Generator

DOKUMENTACJA ANALITYCZNA

Wersja: 0.6
Data modyfikacji: 25 listopada 2007
Autorzy: A.Bąk and M.Kulbacki

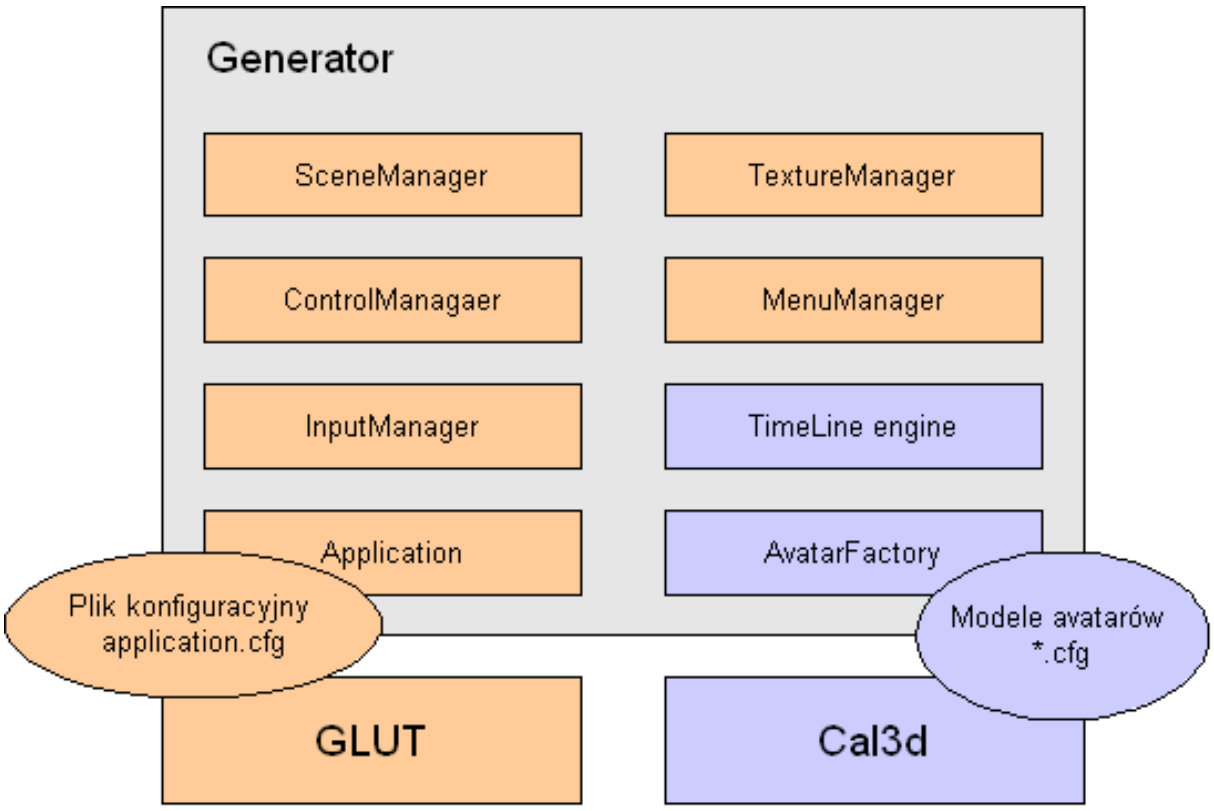
Spis treści

1 Model aplikacji Generator'a

Aplikacja Generator'a bazuje na dwóch głównych bibliotekach:
- GLUT – używana do zarządzania aplikacją okienkową oraz podstawowych operacji graficznych
- Cal3d – używana do reprezentacji modeli avatarów oraz zarządzanie animacjami dla nich

Do szybkiego ustawiania parametrów aplikacji oraz jej składowych modułów służy plik */data/application.cfg*. Modele avatarów oraz animacje dla nich opisane są w standardowym formacie modeli Cal3d (pliki xsf/xaf/xmf/xrf lub csf/caf/cmf/crf opisane skryptami **.cfg)*.

Ogólna ilustracja struktury aplikacji Generator'a została przedstawiona na rysunku 1.



Rysunek 1: Struktura aplikacji Generator'a

1.1 Konfiguracja aplikacji

Konfiguracja aplikacji znajduje się w pliku tekstowym */data/application.cfg*. Format zapisu jest bardzo prosty i wyróżnia dwie podstawowe instrukcje:

1. **komentarz** - umieszczony jest w linii zaczynającej się znakiem#.
2. **definicja parametru** - linia składająca się z trzech jednostek leksykalnych:
   (a) **identyfikatora** parametru (l-wartości)
   (b) operatora przypisania (=)
   (c) **wartości** lub **zbioru wartości** (r-wartości)

Definicje parametrów pogrupowane są w sekcjach. Każda sekcja rozpoczyna się nagłówkiem w formie komentarza. Definicja parametru może przyjmować dwie formy:

1. ***identyfikator parametru = wartość***
2. ***identyfikator parametru = (wartość1, wartość2, . . . , wartość-n)***

1.1.1 Tworzenie wybranych parametrów

Tworzenie menu Menu jest strukturą drzewiastą omówioną dokładnie w rozdziale dotyczącym typu *ft:MenuManager*. Menu buduje się w dwóch krokach, które można cyklicznie powtarać:

1. zdefiniowanie menu głównego z listą parametrów : *identyfikator_ menu = (lista identyfikatorów_ parametrów_ menu oddzielonych przecinkami)*
2. zdefiniowanie każdego prametru menu głównego w postaci: *identyfikator_ parametru menu = (etykieta informacyjna, nazwa tekstury)*

Tworzenie definicji konfiguracji kamer Definicja konfiguracji kamer jest dwuetapowa. W pierwszym kroku tworzy się listę definicji w formacie: *identyfikator = (lista identyfikatorów konfiguracji kamer oddzielona przecinkami)*. W drugim kroku dla każdego identyfikatora konfiguracji kamery przypisuje się następujące parametry:

1. *Hot_key* - klawisz, który uruchamia daną konfigurację kamery. Dostępne wartości: *ft_F1, ft_F2, ft_F3 ,ft_F4, ft_F5 ,ft_F6, ft_F7, ft_F8, ft_F9, ft_F10, ft_F11, ft_F12, ft_KEY_LEFT, ft_KEY_UP, ft_KEY_RIGHT, ft_KEY_DOWN, ft_KEY_PAGE_UP, ft_KEY_PAGE_DOWN, ft_KEY_HOME, ft_KEY_END, ft_KEY_INSERT*.
2. *Typ kamery* - zdefiniowany w systemie typ kamery. Dostępne wartości: *ft_ActiveAvatarCamera, ft_MainCamera*.
3. *Tryb pracy kamery* - można go wybrać z listy czterech dostępnych trybów pracy. Dostępne wartości: *ft_StaticCamera, ft_ThirdPersonCamera, ft_FlyCamera, ft_OrbitCamera*.

4. *Lokalizacja* - orientacja kamery względem punktu, na który kamera "patrzy". Dostępne wartości: *ft_FrontLeft, ft_FrontCenter, ft_FrontRight, ft_Left, ft_Center, ft_Right, ft_BackLeft, ft_BackCenter, ft_BackRight, ft_TopFrontLeft, ft_TopFrontCenter, ft_TopFrontRight, ft_TopLeft, ft_TopCenter, ft_TopRight, ft_TopBackLeft, ft_TopBackCenter, ft_TopBackRight, ft_BottomFrontLeft, ft_BottomFrontCenter, ft_BottomFrontRight, ft_BottomLeft, ft_BottomCenter, ft_BottomRight, ft_BottomBackLeft, ft_BottomBackCenter, ft_BottomBackRight, ft_AutoLocation.*

2 Moduł UpdateManager

Moduł UpdateManager jest jednym z bazowych modułów Generatora. Do jego głównych zadań należy odświeżanie obiektów w każdej klatce symulacji (frame) oraz rozsyłanie wiadomości do zainteresowanych obiektów. Główną klasą modułu jest *UpdateManager*, która posiada tylko jedną instancję (Singleton). Aby obiekty mogły być odświeżanie przez *UpdateManagera* lub odbierać wiadomości muszą być pochodnymi klasy *UpdateObject* oraz być zarejestrowanie w UpdateManager'ze.

2.1 Odświeżanie obiektów

Aby dany obiekt mógł reagować na odświeżenie w każdej klatce animacji powinien pokryć metodę *OnUpdate (float elapsedTime)* z klasy bazowej UpdateObject. Parametr *elapsedTime* określa ile czasu minęło od poprzedniego odświeżenia. UpdateManager będzie wywoływał metodę *OnUpdate* na wszystkich zarejestrowanych obiektach zgodnie z parametrami określonymi dla aktualnej symulacji (klasa *ft::Simulation*). Klasa *ft::Simulation* odczytuje czas przy pomocy metody *getTick()*. W zależności od potrzeby można pobierać czas z dokładnością do mili albo mikrosekund.

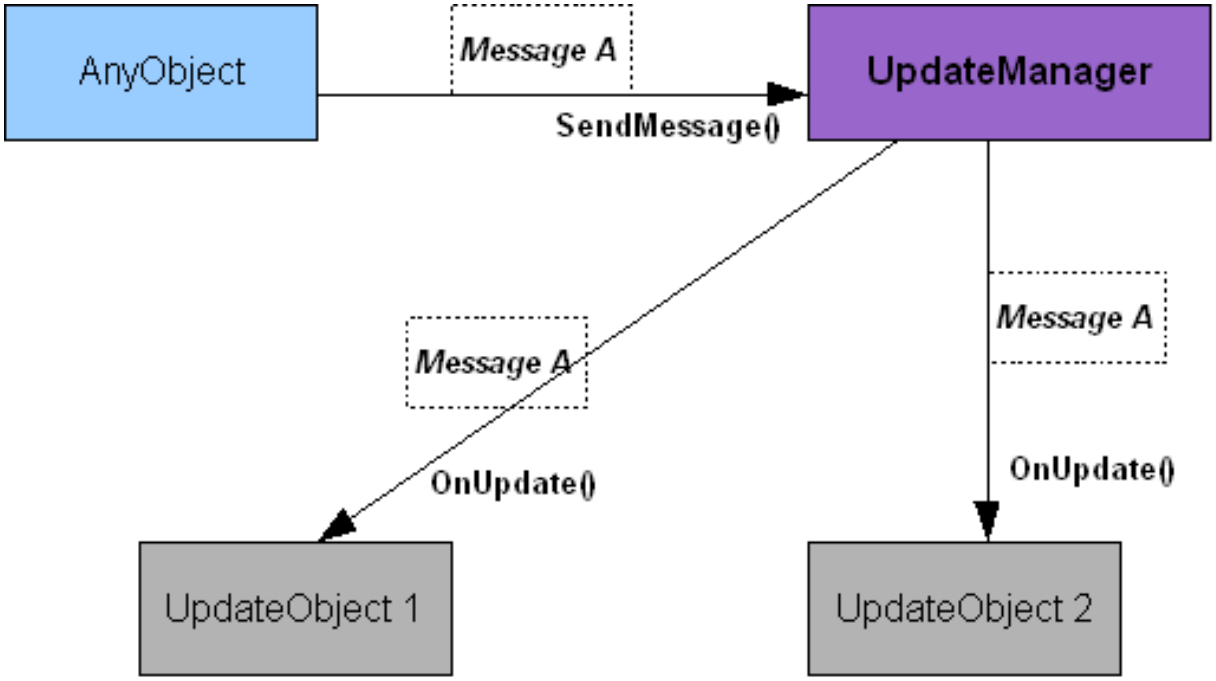
2.2 Rozsyłanie wiadomości

UpdateManager rozsyła wiadomości do zarejestrowanych obiektów za pomocą obiektów klasy *ft::Message*. Wywołuje w tym celu na obiektach funkcje *OnMessage(Message* msg)*, która jest zdefiniowana w klasie *UpdateObject*. Aby obiekt mógł zareagować na wiadomość powinien on pokryć metodę *OnMessage* i zaimplementować w niej rozpoznanie typu wiadomości oraz odpowiednie akcje.

Każdy obiekt w systemie może wysłać wiadomość przez UpdateManager'a używając metody *SendMessage(Message* msg, bool deleteAfterSent)*. Jako parametr *msg* należy podstawić właściwy obiekt typu *Message*, natomiast *deleteAfterSend* określa czy UpdateManager ma zwolnić pamięć dla obiektu *msg* po rozesłaniu go do zarejestrowanych obiektów. Ilustracja przepływu sterowania podczas rozsyłania wiadomości znajduje się na rysunku 2.

3 Avatar'y

Podstawową klasą reprezentującą animowaną postać jest Avatar. Można go dodawać do sceny oraz wykonywać na nim animacje. Obiekty typu Avatar bazują na modelach postaci



Rysunek 2: Przesyłanie wiadomości między obiektami implementującymi interfejs wiadomości UpdateManagera

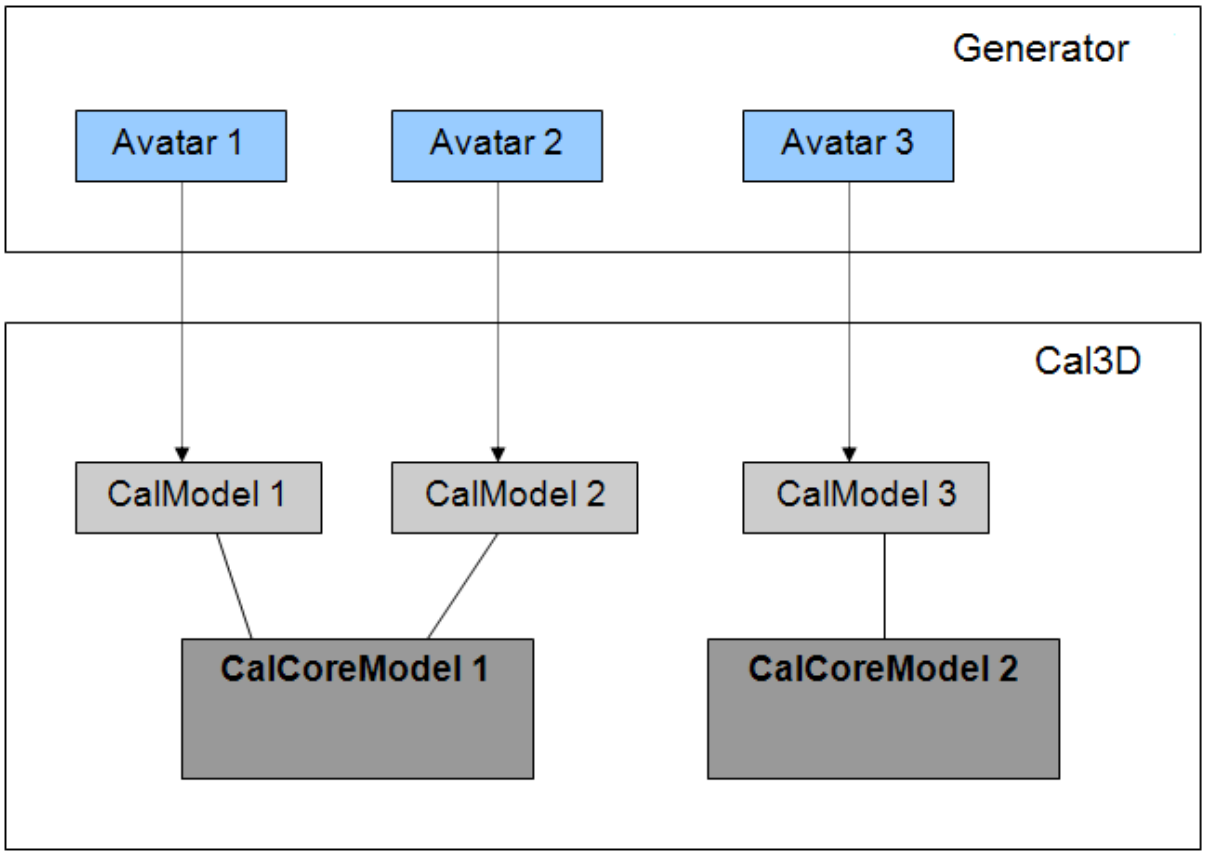
z API Cal3d i można je traktować jako odpowiedniki takich modeli z rozszerzona funkcjonalnością na potrzeby Generatora.

3.1 Powiązania Avatar'ów z modelami w API Cal3D

Podstawowym bytem reprezentującym animowaną postać w API Cal3Dd jest obiekt CalModel. Obiekt CalModel może być utworzony na podstawie odpowiedniego typu. Definicją takiego typu w Cal3d jest obiekt CalCoreModel. Różnica pomiędzy CalCoreModelem i CalModelem jest taka, ze CalCoreModel zawiera definicje mesh'y, animacji, materiałów oraz szkieletu postaci. CalModel jest natomiast specyficzna instancją CalCoreModelu i można go ustawiać na scenie oraz wykonywać na nim animacje. Dla każdego takiego typu może istnieć dowolna ilość obiektów klasy CalModel. Dla każdego utworzonego obiektu klasy Avatar utworzony zostaje osobny obiekt klasy CalModel w enginie Cal3d. Ponadto obiekt Avatar zawiera referencję do odpowiedniego obiektu CalCoreModel, na którego podstawie został utworzony jego CalModel. Dzięki temu z poziomu Avatar'a można odwoływać się do definicji postaci. Na rysunku 3 znajduje się ilustracja przykładowej struktury modeli.

3.2 Tworzenie avatarów

Aby utworzyć obiekt Avatar należy wcześniej utworzyć dla niego odpowiedni obiekt CalModel w Cal3d oraz użyć do tego odpowiedniego typu, czyli obiektu CalCoreModel. Obiekt CalCoreModel tworzony jest na podstawie pliku konfiguracyjnego w formacie Cal3d (.cfg). Nalezy przy tym pamiętać, że ten dany typ reprezentowany przez obiekt Cal-



Rysunek 3: Ilustracja przykładowej struktury modeli opartych na Cal3D

CoreModel wystarczy utworzyć (wczytać z pliku .cfg) tylko jeden raz. Następnie można go używać do tworzenia dowolnej ilości obiektów klasy CalModel.

Po utworzeniu danego obiektu CalCoreModel na podstawie pliku .cfg należy dla niego zainicjować materiały. Z kolei po utworzeniu obiektu CalModel należy odpowiednio zainicjować meshe na podstawie CalCoreModelu. Powyższe operacje należy wykonać w odpowiedniej kolejności.

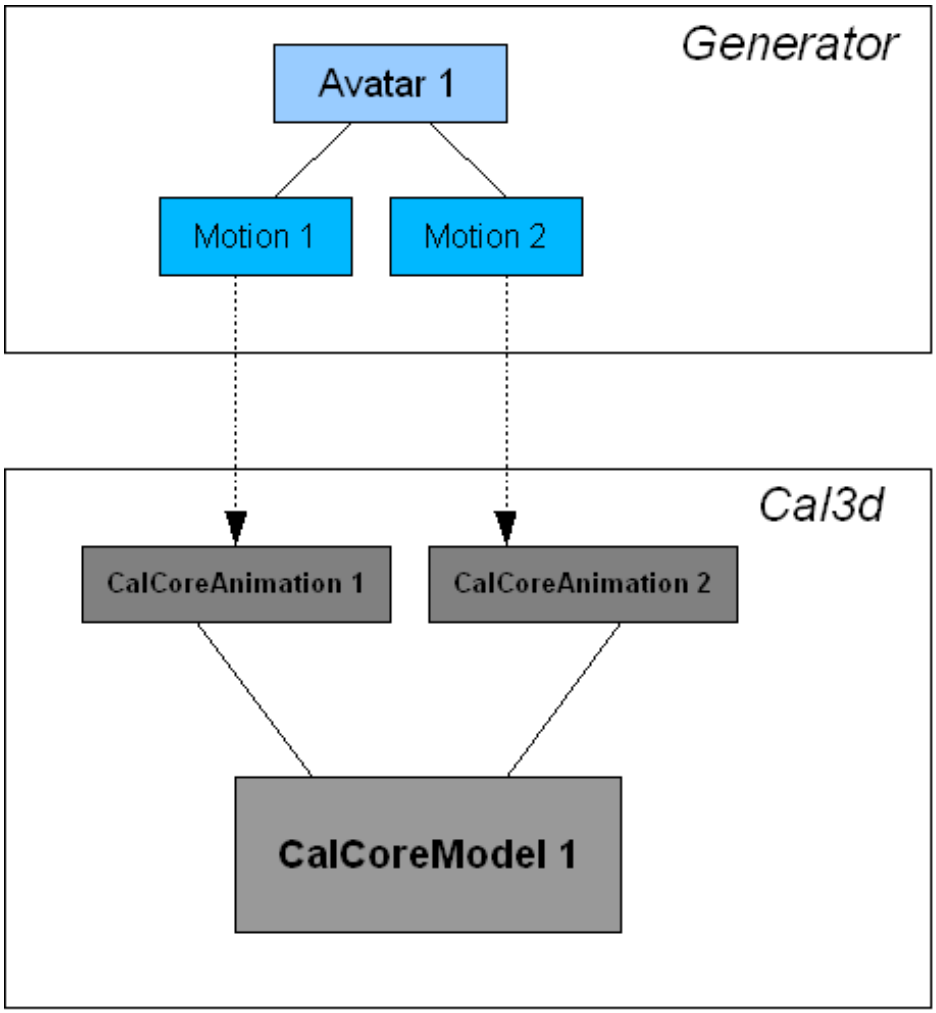
Aby uprościć tworzenie obiektu Avatar oraz jego relacji z modelami Cal3d została utworzona klasa AvatarFactory. Wystarczy wywołać metodę AvatarFactory.CreateAvatar() oraz zadać nazwę dla CalCoreModel'u (która odpowiada nazwie pliku .cfg) oraz dowolną nazwę dla Avatar. Cała operacja tworzenia Avatara odbywa się wewnątrz tej metody.

Dodatkowo klasa AvatarFactory zapewnia zarządzanie CalCoreModel'ami, tak aby każdy osobny typ nie został utworzony więcej niż jeden raz, nawet gdy odwołamy się do niego wiele razy

3.3 Przechowywanie ruchów (animacji) dla Avatarów

Animacje dla avatarów są reprezentowane przez obiekty klasy *Motion*. Są one kolekcjonowane w klasie *MovableAvatar*, która jest pochodną klasy *Avatar*. Jeden obiekt klasy *Motion* powiązany jest z dokładnie jednym obiektem klasy *CalCoreAnimation* w enginie Cal3d. Animacja CalCoreAnimation jest w Cal3d składową CalCoreModel'u i jest wykonywana przez odpowiednie CalModel'e. Ilustracja struktury powiązań dotyczących

animacji znajduje sie na rysunku 4.



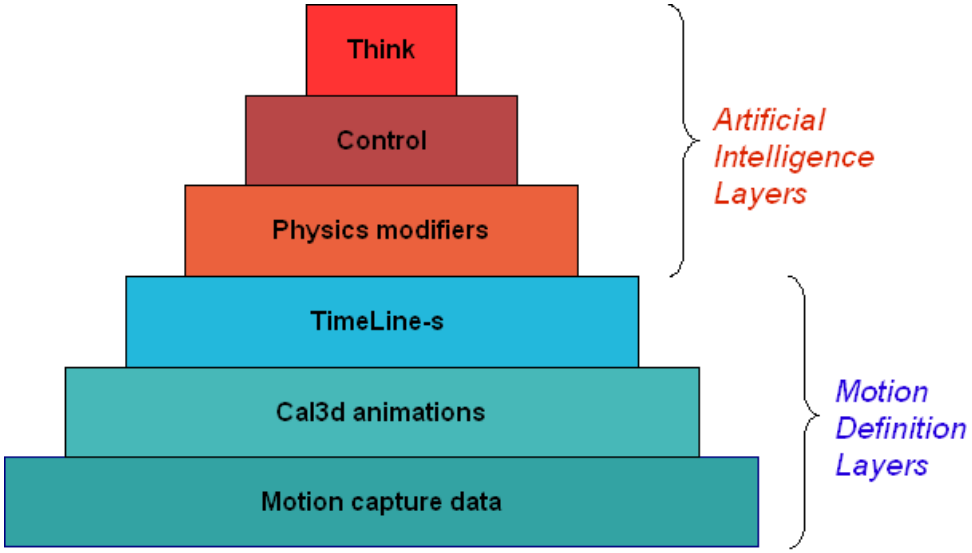
Rysunek 4: Powiązania struktur dotyczących animacji w Cal3D i Generatorze

4 Warstwy logiczne avatara

Strukturę avatara można podzielić na kilka warstw logicznych. Hierarchia w kodzie tych warst została przedstawiona na rys 5. W kolejnych podrozdziałach zostaną opisane funkcjonalności jakie są związane z poszczególnymi warstwami.

4.1 Warstwa danych 'Motion Capure'

Dane ruchów używane przez Generator to dane Motion Capture przekształcone do formatu Cal3d. Tylko w takim formacie dane te sa widoczne z modułów Generatora (nie ma możliwości dostępu do oryginalnego formatu zawartego w plikach motion capture - np. BVH).



Rysunek 5: Warstwy logiczne avatara

4.2 Warstwa animacji Cal3d

Dane ruchu używane w Generatorze mają formę animacji Cal3d. Odgrywanie ruchów polega na odgrywaniu tych animacji przez API zdefinowane przez engine Cal3d. Dostęp do szczegółowych danych dla poszczególnych kości polega na pobieraniu ścieżek animacji (tzw track-ów) z animacji Cal3d. Ścieżki te są parametryzowane czasem - można wyciągnąć dowolną klatkę animacji zadając odpowiedni czas $\mathbf{t}$ z zakresu od 0 do długości (czasu trwania) tej animacji. Dla pośrednich wartości czasu $\mathbf{t}$, które nie mają zdefiniowanych wartości w ścieżce, klatki sa wyliczane z użyciem interpolacji pomiędzy najblizszymi zdefiniowanymi klatkami w ścieżce.

4.3 Warstwa TimeLine'ów

TimeLine'y używane są do wygodnego definiowania ścieżki ruchu. Można tworzyć złożone ruchy, definiować blending pomiędzy nimi a także definiować specjalne modyfikatory, w których oryginalne dane ruchu mogą być modyfikowane w kolejnych krokach symulacji.

4.4 Warstwa 'Physics modifiers'

W warstwie tej przewidziane jest tworzenie modyfikatorów, które dostosowują zdefinowaną TimeLine'ami sekwencję ruchów aby poprawić realizm i podtrzymanie poprawności fizycznych aspektów ruchu np. eliminacja ślizgania się stóp po podłodze, poprawne stawianie kroków podczas skręcania lub wykorzystanie kinematyki odwrotnej.

4.5 Warstwa 'Control'

Warstwa 'Control' ma zapewnić możliwość definiowania akcji oraz powiązań pomiędzy akcjami na wyższym poziomie niż sama definicja sekwencji ruchu. Akcje mogą być wykony-

wane zgodnie ze zdarzeniami pochodzącymi od użytkownika lub zgodnie z algorytmami sztucznej inteligencji w warstwach wyższych.

4.6 Warstwa 'Think'

W warstwie 'Think' przewidziana jest implementacja algorytmów sztucznej inteligencji, które wprowadzają autnomiczne i inteligentne zachowanie avatarów np.: omijanie przeszkód czy wyznaczanie ścieżki do celu.

5 Sterowanie ruchem – TimeLine'y

Idea sterowania ruchem bazuje na pojęciu TimeLine'ow. TimeLine można traktować jako ścieżkę animacji, którą avatar ma za zadanie wykonać. TimeLine reprezentowany jest przez obiekty typu *ft::TimeLine*.

Zadany do wykonania TimeLine jest wypełniony obiektami typu *ft::TimeLineMotion*. Obiekty typu TimeLineMotion mają wskazania na animacje, które są wykonywane przez avatar'a w trakcie wykonywania danego TimeLineMotion'a na ścieżce animacji.

Pomiędzy kolejnymi obiektami TimeLineMotion mogą być zdefiniowane reguły łączenia. Reguły łączenia są reprezentowane przez obiekty typu *ft::TimeLineBlender*.

Podczas wykonywania TimeLineMotion'a, ruch avatara może być modyfikowany w dowolny sposób przez jeden lub kilka modyfikatorów ruchu. Każdy modyfikator jest opisany w obiekcie typu *ft::TimeLineModifier*.

Za wykonywanie TimeLine'ow odpowiedzialne są obiekty typu typu *ft::TimeLineExecutor*. Każdy avatar posiada jeden obiekt typu *ft::TimeLine* oraz jeden obiekt typu *ft::TimeLineExecutor*. Aby rozpocząć wykonywanie TimeLine'a przypisanego do danego avatara należy wywołać metodę *Start()* na TimeLineExecutor'ze tego avatara. Po wywołaniu tej metody rozpoczyna się wykonywanie kolejno wszystkich elementów (ruchów) składowych TimeLine'a.

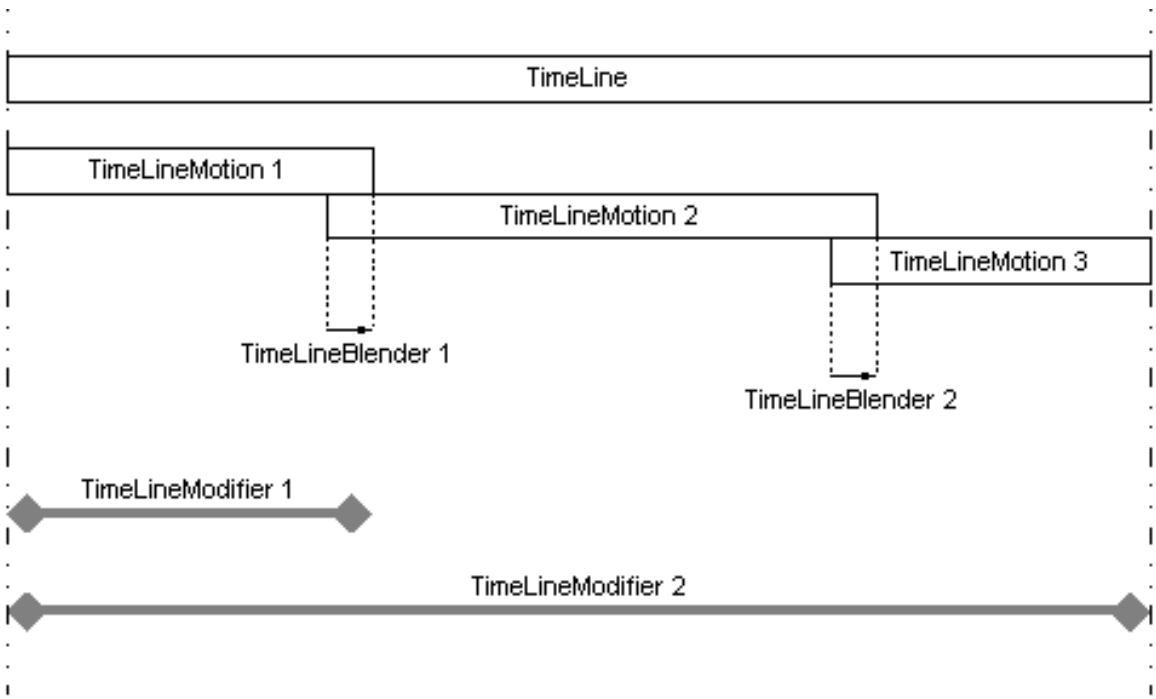
5.1 Struktura TimeLine'a oraz relacje pomiędzy jego składowymi elementami

Schemat przykładowego TimeLine'a znajduje się na rysunku 6.Ilustruje on TimeLine-a składającego się z trzech obiektów składowych: *TimeLineMotion_1*, *TimeLineMotion_2* oraz *TimeLineMotion_3*. Pomiędzy wszystkimi składowymi obiektami zdefiniowane są reguły łączenia: *TimeLineBlender_1* oraz *TimeLineBlender_2*. Poza tym zdefiniowane są dwa modyfikatory: *TimeLineMotdifier_1* (który modyfikuje ruch postaci jedynie w czasie wykonywania *TimeLineMotion_1*) oraz *TimeLineModifier_2* (który modyfikuje ruch podczas wykonywania całego TimeLine'a).

Wykonywanie TimeLine'a z przykładu polega na sekwencyjnym wykonaniu kolejno trzech zdefiniowanych TimeLineMotion'ów.

Reguły łączenia definiuje się dla konkretnego obiektu TimeLineMotion i zostaje on zastosowany pomiędzy tym obiektem a jego następnikiem (jeśli następnik występuje).

Modyfikatory również definiuje się dla obiektów typu TimeLineMotion. W powyższym przykładzie *TimeLineMotdifier_1* jest zdefiniowany dla *TimeLineMotion_1*, natomiast



Rysunek 6: Przepływ informacji w obrębie zdefiniowanego timeline-a

TimeLineMotdifier_2 dla *TimeLine* (takie powiązanie jest możliwe, ponieważ TimeLine jest specyficzną odmianą TimeLineMotion'a).

Każdy TimeLineMotion może być zaznaczony jako obiekt cykliczny, co powoduje, że będzie on wykonywany w pętli, dopóki nie zostanie jawnie przerwany. Dopiero po jego przerwaniu zacznie być wykonywany jego następnik. Istnieje również możliwość zdefiniowania liczby cyklów, po których wykonaniu ruch cykliczny zostanie przerwany automatycznie.

5.2 Struktura obiektów TimeLineMotion

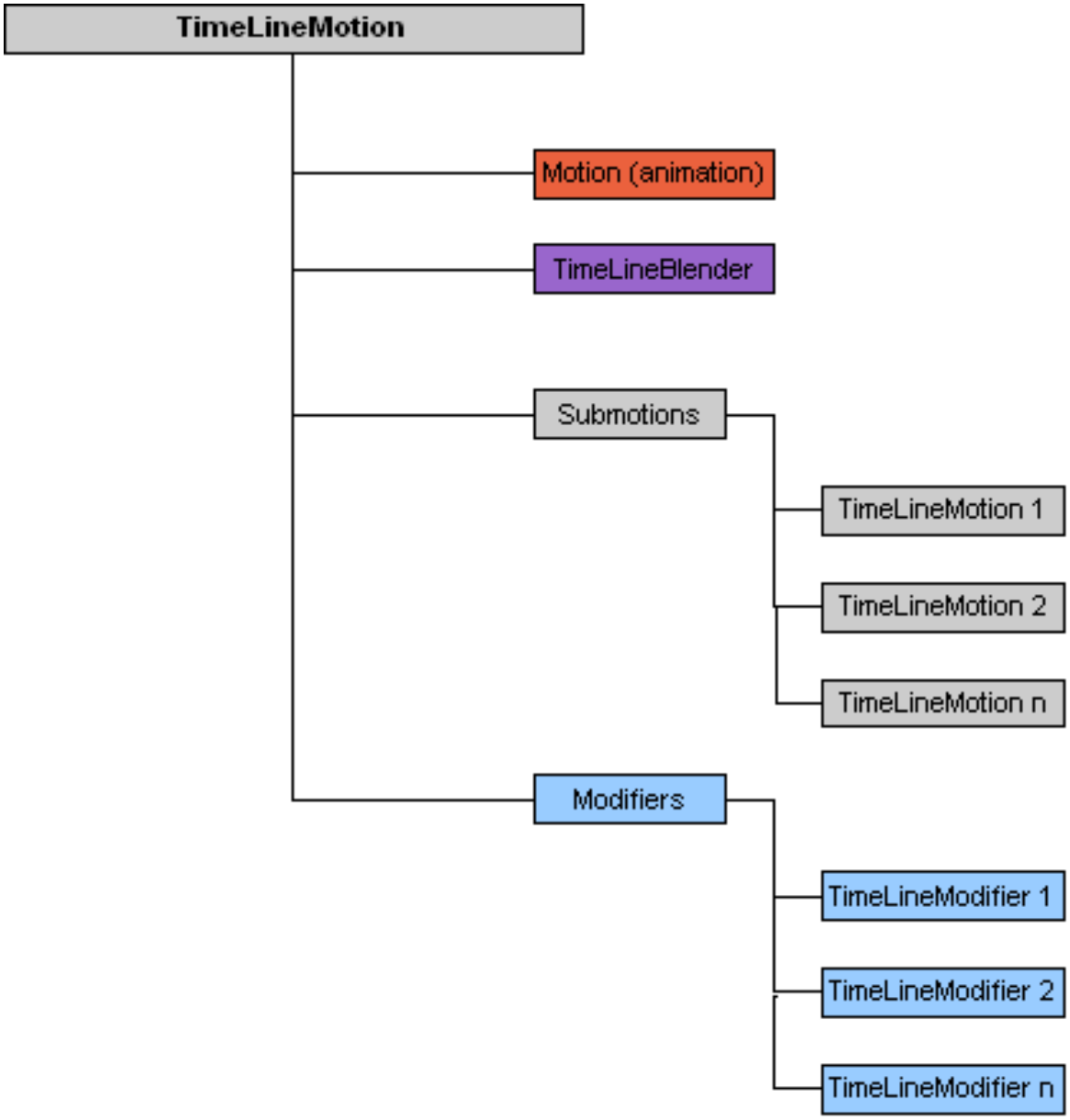
Elementy składowe obiektu TimeLineMotion przedstawione zostały na rysunku 7. W skład obiektu *ft::TimeLineMotion* wchodzą następujące elementy składowe:

a) *Motion (animacja)* – referencja do animacji, która ma być wykonana przez avatara w czasie wykonywania danego TimeLineMotion'a. Referencja do animacji może być pusta. W tym przypadku dany TimeLineMotion sam w sobie nie powoduje żadnego ruchu avatar'a, natomiast mogą go powodować jego elementy składowe z *tracks* i *submotions*.

b) *TimeLineBlender* – definicja łączenia danego TimeLineMotion'a z jego następnikiem. W najprostszym przypadku deiniuje on na ile przed końcem wykonyania animacji aktualnego TimeLineMotion'a ma być wystartowana animacja z następnego TimeLineMotion'a. Sama operacja *blendowania* realizowana jest automatycznie przez engine Cal3d.

c) *Submotions* – może zawierać sekwencje obiektów typu TimeLineMotion, które są wykonywane podczas wykonywania danego TimeLineMotion'a (równocześnie z wykonywaniem jego animacji). Zbiór *submotions* jest wykorzystywany do podziału danego TimeLineMotion'a na „krótsze" obiekty typu TimeLineMotion. Zbiór ten może być pusty – wtedy nie ma żadnego wpływu na ruch avatar'a.

d) *Modifiers* – zawiera zbiór modyfikatorów, które są wykonywane w czasie wykonywa-

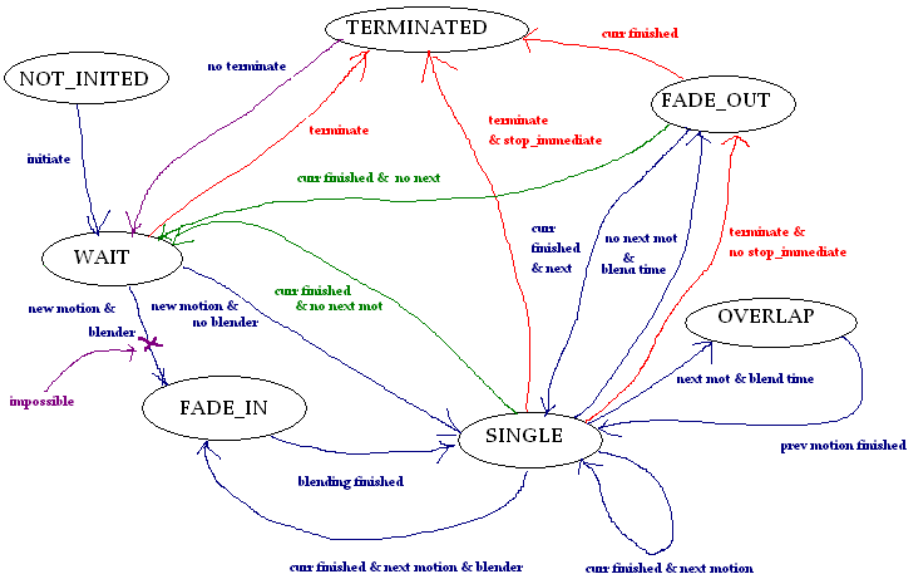


Rysunek 7: Elementy składowe typu *ft::TimeLineMotion*

nia danego TimeLineMotion'a (od początku jego wykonywania do zakończenia wykonywania). Każdy modyfikator może być podzielony dodatkowo na sekwencje „krótszych" modyfikatorów na tej samej zasadzie, zgodnie z którą można podzielić obiekt TimeLineMotion na zbiór submotions. Zbiór *modifiers* może być pusty – wtedy nie ma żadnego wpływu na ruch avatar'a.

5.3 Schemat TimeLineExecutor'a

W TimeLineExecutorze zdefiniowany jest zbiór stanów, pomiędzy którymi może on przechodzić wykonując TimeLine'a. Na rysunku 8 przedstawione są wszystkie możliwe stany jak i dozwolone przejścia pomiędzy nimi.



Rysunek 8: Schemat przejść pomiędzy stanami TimeLineExecutor'a

Znaczenie poszczególnych stanów:

1. 'NOT_INITED' - początkowy stan w jakim pozostaje TimeLineExecutor do momentu inicjacji

2. 'WAIT' - w tym stanie TimeLine nie zawiera żadnych ruchów do wykoniania (jest pusty lub wszystkie składowe ruchy zostały już wykonane)

3. 'FADE_IN' - w tym stanie wykonywany jest jeden ruch (jedna animacja) przy czym jego waga w Cal3d jest interpolowana od 0 do 1

4. 'SINGLE' - wykonywany jest pojedynczy ruch

5. 'OVERLAP' - stan ten reprezentuje żadkładkę- czyli blending pomiędzy dwoma kolejnymi ruchami

6. 'FADE_OUT' - wykonywany jest jeden ruch przy czym jego waga w Cal3d jest interpolowana od 1 do 0

7. 'TERMINATED' - ruch został nagle zatrzymany z zewnątrz przez metodę *TimeLineExecutor::StopRequest()*. Aktualnie wykonywana animacja jest zakończona natychmiast lub z zakonczeniem aktualnej animacji (lub aktualnego cyklu dla animacji cyklicznych)

5.4 Obiekt TimeLineContext

Podczas wykonywania TimeLine'a w TimeLineExecutorze zdefiniowany jest specjalny obiekt opisujący aktualny stan wykonywania tzw. TimeLineContext. Parametry, które są w nim zdefiniowane mogą być w każdym odświeżeniu wykorzystane poza TimeLineExcutorem np. w modyfikatorach. Najbardziej istotne parmetry TimeLineContextu, które mają funkcję informacyjną:

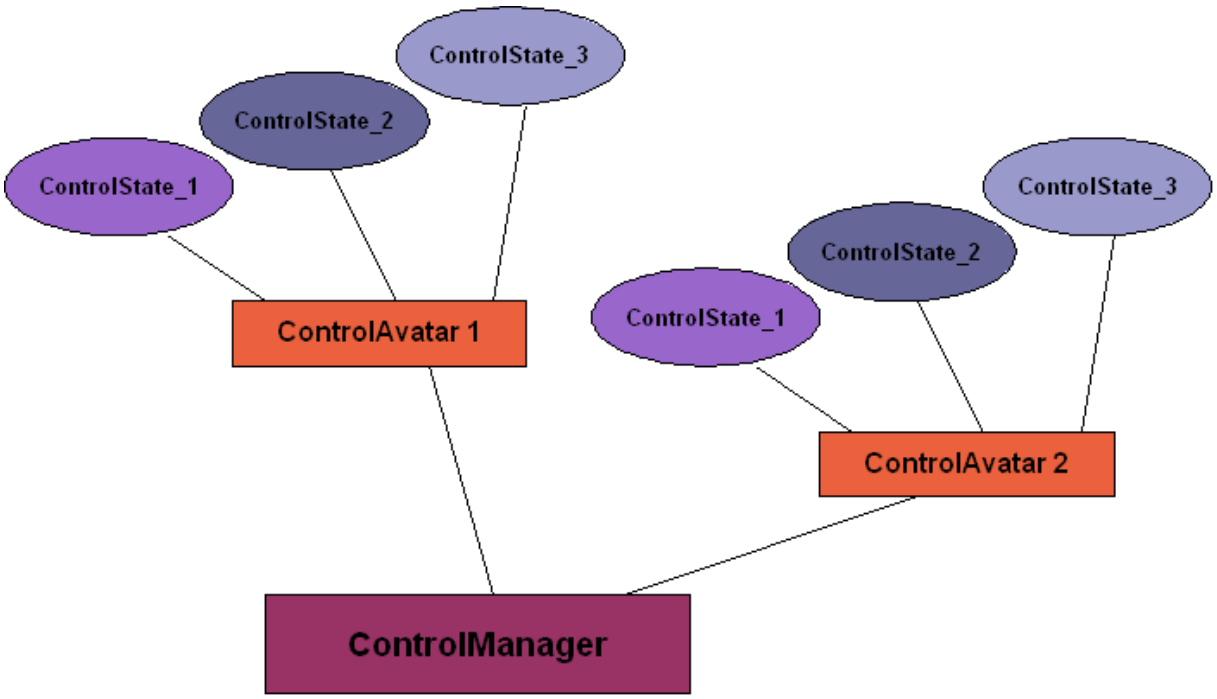
1. **prevAnim** (*CalAnimation**) - animacja wykonywana poprzednio
2. **currAnim** (*CalAnimation**) - animacja, która jest aktualnie wykonywana
3. **prevAnimTime** (*float*) - aktualny czas animacji poprzedniej (istotny na żakładce", gry poprzednia animacja ciągle trwa)
4. **prevAnimDuration** (*float*) - całkowita długość (czas trwania) poprzedniej animacji
5. **currAnimTime** (*float*) - aktualny czas aktualnie wykonywanej animacji
6. **currAnimDuration** (*float*) - całkowita długość (czas trwania) aktualnie wykonywanej animacji
7. **prevOverlap** (*float*) - czas *blendingu* zastosowanego dla poprzedniej animacji
8. **currOverlap** (*float*) - czas *blendingu*, który będzie użyty pomiędzy aktualna a następną animacją
9. **exec_state** (*int*) - aktualny stan w jakim jest TimeLineExecutor
10. **exec_event** (*int*) - informacja o ostatnim zdarzeniu wygenerowanym przez TimeLineExecutor (aktualnie są możliwe dwie wartości EXEC_EVENT_NONE lub EXEC_EVENT_STATE_CHANGED
11. **anim_changed** (*bool*) - ma wartość *true* jeśli w aktualnym odświeżeniu zmieniła się animacja
12. **anim_new_cycle** (*bool*) - ma wartość *true* jeśli w aktualnym odświeżeniu rozpoczął się nowy cykl jeśli właśnie wykonywana jest animacja cykliczna
13. **anim_stopped** (*bool*) - ma wartość *true* jeśli w aktualnym odświeżeniu animacja się skończyła lub została zatrzymana

Oprócz parametrów informacyjnych TimeLineContext zawiera również parametry, które można ustawić z zewnątrz:

1. **stop_immediate** (*bool*) - ustalenie czy zatrzymanie wykonywania TimeLine'a (przejście do stanu TERMINATED) ma być wykonane natychmiast czy po zakończeniu aktualnej animacji (lub aktualnego cycklu dla animacji cyklicznych)
2. **remove_after_execution** (*bool*) - ustalenie czy obiekty TimeLineMotion mają być usuwane z TimeLine'a po wykonaniu TimeLine'a

6 Moduł ControlManager

ControlManager jest modułem odpowiedzialnym za zarządzanie warstwą 'Control'. ControlManager zawiera liste avatarów na scenie, wśród których można definiować aktywnego avatara. Aktywny avatar reaguje na zdarzenia pochodzące od użytkownika (np. z klawiatury) poprzez wykonywanie odpowiednich akcji. Gdy żaden avatar nie jest aktywny w danym momencie to zależnie od zdarzenia akcje wykonywane są przez wszystkie avatary z listy ControlManager'a lub nie wykonywane są wcale.

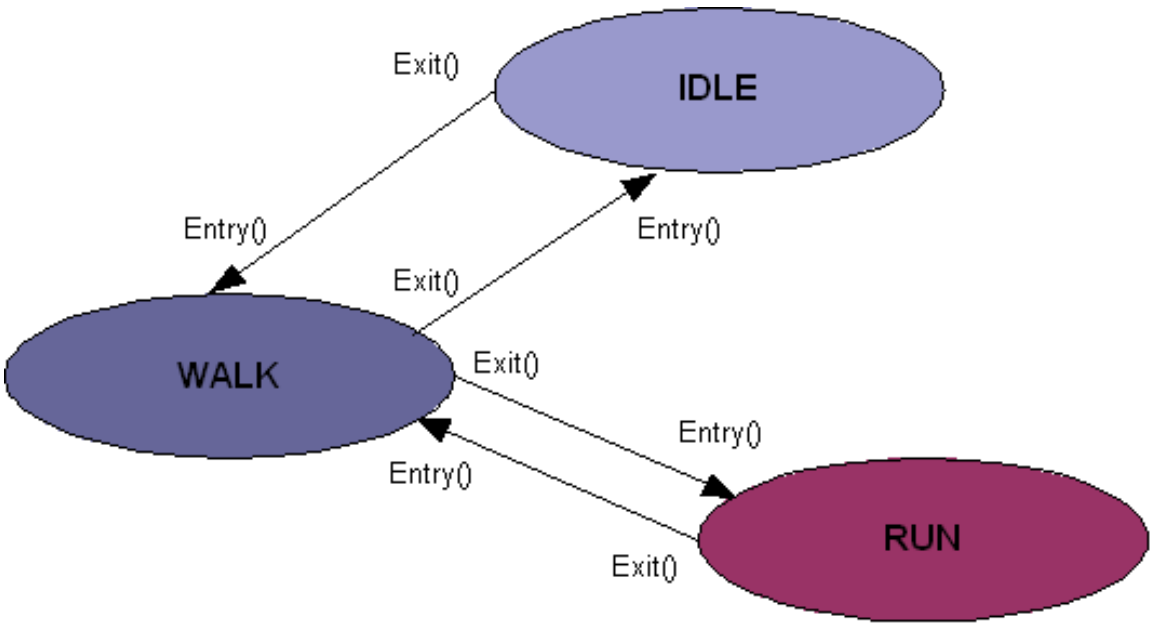


Rysunek 9: powiązania pomiędzy avatarami, stanami i ControlManager'em

Wykonywanie akcji przez avatara polega na przechodzeniu pomiędzy poszczególnymi stanami (tzw. *ControlState's*). Każdy avatar posiada własny i niezależny zbiór stanów, pomiędzy którymi może "przechodzić". Każdy stan ControlState posiada 3 metody, które wołane są w momencie inicjacji stanu (*Init()*), wejścia do tego stanu (*Entry()*) oraz wyjścia ze stanu (*Exit()*).

Na rysunku 9 przedstawione są powiązania pomiędzy avatarami, stanami i ControlManager'em.

Na rysunku 10 pokazany jest przykładowy schemat przejść pomiędzy stanami dla jednego avatara.



Rysunek 10: powiązania pomiędzy avatarami, stanami i ControlManager'em

7 Organizacja obiektow graficznych

7.1 Motywacja

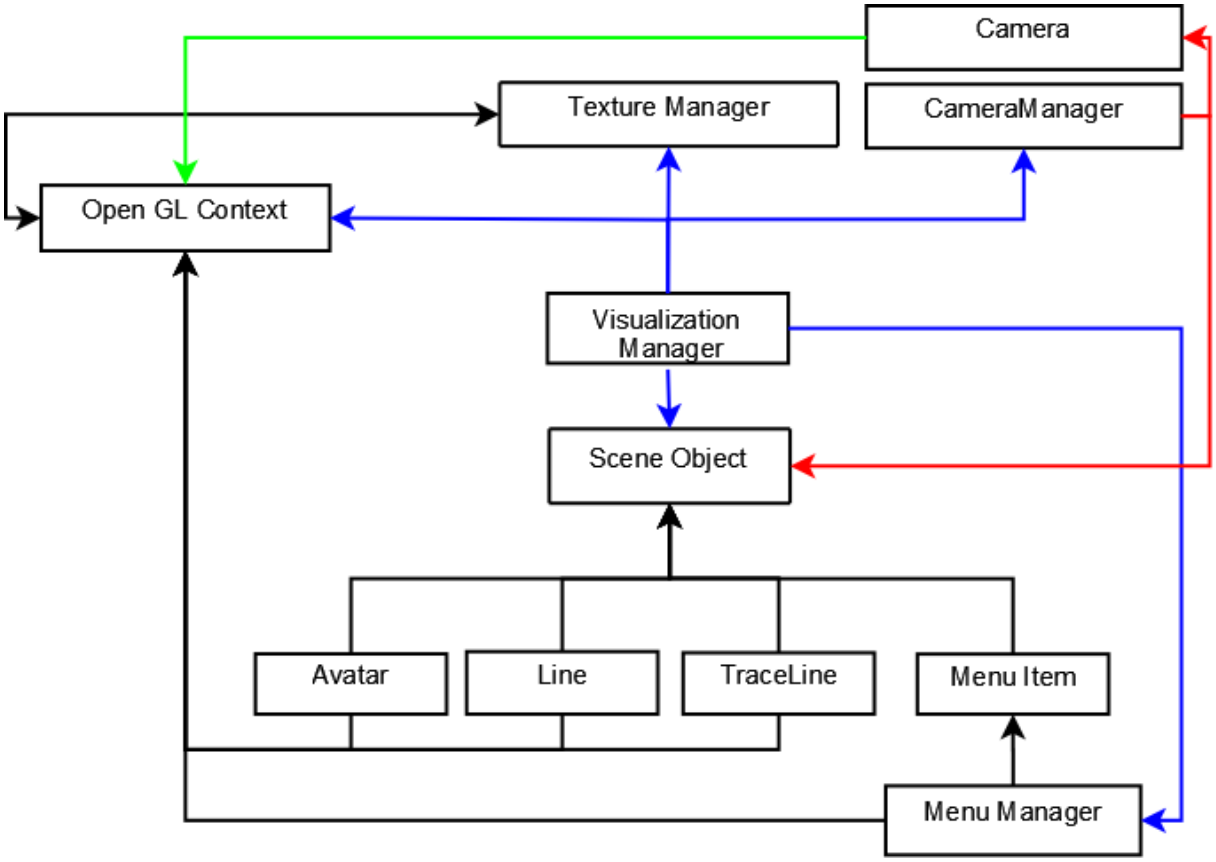
Wizualizacja nie jest głównym celem całego systemu, i przez to nie jest wykonana w sposób kompleksowy i całkowicie uniwersalny. Jednakże mechanizmy do wizualizacji elementów systemu bazują na pewnych założeniach pozwalających zaimplementować je łatwo w innych systemach wizualizacji (np. OSG) lub przy pomocy dowolnego API (np. DirectX). Założono niezależność od standardów korporacyjnych (MS) i wybrano OpenGL API. Aby zminimalizować wpływ strumienia graficznego na całkowitą wydajność systemu, przerzucono część operacji graficznych na procesor akceleratora graficznego przez użycie sprzętowego wspomagania (vertex shader).

7.2 Wizualizacja

Wizualizacja obiektów stanowi niezależny mechanizm generatora i poprzez ściśle określone reguły i interfejsy działa w sposób niezależny od reszty implementacji. Generalną ideę relacji pomiędzy podstawowymi obiektami wizualizacji w systemie przedstawiono na rysunku 11. **Zasady renderowania obiektów graficznych w systemie:**

1. Każdy obiekt który ma być włączony do potoku renderującego musi implementować interfejs (pokrywa metodę *Render*) obiektu *ft::SceneObject*.

2. Każdy obiekt, który ma być włączony do potoku renderującego jest musi być zarejestrowany przez obiekt *ft::VisualizationManager* przy pomocy metody *ft::VisualizationManager::AddObject*



Rysunek 11: Wzajemne relacje współpracy obiektów wizualizacji

3. Obiekt *ft::VisualizationManager* wywołuje cyklicznie, zsynchronizowaną z mechanizmem GLUT, metodę *ft::VisualizationManager::OnRender*, i przetwarza wszystkie zarejestrowane obiekty wywołując metodę Render każdego z nich.

7.2.1 ft::SceneObject

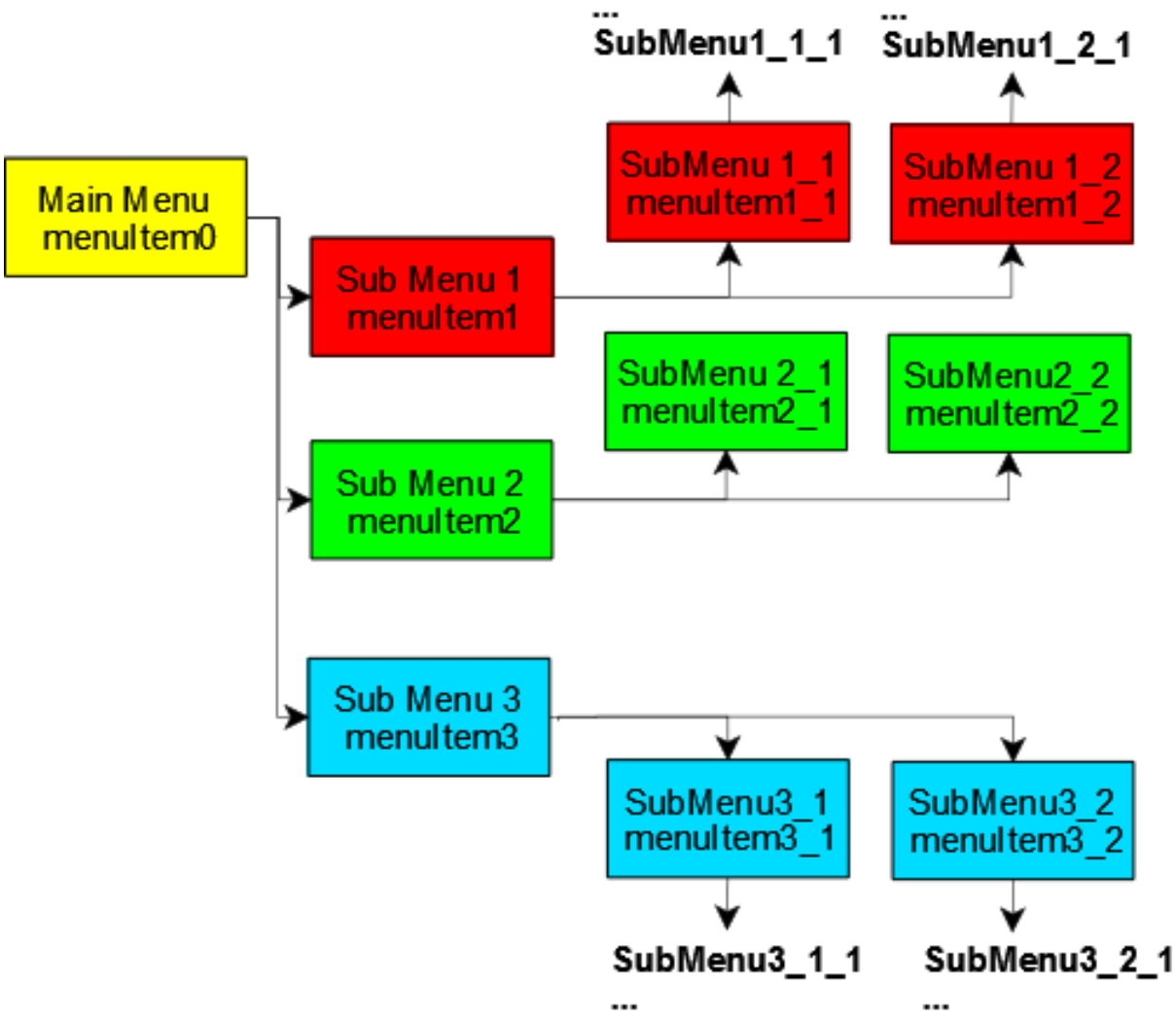
Podstawowy obiekt graficzny. Realizuje bazowy interfejs obiektu sceny (kolor, położenie, nazwa obiektu, aktywność) i udostępnia interfejs renderowania obiektu - metoda *Render* oraz *RenderShadow*.

7.2.2 ft::MenuItem

Podstawowy element menu graficznego, korzysta z bazowych własności typu *ft::SceneObject* Jego kształt i właściwości mogą być dostosowane do specyficznych wymagań poprzez własną implementację metody *Render*. MenuItem implementuje najprostszą postać wzorca composite dzięki czemu może funkcjonować jako struktura drzewiasta co pokazano na rysunku 12.

7.2.3 ft::Line

Pozwala realizować różne warianty linii lub strzałkę w trzech wymiarach. Obiekt można definiować zadając mu początek, koniec, długość, orientację i kolor. Główną motywacją



Rysunek 12: Przykładowa implementacja wielopoziomowego menu przy pomocy obiektu *ft::MenuItem*

było zastosowanie go w charakterze markera. W obszarze renderowania implementuje własną metodę *Render*.

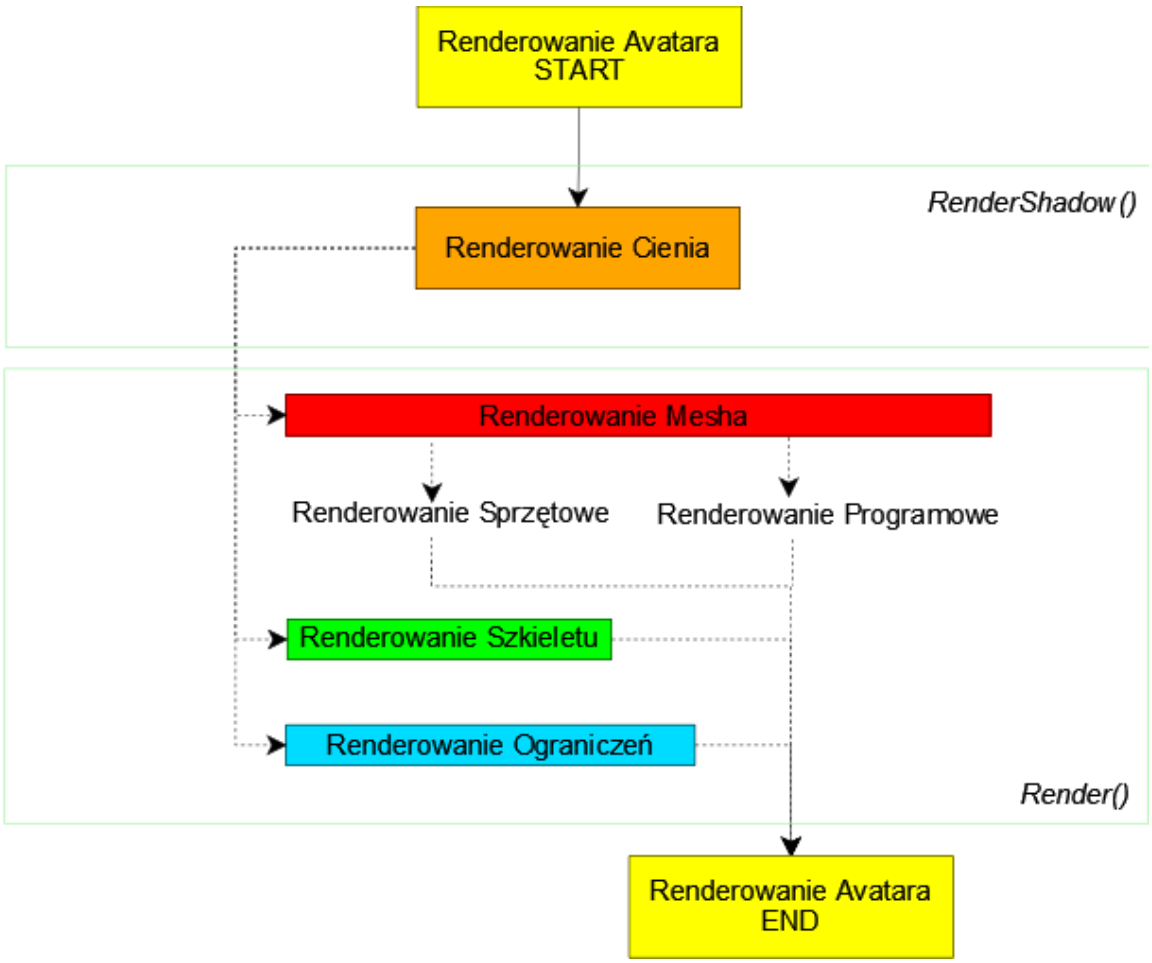
7.2.4 ft::TraceLine

Pozwala realizować linię wielosegmentową połączoną markerami w trzech wymiarach poprzez zadawanie punktu w przestrzeni metodą *ft::TraceLine::AddPoint*. Obiekt może wyświetlać i ukrywać markery, ustawiać kolor każdego segmentu. Doskonale nadaje się do wizualizacji miejsc, w których trzeba śledzić położenie przesuwającego się obiektu. W obszarze renderowania implementuje własną metodę *Render*.

7.2.5 ft:Avatar

W kontekście wizualizacji jest to obiekt graficzny z najbardziej rozbudowaną strukturą renderowania. Obiekt *ft::Avatar* posiada trzy możliwości renderowania: renderowanie

szkieletu, renderowanie ograniczeń każdej kości lub renderowanie siatki modelu (mesha). Dodatkowo ze względu na złożoność siatki modelu wprowadzono możliwość sprzętowego renderowania siatki modelu przy użyciu vertex shadera. Po wyborze metody renderowania Avatara dochodzi jeszcze renderowanie cienia obiektu, które jest realizowane przed renderowaniem całego obiektu przez metodę *RenderShadow* jako element globalnej metody renderowania cienia (np. przez *ft::VisualizationManager*). Przepływ potoku renderującego związanego z renderowaniem avatara przedstawiono na rysunku 13



Rysunek 13: Potok renderowania dla typu *ft::Avatar*. Linie przerywane oznaczają ścieżki alternatywne potoku renderującego.

7.2.6 ft::TextureManager

Ładuje, przechowuje i udostępnia innym obiektom tektury wczytywane z plików. Pozwala przetwarzać pliki graficzne w formatach PCX, BMP i TGA (wyłącznie 24 bitowe). Manager tekstur jest łatwo rozszerzalny i pozwala skorzystać z plików graficznych w innych formatach przez prostą modyfikację jednej metody *ft::TextureManager::LoadTexture*. Poprzez globalną mapę tekstur eliminuje potrzebę wielokrotnego wczytywania tych samych plików teksturami. Aby korzystać z właściwości tego obiektu musi być aktywny kontekst OpenGL do przetwarzania tekstur (glEnable(GL_TEXTURE)).

7.2.7 ft::MenuManager

Zarządza kolekcją obiektów typu *ft::MenuItem*. Tworzy menu graficzne na podstawie definicji w pliku konfiguracyjnym. Obsługuje komunikaty z zewnątrz od obiektu *ft::UpdateManager* i z lokalnych obiektów *ft::MenuItem* oraz generuje komunikat do systemu o wciśnięciu konkretnego przycisku w menu (MSG_MENU_ITEM_SELECTED) dla wszystkich zarejestrowanych obiektów nasłuchujących. Ze względu na interakcję przy pomocy klawiatury i myszy, korzysta z *ft::InputManagera* przy obsłudze komunikatów z tych urządzeń.

7.2.8 ft::VisualizationManager

Centralny element zarządzania elementami graficznymi sceny. Realizuje komunikację z pozostałymi niegraficznymi elementami systemu. Obiekt *ft::VisualizationManager* jest odpowiedzialny za renderowanie wszystkich obiektów graficznych typu *ft:SceneObject*, przy pomocy metody *Render3DObjects*. Wszystkie obiekty, które mają być automatycznie renderowane muszą być uprzednio zarejestrowane do renderowania metodą *ft::VisualizationManager::AddObject*.

7.2.9 ft::OGLContext

Tworzy zawartość renderowania (prymitywy graficzne) przy pomocy API OpenGL. Buduje wizualne, trwałe elementy sceny (podłoga, logo), korzysta z tekstur obiektu *ft::TextureManager*.

7.2.10 ft::Camera

Tworzy cztery rodzaje kamer, oraz aktualizuje widok dla aktywnej kamery. Jest obiektem zarządzanym całkowicie przez *ft::CameraManager*.

7.3 ft::CameraManager

Zarządza realcjami pomiędzy stworzonymi kamerami i obiektami sceny. Tworzy dla dowolnego obiektu sceny kamerę, której identyfikator odpowiada ID przypisanego obiektu sceny. Przetwarza komunikaty z klawiatury i myszy dotyczące aktywnej kamery. Aktualizuje parametry bieżącej kamery oraz widoku dla kontekstu rederowania *ft::VisualizationManager*. *ft::CameraManager* zarządza kolekcją kamer. Do podstawowych operacji należy:

1. Przełączanie bieżącego widoku między zdefiniowanymi kamerami - klawisze **[** oraz **]**
2. Powiększenie obserwowanego fragmentu - klawisz |
3. Zmiana rodzaju bieżącej kamery - klawisz \

Do przełączania się pomiędzy zdefiniowanymi kamerami w systemie służą klawisze **[** oraz **]**. Opcja powiększenia (zoom) ułatwia obserwację szczegółów widoku. Zoom włącza się oraz wyłącza za pomocą jednokrotnego wciścięcia klawisza |. Zmiana rodzaju bieżącej kamery następuje po wciśnięciu klawisza. W systemie zdefiniowano 4 rodzaje kamer:

1. *StaticCamera* - sztywna kamera bez możliwości poruszania, pozwala tylko na statyczny widok z określonego miejsca
2. *ThirdPersonCamera* - kamera z pozycji trzeciej osoby. Podąża za celem i nie można nią sterować
3. *OrbitCamera* - kamera obraca się wokół punktu celu.
4. *FlyCamera* - kamera umożliwia przesuwanie widoku manualnie:
   (a) Do przodu - klawisz **w**.
   (b) Do tyłu - klawisz **s**.
   (c) W lewo - klawisz **a**.
   (d) W prawo - klawisz **d**.
   (e) W górę - klawisz **r**.
   (f) W dół - klawisz **f**.
   (g) Obrót o 360 stopni (odchylenie) - wciśnięty lewy klawisz myszy i przesuwanie jej w poziomie.
   (h) Obrót w zakresie +-90 stopni w górę i w dół od płaszczyzny bieżącego widoku (nachylenie) - wciśnięty lewy klawisz myszy i przesuwanie jej w pionie.

W wypadku, gdy kamera nie jest dowiązana do dynamicznego obiektu sceny, tryb *ThirdPersonCamera* jest dla niej niedostępny. Przykładem tego typu jest główna kamera *MainCamera*. Obiekt, do którego przypisana jest bieżąca kamera jest oznaczony migającym kwadratem w odpowiednim kolorze - w zależności od rodzaju aktywnej kamery. Dla trybu *StaticCamera* jest to kolor zielony, dla *ThirdPersonCamera* kolor pomarańczowy, dla *OrbitCamera* kolor czerwony oraz kolor jasny niebieski dla trybu *FlyCamera*.

7.4 CameraConfiguration

Pozwala zmieniać wartości parametrów zdefiniowanych w systemie kamer i umożliwia dostęp do takich konfiguracji przez zdefiniowanie klawisza dostępu. Definicja odbywa się w pliku konfiguracyjnym aplikacji i została opisana rozdziale dotyczącym konfiguraji.

7.5 Renderowanie Sprzętowe

Każdy obiekt typu *ft::SceneObject* może implementować własne sprzętowe renderowanie przy użyciu języka assemblera ARB: *ARB vertex program* oraz/lub *ARB fragment program*. Aby zrealizować sprzętowe renderowanie wybranego obiektu należy:

1. Utworzyć pliki z kodem vertex shaderów i pixel shaderów dla fragmentów lub całego obiektu graficznego w katalogu **shaders**. Shadery dla pixel shaderów (fragment programy) mają rozszerzenie *.frag* natomiast dla vertex shaderów (vertex programy) rozszerzenie *.vert*.
2. Utworzyć w kodzie obiektu graficznego metody do inicjalizacji i wczytywania shaderów oraz rezerwacji pamięci, analogicznie do metod *Avatar::InitHardwareAcceleration* i *Avatar::loadBufferObject*.

3. Utworzyć w kodzie obiektu graficznego metodę renderującą obiekt przy pomocy sprzętu analogiczną do *Avatar::HardwareRenderModelMesh*

8 System debug'ów

Wszystkie debugi wypisywane na konsole z Generatora, które maja być dodane do oficjalnej wersji w SVN-ie, powinny być wypisywane z użyciem metod zdefiniowanych w klase *ft::Debug* z pakietu */utility*.

Do wypisywania tekstu zaleca się uzywanie polecania *_dbg* (zdefniowanego w pliku */utility.debug.h*) zamiast standardowych polecen *cout*, *cerr* lub *printf*.

Dodatkowo zaleca się używanie warunków z użyciem flag zdefniowanych w klasie *ft::Debug* (np. flaga *ERR* definiuje wypisywania debugow zwiazanych z bledami) .

Przykladowe wypisanie debug'a:

if (Debug::FLAG) _dbg « "To jest debug «< endl;

8.1 Błędy i ostrzeżenia

Do wypisywania bledow i warningow sluza specjalne flagi *Debug:ERR* oraz *Debug::WARN*. Sa one typu bool i kazde wypisanie bledu lub ostrzezenia należy ograniczyc warunkiem z uzyciem tych flag. Dodatkowo ztefiniowane sa slowa kluczowe dla bledow i ostrzezen, których warto uzywac w celu ulatwienia w przeszukiwaniu loga: *Debug::ERR_STR* oraz *Debug::WARN_STR*.

Przykłady:

if (Debug::ERR) _dbg « Debug::ERR_STR « "To jest blad !!! «< endl;

if (Debug::WARN) _dbg « Debug::ERR_WARN « Óstrzezenie !!! «< endl;

8.2 Poziomy debugów dla modulów

Oprócz bledów i ostrzeżeń zostały zdefiniowane poziomy debugów dla modułów. Dla każdego modulu (lub logicznej czesci kodu zwiazanej z dana funkcjonalnoscia) należy dodac statyczna zmienna typu int do klasy *Debug*. Nastepnie wypisujac debugi dla danego modulu należy tej zmiennej uzywac sprawdzajac czy zostal ustawiony wystarczajacy poziom debugow dla modulu np.:

if (Debug::CONTROL>1) _dbg « "To jest debug z ControlManagera «< endl;

Powyzszy debug zostanie wypisany jeśli poziom debugow *CONTROL* zostanie ustawiony na wyzszy niż 1 (czyli np. 2).

Dodawanie zmiennych dla powyzszych debugow wymaga wykonania nastepujacych krokow:

1. dodajemy publiczna statyczna zmienna *CONTROL* do klasy Debug w pliku debug.h

static int CONTROL;

2. dodajemy inicjalizacje zmiennej statyczne CONTROL w pliku debug.cpp

   int Debug::CONTROL = 0;

3. dodajemy operacje wczytania wartosci poziomu dla CONTROL z pliku application.cfg - robimy to w metodzie Debug::LoadLevelsFromConfig():

   CONTROL = ReadLevelFromConfig("DEBUG_CONTROL");

Teraz gdy ustawimy w pliku application.cfg wpis np. *DEBUG_CONTROL = 2* to poziom debugów *CONTROL* zostanie ustawiony na 2.

8.3 Debugi dla projektu 'genenerator'

Poziomy debugów dla modulów w projekcie 'generator' warto dodawać w klasie *GenDebug* (pakiet */app*). Dotyczy to również używania systemu debugów w kodzie - czyli wszędzie tam gdzie użylibysmy klasy *Debug* powinnismy użyć klasy *GenDebug*. Zaletą używania klasy *GenDebug* jest to, że nie musmy zmieniać/kompilować źródeł projektu 'engine'.