowa, która jeszcze szerzej rozwija wachlarz osiągalnych korzyści. Korzyści niekoniecznie zauważalnych przy pobieżnym zgłębieniu tematu, mianowicie pozwala na pozbycie się wielu rozgałęzień z naszego kodu. Przed zagłębieniem się w sposób realizacji programowania bez rozgałęzień, należy wspomnieć czemu technika ta jest warta uwagi i dlaczego powinno się dążyć do likwidacji rozgałęzień w pisanym kodzie. W nowoczesnych procesorach stosuje się technikę Instruction pipelining (strumieniowanie instrukcji) [12] [10], która polega na załadowaniu przyszłych instrukcji i wcześniejszym przygotowywaniu ich do wykonania. Problem pojawia się z określeniem, które instrukcje mają być następnie wykonane, a mianowicie kiedy w programie następują instrukcje skoku warunkowego. Producenci wyposażają swoje produkty w systemy przewidywania rozgałęzień. Jednak mimo wyszukanych i zaawansowanych algorytmów i układów, których zadaniem jest przewidzieć ścieżkę programu, nie jest to idealne rozwiązanie. Dzięki wykorzystaniu wiedzy domenowej, możliwe są do osiągnięcia znacznie lepsze wyniki niż przy użyciu tych narzędzi, w których wyposażane są nowoczesne jednostki centralne. Najłatwiej taką technikę wykorzystania wiedzy domenowej przedstawić na przykładzie [1]:

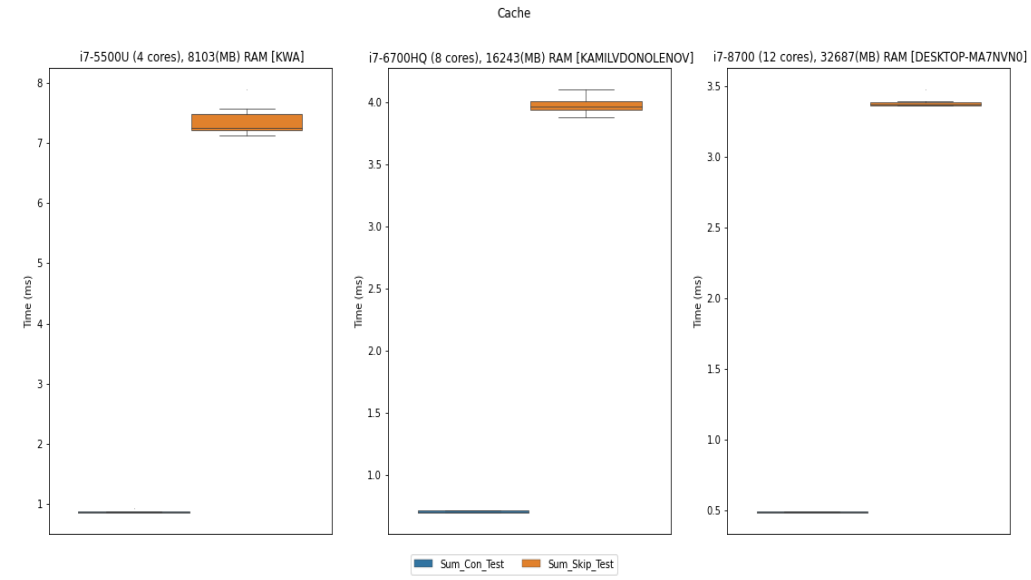
W grze istnieje system autoregeneracji zdrowia, który aktywuje się po odczekaniu odpowiedniej ilości czasu bez otrzymania obrażeń i jak wskazuje nazwa, jego zadaniem jest przywracanie życia jednostki w czasie. W naiwnej wersji istnieje funkcja, która przyjmuje wszystkich posiadaczy zdrowia, sprawdza, czy zdrowie jest mniejsze niż maksymalna ilość zdrowia, sprawdza czy dołączony jest element z danymi o ostatnim czasie obrażeń, sprawdza czy czas, który upłynął od ostatniego obrażenia jest mniejszy niż czas potrzebny do regeneracji. Oczywiście istnieje możliwość nieco wydajniejszej implementacji, poprzez wprowadzenie atrybutu flagi np. „bool hasActiveHPRegeneration”. Odpowiednie aktualizowanie tej zmiennej pozwoli zredukować ilość niepotrzebnych rozgałęzień. Natomiast ten sam przykład można sprowadzić do tabelarycznych danych. Wyróżnić należałoby w tym przypadku następujące tabele: Health i LastDamage. Do wspomnianej funkcji przesyłane będzie złączenie tych dwóch tabel, przy czym wpis w tabeli Health występuje jedynie, gdy zdrowie jest mniejsze niż maksymalne zdrowie i jest większe niż 0, czyli jednostka jest żywa, ale nie w pełni zdrowa. Natomiast tabela LastDamage zawiera wartości posortowane według czasu

zadania obrażeń, dzięki czemu podczas iterowania można precyzyjnie wskazać, w którym momencie należy przerwać iteracje po danych wejściowych i zaprzestać regeneracji zdrowia dla tych jednostek, ponieważ nie spełniają one założonych warunków dla tego procesu. W tym momencie ilość rozgałęzień warunkowych została znacznie ograniczona, funkcja regeneracji zdrowia odpowiadająca za samą regenerację, została pozbawiona szumu zarówno funkcjonalnego (instrukcje i dane nie realizujące bezpośrednio logiki biznesowej), jak i szumu związanego z danymi. Oczywiście można posunąć się dalej i wprowadzić dalsze optymalizacje wydajnościowe, wprowadzając taką tabelę, że jej złączenie z tabelą Health będzie zawierało jedynie te byty, które spełniają założenia funkcji regeneracji zdrowia. Te działanie jednak musi być podyktowane odpowiednimi wskaźnikami, ponieważ istnieje możliwość, że takie rozwiązanie nie tylko pogorszy wydajność pamięciową, ale również tą związaną z czasem odpowiedzi systemu.

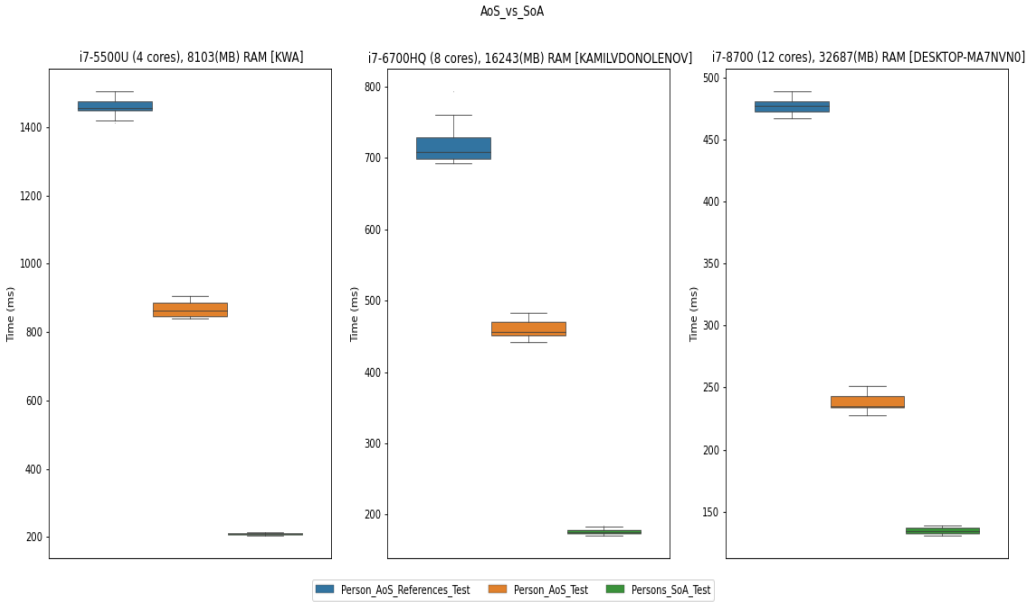
W tym momencie wyłania się obraz pewnego wzorca, wzorzec ten opisuje program, który składa się z funkcji. Funkcje te operują na tabelarycznych i znormalizowanych danych. Wzorzec ten został nazwany ECS, czyli Entity–component–system.

2.2. Badania wydajnościowe

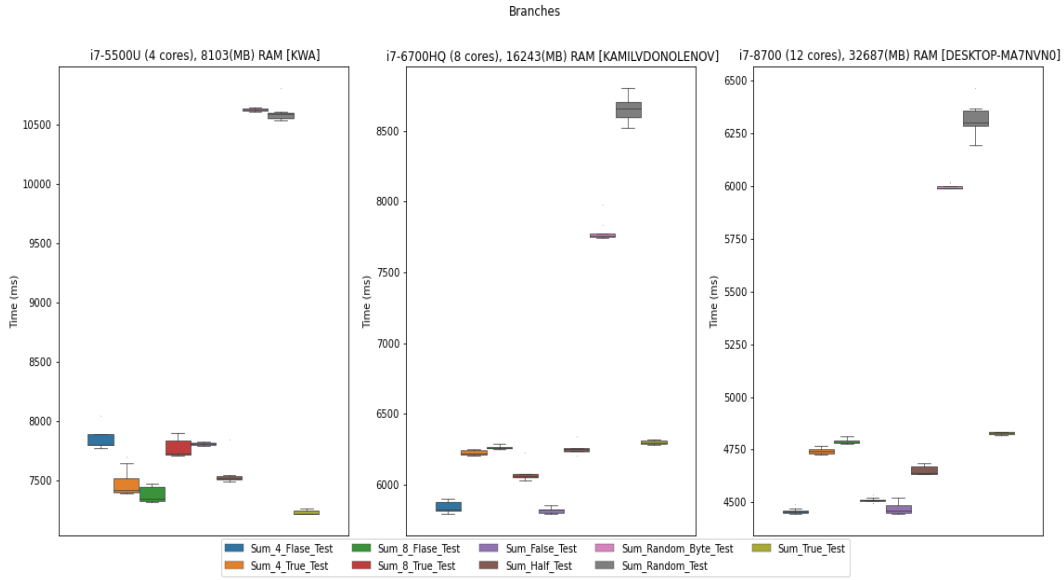
Autor przeprowadził trzy trywialne testy wydajnościowe w silniku Unity z wykorzystaniem paczki Performance Testing Extension. Pomimo, a może zwłaszcza dlatego, iż są proste, to są one dobrym punktem startowym do bardziej wnikliwych analiz i testów, równocześnie wskazując prawdziwość założeń przyjmowanych w projektowaniu zorientowanym na dane. Implementacja testów jest ogólnodostępna na platformie GitHub w repozytorium autora [13]. Rysunek 2.3, Rysunek 2.4 i Rysunek 2.5 Wyniki testów wydajności sterowania przepływem przez dyrektywę if i wersji bez niej. Poszczególne podpisy oznaczają: Sum_True_Test – w każdym rozgałęzieniu podjęta jest gałęź ‘prawda’, Sum_False_Test – w każdym rozgałęzieniu podjęta jest gałąź ‘fałsz’, Sum_Half_Test – połowa gałęzi jest ewaluowana przez wartość ‘prawda’, Sum_4_True_Test - co czwarta gałęź jest ewaluowana przez wartość ‘prawda’, Sum_4_False_Test - co czwarta gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_8_True_Test - co ósma gałęź jest ewaluowana przez wartość ‘prawda’, Sum_8_False_Test - co ósma gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_Random_Test – każda gałęź zależy od pseudolosowej wartości o równej dystrybucji i Sum_Random_Byte_Test – obydwa rozgałęzienia są ewaluowane a ostateczna wartość wyznaczana jest przy pomocy arytmetyki. przedstawiają wyniki testów z różnych platform testowych, platformy pomimo różnic w architekturze i budowie wykazują podobne relacje w wynikach.



Rysunek 2.3 Wyniki testów wydajnościowych wykonywania obliczeń z wykorzystania mechanizmu pamięci podręcznej i bez tego mechanizmu.



Rysunek 2.4 Wyniki testów badających wydajność podejść „tablica struktur z referencjami”, „tablica struktur” i „struktura tablic”.



Rysunek 2.5 Wyniki testów wydajności sterowania przepływem przez dyrektywę if i wersji bez niej. Poszczególne podpisy oznaczają: Sum_True_Test – w każdym rozgałęzieniu podjęta jest gałęź ‘prawda’, Sum_False_Test – w każdym rozgałęzieniu podjęta jest gałąź ‘fałsz’, Sum_Half_Test – połowa gałęzi jest ewaluowana przez wartość ‘prawda’, Sum_4_True_Test - co czwarta gałęź jest ewaluowana przez wartość ‘prawda’, Sum_4_False_Test - co czwarta gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_8_True_Test - co ósma gałęź jest ewaluowana przez wartość ‘prawda’, Sum_8_False_Test - co ósma gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_Random_Test – każda gałęź zależy od pseudolosowej wartości o równej dystrybucji i Sum_Random_Byte_Test – obydwa rozgałęzienia są ewaluowane a ostateczna wartość wyznaczana jest przy pomocy arytmetyki.

Rysunek 2.3 przedstawia wyniki testów wykorzystania pamięci podręcznej procesora. Test polegał na wykonaniu sumowania miliona liczb typu long (64-bitowa wartość całkowita), sumowanie te odbywało się przy wykorzystaniu pętli o określonym rozmiarze. W wersji wykorzystującej mechanizm pamięci podręcznej, pętla sumuje każdą kolejną liczbę. Drugi test również wykonuje milion obrotów pętli, natomiast aby zniwelować użycie pamięci podręcznej, z tablicy wybierana jest co dziesiątą wartość. Jak wskazują na to wyniki wykorzystanie mechanizmu pamięci podręcznej, pozwala na uzyskanie nawet ośmiokrotnie lepszej wydajności.

Rysunek 2.4 przedstawia wyniki testów podejść „tablica struktur z referencjami”, „tablica struktur” i „struktura tablic”. Test polegał na wielokrotnym zsumowaniu rocznych zarobków wyciąganych z różnych struktur danych. W tym teście najlepiej prezentuje się rozwiązanie związane z podejściem „struktura tablic”, prezentując znacznie lepszą wydajność niż konkurencyjne podejścia.

Rysunek 2.5 Wyniki testów wydajności sterowania przepływem przez dyrektywę if i wersji bez niej. Poszczególne podpisy oznaczają: Sum_True_Test – w każdym rozgałęzieniu podjęta jest gałęź ‘prawda’, Sum_False_Test – w każdym rozgałęzieniu podjęta jest gałąź ‘fałsz’, Sum_Half_Test – połowa gałęzi jest ewaluowana przez wartość ‘prawda’, Sum_4_True_Test - co czwarta gałęź jest ewaluowana przez wartość ‘prawda’, Sum_4_False_Test - co czwarta gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_8_True_Test - co ósma gałęź jest ewaluowana przez wartość ‘prawda’, Sum_8_False_Test - co ósma gałęź jest ewaluowana przez wartość ‘fałsz’, Sum_Random_Test – każda gałęź zależy od pseudolosowej wartości o równej dystrybucji i Sum_Random_Byte_Test – obydwa rozgałęzienia są ewaluowane a ostateczna wartość wyznaczana jest przy pomocy arytmetyki. przedstawia wyniki testów wydajności

wykorzystania systemów przewidywania wyników rozgałęzień zawartych w kodzie. Jak widać przy wyraźnych wzorcach rozgałęzień mechanizmy te działają bardzo dobrze, natomiast przy pseudolosowym wyborze rozgałęzień widać zdecydowany spadek wydajności. Dodatkowo zaprezentowany jest wynik rozwiązania bez rozgałęzień, gdzie wyliczono obydwie wartości i wyznaczono ostateczną wartość przy wykorzystaniu rozwiązań arytmetycznych. Podejście te daje różne wyniki, przez ograniczoną liczbę obiektów badawczych nie można jednoznacznie stwierdzić, czy jest to opłacalne (lub pod jakimi warunkami) rozwiązanie.

2.3. Porównanie z popularnymi paradygmatami

2.3.1. Porównanie z programowaniem obiektowym

Programowanie zorientowane obiektowo (object-oriented programming - OOP) jest najpopularniejszym i najbardziej rozpowszechnionym paradygmatem [14]. Wśród 20 najpopularniejszych języków, aż 17 języków wspiera ten paradygmat, a dla 11 jest to główny paradygmat. Jednym z pierwszych języków które korzystały z tego paradygmatu to Smalltalk i Simula. Dały one korzenie dla jednych z najbardziej popularnych języków: C++, Java, C#, Python itd.

Główne cechy OOP [15] [16] [17] [18] [19]:

a) Obiekty - instancja klasy lub prototypu.

b) Klasa lub prototyp – definicja powiązania danych i procedur.

c) Enkapsulacja - możliwość określenia widoczności dla poszczególnych składowych typu (włącznie z metodami), w szczególności do ukrycia ich przed innymi bytami.

d) Dziedziczenie - klasa dziecka dziedziczy właściwości klasy rodzica i zgodnie z algebrą typów wszędzie, gdzie mogliśmy użyć klasy rodzica, możemy przekazać klasę dziecka.

e) Polimorfizm - polimorfizm często wymieniany jest jako cecha języków wspierających programowanie zorientowane obiektowo, jednak wymagany jest jedynie mechanizm dziedziczenia, sam polimorfizm może występować poza obiektowymi normami i w różnych formach występuje w różnych językach niewspierających programowania obiektowego.

Z tym paradygmatem związane jest wiele technik inżynierii i projektowania oprogramowania i tzw. zasad dobrych praktyk w programowaniu obiektowym. Złośliwi mogą stwierdzić, że dla innych paradygmatów nie powstało tak wiele zasad, ponieważ są u podstaw skuteczniejsze i bezpieczniejsze niż OOP [20]. W tym stwierdzeniu będzie trochę racji. Przykładowo, powszechnie przyjmowane jest, że w większości przypadków nie powinniśmy stosować dziedziczenia, tylko użyć kompozycji [21].

Z technicznego punktu widzenia zasady projektowania zorientowanego na dane możemy realizować w większości, jeżeli nie we wszystkich językach zorientowanych obiektowo. Jednak to, że syntaktyka języka umożliwia taki styl to nie znaczy, że semantycznie jest to zgodne z programowaniem zorientowanym obiektowo [15].

Tak więc:

a) Jednym z fundamentów DOD jest odzieranie rozwiązań z abstrakcji, gdzie OOP charakteryzuje się ubieraniem rozwiązań w coraz to grubsze warstwy abstrakcji.

b) W DOD zainteresowanie kieruje się na dane i rozwiązywanie problemu w całości, w OOP natomiast uwaga skupiona jest na takim zaprojektowaniu wycinka świata, żeby rozwiązać problem na poziomie jednostkowym (na poziomie instancji).

c) W DOD maksymalizowane jest wykorzystanie dostępnego sprzętu na jakim ma być wykonywany kod, natomiast w OOP skupia się na stworzeniu uniwersalnego i abstrakcyjnego wycinka świata, na którym można bezpiecznie operować z zewnątrz.

d) Tak jak zostało to opisane w poprzednim rozdziale, DOD jest związane z podejściem struktury tablic (SoA), natomiast w OOP głównym podejściem jest tablica struktur (AoS).

e) W programowaniu obiektowym poprzez znaczne zależności pomiędzy warstwami abstrakcji i obiektami w tej samej warstwie abstrakcji, zrównoleglenie obliczeń i wprowadzenie wielowątkowości często może być nietrywialnym wyzwaniem. W projektowaniu zorientowanym na dane wprowadzenie tych technik optymalizacji jest znacznie prostsze.

2.3.2. Realizacja zasad SOLID

Jednym z fundamentów współczesnego podejścia do OOP są zasady SOLID [15] [22]. Wśród tych zasad można wyróżnić dwa typy: ogólne zasady programowania i obiektowe zasady programowania. Do tej pierwszej grupy należą dwie pierwsze zasady, reszta to przedstawiciele drugiej grupy. Projektowanie zorientowane na dane i wzorzec ECS (opisany w późniejszych rozdziałach) znakomicie realizują te dwie praktyki. Zasada pojedynczej odpowiedzialności jednak jest ciężka do realizacji, przy podejściu programowania obiektowego. Podczas implementacji systemu zaklęć, mając klasę ‘Czarodziej’ wydaje się ona dobrym kandydatem do trzymania poznanych zaklęć i uczenia się nowych. Natomiast w podejściu zorientowanym na dane, granice jednostki implementacyjnej wyznacza rozwiązywany problem, co oznacza, że jedynym powodem do wprowadzenia zmiany w jednostce implementacyjnej jest zmiana sposobu realizacji rozwiązania problemu. Podobnie przy realizacji zasady otwarte-zamknięte, z wykorzystaniem wzorca ECS otrzymujemy niezwykłą otwartość na rozszerzanie. Nowy kod może dołączać własne systemy do świata i przyłączać nowe komponenty do bytów wykorzystywanych przez inne systemy.

2.3.3. Porównanie z programowaniem funkcyjnym

Programowanie funkcyjne (functional programming – FP) jest kolejnym bardzo popularnym paradygmatem, który zyskuje coraz większą popularność. Stare języki pokroju C#, czy nowe taki jak Rust posiadają wsparcie dla tego podejścia. 13 z 20 najpopularniejszych języków wspiera ten paradygmat, a dla 5 z nich jest to główny wspierany sposób programowania. Ciekawe natomiast jest to, że wśród tej piątki znajdują się dwa z głównych języków dla matematyków i naukowców (R i MATLAB), nowy język systemowy, który niesamowicie szybko zyskuje popularność (RUST) i najważniejszy język do tworzenia dynamicznych stron internetowych (JavaScript).

Główne cechy programowania funkcyjnego [15] [23] [19]:

a) Czyste funkcje (matematyczne funkcje) - funkcje w językach funkcyjnych nie posiadają efektów ubocznych.

b) Rekurencja - wykorzystywana jest rekurencja zamiast iteracji.

c) Niezmienne dane - raz przypisanej wartości do zmiennej nie można zmienić na inną wartość, pod daną referencją wartość może zostać wpisana tylko raz.

d) Funkcje wyższego rzędu i funkcja jako typ pierwszoklasowy - oznacza to, że funkcja może jako argument przyjmować inną funkcję, a same funkcje mogą być przechowywane w zmiennych, przesyłane jako parametr i tworzone podczas wykonywania programu (dynamicznie).

Paradygmat ten zyskuje popularność zapewne ze względu na bezpieczeństwo, dzięki temu, że dane są niezmienne, eliminuje on problem złego użycia danych i funkcjonalności, który skutkowałby złym/nieokreślonym stanem całej aplikacji lub jej części. Dodatkowym atutem jest łatwość wprowadzenia wielowątkowości i zrównoleglenia obliczeń.

Po tym wprowadzeniu można przejść do porównania:

a) Na wysokim poziomie abstrakcji, obydwa podejścia można uprościć do modelu przedstawionego na Rysunek 2.1 w rozdziale 2.1 Projektowanie zorientowane na dane. Jednak w DOD funkcje nie muszą być czystymi funkcjami, dane wyjściowe mogą być zmutowanymi danymi wejściowymi.

b) W programowaniu funkcyjnym nacisk jest kładziony na funkcje i deklaratywny opis rozwiązania problemu, natomiast przy programowaniu zorientowanym na dane głównym zainteresowaniem są właśnie danych i imperatywnym rozwiązaniu problemu.

W obydwu podejściach, w stosunkowo prosty sposób można wprowadzić zrównoleglenie i wielowątkowość obliczeń.

2.3.4. Porównanie z programowaniem proceduralnym

Programowanie proceduralne (procedural programming - PP) to jeden z pierwszych paradygmatów, 11 z 20 pozwala programować całkowicie proceduralnie, a na pierwszym miejscu jest język, który jest w całości zgodny z tym paradygmatem (nie jest ani obiektowy, ani funkcyjny). Ten styl programowania jest również mocno związany z automatyzacją i skryptowaniem. W programowaniu proceduralnym istnieją procedury (metody/funkcje), zmienne i moduły. Jest jednym z najwcześniejszych paradygmatów. Można przyjąć, że jest to podstawa, z której wywodzą się inne omawiane tutaj paradygmaty. Paradygmat ten wywodzi się z paradygmatu programowania strukturalnego, który opiera się na zunifikowanym i bezpiecznym sterowaniem przepływem w programie [15]. OOP, FP i DOD są bardziej ewolucją PP niż odmiennym bytem, powstały poprzez założenie dodatkowych ograniczeń i/lub dodanie nowych konceptów. Tak jak programowanie proceduralne wywodzi się z programowania strukturalnego. DOD spełnia więc wszystkie założenia PP, ale wprowadza dodatkowe założenia, koncepty i ograniczenia.

2.4. Wzorzec ECS

Wzorzec ten został wspomniany w rozdziale 2.1 Projektowanie zorientowane na dane. Tak jak zostało to opisane, jest to wzorzec architektoniczny, jego nazwa - Entity–component–system (ECS) - oznacza:

• Entity (Byt) - jest identyfikatorem (zaawansowana wersja indeksu), używanym do identyfikacji i agregacji komponentów.

• Component (Komponent) - jest jednostką danych, powinna być jak najmniejsza jak to możliwe (zamiast struktury z Listing 2.2 preferowane są struktury z Listing 2.3). Komponent nie może posiadać żadnej logiki biznesowej. Mimo, że poprzednie zdanie jest krótkie i proste, to jest ono esencjonalne w definicji komponentu.

• System (System) - jednostka logiczna wykonująca operacje na danych (komponentach), transformująca dane z jednego stanu do kolejnego, jak ukazuje to Listing 2.4.

```
1  public struct Stats
2  {
3      int Health;
4      int Speed;
5      int Mana;
6      int Intelligence;
7  }
```

Listing 2.2 Struktura Stats nie spełniająca założeń komponentu ze względu na przechowywanie wielu niezależnych wartości

```
public struct HealthComponent { public int Value; }
public struct SpeedComponent { public int Value; }
public struct ManaComponent { public int Value; }
public struct IntelligenceComponent { public int Value; }
```

Listing 2.3 Rozbicie struktury Stats na poprawne niezależne komponenty

```
public system DeathSystem(
    Entity entity,
    [Requires] HealthComponent health,
    [None] DeathTag _ )
{
     if( health.Value <= 0 )
     {
         EntityManager.RemoveComponent<HealthComponent>(entity);
         EntityManager.AddComponent<DeathTag>( entity);
     }
}
```

Listing 2.4 Przykładowy system realizujący logikę oznaczania bytu za martwy

System nie może posiadać głębokiego i/lub skomplikowanego stanu (stan systemu może występować jako dane zapisane w ramach pamięci podręcznej, czyli zasoby, których ciągłe pobieranie wiązałoby się z niepotrzebnym kosztem).

Wzorzec ten jest najczęściej wykorzystywany w grach komputerowych [10], ponieważ bardzo dobrze sprawdza się w realizacji oczekiwań jakie stawia się przed nowoczesnymi grami komputerowymi - czyli świetnej optymalizacji (wyliczenie klatki gry w rozdzielczości 4k i w czasie poniżej 16ms) i wielu obiektów, które wchodzą w interakcje pomiędzy sobą jak i z graczem. Jednak wzorzec ten nie opiera się jedynie na wydajności, tylko na prostym dostępie do danych, klarownym i jasnym definiowaniu funkcjonalności, które nie posiadają zależności funkcyjnych (każdy z systemów nie posiada zależności innych niż te do używanych danych), przez co włączanie i wyłączanie poszczególnych systemów jest niezwykle proste. Znacznie wpływa to na czynniki takie jak:

- Testowalność - nie potrzebujemy programować imitacji (mock'ów) innych systemów, żeby przetestować dany system, dzięki temu testy są znacznie prostsze (wystarczy wprowadzić/wygenerować dane wejściowe i przekazać je do systemu) i odporne na zewnętrzny szum takich jak np. zła imitacja zależności.
- Łatwość utrzymania - dzięki temu, że systemy są małe, posiadają mało odpowiedzialności i zmieniają jasno określony zestaw danych, można w prosty sposób określić miejsce błędu. Dodanie nowej funkcjonalności w większości przypadków polega na wprowadzeniu nowego systemu i ewentualnie nowych danych. Przy zmianie danych jest wiadome, które systemy polegają na nich, przez co również wiadome jest, gdzie wprowadzone zmiany mogą skutkować w błędnym działaniu.
- Zabezpieczanie na przyszłe zmiany (future proofing) - tak jak zostało wspomniane w powyższym punkcie, podczas wprowadzania zmian, jedyna trudność polega na zaimplementowaniu, a nie manewrowaniu między zależnościami i poziomami abstrakcji.
- Zrównoleglenie i wielowątkowość - każdy system operuje na znanym zestawie danych, przez co istnieje możliwość aby równocześnie wykonywać kilka systemów, gdzie systemy te operują na rozłącznych zestawach danych. Dodatkowo często prawdą jest, że jedno wykonanie pętli korzysta z danych powiązanych z jednym entity, przez co można zrównoleglić i wykonać w wielowątkowym podejściu jeden system.

2.4.1. Rodzaje implementacji

Wzorzec ECS może być realizowany przez kilka implementacji (z pominięciem podejść nieskalowalnych i nieuniwersalnych, które są szyte na miarę dla jednego sprecyzowanego projektu i jego wymagań):

• Class-based (oparty na klasach) - jest implementacją związaną z programowaniem obiektowym, a nie zorientowanym na dane jak inne. W takiej implementacji byt jest kontenerem, w którym grupowane są powiązane ze sobą komponenty. Przykładowy kod implementacji klasy Entity znajduję się na Listing 2.5.

```
1  class Entity
2  {
3      public HashSet<IComponent> Components;
4      public HashSet<Type> Types;
5  }
```

Listing 2.5 Prosta realizacja Entity w oparciu o klasę

Komponent jest klasą implementującą zadany interfejs lub nadklasę, która posiada tylko dane, tak jak we wzorcu obiektu transferu danych (data transfer object – DTO). Natomiast system jest klasą, która implementuje zadany interfejs lub nadklasę i posiada przynajmniej jedną metodę, która realizuje logikę biznesową.

• Bitset-based (oparty na zbiorach bitowych) - jest to podstawowa i najprostsza implementacja ECS związana z paradygmatem DOD. Można tę implementację przedstawić jako macierz (przykładowa wizualizacja na Rysunek 2.6), gdzie kolumną jest komponent, a wierszem entity (byt). Dzięki temu odpytując się o komponent danego typu uzyskiwana jest tablica tego komponentu. Aby stwierdzić czy dany byt ma dany komponent wystarczy w macierzy wskazać komórkę i przeczytać jej wartość. Sposób ten jest prymitywny i posiada pewien problem - trzymając wszystkie dane w postaci jednej macierzy, tablice komponentów staną się w większości puste, przez co znowu pojawią się problemy z wykorzystaniem pamięci podręcznej. Natomiast wykorzystywanie większej ilości macierzy, wymaga znacznie więcej pracy i utrudnia wprowadzenie zmian.

	Health	Mana	Speed	DeathTag
Entity 1	0	0	0	1
Entity 2	1	1	1	0
Entity 3	1	0	0	0
Entity 4	0	1	0	1

Rysunek 2.6 Przykładowa wizualizacja zbioru bitowego użytego do filtrowania bytów w ECS

• Sparse set-based (oparty na rzadkich zbiorach) - jest to rozwinięcie wersji opartej na zbiorach bitowych, wykorzystujące sprytną strukturę danych jaką jest sparse set (zbiór rzadki), szczególnie przydatny, kiedy nie można pozwolić na marnotrawienie miejsca/pamięci, a operacje przeprowadzane są w większości na danych tablicowych, które są w dużej mierze puste. Gwarantuje to lepszą utylizację pamięci i jest bardziej przyjazne dla systemów pamięci podręcznej (dane są przechowywane liniowo). Możliwe są znaczne usprawnienia, takie jak łączenie kilku tablic w grupę i sortowanie elementów tak, żeby iteracja przez całą grupę była jak najbardziej wydajna (więcej możliwości optymalizacyjnych można znaleźć w implementacji i blogu autora biblioteki EnTT [24])

• Archetype-based (oparty na archetypie) - najpierw należy przybliżyć czym jest archetyp. Archetypem nazywana jest krotka, zawierającą wszystkie typy komponentów jakie należą do danego bytu. Podejście te polega na tworzeniu archetypów i rezerwowania pamięci dla komponentów per archetyp. Żeby sprawdzić czy dany byt posiada komponent danego typu, wystarczy sprawdzić dla jakiego archetypu należy byt i czy dany archetyp zawiera dany komponent. Podejście te jest najbardziej skomplikowanym ze wszystkich opisanych, jednak dobrze wykorzystując mechanizmy, posiadamy najwięcej szans na optymalizację (z pośród wymienionych tu podejść). Taką implementację posiada biblioteka ECS przygotowana przez korporację Unity Technology dla swojego silnika w postaci paczki Entities, na której polega narzędzie opisywane w tej pracy.

2.4.2. Użyteczność przy trenowaniu sztucznej inteligencji

Sztuczna inteligencja wkracza w codzienne życie coraz szybciej. Ten postęp rodzi nowe problemy, jednym z nich jest to, że rozwiązania z zakresu technik uczenia maszynowego wymagają skomplikowanego procesu uczenia modeli. Idealnie byłoby uczyć i testować programy na docelowych platformach, jednak jest to rozwiązanie fizycznie niemożliwe z wielu powodów, np.:

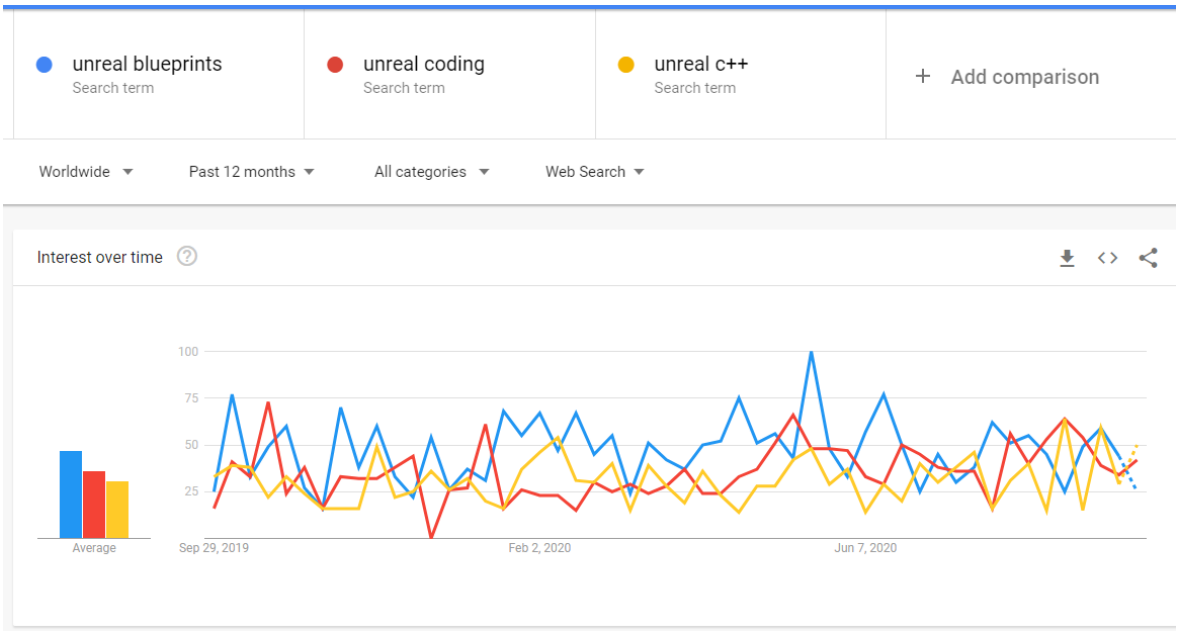
• Ogromne koszty produkcji maszyn, ustawiania platform, likwidacji uszkodzeń i utylizacji uszkodzonych maszyn (samo jeżdżący pojazd ulegający kolizji).

• Wydłużenie czasu uczenia i niska możliwość skalowania (na odcinku jednego pasa jezdni może poruszać się tylko jeden pojazd z ograniczoną, przepisami, prędkością).

Aby pozbyć się tych fizycznych ograniczeń, można zastosować wirtualne środowiska i programy z odpowiednio 'nauczonymi' parametrami dopiero testować w fizycznych środowiskach. Silniki gier są naturalnym wyborem dla tworzenia tych wirtualnych środowisk (gotowe i wysokiej jakości silniki fizyczne i graficzne, rozwiązania zoptymalizowane do szybkich obliczeń, funkcjonalne środowiska graficzne tzw. edytor, gotowe integracje z wieloma platformami) [25]. Dzięki transparentności, małej ilości zależności, znakomitym możliwością optymalizacji i skalowalności zastosowanie wzorca ECS (w wersji zgodnej z założeniami projektowania zorientowanego na dane) przy tworzeniu tego typu symulacji komputerowej, wydaje się naturalnym kierunkiem rozwoju, który pozwoli zaoszczędzić dużo czasu i pieniędzy oraz może przyczynić się do jeszcze szybszego rozwoju tego typu rozwiązań.

2.5. Programowanie wizualne

Programowanie wizualne (visual programming) - technika programowania, gdzie zamiast pisania kodu, układa się i łączy ze sobą odpowiednie elementy wizualne odpowiadające pewnym nazwanym operacjom, normalnie realizowanym przy pomocy kodu. Umożliwia to inne przedstawienie problemu i rozwiązania, co może ułatwiać zrozumienie dla osób nietechnicznych [26] [27]. Technika ta jest często wykorzystywana w silnikach gier, gdyż umożliwia projektantom tworzenie prostych interakcji i mechanik w sposób wizualny (i nie wymagający pisania kodu). Dwa największe silniki posiadają tego typu rozwiązania: Unity kupiło rozwiązanie zwane Bolt [28] i rozwija własne rozwiązanie oparte o swój stos technologiczny DOTS [29], natomiast Unreal Engine posiada swoje własne rozwiązanie zwane Blueprints [30], które jest uznawane za jedno z najlepszych (jeśli nie najlepsze) rozwiązanie tego typu. Programowanie przy użyciu tego systemu jest bardziej popularne niż przy wykorzystaniu języka C++ (na podstawie danych z Rysunek 2.7). Niezwykle popularne rozwiązania związane z programowaniem wizualnym to programowanie shader'ów (programów przeznaczonych do kolorowania pikseli na GPU) [31] [32] [33].



Rysunek 2.7 Porównanie ilości wyszukiwani dla zapytań: „unreal blueprints", „unreal coding" i „unreal c++". Z serwisu https://trends.google.com. Dostęp 29.09.2020.

Może być zrealizowane jako:

• Uniwersalny system, który przetwarza wizualną formę w czasie wykonywania programu i wykonuje odpowiadające jej funkcje wraz z przekazywaniem zadanych danych. Zaletą tego podejścia jest łatwość rozbudowy i brak konieczności kompilacji zmian (możliwość wprowadzania zmian podczas wykonywania programu). Wadą jest oczywiście wydajność takiego rozwiązania, gdzie poza czasem poświęconym na wykonywanie rzeczywistej logiki biznesowej, potrzebny jest czas na kierowanie przepływem i analiza części wizualnej.

• Oddzielenie warstwy wizualnej od warstwy egzekucji kodu, na podstawie danych z warstwy wizualnej generowany jest kod, który zostaje skompilowany i dołączony do bazy kodu całego programu/gry. Zaletami tego rozwiązania jest wydajność, która może być taka sama jak dobrze napisanego kodu. Dodatkowo oprócz optymalizacji przy zmianie części wizualnej na kod otrzymywane są także optymalizacje z kompilacji kodu. Wadą natomiast jest rozrost bazy kodu i wydłużony czas testowania rozwiązań (osoby nietechniczne posiadają predyspozycje do tworzenia programów 'na oślep' tzn. wybierania losowych wartości i funkcji aż do osiągnięcia zadowalającego celu).

• Rozwiązania hybrydowe, gdzie warstwa wizualna wymaga pewnych przekształceń do warstwy wykonywanej przez uniwersalny system. Pozwala na zastosowanie pewnych optymalizacji i interaktywne wprowadzanie zmian, jednak trzeba odczekać czas przetwarzania, zanim zmiany będą widoczne, a wydajność jest gorsza niż przy rozwiązaniach opartych na generowaniu kodu kompilowanego.

2.6. Generacja kodu

Code generation (generacja kodu) - technika pozwalająca przekształcić dane wejściowe w poprawny syntaktycznie kod programu. Jest to bardzo szeroka rodzina technik ułatwiających i wspomagających pracę programistyczną, która zaczyna się od prostych szablonów i systemów uzupełniania składni (IntelliSense i IntelliCode [34]), poprzez rozbudowane szablony uzupełniane dynamicznie wpisanymi danymi (np. generowanie kodu C# z modelu UML w Visual Paradigm [35]) do generowania

całych programów (np. ShaderGraph [36]). Może odnosić się również do procesu przemiany kodu napisanego w języku programowania do kodu maszynowego.

2.7. Systemy sztucznej inteligencji w grach

W grach komputerowych można zauważyć kilka głównych systemów:

• System graficzny - odpowiadający za renderowanie elementów graficznych.

• System UI (User Interface [interfejs użytkownika]) - odpowiadający za graficzny interfejs użytkownika (może opierać się na systemie graficznym).

• System interakcji - odpowiadający za przetwarzanie sygnałów interakcji (np. unifikacja obsługi kontrolerów).

• System dźwiękowy - odpowiada za przetwarzanie źródeł dźwięków i powiązanych z nimi plików audio.

• System zasobów - odpowiadający za zarządzanie zasobami gry i interakcje IO (wejście-wyjście [input-output]).

• System fizyki - odpowiadający za symulowanie praw fizyki (lub ich uproszczonych interpretacji).

• System skryptowania - odpowiadający za wykonywanie logiki biznesowej stworzonej w dedykowanej lub zaadaptowanej dla silnika technologii.

• System mechanik gry - odpowiadający za wykonywanie logiki biznesowej głównych mechanik.

• System sztucznej inteligencji - odpowiadający za sterowanie NPC (postać w grze nie kontrolowana przez gracza [non-player character]).

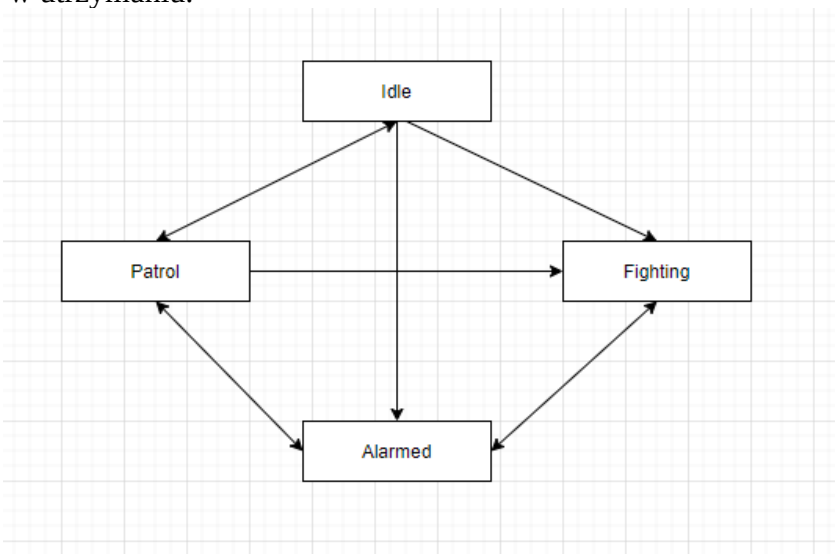
Przy założeniu, że gra musi wykonywać się przy prędkości 60 FPS(klatek na sekundę [frames per second]) to wszystkie te systemy mają łączny budżet ~16.7ms. Kiedy gra posiada znakomity system (lub artefakty związane z jego efektami) graficzny, dźwiękowy, interakcji, mechanik gry, skryptowania, fizyki lub dźwiękowy, to jest to automatycznie zauważalne przez graczy (przykładami tutaj mogą być takie gry jak: Crysis, Terraria, Minecraft, Crypt of the NecroDancer czy Noita). Natomiast gier, w których głównym atrybutem jest sztuczna inteligencja, jest niezwykle mało. Rodzi to swego rodzaju odmianę problemu kury i jajka - nie przeznacza się na ten aspekt dużo czasu i środków, ponieważ nie ma gier, które poświęciły na to dużo czasu i środków. Nie wiadomo więc, czy jest to dobry kierunek. Przy proporcjonalnym podziale wychodzi mniej niż 2ms na każdy z systemów gry. Natomiast uwzględniając wyższy priorytet innych systemów przy przyznawaniu czasów wykonania na klatkę, system sztucznej inteligencji otrzymuje budżet w okolicach 1ms na wykonanie obliczeń z nim związanych. Mając to na uwadze można zauważyć, że aby wykonać ten system zgodnie z najlepszymi praktykami inżynieryjnymi, należy zastosować dużo uproszczeń, optymalizacji i przybliżeń (np. nie potrzeba wyszukiwać najszybszej ścieżki, wystarczy, że będzie jedną z kręgu najszybszych), aby sprostać wymaganiu czasu wykonywania.

2.8. Techniki implementacji sztucznej inteligencji w grach

2.8.1. Maszyna stanów skończonych

Automat skończony, skończona maszyna stanów (finite state machine – FSM) - jedna z podstawowych technik tworzenia zachowań dla postaci sterowanych przez komputer (NPC) [8] [37]. Polega na definiowaniu stanów wraz z przypisanymi do nich zachowaniami i (w rozszerzonej, ale powszechnie używanej formie) łączeniu tych stanów ze sobą tranzycjami (Rysunek 2.8). W najprostszej formie (w programowaniu zorientowanym obiektowo) definiujemy zarządcę maszyny stanów i stan. Maszyna stanów wywołuje na aktualnym stanie cykliczną metodę i pozwala zmienić aktualny stan na inny. Taką implementację można modyfikować/rozbudować o np. dodanie stosu stanów,

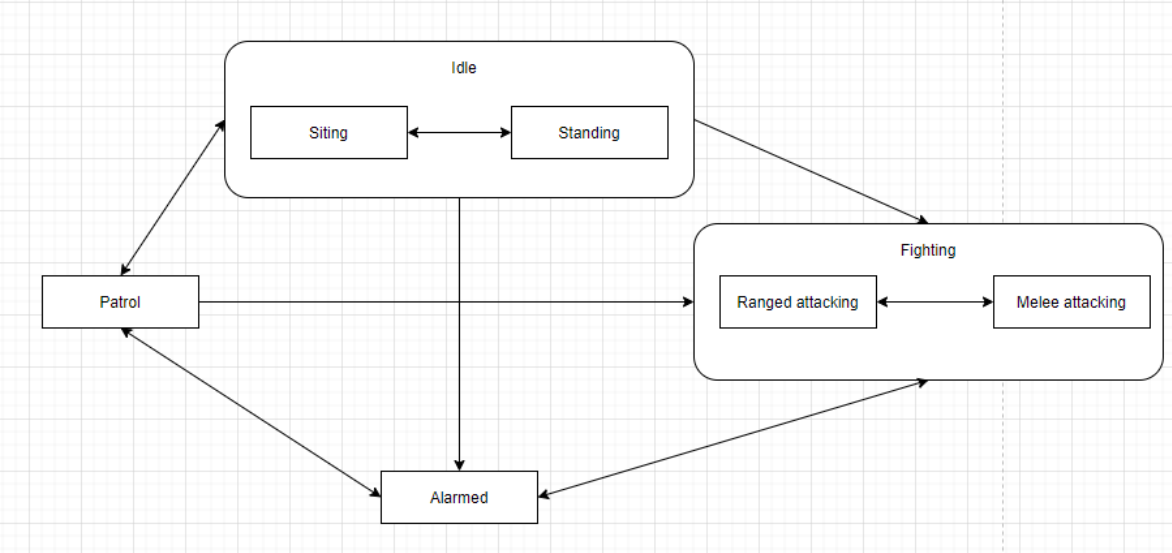
ograniczanie zmian stanów (określenie dla danego stanu w jakie stany może przejść) i/lub automatyczne przejścia realizujące się przy zadanych warunkach. Aby ułatwić pracę z tą techniką, często stosuje się techniki wizualizacyjne łączone z technikami wizualnego programowania i/lub (czasami) z technikami automatycznej generacji kodu. W silniku Unity istnieje podstawowa implementacja maszyny stanów jako system animacji nazwany „animation state machine", można też skorzystać z innych implementacji [38]. Dużą wadą tego typu techniki jest to, że poziom skomplikowania jest większy niż liniowy w stosunku do ilości zadeklarowanych stanów, więc wszelkie zmiany są utrudnione, a cały system staje się trudny w utrzymaniu.



Rysunek 2.8 Przykładowa maszyna stanów skończonych, gdzie prostokąt oznacza stan, linia ze strzałką pojedynczą oznacza przejście jednostronne od stanu bez strzałki do stanu ze strzałką i linia zakończona strzałkami z obydwu stron oznacza połączenie stanów z tranzycją w obydwie strony.

2.8.2. Hierarchiczna maszyna stanów

Hierarchiczna maszyna stanów (hierarchical state machine – HSM) - jest to rozwinięcie skończonej maszyny stanów, gdzie stan może albo bezpośrednio definiować zachowania jakie reprezentuje, albo być zagnieżdżoną maszyną stanów [8] [39], tak jak pokazano na Rysunek 2.9. Oczywiście w pojęciu matematycznym też jest to automat skończony, jednakże w tym opisie nie chodzi o definicję matematyczną, tylko o technikę programowania. Mimo tego nadal jest prawdą, że każda hierarchiczna maszyna stanów jest również skończoną maszyną stanów. Każdą hierarchiczną maszynę stanów można przedstawić jako skończoną maszynę stanów, gdyż HSM jest tylko techniką wprowadzenia abstrakcji, aby uczynić FSM bardziej przyjaznymi projektantom. W opisie FSM jest podany przykład systemu animacji w silniku Unity, ten przykład jest również poprawny w kontekście hierarchicznych maszyn stanów.



Rysunek 2.9 Przykładowa hierarchiczna maszyna stanów, gdzie prostokąt z zaokrąglonymi rogami oznacza stan, który jest zagnieżdżoną maszyną stanów, prostokąt oznacza stan realizujący zachowanie, linia ze strzałką pojedynczą oznacza przejście jednostronne od stanu bez strzałki do stanu ze strzałką i linia zakończona strzałkami z obydwu stron oznacza połączenie stanów z tranzycją w obydwie strony.

2.8.3. Drzewa zachowań

Behaviour tree (drzewo zachowań) - jest to technika modelowania przepływu zachowań [40], najczęściej występuje w formie graficznej (Rysunek 2.10), jednak może także występować w formie kodu. Ten drzewiasty model składa się z liści (węzłów realizujących pojedyncze składowe zachowania), połączeń, węzłów kompozycyjnych (węzłów które posiadają przynajmniej jedno dziecko i kontrolują przepływ sterujący) i węzłów dekoracyjnych (węzłów które posiadają dokładnie jedno dziecko i mogą zmieniać rezultat swojego dziecka). Liście zwracają jeden z 3 rezultatów:

- Sukces - kiedy liść poprawnie wykona swoje zadanie.
- Porażka - kiedy liść nie może poprawnie wykonać swojego zadania.
- Wykonywanie - kiedy wykonywanie liścia nie jest operacją synchroniczną.

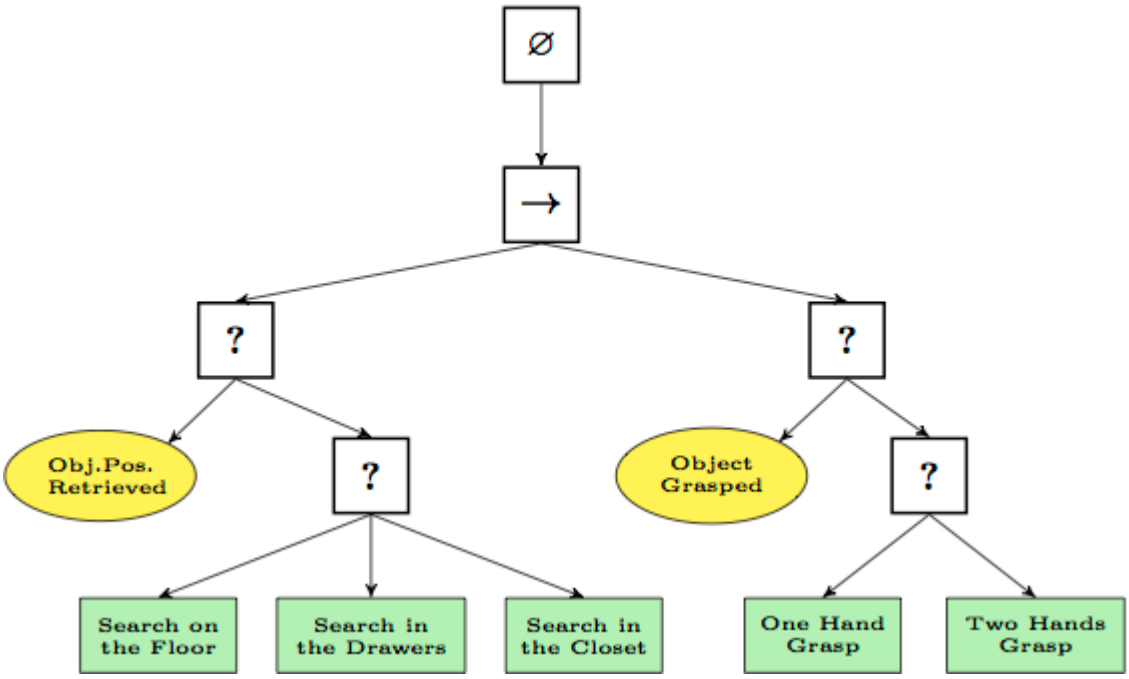
Rozróżnia się następujące węzły kompozycyjne:

- Sekwencja - wykonuje swoje dzieci do póki zwracają sukces. Jeżeli jakiekolwiek dziecko zwróci porażkę, zwraca porażkę, w przeciwnym wypadku zwraca sukces.
- Selektor - wykonuje swoje dzieci aż do pierwszego zwróconego sukcesu. Jeżeli którekolwiek dziecko zwróci sukces, selektor zwraca sukces, w przeciwnym wypadku zwraca porażkę.

Jedne z podstawowych węzłów dekoracyjnych to:

- Negacja - kiedy dziecko zwróci sukces, zmienia rezultat na porażkę, a kiedy dziecko zwróci porażkę to zwraca sukces.
- Wygrana - zawsze zwraca sukces, niezależnie od rezultatu dzieci.
- Powtarzacz - wykonuje węzeł dziecka określoną ilość razy, po wykonaniu zwraca sukces. W bardziej rozbudowanych wersjach może wykonywać aż do określonego rezultatu.

Pozwala to na usunięcie bezpośrednich zależności między zachowaniami/akcjami i ułatwia ponowne użycie pojedynczych zachowań. Przy maszynie stanów stan nie może być użyty wielorazowo. Co zwiększa uniwersalność i łatwość w utrzymaniu rozwiązań opartych o drzewa zachowań względem rozwiązań opartych na maszynach stanów.



Rysunek 2.10 Przykładowe drzewo decyzyjne z https://commons.wikimedia.org/wiki/File:BT_search_and_grasp.png. Dostęp 24.10.2020.

2.8.4. Planowanie akcji zorientowanych na cel

Planowanie akcji zorientowane na cel (Goal-Oriented Action Planning - GOAP) - technika pozwalająca zaplanować sekwencję akcji, które prowadzą do określonego celu (minimalizując koszt osiągnięcia celu) [41] [42]. Technika ta polega na zdefiniowaniu zbioru akcji, każda akcja składa się z:

- warunków wstępnych - zbiór warunków, które muszą wystąpić w świecie, aby dana akcja mogła zacząć się wykonywać.
- efektów - zbiór zmian w stanie świata/kontekstu, które wprowadzi akcja po wykonaniu się.
- implementacji - czyli zbiór instrukcji, które są wykonywane podczas wykonywania akcji.

Dzięki temu akcje nie są ze sobą połączone (brak zależności pomiędzy nimi), poprzez to systemy wykorzystujące GOAP są łatwiejsze w utrzymywaniu, rozbudowywaniu i posiadają niższą złożoność zależności. Dodatkowo GOAP ogranicza interakcje projektanta z systemem, co zmniejsza podatność na błędy ludzkie. Nie jest to jednak system służący do tworzenia zachowań postaci, tylko do wyznaczania sekwencji, aby osiągnąć cel. Technika ta musi być połączona z jakąkolwiek techniką tworzenia zachowań (np. Maszyną stanów lub drzewami zachowań). Połączenie z maszyną stanów eliminuje potrzebę na definiowanie tranzycjii między stanami, czyli największej wady w tej technice.

3. Analiza wymagań projektu

3.1. Presony

3.1.1. Krzysztof Abacki:

Cechy:

- Młody programista gier komputerowych
- 21 lat
- Od 3 miesięcy pracuje przy mobilnej grze kategorii Indie (gra niezależna o niewysokim budżecie)
- Jest to jego pierwsza praca
- Zna OOP, ale dopiero poznaje DOD i ECS
- Mieszka w Tallinnie

Potrzeby:

- Narzędzie, które wcześnie i czytelnie poinformuje go o błędach
- Narzędzie, które pozwoli stopniowo zapoznawać się ze wzorcem ECS
- Narzędzie, które pomoże mu z komunikacją z projektantem
- Narzędzie, które pomoże mu ograniczyć czas zabierany inny programistom na pytania

3.1.2. Alex Baldwin:

Cechy:

- 32 lata
- CTO oddziału dużego studia gier w Vancouver
- Obejmuje te stanowisko od roku
- Pracuje w tej firmie od 4 lat
- Pracuje 12 lat w branży gier komputerowych jako programistka
- Zna DOD i ECS
- Zaczyna nowy projekt gry komputerowej na komputery PC typu AA
- Chce użyć Unity Entities

Potrzeby:

- Narzędzie, które wskaże, jak korzystać z paczki Unity Entities
- Narzędzie, które pomoże jej z komunikacją pomiędzy projektantami a programistami

3.1.3. Jackob Brugière:

Cechy:

- 54 lata
- 4 lata pracuje jako projektant gier komputerowych
- Pracował tylko przy grach mobilnych
- Zaczyna pracę przy nowej grze na Nintendo Switch
- Mieszka we Francji na wsi
- Pracuje zdalnie
- Robi kurs z psychologii

Potrzeby:

- Narzędzie, które będzie intuicyjne i wizualne

• Narzędzie, które zapewni edytowalność tylko tego wycinka domeny, na którym operuje

• Narzędzie, dzięki któremu może przekazać swoją wizję osobie technicznej

3.2. Historyjki użytkownika

Jako, że rodzaje użytkowników zostały zdefiniowane jako persony, to w historyjkach użytkowników używane są ich nazwy (imiona).

1) Jako Krzysztof chcę uzyskać komunikat o błędzie, aby dowiedzieć się co robię źle.
2) Jako Krzysztof chcę uzyskać czytelny komunikat o błędzie, aby móc samemu rozwiązać zaistniały problem.
3) Jako Krzysztof chcę uzyskać komunikat o błędzie jak najszybciej, aby jak najszybciej poprawić błędne ustawienia.
4) Jako Krzysztof chcę móc użyć narzędzia bez długiej fazy zapoznawania się z nim, aby szybko uzyskać efekty.
5) Jako Krzysztof chcę móc przeglądać wygenerowany kod, aby zapoznać się ze wzorcem ECS w implementacji Unity Entities.
6) Jako Krzysztof chcę operować w formie wizualnej, aby móc szybciej i przyjemniej poznać narzędzie.
7) Jako Krzysztof chcę, żeby narzędzie było podobne do innych narzędzi wizualnych, aby móc szybciej i przyjemniej poznać narzędzie.
8) Jako Krzysztof chcę, żeby forma wizualna pozwalała mi na dialog z projektantami, aby móc sprawnie realizować mechaniki gry.
9) Jako Alex chcę uzyskać czytelny komunikat o błędzie, aby móc samemu rozwiązać zaistniały problem.
10) Jako Alex chcę uzyskać w pełni edytowalny kod, aby móc zastosować ręczne optymalizacje.
11) Jako Alex chcę móc przeglądać wygenerowany kod, aby zapoznać się ze wzorcem ECS w implementacji Unity Entities.
12) Jako Alex chcę, aby narzędzie było wizualne, aby mogli z niego korzystać nie tylko programiści, ale również projektanci.
13) Jako Alex chcę, aby narzędzie było zgodne ze standardami innych narzędzi wizualnych, aby zespół nie musiał spędzić dużo czasu ucząc się narzędzia.
14) Jako Alex chcę narzędzie z otwartym kodem, aby móc dostosować je pod wymagania projektu.
15) Jako Jackob chcę uzyskać czytelny komunikat o błędzie, aby móc przesłać go do osoby technicznej.
16) Jako Jackob chcę uzyskać czytelny komunikat o błędzie, aby móc spróbować rozwiązać problem samodzielnie.
17) Jako Jackob chcę operować wizualnie, ponieważ nie potrafię programować.
18) Jako Jackob chcę mieć ograniczony widok do funkcjonalności projektowania, aby nie musieć rozumieć kwestii technicznych.
19) Jako Jackob chcę, aby interfejs graficzny był spójny z innymi popularnymi narzędziami, aby jak najszybciej móc dostarczać wartość w projekcie.
20) Jako Jackob chcę mieć możliwość wizualnego przedstawiania moich koncepcji osobom technicznym, aby mogli oni lepiej zrozumieć moje wymagania.

3.3. Wymagania

Przed rozpoczęciem prac nad danym kamieniem milowym ustalono wymagania funkcjonalne i niefunkcjonalne. Takie podejście pomogło uniknąć zmian w wymaganiach i straty czasu na wyłuskanie wymagań, które wraz z rozwojem projektu stałyby się

nieaktualne lub wadliwe. Przykładem tutaj mogłyby być wymagania dla kamienia milowego, który w trakcie walidacji postępu prac nad projektem okazał się niezgodny z osiągniętym kształtem projektu.

3.3.1. Kamień milowy 1:

Funkcjonalne:

- Generacja pustych komponentów i systemów z poprawnym zapytaniem (query) i przestrzenią nazw (namespace).
- Walidacja ustawienia komponentów w węzłach i blokada generacji przy błędnie ustawionych komponentach lub węzłach
- Zmiana typu komponentu z „Ręcznie-wpisanego" na „Twardo-otypowany" po generacji i rekompilacji
- Zmiana typu komponentu z „Twardo-otypowanego" na „Ręcznie-wpisany", kiedy komponent zostanie usunięty z bazy kodu
- Kolorowanie błędnie ustawionych węzłów na kolor czerwony
- Wyświetlanie błędnych ustawień w węźle

Niefunkcjonalne:

- Widoczne i zrozumiałe komunikaty błędów
- Estetycznie i intuicyjnie wyglądające węzły
- Zgodność estetyczna z wyglądem Unity Pro w trybie ciemnym

3.3.2. Kamień milowy 2:

Funkcjonalne:

- Definiowanie i generacja wielu lambd dla pojedynczego systemu
- Obsługa (wraz z automatycznym wykrywaniem) rodzajów planowania i zmian strukturanych
- Wsparcie opcjonalnych ustawień w lamdbach
- Generowanie poprawnego kodu
- Tworzenie filtrów komponentów

3.3.3. Kamień milowy 3:

Funkcjonalne:

- Analizowanie tworzonych komponentów
- Tryb uproszczony pozwalający na wysokopoziomowe spojrzenie na systemy
- Możliwość wyświetlania podsumowania systemu
- Wczytywanie istniejących systemów jako systemów „tylko do odczytu"

Niefunkcjonalne:

- Zwiększenie przejrzystości narzędzia

4. Projekt

4.1. Konkurencyjne rozwiązania

Na czas rozpoczęcia pracy nad projektem (15.07.2020) nie udało się znaleźć autorowi bezpośrednio konkurencyjnego rozwiązania. Jednak istnieje grono konkurencyjnych narzędzi, z których można czerpać inspirację i powielać część ich rozwiązań. Przedstawiona zostanie lista takich narzędzi wraz z ich analizą, mającą na celu wskazanie rozwiązań, którymi warto się wspierać, jak i różnice względem proponowanego produktu omawianego w pracy. Przedstawione zostaną także mocne i słabe strony tych rozwiązań.

4.1.1. Visual Paradigm

Visual Paradigm jest narzędziem do projektowania systemów informatycznych ze szczególnym nastawieniem na zarządzanie projektem i projektowaniem oprogramowania oraz infrastruktury mu towarzyszącej [35]. Jednak obsługiwane modele projektowania nie mają wsparcia ani do modeli projektowania zorientowanych na dane, ani wzorca entity-component-system, ani silnika Unity. Zarówno dużym plusem i dużym minusem jest rozmiar tego oprogramowania. Pozwala ono na wiele i posiada znaczną ilość zaimplementowanych rozwiązań, co z drugiej strony daje znaczny próg wejścia i mnogość wiedzy, którą trzeba opanować, aby skutecznie używać te narzędzie. Na pewno należy zapoznać się z systemem generowania kodu przez Visual Paradigm, ponieważ jest to bardzo dobre rozwiązanie. W momencie, kiedy w tym narzędziu projektowane jest oprogramowanie, to dla wielu języków programowania, można bez zbędnej pracy, w kilka sekund uzyskać wygenerowany szkielet aplikacji zgodny z założeniami architektonicznymi projektu. Podsumowanie znajduje się w Tabela 4.1 Podsumowanie narzędzia Visual Paradigm.

Tabela 4.1 Podsumowanie narzędzia Visual Paradigm

Inspiracje	Mocne strony	Słabe strony
Rozbudowana generacja kodu z utworzonych diagramów	Dużo możliwości i funkcjonalności	Wysoki próg wejścia
	Możliwość wykorzystania jednego narzędzia w całym procesie wytwarzania oprogramowania	Brak wsparcia dla DOD, ECS i Unity
		Nacisk na metody związane z programowaniem obiektowym
		Narzędzie płatne

4.1.2. Unreal Engine Blueprints

Blueprints Visual Scripting jest oficjalnym systemem wizualnego skryptowania w silniku Unreal Engine [30]. W poprzednim rozdziale 2.5 Programowanie wizualne rozwiązanie te zostało już przybliżone. Jako, że jest to najpopularniejsze rozwiązanie do skryptowania w silniku Unreal, to jest ono oczywistą inspiracją. Oczywiście do wad tego rozwiązania należą, tak jak w Visual Paradigm, brak wsparcia dla DOD, ECS i silnika Unity. Podsumowanie tego narzędzia znajduje się poniżej w Tabela 4.2 Podsumowanie narzędzia Unreal Engine Blueprints.

Tabela 4.2 Podsumowanie narzędzia Unreal Engine Blueprints

Inspiracje	Mocne strony	Słabe strony
Wykonanie, przejrzystość i przyjazność dla nowych użytkowników	Mnogość dostępnych przykładów i poradników	Brak wsparcia dla DOD, ECS i Unity
Generacja kodu na podstawie wizualnej formy	Intuicyjność rozwiązania	Ograniczenie do silnika Unreal Engine
	Darmowe	Niska czytelność i możliwości edycji generowanego kodu

4.1.3. Unity Bolt

Unity Bolt jest narzędziem, które Unity Technologies nabyło od firmy ludiq [28]. Na ten moment jest to jedyne oficjalne i gotowe do użycia w produkcji narzędzie do wizualnego skryptowania dostępne dla silnika Unity. Tak jak Blueprints, tak samo to narzędzie zostało już wspomniane w rozdziale 2.5 Programowanie wizualne. Pod względem podstawowej obsługi, jest to narzędzie zbliżone do rozwiązania od Epic Games, niestety nie generuje ono kodu i tak samo jak poprzednie narzędzia, nie wspiera ani paradygmatu projektowania zorientowanego na dane, ani wzorca ECS, ani paczki Entities. Podsumowanie rozwiązania Bolt znajduje się w Tabela 4.3 Podsumowanie narzędzia Unity Bolt.

Tabela 4.3 Podsumowanie narzędzia Unity Bolt

Inspiracje	Mocne strony	Słabe strony
Wykonanie, przejrzystość i przyjazność dla nowych użytkowników	Zgodność z silnikiem Unity	Brak wsparcia dla DOD i ECS
	Intuicyjność rozwiązania	Brak możliwości generowania kodu (będzie to możliwe w najnowszej wersji bolt 2)
	Darmowe	

4.1.4. DOTS Visual Scripting

Jest to rozwiązanie, które znajduje się we wczesnej fazie eksperymentalnej [29]. W kontekście wymagań technicznych, te narzędzie jest najbardziej konkurencyjne dla zaproponowanego rozwiązania. Jest oparte ono na paczce Entities, jednak ciężko wskazać czy rozwiązanie te jest zgodne z podejściem DOD, ponieważ nie opiera się ono na generacji kodu, a analiza kodu tego rozwiązania wykracza poza zainteresowanie niniejszej pracy. Po parogodzinnym zapoznaniu z narzędziem, można stwierdzić, iż w obecnej formie jest ono zdecydowanie mniej przyjazne od strony interfejsu i obsługi niż Blueprints lub Bolt. Co ciekawe, na pierwszy rzut oka, dzięki zastosowaniu podejścia opartego na węzłach, narzędzie wydaje się intuicyjne, jednak implementacja wielu aspektów jest dziwna i nie zrozumiała dla autora. Znacznie odstaje od podobnych funkcjonalności w innych rozwiązaniach i to w negatywny sposób. Podsumowanie znajduje się poniżej w Tabela 4.4 Podsumowanie narzędzia DOTS Visual Scripting.

Tabela 4.4 Podsumowanie narzędzia DOTS Visual Scripting

Inspiracje	Mocne strony	Słabe strony
Brak (na ten moment, w przyszłości może stanowić główną konkurencję do przedstawianego narzędzia)	Zgodność z paczką Unity Entities	Nieintuicyjne wykonanie
	Wizualne programowanie	Brak możliwości generacji kodu
		Ograniczone możliwości wsparcia paradygmatu programowania zorientowanego na dane

4.1.5. ECS_FSM

Rozwiązanie od użytkownika kaminaritukane udostępnione na platformie GitHub [43]. Jest to mała implementacja skończonej maszyny stanów przy użyciu Unity Entities, niestety na dzień 07.11.2020 narzędzie to nie otrzymało żadnej aktualizacji od 5 miesięcy. Dodatkowo jedyna dokumentacja narzędzia to jeden wpis na blogu. Po analizie tego rozwiązania autor zauważył, iż jest to implementacja maszyny stanów z wykorzystaniem wzorca ECS jednak nie jest zgodne z paradygmatem DOD [1] [44], ponieważ może prowadzić do bardzo nieoptymalnego wykorzystania pamięci i linii pamięci podręcznej. Dodatkowo jest to rozwiązanie oparte w 100% na kodzie (brak wizualizacji dla projektantów). Podsumowanie znajduje się w Tabela 4.5 Podsumowanie narzędzia ECS_FSM.

Tabela 4.5 Podsumowanie narzędzia ECS_FSM

Inspiracje	Mocne strony	Słabe strony
Brak	Zgodność z paczką Unity Entities	Brak projektowania i programowania wizualnego
		Brak zgodności z paradygmatem projektowania zorientowanego na dane
		Brak dokumentacji, rozwoju i wsparcia

4.2. Proces realizacji projektu

Implementacja została podzielona na tzw. kamienie milowe. Pierwszym kamieniem milowym było zaplanowanie projektu (w tym wybór narzędzi) i wyznaczenie pozostałych kamieni milowych. Następnie wyznaczono 3 (początkowo 4) kamienie milowe, które wchodzą w skład pracy i 4 kamienie milowe jako perspektywy rozwoju projektu. Pomocne przy wyznaczeniu tych punktów były persony i historyjki użytkowników, na podstawie których odbywała się klaryfikacja ogólnych motywów poszczególnych kamieni milowych.

0. Zaplanowanie projektu
1. Ustawienie narzędzi, wizualizacja i przykładowa generacja kodu

2. Wiele wyrażeń lambda, komponenty dzielone (shared components) i filtrowanie

3. Refaktoryzacja, usprawnienia wizualne, tworzenie komponentów i narzędzia pomocnicze

4. GOAP

5. Paczka Unity obsługiwana przez Package Manager

6. Tworzenie logiki dla lambd w wersji pełnego programowania wizualnego

Takie podejście pomogło lepiej zaplanować przebieg projektu jak i zapewniło punkty kontrolne, co pozwoliło na usunięcie kamienia milowego nazwanego: "HSM - hierarchiczna maszyna stanów", który początkowo znalazł się w zaplanowanym projekcie, jednak wraz z rozwojem projektu stało się jasne, że takie rozwiązanie nie ma racji bytu i się nie sprawdzi. Podejście takie jest zgodne z metodykami zarządzania projektami Agile-Kanban, podejście te jest bardziej elastyczne niż podejście Scrum, co okazało się bardzo pomocne. Oczywiście w projekcie, gdzie zespół deweloperski składa się z jednej osoby, różnice w metodologiach są nieduże i granica między nimi może się zacierać.

4.3. Narzędzia

4.3.1. Unity

Jest to jeden z największych siników gier na świecie (w swoim S-1 [45] Unity Technology estymuje że 53% z 1000 największych gier dostępnych w Apple Store i Google Play Store i 50% gier mobilnych, komputerowych i konsolowych zostało stworzonych z wykorzystaniem ich technologii). Niestety w obecnej, stabilnej wersji silnika (2020.1), bez zewnętrznych ingerencji, sinik ten nie zapewnia takiego poziomu wydajności (przy nieco gorszych rezultatach audio-wizualnych) jak jej bezpośredni rywale (silnik Unreal od Epic Games, silnik Frostbite od DICE (używany przez Electronic Arts), czy też silnik CryEngine od Crytek). Mimo, iż wydajność nigdy nie była głównym atrybutem, dla którego wybiera się silnik Unity, to stosunkowo niska wydajność jest problemem, który jest widziany i adresowany przez technologie z rodziny DOTS. Ten stos technologiczny mocno wspiera podejście DOD i pozwala zniwelować różnice wydajnościowe pomiędzy technologiami (Unity, mimo iż napisane C++ opiera się na platformie Mono [46], a wymienione powyżej silniki są całkowicie oparte na języku C++). Można by zastanawiać się "skoro jest to tak popularny silnik, to jego użytkownikom nie przeszkadzają aż tak bardzo braki wydajnościowe". Pomijając aspekt inżynieryjny (on zostanie omówiony na końcu), istnieje aspekt ekologiczno-społeczny. Jeżeli przyjmiemy, że 50% gier jest tworzonych przy użyciu technologii Unity i uzyskamy ponad 50% lepszą wydajność (w testach najmniejsze przyśpieszenie wynosiło ~29%, większość testów wskazywała ponad 50% lepszą wydajność, a znaczna część [ponad połowa] testów dla pojedynczej precyzji wskazywała ponad 80% poprawę wydajności) [47], to oszczędność energetyczna powinna być znacząca w skali świata, dodatkowo przyczyni się to zmniejszonego zużycia komponentów komputerów, telefonów i konsol. Impakt ten będzie rósł z roku na rok, gdyż technologie Unity wychodzą coraz śmielej poza branżę gier, w kierunkach takich jak branże filmowa, motoryzacyjna, architektoniczna lub nowoczesny marketing i sprzedaż [45]. Z inżynieryjnego punktu widzenia, systemy, które są produkowane zawsze powinny dbać o to, żeby uzyskać jak najlepsze parametry jakie można osiągnąć.

4.3.2. DOTS

Data-Oriented Technology Stack (DOTS) - jest zbiorem technologii opracowanych na potrzeby silnika Unity. W skład tych technologii wchodzą:

• Wysokowydajny C# (High-Performance C# - HPC#) - podzbiór języka C# umożliwiający zastosowanie wielu zabiegów optymalizacyjnych na poziomie kompilacji,

dzięki czemu można uzyskać wydajność zbliżoną do bardzo wydajnego kodu napisanego w C++, szerzej omówiony w artykule [48].

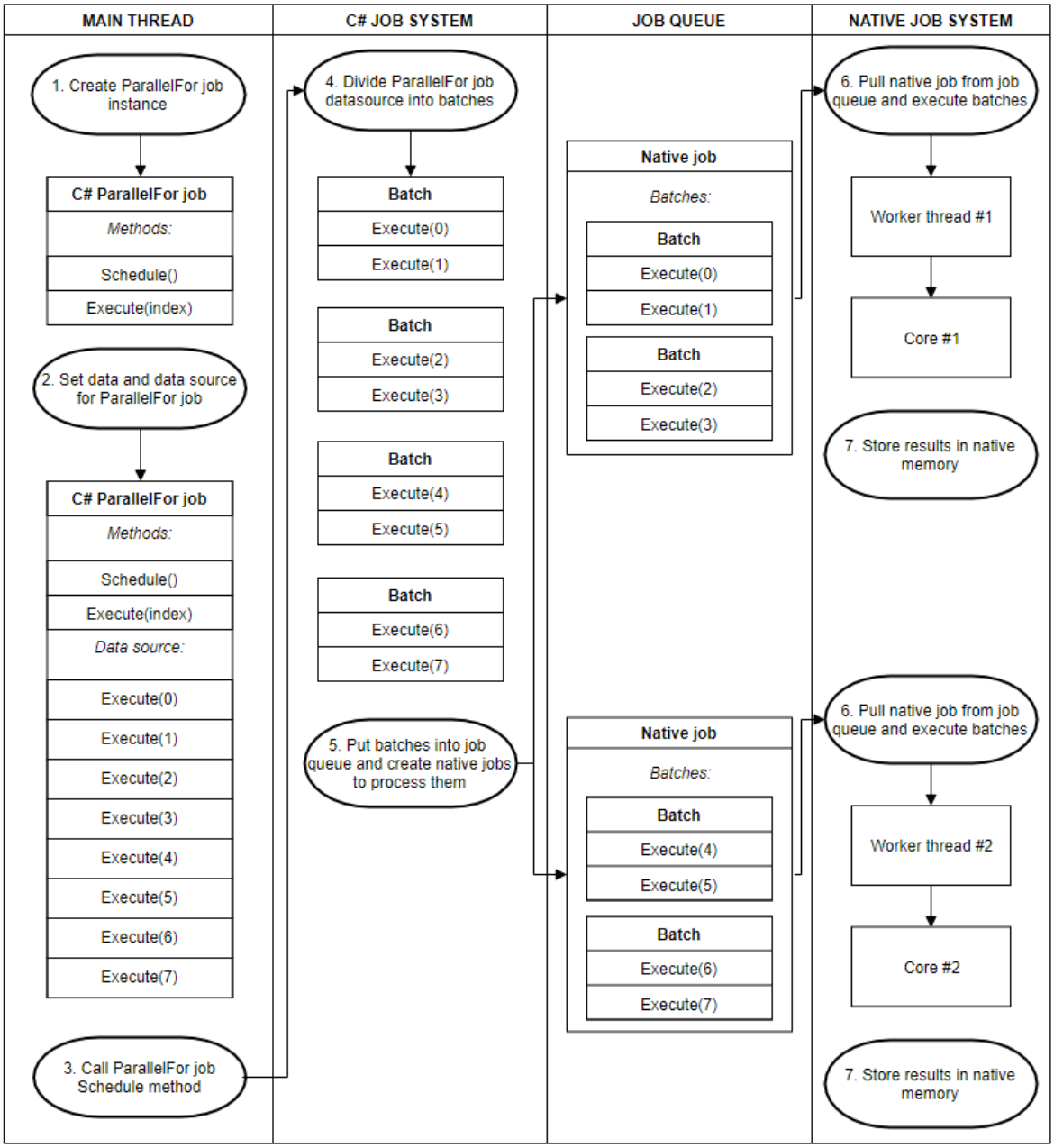
• Kompilator Burst (Burst Compiler) - kompilator dla HPC# pozwalający na uzyskanie bardzo wydajnego kodu maszynowego (wraz z wykorzystaniem możliwości użycia instrukcji wektorowych). W najpopularniejszym porównaniu widać wyraźną przewagę nad MonoJIT i wyraźne zbliżenie do C++ [47].

• System zadań C# (C# Job System) - system zarządzania asynchronicznymi i wielowątkowymi zadaniami. Dzięki wykorzystaniu HPC# pozwala zapewnić wydajne i bezpieczne środowisko dla wielowątkowych obliczeń. Bezpieczne rozumiane jako pozbawione wyścigu zależności (race condition) i zakleszczeń (deadlock). Dodatkowo nie występują tutaj drogie obliczeniowo mechanizmy obronne takie jak semafora.

• Entitas - szkielet aplikacji (framework) wykorzystujący wzorzec Entity component system wraz z systemami pomocowymi (system wizualizacji, system konwersji, system autoryzacji [authoring]). Technologia ta polega na wymienionych wcześniej technologiach.

4.3.3. C# Job System

System zadań C# zapewnia bezpieczną wielowątkowość. Mechanizm jego działania jest prosty w swoich założeniach jednak skomplikowany w implementacji. Obiekty wchodzące w skład silnika Unity są jednowątkowe (zmiana lub dostęp do ich parametrów z innych wątków powoduje wyrzucenie wyjątku), obecnie rozwój technologii podąża w kierunku wieloprocesorowych i wielowątkowych jednostek obliczeniowych, natomiast rozwój mocy pojedynczego rdzenia obliczeniowego jest spowolniony. Dlatego pojawiła się potrzeba umożliwienia łatwego wykonywania obliczeń wielowątkowych (zarówno po stronie logiki gry [gameplay] jak i po stronie komunikacji z kartą graficzną [DirectX12 i Vulcan], pozwalającą na przesyłanie komend do karty graficznej w wielowątkowej formie, aby lepiej wykorzystać dostępne zasoby). Jednak główno-wątkowa stara architektura nie pozwalała na łatwe wykorzystanie obliczeń wielowątkowych. Aby obejść ten problem zdecydowano się na system zadań. Zasadę działania na przykładzie zadania typu ParallelFor prezentuje Rysunek 4.1. Z głównego wątku aplikacji rejestrowane są zasoby i zadania. Następnie w odpowiednim momencie (lub momentach), zostaje wydane polecenie wykonania zarejestrowanych zadań, zadania wykonywane są na osobnych wątkach (z wykorzystaniem możliwości zrównoleglenia), natomiast główny wątek oczekuje na wykonanie zadań, po wykonaniu zadań wznawia on swoją egzekucję instrukcji.



Rysunek 4.1 Zadanie ParallelFor dzieli paczki pomiędzy rdzenie
z https://docs.unity3d.com/Manual/JobSystemParallelForJobs.html. Dostęp 21.10.2020.

4.3.4. Entitas

4.3.4.1 Wprowadzenie:

W tej implementacji wzorca ECS wykorzystano podejście oparte na archetypach. Dodatkowo dane pojedynczego archetypu podlegają fragmentacji, dla każdego archetypu rezerwowany jest fragment pamięci (chunk). Kiedy pamięć ta jest całkowicie wykorzystana, powstaje nowy fragment dla danego archetypu (jest to uproszczenie, ponieważ istnieją jeszcze inne czynniki, przez które dwa byty tego samego archetypu mogą znaleźć się w innych fragmentach, mimo że w każdym z tych fragmentów jest wystarczająca ilość miejsca, aby pomieścić obydwa byty). Zbiór bytów, komponentów i systemów jest przechowywany w świecie (World) i zarządzany przez menedżer bytów (EntityManager). Można tworzyć wiele niezależnych światów, jak i powodować interakcję pomiędzy nimi (przenosząc lub kopiując byty z jednego świata do drugiego). Dzięki

rozbudowanemu systemowi zapytań (EntityQuery), mamy dostęp do odpowiednio przefiltrowanych bytów i połączonych z nimi komponentów. Menedżer bytów pozwala na wprowadzanie natychmiastowych zmian strukturalnych, w jednowątkowej formie, natomiast bufory komend (EntityCommandBuffer) pozwalają na wprowadzanie odroczonych zmian w wielowątkowej formie (tzn. z różnych wątków można rejestrować zmiany strukturalne, które później zostaną odtworzone i zaaplikowane naraz).

4.3.4.2 Komponenty:

Każdy komponent jest unikatowy w obrębie bytu. Komponenty występują w kilku rodzajach, zdefiniowane są przez typ interfejsu, który implementują:

• IComponentData - struktura (struct) posiadająca tylko pola typu Blittable (czyli takie, gdzie reprezentacja w pamięci jest taka sama w wersji zarządzanej [managed code] i nie zarządzanej [unmanaged code]). Jest to podstawowa jednostka danych.

• ISharedComponentData - struktura (struct), która może posiadać pola inne niż Blittable (w tym referencje do obiektów), takie komponenty wyznaczają klasę (w pojęciu matematycznym, nie OOP) bytów wraz z archetypem. W jednym fragmencie archetypu (chunk) trzymana jest jedna instancja typu ISharedComponentData (tzn. porównywanie odbywa się na poziomie wartości komponentu, nie adresu w pamięci [value equality]). Wprowadzenie wielu instancji i/lub wielu komponentów tego rodzaju prowadzi do granulacji pamięci i marnotrawstwa pamięci używanej per fragment.

• Dynamic buffer (dynamiczny bufor) - implementacja dynamicznej listy składającej się z instancji struktury implementującej IBufferElementData (z ograniczeniami tak jak IComponentData). Służy do ominięcia limitu jednej instancji komponentu danego typu per byt, przykładem użycia mogą być koordynaty drogi wyznaczonej przez system szukania drogi (pathfinding).

• System state components (komponenty stanu systemu) - w skład tych komponentów wchodzą komponenty implementujące ISystemStateComponentData, ISystemStateSharedComponentData lub ISystemStateBufferElementData. Komponenty te są podobne do IComponentData, ISharedComponentData lub IBufferElementData, ale nie są automatycznie usuwane przy niszczeniu bytu. Służą do wykrywania zmian w strukturze (dodanie i usunięcie bytu/komponentu) i do śledzenia zaalokowanych zasobów przez dany system, dla danego bytu.

• Chunk component (komponent fragmentu) - połącznie IComponentData i ISharedComponentData. Jest to instancja IComponentData, która jest przechowywana na poziomie fragmentu (per fragment), a nie na poziomie bytu, czyli tak jak ISharedComponentData. W odróżnieniu od ISharedComponentData, nie jest zarządzana automatycznie (automatyczne przeniesienie bytu przy zmianie wartości), lecz jest zarządzany dla całego fragmentu przez programistę.

4.3.4.3 Systemy:

System jest instancją klasy dziedziczącej po SystemBase. Domyślnie system jest instancjonowany w domyślnym świecie, wykonując się w fazie logiki biznesowej, w deterministycznym, ale losowym jej miejscu. Można sterować tym zachowaniem dzięki atrybutom, tzn. można wyłączyć automatyczne tworzenie instancji systemu, można zmienić, w której fazie klatki wykonywany jest system, jak i określić zależności logiczne względem innych systemów (SystemA ma wykonać się przed wykonaniem SystemB). Systemy mogą implementować filtrowanie i przetwarzanie na parę sposobów:

• Domyślnym sposobem jest implementacja składni Entities.ForEach, która pozwala przekazać wyrażenie lambda, które zostanie wywołane dla każdego bytu spełniającego filtry. Wyrażenia lambda są kompilowane do odpowiednich struktur, więc nie trzeba płacić kosztu jaki wiąże się z używaniem składni lambda w języku C#. Taka

implementacja może być wykonywana od razu lub zwracać zadanie C# (C# job), które zostanie obsłużone przez system zadań (job system).

• Przy użyciu własnych zadań (IJobChunk i/lub IJobEntityBatch) możemy realizować systemy (lub części systemów) w bardziej optymalny dla implementowanej logiki sposób.

Manualna iteracja może być realizowana na głównym wątku za pomocą filtrów i natychmiastowego dostępu do bytów (entity) lub fragmentów (chunk), jak i za pomocą zadania IJobParallelFor, gdzie przesyłamy 'ręcznie' wyselekcjonowane tablice komponentów i bytów.

4.3.5. C#

Jako, że praca realizowana jest w silniku Unity, to oczywistym językiem użytym do implementacji, staje się język C#, wieloparadygmowy i wieloplatformowy język ogólnego przeznaczenia, stworzony i rozwijany przez firmę Microsoft.

4.3.6. xNode

Sposobem wizualizacji do narzędzia będącego przedmiotem pracy są węzły (node), wizualizacja taka jest standardem w branży gier komputerowych (np. Unity ShaderGraph, Bolt lub Playmaker). Stworzenie systemu wyświetlania i interakcji z taką wizualizacją nie jest tematem pracy, więc autor wybrał taki, który ma otwarte źródła [49] i z którym autor pracował.

4.3.7. ClickUp

Do zarządzania i monitorowania postępów pracy zastosowano aplikację ClickUp [50]. Jest to darmowe narzędzie (z ograniczeniami), które służy do współpracy i zarządzania projektami. Narzędzie te zostało wybrane ze względu na ogrom funkcjonalności i darmowy dostęp.

4.3.8. Visual Studio Enterprise 2019

Jest to jedno z wiodących zintegrowanych środowisk programistycznych (IDE), posiadające dodatek pozwalający współpracować efektywnie z edytorem Unity. Mimo popularności wymaga zainstalowania dodatkowych paczek, aby umożliwić w pełni wydajną pracę (np. Viasfora [51], Auto Save File [52], Add New File [53] czy Code Maid [54]) po zainstalowaniu i skonfigurowaniu środowiska i paczek, praca z tym narzędziem jest bardzo zbliżona do doświadczeń jakie można uzyskać przy wykorzystaniu narzędzia Rider [55]. Visual Studio zostało wybrane ze względu na znajomość tego narzędzia i brak wsparcia w narzędziu Rider do wielu różnych konfiguracji.

4.3.9. Git i GitHub

Jako system wersjonowania został wybrany system Git, który jest standardem w branży IT. Jako zewnętrzne repozytorium wybrano platformę GitHub, wybór ten jest podyktowany znajomością platformy przez autora i jej pozycji lidera w swojej kategorii, co zapewnia duże wsparcie i bezpieczeństwo.

4.4. Testy

Dla uproszczenia w niniejszym podrozdziale, jeżeli nie jest napisane inaczej, testami nazywane są automatyczne testy jednostkowe. Testy integracyjne w omawianym projekcie odnosiłyby się jedynie do zapisu plików z wygenerowanym kodem w strukturze projektu, więc zostały one przeprowadzone manualnie i z pragmatycznego punku widzenia są zupełnie wystarczające. Testy systemowe ze względu na specyfikę projektu są niemożliwe do przeprowadzenia, ponieważ jeżeli za środowisko docelowe –

produkcyjne zostanie uznane wykorzystanie narzędzia w komercyjnym projekcie gry komputerowej, to niemal niemożliwe staje się zebranie danych z takiego procesu, ponieważ produkcja gry komputerowej trwa nawet kilka lat. Natomiast jeżeli środowiskiem tym określimy uruchomienie narzędzia w silniku Unity, to było ono manualnie testowane przez cały proces wytwarzania artefaktu implementacyjnego. Ostatecznie testy interfejsu, również zostały przeprowadzone manualnie, w świecie gier komputerowych i silnika Unity. Same testy jednostkowe są stosunkowo świeżym konceptem i miało popularnym, istnieją nieśmiałe projekty [56] automatyzacji testów interfejsu użytkownika, jednak są to narzędzia dostępne podczas trybu gry (play mode), natomiast wytwarzane oprogramowanie jest typu tryb edycji (edit mode). W związku z powyższym manualne testy interfejsu użytkownika wydały się najlepszym rozwiązaniem, zwłaszcza z ograniczonymi ramami czasowymi projektu.

Proces implementacji projektu nie był prowadzany w zgodzie z metodyką TDD (rozwój sterowany testami). Co więcej znaczna część napisanego kodu jest nietestowalna automatycznie, ponieważ opiera się na za zewnętrznym kodzie, który także nie wspiera i nie dostarcza żadnych testów. Nie oznacza to jednak, że w takich projektach nie powinno się przeprowadzać żadnych testów. Testy zapewniają żywą (w przeciwieństwie do komentarzy w kodzie) dokumentację i pewien stopień pewności przy refaktoryzacji kodu. Biorą to pod uwagę w miejscach w kodzie, gdzie z pragmatycznego punktu widzenia czas wykorzystany na implementację testów był lepszym wykorzystaniem tego zasobu, niż przeznaczenie go na kolejne funkcjonalności, wprowadzono testy.

Odwracając piramidę testów, testy interfejsu użytkownika polegały na manualnym testowaniu produktu na trzech konfiguracjach rozdzielczości monitorów (1920x1080, 2560x1440 i 3440x1440) i konfiguracjach okna.

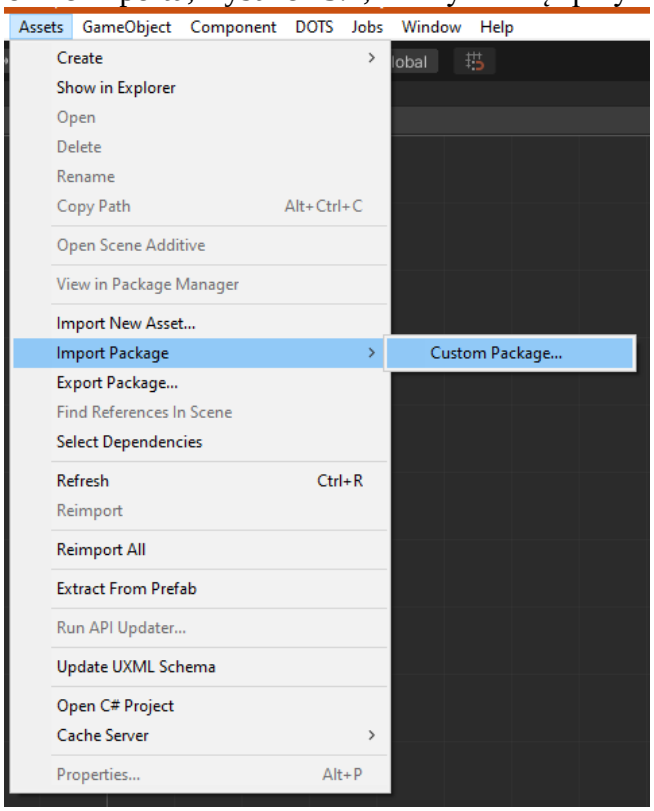
Testy systemowe i integracyjnie zostały przedstawione na początku podrozdziału.

Testy jednostkowe zostały zaimplementowane dla wybranych części systemów, w których najciężej było przeprowadzić testy manualne i/lub nie niosło to znacznych trudności implementacyjnych. Te części systemu to ProcessorsSelector omawiane w podrozdziale 6.6 Generacja kodu i wybrane klasy typu utility (pomocnicze, użytecznościowe).

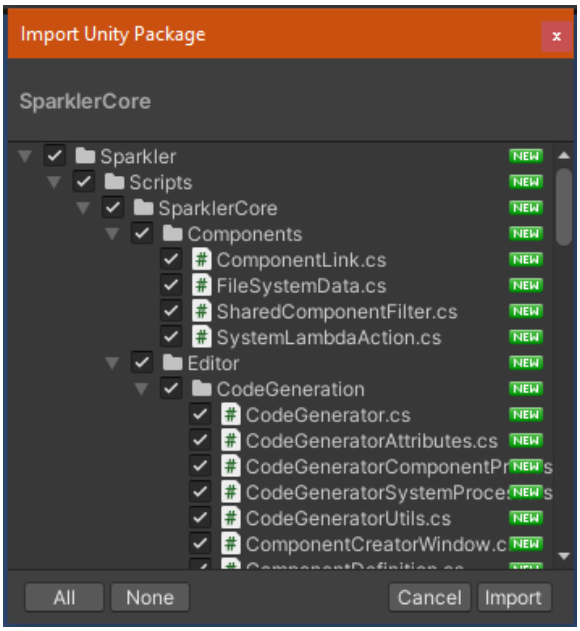
5. Prezentacja projektu

5.1. Instalacja

Aby zainstalować omawiany projekt należy sklonować repozytorium git z platformy GitHub [57]. Wśród pobranych danych znajdować się będzie folder nazwany ExportedPackages, a w nim paczka Unity o nazwie SparklerCore.unitypackage. Jest to stary system paczek Unity, tak jak wspomniano, w możliwościach rozwoju projektu jedną z możliwości jest użycie nowego systemu paczek i narzędzia PackageManager, aby ułatwić cały proces. Po zlokalizowaniu odpowiedniej paczki należy zapisać jej ścieżkę. W projekcie Unity, do którego instalowana jest paczka, należy przejść do opcji menu Assets/Import Package/Custom Package…, tak jak na Rysunek 5.1, w oknie wyboru paczki należy przejść pod zapisaną ścieżkę i wskazać paczkę, o której wspomniano powyżej. Następnie w oknie importu, Rysunek 5.2, należy kliknąć przycisk Import.



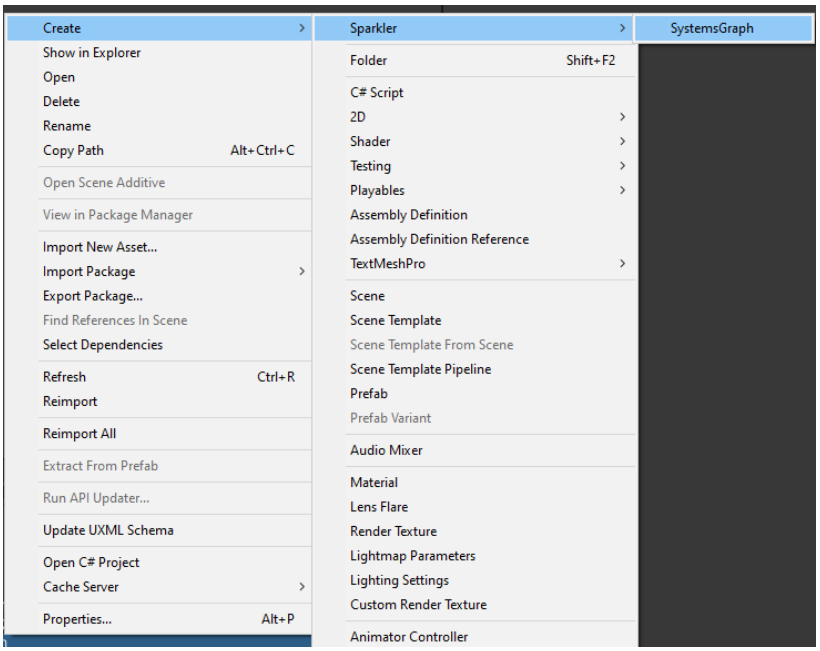
Rysunek 5.1 Menu importu paczek w silniku Unity



Rysunek 5.2 Okno importu paczki SparklerCore.unitypackage

5.2. Użycie

Aby rozpocząć pracę z narzędziem, należy w wybranym folderze utworzyć plik z kanwą do definiowania stanów - SystemGraph, poprzez kliknięcie prawym przyciskiem myszki i wybranie menu Create/Sparkler/SystemsGraph, tak jak wskazuje to Rysunek 5.3, i nadanie nazwy nowemu zasobowi. Po utworzeniu nowego zasobu typu SystemsGraph, należy otworzyć edytor poprzez dwukrotne kliknięcie na stworzonym zasobie. Otworzona zostanie kanwa edytora plików SystemGraph przedstawiona na Rysunek 5.4.

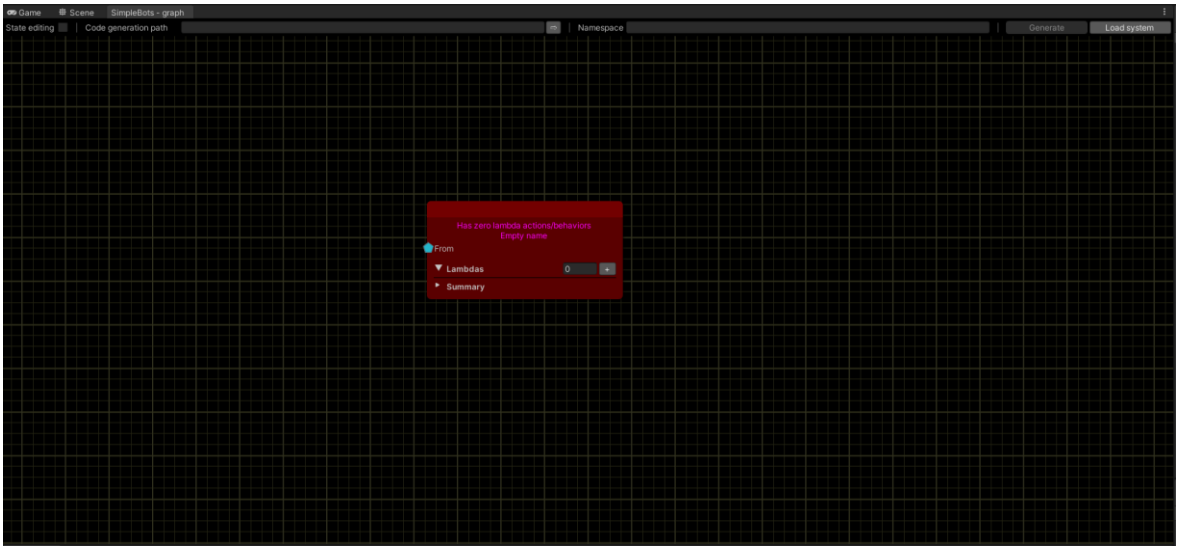


Rysunek 5.3 Menu tworzenia nowego zasobu typu SystemsGraph

Rysunek 5.4 Kanwa edytora nowego zasobu typu SystemsGraph

Na pasku narzędzi należy wskazać ścieżkę generacji kodu i podać przestrzeń nazw, której mają używać zadeklarowane tutaj elementy.

W celu utworzenia węzła z definicją nowego systemu, należy kliknąć prawym przyciskiem myszki w kanwie, węzeł zostanie utworzony natychmiastowo, jeżeli jest to jedyny zadeklarowany typ węzła. Narzędzie posiada tylko predefiniowany węzeł typu SystemNode, natomiast w razie potrzeby można definiować inne typy węzłów poprzez dziedziczenie z typu SystemNode. Po dodaniu węzła pojawi on się na kanwie, w miejscu kliknięcia tak jak pokazuje to Rysunek 5.5.



Rysunek 5.5 Kanwa edytora z nowopowstałym węzłem

Teraz pozostaje już tylko wypełnienie kanwy węzłami oraz ich ustawienie. Po tym należy wygenerować kod i uzupełnić go o brakującą logikę systemów.

5.3. Przykłady

W folderze ExportedPackages znajduje się także paczka Unity z przykładami wykorzystania narzędzia SparklerExamples.unitypackage. Proces importu jest analogiczny

do procesu importu głównego narzędzia. Po zaimportowaniu, w hierarchii projektu pojawi się folder z przykładami. Każdy przykład składa się z pliku SystemsGraph i plików z wygenerowanym i uzupełnionym kodem C#, w taki sposób aby spełniał on założenia przykładu. Tak przygotowany przykład nadaje się do samodzielnego uruchomienia i modyfikowania w celach edukacyjnych i/lub jako podstawa do rozpoczęcia własnego projektu.

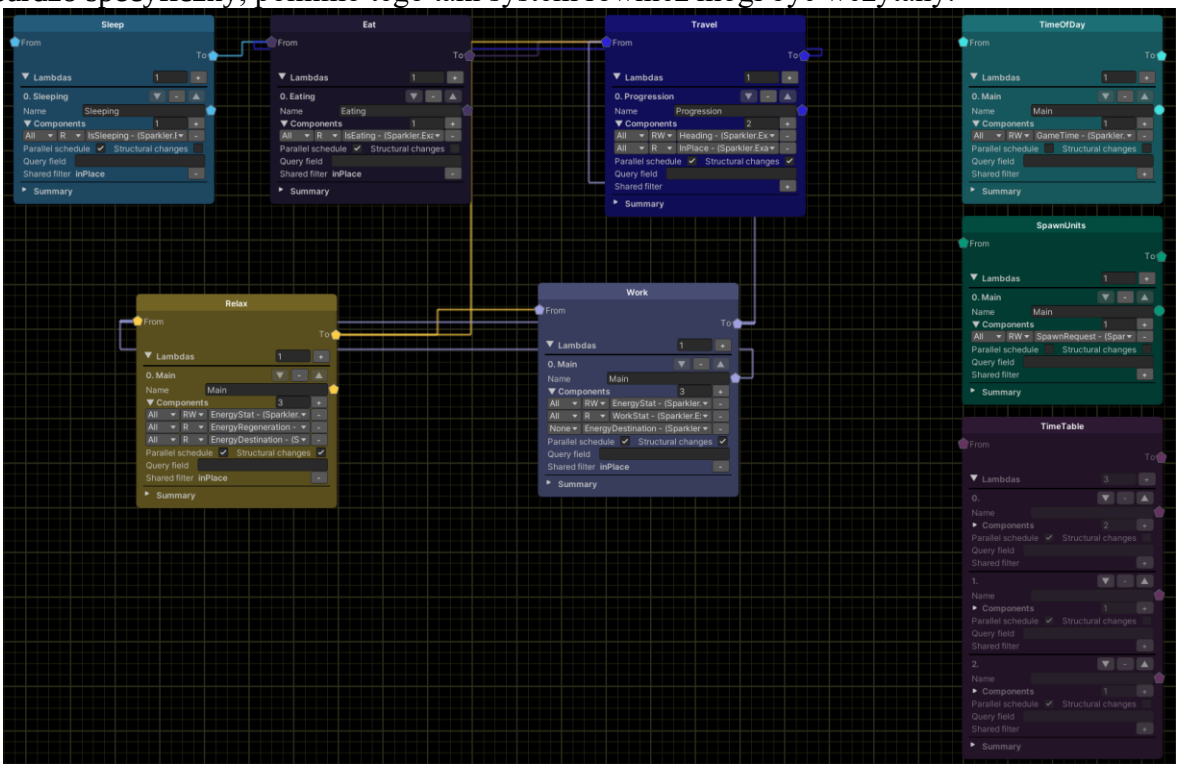
5.3.1. Przykład 1

Pierwszy przykład zawiera węzły prezentujące różne możliwości ustawień lambd w systemach. Kod w tym przykładzie jest nie zmieniony i posiada tylko wygenerowane dane przez narzędzie. Jest to najprostsza prezentacja tego co można ustawić w edytorze węzłów oraz kreatorze komponentów.

5.3.2. Przykład 2

Przykład drugi obrazuje uproszczoną symulację magicznej wioski krasnoludów na specjalnej diecie. Każdy krasnolud je w domu posiłek po przebudzeniu się, następnie udaje się na swoje miejsce pracy. Tam wykonuje pracę lub odpoczywa, jeśli nie ma siły na pracę. Po pracy wraca do swojego domu, je drugi posiłek i odpoczywa, aż do pory snu. Sen i praca są dyktowane przez czas.

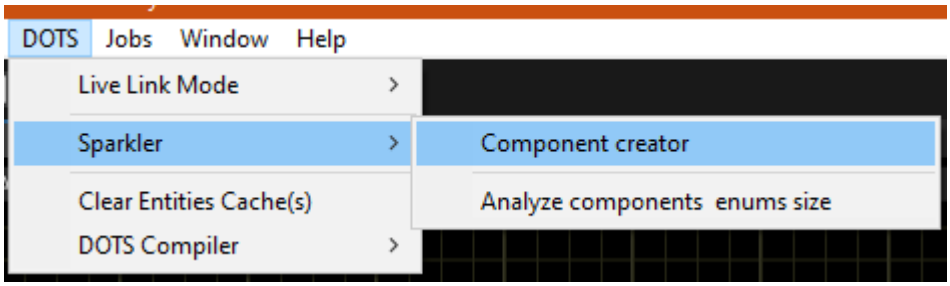
W tym przykładzie zasób grafu węzłów zawiera projekt systemu, przedstawiony na Rysunek 5.6, i na jego podstawie został wygenerowany kod, który następnie został uzupełniony o potrzebną logikę, aby wykonywać zaplanowaną rutynę. Dodatkowo powstał system TimeTableSystem, który został od podstaw wykonany w kodzie, ponieważ jest on bardzo specyficzny, pomimo tego taki system również mógł być wczytany.



Rysunek 5.6 Kanwa wykorzystana do zaprojektowania przykładu 2

5.4. Inne opcje

W menu DOTS/Sparkler/, przedstawionym na Rysunek 5.7, można znaleźć przyciski otwierające dwa dodatkowe, niezależne narzędzia stworzone podczas realizacji projektu aby łatwiej realizować funkcjonalności głównej części projektu. Narzędzie te są omawiane w późniejszych podrozdziałach 6.7 Kreator komponentów i 6.8 Wczytywanie systemu.



Rysunek 5.7 Menu z dwoma dodatkowymi narzędziami instalowanymi wraz z paczką

6. Implementacja

6.1. Zmiany w xNode

Aby zwiększyć komfort pracy z Xnode, wymagane było wprowadzenie szeregu usprawnień:

- Rozróżnienie rodzaju portów (jedno i wielo-połączeniowe)
- Zaznaczanie i operowanie na połączeniach
- Obsługa paska narzędzi
- Obsługa systemu, który wyświetla menu z odpowiednio przefiltrowanymi rodzajami węzłów, kiedy upuszczony zostanie koniec połączenia na wolnym polu (lub automatycznym utworzeniu i połączeniu węzła, jeżeli jedynie jeden węzeł spełnia kryteria filtrowania)
- Ograniczenia wielu typów w połączeniach
- Naprawa oddalenia i zbliżania widoku

Aby zapewnić te usprawnienia wymagane było przestudiowanie całego kodu, kod ten posiada przyjazny interfejs publiczny, jednak zmiany te wymagały ingerencji w niepublicznej siatce połączonych metod, klas i pól (często statycznych). Dodatkowym utrudnieniem był brak testów, co wymuszało manualne testowanie wprowadzanych zmian, a to z kolei wprowadziło mniejszą pewność poprawności wprowadzanych zmian.

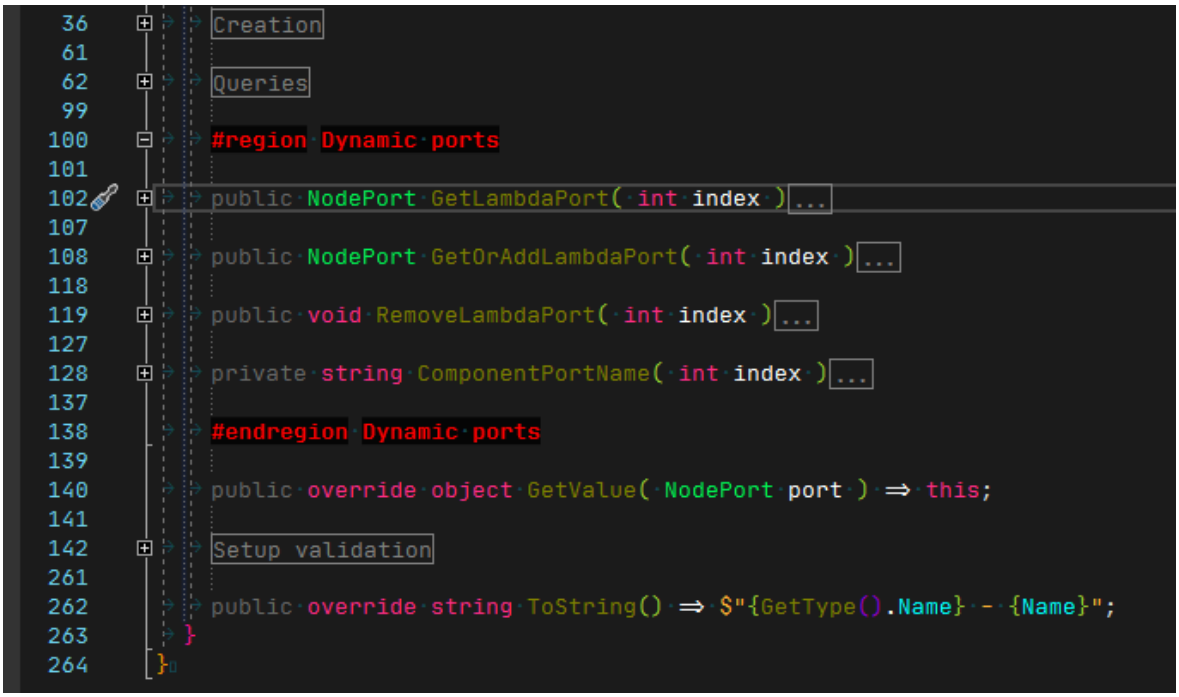
6.2. Strukturyzacja plików z kodem

Narzędzie Visual Studio i język C# umożliwiają organizację kodu w znacznych plikach poprzez stosowanie dyrektyw region tak zaprezentowano na Listing 6.1.

```
#region Nazwa

public void Method1(){
    ...
}

...

public void MethodN(){
    ...
}

#endregion (opcjonalnie, wedle konwencji) Nazwa
```

Listing 6.1 Zastosowanie dyrektywy #region

Takie obszary kodu można zwijać i rozwijać, dzięki czemu nawigacja po dużych plikach z kodem staje się bardziej intuicyjna i szybsza, jednak wyzwaniem jest dobór odpowiednich nazw, ziarnistości i kolejności występowania regionów. Autor stosował arbitralny dobór wymienionych parametrów, tzn. w momencie, kiedy subiektywnie zauważył, iż zastosowanie tej dyrektywy przyniesie korzyści, to ją stosował. Rysunek 6.1 prezentuje przykładowe użycie tego mechanizmu w projekcie.

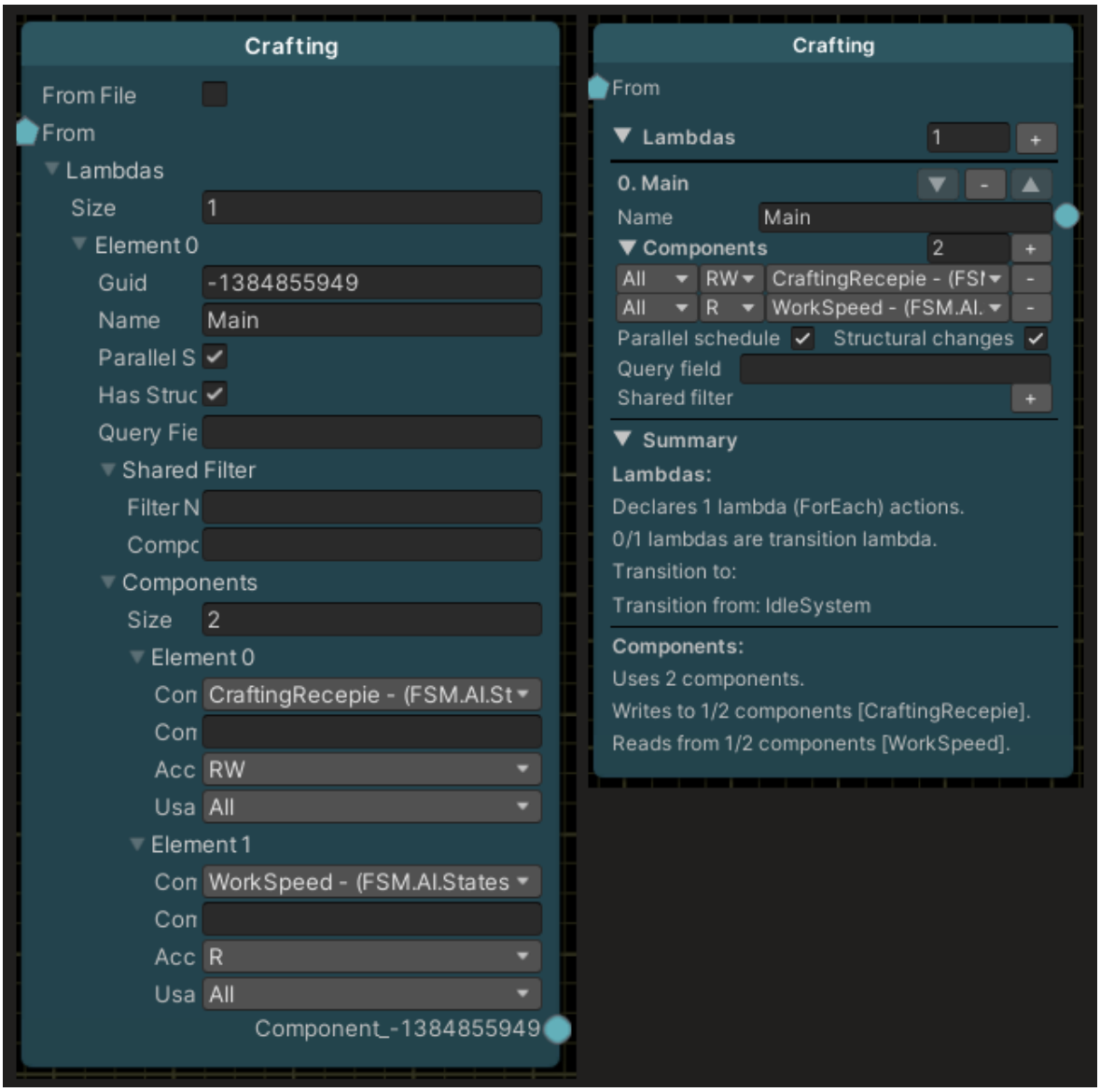


Rysunek 6.1 Zwinięte i rozwinięte dyrektywy region

6.3. Węzły, dane węzłów i wizualizacja

6.3.1. Dostosowane rysowanie:

Aby zapewnić jak najlepsze doświadczenia związane z obcowaniem z narzędziem należało napisać własne tzw. drawer'y i edytor'y, czyli skrypty odpowiedzialne za własną implementację wizualizacji. Było to potrzebne dlatego, że rozwiązanie proponowane przez Unity jest skrajnie nieprzyjemne i nieczytelne. To jak nieczytelnie wygląda domyślne rysowanie zaimplementowane przez Unity, można zaobserwować na Rysunek 6.2. Jednak kod do rysowania w edytorze nie jest ani łatwy, ani przyjemny do pisania. Wymaga wiele doświadczenia i znakomitej znajomości dostępnych funkcjonalności (często skrytych w nieopisanych częściach dokumentacji). Dodatkowo wykorzystanie refleksji [58] i IMGUI [59] [60] nie pozwala na pisanie kodu zgodnego z zasadami programowania obiektowego lub zorientowanego na dane. Przykładowy fragment kodu pozwalający na własną implementację rysowania można zobaczyć na Listing 6.2.



Rysunek 6.2 Po lewej domyślne węzła przez Unity, po prawej rysowanie zaimplementowane w projekcie

```
1  public override void OnGUI(
     Rect position, SerializedProperty property, GUIContent label
     )
2   {
3      EditorGUI.BeginProperty( position, label, property );

4      var wasChanged = GUI.changed;
5      GUI.changed = false;

6      PropertyRect propertyRect = new PropertyRect(position);

7      // Draw name
8      var nameProp = property.FindPropertyRelative("_name");
9      EditorGUI.PropertyField( propertyRect.AllocateLine(), nameProp );

10     // Draw array
```

```
11      var componentsProp = property.FindPropertyRelative("_components");
12      s_cache.Clear();
13      GUIDrawers.DrawArray( ref propertyRect, componentsProp, s_cache );

14      // Draw additional settings
15      propertyRect.AllocateLine();
16      var parallelScheduling = property.FindPropertyRelative("_parallelS
chedule");
17      EditorGUI.LabelField( propertyRect.AllocateWidthFlat( 105 ), s_par
allelContent );
18      EditorGUI.PropertyField( propertyRect.AllocateWidthFlat( 25 ), par
allelScheduling, EmptyContent );

    ...
19  }
```

Listing 6.2 Fragment funkcji OnGUI w klasie SystemLambdaActionDrawer służące do rysowania instancji klasy SystemLambdaAction

6.3.2. Funkcjonalności:

Węzeł (node) może występować w dwóch stanach: do edycji i do projektowania, różniących się możliwościami i widokiem, tak jak przedstawiono na Rysunek 6.3. W każdym z tych stanów dostępny jest:

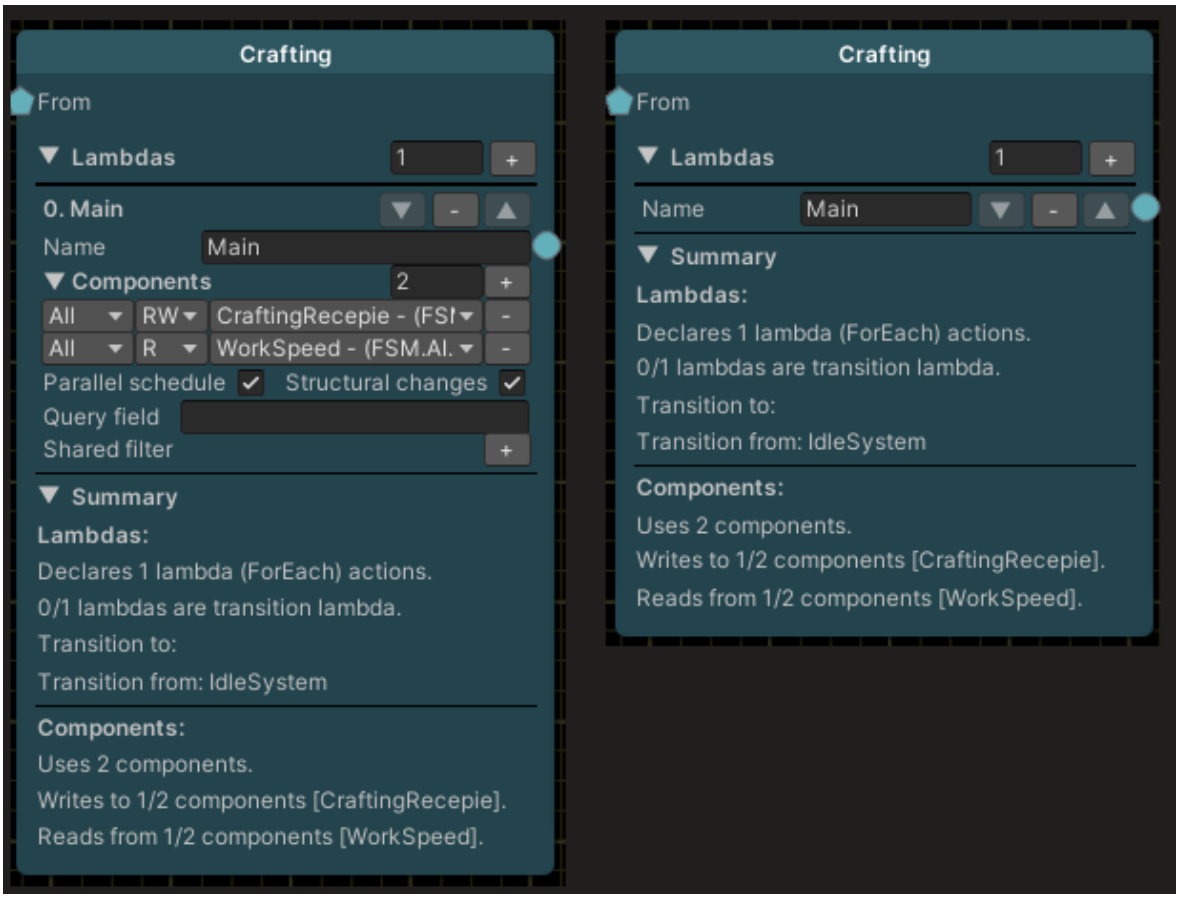
• Widok podsumowania - pozwalający uzyskać streszczenie i statystyki odnośnie systemu reprezentowanego przez węzeł, widok ten można ukryć.

• Widok lambd – akcji, które system może wykonywać, jest to lista, którą można zwijać i rozwijać, dodawać, usuwać i zmieniać kolejność elementów. Zastosowany tutaj własny edytor do list sprawia, że wszystkie te operacje są o wiele bardziej przystępne. Dodatkowo każda lambda posiada port, który pozwala połączyć ją z innym systemem, co sygnalizuje tranzycję (tzn. system wykona takie operacje na komponentach, aby spełniały one filtr przynajmniej jednej lambdy z systemu, do którego ma przejść, jednak nie musi oznaczać zaprzestania wykonywania lambdy, z której tranzycja prowadzi).

W widoku do edycji możliwe są jeszcze dodatkowe interakcje:

• Widok komponentów - lista komponentów, które definiują podstawowe filtry dla lambdy, umożliwia ona dodawanie i odejmowanie definicji filtru komponentu. Definicje te są automatycznie sortowanie, w razie potrzeby użycia nieistniejącego komponentu pojawia się przycisk automatycznie otwierający okno kreatora komponentów i po utworzeniu danego komponentu automatycznie zmienia go na otypowany komponent.

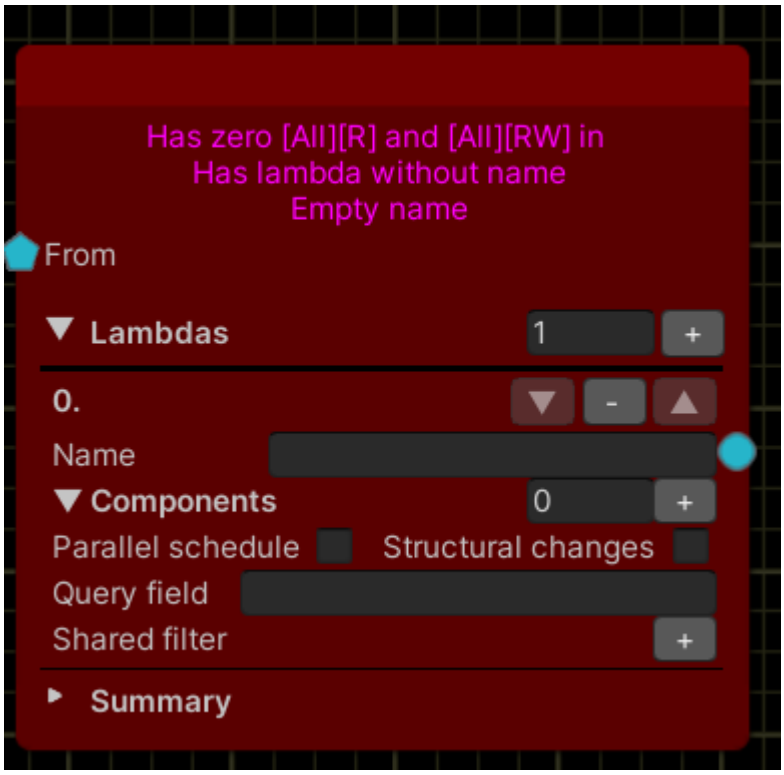
• Rozszerzony widok lambdy – określanie dodatkowych parametrów akcji lambda



Rysunek 6.3 Dwa możliwe widoki węzłów, po lewej: widok do edycji, po prawej: widok do projektowania

6.4. Walidacja ustawień systemów

Aby zapewnić jak najmniejszy próg wejścia, zwłaszcza dla nietechnicznych użytkowników, zdecydowano się poświęcić znaczną część czasu na system walidacji ustawień systemu. W momencie, kiedy znajduje się jakiś błąd w ustawieniu systemu, węzeł zmienia kolor na czerwony i wyświetla na górze komunikaty opisujące błędne ustawienia. Dla lepszej zauważalności, błędy są w kolorze magenta, przykładowe błędy widoczne są na Rysunek 6.4. Kolor ten jest powszechnie używany do sygnalizacji błędów w silniku Unity, jednak nie został wybrany do kolorowania węzła, ponieważ sprawiał, że ciężej było wykonywać operacje potrzebne do pozbycia się błędów.



Rysunek 6.4 Węzeł prezentujący 3 błędy z nim związane

6.5. Pasek narzędzi

Pasek narzędzi (toolbar) był konieczną modyfikacją w narzędziu xNode. Część implementacji można zobaczyć na Listing 6.3, natomiast końcowy efekt zaprezentowany jest na Rysunek 6.5. Taka implementacja umożliwi przełączanie między trybami edycji i projektowania poprzez zaznaczenie State editing, wskazanie ścieżki dla generowanych plików, przestrzeni nazw (namespace) dla generowanego kodu. Posiada również dwa przyciski:

• Generate - wywołujący akcję generacji kodu, kod jest generowany tylko dla węzłów, które są nowe, lub dla których użytkownik wyrazi taką wolę (jest to operacja destruktywna, usuwa poprzedni plik i tworzy całkowicie nowy). Przycisk ten staje się nieaktywny w momencie, kiedy którykolwiek z węzłów posiada błędne ustawiania.

• Load system - wywołuje akcję załadowania węzła z pliku z kodem systemu, załadowany system tworzy węzeł, który jest nieaktywny do edytowania (użytkownik może podglądać wczytane wartości, ale nie może żadnych edytować) i nie bierze udziału w generacji kodu. Kiedy wskazany plik zawiera system już załadowany, wyświetlony zostaje o tym komunikat i nie jest tworzony nowy węzeł.

Rysunek 6.5 Wygląd paska narzędzi

```
1  public override bool HasToolbar => true;
2  public override void OnToolbar()
3  {
4       EditorGUILayout.LabelField( s_stateEditigContent, GUILayout.Width(
 75 ) );
5       Target.StateEditing = EditorGUILayout.Toggle( Target.StateEditing,
 GUILayout.Width( 15 ) );
6       ToolbarSpace();
```

```
7       EditorGUILayout.LabelField( s_generateContent, GUILayout.ExpandWid
th( false ) );
8       Target.CodeGenerationPath = EditorGUILayout.TextField( Target.Code
GenerationPath, GUILayout.ExpandWidth( true ) );
9       if( GUILayout.Button( "\u27b1", GUILayout.ExpandWidth( false ) ) )
10      {
11          var dialogPath = EditorUtility.OpenFolderPanel( "Code director
y", "", "" );
12          if ( !string.IsNullOrWhiteSpace( dialogPath ) )
13          {
14              Target.CodeGenerationPath = PathExtension.AssetsPath( dial
ogPath );
15          }
16      }
17      ToolbarSpace();
18      EditorStyles.label.CalcMinMaxWidth( s_namespaceContent, out var mi
n, out var max );
19      EditorGUILayout.LabelField( s_namespaceContent, GUILayout.Width( m
in ) );
20      Target.Namespace = EditorGUILayout.TextField( Target.Namespace, GU
ILayout.ExpandWidth( true ) );
21      ToolbarSpace();
22      using ( new GUIEnabledScope( Target.StateEditing && Target.nodes.O
fType<SystemNode>().All( n => n.IsRightConfigured().Item1 ) ) )
23      {
24          if ( GUILayout.Button( "Generate", GUILayout.Width( 120 ) ) )
25          {
26              CodeGenerator.Generate( Target );
27          }
28      }
29      if ( GUILayout.Button( "Load system", GUILayout.Width( 120 ) ) )
30      {
31          LoadSystem();
32      }
33  }
34  private static void ToolbarSpace()
35  {
36      EditorGUILayout.LabelField( s_emptyContent, GUILayout.Width( 4 ) )
;
37      EditorGUILayout.LabelField( s_horizontalLine, GUILayout.Width( 4 )
 );
38      EditorGUILayout.LabelField( s_emptyContent, GUILayout.Width( 4 ) )
;
39  }
```

Listing 6.3 Fragment klasy SystemsGraphEditor odpowiedzialny za obsługę paska narzędzi

6.6. Generacja kodu

Generacja kodu jest jednym z podstawowych systemów w pracy. Aby zapewnić dużą rozszerzalność zgodnie z zasadą otwarte-zamknięte (open–closed principle) [15], zastosowano wzorzec podłączenia (plugin pattern) [15] [20]. Generacja kodu opiera

się na przetwarzaczach (processors), które implementują odpowiednie interfejsy. Użytkownik może zdefiniować własne przetwarzacze, które zostaną wykryte i dołączone do procesu generacji kodu dzięki klasie ProcessorsSelector (Listing 6.4). Aby określić kolejność przetwarzaczy, klasy mogą mieć zadeklarowane dwa atrybuty: ProcessAfter i ProcessBefore (Listing 6.5), które wskazują, że dana klasa ma być wykonana po i/lub przed inną klasą przetwarzacza.

```
1  // ICodeGeneratorSystemProcessorBeforeLambda
2  var beforeLambdaProcessors =
ProcessorsSelector.Selectors<ICodeGeneratorSystemProcessorBeforeLambda>()
;
3  foreach ( var processor in beforeLambdaProcessors )
4  {
5      template = processor.ProcessBeforeLambda( systemsGraph, system, te
mplate );
6  }
```

Listing 6.4 Przykładowe użycie klasy ProcessorsSelector

```
1  [ProcessAfter( typeof( ClassTest8_2 ) )]
2  private class ClassTest8_1 : ITest8 { }

3  [ProcessBefore( typeof( ClassTest8_1 ) )]
4  private class ClassTest8_2 : ITest8 { }

5  [ProcessAfter( typeof( ClassTest8_2 ) )]
6  private class ClassTest8_3 : ITest8 { }
```

Listing 6.5 Użycie atrybutów ProcessAfter i ProcessBefore w kodzie testów do klasy ProcessorsSelector

6.6.1. Generacja systemów:

Generacja systemów odbywa się przez przetwarzanie szablonu przygotowanego przez autora, który może być w pełni modyfikowalny według potrzeb projektu, w którym będzie wykorzystywany. Szablon jest przetwarzany przez silnik generacji kodu wykorzystując do tego wyrażenia regularne. Przebieg kodu i struktura tej części projektu jest zarazem najprostsza i najtrudniejsza. Jest to fragment projektu najmocniej związany z domeną biznesową i wymaga dobrej jej znajomości. Jest natomiast napisany w niemalże strukturalnym podejściu, co znacznie ułatwia zrozumienie przepływu i zrozumienie ogólnych konceptów. Przykładowy wygenerowany system znajduje się na Listing 6.6.

```
1  using FSM.AI.States.Components;
2  using Unity.Entities;
3  using Unity.Jobs;

4  namespace FSM.AI.States.Systems
5  {
6      public class CraftingSystem : SystemBase
7      {
8          private EndSimulationEntityCommandBufferSystem _endSimulationC
mdBuffer;

9          protected override void OnCreate()
```

```
10          {
11              base.OnCreate();
12              _endSimulationCmdBuffer = World.GetOrCreateSystem<EndSimul
ationEntityCommandBufferSystem>();
13          }

14          protected override void OnUpdate()
15          {
16              // Assign values to local variables captured in your job h
ere, so that it has everything it
17              // needs to do its work when it runs later. For example, f
loat deltaTime = Time.DeltaTime;
18              // This declares a new kind of job, which is a unit of wor
k to do. The job is declared as an
19              // Entities.ForEach with the target components as paramete
rs, meaning it will process all
20              // entities in the world that have both Translation and Ro
tation components. Change it to
21              // process the component types you want.
22              // -- CraftingSystem_Main
23              var mainCmdBuffer = _endSimulationCmdBuffer.CreateCommandB
uffer().AsParallelWriter();
24              Entities
25                  .WithName( "CraftingSystem_Main" )
26                  .ForEach( ( Entity entity, int entityInQueryIndex, ref
 CraftingRecipe craftingRecipe, in WorkSpeed workSpeed ) =>
27              {
28                  //TODO: Implement state behavior
29              } )
30              .ScheduleParallel();
31              // -- CraftingSystem_TransitionBuilding
32              var transitionbuildingCmdBuffer = _endSimulationCmdBuffer.
CreateCommandBuffer().AsParallelWriter();
33              PrimitiveTag primitiveTag = new PrimitiveTag{ IntVal = 2,
ByteVal = 0, ULongVal = 0, EnumVal = PrimitiveTag.Enum.Val1, FlagEnumVal
= (PrimitiveTag.FlagEnum)(-1), };
34              Entities
35                  .WithName( "CraftingSystem_TransitionBuilding" )
36                  .WithSharedComponentFilter( primitiveTag )
37                  .ForEach( ( Entity entity, int entityInQueryIndex, ref
 CraftingRecipe craftingRecipe, in WorkSpeed workSpeed ) =>
38              {
39                  //TODO: Make transition to one of the following state:
40                  //Building
41              } )
42              .ScheduleParallel();
43              _endSimulationCmdBuffer.AddJobHandleForProducer( this.Depe
ndency );
44          }
45      }
```

```
46 }
```

Listing 6.6 Przykładowy wygenerowany system, zawierający dwie funkcje lambda, gdzie jedna z nich oznaczona jest do realizacji tranzycji do odblokowania innego stanu i filtr na komponent współdzielony

6.6.2. Generacja komponentów:

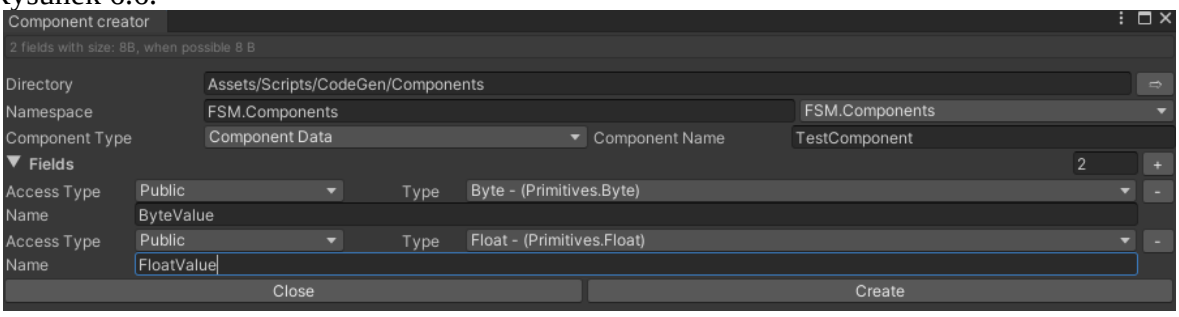
Generacja komponentów opiera się na tych samych narzędziach co generacja systemów, jednak wykorzystuje inne reguły domenowe i wymaga innego szablonu. Efekt wygenerowanego kodu można zobaczyć na Listing 6.7. Natomiast więcej o tworzeniu i projektowaniu komponentów znajdzie się w podrozdziale 6.7 Kreator komponentów6.7.

```
1  using System;

2  using Unity.Entities;

3  namespace FSM.AI.States.Components
4  {
5      [Serializable]
6      public struct WorkSpeed : IComponentData
7      {
8          public float Value;
9      }
10 }
```

Listing 6.7 Wygenerowany komponent

6.7. Kreator komponentów

Kreator komponentów jest oknem edytora służącym do projektowania i tworzenia komponentów w intuicyjny i szybki sposób. Pozwala to na stworzenie całego i zupełnego komponentu, jedynie przy wykorzystaniu edytora Unity. Utworzone tak komponenty są całkowicie edytowalne przy użyciu dowolnego edytora tekstowego, jako iż są to po prostu pliki z kodem C#. Okno pozwala na wybranie wyjściowego folderu, przestrzeni nazw (namespace), nazwy, typu komponentu i dostępnych pól, tak jak można zobaczyć na Rysunek 6.6.



Rysunek 6.6 Okno kreatora komponentów z przykładową definicją komponentu

Dla każdego pola należy wyspecyfikować nazwę, typ i rodzaj dostępu. Dużym problemem była różnica typów używanych w C#, a tym jak są one zdefiniowane w platformie .Net. W C# używa się aliasów dla typów prymitywnych, więc (zamiast System.Int32 używane jest int). Jednak definicja tego typu to definicja z .Net więc system refleksji wyszukuje i przedstawia ten typ jako rozbudowaną formę, a nie alias, co skutkuje nieprzestrzeganiem ogólnie przyjętych konwencji. Aby rozwiązać ten problem, stworzono interfejs IPrimitiveType, który definiuje metodę GetFieldDeclaration. Metoda ta przyjmuje nazwę pola i rodzaj dostępu, a zwraca poprawną deklarację pola. Fragment implementacji

widoczny jest na Listing 6.8, natomiast system generacji kodu dostał specjalną ścieżkę egzekucji dla typów dziedziczących po IPrimitiveType, Listing 6.9.

```
1  namespace Primitives
2  {
3      public interface IPrimitiveType
4      {
5          string GetFieldDeclaration( string fieldName, string accessMod
ifier );
6      }

7      public struct Float : IPrimitiveType
8      {
9          private float _backingField;

10         public string GetFieldDeclaration( string fieldName, string ac
cessModifier ) => $"\t\t{accessModifier} float {fieldName};";
11     }

12     public struct Double : IPrimitiveType
13     {
14         private double _backingField;

15         public string GetFieldDeclaration( string fieldName, string ac
cessModifier ) => $"\t\t{accessModifier} double {fieldName};";
16     }

17     public struct Int : IPrimitiveType
18     {
19         private int _backingField;

20         public string GetFieldDeclaration( string fieldName, string ac
cessModifier ) => $"\t\t{accessModifier} int {fieldName};";
21     }
...
22 }
```

Listing 6.8 Fragment pliku SimpleTypesNames.cs z implementacji IPrimitiveType

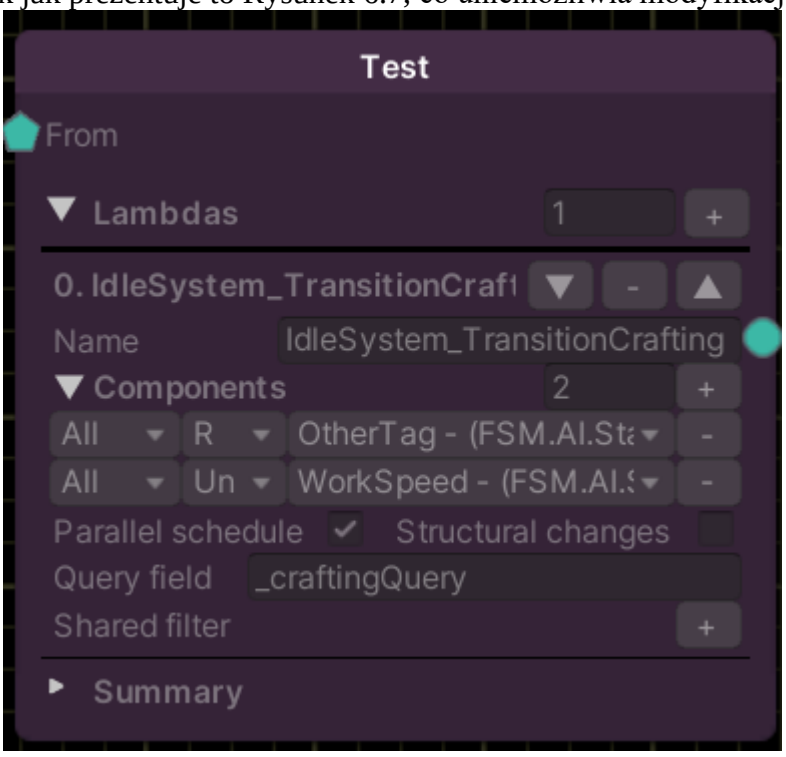
```
1  foreach ( var field in fields )
2  {
3      if ( field.type.Type != null )
4      {
5          if ( typeof( IPrimitiveType ).IsAssignableFrom( field.type.Typ
e ) )
6          {
7              IPrimitiveType primitiveType = Activator.CreateInstance(fi
eld.type.Type) as IPrimitiveType;
8              fieldsBuilder.AppendLine( primitiveType.GetFieldDeclaratio
n( field.name, FieldAccessType( field.accessType ) ) );
9          }
10         else
11         {
```

```
12              fieldsBuilder.AppendLine( $"\t\t{FieldAccessType( field.ac
cessType )} {field.type.Type.Name} {field.name};" );
13          }
14      }
15  }
```

Listing 6.9 Fragment generacji kodu wiedzący o IPrimitiveType

6.8. Wczytywanie systemu

Wczytywanie systemu, który istnieje jako plik z kodem C#, był najbardziej problematyczną częścią. Do implementacji wykorzystano wyrażenia regularne. Jednak autor nie był w stanie przetwarzać efektywnie wczytanego pliku, ze względu na wiele stylów programowania wczytywanego systemu. Aby rozwiązać ten problem zastosowano wstępne oczyszczanie wczytanych linii kodu z problematycznych znaków białych, uzyskując w efekcie zawsze tak samo sformatowany kod. Dzięki temu można było zbudować efektywne wyrażenia regularne do wykrywania odpowiednich fragmentu kodu. Jest to alternatywne podejście względem transformacji danych tekstowych kodu do drzewa składni i uzyskiwania potrzebnych danych z owego drzewa. Wybranie wyrażeń regularnych zamiast drzewa składni jest motywowane chęcią ograniczenia progu wejścia dla osób chcących rozwijać ten projekt. Dobra znajomość wyrażeń regularnych jest obligatoryjna, aby zrozumieć generację kodu, natomiast drzewo składni wymagałoby zrozumienia zupełnie innych algorytmów i struktur danych. Wczytany system w formie wizualnej nie podlega dalszej edycji. Jego komponenty wizualne są rysowane w formie zablokowanej, tak jak prezentuje to Rysunek 6.7, co uniemożliwia modyfikację ich.

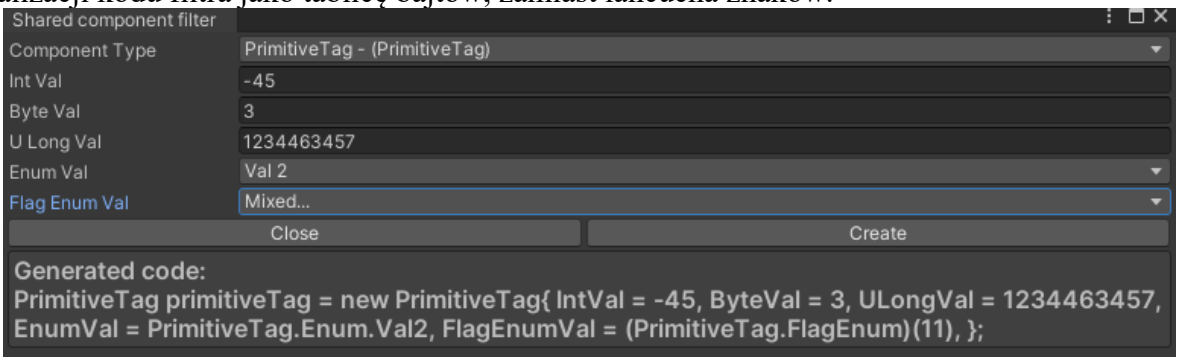


Rysunek 6.7 Wczytany system w formie wizualnej

6.9. Filtr komponentów współdzielonych

Filtrowanie z użyciem shared component (komponentów współdzielonych) jest jednym z mechanizmów filtrowania. Wymaga on przesłania instancji obiektu implementującego ISharedComponent, a system filtrujący byty i komponenty wybierze

tylko fragmenty (chunks), dla których zadanych filtr współdzielonego komponentu jest wartościowo równy (value equality) instancji tego komponentu w danym fragmencie. Oznacza to, że system generowania kodu musi wstawić kod komponentu do generowanego pliku, okno pokazuje kod, który będzie wygenerowany tak jak pokazuje to Rysunek 6.8. Autor rozważał dwie możliwości przechowywania takiego filtru. Pierwsza z nich to przechowywanie zserializowanej instancji komponentu. Drugi sposób to przechowywanie wygenerowanego kodu. Pierwsze rozwiązanie wymaga większej ilości kroków podczas generowania kodu systemu, natomiast drugie rozwiązanie wymaga większej ilości kroków podczas wprowadzania zmian w filtrze. Ze względu na rzadkie wprowadzanie zmian wybrano sposób drugi. Jednak doprowadziło to do problemów z serializacją dokonywaną przez Unity. Rozwiązaniem tego problemu było zastosowanie serializacji kodu filtra jako tablicę bajtów, zamiast łańcucha znaków.



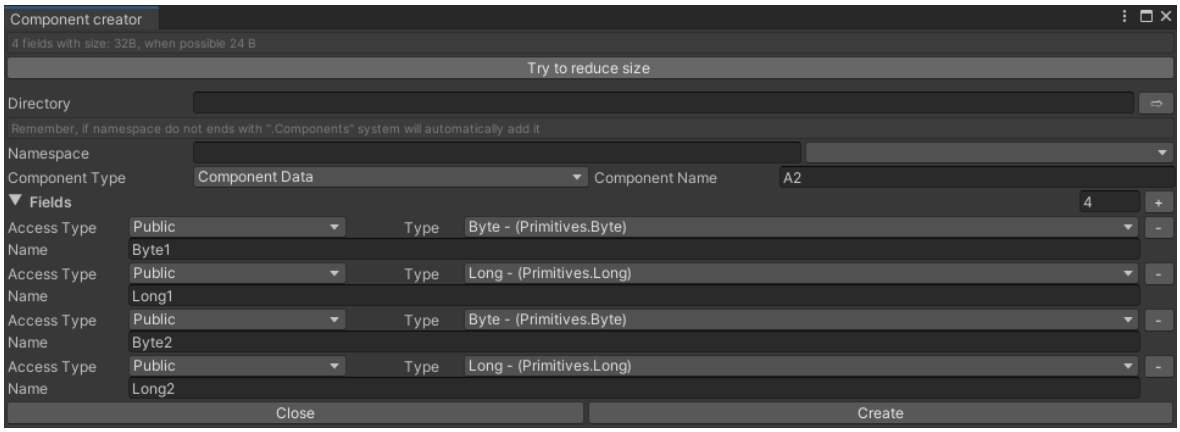
Rysunek 6.8 Okno tworzenia filtrów komponentów współdzielonych

6.10. Rozmiar komponentów

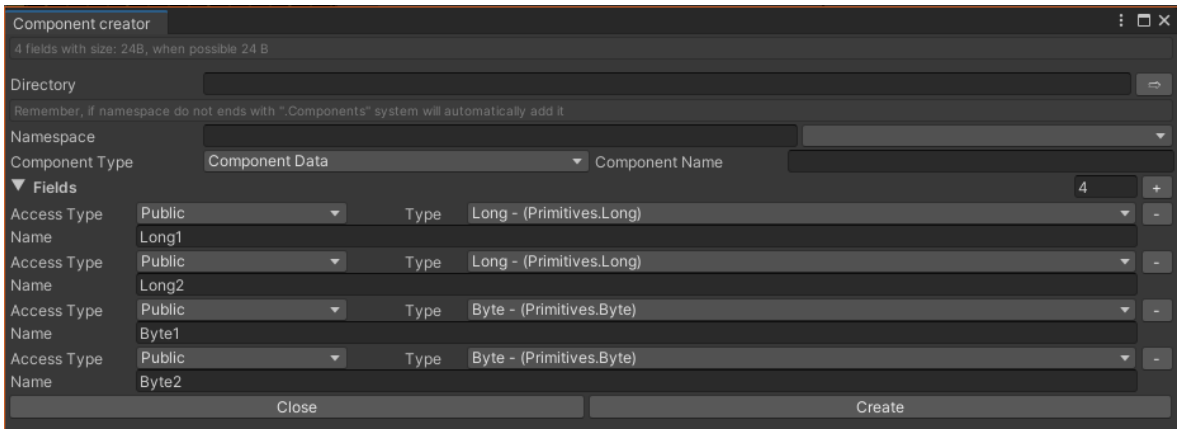
W języku C# podczas definiowania struktury, pola definiowane są w zadanej kolejności, która może wpływać na rozmiar instancji w pamięci. Aby lepiej zobrazować problem, najłatwiej ukazać to na przykładzie.

```
1  public struct A1{
2      public byte Byte1;
3      public long Long1;
4      public byte Byte2;
5      public long Long2;
6  }

7  public struct A2{
8      public long Long1;
9      public long Long2;
10     public byte Byte1;
11     public byte Byte2;
12 }
```

Listing 6.10 Kod dwóch struktur zawierające te same pola zadeklarowane w różniej kolejności



Rysunek 6.9 Definicja struktury A1 z Listing 6.10 w kreatorze komponentów, pokazujący, że ta struktura zajmuje 32 bajty, kiedy możliwe jest żeby zajmowała 24 bajty



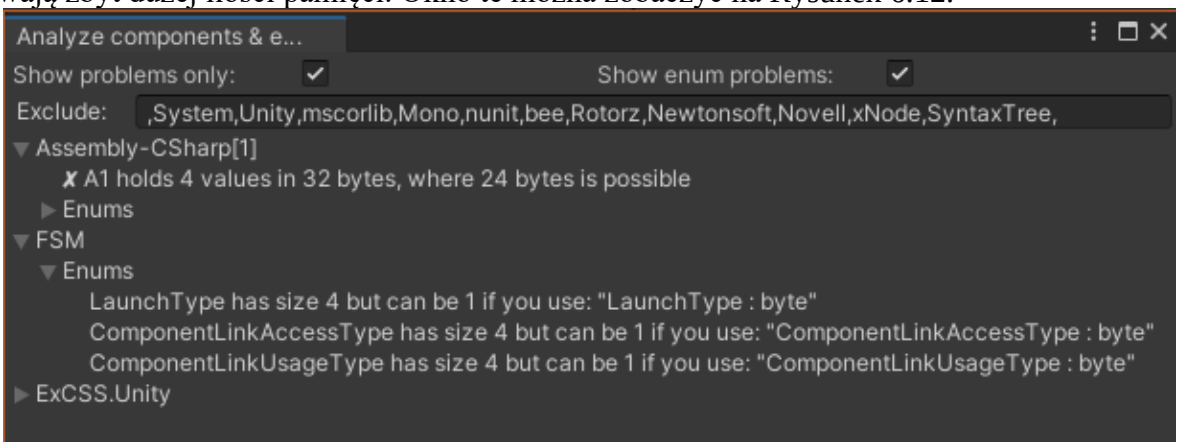
Rysunek 6.10 Definicja struktury A2 z Listing 6.10 w kreatorze komponentów, pokazujący, że ta struktura zajmuje 24 bajty

Jak widać na Listing 6.10, Rysunek 6.9 Definicja struktury A1 z Listing 6.10 w kreatorze komponentów, pokazujący, że ta struktura zajmuje 32 bajty, kiedy możliwe jest żeby zajmowała 24 bajty i Rysunek 6.10. dwie struktury realizujące te same funkcjonalności mogą różnić się rozmiarem o ponad 65%, a jedyne co do tego prowadzi to nieuważność i/lub brak wiedzy osoby definiującej daną strukturę. Dzieje się tak przez wyrównywanie pamięci. Zjawisko to jest przedstawione na Rysunek 6.11, więcej informacji na temat tego zjawiska można znaleźć w artykule [61].

A	B	C	D	E	F	G	H	I
A1	Byte1	*Wyrównanie*						
	Long1	Long1	Long1	Long1	Long1	Long1	Long1	Long1
	Byte2	*Wyrównanie*						
	Long2	Long2	Long2	Long2	Long2	Long2	Long2	Long2
A2	Long1	Long1	Long1	Long1	Long1	Long1	Long1	Long1
	Long2	Long2	Long2	Long2	Long2	Long2	Long2	Long2
	Byte1	Byte2	*Wyrównanie*					

Rysunek 6.11 Zobrazowanie zajmowanej pamięci przy pomocy programu Excel gdzie komórka reprezentuje 1 bajt pamięci

Aby wskazać ten problem projektantowi i/lub programiście w narzędziu zaimplementowano dwa mechanizmy informujące o nadto rozrzutnym wykorzystaniu pamięci. Pierwszy z nich widać na Rysunek 6.9, mechanizm ten jest podsumowaniem i ewentualnym przyciskiem do automatycznej naprawy problemu w oknie kreatora komponentów. Drugi to zupełnie osobne okno do analizy wszystkich zawartych w projekcie struktur, okno te może także znaleźć typy wyliczeniowe, które niepotrzebnie używają zbyt dużej ilości pamięci. Okno te można zobaczyć na Rysunek 6.12.



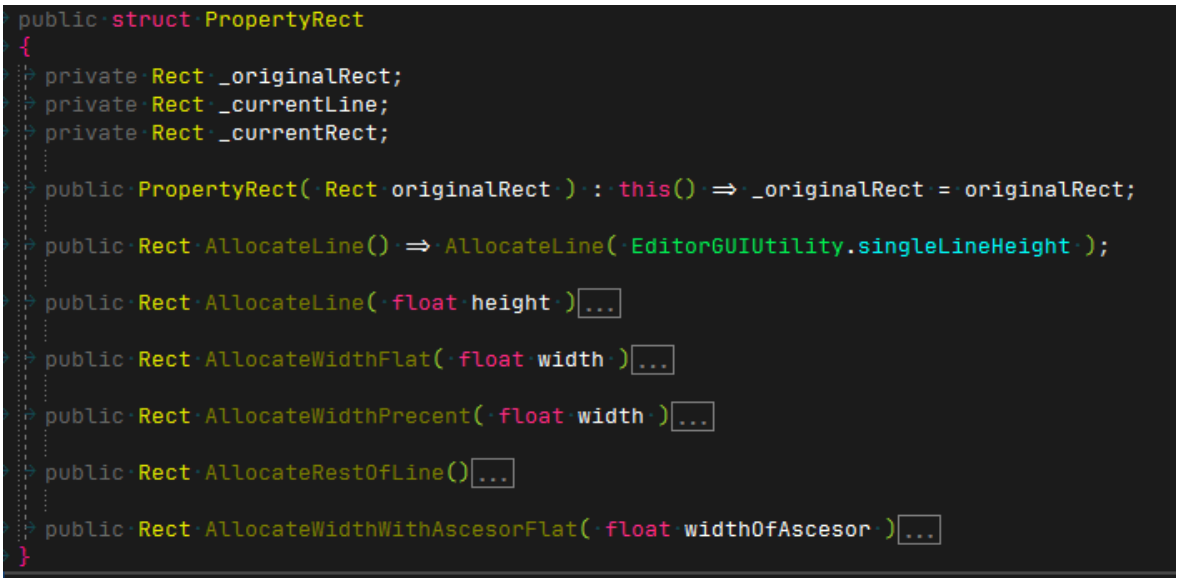
Rysunek 6.12 Okno analizy komponentów i typów wyliczeniowych wskazujące problemy w projekcie

Aby być całkowicie zgodnym z prawdą, to język C# jest wyposażony w atrybut „StructLayout”, w którym można określić sposób generowania układu jako automatyczny. W takim wypadku kompilator uczyni to samo co przycisk w oknie kreatora komponentów, jednak taka struktura nie może być używana kontekście pamięci niezarządzanej (unmanaged memory) [61] [62].

6.11. Pomniejsze artefakty

Wiele mniejszych problemów zostało rozwiązanych podczas procesu realizacji projektu, nie są to problemy na tyle duże i interesujące, aby każdy z nich opisywać w oddzielnym podrozdziale, ale są na tyle interesuje, że warto o nich wspomnieć.

PropertyRect – tryb IMGUI posiada dwie opcje układania elementów (layouting), ręczny i automatyczny, ten drugi jest wygodniejszy ale nie jest dostępny w klasach dziedziczących po klasie PropertyDrawer. Dla ułatwienia i przyśpieszenia pracy przy układaniu ręcznym, stworzono pomocniczą strukturę PropertyRect, której publiczny interfejs można zobaczyć na Rysunek 6.13.



Rysunek 6.13 Publiczny interfejs struktury PropertyRect

GUIEnabledScope – aby rysować elementy jako nieinteraktywne trzeba powtarzać kilka tych samych linii kodu, co jest sprzeczne z zasadą DRY (Nie powtarzaj siebie). Dzięki wykorzystaniu lukru syntaktycznego, autor wprowadził strukturę GUIEnabledScope (Listing 6.11), którą można zastosować tak jak pokazano na Listing 6.12.

```
1  public struct GUIEnabledScope : IDisposable
2  {
3      private bool _oldEnable;
4      public GUIEnabledScope( bool enable, bool force = false )
5      {
6          _oldEnable = GUI.enabled;
7          if ( force )
8          {
9              GUI.enabled = enable;
10         }
11         else
12         {
13             GUI.enabled &= enable;
14         }
15     }
16     public void Dispose() => GUI.enabled = _oldEnable;
17 }
```

Listing 6.11 Kod struktury GUIEnabledScope

```
1  using ( new GUIEnabledScope( false ) )
2  {
3      EditorGUILayout.TextArea(
4          S.Concat() + _componentDefinition.Fields.Length + " fields wit
h size: " + currentSize
5          + "B, when possible " + possibleSize + " B" + ( hasNested ? NE
STED_STRUCTS_WARNING : string.Empty ),
6          EditorStyles.helpBox );
```

```
7  }
```

Listing 6.12 Użycie struktury GUIEnabledScope

Konkatenacja łańcuchów znaków – konkatenacja łańcuchów znaków przy użyciu operatora '+' jest operacją wykorzystującą stosunkowo znaczną ilość pamięci. Autor wprowadził strukturę S, która utylizuje wydajniejszy sposób konkatenacji przez użycie klasy StringBuilder i przeciążenie operatora '+', aby nie wprowadzać pakowania struktur (boxing). Najważniejsze fragmenty struktury S ukazuje Listing 6.13.

```
1  public struct S
2  {
3      private static StringBuilder s_stringBuilder = new StringBuilder(1
024);
    ...
4      public static S Concat( string startingString )
5      {
6          s_stringBuilder.Clear();
7          return new S() + startingString;
8      }
9      public static implicit operator string( S _ ) => s_stringBuilder.T
oString();
10     public override string ToString() => s_stringBuilder.ToString();

11     #region + operator
12     public static S operator +( S left, char right )
13     {
14         s_stringBuilder.Append( right );
15         return left;
16     }
17     public static S operator +( S left, string right )
18     {
19         s_stringBuilder.Append( right );
20         return left;
21     }
22     public static S operator +( S left, bool right )
23     {
24         s_stringBuilder.Append( right );
25         return left;
26     }
    ...
27 }
```

Listing 6.13 Fragmenty struktury S do konkatenacji łańcuchów znaków

Rozmaite rozszerzenia – język C# pozwala na definiowanie metod rozszerzających dla zamkniętych typów. Autor napisał wiele rozszerzeń dla różnych typów. Jednymi z najciekawszych są: ForEach<T> dla IEnumerable<T> (Listing 6.14), Yield<T> dla T (Listing 6.15), SetMemberValue dla MemberInfo (Listing 6.16), GetAllFieldsValues dla object (Listing 6.17) i GetPropertyValue<T> dla SerializedProperty (Listing 6.18).

```
1  public static void ForEach<T>(
       this IEnumerable<T> elements, Action<T> action )
2  {
```

```
3       foreach ( T element in elements )
4       {
5           action( element );
6       }
7   }
```

Listing 6.14 Metoda rozszerzająca ForEach

```
1   public static IEnumerable<T> Yield<T>( this T item )
2   {
3       yield return item;
4   }
```

Listing 6.15 Metoda rozszerzająca Yield

```
1   public static void SetMemberValue(
       this MemberInfo member, object target, object value )
2   {
3       try
4       {
5           if ( member is FieldInfo fieldInfo )
6           {
7               fieldInfo.SetValue( target, value );
8           }
9           else if ( member.MemberType == MemberTypes.Property )
10          {
11              ( (PropertyInfo)member ).SetValue( target, value, null );
12          }
13      }
14      catch ( Exception e )
15      {
16          Debug.LogException( e );
17      }
18  }
```

Listing 6.16 Metoda rozszerzająca SetMemberValue

```
1   public static Dictionary<MemberInfo, object> GetAllFieldsValues(
       this object source )
2   {
3       Dictionary<MemberInfo, object> valuesDictionary = new Dictionary<M
emberInfo, object>();
4       var distinctFields = source.AllFields().GroupBy(f => f.Name).Selec
t(gr => gr.Last());
5       foreach ( MemberInfo distinctField in distinctFields )
6       {
7           if ( distinctField is FieldInfo fieldInfo )
8           {
9               valuesDictionary[distinctField] = fieldInfo.GetValue( sour
ce );
10          }
11          else if ( distinctField is MethodInfo getterInfo )
12          {
```

```
13              valuesDictionary[distinctField] = getterInfo.Invoke( sourc
e, null );
14          }
15      }
16      return valuesDictionary;
17  }
```

Listing 6.17 Metoda rozszrzająca GetAllFieldsValues

```
1   public static object GetPropertyValue(
        this SerializedProperty serializedProperty, int ofUpper = 0 )
2   {
3       string[] slices = serializedProperty.propertyPath.Split('.');
4       Type type = serializedProperty.serializedObject.targetObject.GetTy
pe();
5       object currentValue = serializedProperty.serializedObject.targetOb
ject;
6       for ( int i = 0; i < slices.Length - ofUpper; i++ )
7       {
8           if ( slices[i] == "Array" )
9           {
10              //go to 'data[x]'
11              i++;
12              // extract x
13              var index = int.Parse( DataIndexExtractRegex.Match(slices[
i]).Value );
14              var currentArray = currentValue as IEnumerable;
15              var enumerator = currentArray.GetEnumerator();
16              var hasCurrent = enumerator.MoveNext();
17              for ( int j = 0; j < index; j++ )
18              {
19                  hasCurrent = enumerator.MoveNext();
20              }
21              if ( hasCurrent )
22              {
23                  currentValue = enumerator.Current;
24              }
25              else
26              {
27                  currentValue = null;
28              }
29              if ( type.IsArray )
30              {
31                  type = type.GetElementType(); //gets info on array ele
ments
32              }
33              else
34              {
35                  type = type.GetGenericArguments()[0];
36              }
37          }
38          else
```

```
39          {
40              var fieldInfo = type.GetField(slices[i], BindingFlags.NonP
ublic | BindingFlags.Public | BindingFlags.FlattenHierarchy | BindingFlag
s.Instance);
41              currentValue = fieldInfo.GetValue( currentValue );
42              type = fieldInfo.FieldType;
43          }
44          if ( currentValue == null )
45          {
46              return currentValue;
47          }
48      }
49      return currentValue;
50  }
```

Listing 6.18 Metoda rozszerzająca GetPropertyValue

Zakończenie

W pracy zrealizowano wszystkie postawione cele. Nie oznacza to jednak kompletności uzyskanego rozwiązania. Tak jak autor wskazał w jednym ze wcześniejszych rozdziałów, istnieje wiele funkcjonalności, o które można by wzbogacić przygotowany projekt, jednak jest to typowa cecha projektów informatycznych. Stworzone narzędzie w pełni realizuje cele wybranego wycinka możliwych funkcjonalności i może stanowić znaczną pomoc zarówno dla programistów jaki i dla projektantów gier komputerowych. Przedstawione w projekcie podejścia są ciekawą i wartą rozważenia alternatywą dla standardowego podejścia przy wytwarzaniu oprogramowania.

Warto zauważyć, że wykorzystanie zewnętrznych paczek oprogramowania (xNode) i szkieletów aplikacyjnych (Unity), znacznie przyśpiesza przebieg prac, jednak w dużym stopniu dyktuje architekturę i strukturę realizowanego projektu. Wykorzystanie zewnętrznych narzędzi w dzisiejszym świecie jest konieczne, ponieważ realizowane projekty stają się coraz większe i szersze, natomiast podwojenie liczby członków zespołu rozwijającego oprogramowanie nie powoduje podwojenia efektywności tego zespołu, lecz mniejszy niż dwukrotny wzrost owej wydajności. Biorąc powyższe pod uwagę na programistach rozwijających tego typu oprogramowanie, leży duża odpowiedzialność za wydajność i inżynieryjną zgodność ze sztuką.

Jednym z zauważalnych braków w pracy jest mała ilość testów automatycznych. W branży gier komputerowych nie jest to tak znaczny brak, jak w branży np. aplikacji internetowych, gdzie przed wtyczkami i projektami wolno-źródłowymi stawia się wymagania niemal 100% pokrycia linii kodu przez testy jednostkowe i rozbudowane dokumentacje. Większość testów realizowanych w projektach jakimi są gry komputerowe, to testy manualne, a klasyczne podejście w silniku Unity zorientowane na używanie klas dziedziczących po MonoBehaviour i ScriptableObject znacznie utrudnia testy. Stosowanie wzorca pokornego obiektu (Humble Object Pattern) [15] jest mało powszechne i prowadzi do zwiększenia złożoności przypadkowej [5]. Natomiast podejścia zgodne ze wzorcem ECS wielce zwiększa możliwości testowania oprogramowania wytwarzanego z użyciem silnika Unity.

Autor uważa, że potrzebne jest większe zainteresowanie paradygmatem projektowania zorientowanego na dane. Temat ten jest jedynie płytko zbadany przez internetowych twórców. Brak znaczących akademickich prac i książek spod stajni zaufanych wydawnictw naukowych znacznie ogranicza wzrost popularności tej techniki.

Bibliografia

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

[2] M. Acton, „Data-Oriented Design and C++," w *CppCon*, 2014.

[3] T. Albrecht, „Pitfalls of Object Oriented Programming," Sony Computer Entertainment Europe, 2009.

[4] T. Mytkowicz, A. Diwan, M. Hauswirth i P. Sweeney, „Producing Wrong Data Without Doing Anything Obviously Wrong!," 2009.

[5] F. Brooks, „No Silver Bullet Essence and Accidents of Software Engineering," *IEEE Computer*, tom 20, pp. 10-19, 4 1987.

[6] EnterpriseQualityCoding, [Online]. Available: https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition. [Data uzyskania dostępu: 28 10 2020].

[7] C. Carvalho, „The gap between processor and memory speeds," 1 2002.

[8] R. Nystrom, Game Programming Patterns, Genever | Benning, 2014.

[9] U. Drepper, „What Every Programmer Should Know About Memory," 2007.

[10] D. Masiukiewicz, D. Masiukiewicz i J. SmpBka, „Badanie wzorca architektonicznego Entity-component-system zaprojektowanego z wykorzystaniem techniki Data-oriented design," 2019.

[11] R. Strzodka, „Chapter 31 - Abstraction for AoS and SoA Layout in C++," w *GPU Computing Gems Jade Edition*, W. W. Hwu, Red., Boston, Morgan Kaufmann, 2012, pp. 429-441.

[12] A. Shinsel, Intel, [Online]. Available: https://techdecoded.intel.io/resources/understanding-the-instruction-pipeline/. [Data uzyskania dostępu: 28 10 2020].

[13] K. Socha. [Online]. Available: https://github.com/KamilVDono/DOD_Performance_Benchmarks. [Data uzyskania dostępu: 03 10 2020].

[14] TIOBE, „https://www.tiobe.com/," TIOBE, [Online]. Available: https://www.tiobe.com/tiobe-index. [Data uzyskania dostępu: 07 09 2020].

[15] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st red., USA: Prentice Hall Press, 2017.

[16] P. Van Roy i S. Haridi, „Concepts, Techniques, and Models of Computer Programming," 1st red., The MIT Press, 2004.

[17] M. Samuel, „An Insight into Programming Paradigms and Their Programming Languages," 2018.

[18] M. Gabbrielli i S. Martini, „10. The Object-Oriented Paradigm," w *Programming Languages: Principles and Paradigms*, 1st red., Springer Publishing Company, Incorporated, 2010.

[19] S. Krishnamurthi i K. Fisler, „2 Paradigms as a Classical Notion of Classification," w *Programming Paradigms and Beyond*, 2019.

[20] F. Gasperoni, „Safety, security, and object-oriented programming," *ACM Sigbed Review*, tom 3, 10 2006.

[21] E. Gamma, R. Helm, R. Johnson i J. Vlissides, „Design Patterns: Elements of Reusable Object-Oriented Software," USA, Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, USA: Prentice Hall PTR, 2003.

[23] M. Gabbrielli i S. Martini, „11. The Functional Paradigm," w *Programming Languages: Principles and Paradigms*, 1st red., Springer Publishing Company, Incorporated, 2010.

[24] M. C. (skypjack), „https://skypjack.github.io/," [Online]. Available: https://skypjack.github.io/. [Data uzyskania dostępu: 28 10 2020].

[25] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar i D. Lange, *Unity: A General Platform for Intelligent Agents*, 2018.

[26] K. N. WHITLEY, „Visual Programming Languages and the Empirical Evidence For and Against," *Journal of Visual Languages & Computing*, tom 8, pp. 109-142, 1997.

[27] S. Krishnamurthi i K. Fisler, „5.3 Visual and Blocks-based Languages," w *Programming Paradigms and Beyond*, 2019.

[28] C. Kerr, 04 05 2020. [Online]. Available: https://www.gamasutra.com/view/news/362274/Unity_has_acquired_visual_scripting_solution_Bolt.php. [Data uzyskania dostępu: 05 11 2020].

[29] Unity Software Inc., Unity Software Inc., [Online]. Available: https://forum.unity.com/threads/visual-scripting-roadmap-update.951675/?_ga=2.67252561.139307952.1601050076-1994552821.1458075228. [Data uzyskania dostępu: 20 08 2020].

[30] Epic games, Epic games, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html. [Data uzyskania dostępu: 05 11 2020].

[31] J. Linietsky, A. Manzur i the Godot community, Godot, [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/shading/visual_shaders.html. [Data uzyskania dostępu: 05 11 2020].

[32] J. Lo. [Online]. Available: https://www.shaderweaver.com/. [Data uzyskania dostępu: 05 11 2020].

[33] A. Creations, Amplify Creations, [Online]. Available: http://amplify.pt/unity/amplify-shader-editor/. [Data uzyskania dostępu: 05 11 2020].

[34] Visual Studio - Microsoft, „visualstudio.microsoft," Visual Studio - Microsoft, [Online]. Available: https://visualstudio.microsoft.com/pl/services/intellicode/. [Data uzyskania dostępu: 06 11 2020].

[35] Visual Paradigm, Visual Paradigm, [Online]. Available: https://www.visual-paradigm.com/. [Data uzyskania dostępu: 06 11 2020].

[36] Shader Graph - Unity Technologies, Unity Technologies, [Online]. Available: https://unity.com/shader-graph. [Data uzyskania dostępu: 06 11 2020].

[37] D. M. Bourg i G. Seemann, „AI for Game Developers," O'Reilly Media, Inc., 2004.

[38] V. Pleskonjic. [Online]. Available: https://assetstore.unity.com/packages/tools/stateorio-finite-state-machines-in-c-97813. [Data uzyskania dostępu: 20 08 2020].

[39] M. Yannakakis, „Hierarchical State Machines," 2000.

[40] C. Simpson, „Gamastura," 17 07 2014. [Online]. Available: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php. [Data uzyskania dostępu: 15 11 2020].

[41] S. Rabin i J. Orkin, „AI Game Programming Wisdom 4," Course Technology, Cengage Learning, 2008.

[42] J. Orkin, „Three States and a Plan: The A.I. of F.E.A.R.," 2006.

[43] R. (kaminaritukane). [Online]. Available: https://github.com/kaminaritukane/ECS_FSM. [Data uzyskania dostępu: 05 09 2020].

[44] S. Mertens, „Medium," [Online]. Available: https://medium.com/@ajmmertens/why-storing-state-machines-in-ecs-is-a-bad-idea-742de7a18e59. [Data uzyskania dostępu: 25 08 2020].

[45] Unity Software Inc., [Online]. Available: https://www.sec.gov/Archives/edgar/data/1810806/000119312520227862/d908875ds1.htm. [Data uzyskania dostępu: 15 09 2020].

[46] .NET Foundation and Xamarin, [Online]. Available: https://www.mono-project.com. [Data uzyskania dostępu: 10 09 2020].

[47] S. D. (nxrighthere). [Online]. Available: https://github.com/nxrighthere/BurstBenchmarks. [Data uzyskania dostępu: 10 09 2020].

[48] L. Meijer, „Unity Software Inc.," 26 02 2019. [Online]. Available: https://blogs.unity3d.com/2019/02/26/on-dots-c-c/. [Data uzyskania dostępu: 10 09 2020].

[49] T. Brigsted. [Online]. Available: https://github.com/Siccity/xNode. [Data uzyskania dostępu: 20 07 2020].

[50] ClickUp, [Online]. Available: https://clickup.com. [Data uzyskania dostępu: 28 10 2020].

[51] Viasfora, [Online]. Available: https://viasfora.com. [Data uzyskania dostępu: 29 10 2020].

[52] hrai. [Online]. Available: https://github.com/hrai/auto-save-vs-extension. [Data uzyskania dostępu: 29 10 2020].

[53] M. Kristensen, „Marketplace Visual Studio," [Online]. Available: https://marketplace.visualstudio.com/items?itemName=MadsKristensen.AddNewFile. [Data uzyskania dostępu: 29 10 2020].

[54] S. Cadwallader. [Online]. Available: http://www.codemaid.net. [Data uzyskania dostępu: 29 10 2020].

[55] Jet Brains, Jet Brains, [Online]. Available: https://www.jetbrains.com/rider. [Data uzyskania dostępu: 29 10 2020].

[56] F. K. (taphos), 30 09 2016. [Online]. Available: https://github.com/taphos/unity-uitest. [Data uzyskania dostępu: 08 11 2020].

[57] K. Socha, 04 07 2020. [Online]. Available: https://github.com/KamilVDono/Sparkler. [Data uzyskania dostępu: 12 11

2020].

[58] Microsoft, „Microsoft,” Microsoft, 20 07 2015. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection. [Data uzyskania dostępu: 12 09 2020].

[59] Unity Software Inc., „Unity Software Inc.,” Unity Software Inc., [Online]. Available: https://docs.unity3d.com/Manual/GUIScriptingGuide.html. [Data uzyskania dostępu: 14 09 2020].

[60] Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Immediate_mode_GUI. [Data uzyskania dostępu: 13 09 2020].

[61] K. Kokosa, Pro .NET Memory Management: For Better Code, Performance, and Scalability, 1st red., USA: Apress, 2018.

[62] Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.layoutkind?view=netframework-4.7. [Data uzyskania dostępu: 18 10 2020].

Spis rysunków

Rysunek 2.1 Uproszczony schemat programu w założeniach programowania zorientowanego na dane 2
Rysunek 2.2 Przybliżony, uproszczony schemat programu w założeniach programowania zorientowanego na dane 2
Rysunek 2.3 Wyniki testów wydajnościowych wykonywania obliczeń z wykorzystania mechanizmu pamięci podręcznej i bez tego mechanizmu. 8
Rysunek 2.4 Wyniki testów badających wydajność podejść „tablica struktur z referencjami", „tablica struktur" i „struktura tablic". 8
Rysunek 2.5 Wyniki testów wydajności sterowania przepływem przez dyrektywę if i wersji bez niej. Poszczególne podpisy oznaczają: Sum_True_Test – w każdym rozgałęzieniu podjęta jest gałęź 'prawda', Sum_False_Test – w każdym rozgałęzieniu podjęta jest gałąź 'fałsz', Sum_Half_Test – połowa gałęzi jest ewaluowana przez wartość 'prawda', Sum_4_True_Test - co czwarta gałęź jest ewaluowana przez wartość 'prawda', Sum_4_False_Test - co czwarta gałęź jest ewaluowana przez wartość 'fałsz', Sum_8_True_Test - co ósma gałęź jest ewaluowana przez wartość 'prawda', Sum_8_False_Test - co ósma gałęź jest ewaluowana przez wartość 'fałsz', Sum_Random_Test – każda gałęź zależy od pseudolosowej wartości o równej dystrybucji i Sum_Random_Byte_Test – obydwa rozgałęzienia są ewaluowane a ostateczna wartość wyznaczana jest przy pomocy arytmetyki. 9
Rysunek 2.6 Przykładowa wizualizacja zbioru bitowego użytego do filtrowania bytów w ECS 14
Rysunek 2.7 Porównanie ilości wyszukiwani dla zapytań: „unreal blueprints", „unreal coding" i „unreal c++". Z serwisu https://trends.google.com. Dostęp 29.09.2020. 16
Rysunek 2.8 Przykładowa maszyna stanów skończonych, gdzie prostokąt oznacza stan, linia ze strzałką pojedynczą oznacza przejście jednostronne od stanu bez strzałki do stanu ze strzałką i linia zakończona strzałkami z obydwu stron oznacza połączenie stanów z tranzycją w obydwie strony. 18
Rysunek 2.9 Przykładowa hierarchiczna maszyna stanów, gdzie prostokąt z zaokrąglonymi rogami oznacza stan, który jest zagnieżdżoną maszyną stanów, prostokąt oznacza stan realizujący zachowanie, linia ze strzałką pojedynczą oznacza przejście jednostronne od stanu bez strzałki do stanu ze strzałką i linia zakończona strzałkami z obydwu stron oznacza połączenie stanów z tranzycją w obydwie strony. 19
Rysunek 2.10 Przykładowe drzewo decyzyjne z https://commons.wikimedia.org/wiki/File:BT_search_and_grasp.png. Dostęp 24.10.2020. 20
Rysunek 4.1 Zadanie ParallelFor dzieli paczki pomiędzy rdzenie z https://docs.unity3d.com/Manual/JobSystemParallelForJobs.html. Dostęp 21.10.2020. ... 29
Rysunek 5.1 Menu importu paczek w silniku Unity 33
Rysunek 5.2 Okno importu paczki SparklerCore.unitypackage 34
Rysunek 5.3 Menu tworzenia nowego zasobu typu SystemsGraph 34
Rysunek 5.4 Kanwa edytora nowego zasobu typu SystemsGraph 35
Rysunek 5.5 Kanwa edytora z nowopowstałym węzłem 35
Rysunek 5.6 Kanwa wykorzystana do zaprojektowania przykładu 2 36
Rysunek 5.7 Menu z dwoma dodatkowymi narzędziami instalowanymi wraz z paczką 37
Rysunek 6.1 Zwinięte i rozwinięte dyrektywy region 39
Rysunek 6.2 Po lewej domyślne węzła przez Unity, po prawej rysowanie zaimplementowane w projekcie 40

Rysunek 6.3 Dwa możliwe widoki węzłów, po lewej: widok do edycji, po prawej: widok do projektowania 42
Rysunek 6.4 Węzeł prezentujący 3 błędy z nim związane 43
Rysunek 6.5 Wygląda paska narzędzi 43
Rysunek 6.6 Okno kreatora komponentów z przykładową definicją komponentu 47
Rysunek 6.7 Wczytany system w formie wizualnej 49
Rysunek 6.8 Okno tworzenia filtrów komponentów współdzielonych 50
Rysunek 6.9 Definicja struktury A1 z Listing 5.10 w kreatorze komponentów, pokazujący że ta struktura zajmuje 32 bajty kiedy możliwe jest żeby zajmowała 24 bajty 51
Rysunek 6.10 Definicja struktury A2 z Listing 5.10 w kreatorze komponentów, pokazujący że ta struktura zajmuje 24 bajty 51
Rysunek 6.11 Zobrazowanie zajmowanej pamięci przy pomocy programu Excel gdzie komórka reprezentuje 1 bajt pamięci 51
Rysunek 6.12 Okno analizy komponentów i typów wyliczeniowych wskazujące problemy w projekcie 52
Rysunek 6.13 Publiczny interfejs struktury PropertyRect 53

Spis listingów

Listing 2.1 Przykładowa realizacja klasy piłka i logiki przesuwania jej w technice AoS i SoA 6
Listing 2.2 Struktura Stats nie spełniająca założeń komponentu ze względu na przechowywanie wielu niezależnych wartości 12
Listing 2.3 Rozbicie struktury Stats na poprawne niezależne komponenty 13
Listing 2.4 Przykładowy system realizujący logikę oznaczania bytu za martwy 13
Listing 2.5 Prosta realizacja Entity w oparciu o klasę 14
Listing 6.1 Zastosowanie dyrektywy #region 38
Listing 6.2 Fragment funkcji OnGUI w klasie SystemLambdaActionDrawer służące do rysowania instancji klasy SystemLambdaAction 41
Listing 6.3 Fragment klasy SystemsGraphEditor odpowiedzialny za obsługę paska narzędzi 44
Listing 6.4 Przykładowe użycie klasy ProcessorsSelector 45
Listing 6.5 Użycie atrybutów ProcessAfter i ProcessBefore w kodzie testów do klasy ProcessorsSelector 45
Listing 6.6 Przykładowy wygenerowany system, zawierający dwie funkcje lambda, gdzie jedna z nich oznaczona jest do realizacji tranzycji do odblokowania innego stanu i filtr na komponent współdzielony 47
Listing 6.7 Wygenerowany komponent 47
Listing 6.8 Fragment pliku SimpleTypesNames.cs z implementacji IPrimitiveType 48
Listing 6.9 Fragment generacji kodu wiedzący o IPrimitiveType 49
Listing 6.10 Kod dwóch struktur zawierające te same pola zadeklarowane w różniej kolejności 50
Listing 6.11 Kod struktury GUIEnabledScope 53
Listing 6.12 Użycie struktury GUIEnabledScope 54
Listing 6.13 Fragmenty struktury S do konkatenacji łańcuchów znaków 54
Listing 6.14 Metoda rozszerzająca ForEach 55
Listing 6.15 Metoda rozszerzająca Yield 55
Listing 6.16 Metoda rozszerzająca SetMemberValue 55
Listing 6.17 Metoda rozszrzająca GetAllFieldsValues 56
Listing 6.18 Metoda rozszerzająca GetPropertyValue 57

Spis tabel

Tabela 4.1 Podsumowanie narzędzia Visual Paradigm 24
Tabela 4.2 Podsumowanie narzędzia Unreal Engine Blueprints 25
Tabela 4.3 Podsumowanie narzędzia Unity Bolt 25
Tabela 4.4 Podsumowanie narzędzia DOTS Visual Scripting 26
Tabela 4.5 Podsumowanie narzędzia ECS_FSM 26