JCSI 3 (2017) 59-63
Journal of Computer Sciences Institute

Wysłane: 2016-12-14
Przyjęte: 2016-12-16

Analiza wydajności silnika Unity3D w aspekcie symulacji cząsteczkowych

Mateusz Walczyna*, Małgorzata Plechawska-Wójcik
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie: Dokument ten przedstawia badania w sektorze wydajności cząsteczek utworzonych przez wbudowany w Unity3D system cząsteczek oraz tworzonych poprzez skrypt napisanym w języku C#. Autor skupił się na zbadaniu wpływu poszczególnych zadań na szybkość renderowania klatki, różnicy jakie występują podczas generowania cząsteczek przez system cząsteczek, a tych generowanych przez skrypt oraz jak duży wpływ ma złożoność siatki cząsteczki generowanej na zachwianie wydajności aplikacji. W tym celu, z użyciem Unity3D, Microsoft Visual Studio oraz Blendera, została utworzona aplikacja na komputery z systemem Windows która pozwala na dostosowanie parametrów symulacji (liczby generowanych cząsteczek, kształtu cząsteczki – kula, sześcian).

Słowa kluczowe: cząsteczka; system cząsteczek; Unity3D; wydajność; symulacja

*Autor do korespondencji.

Adres e-mail: mateusz.walczyna@gmail.com

Efficiency analysis of Unity3D engine in terms of particle simulation

Mateusz Walczyna*, Małgorzata Plechawska-Wójcik
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. This document reviews research in efficiency analysys of particles created by Unity 3D built-in particle system compared to those created via sctript written in C# language. Author focused his research on examining the impact of the various tasks on the speed of rendering frames, a difference which occur during the generation of particle by particle system and those generated by a script, and how influential is the complexity of the mesh particles generated on the application performance. For this purpose, using Unity3D, Microsoft Visual Studio and Blender, an applications for computers that are ruunniong on Windows system was created, application allows to adjust the simulation parameters (the number of generated particles, particle shape - sphere, cube).

Keywords: Particle; Particle system; Unity3D; Efficiency; Simulation

*Corresponding author.

E-mail address: mateusz.walczyna@gmail.com

1. Wstęp

Efekty cząsteczkowe są nieodzownym elementem każdej dopracowanej gry dodając jej realizmu lub wręcz odwrotnie ujmując go. Używając systemu cząsteczek we właściwy sposób, ten element pozwala na dodanie odrobiny realizmu w tworzony świat.

Cząsteczki to małe elementy (obrazy 2D lub siatki 3D) [1], które emitowane w dużych ilościach oraz sterowane przez system cząsteczek [2], pozwalają na osiągnięcie efektów takich jak: dym, ogień, deszcz.

Przykładowo każda cząsteczka emitowana przez system cząsteczek który zajmuje się tworzeniem efektu deszczu lub śniegu, jest pojedynczą kroplą lub płatkiem. Podobnie jest w przypadku tworzenia efektu dymu, gdzie pojedyncza cząstka w odcieniu szarości, wyemitowana pojedynczo nie będzie przypominała postaci unoszącej się sadzy, natomiast wyemitowana wielokrotnie przez system cząsteczek będzie sprawiała to wrażenie.

Cząsteczka by spełniała określone zachowania występujące w naturze musi posiadać określone właściwości [3-4].

Właściwości cząsteczki która jest jednym z tysiąca o ile nie większej liczby generowanych elementów to m. in.: wektor prędkości określający kierunek i dystans jaki cząsteczka pokonuje podczas jednej klatki, rozmiar, kolor. Wspomniany kolor może ulegać zmianie zarówno przed jak i w czasie życia cząsteczki lub też zmieniając się w zależności od prędkości czy od czasu jaki ona istnieje.

Niektóre właściwości mogą ulec wpływom sił zewnętrznym takim jak grawitacja np. wektor prędkości może ulec zmianie, może zostać skierowany w dół (dla cząsteczek imitujących krople deszczu) lub w górę (dla cząsteczek imitujących ogień, dym) gdy siła oddziaływania grawitacji jest nieuwzględniana.

Wszystkie parametry określa się przy użyciu systemu cząsteczek, który z użyciem modułu emisji [5] uwolni je do utworzonego wirtualnego świata.

Te parametry to m. in. częstotliwość generowania cząsteczek, liczba cząstek generowana na raz, czas życia cząsteczek, kształt cząsteczki, oraz te parametry które wspomniane były wcześniej, czyli, kolor, wektor prędkości, wpływ grawitacji, rozmiar, a także aktywacja kolizji cząsteczek z innymi obiektami.

Wraz z wprowadzeniem w Unity 5 wydajniejszego silnika fizycznego PhysX [6-9] w wersji 3.3 którego kod źródłowy został udostępniony na licencji open-source pojawiło się pytanie, jak bardzo wydajne jest symulowanie kolizji wielu obiektów(cząsteczek) przez silnik w porównaniu do symulacji cząsteczek utworzonych przez wbudowany od wersji 3.5 Unity, system cząsteczek.

Analiza i porównanie zostało przeprowadzone z użyciem aplikacji desktopowej oraz przy użyciu narzędzi dostarczonych przez środowisko Unity3D (Profiler)[10].

Analiza będzie dotyczyć wbudowanego w Unity3D systemu cząsteczek oraz zostanie on porównany wydajnościowo ze skryptowo tworzonymi cząsteczkami (obiekty typu GameObject z takimi samymi właściwościami). Hipotezy jakie zostały podstawione przed wykonaniem symulacji to:

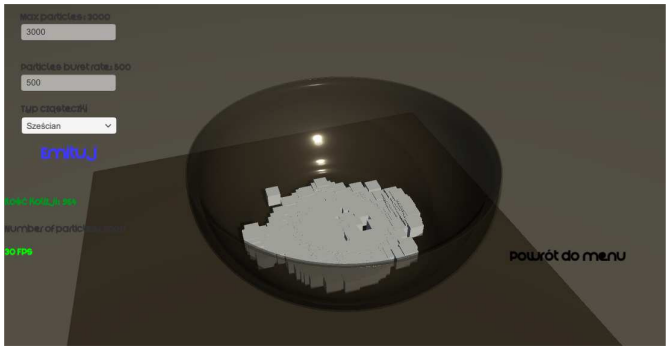
1.1. Złożoność cząsteczki emitowanej przez SC, negatywnie wpływa na wydajność aplikacji.
1.2. Liczba kolizji znacząco wpływa na spadek wydajności
1.3. Symulacja cząsteczek wyemitowanych przez system cząsteczek w większym stopniu obciąża system, niż symulacja cząsteczek wygenerowanych skryptowo.

2. Opis symulacji

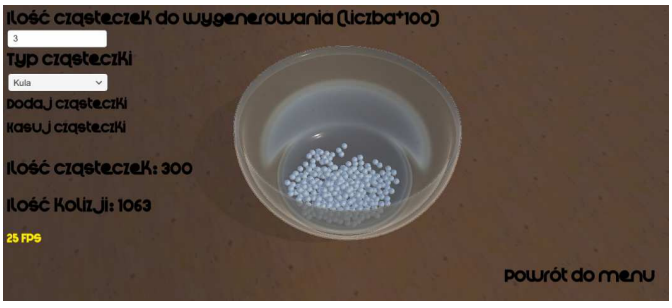
By umożliwić symulację oraz zbiór danych z wykorzystaniem narzędzia „profiler" dostępnego w środowisku Unity[11, 12], została utworzona aplikacja. Program powstał przy użyciu wspomnianego środowiska, skryptów pisanych w języku C# [13-15] w Microsoft Visual Studio, a model naczynia utworzono w programie Blender [16-18]. Aplikacja pozwala na dostosowanie parametrów niezbędnych wspomnianych w powyższym podrozdziale, a następnie emisję sparametryzowanych cząsteczek.

By umożliwić symulację program udostępniał możliwość zmiany, zdefiniowania parametrów dotyczących liczby emitowanych cząsteczek oraz kształtu cząsteczki na każdej z dostępnych (2) scen przeznaczonych do symulacji.

Widoki aplikacji są przedstawione na rysunkach poniżej (Rysunek 1 i 2).



Rys. 1. Scena symulacji systemu cząsteczek



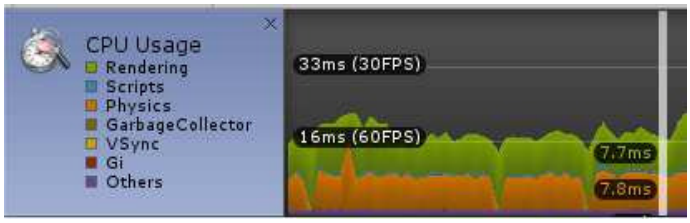
Rys.2. Scena symulacji cząsteczek tworzonych przez skrypt

Aby zbadać trafność postawionych hipotez, poszczególne symulacje przeprowadzono na 4 komputerach z systemem Windows. Parametrem, jaki był w każdej symulacji taki sam to liczba cząsteczek, która wynosiła 1000. Zmianie poddawany był kształt cząsteczki, w zależności od sposobu generowania cząsteczki.

Te przypadki to:

- Przypadek 1 - generowanie cząsteczek (sześcianów) przy użyciu systemu cząsteczek
- Przypadek 2 - generowanie cząsteczek (sześcianów) przy użyciu skryptu
- Przypadek 3 - generowanie cząsteczek (kul) przy użyciu systemu cząsteczek
- Przypadek 4 - generowanie cząsteczek (kul) przy użyciu skryptu

Odczytanie poszczególnych wyników symulacji przebiegało w każdym przypadku przez odczyt danych z *„profilera"* w następujący sposób:
W opisie rysunku „x" to numer maszyny na której została przeprowadzana symulacja, a „y" numer przypadku.



Rys.3. Użycie procesora, przypadek y, maszyna x

Overview	Total	Self	Time ms	Self ms
▶ Physics.Processing	37.3%	7.5%	6.38	1.28
Gfx.WaitForPresent	34.8%	34.8%	5.96	5.96
▶ Camera.Render	11.3%	0.3%	1.94	0.06
▶ BehaviourUpdate	4.0%	2.1%	0.68	0.37
Physics.FetchResults	3.0%	3.0%	0.52	0.52
Physics.UpdateBodies	2.9%	2.9%	0.50	0.50
▶ Physics.ProcessReports	1.7%	0.0%	0.29	0.00
Overhead	1.6%	1.6%	0.28	0.28
Profiler.FinalizeAndSendFrame	1.1%	1.1%	0.19	0.19
▶ Canvas.SendWillRenderCanvases()	0.7%	0.6%	0.12	0.11

Rys.4. CPU, procesy, przypadek y, maszyna x

Korzystając z danych dostarczonych przez to narzędzie można było określić czas generowania poszczególnej klatki, a także określić stopień obciążenia procesora oraz jakie zadanie najbardziej go obciąża. W tym wypadku czas generowania pojedynczej klatki to ~16ms, a proces w największym stopniu obciążający procesor to ten, odpowiedzialny za obliczenia związane z fizyką obiektów.

2.1. Wykorzystane konfiguracje sprzętowe

W tym podrozdziale zostaną przedstawione fizyczne parametry konfiguracji sprzętowych maszyn użytych w celach przeprowadzenia symulacji.

Maszyna 1

- Procesor: Intel Core i5-2430M
- Pamięć RAM : 6GB
- Grafika: Nvidia GeForce GT 520MX

Maszyna 2

- Procesor: Intel Core i7-5500U
- Pamięć RAM: 8GB
- Grafika: AMD Radeon R5 M330

Maszyna 3

- Procesor: Intel Core 2 Quad Q6600
- Pamięć RAM 8GB
- Grafika: Nvidia GeForce 9600 GT

Maszyna 4

- Procesor: Intel Core 2 Duo P8400
- Pamięć RAM: 4GB
- Grafika: Nvidia GeForce 9600m GT

2.2. Kod umożliwiający generowanie cząsteczek przy użyciu skryptu

By umożliwić tworzenie cząsteczek w ten sposób, musiała zostać utworzona metoda, będzie za to odpowiedzialna.

Parametrami metody są:

- Liczba cząsteczek w linii (numberOfSpheres)
- Liczba linii w płaszczyźnie (numberOfLines)
- Numer płaszczyzny (row)

Kod metody przedstawia przykład 1.

Przykład 1. Metoda odpowiedzialna za tworzenie cząsteczek przez skrypt.

```
void createPlainOfSpheres(int numberOfSpheres, int
numberOfLines, int row)
{
  Rigidbody sphereRigidbody;
  int sphereNameIndex =
    GameObject.FindGameObjectsWithTag("sphere").Length;
 for (int j = 1; j <= numberOfLines; j++)
 {
  int sphereIndex = (j - 1 == 0) ? 0 : (j - 1)
  *numberOfSpheres;
   transformVector.y +=
       1.1f*sphereCollider.bounds.size.y *row;
  for (int i = sphereIndex; i < numberOfSpheres * j; i++)
  {
     //indeks dodawany do nazwy obiektu
     sphereNameIndex++;
     sphereRigidbody = new Rigidbody();
    //utworzenie obiektu o zadanym typie (kula lub sześcian)
    sphere = GameObject.CreatePrimitive(objectType);
   //do właściwości komponentu Collider
   //zostaje przypisany materiał
   //(po to by cząsteczka m.in. odbijała się)
   sphere.GetComponent<Collider>().material = pm;
   //nadanie tagu
   sphere.tag = "sphere";
   //dodanie skryptu do obiektu po to by
   //zliczać kolizje występujące na obiekcie
   sphere.AddComponent<ObjectsCollisionScript>();
  //dodanie komponentu Rigidbody
  sphereRigidbody = sphere.AddComponent<Rigidbody>();
  //nadanie obiektowi masy
  sphereRigidbody.mass = Random.Range(4.0f, 500.0f);
  //ustawienie oddziaływania siły grawitacji na obiekt
  sphereRigidbody.useGravity = true;
  //nadanie nazwy obiektowi
  sphere.name = "sphere" + sphereNameIndex;
  //określenie sposobu detekcji kolizji
    sphereRigidbody.collisionDetectionMode =
    CollisionDetectionMode.ContinuousDynamic;
    //nadanie pozycji
    transformVector.x += 1.1f*sphereCollider.bounds.size.x;
    sphere.transform.position = transformVector;
  }
 transformVector = tableCenter;
 transformVector.z += 1.1f*sphereCollider.bounds.size.z * j;
 }
 transformVector = tableCenter;
}
```

Metoda napisana w języku C#[13-15], każdemu tworzonemu prymitywowi (sześcianowi lub kuli) nadawała odpowiednie właściwości przy użyciu odpowiednich komponentów. Tak utworzony obiekt posiada własną siatkę kolizji, masę, materiał, nazwę.

3. Wyniki symulacji

W tym rozdziale zostały zebrane wyniki wszystkich przeprowadzonych symulacji na każdej z maszyn (Tabela 1), przedstawione w postaci tabeli tak, by w łatwy sposób zauważyć różnice w kwestiach liczby generowanych klatek na sekundę jak i czasu generowania klatki. Kolumny na niebiesko oznaczają dane zebrane przy okazji generacji cząstek przy użyciu systemu cząsteczek wbudowanego w Unity, kolumny zielone zaś oznaczają dane zebrane przy okazji emisji cząsteczek przy pomocy skryptu.

Tabela 1. Tabela wyników symulacji

Wyniki symulacji z poszczególnych maszyn	Przypadek 1	Przypadek 2	Przypadek 3	Przypadek 4
1	15 kl/s	80 kl/s	11 kl/s	66 kl/s
	65 ms	12 ms	86 ms	15 ms
2	60 kl/s	170 kl/s	22 kl/s	83 kl/s
	16 ms	6 ms	45 ms	12 ms
3	12 kl/s	66 kl/s	3 kl/s	70 kl/s
	78 ms	15 ms	260 ms	14 ms
4	8 kl/s	13 kl/s	3 kl/s	37 kl/s
	125 ms	74 ms	310 ms	26 ms

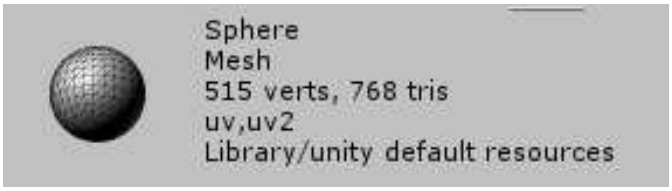
4. Analiza wyników symulacji

Jak można odczytać z tabeli 1, czasy generowania klatek podczas generowania cząsteczek przez wbudowany system cząsteczek(SC) czy czasy cząsteczek generowanych skryptowo różnią się w niektórych przypadkach nawet trzykrotnie (dla wyników symulacji na maszynie 3).

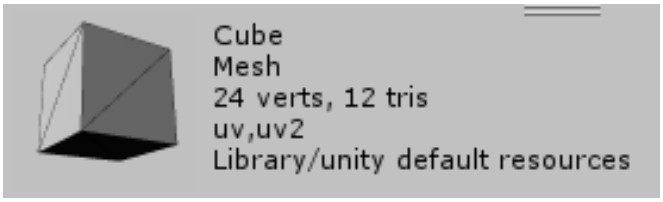
Wynika to z tego, że procesor który zajmuje się także tworzeniem, rysowaniem obiektów na scenie wymaga więcej czasu na stworzenie cząsteczki która przedstawia obiekt kuli, a mniej dla sześcianu.

Jak jest to widoczne na rysunkach poniżej, kula (Rysunek 3), składa się, aż z 515 wierzchołów i 768 trójkątów podczas gdy sześcian (Rysunek 4), zbudowany jest z jedynie 24 wierzchołków i 12 trójkątów, mimo wsparcia ze strony Unity technologią „Dynamic Batch" [20] która usprawnia działanie poprzez zaprzestanie wysyłania zadania przez procesor przy każdym rysowaniu tego samego obiektu, a wysłanie go tylko

raz by zostało wykonane określoną ilość razy dla obiektów o tej samej siatce, wolna szyna między procesorem a układem graficznym napotka problemy przy generowaniu dużej ilości kul, właśnie przez złożoność jej siatki.



Rys. 5. Obiekt kuli używany jako cząsteczka



Rys. 6. Sześcian używany jako cząsteczka

W symulacjach cząsteczek w przypadku tej aplikacji to druga przyczyna spadku liczby klatek generowanych na sekundę.

Pierwszą przyczyną spadku, są symulacje fizyki, zderzeń[21, 22] (w Unity obliczenia przeprowadzane są na procesorze z użyciem PhysX [9] dla prymitywów i bez użycia PhysX dla cząsteczek generowanych przy pomocy SC) co w wypadku generowania cząsteczek przez SC jak i tych tworzonych skryptowo, mogą doprowadzić nawet do zawieszenia aplikacji z braku dostępnych zasobów.

Biorąc pod uwagę czas generowania klatki jak i liczbę klatek na sekundę, można zauważyć, że typ cząsteczki generowanej przez SC, ma istotny wpływ na szybkość generowania z racji swojej mniej lub bardziej złożonej budowy - im mniej skomplikowana jest cząsteczka tym przy zadanej ilości cząsteczek do emisji będzie mniejszym obciążeniem dla procesora.

Podobna sytuacja wystąpiła podczas symulacji przypadków 2 oraz 4, zmiana kształtu cząsteczki emitowanej znacząco (co jest widoczne w tabeli 1) wpłynęła na wydajność aplikacji.

Wyniki we wszystkich przypadkach, są potwierdzeniem hipotezy na temat wpływu złożoności siatki cząsteczki na wydajność aplikacji.

Kolejna hipoteza stawiała pytanie, który sposób symulacji cząsteczek w Unity 3D jest wydajniejszy (czy jest to symulowanie cząsteczek wygenerowanych przez SC czy może tych wygenerowanych skryptowo).

Analizując wyniki z tabeli 1, można jednoznacznie określić która symulacja w mniejszym stopniu obciąża zasoby systemu, a co za tym idzie, czas generowania klatki jest mniejszy.

Hipoteza jest w tym przypadku częściowo prawdziwa, z tego powodu, że cząsteczki tj. prymitywy które znajdowały się w naczyniu pomimo wywołanej większej liczby kolizji, nie zmuszały one procesora do obliczania danych w sposób ciągły, dla wszystkich elementów, a jedynie dla niewielkiej liczby elementów które się poruszały wywołując nowe kolizje. Przy wstępnym zderzeniu wszystkich cząsteczek z naczyniem, czas generowania klatki zwiększa się niemal 10-krotnie, wtedy można powiedzieć że symulacja cząsteczek tworzonych przez SC jest mniej obciążająca system, z racji wywołania mniejszej liczby kolizji.

Te wnioski dają odpowiedź na ostatnią hipotezę odnośnie wpływu liczby kolizji na wydajność aplikacji. Hipoteza jest całkowicie trafna, ponieważ jak wspomniane zostało wyżej, większa liczba kolizji, która występuje w czasie wstępnej zderzeń prymitywów z naczyniem jak i między sobą w większym stopniu obciążają system niż kolizje wywoływane przez cząsteczki wygenerowane przez system cząsteczek.

5. Podsumowanie

Podczas analizy skupiono się na złożoności cząsteczki emitowanej oraz sposobie w jaki ona była emitowana (skryptowo lub przez system cząsteczek).

Jedna z hipotez (1.3) okazała się częściowo trafna , ponieważ cząsteczki (prymitywy) po usytuowaniu się w naczyniu nie powodują powstawania nowych kolizji, przez co, nie istnieje potrzeba ponownych obliczeń dla kolizji na procesorze dla każdego obiektu. Biorąc pod uwagę cząsteczki wygenerowane przez SC, sytuacja jest zupełnie odwrotna. Cząsteczki są w ciągłym ruchu, powodują kolizje w coraz to innych miejscach, stąd procesor jest bardziej obciążony podczas symulacji, niż w przypadku symulacji prymitywów, które obciążają procesor obliczeniami związanymi z fizyką, kolizjami do momentu, gdy nie ułożą się w naczyniu na scenie.

Można dość do wniosku, że emisja przez system cząsteczek wbudowany w Unity jest mniej wydajna, niż emisja cząsteczek przy użyciu skryptu.

Jak zostało wykazane, największym obciążeniem podczas symulacji były obliczenia związane z fizyką, kolizjami.

Wbudowany system daje możliwość prostej, sprawnej konfiguracji w celu uzyskania efektu takiego jak ogień, deszcz, efekty które trudno osiągnąć przy pomocy cząstek generowanych przy użyciu skryptu.

Tworząc te efekty, rzadko kiedy cząsteczki dochodzą do kolizji z innym elementem świata gry, co pozwala na utworzenie ich w większej liczbie oraz nie obciążającej procesora w znacznym stopniu, co jest wystarczające dla twórców gier, by z niego korzystać.

Cząsteczki emitowane w grach, rzadko mają postać siatek 3-wymiarowych, (które zostały użyte w celach symulacji) a są to elementy 2-wymiarowe, które nie obciążają w takim stopniu procesora.

Innymi słowy system cząsteczek daje twórcy większą elastyczność, przy mniejszym nakładzie pracy.

Literatura

[1] https://docs.unity3d.com/Manual/PartSysWhatIs.html, dostęp 11.09.2016
[2] Eric Van de Kerckhove, Introduction to Unity: Particle Systems, https://www.raywenderlich.com/113049/introduction-unity-particle-systems, dostęp 08.09.2016
[3] William T. Reeves, Particle Systems A Technique for Modeling a Class of Fuzzy Objects, „Computer Graphics", wydanie 17, numer 3, Lipiec 1983, s. 360-363

[4] Allen Martin, Particle Systems, https://web.cs.wpi.edu/~matt/courses/cs563/talks/psys.html, dostęp 10.09.2016
[5] https://docs.unity3d.com/Manual/PartSysEmissionModule.html, dostęp 7.09.2016
[6] Adrian Boeing, Thomas Bräunl, Evaluation of real-time physics simulation systems, GRAPHITE '07, s. 284-288
[7] Sean C. Mondesire, Douglas B. Maxwell Jonathan Stevens, Steven Zielinski, Glenn A. Martin, Physics Engine Benchmarking in Three-Dimensional Virtual World Simulation, MODSIM World 2016, s. 5-8
[8] Anthony, High-performace physics in Unity 5, https://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/, dostęp 8.09.2016
[9] http://physxinfo.com/wiki/PhysX_SDK_3.x, dostęp 8.09.2016
[10] https://docs.unity3d.com/Manual/ProfilerWindow.html, dostęp 8.09.2016
[11] https://unity3d.com/, dostęp 8.09.2016
[12] https://en.wikipedia.org/wiki/Unity_(game_engine), dostęp 8.09.2016
[13] Andrew Stellman, Jennifer Greene, Head First C#, O'Reilly Media, Listopad 2007
[14] Alex Okita, Learning C# Programming with Unity 3D, CRC Press, Lipiec 7, 2014
[15] Jeff W. Murray, C# Game Programming Cookbook for Unity 3D, CRC Press, Czerwiec 24, 2014
[16] Ben Simonds, Blender. Praktyczny przewodnik po modelowaniu, rzeźbieniu i renderowaniu, Helion, Maj 22, 2014, s. 49-68
[17] Thorn Alan, Unity i Blender. Praktyczne tworzenie gier, Helion, Kwiecień 3, 2015
[18] https://pl.wikipedia.org/wiki/Blender_(program), 8.09.2016
[19] https://pl.wikipedia.org/wiki/Microsoft_Visual_Studio, 8.09.2016
[20] Yorick, 4 ways to increase performance of your unity game, http://www.paladinstudios.com/2012/07/30/4-ways-to-increase-performance-of-your-unity-game/, dostęp 12.09.2016
[21] http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/RigidBodyCollision.html, dostęp 14.09.2016
[22] http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/RigidBodyDynamics.html, dostęp 14.09.2016