POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ

PRACA DYPLOMOWA MAGISTERSKA
na kierunku INFORMATYKA
specjalność: Inżynieria Komputerowa

Rafał OSTROWSKI
Nr albumu: 195395

Rok akad.: 2009/2010
Warszawa, 11.04.2009

OPRACOWANIE ŚRODOWISKA DO BUDOWY GIER 2D W JĘZYKU JAVA I ZINTEGROWANEGO Z NIM SILNIKA FIZYKI 2D

Zakres pracy:

1. *Wprowadzenie i sformułowanie celu pracy*
2. *Przegląd dostępnych silników fizyki*
3. *Projekt i implementacja silnika fizyki 2D*
4. *Projekt i implementacja środowiska*
5. *Weryfikacja możliwości środowiska na podstawie przykładowej gry*
6. *Podsumowanie i wnioski*

Kierujący pracą: dr inż. Witold Czajewski

Konsultant:

Termin złożenia pracy: *15.09.2010*

OPRACOWANIE ŚRODOWISKA DO BUDOWY GIER 2D W JĘZYKU JAVA I ZINTEGROWANEGO Z NIM SILNIKA FIZYKI 2D

Streszczenie

Celem tej pracy było opracowanie środowiska do budowania gier 2D oraz zintegrowanego z nim silnika fizyki 2D.

Pracę podzielono na dwie zasadnicze części: teoretyczną i praktyczną.

W części teoretycznej pracy opisano i porównano dostępne silniki fizyczne 3D. Zastanowiono się nad przydatnością każdego z nich do budowania komercyjnych gier dostępnych dla szerokiego wachlarza platform docelowych. Ponadto przedstawiono ogólną zasadę działania oraz możliwe zastosowania dla tego typu silników w świecie nauki oraz multimedialnej rozrywki. Następnie na zakończenie części traktującej o silnikach fizyki szczegółowo opisano implementację własnego silnika fizyki 2D potrafiącego symulować bryły sztywne i miękkie.

Celem części praktycznej był projekt i implementacja środowiska do budowy gier 2D oraz integracja go z silnikiem fizyki 2D opisanym pod koniec części teoretycznej. W tej części omówiono modularną budowę środowiska, szczegółowo przedstawiono najważniejsze moduły i opisano mechanizmy rządzące jego pracą. Następnie przedstawiono przykład wykorzystania środowiska do budowy prostej gry 2D.

Przedstawienie działającego projektu takiej gry, na wykonanie którego poświęcono około godzinę, jednoznacznie potwierdza, że udało się osiągnąć cel niniejszej pracy.

W ostatnim rozdziale autor przedstawia problemy napotkane w trakcie pracy nad tytułowym środowiskiem i silnikiem 2D, sposoby ich rozwiązania oraz podsumowuje osiągnięte rezultaty i definiuje przyszłe kierunki rozwoju środowiska.

THE STUDY OF 2D GAMES DEVELOPMENT ENVIRONMENT IN JAVA AND THE INTEGRATED 2D PHYSICS ENGINE

Abstract

The objective of this thesis was to study the 2D games development environment and 2D physics engine integrated with it.

This work is divided into two major parts: theoretical and practical.

Theoretical part includes the description and comparison of available 3D physics engines.

The author analyses usability of each engine in terms of production of commercial games for different platforms available on the market, as well as main principles of their operation and applications in the world of science and multimedia projects. The final part of the chapter, focused on the engines and their functioning, covers detailed description of implementation of the author's 2D physics engine capable of running both rigid and soft bodies.

The aim of the practical part was to describe the process of designing and the implementation of the 2D games development environment integrated with the previously described 2D physics engine. This part covers general description of modularity of development environment design and detailed description of most important modules and mechanisms running the environment. Next, there is a detailed description of the process of development of a model game using the 2D games development environment built for the use of this thesis.

The whole process ends with the fully interactive game, the development of which took less than an hour. This confirms that the main goal of this thesis has been reached.

In the final chapter the author analyses problems which appeared in the process of creation of the 2D engine and the development environment, solutions to these problems, summary of results and also defines future paths of development of this environment.

Politechnika Warszawska Warszawa, 15.09.2010r.

Wydział Elektryczny

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, ze niniejsza praca dyplomowa magisterska pt. OPRACOWANIE ŚRODOWISKA DO BUDOWY GIER 2D W JĘZYKU JAVA I ZINTEGROWANEGO Z NIM SILNIKA FIZYKI 2D:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązujacymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne PolitechnikiWarszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, ze niniejsza wersja pracy jest identyczna z załaczoną wersją elektroniczną.

Rafał OSTROWSKI.....

Spis Treści

1. Wprowadzenie i sformułowanie celu pracy

W czasach, gdy proste gry zdominowały rynek szybkiej elektronicznej rozrywki oraz w czasach serwisów takich jak Google Play, czy iTunes zrodziła się potrzeba szybkiego prototypowania i wykonywania nieskomplikowanych gier. Proces programowania gier od podstaw, w którym każdy moduł tworzonej gry trzeba tworzyć na nowo lub kustomizować dostepne rozwiązania jest mało efektywny i nie pozwala w pełni wykorzystać potencjału dzisiejszego rynku. Dlatego autor tej pracy postanowił stworzyć narzędzie, które będzie pomocne w szybkim tworzeniu nieskomplikowanych gier 2D, zdolnych do uruchonienia na wielu platformach.

Celem pracy jest stworzenie środowiska, które będzie ułatwiać pracę programiście gier 2D, to znaczy gier, w których rozgrywka odbywa się na płaskiej przestrzeni. Do tego rodzaju gier należą gry platformowe, gry z widokiem z góry, oraz wszelkiego rodzaju gry logiczne prezentujące płaską planszę rozgrywki. Większość tego typu tytułów współdzieli bardzo wiele mechanizmów, których tworzenie od zera dla każdego nowego tytułu jest uciążliwe i mało ciekawe. Celem tej pracy jest budowa narzędzia, które zminimalizuje nakład pracy potrzebny do zaprogramowania takich gier, oraz ułatwi ich późniejszą edycję.

Minimalizacja nakładów związanych z rozpoczęciem programowania gry oznacza ułatwienia przy definiowaniu elementów gry, ich wyglądu oraz zachowania w trakcie rozgrywki. Dlatego jednym z wymogów takiego środowiska jest posiadanie dwuwymiarowego edytora geometryczno- graficznego. Ponadto ważnym zagadnieniem będzie mechanizm odpowiedzialny za animowanie elementów gry. W tym celu wykorzystany zostanie mechanizm zwany silnikiem fizyki, który w ogólności służy do symulacji ruchu różnych przedmiotów w taki sposób, aby ruch ten przypominał zachowanie przedmiotów znane z rzeczywistości.

Ponadto stworzenie takiego środowiska oraz własnej implementacji silnika fizyki wymaga dogłębnego zrozumienia mechaniki bryły sztywnej oraz zbudowania/wykorzystania bliblioteki geometrii, co według autora jest bardzo interesującym zadaniem i znajdzie praktyczne zastosowanie w jego pracy zawodowej.

1.1. Układ pracy

Praca oprócz wstępu (rozdział 1) i podsumowania (rozdział 5) posiada dwie główne części traktujące o: silnikach fizyki (rozdział 2) oraz o zintegrowanym środowisku (rozdział 3). W rozdziale poświęconym silnikom fizyki autor opisuje i porównuje silniki komercyjne dostępne na runku, po czym przechodzi do implementacji własnego rozwiązania. Rozdział dotyczący środowiska na początku przedstawia projekt aplikacji w notacji UML, po czym przechodzi do omówienia szczegółów zaimplementowanych wewnątrz środowiska rozwiązań.

Po omówieniu implemetnacji środowiska autor przystępuje do prezentacji osiągniętych efektów (rozdział 4). Służy temu szczegółowy opis procesu tworzenia prostej przykładowej gry, w której gracz steruje miękka kulką. Celem gry jest dotarcie kulką do punktu końcowego planszy. W rodziale 5 autor zebrał podsumowanie i wnioski jakie nasuwają się po wykonaniu implementacji środowiska. Opisuje tam także trudności jakie napotkał w trakcie pracy oraz dzieli się wykorzystanymi rozwiązaniami.

1.2. Czym jest zintegrowane środowisko

Jest to zespół narzędzi programistycznych, które mają ułatwić i przyspieszyć wykonanie gry. Proces ten polega na połączeniu kilku modułów, które w efekcie zapewnią użytkownikowi interakcję ze światem gry, narzucą temu światu pewne zasady działania oraz uruchomią go tak, aby stworzyć iluzję jego autonomicznego życia.

W ogólnym podejściu można wyróżnić takie moduły:

- Silnik fizyki - odpowiada za symulację obiektów świata gry oraz ich interakcję między sobą. Obiekt gracza, jeżeli został wyróżniony w danej grze, również należy do jej świata i oddziałuje z innymi obiektami.

- Silnik grafiki - odpowiada za rysowanie świata gry oraz wszystkich elementów graficznego interfejsu użytkownika gry i środowiska.

- Środowisko skryptowe - pozwala tworzyć rozbudowane algorytmy opisujące prawa rządzące zachowaniem obiektów świata gry (Sztuczna Inteligencja) oraz interakcji między nimi a graczem.

    - Interpreter skryptów
    - API pozwalające skryptom wpływać na świat gry poprzez edycję jego własności fizycznych oraz tego w jaki sposób silnik graficzny prezentuje je graczowi

- Edytor geometrii - służy do definiowania geometrii obiektów świata gry

    - Biblioteka matematyczna z obsługą geometrii.
    - Środowisko graficzne prezentujące edytowany świat wraz z obiektami
    - Zestaw narzędzi operujących na świecie gry i jego obiektach

2. Silniki fizyki

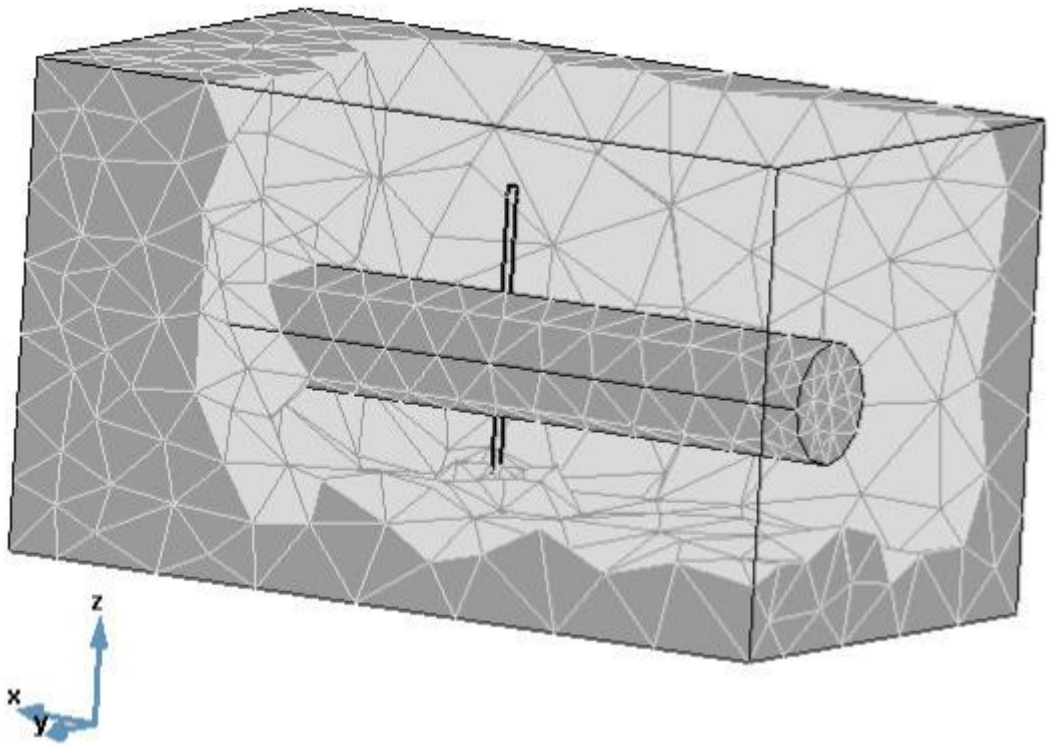
Silnik fizyki to oprogramowanie służące do symulowania konkretnych układów fizycznych takich jak układy brył sztywnych (wraz z detekcją kolizji), układy brył miękkich czy symulacje płynów. Systemy tego rodzaju znajdują szerokie zastosowanie w dziedzinach takich jak grafika komputerowa, gry wideo oraz kino.

Podstawę zastosowań stanowią jednak gry komputerowe, w których silniki pełnią funkcję modułów pośrednich między graczem a światem gry. W tym wypadku symulacja odbywa się w czasie rzeczywistym. Termin Silnik fizyki oznacza czasem także bardzo skomplikowane oprogramowanie, służące do dokładnej symulacji zjawisk fizycznych.

Ogólnie rzecz biorąc rozróżnia się dwa rodzaje silników fizyki: działających w czasie rzeczywistym oraz takich które zapewniają dokładność obliczeń. Silniki, których obliczenia są dokładne wymagają więcej mocy obliczeniowej w celu precyzyjnego rozwiązywania równań fizycznych i z reguły są używane przez naukowców oraz ludzi z branży filmowej. Silniki czasu rzeczywistego, używane w grach komputerowych i innych formach interaktywnej rozrywki korzystają z uproszczonych obliczeń oraz zmniejszonej dokładności w celu zapewnienia płynności rozgrywki.
W większości gier komputerowych czynnikiem przeważającym jest szybkość symulacji, z reguły uzyskuje się ją kosztem dokładności obliczeń. W ogólności obiekty trójwymiarowe w grach reprezentowane są przez dwie niezależne siatki zbudowane z wielokątów. Pierwsza z siatek bardzo dokładnie odwzorowuje wygląd przedmiotu, składa się często z milionów trójkątów i w detalach odwzorowuje wygląd obiektów wewnątrz gry, dla przykładu można rozważyć model czajnika. Druga siatka, w celu możliwie największego przyspieszenia obliczeń zbudowana jest z dużo mniejszej liczby wielokątów i jedynie przypomina kształtem symulowany przedmiot. Ta używana jest do reprezentacji obiektu wewnątrz sceny symulacji silnika fizycznego i nie jest widoczna dla użytkownika. Dla silnika fizycznego czajnik może być opisany przez bardzo ogólny kształt, na przykład sferę. Wtedy niemożliwe jest np. przeplecenie liny bądź cienkiego patyka przez ucho czajnika, ponieważ silnik fizyki nie wie o tym że ucho istnieje, i traktuje czajnik tak samo jak na przykład kulę do kręgli. Ta uproszczona siatka odwzorowująca kształt przedmiotu wewnątrz silnika fizycznego często

nazywana jest siatką kolizji. Może ona być zarówno prostopadłościanem otaczającym, sferą, lub po prostu otoczką wypukłą danego ciała [1]. Silniki, które korzystają jedynie z takich kształtów niedokładnie opisujących symulowane przedmioty zaliczane są do grupy silników nieskomplikowanych. Generalnie, prostopadłościany otaczające są szeroko używane do zawężenia zbiorów kształtów kolidujących między sobą, co ogranicza liczbę porównań bardziej skomplikowanych siatek reprezentujących te przedmioty w momencie wykrywania kolizji.

Alternatywą do wykorzystywania siatek i prostopadłościanów otaczających jest użycie metody elementów skończonych [2]. W tej metodzie model 3D dzielony jest na skończoną liczbę prostych elementów 3D wypełniających zajmowaną przez niego przestrzeń - proces ten nazywa się teselacją wolumetryczną [3].



Rysunek 2.1.: Teselacja wolumetryczna modelu 3D z pokazaną strukturą czworoboków wypełniających model. Źródło: http://www.springerimages.com/Images/Engineering/5-10.1186_1475-925X-4-2-1.

Teselacja produkuje zbiór elementów, które reprezentują własności fizyczne obiektu (twardość, plastyczność, objętość, itp). Po podzieleniu objętości na zbiór przylegających komórek specjalny solver (algorytm rozwiązujący równania fizyczne, od ang. solve - rozwiązywać) używany jest do symulowania naprężeń wewnątrz obiektu. Naprężenia te mogą być wykorzystane do symulacji fragmentacji, deformacji

oraz innych fizycznych zachowań. W miarę zwiększania liczby elementów generowanych przez teselację wzrasta zdolność do prawidłowego odwzorowania fizycznych własności obiektu. Wizualna reprezentacja obiektu na scenie symulacji zastępowana jest reprezentacją znajdującą się wewnątrz systemu elementów skończonych poprzez użycie odpowiednich shaderów (małych programów uruchamianych na GPU) deformacji uruchamianych na karcie graficznej maszyny wykonującej symulację. Do tej pory systemy tego rodzaju nie znajdowały praktycznego zastosowania w grach komputerowych z powodu dużego obciążenia obliczeniowego oraz braku odpowiednich narzędzi wykonujących teselację wolumetryczną modeli 3D wykorzystywanych w grach. Jednak dzięki szybkiemu przyrostowi mocy obliczeniowej komputerów, wykorzystaniu jednostek GPU do wykonywania części obliczeń fizycznych oraz wraz z pojawieniem się odpowiednich narzędzi pozwalających szybko przeprowadzać teselację, wykorzystanie takich systemów stało się możliwe. Pierwszą grą korzystającą z tej metody był tytuł „Star Wars: The Force Unleashed“ wyprodukowany przez studio LucasArts Entertainment. Wykorzystywał on silnik o nazwie Digital Molecular Matter (DMM) stworzony przez firmę Pixelux Entertainment. Rewolucja polegała na umożliwieniu graczowi deformacji i niszczenia elementów sceny takich jak drewno, stal, ciała przeciwników i rośliny. Deformacje i fragmentacje wykonywane przez DMM obliczane były przez solver opary na algorytmach opracowanych przez dr James'a O'brien'a, stanowiących część jego pracy doktorskiej „Graphical Modeling and Animation of Brittle Fracture“ [4]. Przykłady implementacji innych modeli deformacji fizycznych obiektów 3D ciekawie opisano w pozycjach [5] [6] [7].

Rysunek 2.2. Fragmentacja modelu królika wykonana przy użyciu DMM. Źródło: http://graphics.berkeley.edu/papers/Obrien-GMA-1999-08/.

W prawdziwym świecie zjawiska fizyczne zachodzą bez przerwy, przykładem mogą być ruchy Browna [8], chaotyczne ruchy cząsteczek płynu lub gazu wywołane wzajemnymi zderzeniami. W przypadku silnika fizycznego taka ciągła i precyzyjna symulacja jest nie jest wymagana i marnowałaby jedynie ograniczoną moc obliczeniową maszyny wykonującej symulację. Dla przykładu, w trójwymiarowym wirtualnym świecie Second Life, obiekt który spoczywa na podłożu i nie zmienia swojego położenia dłużej niż dwie sekundy jest kwalifikowany przez silnik jako nieruchomy przez co następuje wyłączenie go z grupy aktualnie symulowanych obiektów, zostaje zamrożony. Pozostaje w tym stanie do momentu, dopóki nie dojdzie do kolizji między nim a jakimś innym aktywnie symulowanym ciałem. To zamrażanie nieruchomych ciał pozwala silnikowi fizycznemu na zaoszczędzenie cennych zasobów obliczeniowych maszyny przeprowadzającej symulację i zwiększenie liczby klatek generowanych w jednostce czasu. Takie podejście może jednak nastręczać trudności, dla przykładu bardzo ciężkie i powoli poruszające się wahadło w momencie sukcesywnego zwalniania i zmiany kierunku ruchu może zostać zakwalifikowane jako kandydat do zamrożenia, co zepsułoby symulację. Proces zamrażania często nazywany jest usypianiem obiektów i niektóre silniki fizyczne posiadają mechanizmy umożliwiające ominięcie usypiania konkretnych obiektów na życzenie użytkownika. Brak usypiania pochłania jednak dodatkowe zasoby CPU i bezpośrednio przekłada się na wydłużony czas generowania pojedynczej ramki symulacji.

Jednym z głównych ograniczeń silników fizyki wpływającym na realizm generowanej symulacji jest dokładność liczb zmiennoprzecinkowych używanych do reprezentowania położenia oraz sił działających na symulowane obiekty. Zbyt mała precyzja może powodować błędy zaokrągleń przez co ustalenie prawidłowej pozycji obiektu będzie nieprawdziwe i symulacja będzie niestabilna numerycznie. Błędy te są potęgowane w sytuacjach, gdzie dwa swobodnie poruszające się obiekty połączone są ze sobą z precyzją większą niż ta, którą silnik fizyki może obliczyć. Może to prowadzić do nienaturalnych sytuacji, w których obiekty akumulują energię w wyniku błędów zaokrągleń, co w końcu doprowadza do bardzo widocznego trzęsienia lub nawet rozpadu symulowanych obiektów. Każdy rodzaj poruszających się obiektów, połączonych pewnymi więzami jest podatny na tego rodzaju błędy, jednak szczególnie wrażliwe na to są układy symulujące łańcuchy poddane działaniu bardzo dużych

naprężeń, oraz pojazdy kołowe z aktywnie symulowanymi łożyskami. Zwiększenie precyzji obliczeń pozwala na pozbycie się tego rodzaju błędów reprezentacji położenia i sił działających na obiekty, ale dzieje się to kosztem szybkości wykonywanych obliczeń.

2.1. Silniki naukowe

Historia silników fizyki używanych na potrzeby naukowych symulacji fizycznych sięga lat osiemdziesiątych. Pierwsze zastosowania dotyczyły prognozowania pogody, przepływów mas powietrza w atmosferze oraz pływów oceanicznych. Dziedzina ta znana jest pod nazwą: obliczeniowe modelowanie dynamiki płynów, w której cząsteczki powietrza mają przyporządkowane wektory sił. Wektory te obliczone dla szerokich przestrzeni wspólnie ilustrują ogólne wzorce zmian cyrkulacji powietrza w atmosferze. Na potrzeby spełnienia bardzo wysokich wymogów szybkości i precyzji obliczeń stworzone zostały specjalne procesory wektorowe, które charakteryzuje zdolność wykonania pojedynczej operacji na skończonym zbiorze danych tego samego typu. Dzięki temu w pojedynczym cyklu procesora wektorowego wykonywane są obliczenia na wielu obiektach na raz, co znacząco wpływa na obniżenie czasu przez nie wymaganego.

Przewidywanie pogody w dalszym ciągu jest generalnie bardzo niedokładne z powodu zbyt małego skomplikowania układu ciał modelujących rzeczywiste warunki atmosferyczne. Drobne fluktuacje występujące w świecie rzeczywistym nie znajdują odwzorowania w modelowanej rzeczywistości co drastycznie wpływa na dokładność prognoz długoterminowych.

Bardzo podobne silniki fizyczne modelujące mechanikę płynów są szeroko wykorzystywane podczas projektowania nowych statków powietrznych i wodnych. Są na tyle dokładne, że często eliminują potrzebę testowania brył samolotów i pojazdów w tunelach aerodynamicznych.

Producenci opon wykorzystują silniki fizyczne do symulowania efektywności odprowadzania wody spod kół samochodów w trakcie jazdy po mokrym podłożu, oraz ogólną przyczepność opon na różnych nawierzchniach: asfalcie, żwirze, piasku czy

betonie. Producenci elektroniki używają symulacji mechaniki płynów do badania przepływów powietrza wewnątrz projektowanych przez nich obudów komputerów, a producenci aut wykorzystując metodę elementów skończonych są w stanie symulować i badać szczelność dźwiękową kabin projektowanych przez siebie samochodów. Silniki fizyczne znajdują także zastosowanie w dziedzianach związanych z rzeczwistością rozszerzoną, czyli tam gdzie rzeczywostość spotyka się z rzeczywiśtością wirtualną [9] [10] oraz w dziedzinach medycznych, dla przykładu w symulatorach operacji chirurgicznych [11].

2.2. Silniki stosowane w grach i filmach

Silniki stosowane w grach nie muszą być tak dokładne ja te stosowane do symulacji naukowych. Ich głównym zadaniem jest nadanie elementom gry własności, które przekonają gracza, że świat w którym przyszło mu się poruszać rządzi się tymi samymi prawami co rzeczywistość która go otacza. Dlatego głównym czynnikiem istotnym z punktu widzenia developera gier komputerowych jest szybkość z jaką silnik fizyki potrafi symulować modelowaną rzeczywistość.

Obecnie na rynku gier komputerowych można wyróżnić kilka głównych silników fizyki, które są chętnie stosowane przez producentów zarówno gier klasy AAA jak i domorosłych deweloperów. Poniżej wymieniono i krótko scharakteryzowano najpopularniejsze z nich.

2.2.1. Silnik Nvidia PhysX

Rysunek 2.2.1.1.: Logo silnika Nvidia PhysX. Źródło: http://physxinfo.com.

Pierwszym i wg aytora tej pracy najciekawszym silnikiem jest PhysX. Stworzony przez firmę NovodeX AG z siedzibą w Szwajcarii, został następnie wykupiony przez firmę Ageia, która postanowiła uczynić z niego podstawę systemu symulacji fizycznych wykorzystujących akceleracje sprzętową. Powstały w ten sposób produkt Ageia PhysX PPU (Ageia PhysX, Physics Processor Unit, PhysX processor) to karta rozszerzeń instalowana w komputerze za pomocą złącza PCI lub PCI-Express posiadająca procesor specjalnie zaprojektowany do wykonywania obliczeń fizycznych. Dzięki przeniesieniu obciążenia wynikającego z obliczania symulacji fizycznych ze standardowego CPU na zewnętrzny procesor możliwe było stworzenie realistycznie wyglądających i super-płynnych interaktywnych animacji fizycznych w grach komputerowych oraz systemach przemysłowych takich jak symulatory lotów, symulatory kolei, symulatory jazdy samochodem. Karta weszła na rynek w 2006 roku, dwa lata później NovodexX został wykupiony przez producenta kart graficznych firmę Nvidia. Wtedy zawieszono rozwój i produkcję dedykowanego silnikowi PhysX sprzętu i skupiono się jedynie na rozwoju oprogramowania odpowiedzialnego za obliczenia fizyczne.

Rysunek 2.2.1.2.: Różne warianty kart do akceleracji symulacji fizycznych produkowanych przez firmę Ageia. Od lewej: ASUS PhysX P1, BFG PhysX, ELSA Phynite X100, DELL E1. Źródło: http://physxinfo.com/wiki/Ageia_PhysX_PPU.

Nowy silnik Nvidii powstały na bazie oprogramowania Ageia został przemianowany na Nvidia PhysX i udostępniony do wolnego i darmowego użytkowania w niekomercyjnych zastosowaniach. W celu wykorzystania komercyjnego należy wykupić odpowiednią licencję (koszt rzędu 750 000 dolarów).

Nvidia nie porzuciła całkowicie pomysłu wykorzystania dedykowanego sprzętu do przyspieszenia obliczeń wykonywanych przez silnik. Jako producent procesorów graficznych zadbała o możliwość wykorzystania przez silnik mocy obliczeniowej najnowszych układów graficznych, których zrównoleglona architektura świetnie się do tego celu nadaje (dla przykładu w pracy [12] wykrzystano akceletację sprzętową GPU do stworzenia systemu wykrywania oszustw w grach mutliplayer). Dodatkowo Nvidia PhysX SDK 2.0 (PhysX Software Development Kit, versions 2.3.0 - 2.8.4) wzbogacone zostało o zestaw specjalnych narzędzi ułatwiających pracę z silnikiem, w których skład wchodzą:

- DCC plug-ins: Digital Content Creation plug-ins, czyli wtyczki rozszerzające właściwości programów do tworzenia grafiki 3D o możliwość tworzenia geometrii zrozumiałej dla silnika PhysX. Wtyczki dostępne są dla programów Autodesk 3ds Max oraz Autodesk Maya. Inny bardzo popularny program Autodesk Softimage ma wbudowane możliwości jakie dają pluginy DCC.

- Profilery: narzędzia do badania wydajności silnika w pracy z konkretnymi danymi. Pozwalają w prosty sposób znaleźć słabe punkty projektowanych scen 3D przez prezentację czasów wykonywania obliczeń w zależności od struktury modeli 3D oraz ogólnego stopnia skomplikowania badanej sceny fizycznej.

- Debuggery: narzędzia dla programistów pozwalające na dokładne przyglądanie się obliczeniom wykonywanym wewnątrz silnika fizycznego w trakcie jego działania.

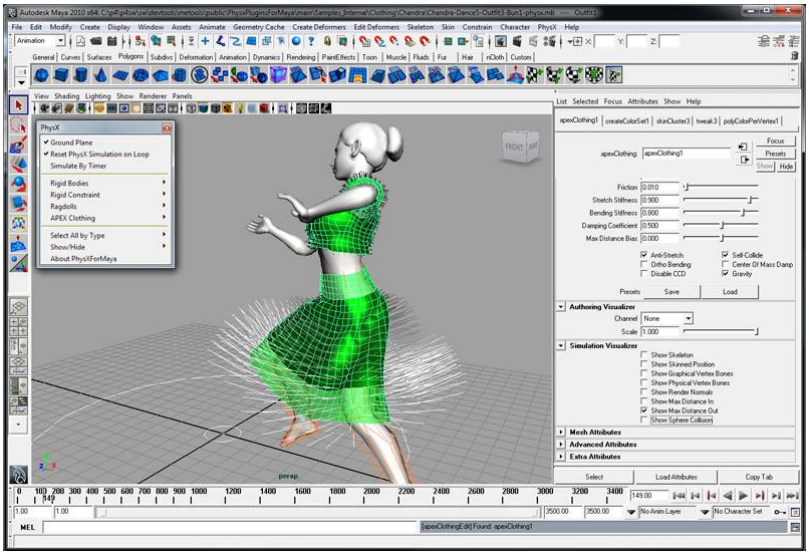
Głównymi zaletami silnika Nvidia PhysX jest wielowątkowa architektura oraz dostępność na wszystkich ważnych platformach: Microsoft Windows, Mac OS X, Linux, PlayStation 3, Xbox 360 oraz Wii. Firma Nvidia na bazie silnika PhyxS stworzyła wieloplatformowy framework APEX zapewniający artystom wysokopoziomowe API do pracy silnikiem. Artysta nie musi w ogóle znać się na programowaniu. Dzięki specjalnemu oprogramowaniu PhysXLab wykorzystującemu APEX możliwe jest manualne projektowanie własności fizycznych tworzonych modeli 3D. W skład APEX wchodzą następujące moduły:

- APEX Destruction – odpowiedzialny modelowanie możliwych do zniszczenia i odkształcenia brył
- APEX Clothing – odpowiedzialny za modelowanie tkanin
- APEX Turbulance – odpowiedzialny za tworzenie dynamicznych efektów dymu, wody, ognia itp.

Więcej informacji o APEX znaleźć można na stronie projektu [13].

Przykład wykorzystania silnika PhysX do symulowania rzeczywistości wirtualnej znaleźć można w pozycji [14].

Na kolejnych stronach zaprezentowano kilka rysunków pokazujących możliwości silnika Nvidia PhysX.



Rysunek 2.2.1.3.: Zrzut ekranu wtyczki rozszerzeń Nvidia PhysX DCC plugin do tworzenia modeli 3D w programie Autodesk Maya. Źródło: http://www.nvidia.com/object/physx_dcc_plugins.html.

Rysunek 2.2.1.4.: Symulacja sukienki przy pomocy silnika fizycznego Nvidia PhysX. Ruchy sukienki modelowane są przez aktywne fizycznie płótno, dzięki czemu artysta tworzący symulację musi dbać jedynie o animację ruchów modelki, pomijając bardzo skomplikowaną animację materiału z jakiego sukienka jest wykonana. Źródło: http://bi.gazeta.pl/im/5/6161/z6161035Q.jpg.

Rysunek 2.2.1.5.: Symulacja dużej liczby sześciennych klocków przez silnik Nvidia PhysX. Źródło: http://www.fmix.pl/album/73104/galeria/1.

2.2.2. Silnik Havok

Rysunek 2.2.2.1.: Logo silnika Havok. Źródło: http://www.havok.com.

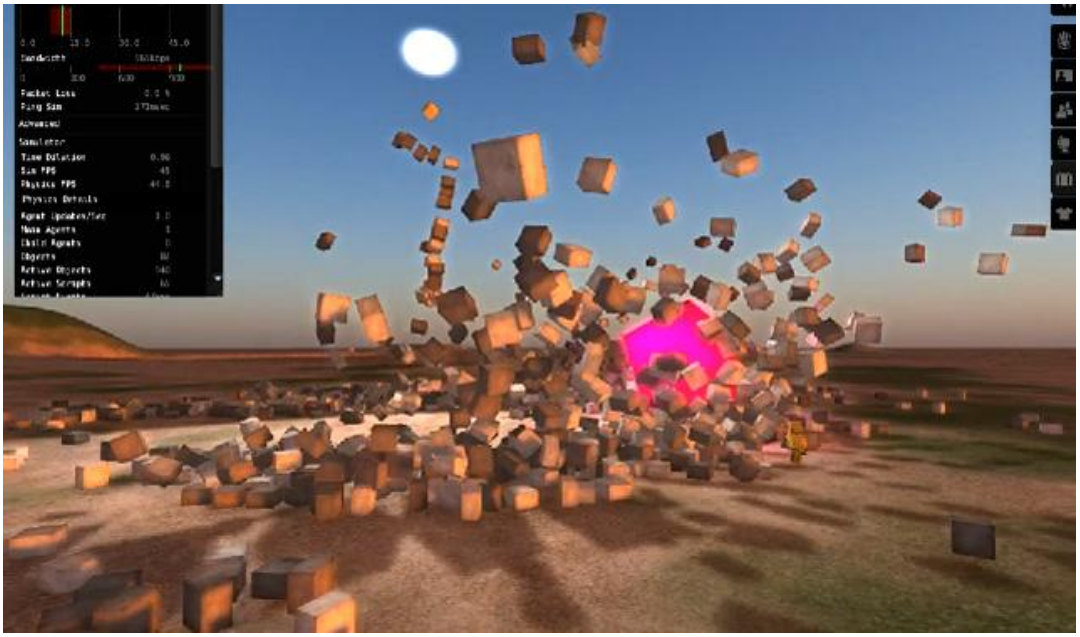
Drugim silnikiem w tym porównaniu jest Havok - silnik fizyki stworzony przez Irlandzką firmę o tej samej nazwie. Havok został specjalnie zaprojektowany z myślą o grach komputerowych i interaktywnych symulacjach. Potrafi animować bryły sztywne w trzech wymiarach oraz potrafi tworzyć wiele różnych rodzajów więzów między nimi. Dzięki temu znajduje szerokie zastosowanie w branży gier komputerowych. Jednym z popularniejszych zastosowań tego silnika jest animowanie tak zwanych ragdoll'i czyli marionetek, których struktura odpowiada budowie ludzkiego ciała.

Oprócz bogatego zbioru różnego rodzaju połączeń między bryłami sztywnymi (tzw. jointów), Havok posiada zoptymalizowany moduł odpowiedzialny za wykrywanie kolizji. System CCD (Continous Collision Detection [15]) pozwala wykrywać kolizje między dwoma poruszającymi się z dużą prędkością ciałami, co daje ogromne możliwości w przypadku tworzenia wszelkiego rodzaju gier strzelanek oraz symulacji militarnych, w których pociski muszą być symulowane zgodnie z zasadami fizyki.

Havok w pełni wspiera wielowątkowość co zapewnia mu świetne osiągi na wielordzeniowych procesorach jednak nie wykorzystuje akceleracji sprzętowej oferowanej przez karty graficzne. Dostępny jest praktycznie na wszystkie istotne platformy: Xbox 360™, PlayStation®3, PC, PlayStation Vita®, Wii™, Wii U™, Windows 8, Android™, iOS, Apple Mac OS oraz Linux. Pierwsza wersja SDK 1.0 przedstawiona została na konferencji poświęconej tworzeniu gier komputerowych Game Developers Conference [16] w 2000 roku. Od tamtej pory Havok przystosował swój produkt do pracy z wieloma platformami, między innymi mobilną platformą Sony Xperia Play.

Firma Havok razem z silnikiem fizyki zapewnia dostęp do bardzo rozbudowanego zbioru narzędzi ułatwiającego pracę z modelami 3D obsługiwanymi przez silnik. Dla artystów dostępne są wtyczki rozszerzające możliwości programów takich jak Autodesk Maya oraz Adobe Director's Shockwave.

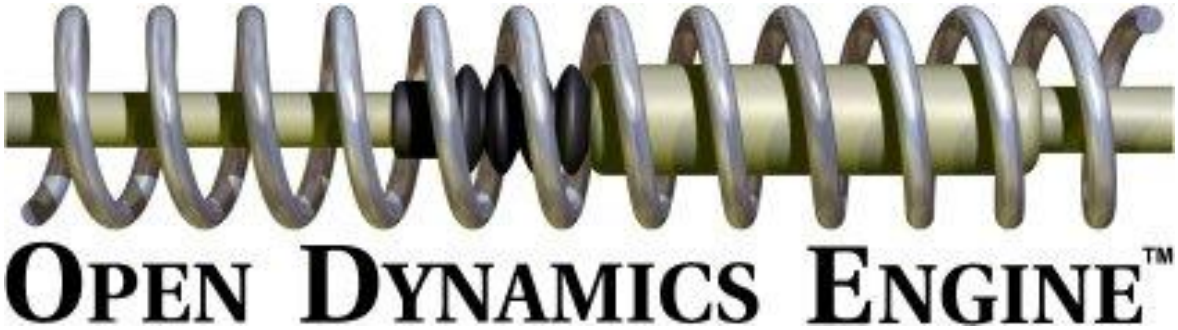
Na następnej stronie przedstawiono kilka grafik prezentujących możliwości silnika fizycznego Havok.



Rysunek 2.2.2.2.: Havok uruchamia mechanizmy fizyczne symulujące świat popularnej gry internetowej Second Life. Źródło: http://www.secondlifeupdate.com/wp-content/uploads/2010/04/Havok7SecondLife.jpg.

Rysunek 2.2.2.3.: Havok uruchomiony na platformie mobilnej Sony Xperia Play. Źródło: http://www.dfxnetwork.org/wp-content/uploads/2011/03/NewImage.jpg.

2.2.3. Silnik Open Dynamics Engine

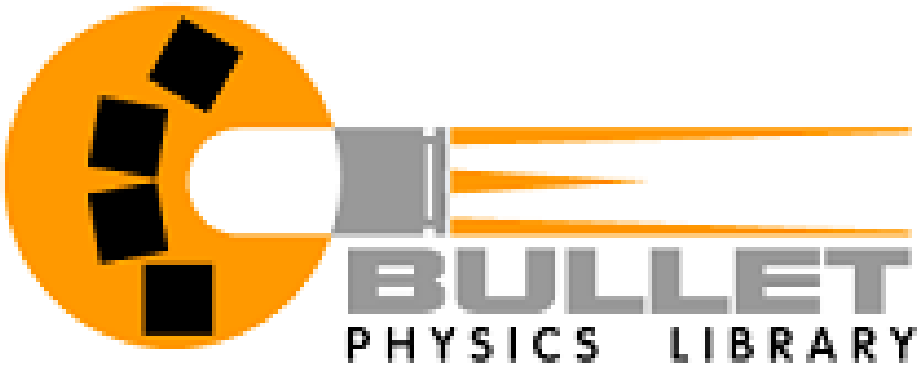


Rysunek 2.2.3.1.: Logo silnika ODE – Open Dynamics Engine. Źródło: http://www.ode.org/.

Open Dynamics Engine jest w tym zestawieniu pierwszym silnikiem o otwartym źródle. Głównym jego twórcą jest Russel Smith, któremu w rozbudowie silnika pomaga mała armia niezależnych programistów z całego świata. ODE to wydajna biblioteka wspierająca symulację fizyczną brył sztywnych w przestrzeni 3D. Dużą zaletą tego silnika jest bogaty zbiór mechanizmów służących do łączenia brył (jointów) oraz wsparcie dla kinematyki odwrotnej. Dzięki temu ODE świetnie nadaje się do modelowania wszelkiego rodzaju pojazdów, maszyn kroczących oraz robotów przemysłowych. Posiada wbudowaną obsługę kolizji pomiędzy elementami sceny. Został specjalnie zaprojektowany do zastosowań w interaktywnych symulacjach czasu rzeczywistego. Ogólnie jest bardzo dobry do zastosowań w których ma się do czynienia z ogromną liczbą szybko poruszających się obiektów, jak również do symulacji bardziej statycznych scen składających się z obiektów tworzących stosy. Wszystkie te cechy czynią go świetnym kandydatem do wykorzystania w grach FPS (First Person Shooter) co zostało dowiedzione przez wykorzystanie ODE do produkcji tytułów takich jak: BloodRayne 2, Call of Juarez, S.T.A.L.K.E.R, Titan Quest, World of Goo, X-Moto, Hell Forces.

Rysunek 2.2.3.2.: Zrzut ekranu z gry Hell Forces, w której symulacje fizyczne obsługuje Open Dynamics Engine. Źdódło: http://www.ode.org/slides/parc/dynamics.pdf.

2.2.4. Silnik Bullet



Rysunek 2.2.4.1.: Logo silnika Bullet Physics Library. Źdódło: http://bulletphysics.org/.

Bullet Physics Library to kolejny w zestawienu silnik fizyki 3D o otwartych źródłach. Jego głównym autorem jest Erwin Coumans, programista który pracował dla fimry Sony Computer Entertainment US R&D, aktualnie pracownik AMD.

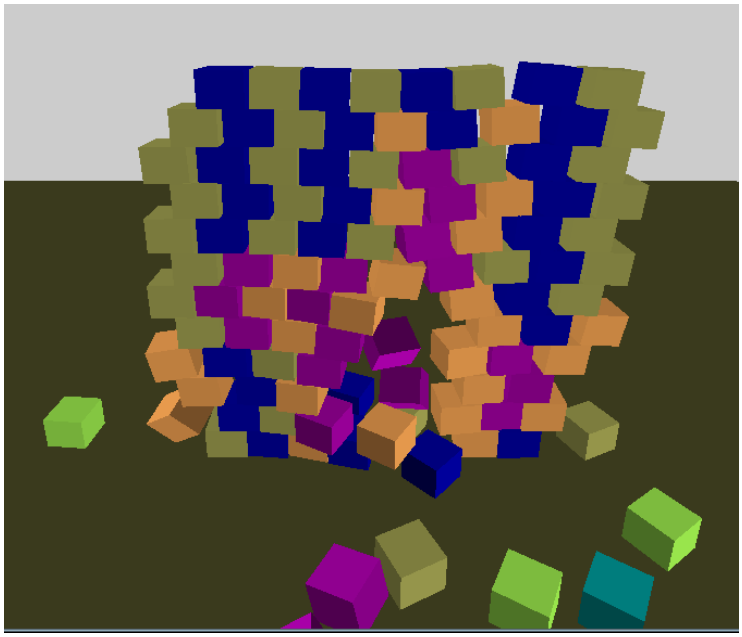
Bullet to bardzo popularny silnik wykorzystywany w profesjonalnej produkcji gier oraz filmów. Jego ogromną zaletą jest licencja zlib license [17], która pozwala na darmowe wykorzystanie silnika w dowolnym projekcie zarówno niekomercyjnym jak i komercyjnym. Bullet został zintegrowany z wieloma zaawansowanymi aplikacjami do tworzenia grafiki oraz efektów 3D. Poniżej znajduje się lista popularnych narzędzi wykorzystujących ten silnik:

- Blender – profesjonalne darmowe narzędzie do tworzenia grafiki 3D
- Carrara Pro – zestaw narzędzi do tworzenia i edycji grafiki 3D
- Cheetah3D – zestaw narzędzi do tworzenia i edycji grafiki 3D dla systemu Apple Max OS X

Bullet swoją popularność zawdzięcza szerokiemu spektrum możliwych do uruchomienia symulacji. Potrafi symulować bryły sztywne oraz miękkie (tzw. soft-bodies). Posiada zaawansowany moduł wykrywania kolizji, łącznie z obsługą kolizji szybko poruszających się obiektów CCD. Jego możliwości pozwalają na tworzenie odkształcalnych obiektów, lin oraz symulacji płacht materiału. Ponadto pozwala tworzyć zaawansowane połączenia między wszystkimi symulowanymi elementami

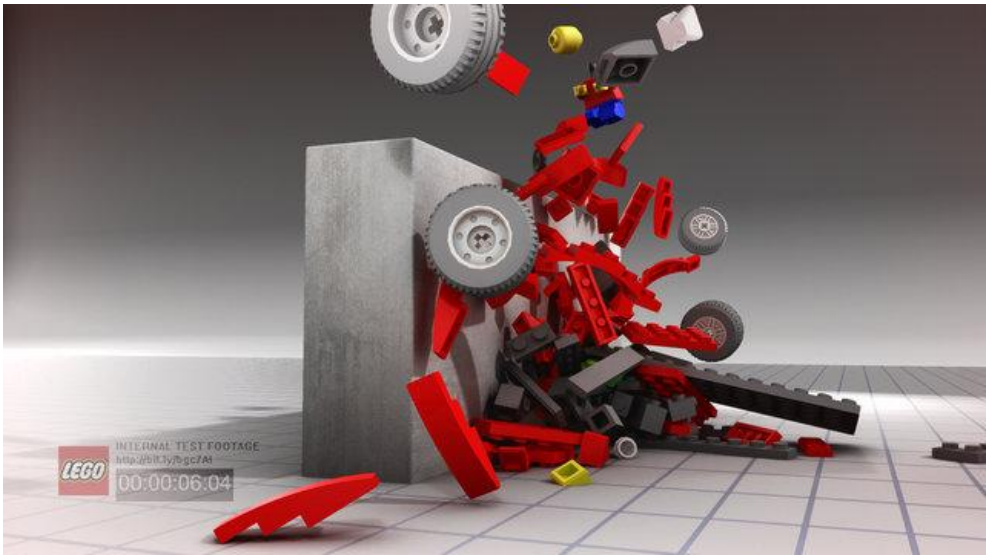
(tzw. joints). Najnowsza wersja Bullet'a została przystosowana do pracy z układami kart graficznych GPU, co pozwala na zrównoleglenie obliczeń symulacji. Dzięki temu Bullet dołączył do grona silników fizyki posiadających możliwość akceleracji sprzętowej, bardzo popularnej w dobie ultraszybkich kart graficznych.
Bullet został wykorzystany przy produkcji następujących filmów oraz gier: 2012 (katastroficzny), Hancock, Bolt, The A-Team, Shrek 4, Grand Theft Auto IV.

Poniżej znajdują się grafiki prezentujące symulacje uruchamiane przez silnik Bullet.



Rysunek 2.2.4.2.: Ściana z klocków symulowana przez silnik Bullet. Stabilność konstrukcji stosowych jest bardzo trudna do uzyskania. Źródło: http://en.wikipedia.org/wiki/Bullet_(software).

Rysunek 2.2.4.3.: Bullet odpowiada za symulacje fizyczne w nowej grze Raptide GP wydanej przez studio Vector Unit. Źródło: http://bulletphysics.org/.



Rysunek 2.2.4.4.: Symulacja crash-testu samochodu z klocków lego stworzona w programie Maya Fuild i uruchomiona przez silnik Bullet. Źródło: http://vimeo.com/16301140.

2.2.5. Silnik Newton

Rysunen 2.2.5.1.: Logo silnika fizycznego Newton Dynamics. Źródło: http://newtondynamics.com/.

Newton Dynamics to kolejny silnik fizyki 3D o otwartych źródłach. Jego twórcami są Julio Jerez oraz Alain Suero. Wykorzystywany jest głównie w niekomercyjnych projektach amatorskich. Jego zaletą jest proste i bardzo dobrze udokumentowane API oraz baza wiedzy (forum) tworzona przez użytkowników.

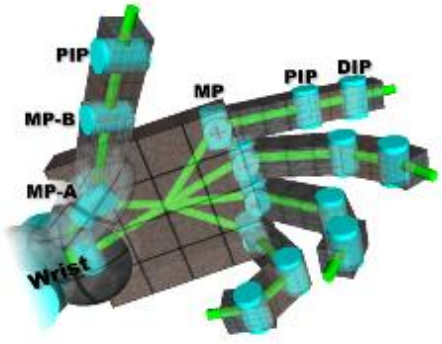
Newton potrafi symulować bryły sztywne w przestrzeni 3D oraz szeroki wachlarz połączeń pomiędzy nimi (tzw. jointy). Dodatkowo posiada moduł odpowiedzialny za zarządzanie scenami symulacji co znacznie ułatwia pracę z silnikiem. Wbudowane mechanizmy pozwalają na szybką prezentację sceny przy wykorzystaniu API OpenGL, które zapewnia szybki rendering obiektów z wykorzystaniu akceleracji sprzętowej.

Posiada także wbudowany moduł wykrywania kolizji, jednak brakuje w nim obsługi szybko poruszających się obiektów (CCD). Zaletą silnika jest wykorzystanie deterministycznego solvera nie bazującego na tradycyjnych metodach LCP (Linear Complementarity Problem [18] [19] [20]), dzięki czemu oprócz zastosowań typowych dla gier komputerowych nadaje się również do poważnych symulacji naukowych. Więcej o LCP można przeczytać w następujących pozycjach [21] [22] [23] [24].

Ciekawym projektem, w którym Newton odpowiada za symulację świata i interakcję z użytkownikiem jest projekt gry, której świat jest realnie odczuwalny dla gracza. Poniżej przedstawiono grafikę prezentującą manipulator służący do przekazywania realnych odczuć z wirtualnego świata gry [25].

Rysunek 2.2.5.2.: HaptiCast – projekt gry, w której gracz może odczuwać właściwości fizyczne napotykanych przedmiotów. Źródło: http://newtondynamics.com/images/HaptiCast.jpg.

Newton został także wykorzystany w projekcie modelującym ludzką dłoń. Wykorzystany został do uruchomienia symulacji fizycznej manipulatora przypominającego dłoń i posłużył do badań nad jego chwytnością [26].



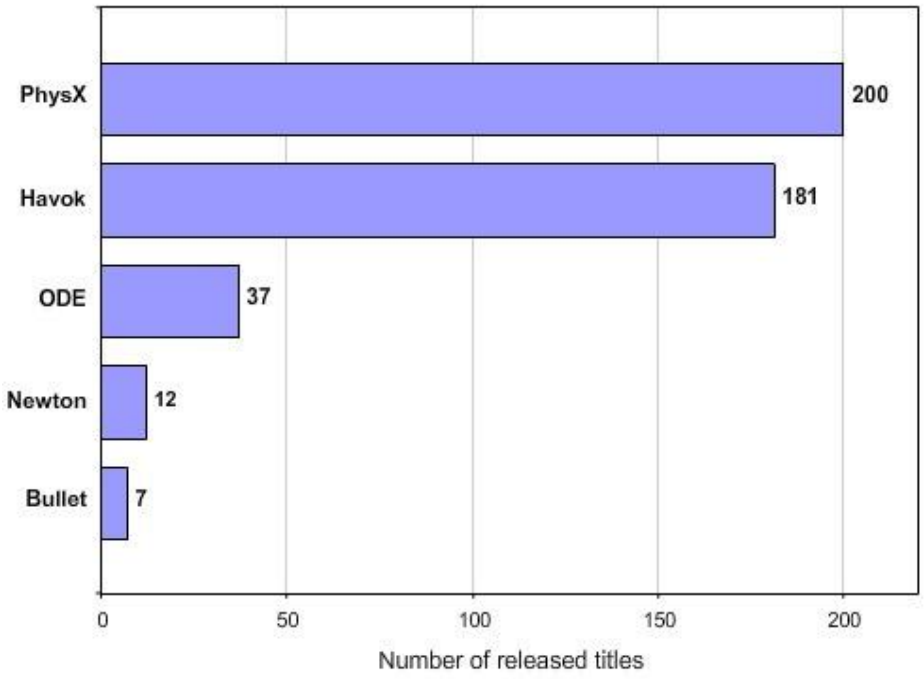
Rysunek 2.2.5.3.: Zrzut ekranu z projektu badającego chwytność manipulatora modelującego ludzką dłoń. Newton Dynamics dzięki swojemu niedeterministycznemu solverowi świetnie nadaje się do tego typu zastosowań. Źródło: http://laral.istc.cnr.it/esm/arm-grasping/.

2.3. Porównanie silników PhysX, Havok i ODE

Wszystkie opisane w poprzedniej części silniki świetnie nadają się do zastosowania w grach i interaktywnych symulacjach fizycznych. Trudno jest wyłonić z pośród nich silnik najlepszy. Dlatego zdecydowano się przytoczyć ciekawe porównanie wykonane przez redaktora Zogrim opublikowane na łamach internetowego serwisu PhysXinfo.com porównujące opisane silniki ze względu na liczbę gier wypuszczonych latach 2005-2009, które korzystały z danego silnika [27].

Autor porównania zauważa, że tak dobrany wskaźnik jest sprawiedliwym porównaniem tego jakie rozwiązania przyjmują się na rynku gier komputerowych. Szczegółowe porównania silników ze względu na zestaw cech, wydajność w konkretnych zastosowaniach czy rodzaje algorytmów wykorzystanych do implementacji solverów są ciekawe z inżynierskiego punktu widzenia, ale nie pomagają znaleźć odpowiedzi na pytanie, który silnik najlepiej nadaje się do tworzenia tytułów zyskujących dużą popularność wśród graczy.

Pierwszym porównaniem jest zestawienie liczby wydanych tytułów oraz badanych przedziałów czasowych (lat) wykorzystujących każdy z silników:



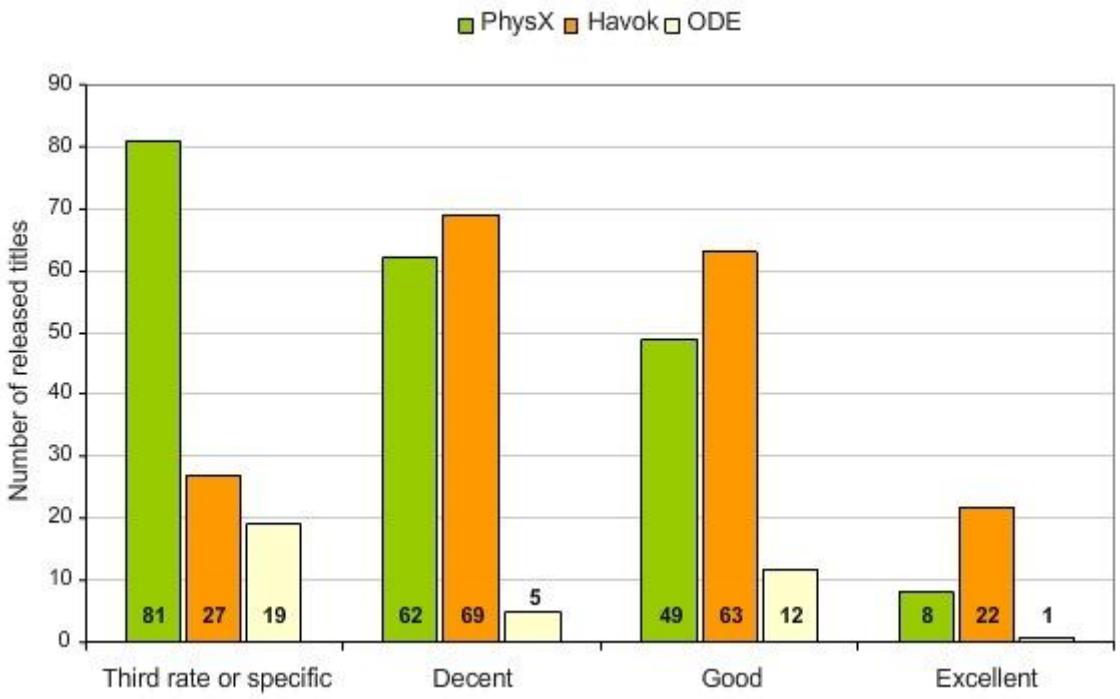
Rysunek 2.3.1.: Prezentuje liczbę gier wydanych przy użyciu każdego z silników w przedziale lat 2005-2009. Źródło: http://physxinfo.com/.

Widać, że prym wiodą trzy silniki: PhysX, Havok oraz ODE. Dwa pozostałe: Newton oraz Bullet znacząco odstają od czołowej trójki, dlatego Zogrim zdecydował się pominąć je w dalszych porównaniach.

Surowe liczby wypuszczonych tytułów nie pokażą jednak szczegółów natury opisanych silników. Kolejne porównanie dotyczy jakości wydanych tytułów. Jakość mierzona jest za pomocą meta-punktów przyznawanych każdemu z tytułów na podstawie średniej ocen znanych krytyków piszących recenzje gier na łamach czasopism i internetu. Oceny takie zbiera i liczy serwis internetowy Metacritic [28].

Tytuły podzielono na kategorie ze względu na liczbę meta-punktów przyznaną przez serwis. Według jakości wyróżniono tytuły:

- Trzeciej kategorii (meta-punkty <50 lub tytuł nie występuje w bazie serwisu)
- Przyzwoite (meta-punkty pomiędzy 50 a 70)
- Dobre (meta-punkty pomiędzy 71 a 85)
- Znakomite (meta-puknty powyżej 85)

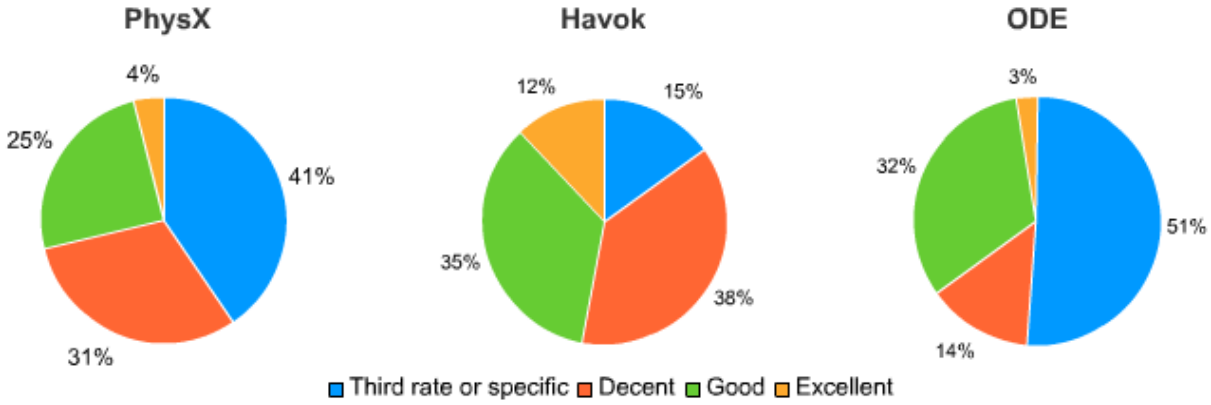


Rysunek 2.3.2.: Prezentuje liczbę gier danej kategorii wydanych przy użyciu każdego z silników w przedziale lat 2005-2009. Źródło: http://physxinfo.com/.

Autor zauważa, że PhysX dzięki otwartej licencji i bogatemu zestawowi funkcji cieszy się uwagą małych teamów developerskich, które stanowią główną część rynku produkcji gier komputerowych. Nadmienia, że PhysX jest głównie wykorzystywany przez zespoły rosyjskie oraz koreańskie, oraz że stanowi domyślny silnik fizyki bardzo popularnego środowiska do tworzenia gier Unreal Engine 3.

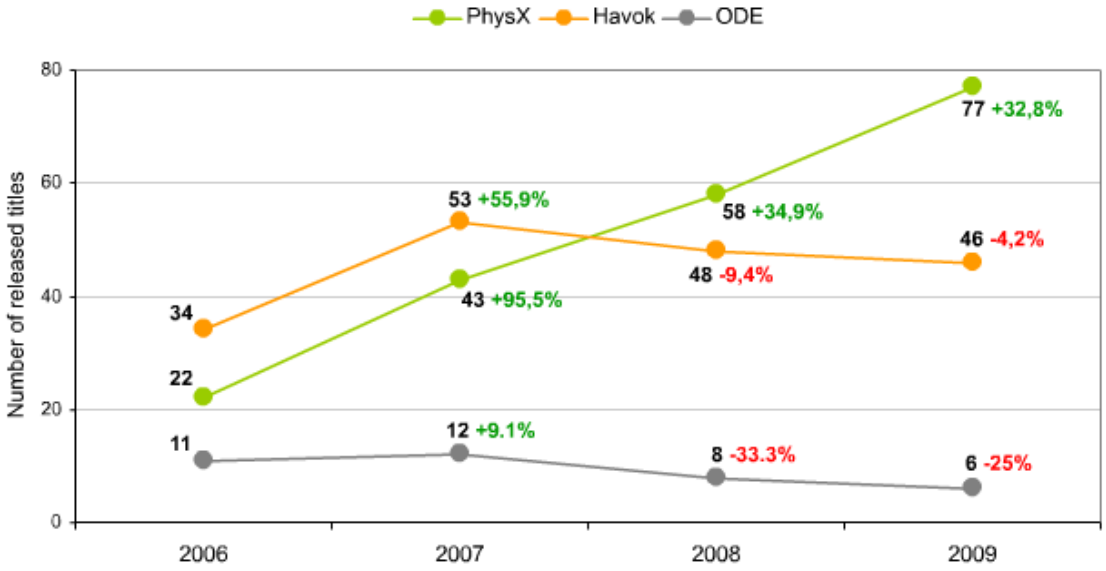
Według oceny autora Havok jest najlepszym rozwiązaniem dla gier klasy AAA. Bogaty zbiór narzędzi użytkowych, zorientowanie głównie na konsole oraz najlepsza dostępna obsługa i wsparcie klienta stawiają go na tej własnie pozycji. Zogrim wymienia tytuły gier, które świadczą o prawdziwości tej tezy: Uncharted 2: Among Thieves oraz Killzone 2.

ODE nie wypada w tym zestawieniu najlepiej nawet pomimo współtworzenia bardzo dobrych tytułów takich jak S.T.A.L.K.E.R. czy Word of Goo, gry która jako jedyna posiada ocenę powyżej 85 meta-punktów.



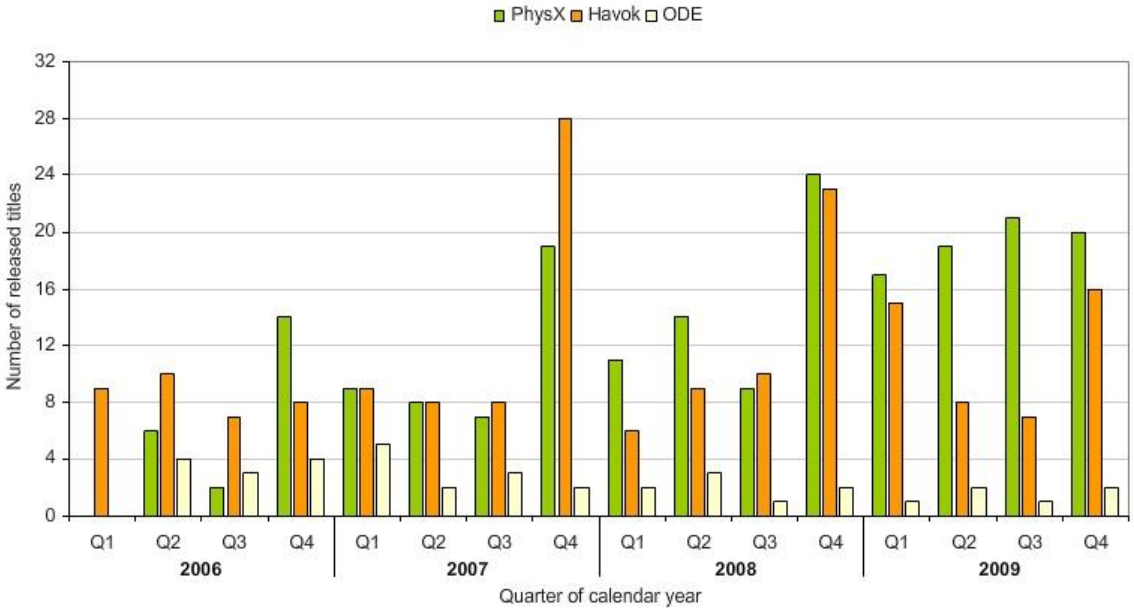
Rysunek 2.3.3.: Prezentuje procentowy rozkład jakości gier, wśród badanych tytułów.
Źródło: http://physxinfo.com/.

Wybór drugiego sposobu na porównanie silników przez Zogrim'a pada na badanie dynamiki zmiany liczby gier wypuszczanych każdego roku przy użyciu każdego z silników. Lata które brane są pod uwagę to okres pomiędzy rokiem 2006 a 2009. Ten konkretny przedział czasu został tak zdefiniowany ponieważ przed rokiem 2006 nie było wydanych żadnych gier na silniku PhysX.



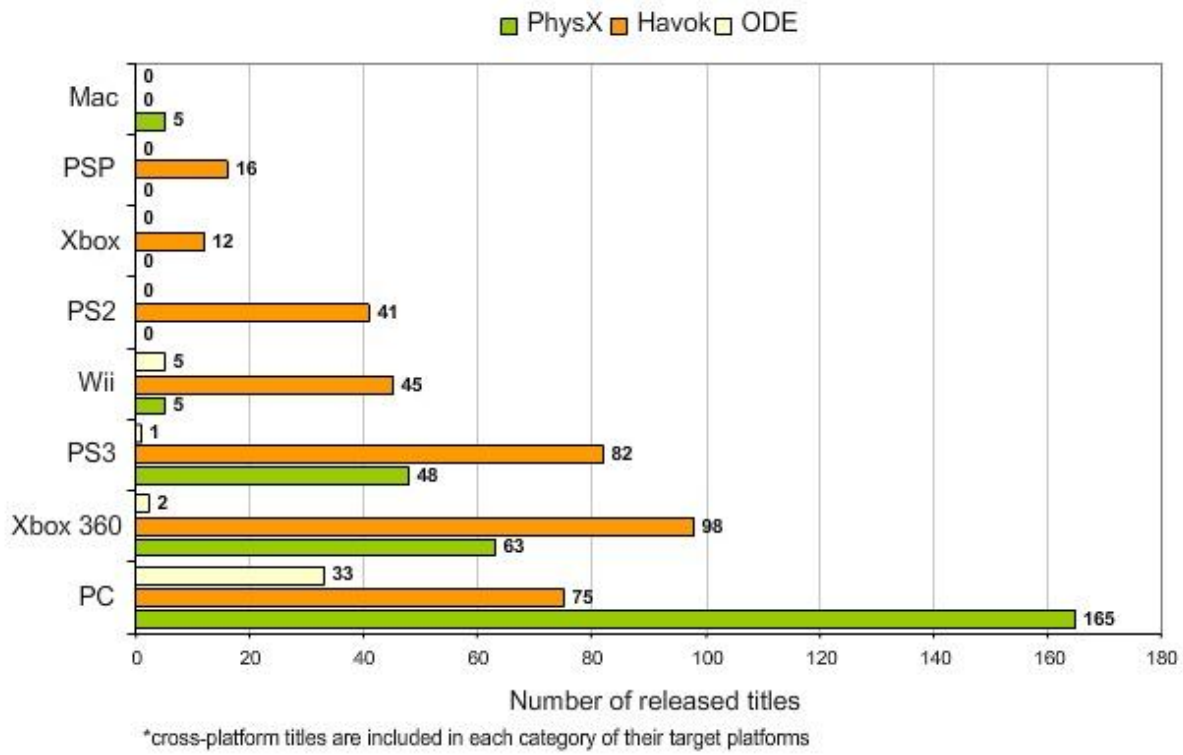
Rysunek 2.3.4.: Prezentuje dynamikę zmian liczby produkowanych gier przy użyciu każdego z porównywanych silników. Na wykresie zaprezentowano liczby określające ilość wydanych tytułów na rok. Źródło: http://physxinfo.com/.

Największy przyrost liczby tytułów odnotował PhysX, przy czym należy zaznaczyć że w tym czasie nie następowała żadna rozbudowa silnika. Havok zachował stabilny trend z lekkim spadkiem w ostatnich latach badanego okresu. Jak wynika z wykresu udział silnika ODE w rynku gier jest znikomy. Developerzy niechętnie wybierają tę bibliotekę, być może dlatego że rozwój ODE został zawieszony w 2007 roku. Poniższy wykres prezentuje te same dane bardziej z większą dokładnością.



Rysunek 2.3.5.: Prezentuje dynamikę znian liczby produkwanych gier przy użyciu każdego z porównywanych silników.Tym razem z dokładnością do kwartałów. Źródło: http://physxinfo.com/.

Ostatnie porównanie wykonane przez Zogrim'a dotyczy platform docelowych, na które były wydawane tytuły wykorzystujące porównywane silniki. Tytuły klasy AAA z reguły celują w rynek konsol do gier, przez co mogą sobie pozwolić na droższe rozwiązania takie jak Havok, podczas gdy twórcy gir na komputery klasy PC z reguły wybierają rozwiązania darmowe.



Rysunek 2.3.6.: Prezentuję liczbę wypuszczonych tytułów na każdą z dostępnych platform. Źródło: http://physxinfo.com/.

Wyraźnie widać że PhysX dominuje na rynku komputerów osobistych, Havok na rynku konsolowym oraz ponadto pojawia się na innych platformach: Xbox, PS2 a także PSP. Autor zauważa że SDK PhysX'a na konsole dostępne było już w 2005 roku i wykorzystywały go wtedy tytuły wydawane na platformy PS3 oraz Wii. Silnik ODE znalazł zastosowanie przy produkcji kilu tytułów na platformę Wii oraz PS3 jednak według autora nie miało to większego znaczenia i ODE dalej pozostaje na końcu zestawienia.

Wykres na następnej stronie prezentuje procentowy relatywny rozkład testowanych tytułów względem ich platform docelowych.

Rysunek 2.3.7.: Prezentuję liczbę wypuszczonych tytułów na każdą z dostępnych platform, tym razem zilustrowano procentowy udział każdego z silników względem dostępnych platform.
Źródło: http://physxinfo.com/.

Konkluzja Zogrim'a jest jednoznaczna: Havok wiedzie prym w sektorze gier z najwyższej półki: klasy AAA. PhysX dzięki wykorzystaniu akceleracji sprzętowej świetnie nadaje się do tytułów mniej skomplikowanych i o niższym budżecie jednak należących do najliczniejszej grupy tytułów ukazujących się na rynku. Sytuacja ODE jest bardzo słaba, jest wykorzystywany głównie do projektów amatorskich z kilkoma nielicznymi odstępstwami od tej zasady.

Autor tej pracy magisterskiej zdecydował się jednak na implementację własnego silnika fizyki. Wpłynęły na to dwa główne czynniki: praca z silnikami takimi jak PhysX czy Havok wymaga długiego czasu nauki API każdego z nich oraz to, że praca nad własnym silnikiem jest bardzo interesującym zagadnieniem. Cała wiedza numeryczno-matematyczna oraz fizyczna zdobyta w trakcie implementacji własnego silnika jest bardzo przydatna programistom różnych specjalności, dlatego autor postanowił od podstaw zbudować własny i prosty dwuwymiarowy silnik fizyki z obsługą brył sztywnych oraz miękkich (ponieważ obserwacja ich zachowania przysparza obserwatorowi wielu pozytywnych bodźców wizualnych).

2.4. Silnik fizyki 2D

Silnik fizyki posiada listy aktorów i aktuatorów składających się na scenę symulacji. Każda z tych list posiada pewną liczbę elementów tego samego rodzaju, na przykład ciała miękkie, cząsteczki, sprężyny itp.

Procedury silnika iterują po tych listach w kolejnych etapach generacji pojedynczego kroku symulacji wykonując niezbędne obliczenia.

W kolejności wykonywania, odpowiednie etapy to:

1. Przygotowanie elementów sceny do obliczenia kolejnej ramki animacji. Polega to na wyzerowaniu lokalnych wypadkowych sił działających na aktorów, które zostały obliczone w poprzedniej ramce symulacji.

2. Obliczenie aktualnych sił działających na aktorów sceny. Wymaga to iteracji po wszystkich aktuatorach sceny i wykonania ich kodu, który przyłoży nowo obliczone siły do powiązanych z nimi aktorów.

3. Obliczenie nowych położeń (i orientacji) aktorów symulacji (całkowanie równań ruchu).

4. Wykrycie kolizji między aktorami.

5. Obliczenie odpowiedzi na wykryte kolizje

Z reguły na pojedynczy krok symulacji fizycznej przypada jedna ramka animacji, jednak w celu uzyskania większej liczby klatek na sekundę można odrysowywać scenę nie co jeden krok silnika fizycznego, ale np. co trzy kroki. Ten trik działa, ponieważ silnik graficzny potrzebuje więcej czasu na odrysowanie pojedynczej klatki animacji niż silnik fizyki na wykonanie pojedynczego kroku czasowego. Dlatego odrysowując co trzeci krok silnika fizycznego, na rendering sceny zużywamy jedynie 1/3 czasu normalnie wykorzystywanego na tworzenie grafiki.

2.4.1. Elementy składowe

Rodzaje elementów jakie silnik fizyki potrafi animować decydują o jego przydatności w konkretnych zastosowaniach. W przypadku dwuwymiarowych gier komputerowych szerokie zastosowanie znajdują cząsteczki, które świetnie modelują dym, ogień, iskry bądź bąble powietrza uciekające ku powierzchni płynu. Bryły sztywne modelujące elementy pojazdów, przeciwników oraz terenu. Bryły miękkie pozwalające symulować elementy, których kształt zmienia się wraz z upływem czasu: balony, opony, komórki organiczne, miękkie piłki.

Implementacja silnika stworzona na potrzeby tej pracy potrafi obsługiwać następujące rodzaje obiektów:

- Aktorzy
    - Cząsteczka (mass/particle)
    - Bryła miękka (soft-body)
    - Bryła sztywna (rigid-body)
- Aktuatory
    - Siła
    - Pole siłowe
    - Moment obrotowy
    - Sprężyna
    - Sprężyna obrotowa

2.4.2. Dynamika punktu i symulacja ruchu

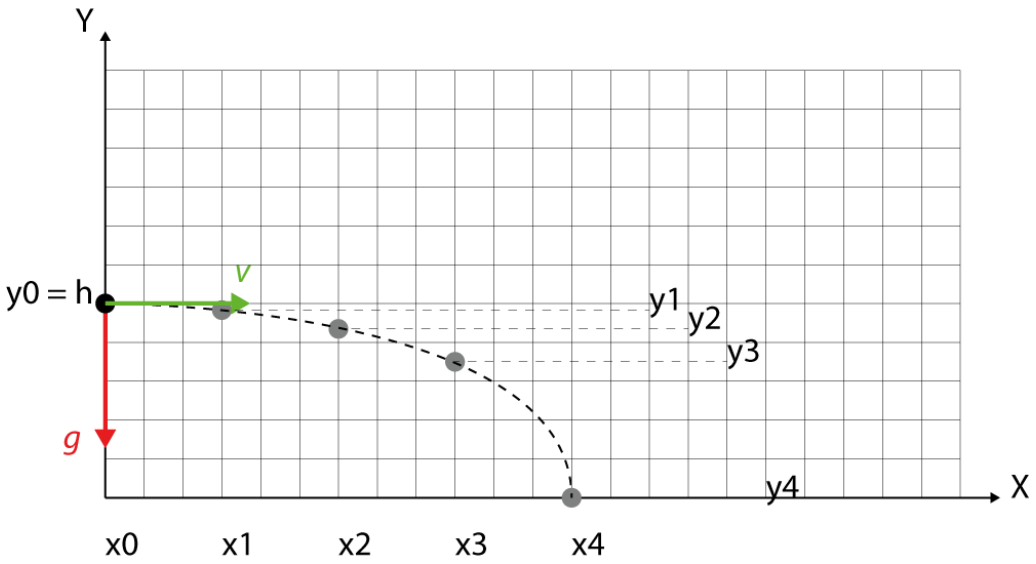
Na przykładzie ruchu pojedynczej cząsteczki przedstawiony rozstanie mechanizm obliczania nowego położenia cząsteczki w kolejnych krokach czasowych animacji. Cząsteczka modelowana jest przez masę punktową. To znaczy, że nie posiada żadnych wymiarów i opisana jest jedynie przez wartość masy oraz położenie w przestrzeni. O zmianie położenia cząsteczki w czasie trwania symulacji decyduje jej prędkość. Dzięki prędkości można obliczyć drogę jaką cząsteczka przebyła względem położenia z poprzedniego kroku czasowego. Oprócz położenia i prędkości potrzebujemy znać przyspieszenie cząsteczki. Przyspieszenie zmienia wartość i kierunek prędkości, dzięki czemu tor ruchu cząsteczki może być zakrzywiany, oraz może ona przyspieszać lub zwalniać. Dodatkowo chcemy wiedzieć jak trudno jest cząsteczkę wprawić w ruch oraz jak trudno jest odchylić jego tor, dlatego potrzebujemy znać wcześniej wspomnianą masę cząsteczki.

Biorąc pod uwagę powyższy opis modelu cząsteczki, do jej opisu fizycznego wykorzystane zostaną następujące elementy:

- położenie p
- prędkość v
- przyspieszenie a
- masa m

Z pierwszej zasady dynamiki Newtona wiadomo, że aby zmienić położenie/prędkość jakiegoś ciała należy przyłożyć do niego siłę. W czasie, gdy siła będzie oddziaływać na ciało, zmianie ulegnie jego prędkość, a za nią zmieni się położenie.

Dla przykładu, poniżej zilustrowano klatki animacji obliczane przez silnik fizki w przypadku symulacji rzutu poziomego. Mamy tu do czynienia z cząsteczką znajdującą się na pewnej wysokości h, na którą oddziałuje pole grawitacyjne jednorodne. Wpływ pola grawitacyjnego oznacza przyłożenie siły do cząsteczki, która zgodnie z drugą zasadą dynamiki, wpływa na wartość jej przyspieszenia, załóżmy że wartość przyspieszenia równa jest g=[0,-10]m/s^2. Poziom ziemi znajduje się na wysokości równej h=0, co ogranicza nasz obszar symulacji do pierwszej ćwiartki układu współrzędnych. Aby cząsteczka nie spadła po prostu w dół, ale wykonała charakterystyczny dla rzutu poziomego lot po paraboli, musimy nadać jej poziomą prędkość początkową v_0. Od wartości tej prędkości zależy dystans jaki przebędzie cząsteczka w kierunku poziomym. Należy też zwrócić uwagę, że ruch względem kierunku poziomego będzie ruchem prostoliniowym jednostajnym, ponieważ jedyne przyspieszenie z jakim mamy w tym wypadku do czynienia działa w kierunku pionowym (grawitacja). Dlatego łatwo zaobserwować, że w kolejnych krokach czasowym cząsteczka przebywa dokładnie ten sam dystans względem osi poziomej.
Początkowy stan sceny symulacji wraz z kolejnymi jej krokami przedstawia poniższy rysunek:



Rysunek 2.4.2.1.: Prezentuję kolejne położenia cząsteczki w ruchu po paraboli spowodowanym rzutem poziomym. Źródło: własne.

W przypadku pokazanym na poprzedniej stronie animacja składa się z 5 klatek. Kolejne położenia cząsteczki opisują wektory $p_i=[x_i,y_i]$, i ϵ <0,4>. Na przykładzie metody Eulera pokazane teraz zostaną niezbędne obliczenia wykonane przez silnik fizyki w celu obliczenia kolejnych pozycji cząsteczki (dokładny opis metody Eulera oraz innych metod całkowania równań ruchu można znaleźć w pozycjach [29] [30] [31]).
Metodę Eulera można przedstawić za pomocą wzoru iteracyjnego:

$$p_{n+1} = p_n + v_{n+1}\triangle t$$

, gdzie p_{n+1} to nowe położenie dla aktualnego kroku czasowego, p_n to położenie z poprzedniego kroku czasowgo, v_{n+1} to prędkość cząsteczki dla aktualnego kroku czasowego, Δt to wartość kroku czasowgo.

W celu obliczenia prędkości cząsteczki dla aktualnego kroku czasowego musimy znać prędkość z poprzedniego kroku i dodać do niej zmianę prędkości, która dokonała się w czasie jaki upłynął od momentu wyznaczania prędkości dla poprzedniego kroku czasowego. Wzór na wyznaczenie wartości aktualnej prędkości to:

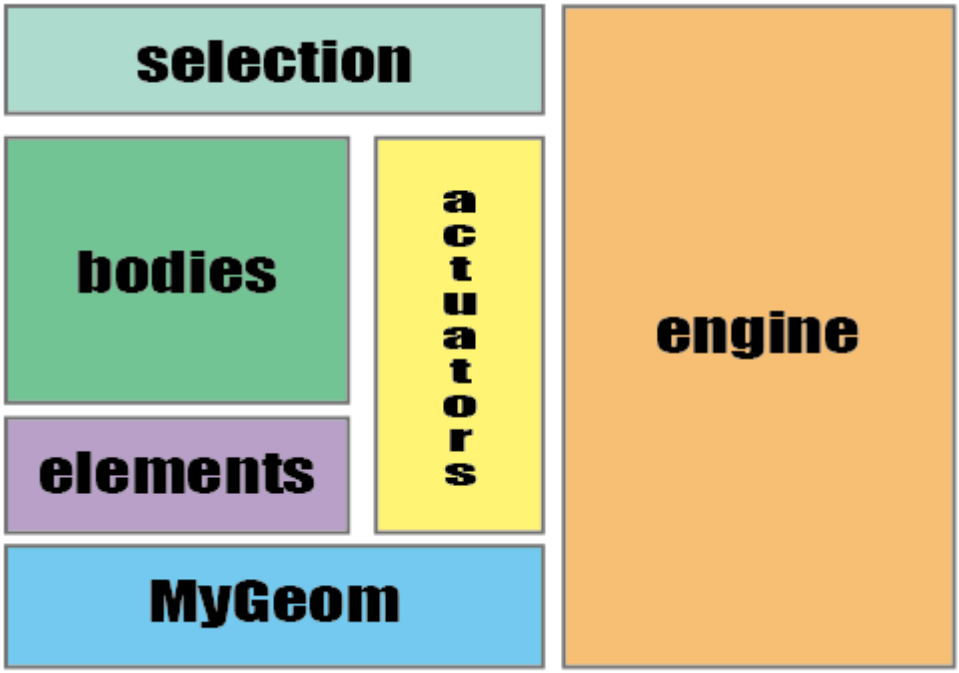
$$v_{n+1} = v_n + \frac{f}{m}\triangle t$$

, gdzie f oznacza siłę wypadkową działającą na cząsteczkę dla aktualnego kroku czasowego, a v_n prędkość z poprzedniego kroku czasowego.

Powyższe wzory pozwalają na obliczenie kolejnych pozycji cząsteczki. Istotnym zagadnieniem w symulowaniu ruchu za ich pomocą jest dobranie odpowiedniej wartości kroku czasowego Δt. Jeżeli wartość ta będzie zbyt duża symulacja stanie się niestablina numerycznie i nie będzie prawidłowo modelować rzeczywistości. Jeżeli wybierzemy tę wartość zbyt małą to symulacja będzie przebiegać bardzo powoli. Dobranie odpowiedniej wartości najlepiej wychodzi przy użyciu metody prób i błędów, przy czym należy pamiętać że im mniejsza wartość kroku czasowego tym dokładniejsza będzie symulacja. Zagadnienie doboru odpowiedniej wartości kroku czasowego w przypadku metody Eulera oraz porównanie jej z innymui metodami całkowania równań ruchu dokładnie opisał Maciej Matyka w pozycji [31].

2.4.3. Implementacja

Biblioteka silnika fizycznego implementuje mechanizmy symulacji obiektów 2D. Składa się z 5 modułów: actuators, bodies, elements, engine oraz selection. Poniższy diagram przedstawia ogólny zarys pakietów składających się na implementację silnika:



Rysunek 2.4.3.1.: Graficzna prezentacja modułów silnika fizycznego. Obraz należy czytać od dołu do góry. Widać wówczas, że podstawą implementacji silnika jest biblioteka geometrycznej MyGeom. Źródło: własne.

Pierwszy z modułów (actuators) definiuje obiekty, które wywierają wpływ na elementy symulowane na scenie. Zawiera on implementację pola siłowego, siły oraz momentu obrotowego.

Pakiet bodies zawiera definicję abstrakcyjnej klasy Body - bazę dla ciał fizycznych typu bryła miękka i bryła sztywna. Ich implementacja zawarta jest odpowiednio w klasach SpringBody oraz RigidBody.

W pakiecie elements znajdziemy podstawowe elementy budujące obiekty symulacji. Znajdują się w nim implementacje cząsteczki (masy punktowej), sprężyny, sprężyny obrotowej, brzegu ciała oraz krawędzi brzegu.

Pakiet engine zawiera implementacje algorytmów potrzebnych do uruchomienia i obliczenia kolejnych klatek symulacji. Jest to rzeczywisty silnik, który wprowadza całą machinę fizyczną w ruch. Zawiera definicję klasy kontekstu symulacji JETContext, definicję algorytmów obliczających kolejne kroki symulacji JETEngine, oraz klasy odpowiedzialne za obliczenia związane z wykrywaniem i rozwiązywaniem kolizji

między obiektami sceny: SBCollider oraz RBCollider. Bardzo ważna jest także klasa przechowująca informacje na temat kolizji dwóch ciał CollisionInfo, która zawiera wszystkie dane szczegółowo opisujące zdeżenie.

W dalszej części pracy szczegółowo znajduje się szczegółowy opis każdego z elementów wraz z opisem implementacji w języku Java.

Opis ten rozpoczyna przedstawienie elementów podstawowych, następnie brył złożonych - miękkiej oraz sztywnej, a kończy go szczegółowe omówienie algorytmów uruchamiających całą machinę fizyczną, czyli tego co daje wrażenie prawdziwego ruchu ciał symulacji.

2.4.3.1. Cząsteczka - klasa Mass

Klasa implementująca cząsteczkę definiuje trzy główne pola wektorowe opisujące położenie, prędkość oraz wypadkową siłę działającą na cząsteczkę. Dodatkowo definiuje cztery wektory pomocnicze wykorzystywane w trakcie obliczania nowego położenia cząsteczki. Wcześniejsze zdefiniowanie tych wektorów pozwala zaoszczędzić czas potrzebny na tworzenie ich instancji, który musiałby upłynąć za każdym razem, gdy przesuwalibyśmy jakąś cząsteczkę. Ponadto w klasie Mass znajdziemy definicję pola mass typu double przechowującego masę cząsteczki oraz pola moveable typu boolean decydującego o jej przesuwalności.

Sercem klasy jest metoda public void simulate(double dt), która implementuje metodę Eulera obliczania położenia cząsteczki w kolejnym kroku czasowym.
Jest ona bardzo ważna nie tylko dla klasy Mass, ale także z punktu widzenia całego silnika fizycznego, ponieważ to ona wprawia całą scenę symulacji w ruch.
Dlatego na następnej stronie na listingu 2.4.3.1.1. przedstawiono fragment jej kodu implementujący metodę Eulera.

```
	public void simulate(double dt)
	{
		// 1 - obliczamy przyspieszenie na poczatku tego kroku
		//     na podstawie aktualnie dzialajacej sily i masy
		a.become( force );
		a.scale( 1./mass );
		// 2 - obliczamy zmiane predkosci w tym kroku czasowym
		dv.become( a );
		dv.scale( dt );
		// 3 - uaktualniamy predkosc masy
		velocity.add( dv );
		// 4 - obliczamy zmiane polozenia masy punktowej
		dr.become( velocity );
		dr.scale( dt );
		// 5 - przesuwamy mase o aktualna zmiane polozenia...
		lastPosition.become( position );
		position.add( dr );
	}
}
```

Listing 2.4.3.1.1.: Implementacja metody Eulera całkowania równań ruchu. W metodzie Mass.simulate() następuje przesunięcie cząsteczki w nowe położenie, aktualne dla następnego kroku czasowego.

2.4.3.2. Sprężyna - klasa Spring

Klasa Spring implementuje sprężynę, jej zadaniem jest utrzymanie dwóch ciał w ustalonej odległości względem siebie.

Implementacja klasy w pierwszej kolejności definiuje dwa pola typu Mass wskazujące na cząsteczki połączone sprężyną. Na podstawie położenia tych cząsteczek obliczana jest aktualna długość sprężyny, która w połączeniu z długością początkową oraz współczynnikami sprężystości i tłumienia pozwala obliczyć siłę z jaką oddziałuje ona na cząsteczki.
Dokładny model fizyczny sprężyny, o stałym współczynniku sprężystości oraz z tłumieniem drgań można znaleźć w pozycjach [29] [30] [31].

Wzór opisujący wartość siły jaką należy przyłożyć do cząsteczek opisany jest równaniem:

$$F = -ks(d - d_0) \circ r_{12} + k_d[(\vec{v_1} - \vec{v_2}) \circ r_{12}] \circ r_{12}$$

, gdzie k_s - współczynnik sprężystości, k_d - współczynnik tłumienia, d - aktualna długość sprężyny, d_0 - długość sprężyny w stanie spoczynku, r_{12} - wektor od cząsteczki pierwszej do drugiej, v_1,v_2 - prędkości cząsteczek.

Działanie sprężyny polega na obliczeniu siły wg. powyższego wzoru i dodanie jej do wypadkowej siły każdej z cząsteczek, z tym że pierwsza cząsteczka dostaje siłę bezpośrednio wyliczoną jak wyżej, natomiast do drugiej, dodajemy siłę o tej samej wartości ale przeciwnym zwrocie.

Opisywana tutaj implementacja sprężyny została rozszerzona o możliwość rozerwania sprężyny. Mechanizm rozerwania polega na zaprzestaniu aplikacji siły generowanej przez sprężynę jej cząsteczkom w momencie, gdy długość sprężyny przekroczy wartość krytyczną. Wartość ta nazwana została stałą rozdarcia i oznaczona przez k_t , indeks t pochodzi od słówka drzeć (ang. tear). Opisuje ona procentowo wartość względem początkowej długości sprężyny, po przekroczeniu której następuje zaprzestanie symulacji danej sprężyny.

Najważniejsze elementy implementacji klasy Spring przedstawiona została na poniższym listingu:

```
public final class Spring  {

    (...)

    public void doYourJob() {
    if (alive) {
          Vec2d r12, m12, Fw;
          double fs, fd;
          r12 = new Vec2d( m1.getPosition() );
          r12.sub( m2.getPosition() );

          // obliczamy aktualna dlugosc sprezyny
          double r12Magnitude = r12.getMagnitude();

          // jezeli sprezyna moze sie przerwac to sprawdzamy czy
          // powinna wg. aktualnego rozciagniecia
          if (tearable && r12Magnitude>startLength*kt) {
                alive= false;
                return;
          }

          // jezeli rozciagniecie sprezyny jest wieksze od "zera"
          // to wykonujemy obliczenia zgodnie z prawem Hook'a
          if (r12Magnitude >= 0.001) {
                r12.normalize();

                // predkosc relatywna czasteczek sprezyny
                m12= new Vec2d( m1.getVelocity() );
                m12.sub( m2.getVelocity() );

                // skladowa sprezystosci
```

```
                    fs = ks * (r12Magnitude - startLength);

                    // skladowa tlumienia
                    fd = kd * m12.dot( r12 );

                    // sila wypadkowa oddzialywania sprezyny
                    Fw = new Vec2d( r12 );
                    Fw.scale( fs+fd );

                    // aplikacja sily czasteczkom sprezyny
                    m1.getForce().sub( Fw );
                    m2.getForce().add( Fw );
                }
            }
}
(...)
}
```

Listing 2.4.3.2.1.: Implementacja metody Spring.doYourJob() obliczającej i aplikującej wpływ sprężyny liniowej na cząsteczki, do których została przytwierdzona.

2.4.3.3. Sprężyna obrotowa - klasa RSpring

Klasa RSpring implementuje sprężynę obrotową, której zadaniem jest zachowanie kąta o ustalonej wartości między dwoma odcinkami złączonymi w jednym punkcie. Odcinki te definiują trzy punkty zaczepienia sprężyny obrotowej. Każdy z nich może być związany z innym ciałem - w zależności od tego z czym są powiązane tworzą różne układy fizyczne. Implementacja sprężyny obrotowej w pierwszej kolejności definiuje trzy cząsteczki, na których sprężyna jest rozpięta. Są to pola m1,m2,m3 typu Mass. Następnie zdefiniowane są pola określające własności fizyczne: stała sprężystości oraz tłumienia. Ostatnim znaczącym polem sprężyny obrotowej jest początkowa wartość kąta, do zachowania której sprężyna będzie dążyć wraz z postępem czasu symulacji.

Dokładny model fizyczny sprężyny obrotowej, wraz z wyprowadzeniem opisany jest w [30], jest to jednak model nie uwzględniający tłumienia.

Dla celów tej pracy model ten wzbogacono o symulację tłumienia drgań w zależności od relatywnej prędkości dwóch skrajnych cząsteczek sprężyny.

Pełny opis modelu sprężyny obrotowej, wraz z tłumieniem drgań przedstawiony jest poniżej:

$$\triangle\alpha = \alpha - \alpha_0$$

$$v_1' = \vec{v_1} \circ \vec{a_{\perp N}}$$

$$v_2' = \vec{v_2} \circ \vec{b_{\perp N}}$$

$$f_d = -k_d(v_1' - v_2')$$

$$f_1 = -k_s \frac{\triangle\alpha}{\|\vec{a}\|}$$

$$f_3 = k_s \frac{\triangle\alpha}{\|\vec{b}\|}$$

$$\vec{F_1} = \vec{a_{\perp N}}(f_1 + f_d)$$

$$\vec{F_3} = \vec{b_{\perp N}}(f_3 - f_d)$$

$$\vec{F_2} = -\vec{F_1} - \vec{F_3},$$

,gdzie Δα to odchylenie kątowe sprężyny od położenia równowagi, a i b to wektory kolejno między cząsteczkami: 2 i 1 oraz 2 i 3. v_1' i v_2' to rzuty prędkości cząsteczek na znormalizowane wektory prostopadłe do wektorów a i b. f_d , f_1 i f_3 to wartości kolejno: siły tłumienia, siły oddziałującej na cząsteczkę numer 1 oraz siły oddziałującej na cząsteczkę numer 3. F_1, F_2 i F_3 to siły przykładane do odpowiednich cząsteczek.
Suma sił F_1, F_2 i F_3 wynosi zero, co zapewnia spełnienie zasady zachowania energii.

Poniżej przedstawiono najważniejsze fragmenty kodu implementującego sprężynę obrotową:

```
public final class RSpring
{
    // metoda aplikująca wpływ sprężyny na połączone nią cząsteczki
    public void doYourJob()
    {
        if (alive) {
            double dAngle=0.;
            double angle=getAngle();

            Vec2d aPN = a.getPerp().getNormalized();
            Vec2d bPN = b.getPerp().getNormalized();

            dAngle = angle - startAngle;
            if (angle<0) dAngle += 2*Math.PI;

            double F1 =  -ks*dAngle / getA().getMagnitude();
```

```
            double F3 =  ks*dAngle / getB().getMagnitude();

            double v1 = m1.getVelocity().dot(aPN);
            double v3 = m3.getVelocity().dot(bPN);

            double FD = -kd*(v1-v3);

            Vec2d f1 = new Vec2d(aPN.getScaled(F1+FD));
            Vec2d f3 = new Vec2d(bPN.getScaled(F3-FD));

            Vec2d f2 = f1.getScaled(-1);
            f2.add(f3.getScaled(-1));

            // aplikacja obliczonych sił cząsteczkom sprężyny
            m1.getForce().add( f1 );
            m2.getForce().add( f2 );
            m3.getForce().add( f3 );
        }
    }
    (...)
}
```

Listing 2.4.3.3.1.: Implementacja metody RSpring.doYourJob() obliczającej i aplikującej wpływ sprężyny obrotowej na cząsteczki, do których została przytwierdzona.

2.4.3.4. Bryła miękka - klasa SpringBody

Ciało miękkie to zbiór cząsteczek połączonych sprężynami lub sprężynami obrotowymi. Na części cząsteczek rozpięty jest wielokąt zamknięty ograniczający wnętrze bryły miękkiej, służy on wykrywaniu kolizji z innymi elementami sceny. Bryła miękka posiada przypisany indeks wskazujący materiał na globalnej liście materiałów sceny z jakiego jest wykonana. Tablica materiałów definiuje współczynniki tarcia i sprężystości, gdy dochodzi do zderzenia dwóch ciał. Ponadto przechowuje informacje czy moduł zajmujący się obsługą zderzeń powinien rozpatrywać zderzenia ciał o danych materiałach.

Symulacja bryły miękkiej w każdym kroku symulacji sceny wygląda następująco:

1. wywołanie metody accumulateExternalForces(), która odpowiada za oddziaływania zewnętrzne takie jak grawitacja, ciśnienie otoczenia - w zależności od implementacji

2. wywołanie metody accumulateInternalForces(), wewnątrz której następuje obliczenie oraz przyłożenie sił pochodzących od sprężyn bryły miękkiej

3. wywołanie metody simulate(), która przemieszcza wszystkie cząsteczki bryły miękkiej zgodnie z wcześniej obliczonymi siłami

Najważniejsze fragmenty implementacji bryły miękkiej przedstawiono na poniższych listingach kodu:

```
public final class SpringBody extends Body
{
    static int sbCount;
    private int id;
    public String name;

    private ArrayList<Mass> massList;
    private ArrayList<Spring> springList;
    private ArrayList<RSpring> rSpringList;

    private Border border;
    private final BoundingArea boundingArea;
    private int material;
```

Listing 2.4.3.4.1.: Pola klasy SpringBody przechowujące nazwę ciała, listę cząsteczek, listę sprężyn, listę sprężyn obrotowych, granicę, prostokąt otaczający oraz indeks materiału ciała.

```
    public SpringBody() {
        massList = new ArrayList<Mass>();
        springList = new ArrayList<Spring>();
        rSpringList = new ArrayList<RSpring>();
        border = new Border();
        boundingArea = new BoundingBox();
        material = 0;
        id = sbCount++;
        name = "Spring Body "+id;
    }
```

Listing 2.4.3.4.2.: Konstruktor bryły miękkiej odpowiada za inicjalizację pamięci na elementy składowe bryły miękkiej oraz nadanie unikalnej nazwy nowej bryle.

```
    synchronized public void accumulateExternalForces(Vec2d a) {
        Vec2d gForce = new Vec2d();
        Collection<Mass> massCol = getMassListSync();
        synchronized(massCol) {
            for(Mass m : massCol)
            {
                gForce.become(a);
                gForce.scale(m.getMass());
                m.getForce().add(gForce);
            }
        }
    }
```

Listing 2.4.3.4.3.: Metoda służąca do obliczania wpływy czynników zewnętrznych na bryłę miękką. Czynniki te przekazywane są do ciała metody w postaci wypadkowego wektora siły oddziałującego na każdą z cząsteczek ciała; ten sposób przekazywania parametru ogranicza czynniki zewnętrzne aplikowane w tym miejscu jedynie do sił objętościowych.

```
    synchronized public void accumulateInternatForces() {
        Collection<Spring> springCol = getSpringListSync();
        synchronized(springCol) {
            for(Spring s : springCol)
                s.doYourJob();
        }

        Collection<RSpring> rSpringCol = getRSpringListSync();
        synchronized(rSpringCol) {
            for(RSpring rs : rSpringCol)
                rs.doYourJob();
        }
    }
```

Listing 2.4.3.4.4.: Implementacja metody obliczającej wpływ czynników wewnętrznych bryły miękkiej. To miejsce na obliczenie wpływu czynników takich jak sprężyny wewnętrzne, ciśnienie gazu zamkniętego w bryle etc.

```
    public void simulate(double dt) {
        getBorder().calculateOrientation();
        getBoundingArea().reset();

        Collection<Mass> massCol = getMassListSync();
        synchronized(massCol) {
            for(Mass mass : massCol) {
                mass.simulate(dt);
                getBoundingArea().extendToContain(mass.getPosition());
            }
        }
    }
}
```

Listing 2.4.3.4.5.: Definicja metody wykonującej całkowanie równań ruchu dla każdej cząsteczki bryły sztywnej. W tym miejscu następuje prawdziwe przesunięcie bryły miękkiej w nowo obliczone położenie.

2.4.3.5. Bryła sztywna - klasa RigidBody

Bryła sztywna to zbiór cząsteczek, których wzajemne położenie nie zmienia się w trakcie ruchu. Na części cząsteczek rozpięty jest wielokąt wyznaczający granicę bryły, który wykorzystywany jest przy wykrywaniu kolizji bryły z innymi elementami sceny. Implementacja bryły sztywnej przechowuje dwie listy cząsteczek: bazową i dynamiczną. Bazowa definiuje geometrię bryły sztywnej i nie zmienia się w czasie działania symulacji. Rozłożenie cząsteczek bazowych jest takie, że środek ciężkości bryły znajduje się dokładnie w początku układu współrzędnych sceny. Dzięki temu można je szybko przekształcić, tak aby odpowiadały aktualnemu położeniu bryły na scenie symulacji. Ten aktualny stan przechowywany jest w liście dynamicznej, która odświeżana jest w każdej klatce animacji na podstawie położenia i kąta obrotu bryły sztywnej względem układu sceny.

Przez swoje właściwości bryła sztywna bardzo podobna jest do zwykłej cząsteczki. Różnicą jest to, że oprócz ruchu prostoliniowego może się także obracać. Dlatego, aby opisać jednoznacznie właściwości bryły sztywnej potrzebujemy tych samych zmiennych co do opisu cząsteczki, oraz kąta o jaki bryła sztywna jest obrócona.

Symulacja ruchu bryły sztywnej sprowadza się do symulacji ruchu prostoliniowego jej środka ciężkości oraz do symulacji ruchu obrotowego wszystkich cząsteczek bryły sztywnej wokoło jej środka ciężkości.

Równania symulujące ruch prostoliniowy są takie same jak dla cząsteczki dlatego nie będą tutaj umieszczone po raz drugi.

Efekt rotacji bryły sztywnej fizycznie opisany jest analogicznie do ruchu prostoliniowego, czyli przy pomocy aktualnego kąta obrotu, prędkości obrotowej oraz przyspieszenia obrotowego. Wzór na zmianę kąta obrotu w danej klatce animacji przedstawiony jest poniżej:

$$\alpha = \alpha_0 + (\omega + e \cdot \Delta t)\,\Delta t$$

,gdzie α to aktualny kąt obrotu bryły, α_0 to kąt obrotu bryły z poprzedniej klatki animacji, ω to aktualna prędkość obrotowa, e to przyspieszenie obrotowe, oraz Δt to wartość kroku czasowego symulacji.

Przyspieszenie obrotowe e oblicza się jako stosunek momentu obrotowego przyłożonego do bryły oraz jej momentu bezwładności. Jest to operacja analogiczna do obliczania przyspieszenia liniowego jako stosunku siły działającej na cząsteczkę do jej masy.

Najważniejsze fragmenty implementacji bryły sztywnej przedstawione są na poniższch listingach:

```
public final class RigidBody extends Body {

    static int rbCount;
    int id;
    private String name;

    // położenia cząsteczek bazowych
    public final List<Vec2d> baseVecList = new ArrayList<Vec2d>();

    // cząsteczki dynamiczne
    public final List<Mass> dynVecList = new ArrayList<Mass>();

    public final Border border = new Border();
    public final BoundingArea boundingArea = new BoundingBox();

    // indeks materiału bryły sztywnej
    private int material;

    public final Vec2d force = new Vec2d();
    public final Vec2d velocity = new Vec2d();
    public final Vec2d position = new Vec2d();
    public double angle = 0.;
    public double torque = 0.;       // Torque
    public double omega = 0.;
    public double mass = 0.;
    public double mI = 0.;
```

Listing 2.4.3.5.1.: Pola składowe klasy RigidBiody: lista położeń bazowych cząsteczek, lista cząsteczek (dynamicznych) prezentowanych w symulacji, granica, prostokąt otaczający oraz indeks materiału bryły sztywnej.

```
    /**
     * Całkowanie równań ruchu bryły sztywnej
     * @param dt Krok czasowy symulacji.
     */
    public void simulate(double dt) {

        // Ruch postępowy
        Vec2d a = force.getScaled(1./mass);
        Vec2d dv = a.getScaled(dt);
        velocity.add(dv);
        Vec2d dr = velocity.getScaled(dt);
        position.add(dr);
        position.add(separationV);

        // Ruch obrotowy
        double e = torque/mI;
        double dw = e*dt;
        omega += dw;
        double dang = omega*dt;
        angle += dang;

        updateDynamicShape(dt);
    }
```

Listing 2.4.3.5.2.: Implementacja metody Eulera całkującej równania ruchu bryły sztywnej. Wywołanie tej metody przesuwa bryłę sztywną w położenie aktualne dla następnego kroku czasowego.

```
    /**
     * Buduje dynamiczną strukturę bryły sztywnej.
     * @param dt Krok czasowy symulacji
     */
    protected void updateDynamicShape(double dt) {
        for(int i=0; i<baseVecList.size(); i++) {
            Vec2d v = baseVecList.get(i).getRotated(angle).
                      getAdded(position);

            dynVecList.get(i).getLastPosition().become(
                              dynVecList.get(i).getPosition());

            dynVecList.get(i).getPosition().become(v);

            // Set mass velocity from last and actual mass position
            // and time step
            Vec2d derivedMassVel = new Vec2d(v);
            derivedMassVel.sub(dynVecList.get(i).getLastPosition());
            derivedMassVel.scale(1./dt);
            dynVecList.get(i).getVelocity().become(derivedMassVel);
        }
    }
```

Listing 2.4.3.5.3.: Metoda przekształcająca kształt bryły sztywnej z lokalnego układu współrzędnych do współrzędnych sceny symulacji. Następuje tu obrót kształtu bazowego dookoła jego środka ciężkości i następnie przesunięcie liniowe o wartość wektora położenia. Dodatkowo obliczana i ustawiana jest aktualna prędkość dla każdej cząsteczki bryły sztywnej, tak aby możliwe było obliczenie tłumienia w oddziaływaniach sprężystych, które zależą od prędkości cząsteczek, na których sprężyna jest rozpięta.

2.4.3.6. Bryła kompozytowa - połączenie klas SpringBody i RigidBody

Bryła kompozytowa to konstrukcja zbudowana z dwóch niezależnych brył: miękkiej i sztywnej. W tym układzie ważniejszą rolę odgrywa bryła miękka ponieważ to ona jest brana pod uwagę w momencie wykrywania kolizji z innymi elementami sceny. Bryła sztywna pełni jedynie funkcję szkieletu lub pamięci kształtu dla bryły miękkiej. Dzięki temu bryła miękka może zawierać bardzo niedużo sprężyn, dodatkowo mogą one być dowolnie miękkie, a i tak nie spowoduje to nieodwracalnego odkształcenia bryły miękkiej.

Dzięki temu można tworzyć elementy sceny, które będą bardzo plastyczne, jednak niezależnie od odkształcenia zawsze powrócą do swojego pierwotnego kształtu.

Konstrukcja bryły kompozytowej polega na wykonaniu poniższych kroków:

1. Stworzeniu dokładnej kopii bryły miękkiej w postaci bryły sztywnej. To znaczy, że bryła sztywna będzie posiadać tyle samo i tak samo rozłożonych cząsteczek, przy czym bryła sztywna nie potrzebuje granicy, ponieważ nie bierze udziału w kolizjach na scenie symulacji.

2. Połączeniu sprężynami wszystkich cząsteczek bryły miękkiej z odpowiadającymi im cząsteczkami bryły sztywnej. Od wartości współczynnika sprężystości tych sprężyn zależeć będzie jak szybko bryła miękka powracać będzie do pierwotnego kształtu.

3. Ustawieniu bryle sztywnej widzialności, tak aby nie była renderowana podczas symulacji oraz zdefiniowania takiego materiału, który wykluczy ją z jakichkolwiek kolizji na scenie.

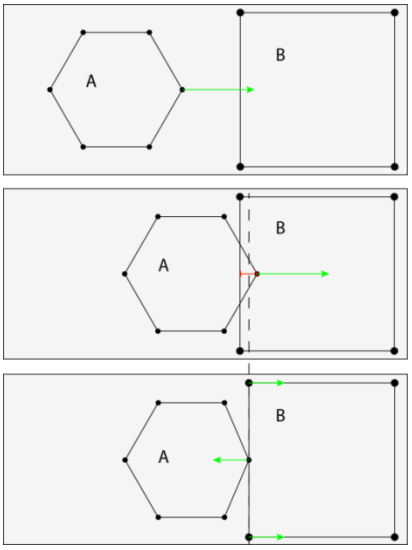
4. Tak utworzony układ może być wykorzystany do modelowania skomplikowanych elementów sceny, na przykład kół samochodowych, które mimo dużej sztywności powinny trzymać dobry kontakt z podłożem (czyli być możliwie plastyczne).

2.4.3.7. Wykrywanie kolizji - klasy SBCollider, RBCollider oraz CollisionInfo

System wykrywania kolizji podzielony jest na dwie części: SBCollider odpowiedzialny za rozwiązywanie kolizji brył miękkich oraz RBCollider odpowiedzialny za rozwiązywanie kolizji brył sztywnych.

Zasada działania obu systemów jest taka sama: po przemieszczeniu obiektów w aktualnym kroku symulacji następuje sprawdzenie czy jakieś ciała zachodzą na siebie wzajemnie. Załóżmy, że testujemy czy występuje kolizja ciała A z ciałem B, aby wykryć czy nastąpiła kolizja sprawdza się czy jakaś cząsteczka ciała A leży wewnątrz ciała B, oraz odwrotnie - czy jakaś cząsteczka ciała B znalazła się wewnątrz ciała A. W wypadku wykrycia takich cząsteczek oblicza się jak głęboko cząsteczka spenetrowała drugą bryłę oraz w jakim miejscu nastąpiło przecięcie się drogi cząsteczki z aktualnego kroku czasowego z krawędzią drugiej bryły. Na podstawie tych danych oblicza się o jaką drogę należy przesunąć cząsteczkę oraz ścianę drugiego ciała, aby w kolejnym kroku czasowym kolizja nie występowała. Ponadto należy obliczyć jak w trakcie zderzenia zmieniły się prędkości cząsteczek biorących udział w kolizji, tej penetrującej oraz tych, na których rozpięta jest ściana granicy drugiej bryły.

Poniższy rysunek prezentuje sytuację, w której dwie bryły miękkie zderzają się ze sobą. Trzy klatki pokazują sytuację w ramce animacji tuż przed zderzeniem, ramkę w której dochodzi do zderzenia, oraz tę tuż po rozwiązaniu zderzenia.



Rysunek 2.4.3.7.1.: Przykładowa kolizja dwóch brył miękkich, klatka 1: ciała przed zderzeniem, klatka2: wykrycie kolizji, klatka 3: obliczenie odpowiedzi kolizji oraz separacja geometryczna ciał. Źródło: własne.

Kod implementujący mechanizm rozwiązywania zderzeń brył miękkich prezentują listingi zamieszczone poniżej.

```
public class SBCollider {

    private static final CollisionInfo infoAway = new CollisionInfo();
    private static final CollisionInfo infoSame = new CollisionInfo();
    private static final Vec2d hitPt = new Vec2d();
    private static final Vec2d norm = new Vec2d();
    private static final Vec2d bVel = new Vec2d();
    private static final Vec2d relVel = new Vec2d();
    private static final Vec2d helpVecA = new Vec2d();
    private static final Vec2d helpVecB = new Vec2d();
    private static final Vec2d numV = new Vec2d();
    private static final Vec2d tangent = new Vec2d();
    private static final double[] edgeD = new double[1];

    public static double mPenetrationThreshold = 25;
    public static int mPenetrationCount=0;
```

Listing 2.4.3.7.1.: Pola pomocnicze klasy SBCollider przechowujące pełne informacje o aktualnie rozwiązywanej kolizji brył miękkich.

```
    public static void collisionCheck_SpringBodies(JETContext c) {
        SpringBody sb1,
                   sb2;

        for (int i=0; i<c.springBodyList.size()-1; i++)
            for (int j=i+1; j<c.springBodyList.size(); j++)
            {
                sb1 = c.springBodyList.get(i);
                sb2 = c.springBodyList.get(j);

                MaterialPair matP = c.materials[sb1.getMaterial()]
                                               [sb2.getMaterial()];
                if (matP.collide) {
                    collide(c, sb1, sb2 );
                    collide(c, sb2, sb1 );
                }
            }
    }
```

Listing 2.4.3.7.2.: Metoda sprawdzająca czy występuje kolizja między bryłami miękkimi w danym kontekście symulacji.

```
public static void collide(JETContext c, SpringBody sb1, SpringBody
sb2) {
    for(int i=0; i<sb1.getBorder().getWallsCount(); i++ ) {
        Mass mass = sb1.getBorder().getWall(i).getM1();
        Vec2d ptNorm = sb1.getBorder().getMassNormal(mass);
        // Przygotuj zmienne tymaczasowe do obsługi następnej kolizji
        boolean found = false;
        for (int j=0; j<sb2.getBorder().getWallsCount(); j++)
        {
            Wall wall = sb2.getBorder().getWall(j);
            double dist = b2.getBorder().getClosestPointOnEdgeSquared(
                                                mass.getPosition(),
                                                wall, hitPt,
                                                norm, edgeD);
            double dot = ptNorm.dot(norm);
            if (dot <= 0.0f) {
                if (dist < closestAway) {
                    closestAway = dist;
                    infoAway.bodyBwall = wall;
                    infoAway.edgeD = edgeD[0];
                    infoAway.hitPt.become(hitPt);
                    infoAway.norm.become(norm);
                    infoAway.penetration = dist;
                    found = true;
                }
            } else {
                if (dist < closestSame) {
                    closestSame = dist;
                    infoSame.bodyBwall = wall;
                    infoSame.edgeD = edgeD[0];
                    infoSame.hitPt.become(hitPt);
                    infoSame.norm.become(norm);
                    infoSame.penetration = dist;
                }
            }
        }

        // Sprawdziliśmy wszystkie krawędzie ciała bodyB
        // Czas dodać wykryte kolizjie do listy kontekstu symulacji
        if ( (found) && (closestAway > mPenetrationThreshold) &&
            (closestSame < closestAway) ) {
            infoSame.penetration = Math.sqrt(infoSame.penetration);
            c.sbColInfoList.get(c.sbColInfoCount++).set(infoSame);
        }
        else {
            infoAway.penetration = Math.sqrt(infoAway.penetration);
            c.sbColInfoList.get(c.sbColInfoCount++).set(infoAway);
        }

        // Jeżeli lista kolizji nie może pomieścić więcej elementów
        // rozszerzamy ją o kolejne 50 pustych miejsc na nowe kolizje
        if (c.sbColInfoCount>=c.sbColInfoList.size())
            for(int abcd=0; abcd<50; abcd++)
                c.sbColInfoList.add(new CollisionInfo());
    }
}
```

Listing 2.4.3.7.3.: Metoda implementująca wykrywanie kolizji pomiędzy dwoma bryłami miękkimi sb1 i sb2. Działanie polega na sprawdzeniu, czy jakaś masa ciała 1 znajduje się wewnątrz ciała 2, oraz gdy taki związek zachodzi – wygenerowania i odłożenia informacji o tym na listę kolizji kontekstu symulacji.

Rozwiązywanie zderzeń brył sztywnych działa analogicznie: znajdujemy miejsce zderzenia dwóch ciał, kierunek w którym nastąpiło zderzenie oraz relatywną prędkość obiektów biorących w nim udział. Na podstawie tych danych obliczany jest impuls, który posłuży nam do zmiany wartości prędkości obu ciał. Dokładny opis algorytmu można znaleźć w pozycjach [30] [31] [32].

Kod implementujący mechanizm rozwiązywania zderzeń brył sztywnych prezentuje poniższy listing:

```
public class RBCollider {

    private static final CollisionInfo infoAway = new CollisionInfo();
    private static final Vec2d hitPt = new Vec2d();
    private static final Vec2d norm = new Vec2d();
    private static final double[] edgeD = new double[1];
    public static double mPenetrationThreshold = 250;

    public static void collisionCheck_RigidBodies(JETContext c) {
        RigidBody   rb1, rb2;

        for (int i=0; i<c.rigidBodyList.size()-1; i++)
            for (int j=i+1; j<c.rigidBodyList.size(); j++)
            {
                rb1 = c.rigidBodyList.get(i);
                rb2 = c.rigidBodyList.get(j);
                MaterialPair matP = c.materials[rb1.getMaterial()]
                                               [rb2.getMaterial()];
                if (matP.collide) {
                    collide(c, rb1, rb2 );
                    collide(c, rb2, rb1 );
                }
            }
    }
```

Listing 2.4.3.7.4.: Pola pomocnicze klasy RBCollider przechowujące pełne informacje o aktualnie rozwiązywanej kolizji brył sztywnych.

```
public static void collide(JETContext c, RigidBody rbA, RigidBody rbB)
{
    for(int i=0; i<rbA.border.getWallsCount(); i++ ) {
        Mass mass = rbA.border.getWall(i).getM1();
        Vec2d ptNorm = rbA.border.getMassNormal(mass);
        edgeD[0] = 0;

        for (int j=0; j<rbB.border.getWallsCount(); j++) {
            Wall wall = rbB.border.getWall(j);

            // test against this edge.
            double dist = rbB.border.getClosestPointOnEdgeSquared(
                                                    mass.getPosition(),
                                                    wall, hitPt,
                                                    norm, edgeD);

            double dot = ptNorm.dot(norm);
            if (dot <= 0. && dist < closestAway) {
                closestAway = dist;
                infoAway.bodyBwall = wall;
                infoAway.edgeD = edgeD[0];
                infoAway.hitPt.become(hitPt);
                infoAway.norm.become(norm);
                infoAway.penetration = dist;
            }
            else continue;
        }

        Vec2d vA = new Vec2d(mass.getPosition());
        rbA.sceneToBase(vA);
        vA.makePerp();
        vA.scale(rbA.omega);
        vA.rotate(rbA.angle);
        vA.add(rbA.velocity);

        hitPt.become(mass.getPosition());
        Vec2d vB = new Vec2d(hitPt);
        rbB.sceneToBase(vB);
        vB.makePerp();
        vB.scale(rbB.omega);
        vB.rotate(rbB.angle);
        vB.add(rbB.velocity);

        Vec2d vAB = new Vec2d(vA.getSubbed(vB));
        if (vAB.dot(infoAway.norm) >= 0) continue;

        infoAway.vA.become(vA);
        infoAway.vB.become(vB);
        infoAway.vAB.become(vAB);
        infoAway.penetration = Math.sqrt(infoAway.penetration);
        c.rbColInfoList.get(c.rbColInfoCount++).set(infoAway);

        if (c.rbColInfoCount>=c.rbColInfoList.size())
            for(int abcd=0; abcd<50; abcd++)
                c.rbColInfoList.add(new CollisionInfo());
    }
}
```

Listing 2.4.3.7.5.: Metoda implementująca wykrywanie kolizji pomiędzy dwoma bryłami sztywnymi rbA i B. Działanie polega na sprawdzeniu, czy jakaś masa ciała A znajduje się wewnątrz ciała B, oraz gdy taki związek zachodzi – wygenerowania i odłożenia informacji o tym na listę kolizji kontekstu symulacji

```
public static void handleCollisions(JETContext c) {
    double e = 0;
    for(int i=0; i<c.rbColInfoCount; i++) {
        CollisionInfo col = c.rbColInfoList.get(i);
        RigidBody rbA = (RigidBody) col.body1;
        RigidBody rbB = (RigidBody) col.body2;

        double vABmag = col.vAB.getMagnitude();
        if (vABmag>=.01)
        {
            // Impuls liniowy
            double j = col.vAB.getScaled(-(1+e)).dot(col.norm)/
                       col.norm.dot(col.norm.getScaled(1/rbA.mass+
                       1/rbB.mass));

            rbA.velocity.add( col.norm.getScaled(j/rbA.mass));
            rbB.velocity.sub( col.norm.getScaled(j/rbB.mass));

            // Impuls obrotowy
            Vec2d rAPp = new Vec2d(col.bodyApm.getPosition());

            rAPp.sub(PhysicsUtils.getCenterOfMasses(rbA.dynVecList));
            rAPp.makePerp();
            Vec2d rBPp = new Vec2d(col.hitPt);
            rBPp.sub(PhysicsUtils.getCenterOfMasses(rbB.dynVecList));
            rBPp.makePerp();

            j = col.vAB.getScaled(-(1+e)).dot(col.norm) /
               (col.norm.dot(col.norm.getScaled(1/rbA.mass +
                                                1/rbB.mass))+
                Math.pow(rAPp.dot(col.norm), 2)/rbA.mI +
                Math.pow(rBPp.dot(col.norm), 2)/rbB.mI);

            rbA.omega += rAPp.dot( col.norm.getScaled(j) )/rbA.mI;
            rbB.omega -= rBPp.dot( col.norm.getScaled(j) )/rbB.mI;
        }

        if (col.penetration>0) {
            Vec2d move = col.norm.getScaled(col.penetration*.5);

            rbA.separationV.add(move);
            rbA.separationV.scale(.25);
            rbB.separationV.sub(move);
            rbB.separationV.scale(.25);
        }
    }
}
```

Listing 2.4.3.7.6.: Metoda implementująca rozwiązywanie kolizji między bryłami sztywnymi wykorzystujące metodę impulsową Newtona dokładnie opisaną w [32].

2.4.3.8. Symulacja sceny - klasy JETContext, JETEngine oraz Animator

Symulacja dokonywana jest na pewnym ściśle określonym zbiorze elementów, do grupowania których służy kontekst symulacji - klasa JETContext. Nie posiada ona implementacji żadnych istotnych z punktu widzenia symulacji algorytmów, służy jedynie do przechowywania i udostępniania elementów algorytmom silnika fizycznego. Odbywa się to przy użyciu generycznego kontenera dostępnego w standardowej bibliotece języka Java: ArrayList.

Klasa JETContext przemyślana jest tak, aby była przydatna zarówno w przypadku, gdy dostęp do elementów sceny musi być synchronizowany jak i wtedy gdy nie trzeba dbać o synchronizację. W tym celu wykorzystano interfejs Collection, który razem ze słowem kluczowym synchronized automatycznie synchronizuje dostęp do elementów przechowywanych w kontenerze.

Najważniejszymi elementami klasy JETContext są listy przechowujące elementy tego samego typu:

- lista brył miękkich
- lista brył sztywnych
- lista sprężyn (globalnych obiektów sceny łączących różne elementy typu springbody z rigidbody)
- lista sprężyn obrotowych (globalnych obiektów sceny)
- lista momentów obrotowych

Ponadto kontekst symulacji przechowuje informacje o zderzeniach, do których doszło w aktualnym kroku symulacji.

Informacje te podzielone są na dwie listy, dla obiektów miękkich oraz dla obiektów sztywnych.
Klasa JETEngine implementuje algorytmy służące do wykonywania obliczeń na kontekście symulacji w celu otrzymania kolejnych kroków symulacji.

W każdym kroku symulacji należy wykonać następujące czynności:

1. Przygotować mechanizmy do rozpoczęcia obliczania kolejnego kroku symulacji. Polega to na wyzerowaniu wszystkich sił wypadkowych działających na symulowane elementy. Siły te będą na nowo obliczone na podstawie energii skumulowanej wewnątrz elementów, a przechowywanej w postaci przyspieszenia liniowego i/lub kątowego oraz masy.
2. Obliczeniu wpływu aktuatorów globalnych sceny na jej elementy. Polega to na przeiterowaniu po wszystkich sprężynach, sprężynach obrotowych oraz momentach bezwładności danego kontekstu sceny i wykonaniu ich kodu symulującego.
3. Obliczenie i uwzględnienie wpływu sił zewnętrznych takich jak grawitacja.
4. Przemieszczenie elementów sceny zgodnie z nowo obliczonymi siłami jakie na nie działają w danym kroku symulacji.
5. Sprawdzeniu czy wystąpiły jakieś kolizje i zapisaniu informacji o nich wewnątrz kontekstu symulacji.
6. Rozwiązaniu kolizji z kroku nr.5

W między-czasie silnik fizyki wysyła notyfikacje do silnika skryptowego o tym co aktualnie robi. Pozwala to programiście gier rejestrować wywołania swojego kodu w określonych momentach symulacji. Lista dostępnych zdarzeń na które można zarejestrować wywołanie jakiegoś skryptu dostępna jest w pliku jetty.scripts.JetEvents.js, i przedstawia się następująco:

```
var JetSystem = {
    // Predefined string constants for all event kinds
    EVENT_TYPE_ONSTEP: "onStep",
    EVENT_TYPE_ONPREPAINT: "onPrePaint",
    EVENT_TYPE_ONPOSTPAINT: "onPostPaint",
    EVENT_TYPE_ONKEYUP: "onKeyUp",
    EVENT_TYPE_ONKEYDOWN: "onKeyDown",
    EVENT_TYPE_ONKEYTYPED: "onKeyTyped",
    EVENT_TYPE_ONBODYCOLLISION: "onBodyCollision", (...)
}
```

Listing 2.4.3.8.1.: Wykaz zdarzeń obsługiwanych z poziomu silnika skryptowego. Zdarzenia te generowane są przez kod natywny języka Java i pozwalają programiście gier reagować z poziomu skryptów na zdarzenia generowane przez silnik fizyki.

Animator to klasa środowiska JET, która bezpośrednio korzysta z klas JETContext oraz JETEngine. Rozszerza klasę wątku Thread, dzięki czemu może być wykorzystana do uruchomienia symulacji sceny poza głównym wątkiem aplikacji.

Każdy animator posiada kopię kontekstu sceny symulacji oraz wykorzystuje algorytmy klasy JETEngine.
Ponadto posiada referencję do panelu graficznego, na którym wyświetlana jest animacja przepisanej mu sceny dzięki czemu może powiadomić go w odpowiednim momencie, aby wyświetlił ostatnio obliczone położenia obiektów symulacji.

Poniżej przedstawiono najważniejsze elementy klasy Animator:

```
public final class Animator extends Thread implements Runnable
{
    public final JETContext ctx = new JETContext();
    public GraphicPanel gp;
    private final Scene scene;
    private boolean stayAlive = true;

    public Animator(String sceneName) {
        this.scene = CC.getScene(sceneName);

        // Make a copy of the whole items tree
        GroupItem topGorup = (GroupItem) scene.IC.topGroup.getCopy();

        // build physics context to simulate
        JETContext sctx = scene.SMC.buildJETContext(topGorup);
        ctx.appendJETContext(sctx);
        ctx.setValuesAndConstantsByJETContext(sctx);
        ctx.setMaterialsByJETContext(sctx);

        // set animator graphic panel
        SwingSimulationGraphicPanel swingSimGP =
        (SwingSimulationGraphicPanel) scene.GC.swingSimulationGP;

        swingSimGP.rootGroup = topGorup;
        gp = swingSimGP;

        // prepare scripts
        scene.SC.compileAll();
    }
    (...)
}
```

Listing 2.4.3.8.2.: Implementacja klasy Animator, jej konstruktora oraz główne pola składowe klasy: aktualnie animowany kontekst, referencja do panelu graficznego prezentującego przebieg symulacji, referencja do obiektu sceny symulacji, pole boolowskie decydujące o życiu obiektu animatora.

```
public void run()
{
    while( stayAlive ) {
        try
        {
            for (int i=0; i<CC.getMainLoopNoRedrawSteps(); i++)
            {
                JETEngine.prepareBeforeSimulationStep(ctx);
                JETEngine.runActuators(ctx);

                Vec2d grav = new Vec2d(0.,ctx.gravity);
                Collection<SpringBody> sbCol =
                                            ctx.getSpringBodyListSync();
                synchronized(sbCol) {
                    for(SpringBody sb : sbCol) {
                        if (!sb.fixed && ctx.isApplyGravity())
                            sb.accumulateExternalForces(grav);
                        sb.accumulateInternatForces();
                    }
                }

                Collection<RigidBody> rbCol =
                                            ctx.getRigidBodyListSync();
                synchronized(rbCol) {
                    for(RigidBody rb : rbCol) {
                        if (!rb.fixed && ctx.isApplyGravity())
                            rb.applyForce(grav, rb.position);
                        rb.accumulateInternatForces();
                    }
                }

                // Notyfikuj skrypty o kolejnym kroku animacji
                scene.SC.notifyOnStep();
                JETEngine.performSimulationStep(ctx);

                // Wykryj i rozwiąż kolizje
                if (ctx.isFindCollisions())
                    JETEngine.checkCollisions(ctx);
                if (ctx.isResolveCollisions())
                    JETEngine.handleCollisions(ctx);
                if (ctx.isApplyDamping())
                    JETEngine.applyDamping(ctx);
            }

            // Redrawing
            ((Component)gp).repaint();
            (...)
            Thread.sleep(CC.getMainLoopSleepTime());
        }
        catch (InterruptedException ex){ return; }
        catch (Exception ex){ ex.printStackTrace(); }
    }
}
```

Listing 2.4.3.8.3.: Główna metoda klasy Animator zbierająca w całość wszystkie wcześniej opisane elementy silnika fizycznego. Przeprowadza obliczenia związane z wykonaniem pojedynczego kroku symulacji, notyfikuje silnik skryptowy o wykonywanych czynnościach oraz nakazuje panelowi graficznego odrysowanie aktualnego stanu sceny symulacji

3. Projekt i implementacja środowiska

W tym rozdziale przedstawiony zostanie projekt środowiska oraz szczegółowa implementacja wybranych jego elementów. W podrodziale 3.1 omówione zostaną przypadki użycia środowiska z punktu widzenia użytkownika-programisty. Następnie w podrozdziale 3.2 omówiony będzie podział całej aplikacji na komponenty, po czym nastąpi opis własnej implementacji biblioteki do obsługi dokowalnych paneli budujących GUI środowiska – rozdział 3.3. W podrodziałach 3.4 oraz 3.5 szczegółowo omówione zostaną kluczowe struktury i mechanizmy usprawniające pracę programisty gier.

Środowisko ma upraszczać proces tworzenia gier dwuwymiarowych, których świat symulowany jest zgodnie z prawami fizyki. Projekt środowiska zakłada, że użytkownik będzie pracował nad jednym projektem gry w tym samym czasie. Każdy projekt gry składać będzie się z dowolnej liczby tzw. scen, które dokładnie opisane zostaną w dalszej części tego rozdziału.

Sceny można grupować w hierarchii przypominającej strukturę folderów na dysku komputera. Struktura ta jest drzewem składającym się z dwóch rodzajów węzłów:

- filtrów
- scen

Filtry to specjalne kontenery, które mogą przechowywać dowolną liczbę innych filtrów oraz scen. Filtrom można nadawać nazwy. Pozwalają one na utworzenie dowolnej hierarchii umożliwiającej wygodne grupowanie scen projektu.

Sceny są pojęciem bardzo abstrakcyjnym. Oznaczają zarówno plansze, po których gracz będzie poruszał się w czasie rozgrywki, jak również statyczne widoki prezentowane we wszystkich innych momentach interakcji gracza z grą, dla przykładu poniżej wymieniono kilka możliwych typów scen:

- pasywne: prezentujące logo gry
- pasywne: prezentujące autora i dodatkowe informacje o grze

- interaktywne: reklamujące inne gry danego autora
- interaktywne: ekrany menu, wyboru trybu gry, ustawień itp.
- interaktywne: plansze gry, na których toczy się właściwa rozgrywka

Sceny składają się z elementów zwanych item'ami. Każdy element, który będzie widziany przez gracza, pasywny bądź aktywny, będzie reprezentowany graficznie na scenie przez obiekt klasy Item. Również dźwięki, odgrywane w odpowiednich momentach gry, reprezentowane są przez odpowiedni typ item'u.

Item'y reprezentowane są na scenie w postaci kształtów geometrycznych.
Część z nich jest edytowalna, a część nie: zależnie od pełnionej funkcji.
Dla przykładu obiekty aktywne fizycznie są w pełni edytowalne, a elementy takie jak definicje momentów obrotowych, bądź dźwięków są edytowalne tylko w pewnym ograniczonym zakresie.

3.1. Przypadki użycia

Scenariusze użycia definiujące możliwe przypadki użycia aplikacji oraz ważniejszych części przedstawiono na poniższych diagramach use cases w notacji UML:

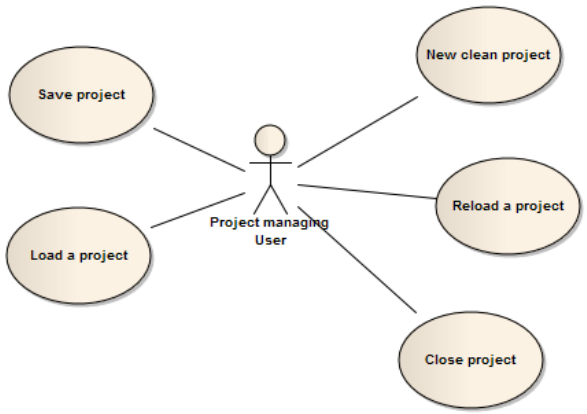


Diagram 3.1.1.: Diagram prezentujący akcje, które użytkownik będzie mógł wykonać w kontekście projektu gry.

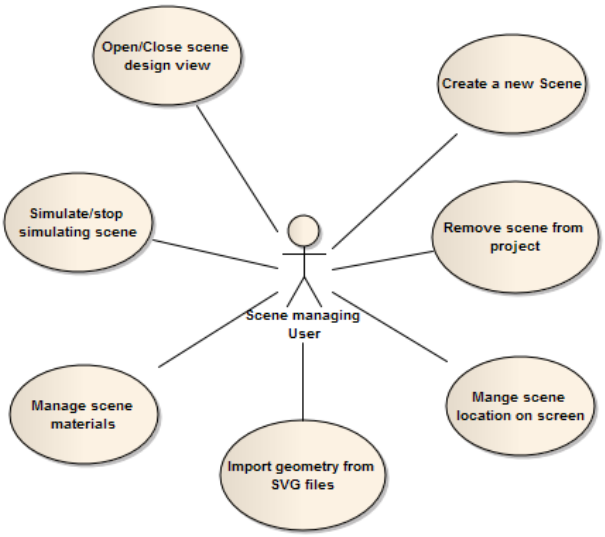


Diagram 3.1.2.: Diagram prezentuje akcje, które użytkownik może wykonać ze scenami: dodać bądź usunąć scenę, otworzyć/zamknąć tryb edycji sceny, przemieścić okienko edycji sceny w dowolne miejsce ekranu, załadować definicję geometrii z pliku SVG do sceny itp.

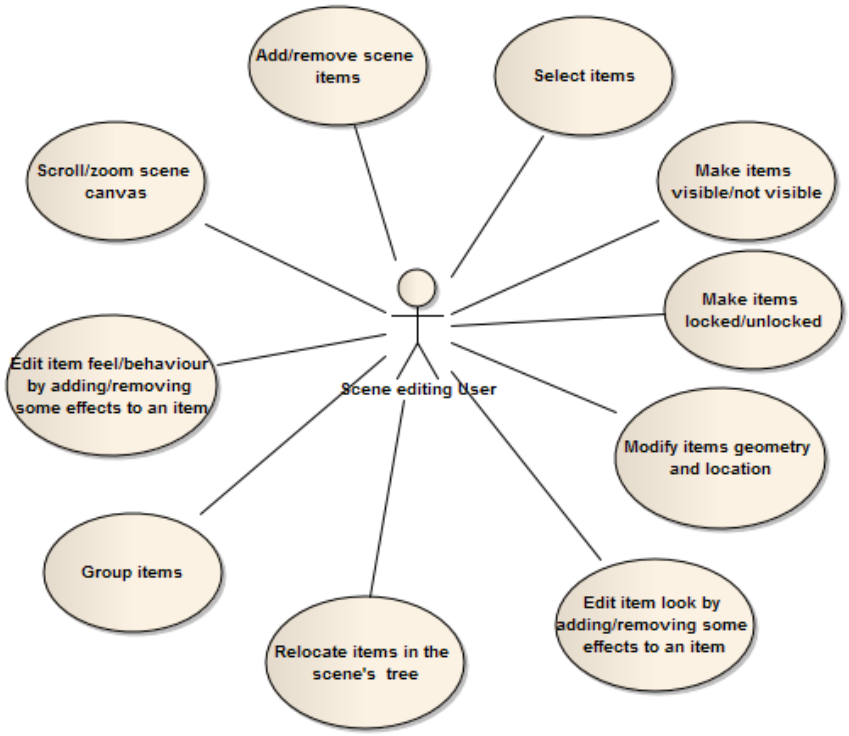


Diagram 3.1.3.: Prezentuje akcje, które użytkownik może wykonać wewnątrz sceny symulacji. Dotyczy to wszystkiego co jest związane z graficzną prezentacją sceny, jej właściwościami fizycznymi oraz elementami wewnątrz niej zdefiniowanymi.

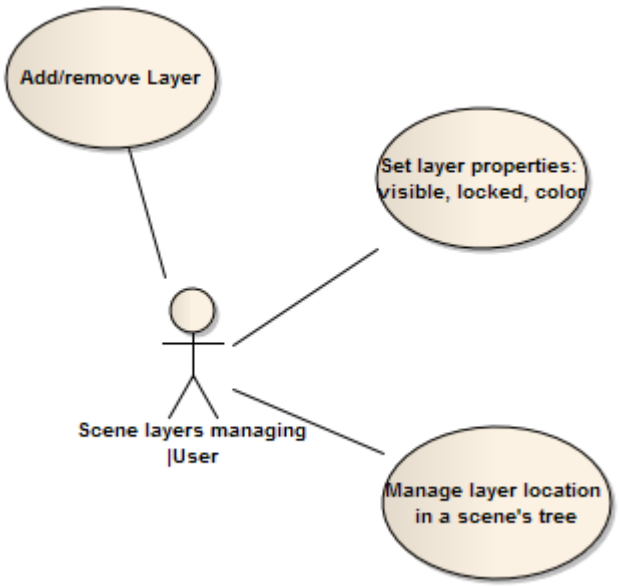


Diagram 3.1.4.: Prezentuje akcje, które można wykonać z elementami sceny reprezentującymi warstwy. Warstwy to kontenery pozwalające grupować elementy sceny w główne grupy, np. statyczne elementy tła, fizycznie aktywne podłoże, fizycznie aktywne przeszkody itp.

3.2. Komponenty

Projekt zakłada podział aplikacji na kilka głównych komponentów. Komponenty nazywane są centrami, każde z centrów odpowiada za zarządzanie innym systemem aplikacji.

Z powodu dużego skomplikowania zagadnienia jakim jest projektowane środowisko wymagane było podzielenie całej aplikacji na kilka powiązanych między sobą systemów, wśród których wyróżniono następujące główne elementy:

- System zarządzający zdarzeniami (działania użytkownika, narzędzi, efektów, kontrolek środowiska – wszystkie są obsługiwane i rozsyłane w odpowiednie miejsca środowiska przez ten system)
- System zarządzania narzędziami środowiska, głównie edytującymi geometrię i własności elementów sceny
- System zarządzający efektami nakładanymi na elementy sceny, które modyfikują ich wygląd lub własności fizyczne
- System serializująco ładujący zarządzający mechanizmami eksportowania oraz importowania elementów sceny, scen, oraz projektów do wybranych formatów plików
- System zarządzający śledzeniem zmian wewnątrz scen w trybie edycji, pozwalający cofnąć bądź powtórzyć ostatni zestaw poleceń

- Moduł pozwalający zarządzać elementami sceny z poziomu kodu źródłowego aplikacji, ważny z punktu widzenia generacji odpowiednich zdarzeń rozsyłanych wewnątrz środowiska

- System zarządzania symulacją sceny, zajmujący się konwertowaniem kontekstu edycji sceny w kontekst fizyczny, zrozumiały dla silnika fizycznego

- Moduł silnika fizycznego odpowiedzialny za reprezentację obiektów fizycznych wewnątrz środowiska oraz prowadzenie symulacji sceny

- System zarządzający graficznym interesem aplikacji.

- System zarządzający projektem oraz jego scenami

Systemy te zobrazowano na następnej stronie w postaci diagramu komponentów w notacji UML.

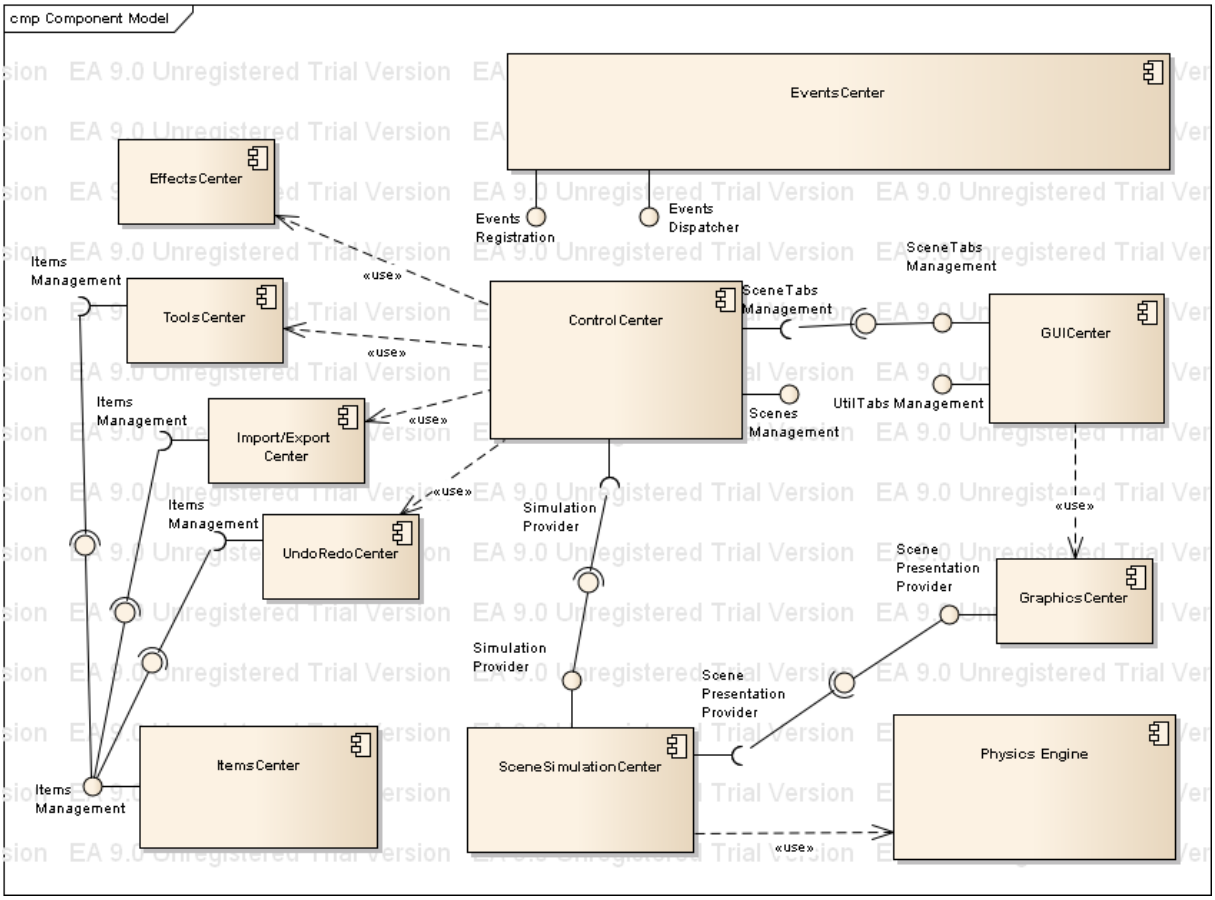


Diagram 3.2.1.: Prezentuje moduły składające się na ogólną konstrukcję środowiska.

3.2.1. ControlCenter – centrum sterowania aplikacji

Głównym zarządcą aplikacji jest komponent ControlCenter. Jako centrum sterowania projektem i jego scenami, które dostępne musi być z każdego miejsca kodu od początku życia aplikacji aż do jego końca, zaprojektowany został zgodnie z wzorcem projektowym singleton [33].

Posiada informacje o stanie aktualnego projektu oraz wszystkich aspektów z nim związanych: nazwie, lokacji na dysku komputera itp. Zarządza wysoko poziomowymi operacjami jakie użytkownik może wykonać wewnątrz środowiska względem projektu oraz jego scen, w szczególności:

- zarządzanie aktualnym projektem: import/export do pliku oraz tworzenie nowego
- zarządzanie scenami projektu
- zarządzanie symulacjami scen projektu
- zarządzanie dostępem do komponentu GUICenter, odpowiedzialnego za sterowanie zachowaniem GUI aplikacji

Na następnej stronie przedstawiono diagramy sekwencji prezentujące przykładowe sytuacje wymagające interakcji z modułem ControlCenter.

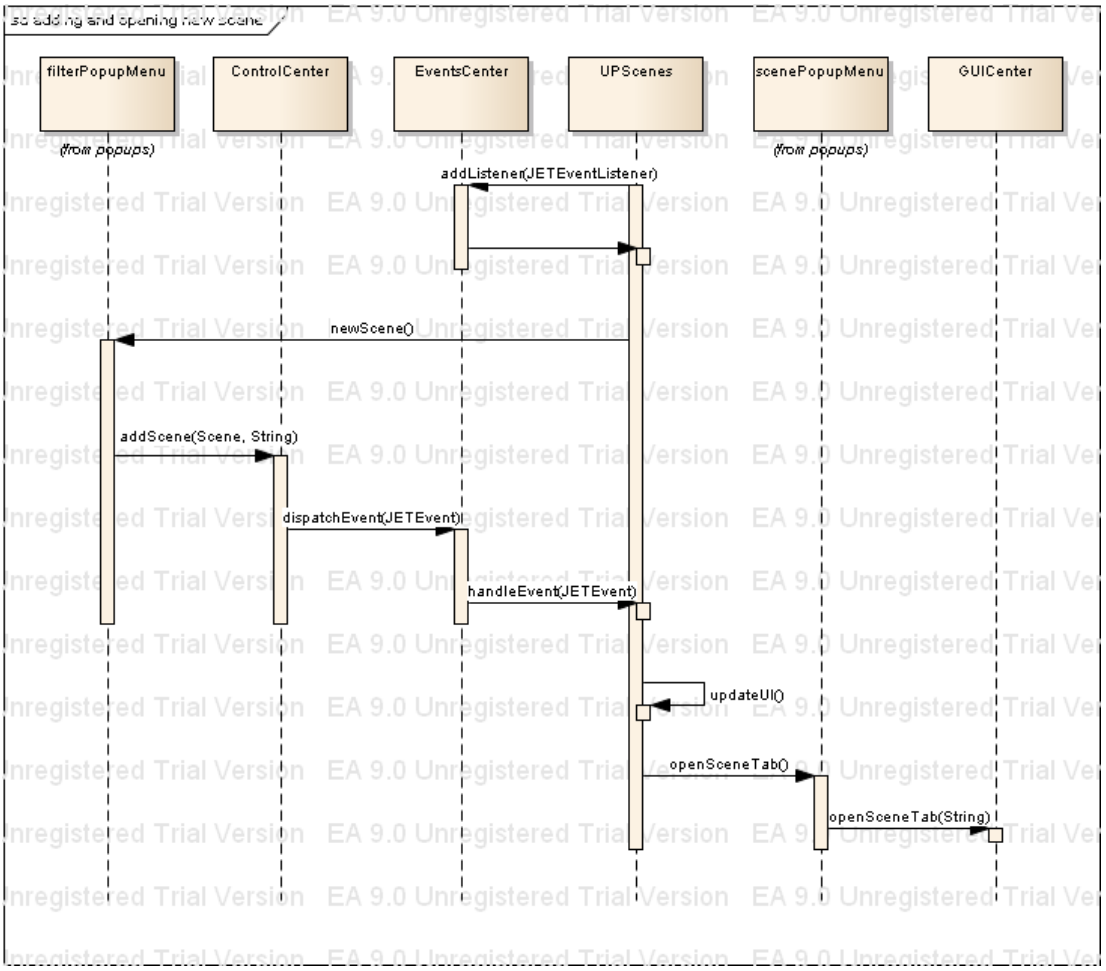


Diagram 3.2.1.1.: Przedstawia proces dodawania nowego obiektu sceny z pomocą wykorzystania GUI środowiska. Użytkownik z panelu Scenes wybiera przez menu kontekstowego filtru akcję dodania nowej sceny, a następnie przez menu kontekstowe nowo dodanej sceny wywołuje akcję otwarcia jej zakładki w jednym z dostępnym managerów zakładek typu CenterTabsPane.

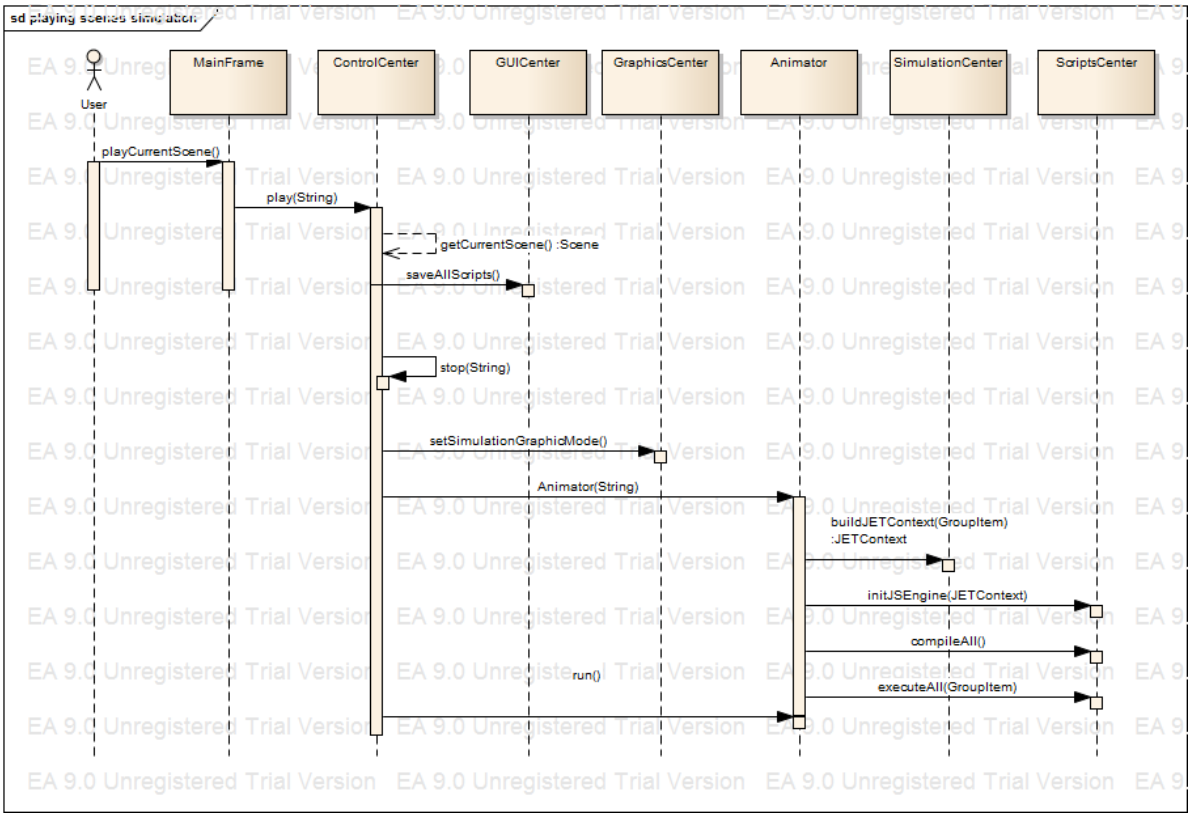


Diagram 3.2.1.2.: Przedstawia kroki wykonywane przez moduł ControlCenter w momencie uruchamiania symulacji aktualnej sceny. Użytkownik klika przycisk „play" znajdujący się w głównym oknie aplikacji, co powoduje zapisanie i skompilowanie wszystkich skryptów sceny, ustawienie trybu prezentacji symulacji wewnątrz zakładki GUI aktualnej sceny, zbudowanie fizycznego kontekstu symulacji sceny i uruchomienie wątku Animatora zarządzającego symulacją.

3.2.2. GUICenter – centrum sterowania GUI aplikacji

Drugim w kolejności, najważniejszym modułem aplikacji jest system zarządzający graficznym interfejsem aplikacji: klasa GUICenter. Jego dostępność w każdym momencie działania programu zapewnia fakt bycia publicznym, finalnym polem singletonu ControlCenter. Odpowiada za przechowywanie i zarządzanie wszystkim okienkami aplikacji: głównej formatki, dokowalnych paneli narzędzi oraz okienek dialogowych.

Kiedykolwiek trzeba zarejestrować, pokazać bądź ukryć jakiś element GUI aplikacji, należy korzystać z funkcjonalności tej klasy. Zapewnia to propagację odpowiednich zdarzeń w czasie wykonywania powyższych czynności, a to z kolei zapewnia reakcję ze strony odpowiednich dalszych modułów aplikacji.

GUICenter przechowuje listę wszystkich paneli środowiska: użytkowych, centralnych oraz dialogowych. Należą do nich panele konkretnych narzędzi, efektów oraz specjalne panele prezentujące bazowe funkcje środowiska takie jak prezentacja dostępnych narzędzi, konsola tekstowa silnika skryptowego, okienko zarządzania materiałami, wirtualną klawiaturą, ustawieniami symulacji itp.

System bazuje na bibliotece MyGUI, która zapewnia mechanizmy zarządzania okienkami dokowalnymi. Ważną jego cechą jest możliwość rejestracji dodatkowych paneli i elementów GUI. Pozwala to w dowolnym momencie działania programu wzbogacić GUI środowiska o dowolny element, co w przyszłości bardzo ułatwi tworzenie rozszerzeń do aplikacji.

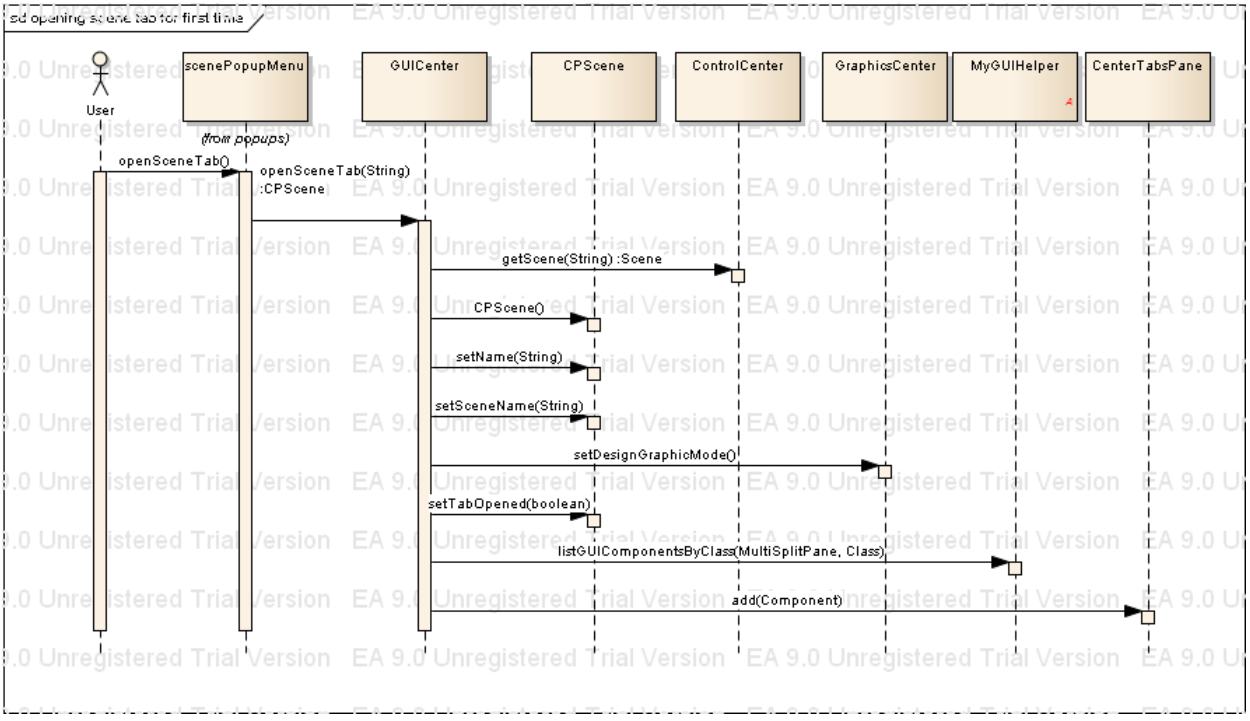


Diagram 3.2.2.1.: Przedstawia kroki wykonywane przez GUICenter w momencie otwierania panelu prezentującego scenę w trybie edycji.

3.2.3. EventsCenter – centrum zarządzania zdarzeniami

Moduł odpowiedzialny za wewnętrzną komunikację między wszystkimi komponentami środowiska zapewnia system propagacji zdarzeń EventsCenter. Jest to singleton pozwalający każdemu modułowi zarejestrować się jako część kodu

zainteresowana występowaniem konkretnych zdarzeń, generowanych przez różne części aplikacji.

Dla przykładu panel wyświetlający strukturę elementów sceny „Layers" zainteresowany jest zdarzeniami związanymi ze zmianą pozycji elementów sceny wewnątrz głównego jej drzewa elementów. W tym celu rejestruje się on w systemie EventsCenter jako słuchacz zdarzeń typu ItemEvent.kItemTreePositionChangedAction przez co za każdym razem, gdy jakiś element sceny zostanie przemieszczony wewnątrz drzewa elementów, zostanie wysłany do niego obiekt zdarzenia opisujący który element został przemieszczony. Następnie panel Layers będzie w stanie odświeżyć swój widok, aby pokazać aktualny stan drzewa elementów sceny użytkownikowi środowiska.

Wszystkie zdarzenia generowane wewnątrz środowiska posiadają informacje o nadawcy oraz o typie zdarzenia. Obiekty rejestrujące się wewnątrz systemu EventsCenter przechowywane są w strukturze zbudowanej z podwójnie zagnieżdżonej hash-mapy, mapującej nadawców oraz typy zdarzeń na obiekty słuchaczy zainteresowane konkretnymi zdarzeniami.
W pierwszym szeregu nazwy identyfikujące nadawców mapowane są na obiekty hash-map mapujące typy zdarzeń na zbiór obiektów nasłuchujących danego zdarzenia. Dzięki temu w rezultacie pojawienia się zdarzenia od konkretnego nadawcy o konkretnym typie wiadomo jaki zbiór słuchaczy powinien zostać powiadomiony o zdarzeniu.

Elementy środowiska, które chciałyby zarejestrować się jako słuchacze zdarzeń muszą implementować pewien zestaw metod zrozumiały dla systemu EventsCenter.

Interfejs JETEventListener, bo o nim tu mowa deklaruje dwie metody:

- handleEvent(JETEvent): boolean
- getRegistryPoints():Map<String, Set<String>>

Pierwsza to punkt wejściowy obiektu słuchacza, do którego przekazywane są zdarzenia. W tej metodzie jest miejsce na wykonanie wszystkich czynności jakie obiekt słuchacza powinien zrobić w reakcji na zaistniałe zdarzenie.
Dla przykładu wcześniej wspomniany panel Layers buduje w tym miejscu nowe prezentowane przez siebie drzewo elementów sceny i prosi moduł graficzny o odświeżenie swojego widoku wewnątrz GUI środowiska.

Druga metoda służy do przekazania modułowi EventsCenter informacji, o tym o jakich zdarzeniach obiekt słuchacza chce być notyfikowany.
Dla przykładu okienko Layers posiada następującą implementację tej metody :

```
@Override
public Map<String, Set<String>> getRegistryPoints() {
    Map<String,    Set<String>>    senders    =    new    HashMap<String,
Set<String>>();

    // Register for items events
    HashSet itemsActions = new HashSet<String>();
    itemsActions.add(ItemEvent.kItemAddedAction);
    itemsActions.add(ItemEvent.kItemRemovedAction);
    itemsActions.add(ItemEvent.kItemSelectionChangedAction);
    itemsActions.add(ItemEvent.kItemTransformedAction);
    itemsActions.add(ItemEvent.kItemTreePositionChangedAction);
    itemsActions.add(ItemEvent.kItemNameChangedAction);

    senders.put(JETEvent.kItemsSender, itemsActions);

    // Register for scenes events
    HashSet scenesActions = new HashSet<String>();
    scenesActions.add(SceneEvent.kSceneAddedAction);
    scenesActions.add(SceneEvent.kSceneRemovedAction);
    scenesActions.add(SceneEvent.kCurrentSceneChangedAction);

    senders.put(JETEvent.kScenesSender, scenesActions);

    (...)    // Rejestracja na inne akcje została pominięta ...

    return senders;
}
```

Listing 3.2.3.1.: Ciało metody LayersTreeModel.getRegistryPoints() deklarujące, że model drzewa panelu Layers zainteresowany jest otrzymywaniem informacji o zdarzeniach dotyczących konkretnych akcji wykonywanych na obiektach scen oraz na elementach scen.

Dzięki temu okienko Layers (a tak naprawdę model danych odpowiedzialny za zawartość prezentowanego wew. okienka Layers drzewa) będzie mogło podejmować odpowiednie działania w momencie, gdy użytkownik środowiska wykona jakąś akcję związaną ze scenami gry, bądź elementami tych scen.

Możliwość wyboru rejestracji tylko na konkretne typy zdarzeń pozwala zaoszczędzi czas wykonywany na notyfikację zainteresowanych obiektów.
Gdyby każdy obiekt rejestrował się na wszystkie możliwe zdarzenia, narzut czasowy związany z meldowaniem pojedynczego zdarzenia znacznie utrudniałby płynną pracę ze środowiskiem.

Na następnej stronie przedstawiono diagram sekwencji pokazujący kroki jakie obiekt słuchacza musi wykonać, aby zarejestrować się wewnątrz modułu EventsCenter jako słuchacz, oraz w jak wygląda droga od zaistnienia zdarzenia do momentu meldunku o nim wszystkim zainteresowanych słuchaczom.

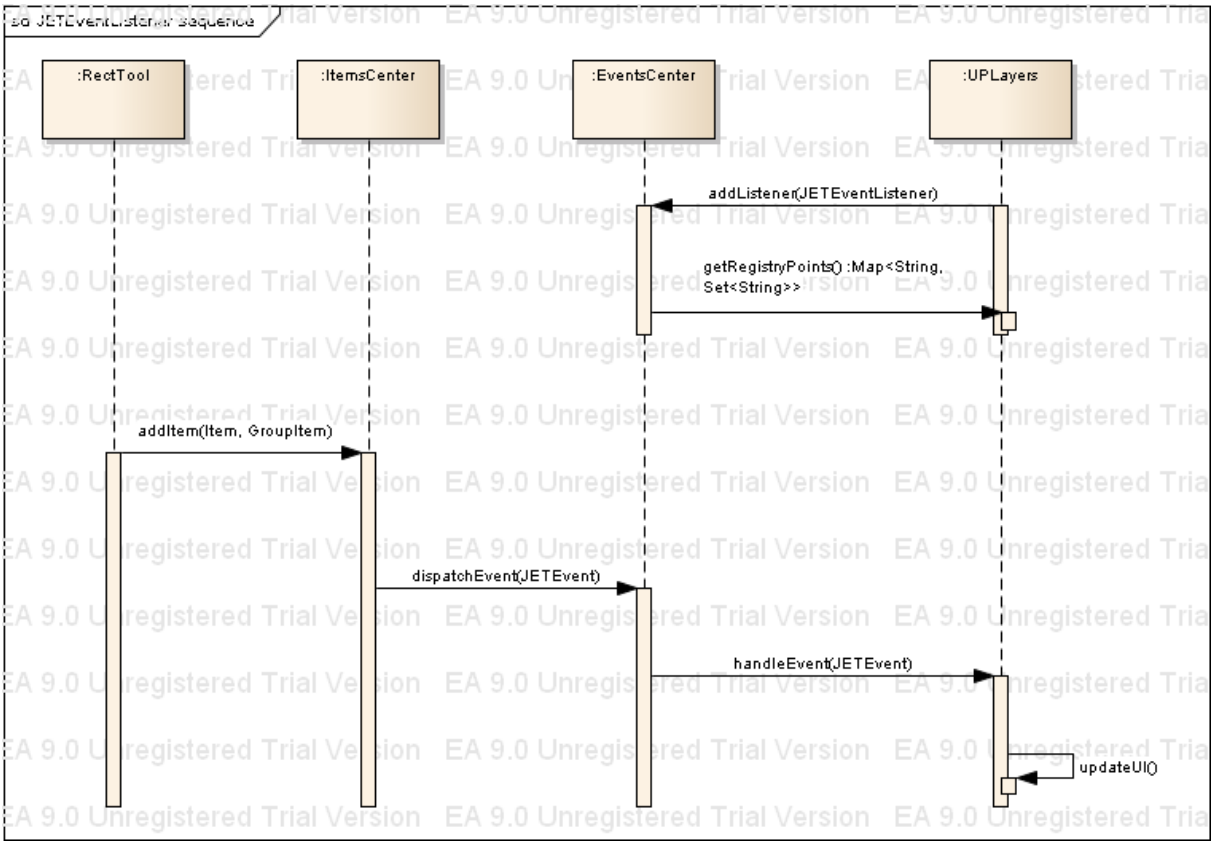


Diagram 3.2.3.1.: Diagram sekwencji prezentujący proces przekazywania zdarzenia informującego o dodaniu nowego elementu do drzewa elementów sceny.

3.2.4. ItemsCenter – zarządzanie elementami sceny z poziomu kodu źródłowego

Użytkownik może zmieniać scenę w trybie edycji. Ma do tego celu udostępnionych kilka narzędzi oraz efektów. Poprzez akcje wykonane za pomocą narzędzi, bądź w wyniku zaaplikowania konkretnego efektu zmienia się stan geometryczny, fizyczny bądź graficzny elementu sceny. Każda z takich akcji powinna generować odpowiednie zdarzenie, które dotrzeć powinno do modułów środowiska nim zainteresowanych.

Aby to umożliwić powstał interfejs/moduł służący do zarządzania zmianami elementów sceny oraz głównego drzewa elementów sceny. ItemsCenter to klasa, której wszystkie metody są statyczne. Jest to zbiór mechanizmów, dzięki którym programista może budować oraz modyfikować drzewo elementów sceny nie zastanawiając się w ogóle nad systemem zdarzeń, które są generowane za niego właśnie przez metody klasy ItemsCenter.

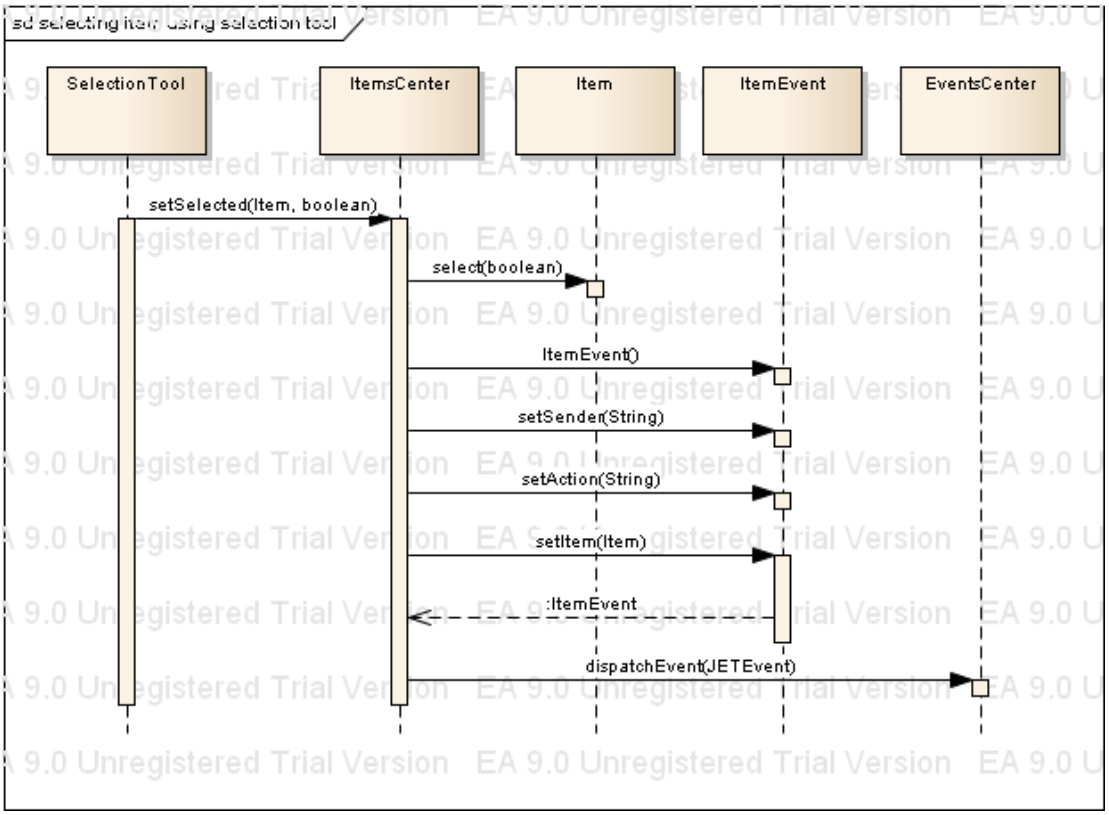


Diagram 3.2.4.1.: Prezentuje przykład wykorzystania modułu ItemsCenter do selekcji elementu sceny z użyciem narzędzia selekcji. Użytkownik wybiera element ze sceny, który ma zostać wyselekcjonowany, co powoduje wywołanie metody ItemsCenter.select(item, boolean), która ustawia flagę selected na wybranym elemencie, a następnie tworzy obiekt opisujący zdarzenie selekcji i melduje je do centrum zarządzania zdarzeniami EventsCenter.

Wyodrębnienie takiego modułu ma też duże znaczenie w przypadku pisania rozszerzeń do środowiska (tzw. pluginów). Rozszerzenia to dynamicznie ładowane moduły, które dostarcza się w postaci niezależnych plików binarnych (w przyp. Javy, pliki te zawierają tzw. byte code). Aplikacja, w tym wypadku nazywana gospodarzem (ang. Host) ładuje takie pliki w momencie kiedy pojawia się potrzeba wykorzystania ich funkcjonalności. Dobrym przykładem takiego dynamicznie ładowanego modułu może być mechanizm zapisujący lub ładujący dane z/do aplikacji z plików o danym formacie. Jeżeli obsługa danego formatu nie wchodzi w skład domyślnych formatów środowiska to wykorzystując moduły ładowane dynamicznie można bardzo niskim kosztem rozszerzyć możliwości aplikacji o obsługę danego formatu. Ten typ pluginów jest szeroko stosowany w wielu programach komercyjnych do obróbki geometrii wektorowej, na przykład w Adobe Illustrator.

Dzięki zebraniu całej funkcjonalności do zarządzania drzewem elementów sceny wewnątrz jednego modułu, przy okazji zdefiniowane zostało kompletne API (ang. Application Programming Interface), z którego mogą korzystać moduły rozszerzające możliwości środowiska. Zgromadzenie całej funkcjonalności w jednym miejscu ułatwia zarządzanie i śledzenie mechanizmów udostępnianych modułom rozszerzającym, a to ma duże znaczenie z punktu widzenia spójności interfejsu wystawianego na zewnątrz aplikacji. Poniżej przedstawiono diagram UML prezentujący możliwe do wykonania czynności przez moduł ItemsCenter.

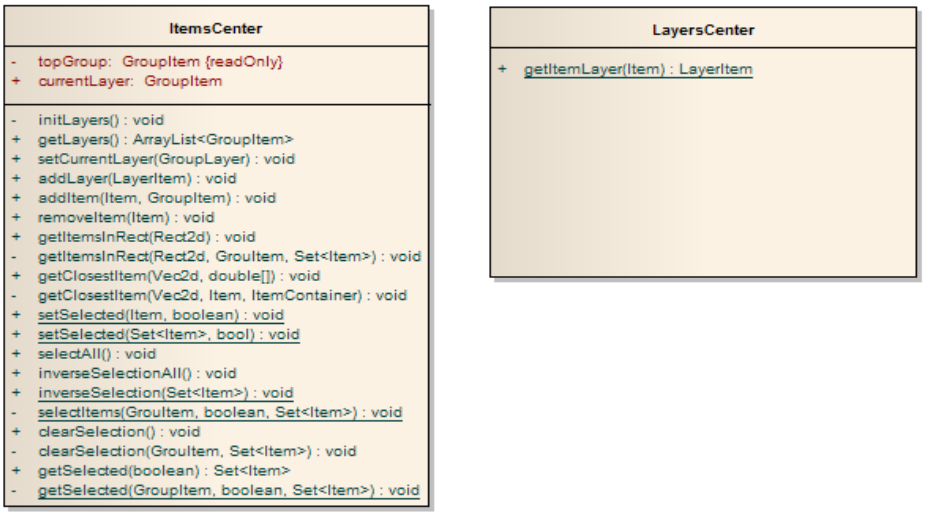


Diagram 3.2.4.2.: Diagram prezentujący klasy umożliwiające programiście zarządzanie itemami wewnątrz sceny.

3.2.5. UndoRedoCenter – system śledzący zmiany podczas edycji sceny

Moduł UndoRedoCenter zapewnia możliwość swobodnego cofania i powtarzania akcji wykonywanych podczas edycji sceny. Akcje, które mogą zostać cofnięte bądź powtórzone należą do grupy akcji zmieniających własności geometryczne, graficzne bądź fizyczne elementów sceny. Pozwala to użytkownikowi na swobodne eksperymentowanie z wyglądem elementów i ułatwia eksplorację różnych dróg do uzyskania zadowalającego efektu, bez konieczności wykonywania tej samej pracy wiele razy.

Sposób w jaki zmiany są śledzone wewnątrz implementacji modułu sprowadza się do kolejkowania akcji wykonywanych przez użytkownika. W momencie, kiedy użytkownik chce cofnąć ostatnio wykonaną czynność wciska kombinację klawiszy Ctrl+Z, a moduł automatycznie wykonuje kod, który niweluje ostatnio wprowadzone zmiany. Analogicznie, po tym jak akcja została cofnięta, użytkownik może powrócić do stanu sprzed jej cofnięcia wciskając kombinację klawiszy Ctrl+Shift+Z (skróty klawiszowe zostały zaadoptowane z programu Adobe Ilustrator).

Moduł UndoRedoCenter bazuje na założeniu, że akcje które można cofnąć i powtórzyć są atomowe względem sceny i jej elementów. Atomowość w tym znaczeniu oznacza, że akcja wywołana przez użytkownika wykona się w całości lub wcale. Dla przykładu akcja dodania nowego elementu do sceny, np. prostokąta składa się z kilku mniejszych akcji wykonywanych przez użytkownika. Aby dodać prostokąt z użyciem narzędzia do tworzenia prostokątów użytkownik oraz narzędzie do tworzenia prostokątów muszą wykonać następujące kroki ([U] oznacza akcje wykonywane przez użytkownika, [N] akcje wykonywane przez narzędzie):

1. [U] Wcisnąć lewy przycisk myszy w wybranym punkcie edytora geometrycznego sceny
2. [U] Przeciągnąć wskaźnik myszy po edytowanym obszarze sceny definiując tym samym wymiary nowego prostokąta
3. [U] Zwolnić lewy przycisk myszy
4. [N] Oblicza pozycję i wymiary prostokąta rozciągniętego przez użytkownika w punktach 1-3
5. [N] Tworzy element sceny na podstawie obliczonego w pkt.4 prostokąta
6. [N] Dodaje nowo utworzony element do drzewa elementów sceny

W każdym momencie między wypisanymi powyżej pod-akcjami może dojść do przerwania procesu tworzenia nowego prostokąta, np. przez wykrycie jakiegoś bliżej nieokreślonego wyjątku. Dlatego aby system UndoRedoCenter mógł pozwolić użytkownikowi na cofanie i powtarzanie tego typu akcji, zamykane one są w małe obiekty zapewniające im atomowość.

Pomysł polega na tym, aby każda akcja wykonywana przez użytkownika, narzędzie bądź efekt, która może być cofana i powtarzana rozszerzała klasę abstrakcyjną UndoRedo. Klasa ta posiada deklaracje dwóch metod :

- redo()
- undo()

Pierwsza służy do wywoływania atomowego kodu wykonania akcji,
druga metoda cofa zmiany wprowadzone podczas wykonania pierwszej.
Jeżeli jakiś błąd pojawi się podczas wykonywania metody redo(), to obiekt UndoRedo nie zostanie odłożony do kolejki zapamiętanych akcji modułu UndoRedoCenter a środowisko będzie znajdowało się w stanie tuż sprzed wywołania metody redo().

Na następnych dwóch stronach zobrazowano (w postaci listingu 3.2.5.1. oraz diagramu 3.2.5.2) wykorzystanie obiektów typu UndoRedo oraz modułu UndoRedoCenter na przykładzie dodawania nowego elementu sceny będącego prostokątem do drzewa elementów sceny.

```
class URCreateRect extends UndoRedo {

    public final GraphicPanel gp;
    public final ShapeItem item;
    public final int[] treeLocation;

    public URCreateRect(GraphicPanel gp, ShapeItem item, int[]
                        treeLocation) {
        this.gp = gp;
        this.item = item;
        this.treeLocation = new int[treeLocation.length];
        System.arraycopy(treeLocation, 0, this.treeLocation, 0,
                         treeLocation.length);
    }

    @Override
    public void undo() {
        Scene scene = ControlCenter.getInstance().
                      getScene(gp.getSceneName());
        scene.IC.removeItem(item);
    }



    @Override
    public void redo() {
        Scene scene = ControlCenter.getInstance().
                      getScene(gp.getSceneName());
        scene.IC.addItemAtLoc(item, treeLocation);
    }
}
```

Listing 3.2.5.1.: Kod implementujący obiekt URCreateRect tworzący prostokąt wewnątrz narzędzia RectTool. Metody undo() i redo() stanowią atomowe operacje wykonywane na rzecz drzewa elementów sceny, dodające i usuwające konkretny egzemplarz elementu będącego prostokątem.

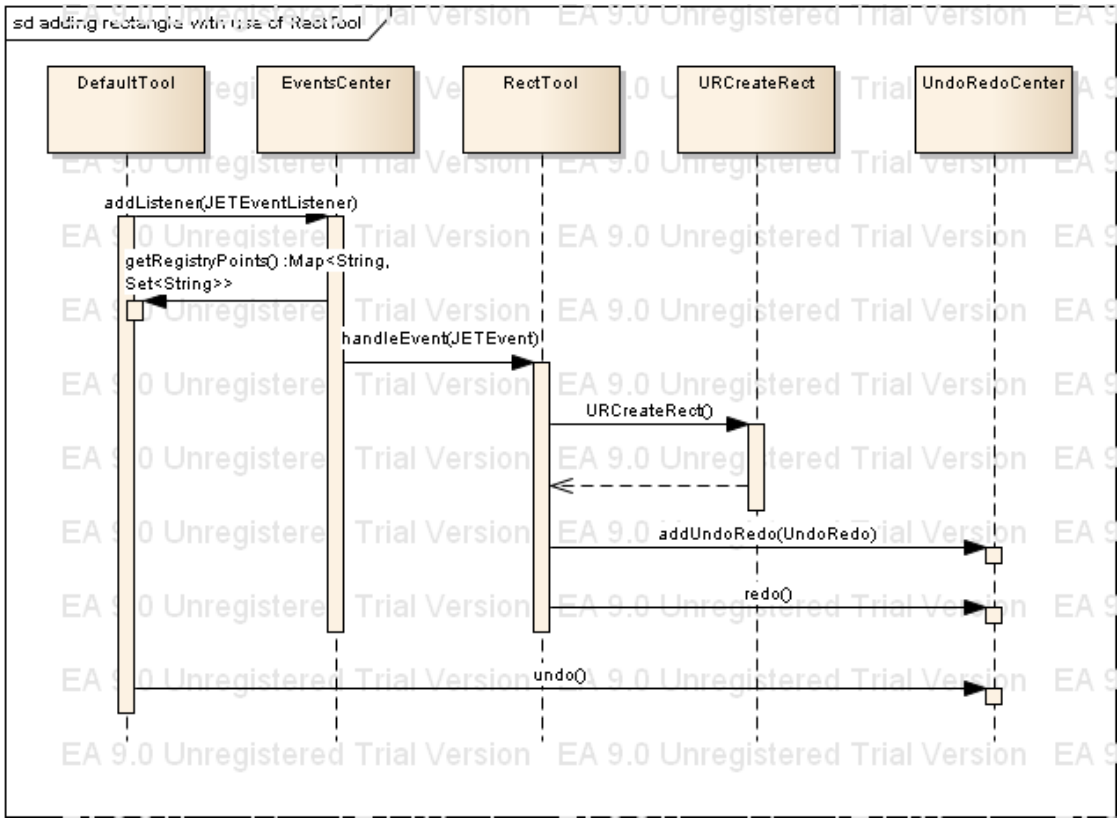


Diagram 3.2.5.2.: Prezentuje kroki wykonywane przez narzędzie do tworzenia prostokątów RectTool wymagane w celu utworzenia i zarejestrowania (wewnątrz modułu UndoRedoCenter) obiektu URCreateRect, który wykonuje kod dodania bądź usunięcia konkretnego prostokąta, zapewniając tym samym możliwość cofnięcie bądź ponownego wykonania instrukcji stworzenia prostokąta.

3.3. Dokowalne GUI

Środowisko IDE powinno dawać użytkownikowi możliwość dowolnego rozmieszczenia narzędzi na obszarze roboczym. Domyślne możliwości biblioteki Swing ograniczają się do wyświetlania okien zawierających statycznie rozmieszczone komponenty graficznego interfejsu użytkownika.

3.3.1. Panel wielodzielony - klasa MultiSplitPane

W celu umożliwienia użytkownikowi dowolnego rozmieszczania okien i narzędzi na dostępnym obszarze monitora, na bazie biblioteki Swing stworzona zastała biblioteka MyGUI. Rozszerza ona funkcjonalność Swing'a o możliwość dokowania typowych paneli JPanel wewnątrz specjalnie przygotowanych wielodzielonych paneli klasy MultiSplitPane (MSP). Poniżej przedstawiono diagram UML prezentujący hierarchię dziedziczenia klasy MultiSplitPane oraz klasy pomocniczej MultiSplitPaneCell.

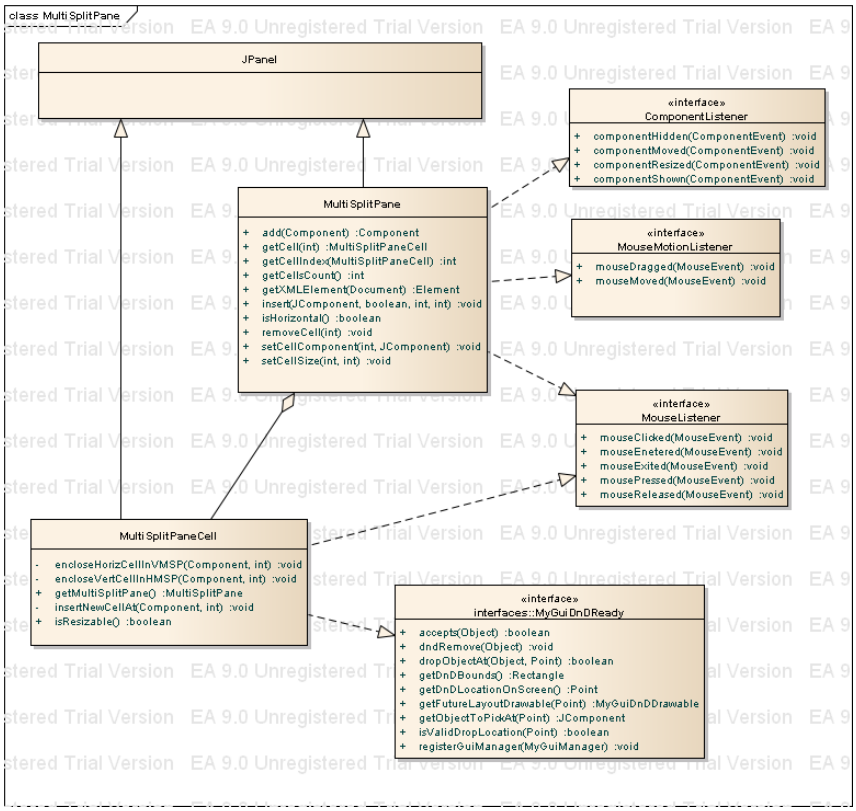
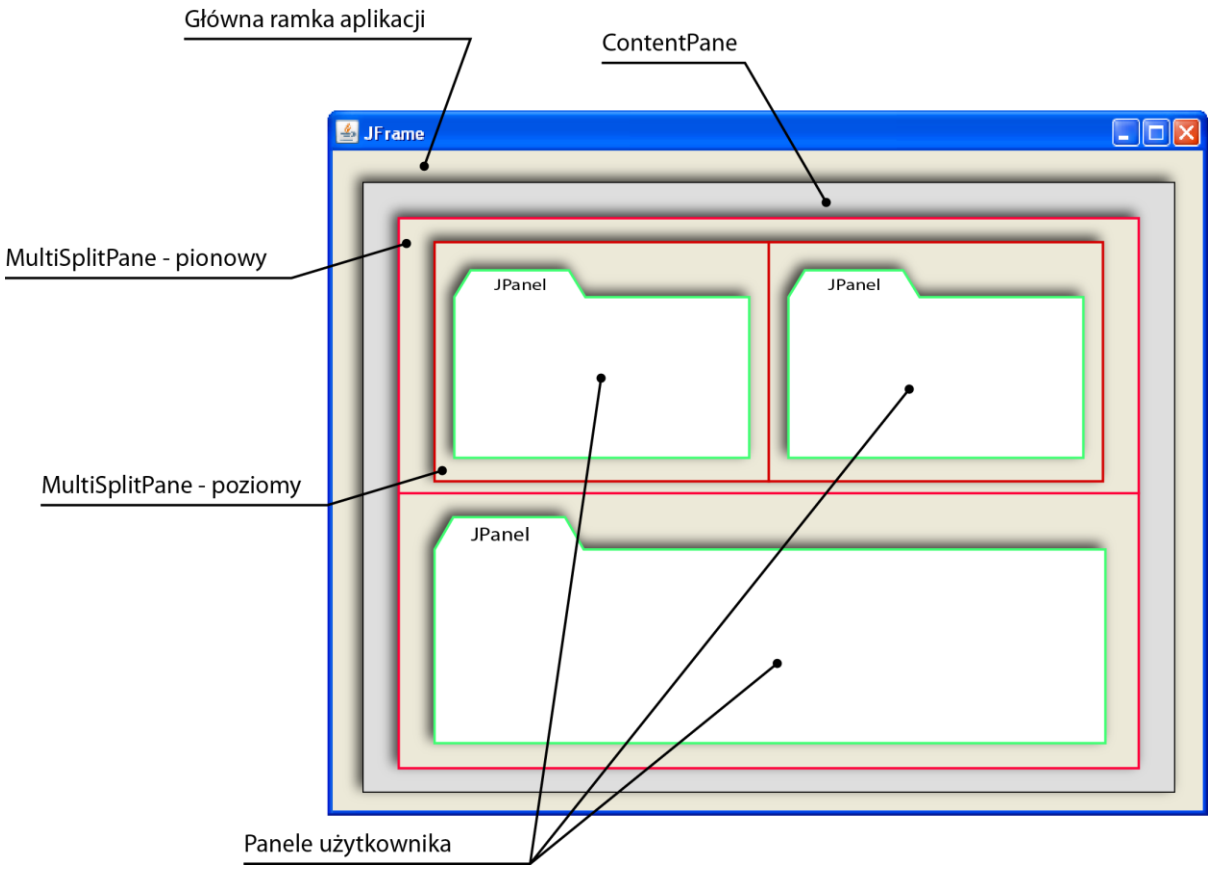


Diagram 3.3.1.1.: Prezentuje hierarchię dziedziczenia klasy MultiSplitPane implementującej funkcjonalność panelu wielodzielnego.

Każdy panel MultiSplitPane może zostać podzielony na dowolną liczbę komórek, z których każda może przechowywać kolejne panele MultiSplitPane lub inne komponenty.

Poniższy rysunek prezentuje przykładowy układ kilku paneli zagnieżdżonych w prostej strukturze komponentów:



Rysunek. 3.3.1.1.: Uproszczona struktura komponentów graficznych typu MultiSplitPane z uwzględnieniem głównych kontenerów graficznych wystawianych przez bibliotekę SWING: JFrame oraz ContentPane. Przedstawiony przypadek prezentuje dwa obiekty typu MultiSplitPane: pierwszy dwu komórkowy pionowy, drugi dwu komórkowy poziomy, znajdujący się w pierwszej komórce pierwszego panelu wielodzielnego. Do pozostałych komórek włożono przykładowe obiekty klasy JPanel w celu prezentacji ich ewentualnego przeznaczenia. Źródło: własne.

Przez połączenie pionowych i poziomych paneli MultiSplitPane można uzyskać dowolnie skomplikowaną hierarchię elementów GUI. Każdą komórkę panelu MultiSplitPane można traktować jako kolejny wielodzielony panel lub kontener paneli-zakładek.

Komórka panelu wielodzielonego to obiekt klasy MultiSplitPaneCell. Implementuje ona funkcjonalność odpowiedzialną za przechowywanie managerów zakładek aplikacji, oraz odpowiada za implementację części mechanizmu DragAndDrop (DnD) pozwalającego dynamicznie zmieniać ich położenie na obszarze roboczym aplikacji. Decyduje o obszarach aktywnych DnD, z których użytkownik jest w stanie „wyjąć" obiekt będący potomkiem komórki panelu wielodzielonego i opuścić w nowe miejsce dokowania. Ponadto implementuje funkcjonalność zamiany pojedyńczej komórki MSP w pionowy bądź poziomy panel MSP, dzięki czemu możliwe jest dynamiczne zmienianie hierarchii paneli MSP aplikacj.

3.3.2. Grupowanie zakładek - klasa TabsPane

Klasa TabsPane jest przykładem panelu-managera pozwalającego grupować panele użytkownika w zakładki. Poniżej przedstawiono diagram UML prezentujący hierarchię dziedziczenia oraz wewnętrzną strukturę klasy TabsPane.

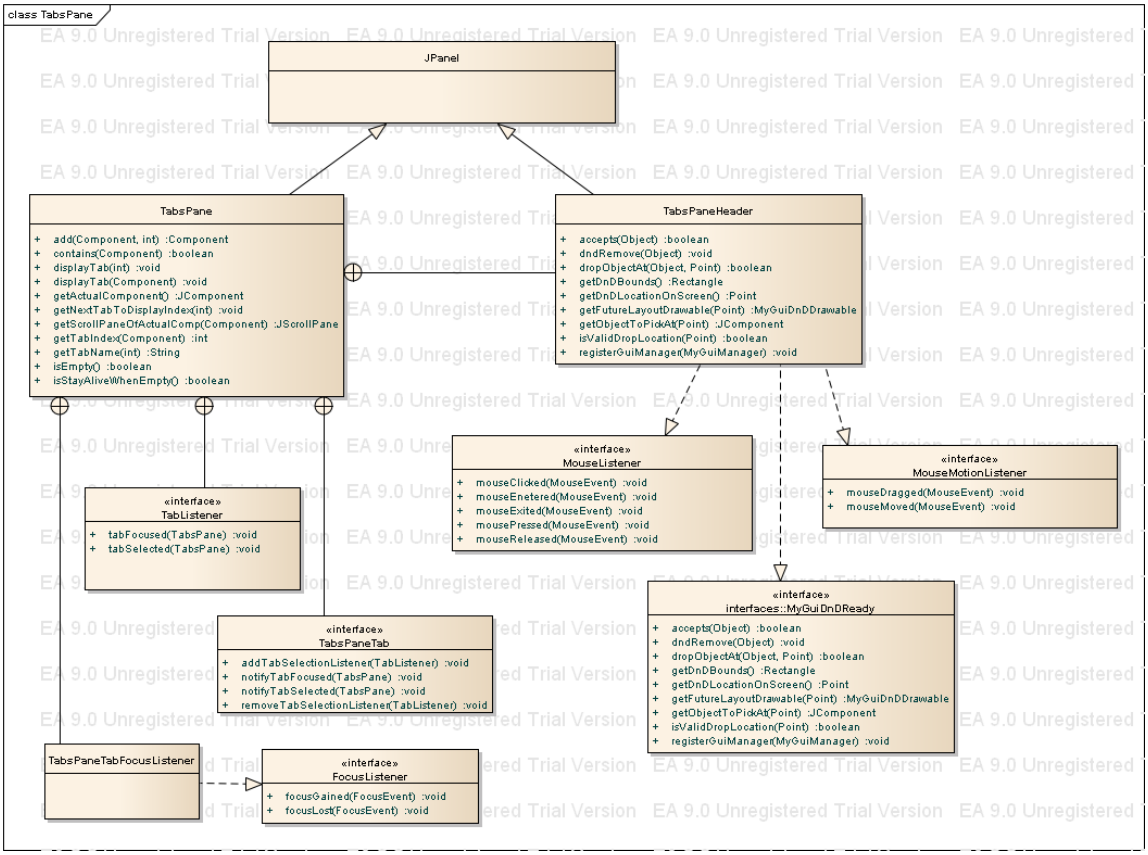


Diagram 3.3.2.1.: Prezentuje hierarchię dziedziczenia klasy TabsPane oraz zależności między nią a klasą odpowiedzialną za prezentację nagłówka managera zakładek. Dodatkowo pokazane zostały interfejsy implementowane przez klasę nagłówka w celu umożliwienia interakcji z użytkownikiem.

Klasa TabsPane dziedziczy po klasie JPanel co czyni ją pełnoprawnym elementem GUI budowanego z pomocą biblioteki Swing. Dodatkowo klasa JPanel zapewnia mechanizmy czyniące TabsPane kontenerem zdolnym przechowywać i prezentować elementy dzieci.

Jeżeli zakładka dodawana do obiektu TabsPane implementuje interfejs TabsPaneTab to będzie notyfikowana o tym czy została właśnie ustawiona jako aktualnie prezentowana zakładka managera zakładek, czy przestała taką być. Wszystkie inne obiekty dziedziczące po klasie Component z biblioteki Swing są prawidłowymi kandydatami do bycia zakładkami, jednak nie będą notyfikowane o powyższych zdarzeniach.
Managery zakładek biorą aktywny udział w mechanizmie przeciągania i opuszczania zakładek Drag And Drop. Sam mechanizm został szczegółowo opisany w dalszej części tej pracy. Nadmienić jednak należy, że elementem aktywnym managerów zakładek względem mechanizmu DnD jest obiekt nagłówka managera zakładek TabsPaneHeader. Implementuje on interfejs MyGuiDnDReady, który pozwala mu zarejestrować się wewnątrz głównego mechanizmu sterowania procesem DnD i brać czynny udział w procesie decydowanie o tym skąd można przeciągania i gdzie opuścić elementy GUI zbudowane na podstawie biblioteki MyGUI.

Na potrzeby środowiska JET klasa TabsPane została rozszerzona przez dwie klasy pochodne:

- CenterTabsPane
- UtilTabsPane

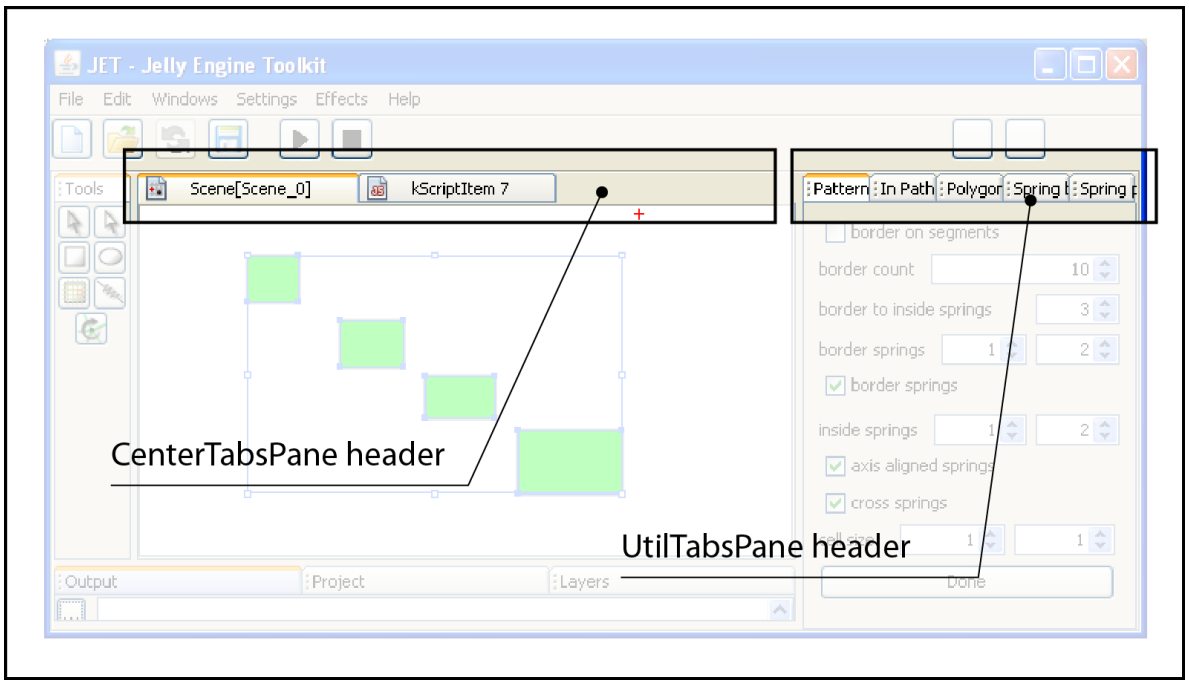
Pierwsza służy do dokowania paneli prezentujących scenę symulacji oraz edycji skryptów. Nazwa pochodzi stąd, że zwykle panele tego rodzaju umieszczane są w centrum obszaru roboczego aplikacji. Druga służy do grupowania paneli prezentujących zakładki narzędzi. Te dwie grupy managerów zostały rozróżnione ponieważ przechowywane przez nie panele są inaczej wykorzystywane przez użytkownika, a przez to wymagają innego sposobu interakcji.

Wizualnie rozróżnienie to dotyczy sposobu prezentacji nagłówków w managerach paneli, które decydują o tym w jaki sposób wyświetlane są tytuły zakładek. Sposób prezentacji nagłówka zaszyty jest wewnątrz implementacji zagnieżdżonej klasy TabsPaneHeader. Każda z klas rozszerzających klasę TabsPane może korzystać z domyślnego nagłówka zdefiniowanego wewnątrz klasy TabsPane.TabsPaneHeader, lub tak jak dzieje się w wypadku klas CenterTabsPane i UtilTabsPane może go rozszerzyć i zdefiniować własną jego implementację, co sprowadza się do zmiany sposobu prezentacji nagłówków zakładek w metodzie odrysowującej nagłówek managera zakładek paintComponent(Graphics g).

W CenterTabsPane tytuł zakładki zajmuje dokładnie tyle miejsca ile potrzeba na jego wyświetlenie, te tytuły które nie mieszczą się w szerokości kontenera managera zakładek schowane są poza jego krawędziami. Dostać się do nich można poprzez przewijanie tytułów w lewo bądź w prawo. Dodatkowo panele centralne można zmaksymalizować, tak aby były prezentowane na możliwie największej powierzchni roboczej ekranu.

W przypadku UtilsTabsPane tytuły zakładek zajmują, po równo, całą szerokość panelu-kontenera niezależnie od ich liczby. Pozwala to na szybki dostęp do każdego z narzędzi nawet wtedy, gdy jest ich tak wiele, że będąc wyświetlane wewnątrz kontenera CenterTabsPane nie wszystkie byłyby widoczne na ekranie.

Na następnej stronie przedstawiono różnicę w wyglądzie headerów komponentów CenterTabsPane oraz UtilTabsPane:



Rysunek 3.3.2.1.: Prezentacja wyglądu headerów różnych managerów zakładek: CenterTabsPane oraz UtilsTabsPane. Źródło: własne.

Powodem takiego sposobu prezentacji jest fakt, że użytkownik dużo częściej przełącza się między zakładkami narzędzi aniżeli zakładkami centralnymi (sceny/edycji skryptu/ dokumentu). Dlatego warto poświęcić elegancję prezentacji względem szybkości dostępu do danej zakładki.

3.3.3. Mechanizm Drag And Drop

Mechanizm zarządzający procesem przeciągnij i opuść zaimplementowany jest wewnątrz klasy MyGUIManager oraz przez wystawienie interfejsu MyGUIDnDReady. Pozwala on na przemieszczanie zakładek aplikacji i dokowanie ich w innych miejscach obszaru roboczego.

Komponenty GUI rejestrują się w mechanizmie DnD przez wywołanie metody MyGUIManager.registerDnDReady(). Powoduje to dodanie obiektu listener'a do listy listener'ów komponentu, tak że ten może brać udział w procesie przeciągania. Aby komponent mógł zostać zarejestrowany jako potencjalny uczestnik mechanizmu DnD musi implementować interfejst MyGUIDnDReady.

Poniżej znajduje się opis funkcjonalności deklarowany przez ten interfejs:

- void registerGuiManager(MyGuiManager manager); - pozwala komponentowi na kontakt z zarządzającym nim MyGUIManager'em
- void dndRemove(Object o); - usuwa komponent-dziecko w momencie jego dokowania w nowej lokacji wskazanej przez użytkownika
- Point getDnDLocationOnScreen(); - zwraca lokację komponentu względem obszaru roboczego
- Rectangle getDnDBounds(); - zwraca prostokąt otaczający komponent wyrażony w przestrzeni obszaru roboczego
- JComponent getObjectToPickAt(Point p); - zwraca komponent-dziecko dostępny do przeciągania pod wskazaną lokalizacją
- boolean isValidDropLocation(Point p); - zwraca informację czy wskazana przez użytkownika lokacja pozwala na upuszczenie przeciąganego komponentu
- boolean accepts(Object o); - służy do sprawdzania czy komponent danego typu może zostać upuszczony na aktualny komponent MyGUIDnDReady.
- abstract boolean dropObjectAt(Object o, Point p); - dokuje przeciągany komponent we wskazanym miejscu aktualnego komponentu MyGUIDnDReady
- MyGuiDnDDrawable getFutureLayoutDrawable(Point p); - zwraca obiekt, który wizualnie prezentuje ramkę pokazującą przyszłe położenie przeciąganego komponentu, gdyby został on opuszczony na aktualny komponent MyGUIDnDReady we wskazanym miejscu

Wszystkie te metody są wywoływane przez MyGUIManager'a w odpowiednich momentach procesu DragAndDrop. Każdy komponent implementujący ten interfejs może udostępniać swoje komponenty potomne do przeciągania i dokowania w innych miejscach obszaru roboczego.

3.3.4. Centrum sterowania DnD - klasa MyGUIManager

MyGUIManager jest singletonem odpowiedzialnym za wykrywanie gestu rozpoczynającego proces przeciągania oraz za wykonanie czynności doprowadzających do zadokowania przeciąganego komponentu w momencie wykonania gestu kończącego ten proces. Gest rozpoczęcia polega na przeciągnięciu myszy, z wciśniętym lewym przyciskiem, na odległość większą niż 35 pikseli. Gdy zostanie wykryty, komponent

implementujący interfejs MyGUIDnDReady, nad którym został wykonany jest proszony o przekazanie managerowi referencji do komponentu-potomka, który zostanie przeciągnięty i opuszczony w nowe miejsce. Od tej pory MyGUIManager dba o to, aby nad każdym komponentem GUI, nad którym użytkownik może upuścić aktualnie przeciągany komponent wyświetlana była ramka pokazująca przyszłe położenie przeciąganego komponentu. W momencie, gdy użytkownik puści lewy przycisk myszy - MyGUIManager znajduje komponent MyGUIDnDReady, nad którym wystąpił gest upuszczenia, i poprzez wywołanie metody dropObjectAt(Object o, Point p) czyni przeciągany komponent jego potomkiem.

Klasy implementiujące interfejs MyGUIDnDReady to następujące managery zakładek:

- MultiSplitPaneCell
- TabsPaneHeader
- CenterTabsPaneHeader
- UtilsTabsPaneHeader

Dlatego jedynymi przeciągalnymi komponentami GUI w aplikacji są panele-zakładki. Elementami aktywnymi względem mechanizmu DragAndDrop są nagłówki managerów zakładek, a nie same managery, ponieważ to nagłówki stanowią interfejs użytkownika mówiący o tym co jest wewnątrz managera zakładek. Sam manager wyświetla na raz tylko jedną zakładkę przez co niemożliwe byłoby podniesienie i przeciąganie zakładki innej niż aktualnie prezentowana przez managera.

Nagłówki managerów są zdefiniowane w postaci chronionych klas zagnieżdżonych wewnątrz klas managerów, dzięki czemu strukturalnie rzecz biorąc to managery zakładek są elementami aktywnymi względem procesu DnD, jednak oddzielenie ich implementacji od implementacji managerów czyni całą konstrukcję łatwiejszą w utrzymaniu i ewentualnym rozszerzaniu.

3.4. Model danych sceny

Zasada KISS (Keep It Simple Stupid) [34] polega na tym, aby robić rzeczy w możliwie prosty sposób. W wypadku programowania sprowadza się to do pisania aplikacji w taki sposób, aby ograniczyć liczbę klas projektu oraz aby algorytmy wykorzystywane wewnątrz ich implementacji były możliwie krótkie i proste do zrozumienia. Dodatkowo zależności pomiędzy klasami powinny być możliwie proste.

Projektując model danych elementów sceny starano się trzymać tej zasady. Zaowocowało to ujednoliceniem sposobu w jaki programista/artysta podchodzi do tworzenia i pracy ze wszystkimi elementami sceny.
W osiągnięciu tego celu bezpośrednią pomocą było wykorzystanie zasad polimorfizmu [35] świetnie wpisującego się i znajdującego praktyczne zastosowanie w przypadku zastosowanego modelu danych elementów sceny.

Zdecydowano się na projekt, w którym wszystkie elementy u źródła są jednakowe, to znaczy że wszystkie możliwe elementy sceny dziedziczą po tej samej klasie bazowej Item. Pozwala to na uproszczenie mechanizmów rządzących pracą narzędzi, które operują na elementach sceny symulacji.
Dotyczy to wszystkich rodzajów mechanizmów: od narzędzi do edytowania geometrii, poprzez definicję wyglądu (barwy, tekstury), po mechanizmy zajmujące się zapisywaniem i wczytywaniem elementów sceny.

Środowisko, dla każdej sceny z osobna, udostępnia użytkownikowi płaską przestrzeń geometryczną, na której możliwe jest rozlokowanie jej elementów. W fazie projektowej, czyli wtedy gdy programista tworzy bądź edytuje już utworzone plansze gry, każdy z tych elementów to kształt geometryczny opisany przez krzywą Bezier'a. Krzywe Bezier'a charakteryzują się tym, że w bardzo przyjazny dla użytkownika sposób opisują skomplikowane krzywe, dobrze modelujące kształty występujące w rzeczywistym świecie. Daje to dużą swobodę w tworzeniu przeróżnych konstrukcji geometrycznych przy minimalnym wkładzie pracy użytkownika. Więcej o krzywych Bezier'a można znaleźć w pozycji [36].

3.4.1. Podstawowe elementy sceny - klasa abstrakcyjna Item

Projekt danych sceny definiuje kilka typów elementów, z których każdy jest szczególnym przypadkiem klasy Item. Element bazowy hierarchii opisany jest przez tą właśnie abstrakcyjną klasę. Definiuje ona podstawowe własności geometryczne takie jak krzywa beziera elementu oraz kilka własności używanych przez edytor sceny w trybie edycji do pracy z nimi: to czy element jest widoczny, czy jest zaznaczony oraz czy jest zablokowany.

Ponadto klasa Item troszczy się o implementację mechanizmów pozwalających na budowę hierarchii drzewiastej z poszczególnych elementów sceny. Elementem umożliwiającym budowę takiej struktury jest klasa GroupItem, która posiada listę swoich elementów dzieci. Każdy element sceny posiada referencję do jakiejś instancji klasy GroupItem będącej jej rodzicem.
Każda scena projektu posiada tzw. grupę najwyższą (TopGroup), która jest niejawnie obecna wewnątrz sceny i stanowi korzeń drzewa elementów sceny. Jej niejawność polega na tym, że nie jest prezentowana wewnątrz panelu „Layers” prezentującego drzewo elementów sceny, ale jest punktem wejściowym dla wszystkich algorytmów operujących na drzewie elementów sceny, takich jak np. wyszukiwanie elementów na podstawie ich nazwy.

Dodatkowo w celu ułatwienia pracy z drzewem elementów sceny klasa Item posiada metody pozwalające iterować po elementach sąsiadujących z aktualnie interesującą nas instancją klasy Item: getSibling(), getPriorSibling(), oraz w wypadku gdy mamy do czynienia z elementem typu GroupItem metoda getChildList(). Metody te kolejno zwracają element sceny będący elementem następującym względem aktualnego elementu sceny, elementem go poprzedzającym oraz listą elementów dzieci tego elementu. W celu umożliwienia iteracji po hierarchii drzewa „w górę”, czyli od interesującej nas instancji klasy Item do korzenia drzewa, dostępne jest publiczne pole parent zdefiniowane wew. klasy Item. Jeżeli pole to równe jest null oznacza to, że dostaliśmy się do instancji klasy Item będącej korzeniem drzewa elementów sceny, bądź element ten nie należy do żadnego drzewa elementów sceny.

Ponieważ klasa Item definiuje geometrię elementu, posiada ona także podstawową implementację pozwalającą na edycję tej geometrii. Praca z obiektami geometrycznymi wewnątrz środowiska polega na manipulacji nimi za pomocą macierzy przekształceń. Rachunek macierzowy w przestrzeni 2D jest relatywnie nieskomplikowany względem możliwości jakie daje w celu ujednolicenia sposobu podejścia do opisu operacji geometrycznych wykonywanych na obiektach.
Dlatego klasa Item definiuje dwie metody pozwalające zmieniać kształt krzywej bezeira elementu, są to metody:

- transform(Matrix m)
- transform(Matrix m, Collection<Integer> segIndices)

Implementacja pierwszej z nich polega na zastosowaniu przekształcenia geometrycznego zawartego w macierzy do wszystkich segmentów krzywej bezier'a. Pozwala to na wykonywanie transformacji homogenicznej na wybranej krzywej.

Druga metoda stosuje transformację tylko względem wybranych segmentów krzywej, przez co możliwa jest jej dowolna deformacja. W tym przypadku oprócz macierzy przekształceń należy przekazać informację o tym, które z segmentów krzywej beziera mają zostać zmienione.

Wykorzystanie podejścia macierzowego pozwala w bardzo prosty sposób zatomizować akcje zmieniające własności geometryczne elementów, co bardzo dobrze wpisuje się w model działania modułu UndoRedoCenter, który bazuje na takich właśnie atomowych akcjach. W celu stworzenia obiektu UndoRedo przeprowadzającego transformację wybranego elementu, wszystko czego potrzeba to referencja do zmienianego elementu, macierz przekształceń oraz ewentualnie indeksy segmentów transformowanej krzywej. Gdy zajdzie potrzeba cofnięcia transformacji, wystarczy jedynie obliczyć macierz odwrotną do bazowej i zastosować ją jako macierz transformacji, co przywróci kształt krzywej bezier'a do postaci sprzed wykonania akcji transformacji.
Powyższy mechanizm zostanie dokładnie opisany w dalszej części pracy poświęconej narzędziu selekcji.

Ponadto wewnątrz implementacji klasy Item znajduje się kontener przechowujący efekty nałożone na instancję elementu. Efekty zostaną dokładnie opisane w dalszej części pracy, teraz istotne jest jedynie to, że klasa Item posiada listę zdolną przechowywać pewien zbiór takich efektów oraz że służą one do manipulowania własnościami geometrycznymi, graficznymi oraz fizycznymi elementów sceny, którym zostały przypisane.

Poniżej przedstawiono diagram UML prezentujący opis hierarchii klas reprezentujących poszczególne rodzaje elementów.

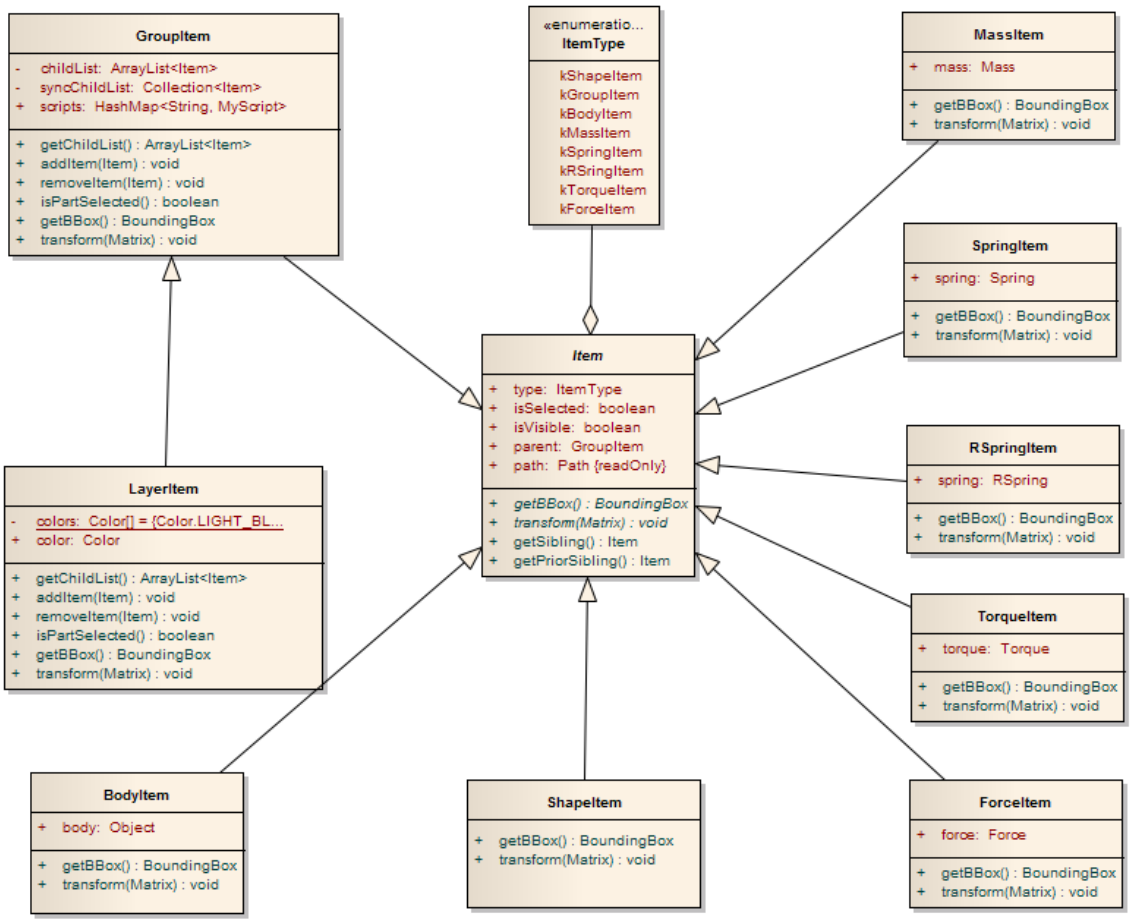


Diagram 3.4.1.1.: Diagram prezentujący strukturę klas opisujących elementy projektowe sceny tzw. Itemy.
Wszystkie dostępne typy itemów dziedziczą po klasie bazowej Item, która definiuję podstawowe własności wspólne dla wszystkich elementów, np.: kształt geometryczny oraz atrybuty elementu takie jak selekcja i widzialność.

3.4.2. Elementy typu GroupItem oraz LayerItem

Sercem modelu danych jest wcześniej opisana klasa Item.
Drugim według ważności elementem jest klasa GroupItem, powalająca na grupowanie elementów sceny w struktury drzewiaste. Klasą bezpośrednio dziedziczącą po GroupItem i posiadającą specjalne traktowanie wewnątrz aplikacji jest klasa LayerItem. Definiuje ona warstwy sceny oraz jest szczególnie traktowana ze względu na zastosowanie i sposób prezentacji warstw użytkownikowi.

Warstwy na poziomie implementacji nie różnią się niczym od grup, zostały jednak wyróżnione celem umożliwienia użytkownikowi łatwiejszej organizacji drzewa elementów sceny w kategorie. Pozwalają na logiczną separację modułów budujących scenę. Projekt planszy gry platformowej z reguły składa się z kilku niezależnych elementów takich jak tło, podłoże po którym porusza się gracz, różne obiekty wrogów, przeszkód itp.. W trakcie pracy nad taką sceną wygodnie byłoby użytkownikowi, gdyby mógł prezentować tylko wybrane grupy elementów w tym samym czasie. Zgrupowanie elementów w warstwy automatycznie wyróżnia takie kategorie i pozwala zwiększyć czytelność sceny w trybie edycji. Dla przykładu model sceny prezentującej planszę gry „Mario Bros” można podzielić na następujące warstwy :

1. Tło sceny
2. Elementy HUD (ang. Head-Up Display)
3. Aktywne fizycznie elementy świata: podłoże, przeszkody
4. Wrogowie
5. Bohater

Wyróżnione powyżej elementy zaznaczono na rysunku 3.4.2.1. umieszczonym na następnej stronie.

Rysunek 3.4.2.1.: Zrzut ekranu z gry Mario Bros. Prezentacja przykładowego podziału sceny gry na warswy, w kolejności: tło sceny, elementy HUD, podłoże i przeszkody, wrogowie, bohater. Źródło: http://pajamasdiary.cocolog-nifty.com/.

Dzięki wydzieleniu takich kategorii użytkownik w trakcie pracy nad tłem może łatwo wyłączyć widzialność lub aktywność warstw prezentujących np. wrogów i bohatera, co pozwoli mu dokładniej operować i skupić się na elementach tła sceny, które aktualnie edytuje.

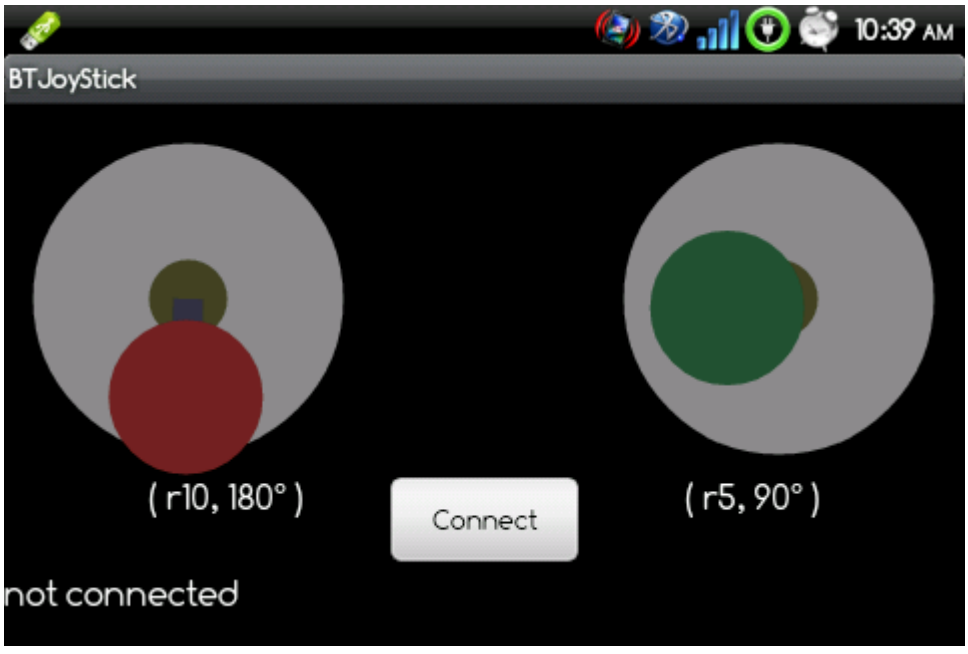
Grupowanie poza logicznym podziałem elementów sceny na kategorie pozwala ponadto na łatwe i wielokrotne powielanie takich samych elementów sceny. Dla przykładu raz stworzona grupa zawierająca elementy geometryczne definiujące wygląd obiektu wroga oraz elementy skryptowe definiujące jego zachowanie może być traktowana jako forma, z której za pomocą operacji kopiowania można stworzyć dowolną liczbę tak samo wyglądających i zachowujących się obiektów – wrogów, które wystarczy rozmieścić na scenie w odpowiednich dla nich miejscach.

3.4.3. Elementy typu ShapeItem

Klasa ShapeItem służy jedynie do reprezentacji krzywych wewnątrz sceny. Implementuje ona wszystkie metody abstrakcyjne deklarowane przez klasę Item przez co umożliwia instancjalizację podstawowych elementów udostępnianych przez edytor scen: krzywych bezier'a.

Elementy ShapeItem są pełnoprawnymi elementami sceny, można edytować ich geometrię oraz nakładać na nie efekty zmieniające reprezentację graficzną.
Ich zastosowanie ogranicza się głównie do reprezentacji elementów graficznych sceny, takich jak elementy tła bądź inne statyczne lub skryptowo animowane grafiki wektorowe.

W przyszłości planowane jest dodanie efektów, które wspólnie z elementami typu ShapeItem będą wpływać na zachowanie innych elementów sceny. Dla przykładu efekt ograniczający położenie ciała sztywnego. Jego zadaniem będzie dbanie o to, aby położenie danego ciała sztywnego ograniczone było do obszaru płaskiego wyznaczonego przez inny element typu ShapeItem. Efekt ten będzie posiadał referencję do elementu ograniczającego i przy każdym odświeżeniu swojego stanu sprowadzi element opisujący ciało sztywne do pozycji nie przekraczającej zdefiniowanej bariery. Znajdzie to szerokie zastosowanie w konstrukcji elementów wchodzących w interakcję z użytkownikiem typu graficzny joystick wyświetlany na ekranie. Przykład takiego elementu przedstawiony został na poniższym rysunku.



Rysunek 3.4.3.1.: Przykładowy graficznie reprezentowany joystick dotykowy. Źródło: http://projectproto.blogspot.com/.

Przykładowy joystick zbudowany może być z dwóch elementów ShapeItem: dużego koła reprezentującego możliwe wychylenie joysticka oraz mniejszego koła reprezentującego drążek. Zasada działania takiego joysticka polega na tym, że mniejszy okrąg śledzi ruchy kciuka gracza, a efekt na niego nałożony nie pozwala mu opuścić obszaru ograniczonego większym okręgiem. W grupie elementów budujących taki drążek znalazłby się również element skryptowy, który obliczałby wychylenie centrum drążka względem centrum obszaru ograniczającego. To wychylenie przedstawione w postaci wektora wpływałoby bezpośrednio na siłę działającą bezpośrednio na element reprezentujący bohatera gry. W ten sposób wykorzystując niedużą liczbę elementów sceny można skonstruować dość zaawansowany i popularnie wykorzystywany system sterowania głównym elementem reprezentującym bohatera gry lub na przykład położenie kamery na scenie symulacji.

3.4.4. Elementy typu BodyItem

Elementy typu BodyItem służą do opisu ciał fizycznych wewnątrz sceny. Klasa BodyItem rozszerza funkcjonalność klasy Item o możliwość modelowania elementów, które później biorą aktywny udział w symulacji przeprowadzanej przez wbudowany silnik fizyki środowiska. Wykorzystanie elementów typu BodyItem pozwala tworzyć animowane sceny gry, w których elementy poruszają się po nich i oddziałują ze sobą zgodnie z zasadami fizyki klasycznej.

Z punktu widzenia implementacji elementy BodyItem są jedynie kontenerami przechowującymi obiekty opisujące fizycznie aktywne elementy sceny. Same w sobie nie potrafią stworzyć żadnego obiektu zrozumiałego dla wbudowanego silnika fizycznego, do tego celu służą im odpowiednie efekty, które niejako rozpinają fizyczne ciała na geometrii zdefiniowanej przez elementy BodyItem.
Same efekty oraz mechanizmy ich działania zostaną dokładnie opisane w dalszej części tej pracy. W tym momencie istotne jest jedynie to, że elementy BodyItem służą do reprezentacji ciał fizycznych na scenie w trybie edycji.

W momencie uruchomienia symulacji wspomniane wcześniej centrum zarządzania symulacją SimulationCenter wykorzystuje wszystkie elementy sceny typu BodyItem do zbudowania fizycznego kontekstu symulacji JETContext. Proces ten polega na prostym

przekazaniu instancji obiektów BodyItem.body przechowywanych wewnątrz elementów BodyItem do odpowiednich list kontekstu symulacji. W trakcie symulacji przy obliczaniu kolejnych kroków animacji oraz wykonywaniu skryptów wykorzystywane są właśnie te instancje, które identyfikowane są na podstawie nazw nadanych elementom BodyItem. Sposób identyfikacji jest ważny z punktu widzenia szybkości wykonywania skryptów użytkownika operujących na elementach aktywnych fizycznie, oraz z punktu widzenia projektu sceny i sposobu jej animacji. Ponieważ podczas uruchamiania symulacji tworzona jest głęboka kopia całego drzewa symulacji sceny, elementy skryptowe i wszystkie zależności pomiędzy dowolnymi elementami sceny muszą być niezależne od konkretnych instancji. Z tego powodu zdecydowano się na identyfikowanie elementów na podstawie nazw im nadanych w trybie edycji sceny. W momencie kopiowania obiektów tworzone są zupełnie nowe i niezależne instancje, którym nadawane są nazwy elementów, na podstawie których zostały stworzone. Projekt aplikacji w żaden sposób nie ogranicza sposobu nazewnictwa ani nie wymaga pojedynczych wystąpień nazw wewnątrz drzewa. To użytkownik ma troszczyć się o to, aby nazwy elementów nie było wieloznaczne (aby jedna nazwa nie identyfikowała więcej niż jednego elementu). Takie podejście upraszcza projekt i implementację modelu elementów sceny i jednocześnie daje użytkownikowi dużą swobodę w sposobie wykorzystania mechanizmu identyfikacji poszczególnych elementów. Ważny w tym kontekście jest sposób dostawania się ko konkretnych elementów przez ich nazwę z poziomu elementów skryptowych, ale to zagadnienie opisane zostanie dokładnie przy okazji opisu elementów typu ScriptItem.

3.4.5. Elementy typu ScriptItem

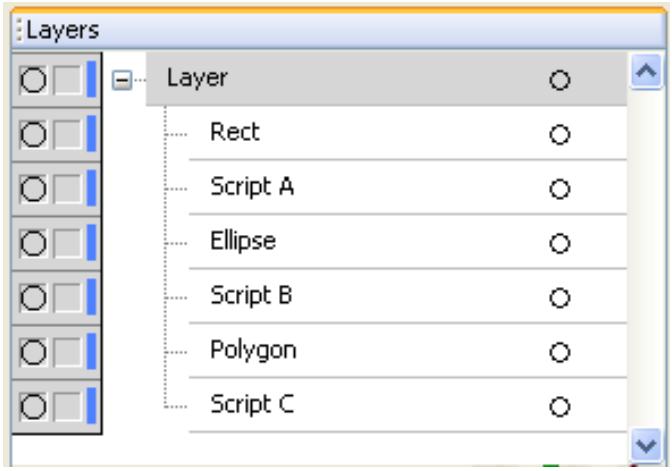
Środowisko posługuje się skryptami w języku JavaScript do ingerencji w przebieg symulacji fizycznej sceny oraz do obsługi interakcji gracza ze środowiskiem gry. Użytkownik tworząc grę może zarejestrować fragmenty skryptów jako słuchaczy na różne zdarzenia występujące w trakcie symulacji sceny. Rodzaje zdarzeń na jakie użytkownik może nasłuchiwać wymienione zostały w rozdziale 2.4.3.8.

Kod użytkownika włączany jest w projekt gry w postaci elementów skryptowych ScriptItem dołączanych do drzewa elementów sceny w ten sam sposób jak wszystkie inne elementy. Jedyną różnicą jest fakt, że skrypty w przeciwieństwie do wszystkich

innych elementów nie posiadają reprezentacji graficznej w przestrzeni sceny. Jedynym miejscem gdzie można nimi manipulować jest panel „Layers” prezentujący drzewo elementów sceny. Skrypty mogą należeć do dowolnych warstw oraz grup sceny.

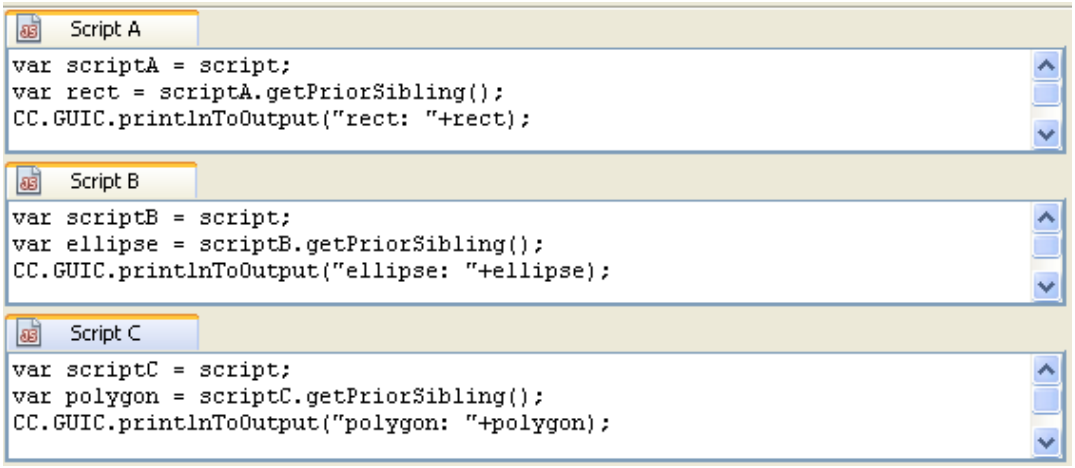
Położenie skryptów w drzewie elementów sceny jest nie bez znaczenia, i wpływa znacząco na efekty spowodowane wykonaniem ich kodu. Projekt zakłada, że użytkownik pisząc kod wewnątrz edytora tekstowego skryptu znajduje się w pewnym ściśle określonym kontekście elementu skryptowego. Kontekst ten to globalny kontekst silnika skryptowego, czyli najwyższa przestrzeń nazw, w której znajduje się specjalne pole „script” identyfikujące instancję elementu ScriptItem do którego dany skrypt należy. Pole script porównać można do globalnego obiektu „document” definiowanego przez model DOM, opisujący kontekst dostępny dla skryptów stron internetowych WWW uruchamianych po stronie przeglądarki [37].
Różnica polega na tym, że w przypadku skryptów budujących strony internetowe obiekt dokument wskazuje tę samą instancję, gdy w przypadku elementów ScriptItem obiekt scripts mimo, że znajduje się zawsze w tej samej przestrzeni nazw wskazuje inny obiekt ScriptItem dla każdego skryptu.

W celu dokładnego wyjaśnienia mechanizmu działania globalnego pola script wygodnie jest posłużyć się jakimś konkretnym przykładem. Załóżmy, że przykładowa scena posiada drzewo elementów zdefiniowane tak jak rysunku 3.4.5.1 znajdującym się na następnej stronie.



Rysunek 3.4.5.1.: Drzewo elementów sceny prezentowane przez panel „Layers” prezentujące przykładową konfigurację elementów typu ScriptItem oraz dodatkowych elementów geometrycznych. Źródło: własne.

Do tego kod poszczególnych skryptów jest taki jak pokazany na zrzucie ekranu:



Rysunek 3.4.5.2.: Zrzut ekranu prezentujący kod źródłowy każdego z przykładowych skryptów. Każdy ze skryptów odwołuje się w ten sam sposób do elementu ScriptItem go reprezentującego. Źródło: własne.

W momencie uruchomienia symulacji moduł sceny odpowiedzialny za obsługę skryptów ScriptsCenter iteruje po drzewie elementów sceny w poszukiwaniu elementów typu ScriptItem. Iteracja odbywa się rekursywnie przy wchodzeniu na niższe poziomy drzewa w momencie ich napotkania.

Gdy ScriptsCenter napotka jakiś element ScriptItem, wykonuje następujące kroki :

- ustaw globalne pole script, tak aby wskazywało element ScriptItem aktualnie wykonywanego skryptu
- wykonaj napotkany skrypt wewnątrz aktualnego kontekstu silnika skryptowego
- szukaj następnego elementu typu ScriptItem

Powyższy algorytm wykonywania skryptów można porównać do tekstowego sklejania poszczególnych elementów ScriptItem w jeden długi skrypt, przy czym jako separator w trakcie sklejania kolejnych elementów stosuje się przypisanie globalnemu polu „script” referencji do elementu ScriptItem, którego kod jest aktualnie doklejany do całościowego skryptu sceny.

Postać takiego całościowego „sklejonego” skryptu sceny w przypadku opisanym powyżej prezentuje się tak jak pokazuje to poniższy listing:

```
// dostanie się do korzenia drzewa elementów sceny
var topGroup = ItemUtils.getTopGroup(script);

// inicjalizacja globalnego pola script
var script = ItemUtils.getItemByName("Script A", topGroup);

var scriptA = script;
var rect = scriptA.getPriorSibling();
CC.GUIC.printlnToOutput("rect: "+rect);

// redefinicja globalnego pola script
script = ItemUtils.getItemByName("Script B", topGroup);

var scriptB = script;
var ellipse = scriptB.getPriorSibling();
CC.GUIC.printlnToOutput("ellipse: "+ellipse);

// redefinicja globalnego pola script
var script = ItemUtils.getItemByName("Script C", topGroup);

var scriptC = script;
var polygon = scriptC.getPriorSibling();
CC.GUIC.printlnToOutput("polygon: "+polygon);
```

Listing 3.4.5.1.: Skrypt jaki wykonuje się w przypadku uruchomienia symulacji przykładowej sceny opisanej w przykładzie przytoczonym powyżej.

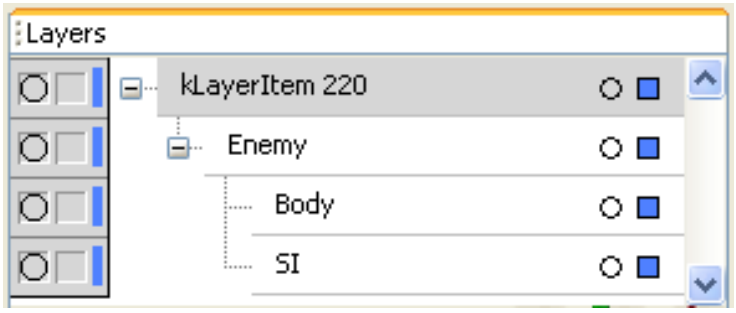
Wykonanie powyższego skryptu powoduje wyświetlenie w panelu wyjścia konsoli silnika skryptowego następującego tekstu:

```
polygon: Polygon
ellipse: Ellipse
rect: Rect
```

Listing 3.4.5.2.: Zawartość panelu konsoli prezentującego strumień wyjściowy silnika skryptowego po wykonaniu skryptów wszystkich elementów ScriptItem przykładowej sceny.

Jak wynika z powyższego listingu, pole globalne script dla każdego elementu ScriptItem wskazuje na inny element skryptowy sceny. Mechanizm takiego wystawiania referencji do aktualnego elementu skryptowego pozwala na budowanie

specjalnych struktur wewnątrz drzewa elementów sceny, charakteryzujących się możliwością łatwego ich powielania oraz prostą przenośnością pomiędzy różnymi scenami gry. Model drzewa elementów sceny został specjalnie zaprojektowany tak, aby możliwe było zbudowanie złożonego elementu sceny (grupy elementów) i późniejsze wykorzystanie go do tworzenia identycznych aczkolwiek niezależnych elementów. Dla przykładu załóżmy, że użytkownik chce stworzyć elementy sceny opisujące wrogów bohatera gry Mario Bros. Dla uproszczenia przyjmijmy, że element wroga graficznie będzie reprezentowany przez pojedynczy, fizycznie aktywny element BodyItem, oraz że cała logika jego zachowania zaszyta będzie wewnątrz pojedynczego elementu skryptowego ScriptItem. Na potrzeby tego przykładu nie trzeba definiować dokładnych mechanizmów sztucznej inteligencji rządzących zachowaniem elementu wroga na scenie względem innych jej elementów. Ważny jest jedynie sposób w jaki algorytmy zaszyte w elemencie ScriptItem będą się odnosić do innych elementów składających się na strukturę elementu wroga. Drzewo elementów sceny dla powyższego przykładu prezentuje zrzut ekranu zaprezentowany na rysunku 3.4.5.3:



Rysunek 3.4.5.3.: Zrzut ekranu panelu Layers prezentującego strukturę elementu sceny opisującego przykładowy obiekt wroga w grze Mario Bros. Element BodyItem „Body” będący reprezentacją fizyczną wroga na scenie symulacji, oraz element ScriptItem „SI” implementujący mechanizmy rządzące zachowaniem wroga zgrupowane wewnątrz elementu „Enemy” wspólnie tworzą obiekt opisujący przykładowy egzemplarz wroga na scenie gry. Źródło: własne.

Jak widać na powyższym rysunku, oba wyżej wymienione elementy zostały zamknięte wewnątrz grupy, której nadano nazwę „Enemy”. Wszystkie trzy elementy sceny: grupa „Enemy”, ciało „Body” oraz element skryptowy „SI” tworzą w tym momencie pojedynczy egzemplarz obiektu wroga. Wewnątrz elementu skryptowego mogą występować instrukcje operujące na obiekcie Body, na przykład zmieniające jego prędkość co powodowałoby przemieszczanie się wroga w lewo lub prawo.

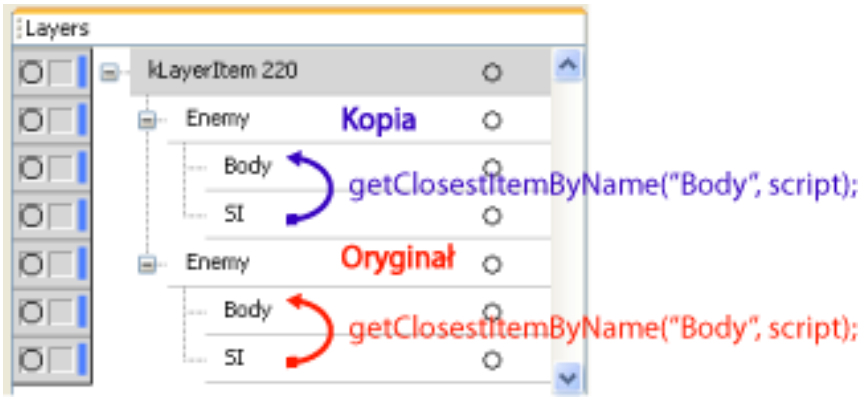
Przykładowy skrypt wpływający na prędkość ciała fizycznego zaszytego wewnątrz elementu Body mógłby wyglądać tak jak listingu 3.4.5.3. przedstawionym na następnej stronie.

```
var keyA = Keyboard.getKey("A");
var keyD = Keyboard.getKey("D");

// odnalezienie najbliższego elementu o nazwie "Body"
var bodyItem = ItemUtils.getClosestItemByName("Body", script);

JetSystem.addEventListener("onStep", function() {
      if (keyA.isPressed()) {
                  bodyItem.body.setVelocity( new Vec2d(-100,0) );
      }
      else if (keyD.isPressed()) {
                  bodyItem.body.setVelocity( new Vec2d(100,0) );
      }
      else {
                  bodyItem.body.setVelocity( new Vec2d.zero );
      }
});
```

Listing 3.4.5.3.: Fragment skryptu sterującego poruszaniem się obiektu wroga. Skrypt najpierw dostaje się do obiektów opisujących klawisze A i D, a następnie poprzez wbudowany system zarządzania zdarzeniami z poziomu skryptów nadaje im funkcje zmieniające prędkość fizycznego elementu sceny opisanego przez element BodyItem o nazwie „Body".

Dzięki temu, że skrypt w referencja do elementu „Body" jest pozyskiwana przy pomocy metody ItemUtils.getClosestItemByName(), cały obiekt wroga „Enemy" jest kopiowalny i przenośny między scenami. Jego kopiowalność polega na tym, że grupa Enemy powielona wewnątrz zachowuje spójność i zależności zapisane w skryptach. Jest to możliwe ponieważ Metoda ItemUtils.getClosestItemByName() wyszukuje elementu na podstawie nazwy w najbliższym otoczeniu elementu odniesienia przekazanego jako drugi parametr jej wywołania. Algorytm takiego wyszukiwania polega na sprawdzeniu w pierwszej kolejności wszystkich sąsiadów elementu odniesienia. Jeżeli żaden z sąsiadów nie jest elementem o poszukiwanej nazwie to w następnej koleności sprawdzane są dzieci sąsiadów elementu odniesienia (o ile sąsiedzi są grupami) i dalej rekursywnie dzieci ich dzieci itd. Jeżeli algorytm nie znajdzie szukanego elementu schodząc „w dół" drzewa elementów sceny, w drugiej kolejności zaczyna takie same poszukiwania „w górę", czyli sprawdzając po kolei rodziców elementu odniesienia i ich dzieci. Takie podejście gwarantuje, że skrypy wewnątrz skopiowanej grupy Enemy dostanie się do prawidłowego elementu Body, niezależnie ile obiektów Enemy znajdzie się wewnątrz drzewa elementów sceny.

Na rysunku 3.4.5.4. przedstawiono do którego elementu Body dostanie się każdy z elementów SI, w przypadku gdy skopiujemy na scenie jeden raz element Enemy.



Rysunek 3.4.5.4.: Prezentuje drzewo elementów sceny po jednokrotnym skopiowaniu elementu „Enemy". Kolorem czerwonym oznaczono oryginalny egzemplarz elementu wroga, kolorem niebieskim jego kopię. Strzałki pokazują, który z elementów Body zostanie odnaleziony przez wywołanie metody ItemUtils.getClosestItemByName(„Body", scripts); jako najbliższy z elementów względem elementu skryptowego, w kontekście którego metoda ta jest wywoływana. Źródło: własne.

3.4.6. Elementy typu SpringItem i RSPringItem

Element SpringItem służy do reprezentacji sprężyn na scenach gry. Klasa SpringItem tak jak wyżej opisane elementy sceny rozszerza abstrakcyjną klasę Item. Deklaruje w niej publiczne i finalne pole „spring", które przechowuje referencję do obiektu silnika fizycznego opisującego sprężynę.

Domyślna reprezentacja graficzna sprężyny na scenie w trybie edycji to odcinek, którego końce wyznaczają miejsca zakotwienia sprężyny. Od strony implementacji odcinek ten jest reprezentowany przez krzywą bezier'a przechowywaną w odziedziczonym po klasie Item polu path. Narzędzie do ręcznego tworzenia sprężyn w momencie tworzenia nowej sprężyny dba o odpowiednie zainicjalizowanie pola path dwoma segmentami, tak aby wyznaczały odcinek. Nic jednak nie stoi na przeszkodzie, aby dowolna inna krzywa została sprężyną wewnątrz sceny symulacji. Zasada rządząca definiowaniem punktów kotwicznych jest taka, że pierwszy i ostatni segment krzywej są wykorzystywane do wyznaczania tych punktów.

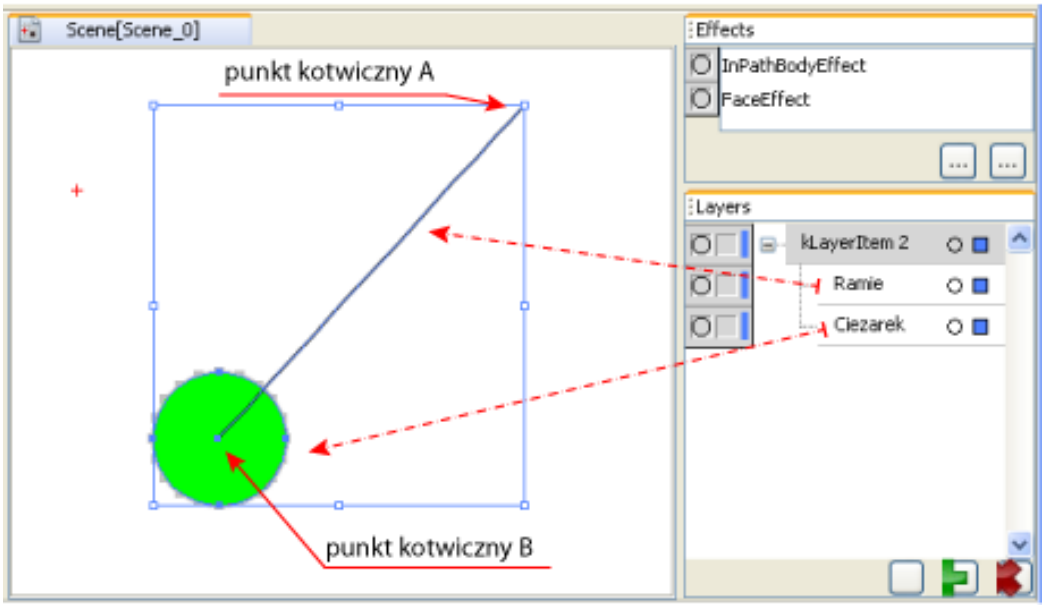
Pole „spring" klasy SpringItem opisuje jedynie przyszłe położenie sprężyny na symulowanej scenie oraz definiuje jej własności fizyczne. Instancja klasy Spring (klasy opisującej sprężynę w bibliotece silnika fizycznego), która rzeczywiście bierze udział w symulacji tworzona jest w momencie uruchomienia symulacji, kiedy moduł aplikacji SimulationCenter buduje kontekst fizyczny sceny. Dzieje się to wewnątrz wywołania

metody SimulationCenter.buildJETContext(), która iterując po Drewie elementów sceny tworzy ich fizyczne odpowiedniki oddawane pod kontrolę kontekstowi symulacji JETContext.

Kiedy algorytm tej metody iterujący po drzewie elementów sceny napotka element typu SpringItem wykonuje szereg kroków, które pozwalają stworzyć instancję klasy Spring oraz odpowiednio ją osadzić wewnątrz budowanego kontekstu JETContext. Na samym początku znajdowane są punkty kotwiczne sprężyny zdefiniowane w systemie geometrycznym sceny. Na podstawie tych punktów inny algorytm wyszukuje elementy sceny leżące bezpośrednio pod nimi. Zasada wyszukiwania tych elementów jest taka, że dla danego punktu kotwicznego zostanie znaleziony element opisujący ciało fizyczne, którego granica zawiera punkt graniczny, oraz którego odległość w drzewie elementów sceny od elementu opisującego sprężynę jest najmniejsza i element ten występuje przed elementem sprężyny w drzewie elementów sceny. Graficznie sprowadza się to do zależności, w której dla danego punktu kotwicznego zostanie znaleziony taki element sceny, który leży bezpośrednio pod sprężyną i zawiera w swojej granicy jej punkt kotwiczny. Jeżeli dla danego punktu kotwicznego nie zostanie znaleziony żaden element spełniający powyższe kryteria to dany koniec sprężyny będzie sztywno przytwierdzony do tła sceny, co oznacza że nie będzie zmieniał swojego położenia w trakcie trwania symulacji.

Przypadek taki znajduje praktyczne zastosowanie w momencie, gdy zajdzie potrzeba modelowania wahadła. Wtedy sprężyna odpowiadająca ramieniu wahadła, ma jeden koniec umieszczony nad jakimś wcześniej utworzonym elementem BodyItem modelującym ciężarek, natomiast drugi koniec sprężyny nie wskazuje żadnego elementu, przez co modeluje statyczne mocowanie ramienia wahadła do tła sceny.

Przykładowa konfiguracja wyżej opisanego wahadła zilustrowana została na rysunku 3.4.6.1 znajdującym się na kolejnej stronie.



Rysunek 3.4.6.1.: Prezentuje model wahadła, którego ramię modelowane jest przez element typu SpringItem. W celu lepszego zobrazowania położenia sprężyny na rysunku zostały jej punkty kotwiczne: A i B. Źródło: własne.

Element typu RSringItem analogicznie do elementu SpringItem opisuje sprężynę obrotową, to znaczy taką, która utrzymuje zadaną wartość kąta między swoimi ramionami. Różni się od zwykłej sprężyny liniowej tym, że oprócz dwóch krańcowych punktów kotwicznych definiuje jeszcze jeden, który wspólnie z dwoma poprzednimi opisuje kąt geometryczny (dwa odcinki współdzielące jeden koniec).

Mechanizm obsługi sprężyny obrotowej działa na dokładnie takich samych zasadach jak analogiczny mechanizm sprężyny liniowej, dlatego pominięto jego dokładny opis. Element RSPringItem nie posiada jeszcze kompletnej implementacji dlatego korzystnie z niego podczas budowy scen gry jest aktualnie niemożliwe.

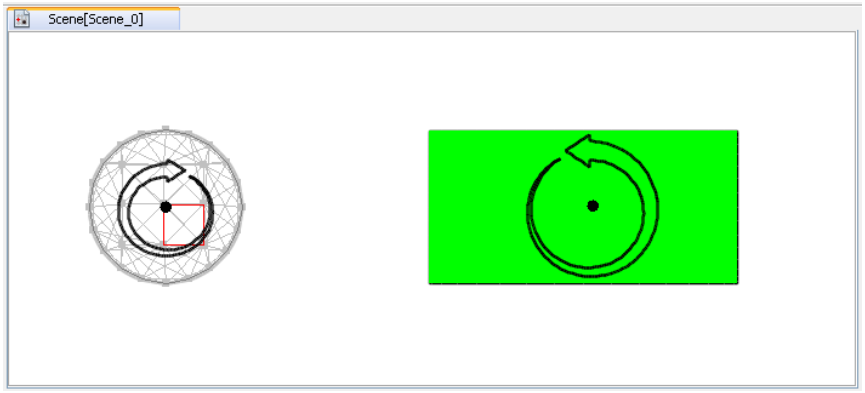
Fakt wykorzystywania przez elementy SpringItem oraz RSpringItem standardowego podejścia do definiowania geometrii, w postaci krzywych bezier'a, pozwala na ich manipulację za pomocą standardowych narzędzi środowiska do edycji krzywych. Ogranicza to ilość pracy związanej z tworzeniem dedykowanych narzędzi do edycji położenia i geometrycznych własności sprężyn i wg autora stanowi dowód na prawidłowe zastosowanie zasady KISS w przypadku projektu elementów opisujących sprężyny.

3.4.7. Elementy typu TorqueItem

Element TorqueItem daje użytkownikowi możliwość definiowania momentów obrotowych wewnątrz sceny symulacji. Klasa TorqueItem rozszerza abstrakcyjną klasę Item i definiuje pole „torque” przechowujące instancję klasy Torque opisującej implementajcę momentu obrotowego wewnątrz silnika fizycznego.

Moment obrotowy wprawia ciała fizyczne w ruch obrotowy. Implementacja elementu opisującego moment obrotowy na scenie definiuje punkt przyłożenia momentu oraz ciało, na które wywiera on swój wpływ. Punkt przyłożenia musi znajdować się wewnątrz ciała do którego moment obrotowy ma zostać zastosowany.

Reprezentacja graficzna momentu na scenie wykorzystuje pole path bazowej klasy Item definiujące geometrię każdego z elementów sceny. Pole to jest inicjalizowane w momencie tworzenia nowego elementu typu ToqrqueItem. Definiowana przez nie krzywa to ścieżka prezentująca zakrzywioną strzałkę pokazującą kierunek, w którym moment obrotowy będzie skręcać przypisane mu ciało. Krzywa ta może być geometrycznie modyfikowana (skalowana i translowana) przy użyciu narzędzi środowiska, jednak w żaden sposób nie wpływa to na sam obiekt fizyczny opisujący moment obrotowy. Jedynym ważnym czynnikiem jest centrum prostokąta otaczającego tą krzywą, gdyż ono definiuje punkt przyłożenia momentu obrotowego na scenie. Rysunek poniżej prezentuje dwa przykładowe momenty obrotowe przyłożone w centrach elementów opisujących bryłę miękką (okrąg) oraz bryłę sztywną (prostokąt).



Rysunek 3.4.7.1.: Prezentuje dwa ciała: miękkie i sztywne, na które nałożono momenty obrotowe. Ciało miękkie (okrąg) będzie obracało się w zgodnie z kierunkiem ruchu wskazówek zegara, ciało sztywne (prostokąt) obracane będzie w lewo. Punkty przyłożenia momentów obrotowych reprezentowane są przez widoczne czarne punty wewnątrz strzałek. Źródło: własne.

Dla ułatwienia odpowiedniego pozycjonowania elementu momentu obrotowego jego graficzna reprezentacja oprócz ścieżki w kształcie strzałki odrysowuje także jego punkt centralny. Jest on nieaktywny dla narzędzi edytujących geometrię elementu i służy jedynie reprezentacji centrum krzywej momentu. Obliczany jest na podstawie prostokąta otaczającego tą krzywą.

Wartość momentu obrotowego deklaruje się podczas jego definiowania na scenie w momencie tworzenia elementu za pomocą narzędzia o nazwie „Torque" dokładnie opisanego w dalszej części tej pracy. Na wartość momentu obrotowego wpływ mogą mieć także skrypty użytkownika co zapewnia możliwość jej zmiany w trakcie działania symulacji.

Mechanizm dodawania momentów obrotowych do kontekstu symulacji działa analogicznie do mechanizmów obsługujących elementy opisujące sprężyny. Algorytm budujący kontekst symulacji w momencie napotkania elementu typu TorqueItem znajduje najbliższy mu element (opisujący ciało fizyczne) w drzewie symulacji, leżący pod elementem obrotowym i „przypina" moment do tego ciała. Zasada przypinania różni się w zależności od typu ciała fizycznego. Ciała miękkie posiadają dyskretny rozkład masy zdefiniowany przez zbiór cząsteczek, ciała sztywne posiadają ciągły rozkład masy opisany przez wielokąt opisujący ciało. W wypadku brył miękkich znajdowana jest cząsteczka, która znajdujące się najbliżej centrum momentu obrotowego i ona staje staje się punktem przyłożenia momentu. Następnie w trakcie symulacji moment obrotowy oddziałuje na każdą cząsteczkę ciała miękkiego przez zmianę wartości siły wypadkowej działającej na cząsteczkę. W wypadku brył sztywnych tworzony jest dodatkowy punkt opisujący przyłożenie momentu obrotowego, który przechowywany jest wewnątrz specjalnej listy punktów zewnętrznych obiektu bryły sztywnej. Oddziaływanie momentu na ciało sztywne polega na zmianie wartości przyspieszenia kątowego ciała sztywnego.

3.5. Edytor sceny symulacji

Scena symulacji reprezentowana jest przez graficzną dwuwymiarową przestrzeń, na której użytkownik rozmieszcza elementy opisujące różne obiekty biorące aktywny udział w symulacji fizycznej sceny, elementy służące do budowania graficznej prezentacji sceny w postaci statycznych obrazów bądź grafik wektorowych oraz elementy wpływające na zachowanie innych obiektów sceny.

Układ współrzędnych sceny ma swój początek w lewym górnym rogu panelu prezentującego widok sceny. Oś x skierowana jest w prawo a oś Y do dołu, układ ten odpowiada układom współrzędnych spotykanych w większości bibliotek graficznych oraz odpowiada systemowemu układowi ekranu. Zastosowanie tego samego układu wpółrzędnych ogranicza ilość operacji matematycznych niezbędnych do przetransformowania geometrii elementów do postaci prezentowanej na scenie symulacji.

Pole powierzchni sceny jest teoretycznie nieograniczone. Wartości współrzędnych opisujących pozycję elementów sceny ogranicza dokładność z jaką Java reprezentuje liczby typu double. Użytkownik ma wgląd na część sceny symulacji wyciętą z całości oknem o wymiarach zdefiniowanych przez aktualną wielkość panelu prezentującego scenę oraz wartość skali powiększenia sceny w danym panelu. Po scenie można się swobodnie przemieszczać przeciągając ją wraz z wciśniętym prawym przyciskiem myszy, scroll myszy służy do zmieniania wartości zoom'a sceny.

3.5.1. Narzędzia edytora

Do pracy z elementami sceny stworzony został zbiór specjalnych narzędzi operujących na elementach sceny. Wszystkie akcje jakie narzędzie może wykonać na scenie z jej elementami definiuje moduł ItemsCenter środowiska. Zebranie wszystkich możliwych operacji na elementach sceny w jednym miejscu gwarantuje generację odpowiednich zdarzeń przez narzędzie.
Użytkownik komunikuje narzędziu jaką akcję powinno wykonać przez pośrednie wygenerowanie odpowiednich zdarzeń.

Na podstawie rodzajów tych zdarzeń można wyróżnić następujące grupy narzędzi :

- narzędzia edytujące geometrię elementów
- narzędzia edytujące właściwości fizyczne elementów
- narzędzia edytujące właściwości graficzne elementów
- narzędzia edytujące właściwości użytkownika elementów (selekcję, widoczność, edytowalność)

Narzędzia edytujące geometrię obiektów sterowane są zdarzeniami generowanymi z poziomu panelu graficznego sceny, opisują one akcje użytkownika generowne przy pomocy myszy i klawiatury. Wpływają na postać krzywej bezier'a edytowanego elementu, działają na elementach typu: ShapeItem, BodyItem, SpringItem oraz TorqueItem.
Narzędzia edytujące własności fizyczne wpływają na to jaki rodzaj ciała fizycznego opisuje dany element, działają z elementami typu: ShapeItem.
Narzędzia edytujące własności graficzne obiektów sceny wpływają na efekty nakładane na elementy sceny, działają ze wszystkimi rodzajami elementów gdyż dzięki temu, że efekty prezentujące elementy w trakcie symulacji współpracują ze wszystkimi typami elementów.
Narzędzia do edycji własności użytkownika decydują o prezentacji, blokowaniu edytowalności i selekcji elementów sceny w trybie edycji. Narzędzia edytujące te własności pracują ze wszystkimi typami elementów, ponieważ własności te zdefiniowane są wewnątrz klasy Item, będącej bazową klasą hierarchii dziedziczenia elementów sceny.

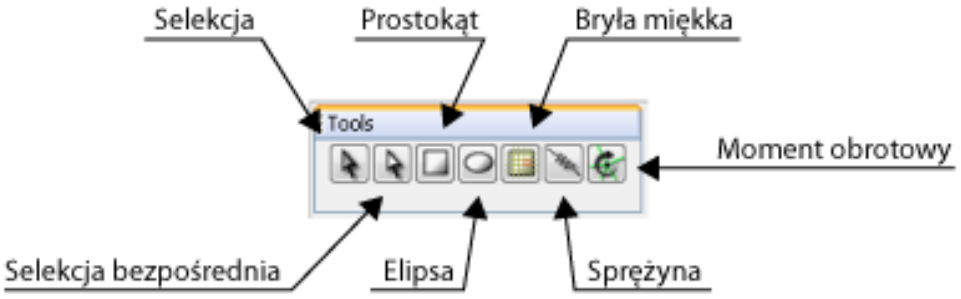
Cykl życia każdego narzędzia jest identyczny: podczas uruchamiania aplikacji metoda initTools() centrum sterowania środowiska ControlCenter tworzy instancję każdego z narzędzi i rejestruje je wewnątrz moduły zarządzającego narzędziami ToolsCenter. Moduł ten przechowuje instancje narzędzi oraz zajmuje się ich wystawieniem i prezentacją przez panel narzędzi użytkownika.

Każde narzędzie opisane jest przez klasę rozszerzającą możliwości klasy abstrakcyjnej Tool, która deklaruje interfejs narzędzia w postaci trzech metod:

- `String getName()` – zwracającej unikatową nazwę narzędzia
- `void paintTool(GraphicPanel gp)` – odrysowującą prezentację zmian wprowadzanych przez narzędzie do edytowanej sceny
- `void madeCurrent()` – metody wywoływanej w momencie, gdy narzędzie zostaje wybrane przez użytkownika jako aktualnie aktywne narzędzie.

Panel graficzny przy każdym odrysowaniu odwołuje się do modułu ToolsCenter w celu wydobycia referencji do aktywnego narzędzia i odrysowuje je na scenie przez wywołanie metody paintTool().

Poniżej przedstawiono zrzut ekranu środowiska prezentujący panel narzędzi użytkownika wraz z opisem odpowiednich ikon:



Rysunek 3.5.1.1.: Zrzut ekeranu prezentujący panel narzędzi Tools środowiska. Źródło: własne.

3.5.1.1. Selekcja

Narzędzie selekcji służy do ustawiania flagi selected na elementach sceny. Flaga selected należy do zbioru własności użytkownika i określa, które elementy sceny są brane pod uwagę w takcie wykonywania dowolnych operacji zleconych przez użytkownika.

Narzędzie opisuje pojedyncza klasa SelectionTool, która rozszerza abstrakcyjną klasę Tool. Implementacja narzędzia selection definiuje reakcje na zdarzenia generowane przez użytkownika w trakcie pracy z panelem graficznym sceny, oraz posiada zestaw mechanizmów prezentujących graficzne informacje o aktualnie wyselekcjonowanych elementach.

Wyselekcjonowanie konkretnych elementów polega na wskazaniu ich wskaźnikiem myszy lub otoczeniu ich prostokątem rozciąganym za pomocą tego wskaźnika. Do tych celów wykorzystywane są trzy rodzaje zdarzeń generowanych przez panel graficzny sceny (zdefiniowane wewnątrz klasy GPEvent):

```
public final static String kMouseDownAction = "mouse down action";
public final static String kMouseUpAction   = "mouse up action";
public final static String kMouseDragAction = "mouse drag action";
```

Listing 3.5.1.1.1.: Rodzaje zdarzeń GPEvent generowanych przez graficzny panel sceny wykorzystywane do obsługi narzędzia selekcji.

Zdarzenie kMouseDownAction oznacza, że użytkownik przycisnął jakiś przycisk myszy, narzędzie selekcji szuka wtedy elementu leżącego najbliżej wskaźnika myszy. Jeżeli odległość od takiego elementu jest mniejsza niż pewna wartość graniczna zdefiniowana w polu „minSelectionDist” klasy SelectionTool to taki element zostaje ustawiony jako zaznaczony. Dzieje się to z pomocą modułu ItemsCenter, który wystawia metodę „setSelected(Item item, boolean selected)”. Metoda ta ustawia odpowiednią flagę użytkownika wybranego elementu oraz generuje zdarzenie opisujące ackję selekcji danego elementu. To z kolei pozwala innym narzędziom środowiska dostosować prezentowane przez nie dane do nowego stanu selekcji sceny.

W przypadku kiedy znaleziony najbliższy element leży dalej niż zdefiniowana graniczna wartość odległości, narzędzie selekcji przechodzi w tryb definiowania prostokąta, który posłuży do zaznaczenia grupy elementów, które znajdą się w jego obrysie w momencie, gdy użytkownik zwolni trzymany przez siebie lewy przycisk myszy.

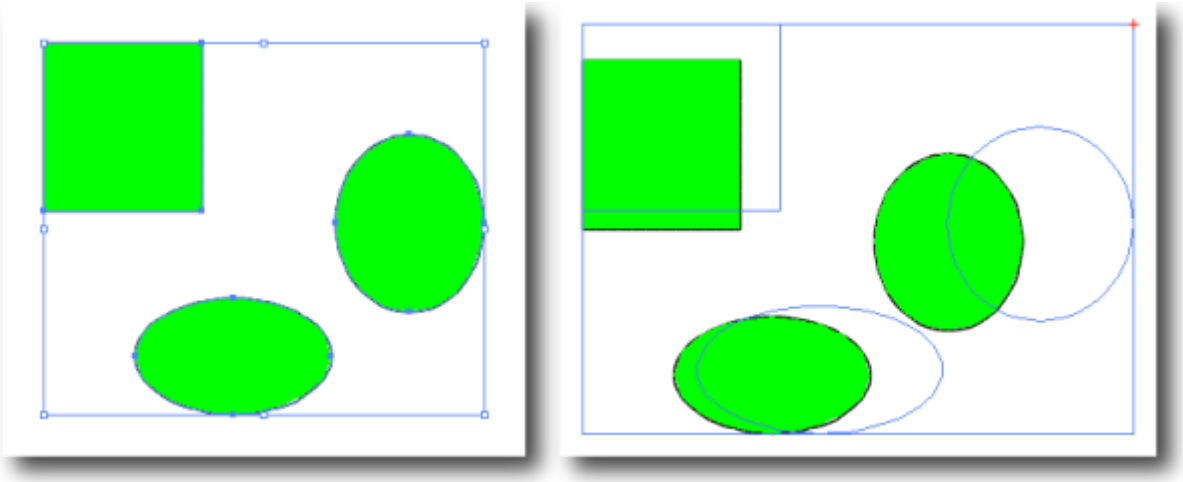
Narzędzie selekcji definiuje zestaw stanów opisujących jego aktualny tryb pracy. Służy on wewnątrznej implementacji do prawidłowego interpretowania przychodzących zdarzeń i definiuje akcje wykonywane wewnątrz mechanizmów obsługujących interakcję z użytkownikiem. Wewnątrz implementacji zestaw tych stanów reprezentowany jest przez strukturę enum (typ wyliczeniowy) zaprezentowaną na poniższym listingu:

```
protected enum Mode {
      None,
      SelectItemsInRect,
      Drag,
      Scale
}
```

Listing 3.5.1.1.2.: Enumeracja opisująca możliwe stany narzędzia selekcji. Opisuje aktualną czynność wykonywaną przez narzędzie i definiuje jak narzędzie reaguje na przychodzące zdarzenia.

Oprócz ustawiania flagi „selected”, zdefiniowanej wewnątrz klasy Item, na elementach wybranych przy użyciu narzędzia selekcji, służy ono także wykonywaniu na nich podstawowych transformacji geometrycznych: translacji, skalowania oraz obrotów. W tym celu narzędzie rysuje prostokąt otaczający dookoła aktualnie wybranych elementów. W rogach tego prostokąta oraz w połowie długości jego krawędzi renderowane są małe kwadraty wskazujące obszary aktywne prostokąta. Przeciągając te kwadraty użytkownik może niezależnie zmieniać skalę poziomą i pionową grupy wybranych elementów.

Przeciągając kontury elementów użytkownik może przesuwać grupę aktualnie zaznaczonych elementów. Na następnej stronie przedstawiono przykładowy zrzut ekranu z momentu, w którym użytkownik wykorzystuje opisaną funkcjonalność.



Rysunek 3.5.1.1.1.: Prezentuje wygląd ramki narzędzia SelectionTtool otaczającej wybrane elementy sceny. Po lewej stronie pokazany jest wygląd ramki wraz z jej aktywnymi regionami (kwadraty w rogach i środkach krawędzi) natomiast po prawej stronie prezentowany jest wygląd ramki w trakcie skalowania grupy aktualnie zaznaczonych elementów. Źródło: własne.

Narzędzie selekcji jest w stanie obsługiwać kilka scen w tym samym czasie. To znaczy tylko jedna scena może być edytowana w danym momencie, jednak informacje o aktualnie wybranych elementach oraz prostokąt je otaczający są przechowywane wewnątrz narzędzia, tak że gdy użytkownik ustawi daną scenę gry jako aktywną narzędzie selekcji załaduje odpowiednie dane opisujące stan narzędzia dla tej konkretnej sceny.

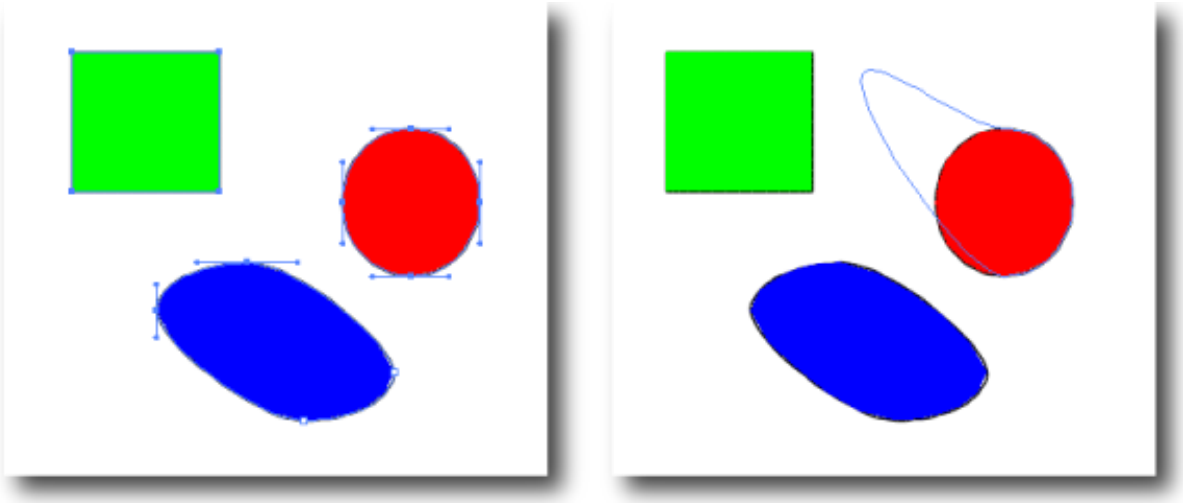
3.5.1.2. Selekcja bezpośrednia

Narzędzie do selekcji bezpośredniej pozwala na edytowanie własności geometrycznych elementów sceny. Służy ono do zmiany położenia poszczególnych segmentów oraz puntów kontrolnych krzywych opisujących elementy sceny. Jego działanie jest zbliżone do narzędzia selekcji, jednak pozwala na aplikowanie transformacji nie homogenicznych.

Narzędzie selekcji bezpośredniej wykorzystuje te same zdarzenia spływające z panelu graficznego aktualnie edytowanej sceny co wcześniej opisane narzędzie selekcji. Pozwala ono na zaznaczanie dowolnych segmentów krzywych należących do różnych elementów sceny, oraz na przesuwania wybranego zbioru segmentów po scenie symulacji.

Klasa DirectSelectionTool rozszerza abstrakcyjną klasę Tool i zapewnia mechanizmy do wykrywania które segmenty krzywych są wskazywane przez wskaźnik myszy. Tak samo jak narzędzie selekcji pozwala na edycję wielu scen jednocześnie dzięki lokalnemu przechowywaniu informacji o aktualnie edytowanych elementach każdej ze scen.

Poniżej na rysunku 3.5.1.2.1. przedstawiono zrzuty ekranu prezentujące wygląd elementów sceny w trakcie edycji przy pomocy narzędzia selekcji bezpośredniej.

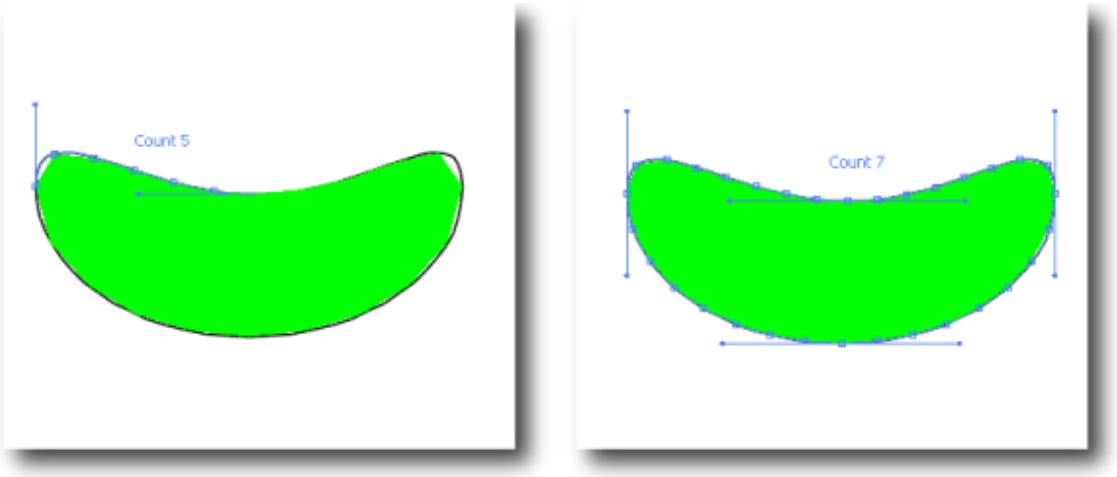


Rysunek 3.5.1.2.1.: Prezentuje wyselekcjonowane element sceny w trybie edycji przez narzędzie selekcji bezpośredniej. W lewej części prezentowane są elementy sceny wraz z zaznaczonymi ich obrysami, segmentami krzywych bezier'a oraz punktami kontrolnymi tych segmentów. Po prawej stronie zaprezentowano wygląd sceny w trakcie przeciągania jednego z segmentów czerwonego elementu. Źródło: własne.

Narzędzie selekcji bezpośredniej posiada jeszcze jedno zastosowanie. Służy do definiowania liczby tzw. Punktów pośrednich segmentów krzywych opisujących elementy. Każdy z odcinków krzywej bezier'a znajdujący się pomiędzy dwoma sąsiadującymi segmentami posiada kilka opisanych na nim punktów, które wykorzystywane są do rozpinania wielokątów na krzywych elementów. Pozwala to na tworzenie uproszczonej wersji geometrii elementu i przyspiesza operacje graficzne potrzebne do odrysowania takiego kształtu.

Użytkownik może zmienić liczbę punktów pośrednich segmentów krzywej przy użyciu scroll'a myszy w pobliżu wybranego segmentu. Dodatkowo musi trzymać wciśnięty klawisz control na klawiaturze. Jeżeli zależy mu na zmianie liczby punktów pośrednich wszystkich segmentów krzywej na raz to może to uczynić dodatkowo przytrzymując klawisz shift.

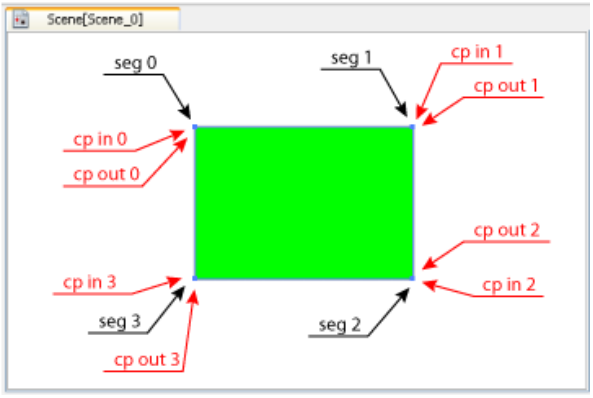
Poniżej przedstawiono zrzut ekranu prezentujący proces definiowania liczby punktów pośrednich pojedynczego segmentu krzywej oraz wszystkich segmentów w jednym momencie.



Rysunek 3.5.1.2.2.: Prezentuje narzędzie do selekcji bezpośredniej w trybie edycji punktów pośrednich krzywej opisującej geometrię elementu sceny. Po lewej narzędzie edytuje pojeduńczy segment, po prawej wszystkie segmenty w jednym momencie. Źródło: własne.

3.5.1.3. Narzędzie Prostokąt

Narzędzie implementowane przez klasę RectTool służy do dodawania elementów typu ShapeItem o kształcie prostokąta do drzewa elementów aktualnie edytowanej sceny. Tak jak wszystkie narzędzia klasa RectTool dziedziczy po abstrakcyjnej klasie Tool, dzięki czemu może zostać zarejestrowana wewnątrz modułu ToolsCenter środowiska. Na poniższej grafice zaprezentowano elementu ShapeItem w kształcie prostokąta oraz dodatkowo oznaczono cztery segmenty wraz z punktami kontrolnymi.



Rysunek 3.5.1.3.1.: Element typu ShapeItem w kształcie prostokąta ze wskazanymi i ponumerowanymi segmentami krzywej bezeir'a oraz ich punktami kontrolnymi. Źródło: własne.

Narzędzie RectTool jest bardzo nieskomplikowane w implementacji. Do komunikacji z użytkownikiem wykorzystuje te same zdarzenia generowane przez panel graficzny sceny co opisane wcześniej narzędzia. Użytkownik przeciągając wskaźnik myszy wraz z przyciśniętym lewym przyciskiem definiuje prostokąt wewnątrz sceny symulacji. Narzędzie odrysowuje cienką ramkę w kolorze aktualnie aktywnej warstwy w celu prezentacji przyszłego położenia oraz wielkości nowego elementu. Gdy lewy przycisk myszy zostaje puszczony narzędzie oblicza wymiary dla nowego elementu i tworzy go w pamięci operacyjnej na razie jeszcze nie dodając go do drzewa elementów aktywnej sceny. W celu umożliwienia cofnięcia akcji dodania nowego elementu prostokąta tworzony jest obiekt UndoRedo opisany przez pakietową klasę narzędzia URCreateRect, która implementuje mechanizmy dodające i usuwające nowy element do i z drzewa elementów sceny.
Po rejestracji obiektu UndoRedo, wykonywana jest jego metoda redo(), która faktycznie dodaje prostokąt do drzewa elementów sceny i czyni go w pełni funkcjonalnym dla użytkownika.

Poniżej przedstawiono diagram sekwencji prezentujący kroki podejmowane przez środowisko i narzędzie RectTool w celu utworzenia elementu prostokąta:

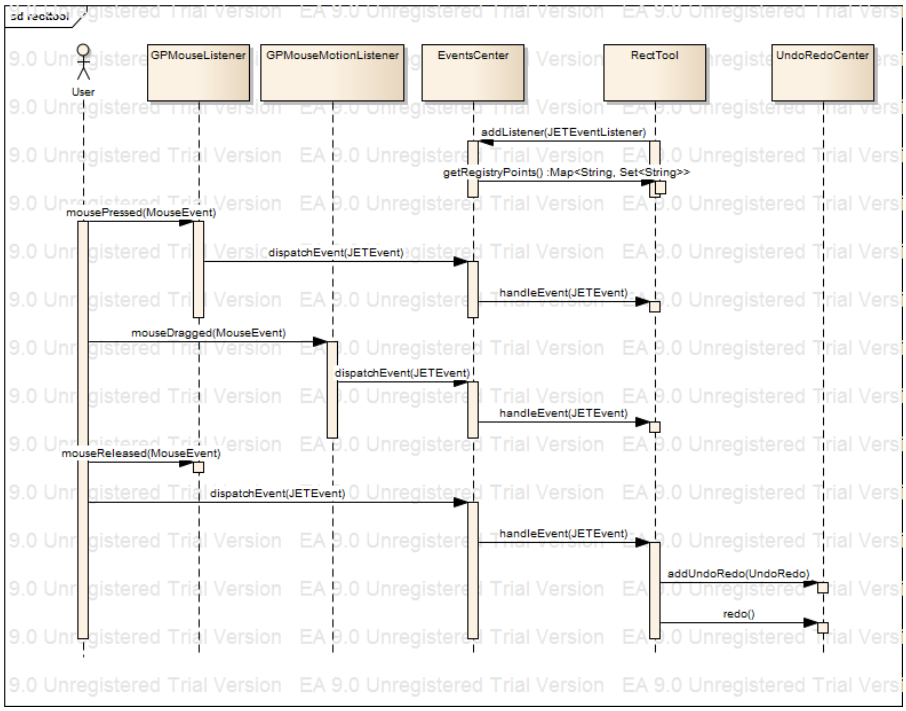


Diagram 3.5.1.3.1..: Diagram sekwencji prezentujący kroki wykonywane przez użytkownika w celu utworzenia element typu ShapeItem w momencie, gdy aktywnym narzędziem jest narzędzie do tworzenia prostokątów RectTool.

3.5.1.4. Narzędzie Elipsa

Narzędzie EllipseTool służy do tworzenia elementów typu ShapeItem prezentujących kształt elipsy. Klasa EllipseTool rozszerza abstarkcyjną klasę Tool, przez co może zostać zarejestrowana i zarządzania przez moduł ToolsCenter. Na ponizszej grafice zaprezentowano widok elementu ShapeItem w kształcie elipsy oraz dodatkowo oznaczono cztery segmenty krzywej beziera elementu wraz z punktami kontronlymi.

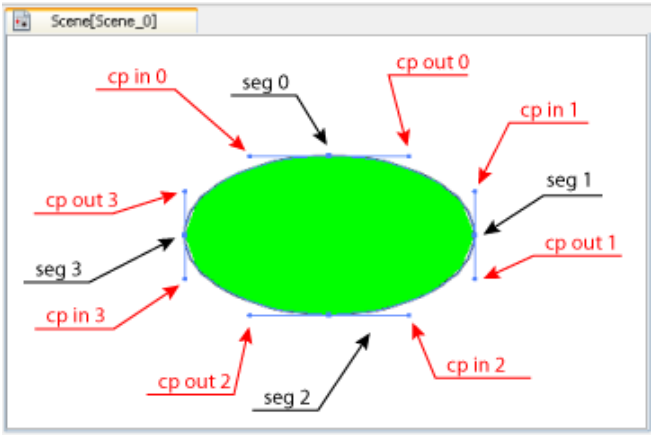
Tworzenie elipsy przebiega dokładnie według tych samych zasad co tworzenie prostokąta. Jedyną różnicą jest rysownany przez narzędzie kształt w trakcie definiowania rozmiarów elipsy oraz tworzony przez narzędzie element typu ShapeItem. Wpinanie nowo utworzonego elementu do drzewa elementów sceny przebiega również przy wykorzystaniu obiektu typu UndoRedo, jednak w wypadku elipsy opisany jest on przez klasę URCreateEllipse.

Aby opisać kształt elipsy za pomocą krzywej beziera nalezy zdefiniować cztery segmenty, oraz dokładnie określić położenie ich punktów kontrolnych. Wartość o jaką punkty kontrolne oddalone są od swoich segmentów oznaczana jest jako kappa i obliczana jest według poniższego wzoru:

$$\kappa = 4\frac{\sqrt{2}-1}{3}$$

Oddalenie punktu kontrolnego od segmentu równe jest iloczynowi wartości kappa i odpowiedniego promiania elipsy.

Rysunek 3.5.1.4.1. prezentowany na następnej stronie opisuje rozłożenie segmentów ścieżki opisującej elipsę oraz podaje pozycje, na których rozmieszczono ich punkty kontrolne.



Rysunek 3.5.1.4.1.: Element typu ShapeItem w kształcie elipsy ze wskazanymi i ponumerowanymi segmentami krzywej bezeir'a oraz ich punktami kontrolnymi. Źródło: własne.

3.5.1.5. Narzędzie Sprężyna

Klasa SpringTool opisuje narzędzie do dodawania sprężyn do drzewa elementów sceny. Tak jak wszystkie wyżej opisane narzędzia dziedziczy po abstrakcyjnej klasie Tool, dzięki czemu instancja narzędzia może być zarządzana przez moduł ToolsCenter.

Narzędzie SpringTool wykorzystuje zdarzenia generowane przez panel graficzny sceny do odgadnięcia poczynań użytkownika. Dodawana sprężyna opisana jest przez odcinek, którego końce ustawiane są w miejscach wskazanych przez użytkownika pojedynczym kliknięciem powierzchni sceny.

Klasa SpringTool wykorzystuje prosta maszynę stanów do definiowania aktualnie wykonywanej operacji. Stany te zdefiniowane są przez pole o typie wyliczeniowym zdefiniowanym jak na poniższym listingu:

```
protected enum Mode {
    None,
    SetSecondPoint
}
```

Listing 3.5.1.5.1.: Typ wyliczeniowy wykorzystywany przez narzędzie SpringTool do opisu możliwych stanów rządzących reakcją narzędzia na zdarzenia generowane z panelu graficznego sceny.

Gdy narzędzie notyfikowane jest o zdarzeniu kliknięcia panelu graficznego i znajduje się w stanie „None” zapamiętywany jest punkt kotwiczny A wskazany na panelu i

narzędzie przechodzi w stan „SetSecondPoint”. W tym stanie kolejne zdarzenie opisujące kliknięcie panelu graficznego interpretowane jest jako wskazanie drugiego punktu kotwicznego B sprężyny. W tym momencie narzędzie tworzy element typu SpringItem, ustawia jego ścieżkę tak aby reprezentowała odcinek z punktu A do B oraz tworzy obiekt URCreateSpring, który wpina element sprężyny do drzewa elementów sceny.

Użytkownik może zdefiniować ciąg sprężyn przytrzymując klawisz ctrl podczas klikania panelu graficznego. W tym wypadku narzędzie po stworzeniu jednego elementu SpringItem automatycznie przejdzie w stan „SetSecondPoint” ustawiając punkt kotwiczny A nowej sprężyny w miejsce punktu kotwicznego B poprzednio utworzonej sprężyny. Pozawala to na szybkie łączenie elementów sceny w łańcuchy.

3.5.2. Efekty

Efekty są mechanizmami służącymi do definiowania różnych właśności elementów sceny. Obiekt efektu nakładany jest na konkrenty element i jako taki może wpływać na sposób jego prezentacji graficznej, strukturę i własności fizyczne, interakcję z innymi elementami sceny i wiele innych.

Efekty mają nieograniczone możliwości wpływania na konkretne własności elementów. Każdy z elementów sceny przechowuje listę swoich efektów wewnątrz pola typu List odziedziczonego po abstarkcyjnej klasie Item.
W momencie, gdy jakaś część środowiska zmienia własności geometryczne elementu automatycznie uruchamiany jest proces odświerzania efektów tego konrentego elementu. W tym momencie efekty aplikują swoje zmiany do przypisanego elementy w pełni zautomatyzowany sposób. Dzięki temu wyeliminowano konieczność każdorazowego ręcznego przebudowywania geometrii czy wyglądu elementów sceny, zamiast tego nakłada się konkrenty efekt który sam wie jakie zmiany wprowadzić do elementu. Parametry tych zmian użytkownik definiuje w trakcie nakładania efektu na element, oraz może je później edytować za pomocą paneli narzędziowych towarzyszących efektom.

Każdy efekt środowiska musi implementować interfejs JETEffect definiujący kilka metod pozwalających środowisku nim zarządzać. Poniżej zaprezentowano definicję interfejsu JETEffect:

```
public interface JETEffect {
    // Performs all operation/object processing that is required to
apply current effect
    // to specified object.
    public void update();

      // Lets the effect accumulate items transformations
    public void transform(Matrix m);

      // Lets the effect to be copied
    public JETEffect copyEffect(Item newItem);

      // Draws effect graphics on the scene's graphic panel
    public void draw(GraphicPanel gp);

      // Enabled rendered to check wheather effect should be drawn on
scene's graphic panel
      public boolean isVisible();

    // Enabled user to turn effect's rendering on and off
      public void setVisible(boolean b);
}
```

Listing 3.5.2.1.: Definicja interfejsu opisującego zbiór wszystkich działań możliwych do wykonania na efektach nakładanych na elementy sceny.

Metoda update() to miejsce, w którym efekt aplikuje swoje zmiany do elementy do którego jest przypisany. Dla przykładu efekt nakładający na element wypełnienie kolorem (FaceEffect) w tym miejscu tworzy wielokąt, który będzie później rysowany w panelu graficzny sceny wypełniając krzywą elementy zdefiniowanym przez użytkownika kolorem.

Metoda transform() pozwala efektowi na zapamiętywanie zmian geometrii wprowadzanych do elementów przez inne moduły środowiska. Pozwala to optymalizować czas potrzebny efektowi na odświerzenie swojego stanu względem aktualnego stanu elementu. Dla przykładu efekt nakładający na ciało sztywne wypełnienie kolorem nie musi tworzyć nowego wielokąta opisującego to wypełnienie za każdym razem, gdy efekt jest odświerzany. Wystarczy że początkowo utworzony wielokąt będzie transformował na podstawie macierzy spływających do efektu przez metodę transform().

Metoda copyEffect() tworzy kopię efektu i znajduje zastosowanie w przypadku kopiowania elementów sceny. Każdy efekt posiada niezależną implementację, dlatego musi także potrafić tworzyć swoje kopie w odpowiedni sposób.

Metoda draw() wykorzystywana jest podczas odrysowywania elementu sceny, na który efekt został nałożony. W momencie odrysowywania elementu przez moduł graficzny, lista przechowująca efekty elementu jest iterowana przez mechanizmy panelu graficznego i w kolejności każdy z efektów rysuje na scenie odpowiadającą mu grafikę. Dla przykładu efekt wypełnienia odrysowuje w tym momencie wypełniony kolorem wielokąt opisany na krzywej elementu sceny.

Metody isVisible() oraz setVisible() służą do informowania mechanizmu odrysowującego efekt czy ten powinien być odrysowany. Dodatkowo środowisko posiada panel narzędziowy prezentujący efekty nałożone na zaznaczone elementy sceny. W panelu tym można definiować widzialność wszystkich efektów konkretnych elementów komponując w ten sposób wygląd sceny.

3.5.2.1. InPathBody - efekt tworzący bryłę miękką

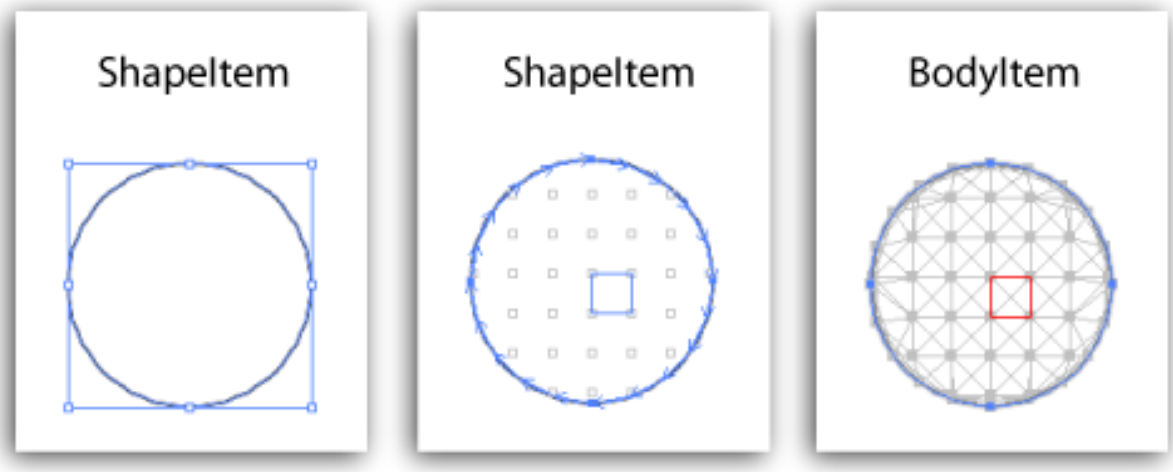
Efekt InPathBody służy do definiowania wewnętrznej struktury fizycznej brył miękkich opisanych na scenie za pomocą elementów typu BodyItem.

Efekt stosuję się z elementami typu ShapeItem, nakłada się go przy użyciu narzędzia PatternTool. W momenice nałożenia efektu na kształt geometryczny ShapeItem następuje konwersja elementu do typu BodyItem i zaaplikowanie temu nowemu elementowi efektu InPathBody.
Działanie efektu polega na rozpięciu na elemencie BodyItem siatki cząsteczek połączonych sprężynami oraz zdefiniowaniu brzegu ciała wzdłuż krzywej elementu. Utworzone w ten sposób elementy fizyczne budują bryłę miękką typu SpringBody przechowywaną wewnątrz pola „body” klasy BodyItem.

Efekt InPathBody współpracuje z dwoma niezależnymi narzędziami środowiska: narzędziem PatternTool oraz panelem użytkowym tego narzędzia.

Narzędzie PatternTool zbudowane jest na bazie abstrakcyjnej klasy Tool i umożliwia uzytkownikowi wizualne zdefiniowanie siatki cząsteczek i sprężyn nakładanych na element sceny. Proces definiowania tej struktury polega na rozciągnięciu prostokąta opisującego pojedyńczą komórkę siatki wewnątrz elementu który ma stać się ciałem miękkim. Po zdefiniowaniu komórki siatki, narzędzie wykorzystując algorytmy zaszyte wewnątrz efektu InPathBody generuje przykładowy wygląd efektu końcowego. Po zatwierdzeniu wyglądu siatki narzędzie PatternTool pobiera listę parametrów, które posłużyły do jej utworzenia i tworzy instancję efektu, którą nakłada na docelowy element sceny. Operacja ta wykorzystuje moduł UndoRedoCenter do zarejestrowania operacji konwersji elementu z typu ShapeItem do BodyItem i umożliwienia późniejszego cofnięcia tej operacji.

Na rysunku 3.5.2.1.1. znajdującym się poniżej zilustrowano kroki wykonywane przez użytkownika w trakcie nakładania efektu InPathBody na przykładowy element ShapeItem.



Rysunek 3.5.2.1.1..: Prezentuje kroki wykonywane przez użytkownika przy nakładaniu efektu InPathBody za pomocą narzędzia PatternTool. Krok 1 prezentuje element typu ShapeItem, krok drugi to definicja wzorcowej komórki siatki za pomocą narzędzia PatterTool, w tym kroku widać również przyszłe położenie cząsteczek siatki obliczone przez narzędzie PatternTool. Krok 3 to element typu BodyItem z zaaplikowanym efektem InPathBody. Źródło: własne.

Parametry opisujące jak ma wyglądać struktura siatki nakładanej na element sceny przechowywane są przez panel narzędziowy narzędzia PatternTool. Użytkownik może tam zdefiniować takie aspekty tworzonej siatki jak:

- `borderOnSegs`: definiuje czy cząsteczki brzegu ciała mają być rozłożone wg definicji punktów pośrednich segmentów krzywej elementu

- `border count`: ile cząsteczek tworzy brzeg ciała. Aktywne tylko gdy borderOnSegs jest nieaktywne

- `border to inside springs`: ile sprężyn ma łączyć pojedynczą cząsteczkę brzegu z różnymi cząsteczkami wnętrza ciała.

- `border springs`: definiuje przedział <a,b> określający co ile cząsteczek brzegu powinny występować sprężyny je łączące. Dla przykładu <1,2> oznacza, że sprężyny rozpięte na cząsteczkach brzegu będą występować co jedną oraz co drugą cząsteczkę

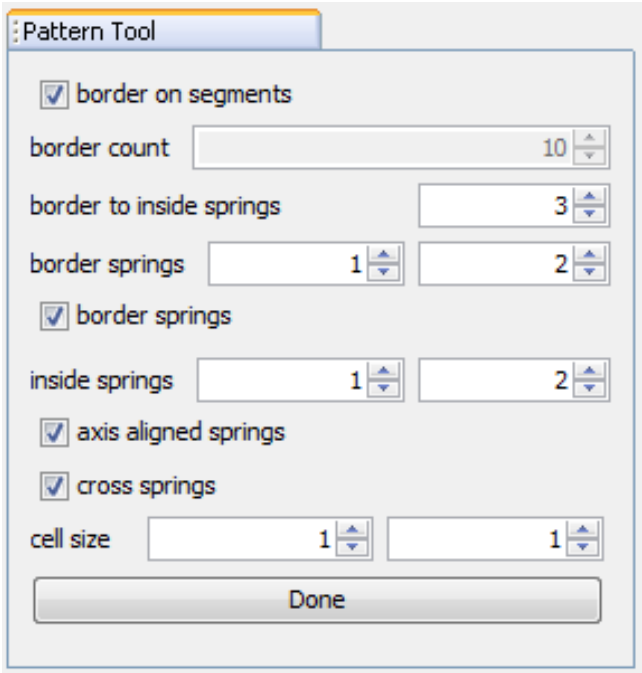
- `border springs`: flaga definiująca czy efekt ma budować brzeg ciała miękkiego

- `inside springs`: definiuje przedział <a,b> określający co ile cząsteczek będą rozpinana sprężyny wewnątrz ciała. Działa na tej samej zasadzie co definicja przedziału border springs

- `axis aligned springs`: flaga mówiąca czy wewnątrz siatki mają pojawiać się poziome i pionowe sprężyny

- `cross springs`: flaga mówiąca czy wewnątrz siatki mają pojawiać się ukośne sprężyny

Poniżej przedstawiono widok panelu narzędziowego narzędzia PatternTool:



Rysunek 3.5.2.1.2.: Wygląd panelu narzędziowego narzędzia PatternTool służącego do definiowania parametrów efektów InPathBodyItem. Źródło: własne.

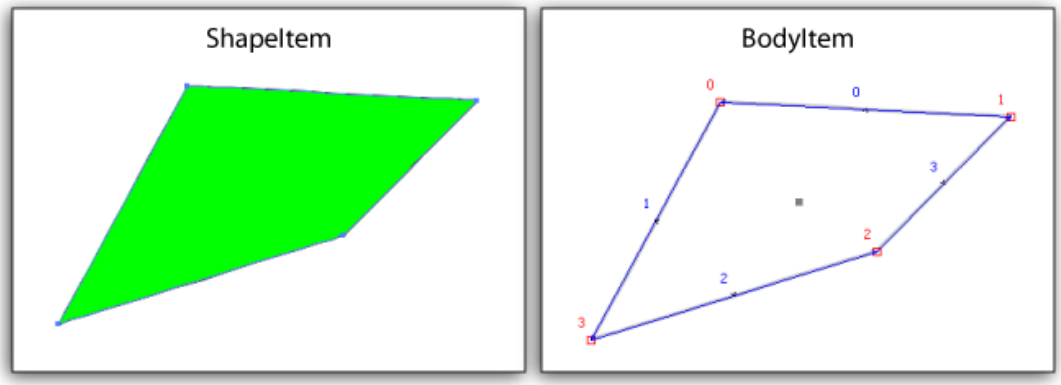
3.5.2.2. InPathRigidBody - efekt tworzący bryłę sztywną

Efekt InPathRigidBody służy do definiowania struktury bryły sztywnej. Stosuje się go na elementach typu ShapeItem. Zasada działania tego efektu jest taka sama jak efektu InPathBody z tą różnicą, że zamiast bryły miękkiej efekt ten tworzy instancję klasy RigidBody, którą wpina do elementu typu BodyItem.

Efekt InPathRigidBody współpracuje z panelem narzędziowym InPathRigidBodyOptions, który służy do nakładania efektów na zbiór elementów sceny. Panel wystawia pojedynczy parametr sterujący sposobem tworzenia ciał sztywnych: masę każdego z nich. Użytkownik, aby utworzyć bryłę sztywną zaznacza element lub grupę elementów typu ShapeItem i klika przycisk „Done" panelu InPathRigidBody. W tym momencie wszystkie zaznaczone elementy konwertowane są do typu BodyItem, oraz nakładane na nie są osobne obiekty efektu typu InPathRigidBody.

Efekt tworzy bryłę sztywną na podstawie geometrii docelowego elementu ShapeItem. Wielokąt definiujący bryłę sztywną tworzony jest na bazie segmentów i punktów pośrednich ścieżki elementu. Masa utworzonego ciała sztywnego równa jest wartości podanej przez użytkownika w polu „Mass" panelu narzędziowego InPathRigidBodyOptions.

Poniżej zaprezentowano kolejne kroki konwertowania elementu typu ShapeItem do elementu typu BodyItem:



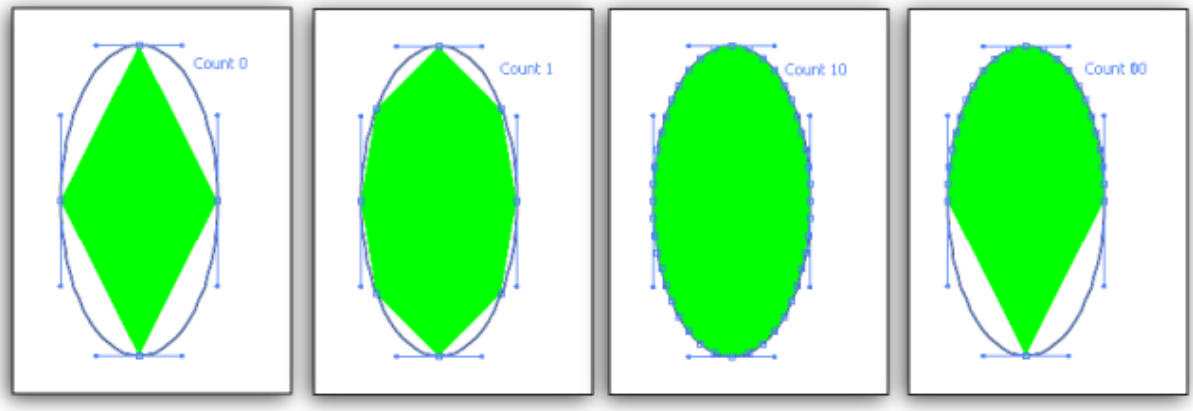
Rysunek 3.5.2.2.1.: Prezentuje element typu ShapeItem, który został przekonwertowany przy użyciu panelu narzędziowego InPathRigidBodyOptions do elementu typu BodyItem z nałożonym efektem typu InPathRigidBody budującym w nim bryłę sztywną. Na niebiesko efekt renderuje indeksy krawędzi, na czerwono indeksy wierzchołków bryły sztywnej. Źródło: własne.

3.5.2.3. PolygonFace - efekt nadający elementom wypełnienie kolorem

Efekt PolygonFace służy do nadawania elementom wypełnienia kolorem. Efekt współpracuje z dowolnymi elementami sceny, gdyż jedynym jego wymaganiem jest, aby element posiadał pole path, na którym rozciągnięty zostanie wielokąt stanowiący wypełnienie elementu. Pole path zdefiniowane jest wewnątrz klasy bazowej Item, przez co dostępne jest u każdego z elementów sceny. Klasa PolygonFace rozszerza interfejs JETEffect dzięki czemu może być wpięta do i zarządzana przez moduł obsługi efektów środowiska.

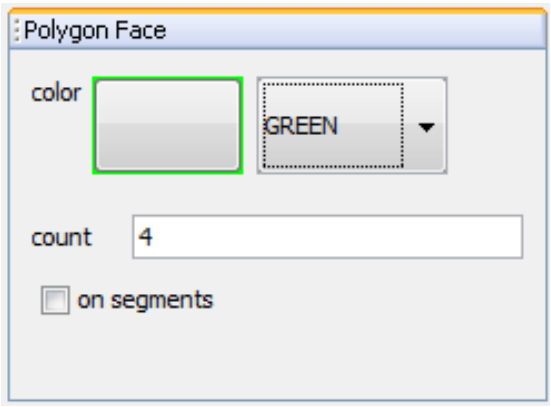
Wielokąt tworzony jest na podstawie ścieżki opisującej geometrię elementu. Jego wierzchołki znajdują się w punktach zdefiniowanych przez segmenty krzywej oraz w punkty pośrednie elementu. Użytkownik nakłada efekt na dowolną liczbę elementów na raz za pomocą panelu narzędziowego PolygonFace. Aby nałożyć efekt na element bądź grupę elementów należy zaznaczyć docelowe elementy i kliknąć przycisk „color” w panelu PolygonFace.
Mechanizmy panelu odwiedzą każdy z wyselekcjonowanych elementów i nałożą na każdy z nich osobny obiekt efektu. Jeżeli któryś z elementów będzie posiadał już taki efekt to panel PolygonFace zmieni jedynie definicję jego koloru.

Poniżej przedstawiono kilka postaci efektu PolygonFace generowanego na tym samym elemencie w zależności od liczby punktów pośrednich na każdym z segmentów ścieżki element:



Rysunek 3.5.2.3.1.: Prezentacja różnych wersji wielokąta utworzonego przez efekt PolygonFace w zależności od liczby punktów pośrednich każdego z segmentów ścieżki elementu. Źródło: własne.

Panel narzędziowy PolygonFace służący do nakładania i edytowania własności efektów PolygonFace nałożonych na elementy przedstawiony został na rysunku na następnej stronie.



Rysunek 3.5.2.3.2.: Panel narzędziowy PolygonFace służący do nakładania i edytowania efektów typu PolygonFace. Źródło: własne.

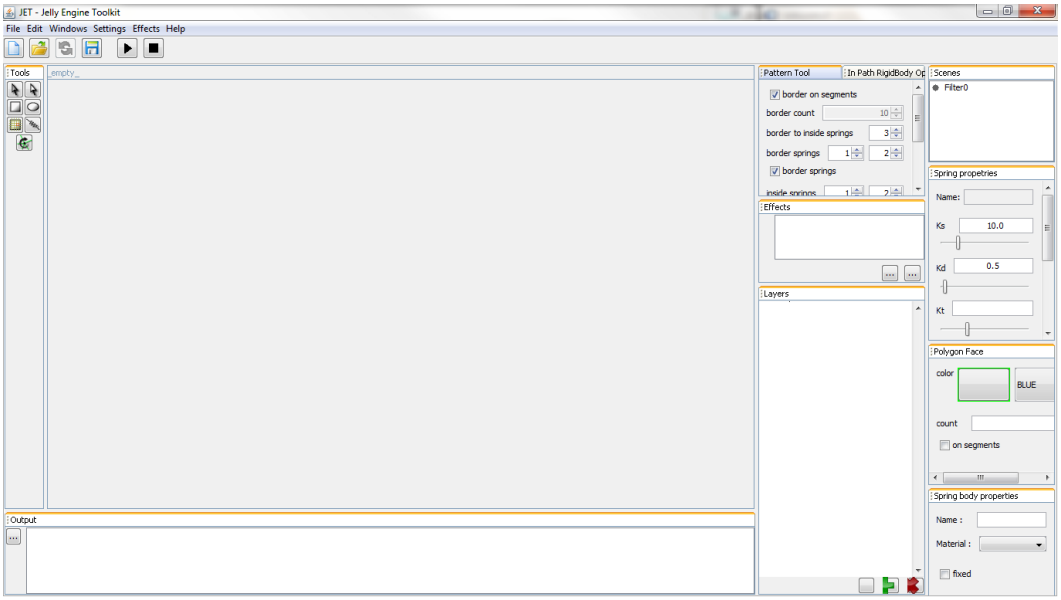
4. Prezentacja środowiska – przykładowa gra

W poprzednich rozdziałach szczegółowo przedstawiono poszczególne moduły oraz mechanizmy budujące środowisko będące sednem tej pracy magisterskiej. Proces tworzenia przykładowej gry przy użyciu tego środowiska przedstawiony zostanie przez opis czynności, które należy wykonać w celu stworzenia grywalnej sceny.

Przykładowa gra polegać będzie na sterowaniu miękką piłeczką/kulką, którą należy przemierzyć scenę gry. Będzie ona tak stworzona, aby zapewnić graczowi miłe doznania wizualne oraz zawierać będzie pewne utrudnienia, które wniosą do gry element nagrody. W trakcie gry liczony będzie czas spędzony na danej scenie. Im krótszy będzie czas przebycia całego poziomu tym lepszy wynik gracz otrzyma za jego przebycie.

4.1. Tworzenie i otwieranie sceny

Po uruchomieniu aplikacji twórcy gier ukazuje się następujący widok środowiska:

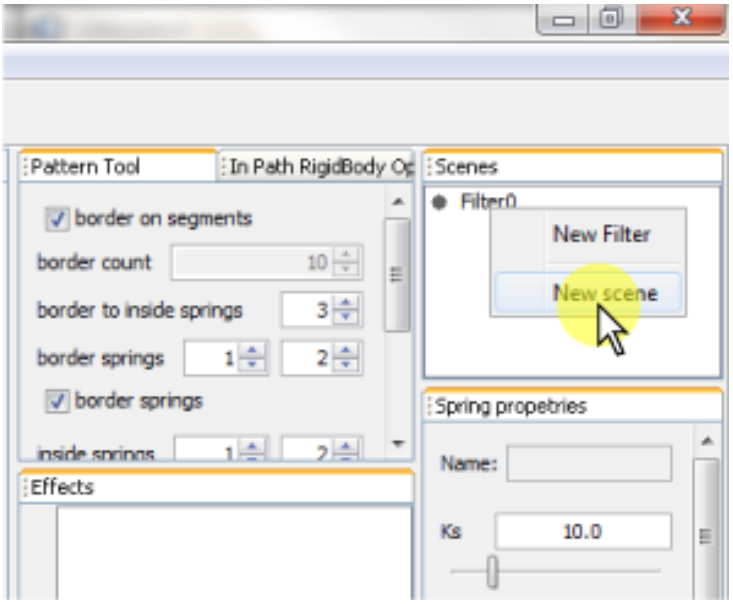


Rysunek 4.1.1.: Widok środowiska tuż po uruchomieniu. Źródło: własne.

Patrząc od lewej strony środowisko prezentuje panel narzędzi do edycji sceny. W samym środku znajduje się przestrzeń, która po utworzeniu scen będzie prezentować panele z ich zawartością. Po prawej stronie znajdują się panele narzędziowe: własności

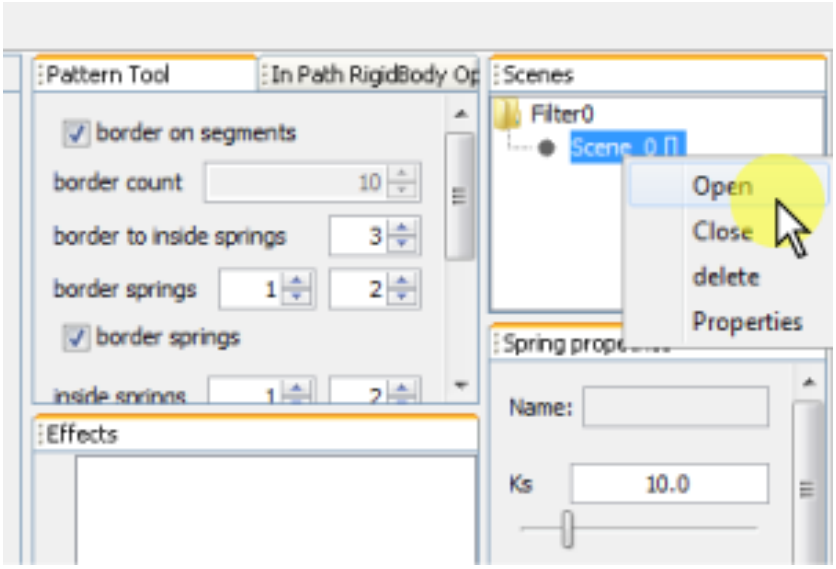
efektów (PattrnTool oraz InPathRigidBody), lista efektów zaznaczonych na scenie elementów (Effects), panel prezentujący drzewo elementów sceny (Layers), panel prezentujący sceny projektu (Scenes), panel własności sprężyn (Spring properties), panel własności efektu FaceEffect (PolygonFace) oraz panel własności brył miękkich (Spring body properties).
U dołu ekranu znajduje się panel prezentujący wyjście standardowego strumienia silnika skryptowego, natomiast u góry znajduje się menu środowiska oraz sześć przycisków służących do zarządzania projektem oraz symulacją aktywnej sceny.

W celu utworzenia sceny należy w panelu Scenes prawym przyciskiem kliknąć główny folder scen „Filter0" i wybrać opcję „New scene". Ilustracja tego manewru została pokazana na poniższym zrzucie ekranu:



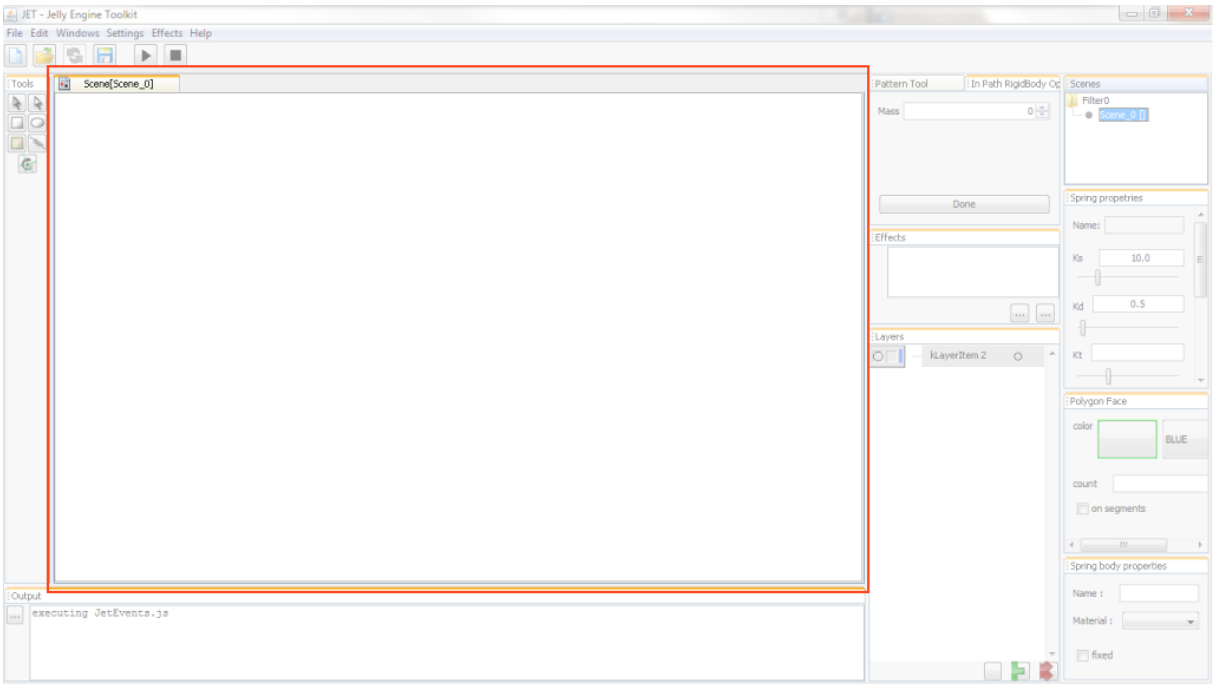
Rysunek 4.1.2.: Tworzenie nowej sceny przy użyciu panelu narzędziowego "Scenes". Źródło: własne.

Gdy scena jest już utworzona należy otworzyć ją w trybie edycji. W tym celu z menu kontekstowego sceny (dostępnego pod prawym przyciskiem myszy) należy wybrać opcję „Open", tak jak to pokazano na rysunku 4.1.3. znajdującym się na następnej stronie.



Rysunek 4.1.3.: Otwieranie sceny do trybu edycji. Źródło: własne.

Z tak przygotowanym środowiskiem można przystąpić do tworzenia geometrii sceny i pisania samej gry. Poniżej przedstawiono zrzut ekranu prezentujący wygląd środowiska z otwartą w trybie edycji sceną „Scene_0”:

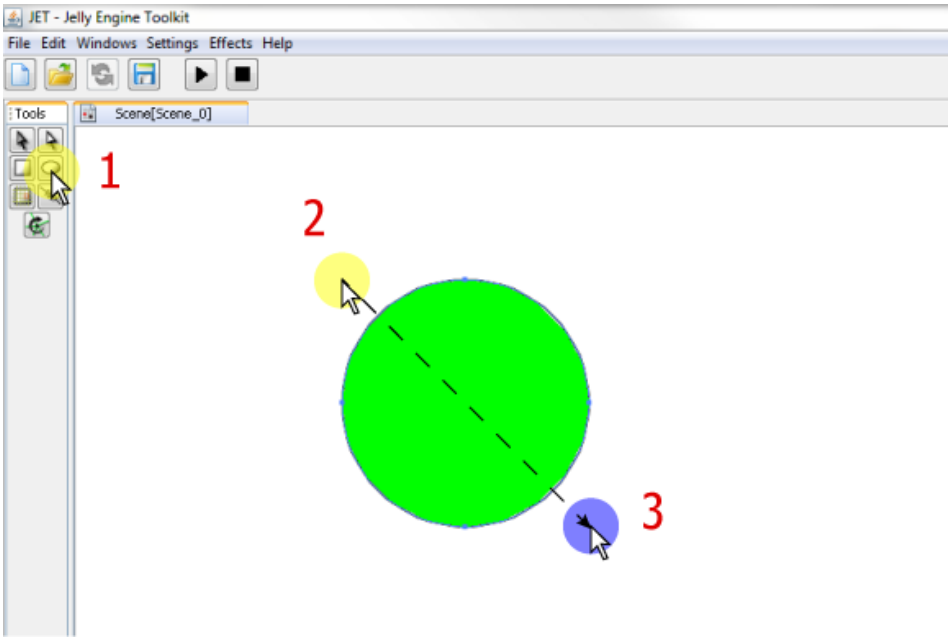


Rysunek 4.1.4.: Widok środowiska z otwartą do edycji sceną „Scene 0”. Obszar prezentujący zakładkę sceny w trybie edycji został wyróżniony czerwonym prostokątem. Źródło: własne.

4.2. Tworzenie i edytowanie elementów

Dodawanie elementów do sceny wymaga wykorzystania narzędzi użytkownika. W tej części pracy zaprezentowany zostanie proces tworzenia głównego bohatera gry: czerwonej miękkiej piłeczki oraz statycznych elementów sceny: szarego sztywnego podłoża.

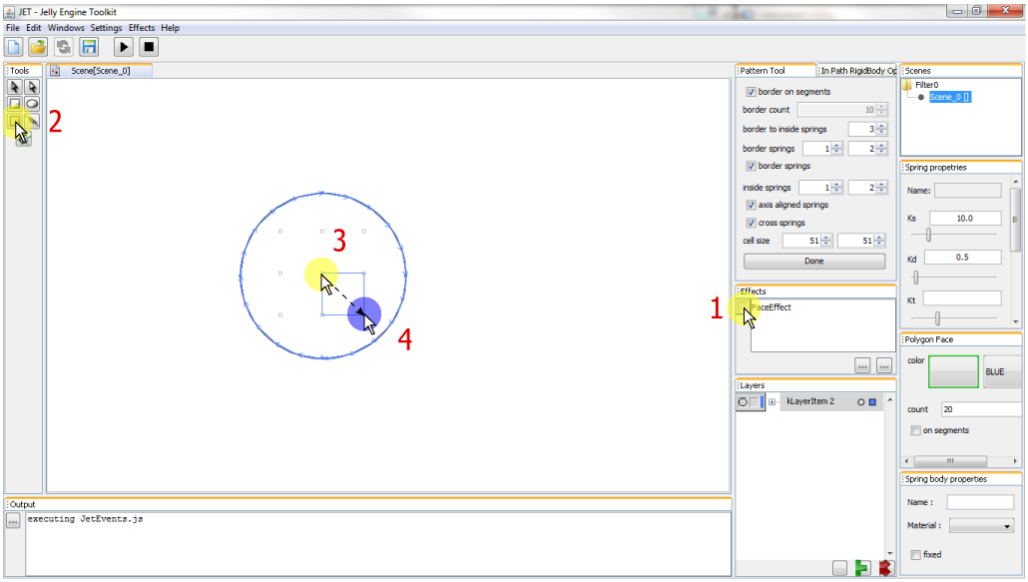
Utworzenie elementu bohatera sprowadzać się będzie do umieszczenia na scenie obiektu reprezentującego bryłę miękko w kształcie okręgu. W tym celu należy dodać do sceny element ShapeItem w kształcie okręgu. Wykonać to zadanie można przy użyciu narzędzia ellipse dostępnego do wyboru z panelu narzędziowego „Tools". Proces dodawania okrągłego elementu ShapeItem zilustrowano na poniższym rysunku. Żółte przezroczyste okręgi znajdujące się pod wskaźnikiem myszy oznaczają przyciśnięcie lewego przycisku myszy, niebieskie oznaczają puszczenie lewego przycisku. Dodatkowo kolejne akcje myszy zostały ponumerowane w celu pokazania kolejności występowania.
Przerywaną strzałką łączącą miejsca przyciskania i puszczania lewego przycisku myszy oznaczono gest przeciągania wskaźnika myszy razem z wciśniętym lewym przyciskiem myszy.



Rysunek 4.2.1.: Tworzenie element typu ShapeItem w kształcie okręgu. Źródło: własne.

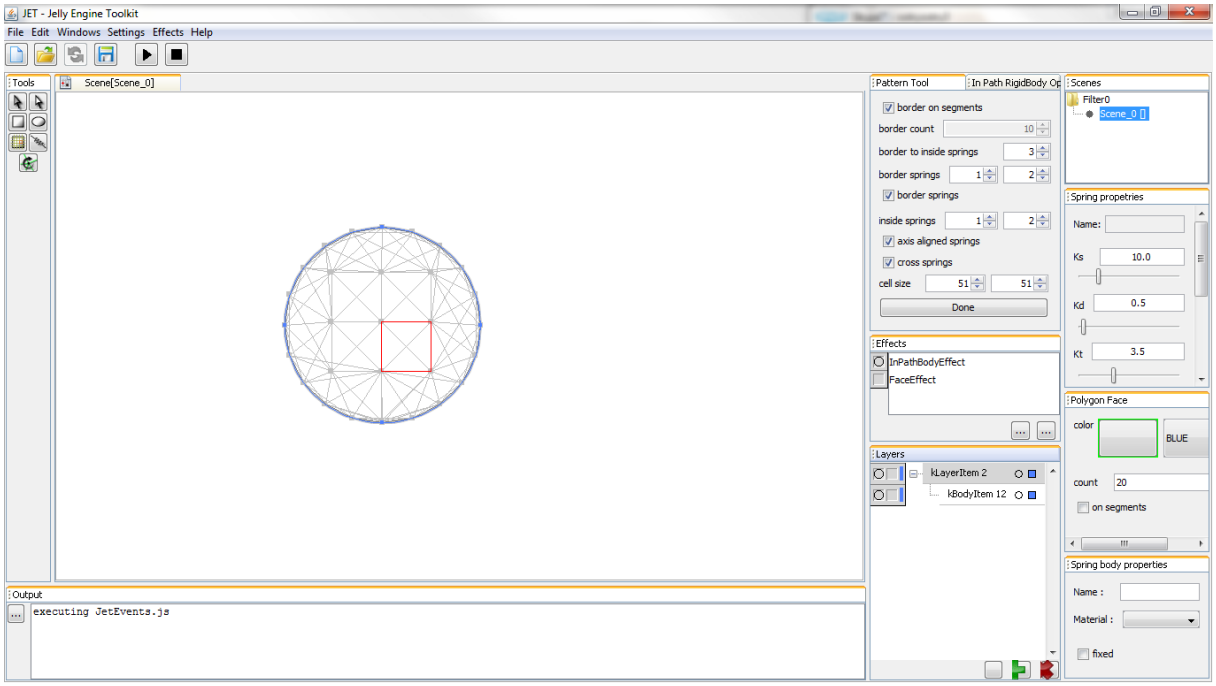
W celu zamiany tak utworzonego element typu ShapeItem na element typu BodyItem opisujący bryłę miękką należy wykorzystać narzędzie PatternTool. Umożliwi ono

zdefiniowanie wewnętrznej struktury nowo powstałego elementu SpringBody, przekonwertuje element okręgu z typu ShapeItem do typu BodyItem oraz nałoży na niego efekt InPathBodyEffect. Efekt ten dbać będzie o definiowanie struktury bryły miękkiej rozpiętej na krzywej elementu BodyItem zgodnie ze zdefiniowaną przez narzędzie PatterTool strukturą. Poniżej w znanej już konwencji graficznej przedstawiono kroki, które wykonają wszystkie powyższe czynności. Scrollem myszy zwiększono zoom sceny, tak aby móc dokładniej zdefiniować wewnętrzną strukturę piłeczki.



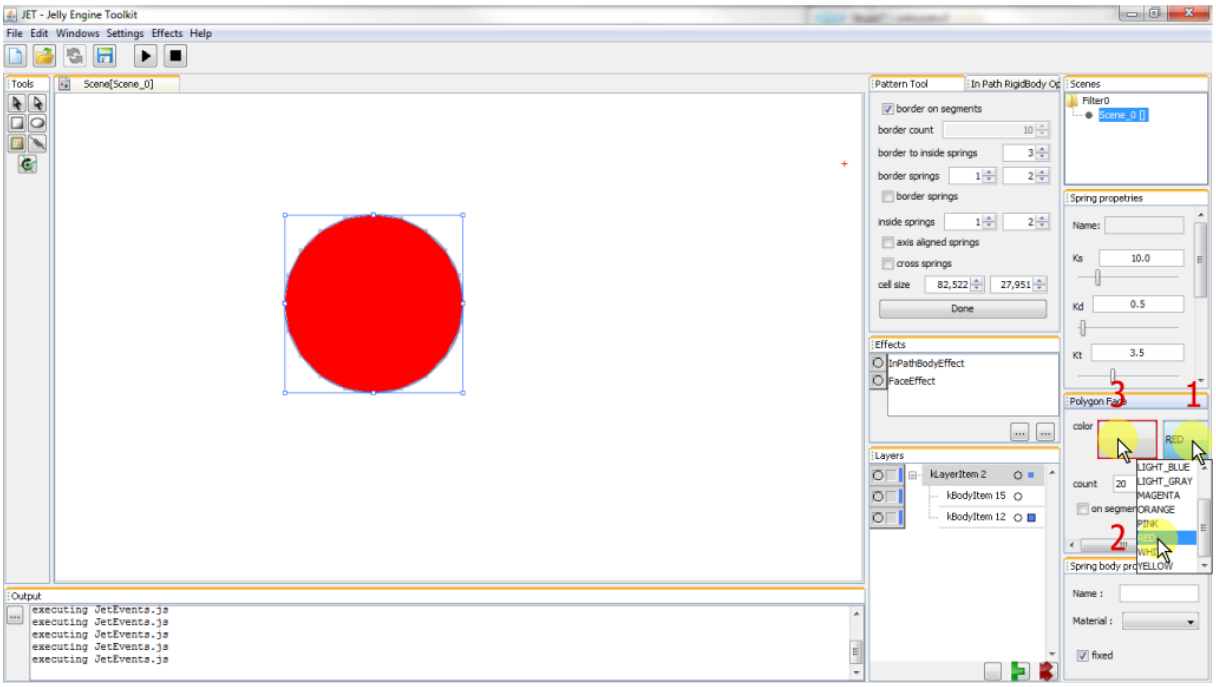
Rysunek 4.2.2.: Kroki służące zdefiniowaniu struktury bryły miękkiej bohatera gry – miękkiej piłeczki. Krok nr. 1 służy wyłączeniu rysowania efektu wypełnienia co pomaga w dokładnym wypozycjonowaniu struktury bryły miękkiej. Źródło: własne.

Po zdefiniowaniu struktury bryły miękkiej należy zedytować własności narzędzia PatternTool w panelu narzędziowym o tym samym tytule. Następnie przyciskiem ok, lub klawiszem enter należy zaakceptować zmiany wprowadzone narzędziem PatternTool. Efekt wykonania powyższych kroków przedstawiono rysunku 4.2.3. znajdującym się na następnej stronie.



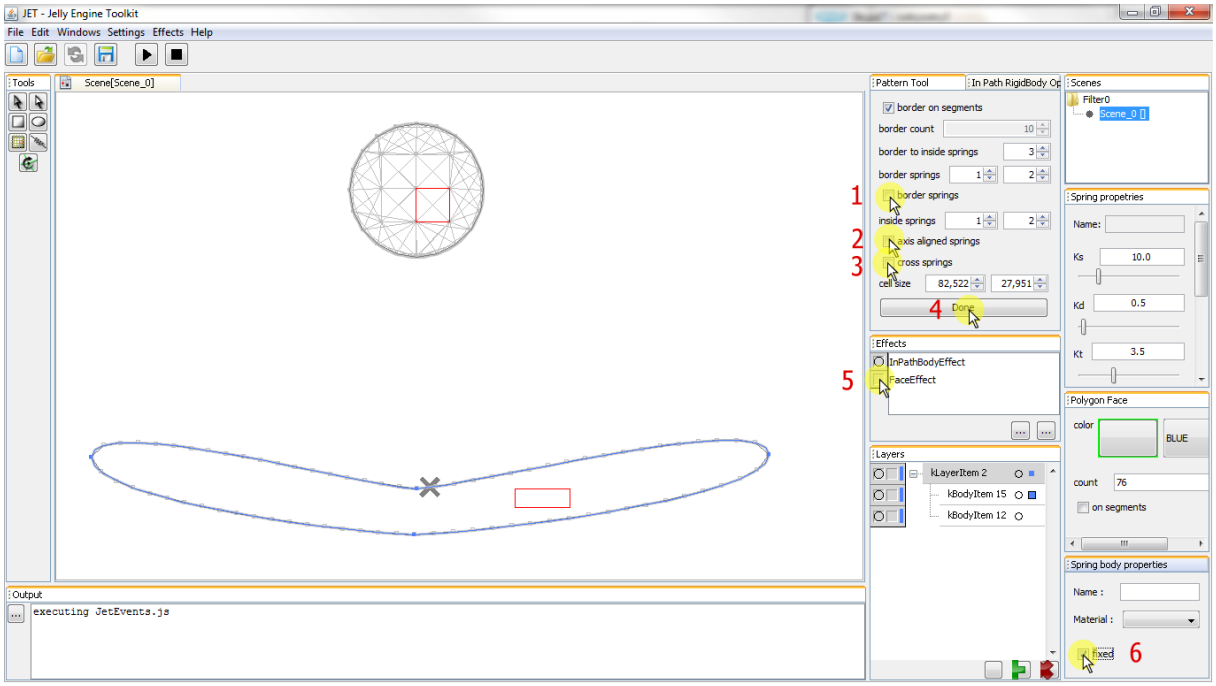
Rysunek 4.2.3.: Widok środowiska wraz ze sceną symulacji z elementem opisującym bryłę miękką. Rysowanie efektu FaceEffect elementu bryły miękkiej zostało wyłączone dzięki czemu widoczna jest wewnętrzna struktura ciała miękkiego rysowana przez efekt InPathBodyEffect .Źródło: własne.

Założenia gry mówiły czerwonej piłeczce. Elementy ShapeItem tworzone za pomocą narzędzi prostokąta i elipsy domyślnie mają nakładany efekt FaceEffect definiujący wypełnienie kolorem zielonym. W celu zmiany tego koloru na czerwony należy zaznaczyć element piłeczki i przy pomocy panelu narzędziowego „Polygon face" dokonać zmiany koloru wykorzystywane przez efekt FaceEffect. Kolor wybrać można spośród palety predefiniowanych kolorów za pomocą listy rozwijalnej w prawym górnym roku panelu. Spowoduje to zmianę podświetlenia przycisku „color". Po przyciśnięciu tego przycisku wszystkie zaznaczone elementy posiadające efekt FaceEffect zostanę zmienione tak, aby było ich efekty korzystały z nowo zdefiniowanego koloru. Czynności opisane powyżej zilustrowano na rysunku 4.2.4. znajdującym się na następnej stronie.



Rysunek 4.2.4.: Prezentuje kroki potrzebne do zmiany koloru piłeczki. Źródło: własne.

Posiadając aktywnie fizyczny element modelujący bohatera gry: piłeczkę należy przystąpić do tworzenia podłoża. Pozwoli to przetestować własności fizyczne modelu bryły miękkiej i dostosować je do wartości spodziewanych przez autora gry. W tym celu do sceny symulacji dodawany jest nowy element typu ShapeItem (za pomocą narzędzia ellipse). Następnie jego geometria zmieniana jest za pomocą narzędzia selekcji bezpośredniej do formy rządanej przez twórcę gry (w tym wypadku w kształt przypominający rogala) oraz w ten sam sposób co poprzednio (przez narzędzie PatternTool) nakładany jest na ten element efekt InPathBodyEffect. Za pomocą panelu narzędziowego PatternTool zmieniana jest struktura elementu modelującego podłoże: nie będzie on posiadał żadnych sprężyn, jedynie cząsteczki na brzegu elementu oraz łączące je ściany, które zbudują aktywny w trakcie kolizji brzeg ciała. Następnie ciało miękkie ustawiane jest jako statyczne (nie podlegające animacji) przy użyciu panelu narzędziowego „Spring body properties". W tym celu należy zaznaczyć ciało na scenie symulacji przy użyciu narzędzia selekcji i w panelu „Spring body properties" zaznaczyć checkbox o nazwie fixed. Można także wyłączyć rysowanie efektu odrysowującego wypełnienie elementu, robi się to w panelu Effects klikając ikonę okręgu obok nazwy efektu „FaceEffect". Wszystkie te czynności w ogólny sposób zobrazowano na rysunku 4.2.5. znajdującym się na następnej stronie.

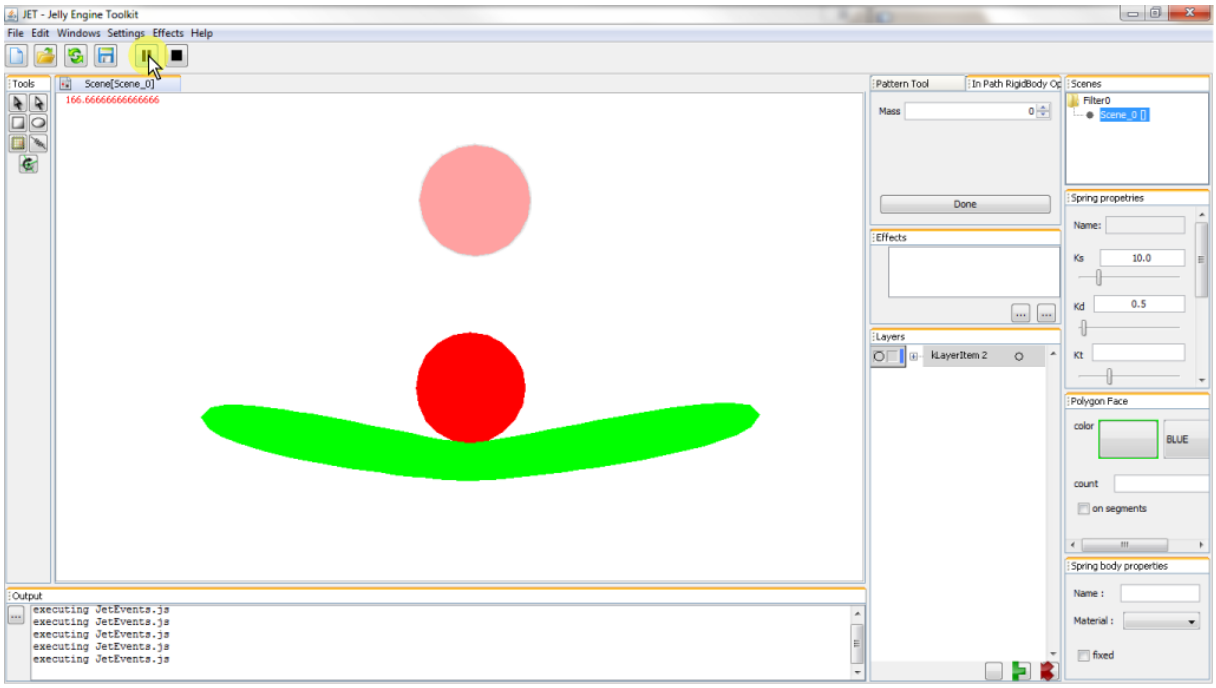


Rysunek 4.2.5.: Kroki wykonane w celu stworzenia statycznego element podłoża. Źródło: własne.

Kroki od 1 do 4 odnoszą się do wewnętrznej struktury bryły miękkiej budowanej przez efekt InPathBodyEffect. Krok 1 wyłącza generowanie sprężyn na brzegu ciała, krok 2 wyłącza generowanie sprężyn pionowych i poziomych wewnątrz ciała miękkiego, natomiast krok 3 wyłącza generowanie sprężyn ukośnych. Krok 4 to kliknięcie przycisku „Done", który powoduje zaaplikowanie efektu do nowego elementu typu BodyItem. Krok 6 powoduje ustawienie flagi fixed na obiekcie SpringBody modelującym bryłę miękką wygenerowaną przez efekt InPathBodyEffect. Spowoduje to wyłączenie ciała miękkiego z wykonywania procedur animacji, przez co silnik fizyki będzie jedynie obsługiwał jego kolizje z innymi fizycznie aktywnymi elementami sceny.

Tak stworzona scena jest już w pełni gotowa do uruchomienia animacji. Uruchamiając animację można zaobserwować własności fizyczne stworzonych elementów i dostosować je do własnych potrzeb. W tym momencie należy sprawdzić czy tak utworzona piłeczka odbija się i deformuje zgodnie z wolą twórcy gry. Jeżeli jej zachowanie w trakcie animacji odbiegać będzie od zamysłów twórcy, można wpłynąć nie nie edytując własności fizyczne ciała miękkiego modelującego piłkę.

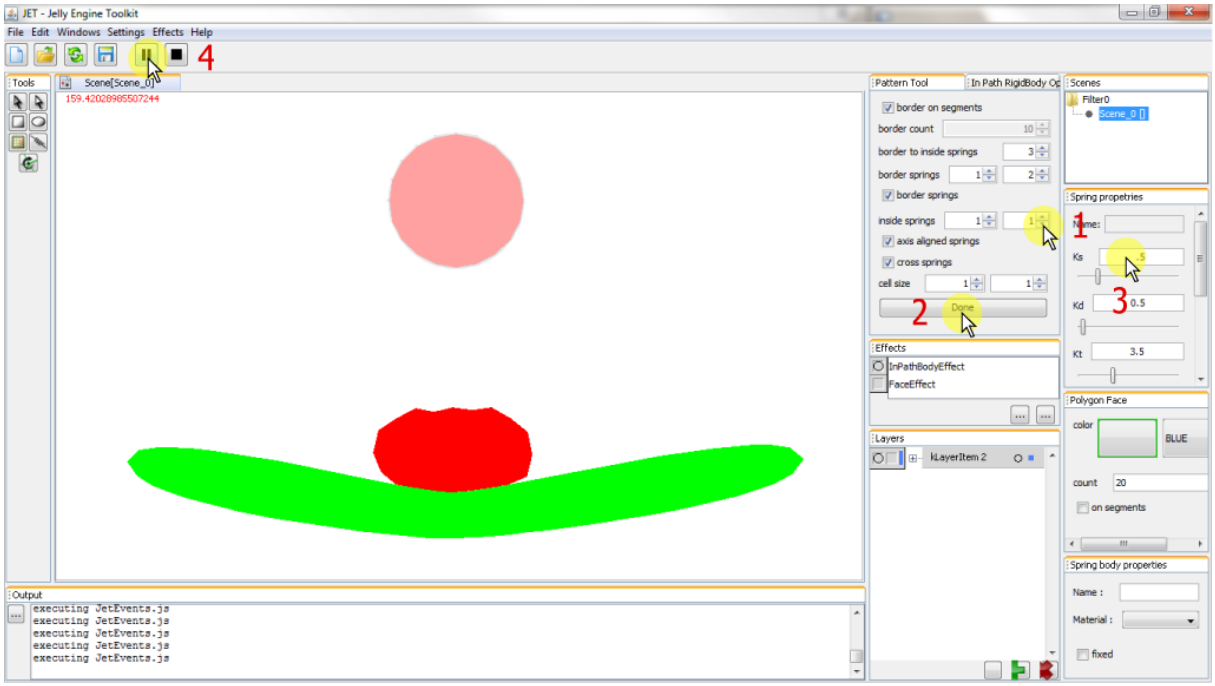
W celu uruchomienia symulacji sceny, należy kliknąć przycisk „play” (czarny trójkąt) znajdujące się u góry okna środowiska. Do zatrzymywania animacji służy znajdujący się obok przycisk „stop” (czarny kwadrat). Na poniższej grafice przedstawiono zrzut ekranu z animacji piłeczki, w której piłka pod wpływem grawitacji spada, odbija się po czym zastyga w bezruchu na zbudowanym uprzednio podłożu.



Rysunek 4.2.6.: Prezentuje scenę symulacji w trakcie animacji, piłeczka pod wpływem grawitacji spadła na element modelujący podłoże. Dla ułatwienia pokazano także pozycję początkową piłki.
Źródło: własne.

W trakcie obserwacji animacji autor przykładu zauważył, że piłeczka jest bardzo sztywna. W przyszłości, gdy poruszać się będzie po scenie jej sztywność będzie powodować słaby kontakt z podłożem, co wpłynie negatywnie na wrażenie kontroli ruchów piłeczki. Dlatego autor uznał, że należy zmienić sztywność modelu piłki. W tym celu w trybie edycji sceny należy za pomocą narzędzia selection tool zaznaczyć piłeczkę i zmienić jej wewnętrzną strukturę ograniczając liczbę sprężyń wewnątrz piłeczki. W tym celu należy z w panelu narzędziowym PatternTool zmienić zakres sprężyn „indise springs”, tak aby drugi spinner zawierał wartość równą 1. Oznaczać to będzie, że sprężyny wewnętrznej struktury piłeczki rozpinane będą co 1 cząsteczkę, a nie jak do tej pory co jedną i co dwie cząsteczki. Wpłynie to na zmniejszenie sztywności piłeczki. Następnie należy zmieniając sztywność wszystkich pozostałych sprężyn modelu piłeczki na wartość mniejszą niż domyślna równa 10. W tym celu w

panelu narzędziowym „Spring properties” należy zmienić stałą sprężystości sprężyn „ks” na wartość równą 1. Zmniejszy to sztywność piłeczki dziesięciokrotnie, względem sztywności domyślnej efektu InPathBodyEffect. Na poniższym zrzucie ekranu zaprezentowano kroki jakie twórca gry musi wykonać w celu wprowadzenia powyższych modyfikacji. Zrzut ekranu pokazuje stan piłeczki z nowymi własnościami fizycznymi w momencie zderzenia z podłożem.



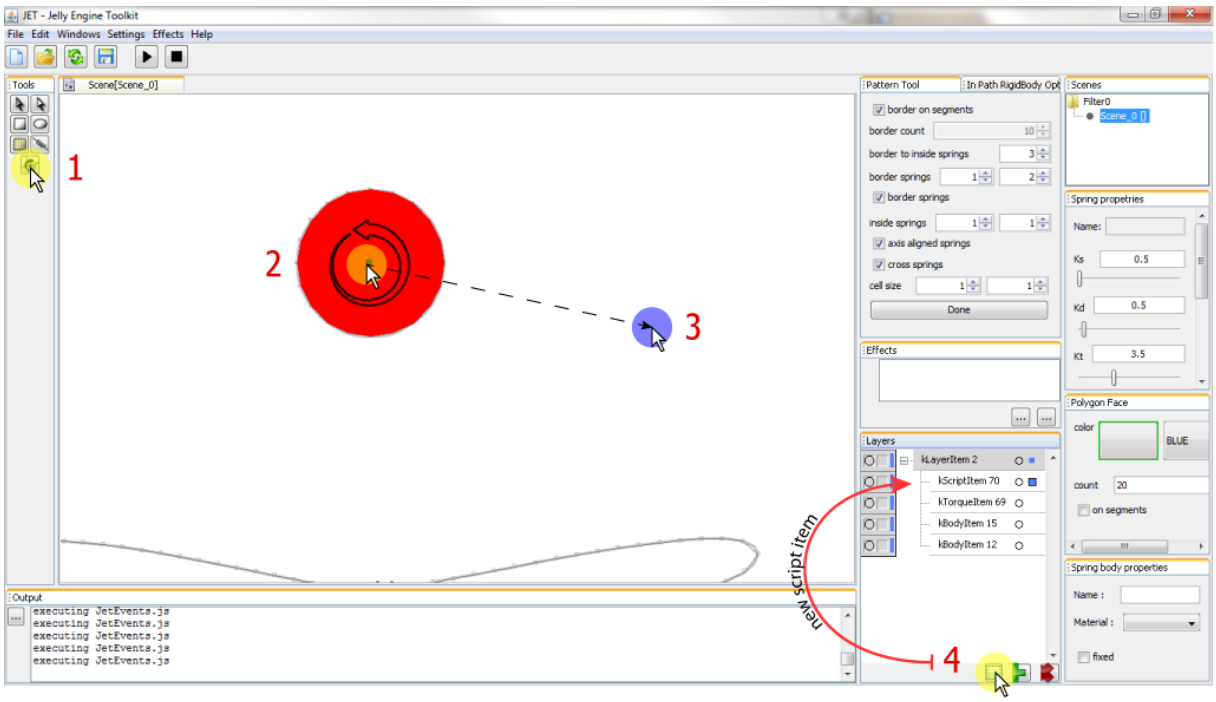
Rysunek 4.2.7.: Prezentuje przykładową scenę symulacji ze zmienionymi własnościami piłeczki. Wyraźnie widać odkształcenie bryły miękkiej modelującej piłeczkę w momencie kontaktu z podłożem. Źródło: własne.

Odkształcenie spowodowane zderzeniem pokazuje wpływ wprowadzonych w modelu piłeczki zmian. Dzięki nim w trakcie ruchu piłeczka będzie miała lepszy kontakt z podłożem, będzie się mniej odbijać co zapewni lepszą przyczepność i wpłynie na responsywność sterowania piłeczką przez użytkownika.

4.3. Uruchamianie piłeczki

W celu uruchomienia piłeczki, tak aby gracz mógł nią sterować należy zdefiniować moment obrotowy zamocowany do centralnej cząsteczki piłeczki. Następnie należy utworzyć element skyptowy, który zapewni zmianę wartości momentu obrotowego zależenie od tego, który przycisk na klawiaturze będzie wciśnięty przez gracza. Dla celów tego przykładu przyjęto, że gracz będzie sterował

ruchami piłeczki za pomocą klawiszy „A” oraz „D”. W momencie, gdy przyciśnięty będzie klawisz „A”, skrypt będzie ustawiał wartość momentu obrotowego na liczbę dodatnią, w momencie przyciśnięcia klawisza „D” wartość momentu obrotowego ustawiana będzie na liczbę ujemną. Dzięki temu piłeczka będzie obracała się w lewo bądź w prawo. Gdy żaden klawisz nie będzie przyciśnięty wartość momentu obrotowego będzie ustawiana na wartość równą 0, co jest jednoznaczne z wyłączeniem działania momentu obrotowego. Kroki niezbędne do stworzenia i umieszczenia momentu obrotowego oraz dodania pustego elementu skryptowego do drzewa sceny zaprezentowano na poniższym rysunku:

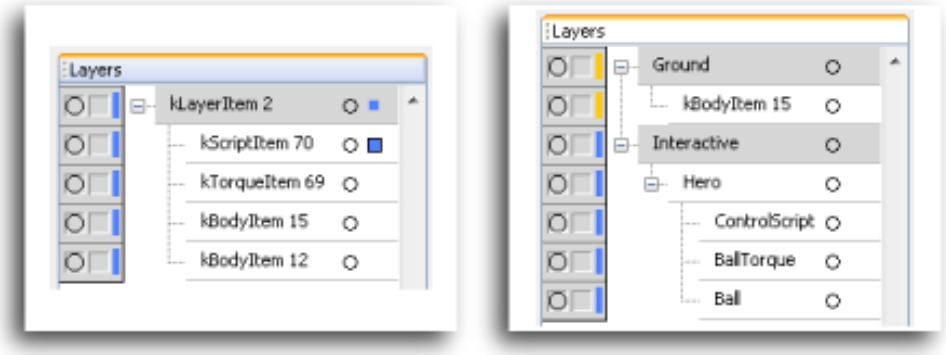


Rysunek 4.3.1.: Prezentacja kroków służących dodaniu elementu momentu obrotowego (kroki 1-3) oraz elementu skryptu (krok 4). Źródło: własne.

Krok 1 to wybranie narzędzia „Torque”, które posłuży do zdefiniowania elementu typu TorqueItem. Kroki 2 oraz 3 tworzą element momentu obrotowego. W tym momencie nie istotny jest kierunek momentu, gdyż jego wartość będzie dynamicznie zmieniana przez użytkownika klawiszami A i D. Krok 4 to dodanie elementy skryptowego typy ScriptItem do drzewa elementów sceny.

Następnym krokiem w pracy z tworzeniem interaktywnej piłeczki będzie pogrupowanie i nazwanie powstałych do tej pory elementów sceny, tak aby ułatwić twórcy gry dostęp

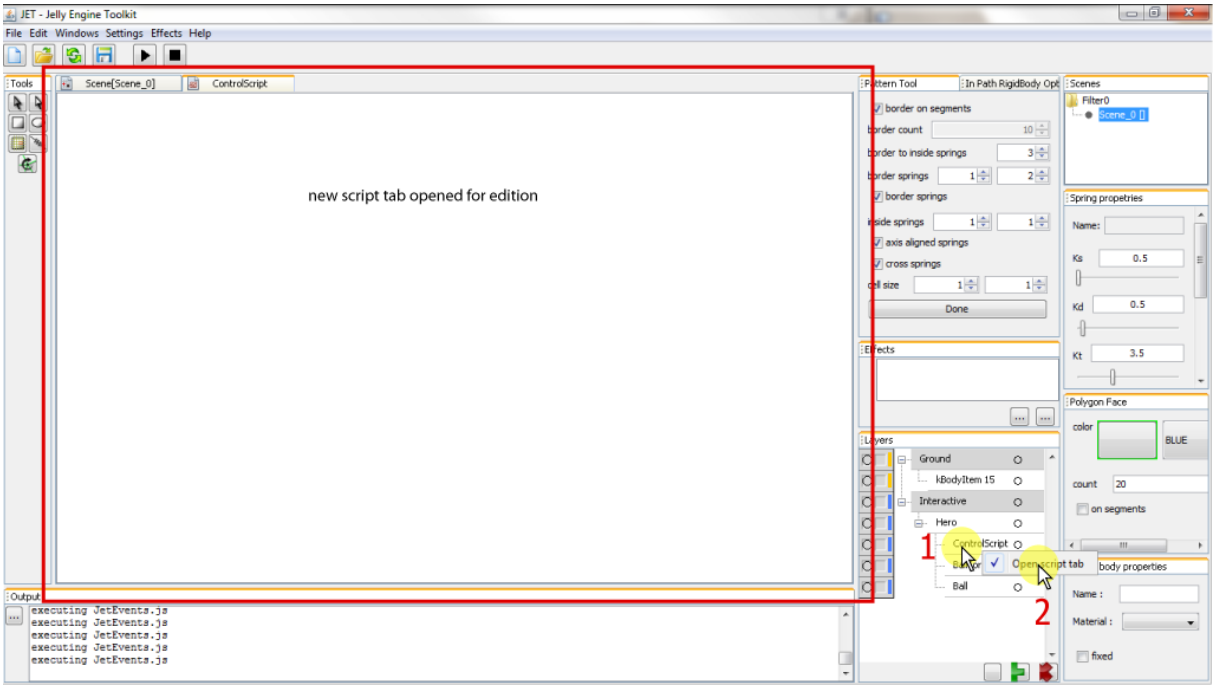
do poszczególnych elementów oraz logicznie podzielić zawartość sceny na warstwy: bohatera (element piłeczki, element momentu obrotowego oraz element skryptowy) i podłoża. Na poniższej grafice przedstawiono wygląd drzewa symulacji sceny przed i po opisanych powyżej zmianach.



Rysunek 4.3.2. : Prezentuje zastosowane zmiany do struktury drzewa elementów sceny oraz zmiany w nazwach poszczególnych elementów. Źródło: własne.

Nazwy elementów edytuje się w okienku dialogowym dostępnym po dwukrotnym kliknięciu nazwy elementu w panelu narzędziowym Layers. Zmiany położenia elementów wewnątrz drzewa elementów sceny dokonywane są za pomocą przeciągania elementów w położenie docelowe. Do dodawania i usuwania warstw wewnątrz sceny służą przyciski oznaczone jako „+” oraz „x” znajdujące się u dołu panelu narzędziowego Layers.

Z tak przygotowanym drzewem elementów sceny można przystąpić do edycji skryptu, który przypisze klawiszom A i D ich zaplanowane wcześniej funkcje uruchamiania momentu obrotowego, którego nazwę zmieniono w poprzednim kroku na „BallTorque”. W celu otwarcia skryptu do edycji należy za pomocą prawego przycisku myszy przywołać jego menu kontekstowe klikając jego nazwę wewnątrz panelu Layers. Następnie należy wybrać opcję „Open script tab”, co spowoduje dodanie nowego panelu edycji skryptu. Kroki te zilustrowano na rysunku 4.3.2., znajdującym się na następnej stronie, prezentującym stan środowiska tuż po otwarciu skryptu „ControlScript” do edycji.



Rysunek 4.3.2.: Prezentacja kroków służących otwarciu skryptu do edycji. Źródło: własne.

Poniżej znajduje się listing skryptu wraz z komentarzami, który uruchamia klawisze A i D, tak aby wprawiały piłeczkę w ruch obrotowy.

```
// dostajemy się do korzenia drzewa elementów sceny
var tg = ItemUtils.getTopGroup(script);

// dostajemy się do elementu momentu obrotowego
var ballTorque = ItemUtils.getItemByName("BallTorque", tg);

// dostajemy się do obiektów opisujących klawisze A i D
var keyA = Keyboard.getKey("A");
var keyD = Keyboard.getKey("D");

// ustalamy wartość momentu obrotowego
var tVal = 1000;

// zmienne do sterowania zoomem kamery
var vs = 1,
    dvs = .001;

// rejestracja funkcji obsługi zdarzeń wołanej przy każdym kroku
// symulacji silnika fizycznego (zdarzenie onStep)
JetSystem.addEventListener("onStep", function() {
      // jeżeli przyciśnięty jest klawisz A to ustaw wartość momentu
      //obrotowego tak aby obracał piłeczkę w lewo
      if (keyA.isPressed()) {
          ballTorque.torque.setValue(tVal);
          vs -= dvs;
      }
```

```
        // jeżeli przyciśnięty jest klawisz D to ustaw wartość momentu
        //obrotowego tak aby obracał piłeczkę w prawo
        else if (keyD.isPressed()) {
            ballTorque.torque.setValue(-tVal);
            vs -= dvs;
        }
        // jeżeli nie przyciśnięto żadnego klawisza wyłącz moment
        //obrotowy
        else {
            ballTorque.torque.setValue(0);
            vs += dvs/2;
        }
});
```

Listing 4.3.1.: Ciało skryptu uruchamiającego interakcję pomiędzy graczem, a modelem piłeczki. Klawisz A powoduje obrót piłeczki w lewo, klawisz D w prawo. Resztą symulacji, czyli odpowiednim przemieszczeniem piłeczki względem podłoża, zajmuje się silnik fizyki i mechanizmy w nim zawarte.

4.4. Kamera śledząca

Kolejnym etapem tworzenia przykładowej gry będzie stworzenie mechanizmu dbającego o usytuowanie bohatera-piłeczki po środku panelu graficznego prezentującego scenę w trakcie rozgrywki.

Funkcjonalność taką można uzyskać wykorzystując elementy skryptowe. Po dodaniu nowego skryptu o nazwie CameraScript należy zadbać o to, aby przy każdej zmianie położenia piłeczki środek ekranu śledził jej położenie. W tym celu należy wydobyć od silnika fizycznego prostokąt otaczający piłeczkę i znaleźć jego punkt centralny „center". Następnie należy dostać się do panelu graficznego prezentującego scenę gry w trakcie rozgrywki i wydobyć jego wymiary. Wymiary panelu graficznego posłużą do obliczenia wektora przesunięcia punktu zaczepienia kamery do połowy ekranu. Przesunięcie to musi mieć wartość połowy szerokości i wysokości ekranu, tak aby piłeczka znajdowała się dokładnie po środku panelu. Obie wartości nazwano kolejno „w" oraz „h", od angielskich nazw szerokości i wysokości: width i height.

Znając te trzy wartości należy dostać się do panelu graficznego prezentującego symulację sceny. Znajduje się on wewnątrz centrum graficznego sceny GraphicsCenter (w kodzie dostępne przez skrót GC) i dostępne jest jako pole publiczne obiektu sceny.

Ostatnią funkcją skryptu CameraScript będzie animowanie wartości zoom'a sceny w trakcie rozgrywki. Ponieważ piłeczka będzie poruszać się z różną prędkością po scenie,

należy umożliwić graczowi odpowiednio daleką widoczność w kierunku poruszania się piłeczki, tak aby mógł odpowiednio wcześnie reagować na zmiany kształtu podłoża. W tym celu powiększenie sceny (zoom) będzie maleć wraz ze wzrostem prędkości piłeczki, oraz rosnąć w miarę zwalniania bohatera. Implementacja tej funkcji jest o tyle prosta, że prędkość piłeczki rośnie w miarę przytrzymywania, któregoś z klawiszy A lub D. Dlatego wygodnie i prosto będzie uzależnić wartość zoom'a od czasu przyciśnięcia jednego z tych klawiszy. Kod za to odpowiedzialny jest już częścią poprzedniego skryptu ControlScript, są to linie zmieniające wartość zmiennej „vs" wewnątrz kodu obsługi klawiszy A i D.

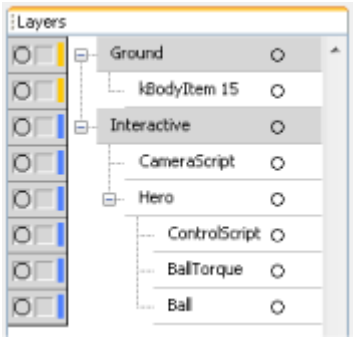
Poniżej znajduje się skomentowany kod elementu skryptowego CameraScript.

```
// dostajemy się do korzenia drzewa elementów sceny
var tg = ItemUtils.getTopGroup(script);

// dostań się do elementu piłeczki
var ball = ItemUtils.getItemByName("Ball", tg);
// dostań się do obiektu sceny
var scene = CC.getScene("Scene_0");

// Stałe określające maksymalną i minimalną wartość
// zoom'a sceny (view scale)
var maxS = .3,  // maksymalna wartość zoom'a
    minS = .8;  // minimalna wartość zoom'a

// wektor przechowujacy obliczone centrum bohatera
var center = new Vec2d();
// wektor pomocniczy
var tmp = new Vec2d();

// rejestracja funkcji oblsugi zdarzeń wołanej przy
// każdym kroku  symulacji silnika fizycznego (zdarzenie onStep)
JetSystem.addEventListener("onStep", function() {
    // znajdz centrum piłeczki jako centrum jej prostokąta
    // otaczającego
    center = ball.body.getBoundingArea().getCenter();

    // znajdz połowy wymiarów ekranu prezentującego symulację
    var w = scene.GC.swingSimulationGP.getWidth()*.5;
    var h = scene.GC.swingSimulationGP.getHeight()*.5;

    // ogranicz wartość zoom'a do limitów zdefiniowanych
    // przez stałe minS i maxS
    if (vs>minS) vs = minS;
    else if (vs<maxS) vs = maxS;

    // ustaw wartość zoom'a
    scene.GC.swingSimulationGP.setViewScale(vs);
```

```
    // oblicz przesunięcie kamery od początku układu współrzędnych
    // sceny silnika fizycznego do pozycji piłeczki na scenie
    // symulacji i przetransformuj tą wartość z koordynatów silnika
    // fizycznego do koordynatów panelu graficznego
    tmp.become(center);
    tmp.scale(vs);
    tmp.x -= w;
    tmp.y -= h;
    tmp.scale(-1);

    // ustaw wartość przesunięcia kamery panelu graficznego
    // rezentującego symulację sceny
    scene.GC.swingSimulationGP.trans.become(tmp);
});
```

Listing 4.4.1. : Kod skryptu odpowiedzialnego za przesunięcie kamery w miejsce znajdowania się bohatera-piłeczki.

Poniżej (rysunek 4.4.1.) przedstawiono widok drzewa elementów sceny po dodaniu i przepozycjonowaniu elementu skryptowego CameraScript.



Rysunek 4.4.1.: Wygląd drzewa elementów sceny prezentowanego w panelu Layers po dodaniu elementu CameraScript. Źródło: własne.

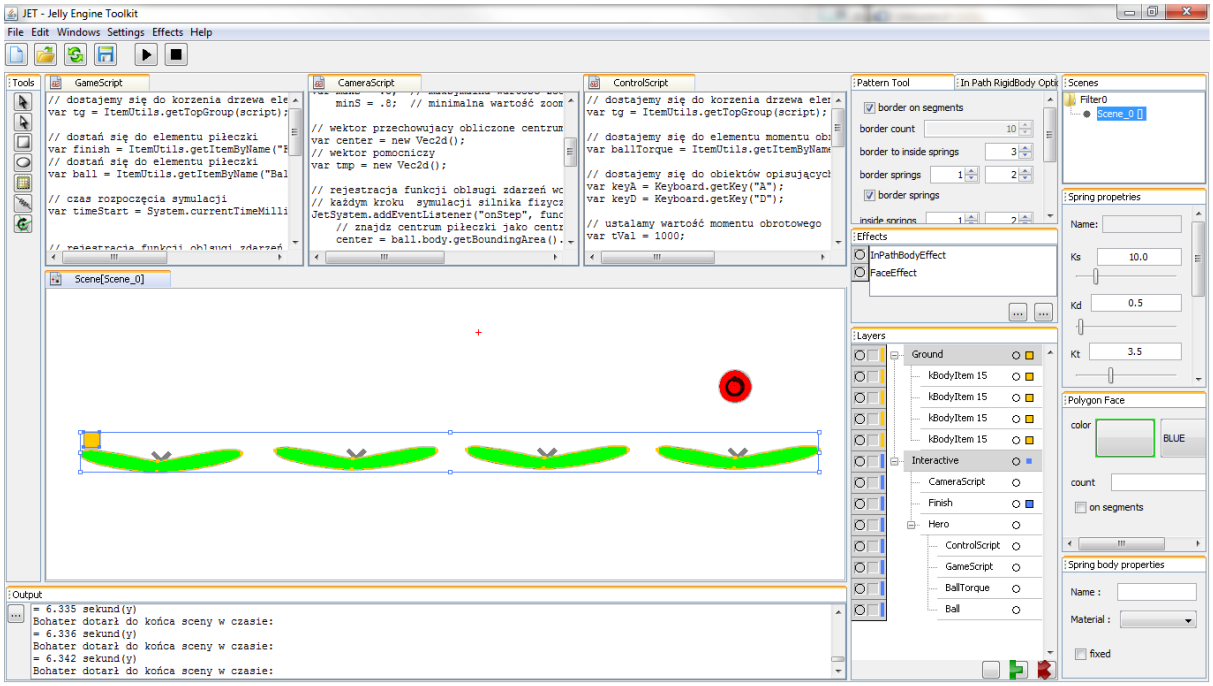
Skrypt CameraScript dodany został bezpośrednio do warstwy Interactive, a nie do grupy Hero, ponieważ pozycja kamery będzie mogła śledzić tylko pojedynczy element sceny. Grupę Hero można skopiować kilkakrotnie na scenie, wszystkie kopie będą reagować na wciskanie klawiszy A i D tak samo, jednak kamera śledzić będzie tylko jeden element o nazwie Ball znajdujące się w drzewie elementów sceny najbliżej elementu CameraScript.

4.5. Akcja kończąca symulację sceny

Gracz w celu zakończenia gry musi dotrzeć do końca trasy na scenie gry. W przypadku tej przykładowej sceny elementem końcowym będzie dodatkowe ciało miękkie. Umieszczone na końcu trasy zaplanowanej dla danej sceny, ciało to będzie

spoczywać na podłożu i czekać na przybycie bohatera. W momencie kontaktu bohatera z tym ciałem nastąpi zakończenie liczenia czasu spędzonego na scenie i tym samym zakończenie przykładowej gry.

Przykładowa scena symulacji została rozbudowana w celu symulacji rzeczywistej trasy, którą pokonać musi bohater gry. Dodatkowo metodami opisany w poprzednich rozdziałach dodano do sceny dodatkowy element stanowiący punkt docelowy dla wędrówki gracza po scenie: pomarańczowy prostokąt. Aktualny widok sceny oraz jej drzewa elementów przedstawiony został na poniższym zrzucie ekranu.



Rysunek 4.5.1.: Widok przykładowej kompletnej sceny symulacji z obecnymi wszystkimi elementami budującymi interaktywną grę. Źródło: własne.

```
// dostajemy się do korzenia drzewa elementów sceny
var tg = ItemUtils.getTopGroup(script);

// dostań się do elementu piłeczki
var finish = ItemUtils.getItemByName("Finish", tg);
// dostań się do elementu piłeczki
var ball = ItemUtils.getItemByName("Ball", tg);

// czas rozpoczęcia symulacji
var timeStart = System.currentTimeMillis();

// rejestracja funkcji oblsugi zdarzeń wołanej przy
// każdym kroku  symulacji silnika fizycznego (zdarzenie onStep)
JetSystem.addEventListener("onStep", function() {

    // czas bohatera spędzony na scenie gry
    var time = System.currentTimeMillis() - timeStart;

    // test czy nastąpiła kolizja pomiędzy elementami ball oraz finish
    if (ctx.colTab[ball.body.cti][finish.body.cti]) {
        // wyswietl komunikat o zakonczeniu sceny
        CC.GUIC.println\nToOutput("Bohater zakończył scenę w czasie: ");
        CC.GUIC.println\nToOutput("= "+ (time/1000.) +" sekund(y)");

        // Poniżej wykomentowany fragment kodu w prawdziwej grze
        // przenosiłby gracza do sceny prezentującej menu,
        // lub wybór kolejnej planszy gry...
        /*JetSystem.playScene("Scene_name");*/
    }
});
```

Listing 4.5.1.: Kod implementujący element GameScript odpowiedzialny za liczenie czasu spędzonego na scenie przez gracza oraz zakończenie rozgrywki przez wypisanie komunikatu o dotarciu do końca sceny.

5. Podsumowanie i wnioski

Opracowane środowisko do budowy gier 2D pozwala na szybkie prototypowanie prostych gier. Wbudowany edytor geometryczny ułatwia tworzenie skomplikowanych kształtów budujących plansze rozgrywki. Wraz z narzędziami do definiowania własności fizycznych pozwalają w krótkim czasie na budowę ciekawych i skomplikowanych lokacji, które stają się światem dla interaktywnych elementów świata gry. Implementacja środowiska wykorzystuje polimorfizm i obiektowe techniki tworzenia kodu oferowane przez język Java. Pozwala to na łatwe rozszerzanie możliwości środowiska poprzez dodawanie nowych funkcji w postaci niezależnych rozszerzeń, których dołączanie do aktualnej wersji kodu wymaga poznania jedynie części z głównych modułów budujących środowisko.

Opracowanie silnika fizyki 2D obsługującego bryły miękkie i sztywne okazało się być zadaniem nie prostym. Autor musiał niejednokrotnie posiłkować się kodem zapożyczonym z przykładów szeroko dostępnych w internecie, dla przykładu bardzo pomocne okazały się pozycje [32] [38]. Jednak wiedza zdobyta w trakcie opracowywania algorytmów optymalnych obliczeniowo i pracujących z danymi geometrycznymi przyniosła wiele korzyści, które autor zdążył wkorzystać w pracy zawodowej. Ponadto stworzony silnik i rezultaty dzięki niemu uzyskiwane uświadomiły autorowi ogrom skomplikowania opisywanych w części teoretycznej tej pracy silników fizycznych 3D.

Praca nad biblioteką MyGUI do zarządzania dokowalnymi panelami przyniosła pogłębienie wiedzy związanej z funkcjonowaniem biblioteki Swing, szeroko wykorzystywanej przy tworzeniu zaawansowanych międzyplatformowych interfejsów użytkownika. Przyniosła także głębokie zrozumienie działania wzorca projektowego MVC (Model View Controler) dokładnie opisanego w pracy [39], który dzięki swojej prostocie i klarownemu opisowi struktury i mechanizmów GUI znajduje szerokie wykorzystanie w profesjonalnych aplikacjach.

Projektowanie i implementowanie mechanizmów budujących samo środowisko okazało się najciekawszym elementem całej pracy. Zaprojektowanie systemu obsługi zdarzeń wewnętrznych środowiska, pozwalających na komunikację między poszczególnymi

modułami, wymagało implementacji mechanizmów bezpiecznych wielowątkowo i pozwoliło autorowi lepiej poznać narzędzia do projektowania systemów wielomodułowych (np. narzędzie Enterprise Architect do projektowania w języku UML).

Projekt hierarchicznej struktury scen gry, przechowujących wiele elementów różnych typów, do pracy z którymi wystarczy nieduży zbiór narzędzi czyni pracę ze środowiskiem przyjemną i łatwą do opanowania. Stanowił on pewne wyzwanie dla autora, który uważa, że dzięki wykorzystaniu wcześniej wspomnianej zasady KISS [34] udało się je zrealizować zadowalająco.

Dodatkowym utrudnieniem przy budowie tak złożonego systemu była praca z repozytorium liczącym ponad 30 000 linii kodu w trzech językach (Java, JavaScript oraz XML), na które składała się rdzenna aplikacja oraz trzy niezależne biblioteki. Bardzo pomocne w opanowaniu całego projektu okazało się środowisko programistyczne NetBeans w wersji 7.2, które zostało wykorzystane do pracy z kodem aplikcaj.

Projekt i dalsza rozbudowa środowiska nie zostaje bynajmniej zawieszona wraz z zakończeniem tej pracy magisterskiej. Jest jescze wiele rzeczy, które autor chciałby widzieć w końcowej wersji aplikacji oraz miejsca w już obecnym kodzie, które należy usprawnić.

Dotyczy to między innymi biblioteki geometrii MyGeom. Aktualnie wszystkie krzywe bezeir'a konwertowane są do przybliżających je wieolkątów, które to są rzeczywiście aktywnymi elementami geometrycznymi wewnątrz środowiska. Powoduje to duży narzut czasowy podczas pracy algorytmów geometrycznych i powinno być zastąpione podejściem czysto algebraicznym (bez przybliżeń wielokątami).

Innym problemem jest szybkość i możliwości biblioteki Swing. Nie nadaje się ona do pracy z dużą ilością elementów ponieważ zbyt wolno rysuje prymitywy graficzne, oraz brak jej wsparcia dla renderowania teksturowanych wielokątów. Jedynym rozsądnym kierunkiem w tym wypadku jest implementacja paneli graficznych scen z

wykorzystaniem technologii OpenGL. Projekt modułu graficznego odpowiedzialengo za renderowanie scen w trybie edycji i symulacji jest przygotowany do pracy z różnymi bibliotekami graficznymi. Dzięki temu implementacja paneli graficznych korzystających z akceleracji sprzętowej będzie bardzo prosta.

Kolejnym krokiem w rozbudowie środowiska będzie stworzenie aplikacji klienckich na różne platformy, tak aby gry tworzone w środowisku mogły być na nich uruchamiane. Planowane platformy to: Android, iOS oraz aplikacje webowe wykorzystujące technologię HTML/SVG + JavaScript. Aplikacje takie ładować będą projekty gry i odtwarzać je w ten sam sposób w jaki robi to moduł symulacji scen środowiska. Dzięki temu raz stworzona w środowisku gra będzie mogła być uruchamiana na wielu platformach bez potrzeby przystosowywania jej do każdej z nich oddzielnie.

Bibliografia

[1] S. Yanchun i S. Xingyi, „Research and improvement of collision detection based on oriented bounding box in physics engine," w *Communication Software and Networks (ICCSN)*, 2011.

[2] J. Rojek, „Modelowanie i symulacja komputerowa złożonych zagadnień mechaniki nieliniowej metodami elementów skończonych i dyskretnych," *Prace Instytutu Podstawowych Problemów Techniki PAN,* pp. 1-331, 2007.

[3] Shenton i Z. D.Cendes, „Three-Dimensional finite element mesh generation using delaunay tesselation," pp. 2535- 2538, 11 1985.

[4] O. James i H. Jessica, „Graphical modeling and animation of brittle fracture," w *SIGGRAPH '99 Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1999.

[5] K. C. Han i S. K. Hyun, „Deformation and Simulation of 3D Contents for the Digilog Book," w *International Symposium on Ubiquitous Virtual Reality*, Gwangju, 2010.

[6] D. Zhi, L. Haozhe, L. Shiqiu i Z. Yunxin, „GKJ and clustering based algorithm for collision detection on deformable body," w *Multimedia Technology (ICMT)*, 2011.

[7] D. Zhi, „Physically-based real-time simulation algorithm of vehicle shell collision and deformation," w *Computer Science and Automation Engineering (CSAE)*, 2011.

[8] M. Smoluchowski, „Biblioteka Wirtualna Nauki," 1914. [Online]. Available: http://matwbn.icm.edu.pl/ksiazki/pmf/pmf25/pmf2517.pdf. [Data uzyskania dostępu: 03 09 2012].

[9] C. Chae, „Introduction of Physics Simulation in Augmented Reality," w *Ubiquitous Virtual Reality (ISUVR)*, 2008.

[10] D. Beaney, „Forked! A demonstration of physics realism in augmented reality," w *Mixed and Augmented Reality (ISMAR)*, 2009.

[11] S. Nourian, „Role of extensible physics engine in surgery simulations," w *Haptic Audio Visual Environments and their Applications*, 2005.

[12] A. Tsampikakis, „Cheat Detection Processing A GPU versus CPU Comparison,"

w *Annual Workshop on Network and Systems Support for Games (NetGames '10)*, 2010.

[13] „Nvidia Developer Zone Apex," [Online]. Available: http://developer.nvidia.com/content/apex. [Data uzyskania dostępu: 05 09 2012].

[14] H. Niu, Y. Gao i Z. Hou, „Application research of PhysX engine in virtual environment," w *Audio Language and Image Processing (ICALIP)*, 2010.

[15] H. Bing, W. Yangzihao i Z. Jia, „An improved method of continuous collision detection using ellipsoids," w *Computer-Aided Industrial Design & Conceptual Design*, 2009.

[16] „Game Developers Conference," [Online]. Available: http://www.gdconf.com/. [Data uzyskania dostępu: 03 09 2012].

[17] „zlib license," [Online]. Available: http://www.gzip.org/zlib/zlib_license.html. [Data uzyskania dostępu: 03 09 2012].

[18] C. Richard, i. P. Jong-Sh i S. Richard, The linear complementarity problem. Computer science and scientific computing, Academic Press, 1992.

[19] R. Cottle, J.-S. Pang i V. Venkateswaran, Sufficient matrices and the linear complementarity problem. Linear Algebra and its Applications, 1989.

[20] Csizmadia, „New criss-cross type algorithms for linear complementarity problems with sufficient matrices," pp. 247-266, 2006.

[21] J. Lloyd, „Fast Implementation of Lemke's Algorithm for Rigid Body Contact Simulation," w *Robotics and Automation (ICRA)*, 2005.

[22] D. Banxiang, Z. Xiaoping i W. Jiaoyu, „Inexact Parallel Relaxed Multisplitting Algorithm for Linear Complementarity Problem," w *International Conference on Computational Intelligence and Natural Computing*, 2007.

[23] X. Wang, „Interior point method for solving linear complementarity problems with P*-matrix," w *International Conference on Computer, Mechatronics, Control and Electronic Engineering (CMCE)*, 2010.

[24] W. Guangbin, W. Hao i T. Fuping, „Synchronous Multi-splitting and Schwarz Methods for Solving Linear Complementarity Problems," w *ISCSCT*, 2008.

[25] A. Sheldon i M. Javier, „HaptiCast: A Physically-Based 3D Game with Haptic Feedback Abstract".

[26] M. Gianluca, C. Angelo i N. Stefano, „Evolution of Prehension Ability in an Anthropomorphic Neurorobotic Arm,” *Front Neurorobotics,* 2007.

[27] Zogrim, „PhysxInfo.com,” 7 12 2009. [Online]. Available: http://physxinfo.com/articles/?page_id=154. [Data uzyskania dostępu: 02 09 2012].

[28] „Metacritic.com,” [Online]. Available: http://www.metacritic.com. [Data uzyskania dostępu: 10 08 2012].

[29] D. Bourg, Fizyka dla programistow gier, Helion, 2003.

[30] J. Matulewski, Grafika, fizyka, metody numeryczne: symulacje fizyczne z wizualizacją 3D, Wydawnictwo Naukowe PWN, 2010.

[31] M. Matyka, Symulacje komputerowe w fizyce, Helion, 2002.

[32] C. Hecker, „Chris Hecker's Homepage,” 20 02 2011. [Online]. Available: http://chrishecker.com/Rigid_body_dynamics. [Data uzyskania dostępu: 15 07 2012].

[33] J. W. Cooper, Java. Wzorce projektowe, Helion, 2001/12.

[34] fhanik. [Online]. Available: http://people.apache.org/~fhanik/kiss.html. [Data uzyskania dostępu: 29 07 2012].

[35] B. Eckel, Thinking in C++, Helion, 2002/09.

[36] P. Kiciak, Podstawy modelowania krzywych i powierzchni. Zastosowania w grafice komputerowej, WNT, 2000.

[37] W. Consortium, „W3C,” [Online]. Available: http://www.w3.org/DOM/DOMTR. [Data uzyskania dostępu: 12 08 2012].

[38] Walaber, „Walaber blog,” [Online]. Available: http://www.walaber.com/. [Data uzyskania dostępu: 3 7 2012].

[39] A. Leff i J. Rayfield, „Web-application development using the Model/View/Controller design pattern,” w *Enterprise Distributed Object Computing Conference*, 2001.

[40] C. Horstmann i G. Cornel, Java 2: Podstawy, Helion, 2003.

[41] C. Horstmann i G. Cornel, Java 2: Techniki zaawansowane, Helion, 2003.