|  |
| --- |
| [Wpisz nazwę firmy] |
| Dokumentacja silnika StarWars Engine |
| [Wpisz podtytuł dokumentu] |

|  |
| --- |
| Witold Dzięcioł  [Wybierz datę] |

Spis treści

[1 Przewodnik dla użytkowników 4](#_Toc407124633)

[2 Dokumentacja dla twórców 4](#_Toc407124634)

[2.1 Opis tematyczny 4](#_Toc407124635)

[2.1.1 Eventy 4](#_Toc407124636)

[2.1.2 Inicjowanie silnika 4](#_Toc407124637)

[2.1.3 Interfejs graficzny 4](#_Toc407124638)

[2.1.4 Kamera i sterowanie widokiem 4](#_Toc407124639)

[2.1.5 Kontrolery 4](#_Toc407124640)

[2.1.6 Obsługa klawiatury, myszy itp. 4](#_Toc407124641)

[2.1.7 Pętla główna silnika 5](#_Toc407124642)

[2.1.8 Potok przetwarzania obiektów 6](#_Toc407124643)

[2.1.9 Renderowanie GUI 9](#_Toc407124644)

[2.1.10 Renderowanie sceny 9](#_Toc407124645)

[2.1.11 Układ współrzednych 9](#_Toc407124646)

[2.2 Podstawowe obiekty i struktury danych 9](#_Toc407124647)

[2.2.1 event\_mapping 9](#_Toc407124648)

[2.2.2 InputAbstractionLayer 9](#_Toc407124649)

[2.2.3 InputAbstractionLayer\_base 17](#_Toc407124650)

[2.2.4 input\_mapping 18](#_Toc407124651)

[2.2.5 MaterialObject 19](#_Toc407124652)

[2.2.6 MeshObject 19](#_Toc407124653)

[2.2.7 MeshPart 19](#_Toc407124654)

[2.2.8 Model3DFromFile 19](#_Toc407124655)

[2.2.9 referenced\_object 27](#_Toc407124656)

[2.2.10 TextureObject 31](#_Toc407124657)

[2.2.11 Vertex\_Normal\_TexCords1 31](#_Toc407124658)

[2.2.12 Vertex\_Normal\_TexCords2 31](#_Toc407124659)

[2.2.13 Vertex\_TexCords1 31](#_Toc407124660)

[2.2.14 VertexUI 31](#_Toc407124661)

[2.3 Klasy predefiniowane 31](#_Toc407124662)

[2.4 Klasy bazowe dla obiektów predefiniowanych 31](#_Toc407124663)

[2.4.1 Animation\_object 31](#_Toc407124664)

[2.4.2 Base\_AI\_controller 31](#_Toc407124665)

[2.4.3 Base\_input\_controller 31](#_Toc407124666)

[2.4.4 Camera\_object 31](#_Toc407124667)

[2.4.5 Collision\_object 31](#_Toc407124668)

[2.4.6 Complex\_object 31](#_Toc407124669)

[2.4.7 Controller 32](#_Toc407124670)

[2.4.8 Dynamic\_mesh\_object 32](#_Toc407124671)

[2.4.9 Dynamic\_object 33](#_Toc407124672)

[2.4.10 Event 33](#_Toc407124673)

[2.4.11 Instanced\_mesh 33](#_Toc407124674)

[2.4.12 Object 33](#_Toc407124675)

[2.4.13 Physical\_object 35](#_Toc407124676)

[2.4.14 Static\_object 35](#_Toc407124677)

[2.4.15 Trigger\_object 35](#_Toc407124678)

[2.5 Interfejsy 35](#_Toc407124679)

[2.5.1 engine\_interface 35](#_Toc407124680)

[2.5.2 GamePlay 37](#_Toc407124681)

[2.5.3 Loader 37](#_Toc407124682)

[2.6 Moduły silnika 39](#_Toc407124683)

[2.6.1 CollisionEngine 39](#_Toc407124684)

[2.6.2 ControllersEngine 39](#_Toc407124685)

[2.6.3 DisplayEngine 39](#_Toc407124686)

[2.6.4 Engine 41](#_Toc407124687)

[2.6.5 FableEngine 47](#_Toc407124688)

[2.6.6 ModelsManager 47](#_Toc407124689)

[2.6.7 MovementEngine 54](#_Toc407124690)

[2.6.8 PhysicEngine 54](#_Toc407124691)

[2.6.9 UI\_Engine 54](#_Toc407124692)

[2.7 Stałe i makrodefinicje 54](#_Toc407124693)

[2.7.1 \_\_UNUSED 54](#_Toc407124694)

[2.7.2 EVENT\_TYPE 55](#_Toc407124695)

[2.7.3 FIXED\_MOVE\_UPDATE\_INTERVAL 55](#_Toc407124696)

[2.7.4 FRAMES\_PER\_SEC\_UPDATE 55](#_Toc407124697)

[2.7.5 MESH\_LOADING 55](#_Toc407124698)

[2.7.6 Modules 55](#_Toc407124699)

[2.7.7 STANDARD\_ABSTRACTION\_LAYER 56](#_Toc407124700)

[2.7.8 STANDARD\_ABSTRACTION\_LAYER\_COUNT 57](#_Toc407124701)

[2.7.9 WRONG\_ID 57](#_Toc407124702)

[2.7.10 WRONG\_MATERIAL\_ID 57](#_Toc407124703)

[2.7.11 WRONG\_MESH\_ID 57](#_Toc407124704)

[2.7.12 WRONG\_MODEL\_FILE\_ID 57](#_Toc407124705)

[2.7.13 WRONG\_TEXTURE\_ID 57](#_Toc407124706)

[3 API dla użytkowników silnika 57](#_Toc407124707)

# Przewodnik dla użytkowników

# Dokumentacja dla twórców

## Opis tematyczny

### Eventy

### Inicjowanie silnika

### Interfejs graficzny

### Kamera i sterowanie widokiem

### Kontrolery

Do czego służą kotrolery:

Kotrolery pre i post:

Istnieją kotrolery, których ruch uzależniony jest od położenia innych obiektów. Z racji tego nie mogą one zostać wywołane przed policzeniem przemieszczeń, bo byłyby zawsze opóźnione o jedną klatkę. Pozostałe kotrolery służą do wpływania na ruch obiektów, dlatego nie mogą być wywołane, po obliczeniu położeń obiektów. Z tego względu kotrolery obsługiwane są dwukrotnie, jedne w funkcji ControllersEngine:: proceed\_controllers\_pre tuż przed wywołaniem MovementEngine:: proceed\_movement, a drugie tuż po tym wywołaniu w funkcji ControllersEngine:: proceed\_controllers\_post.

Do kontrolerów post zaliczamy kotrolery śledzące jakis obiekt, do kotrolerów pre wszystkie kontrolery odpowiedzialne za AI oraz sterujące ruchem na podstawie klawiatury czy myszy.

### Obsługa klawiatury, myszy itp.

Definicja:

" Engine\UI\_engine\UI\_engine.cpp"

Informacje ogólne:

Za obsługę urządzeń wejściowych odpowiada klasa [UI\_engine](#_UI_Engine). Inicjalizacja odpowiednich modułów DirectXa następuje w funkcji [UI\_engine](#_UI_Engine)::init\_direct\_input. Stan urządzeń wejściowych sprawdzany jest w każdym obiegu pętli przetwarzania obiektów (patrz: [Potok przetwarzania obiektów](#_Potok_przetwarzania_obiektów)) w funkcji [UI\_engine](#_UI_Engine)::proceed\_input.

Obsługa urządzeń za pomocą DirectXa polega na odczytywaniu stanów wszystkich przycisków z odpowiednich tablic.(Po dokładniejsze informacje odsyłam do MSDNu temat Direct Input). Nie istnieje tu mechanizm wysyłania komunikatów systemowych tak jak w WinAPI lub innych platformach obsługujących GUI, dlatego , tak gdzie taka funkcjonalność jest potrzebna, musimy ją napisać sami używając mechanizmu eventów.

Warstwa abstrakcji:

{do przemyślenia, stworzenia, napisania}

Ten sam klawisz na klawiaturze może robić różne rzeczy w zależności od kontekstu w jakim został przyciśniety. Inaczej trzeba zareagować, kiedy myszka jest kliknięta w głównej fazie gry, inaczej gdy jesteśmy w menu. Silnik musi dawać możliwość łatwej obsługi oraz tworznia wielu różnych kontekstów, do czego potrzebne jest wprowadzenie dodatkowej warstwy abstrakcji pomiędzy bezpośrednim odczytem stanu urządzeń wejściowych, a wykonaniem jakiejś akcji.

Zacznijmy od zdefiniowania kto i w jaki sposób może uzyskać dostęp do odczytu z urządzeń wejściowych. Są to:

* kontrolery dziedziczące po [Standard\_input\_controller](#_Standard_input_controller) – wymagają dostępu do jednej lub kilku warstw abstrakcji w zależności od tego, co obsługują
* obiekt [GamePlay](#_GamePlay) – powinien dostać event informujący o wciśnięciu (puszczeniu) klawisza
* obiekt [GamePlay](#_GamePlay) – może chcieć w dowolnym momencie sprawdzić czy jakiś przycisk jest wcisnięty {opcja do rozważenia}
* obiekt [UI\_engine](#_UI_Engine) – musi odczytywać stany urządzeń, aby sprawdzać wciśnięcie przycisku (przycisku GUI, nie na klawiaturze) lub interakcję z innymi kontrolkami. Następnie ma wysłać event wysokopoziomowy o interakcji.

### Pętla główna silnika

Definicja:

" Engine\_window\_functions.cpp"

Kod:

int Engine::main\_loop()

{

MSG msg;

time\_previous = GetTickCount(); //inicjujemy licznik czasu

elapsed\_time = time\_previous;

// Main message loop:

while (TRUE)

{

if ( directX\_ready )

render\_frame();

while (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

if (msg.message == WM\_QUIT)

break;

}

return (int)msg.wParam;

}

Opis:

Pętla główna silnika zastępuje Windowsowską pętlę komunikatów, wywołując jednocześnie pętlę główną silnika (zobacz temat: [Potok przetwarzania obiektów](#_Potok_przetwarzania_obiektów)). Musimy obsługiwać komunikaty windowsa, ponieważ w przeciwnym razie mógłby on uznać, że nasz program się zawiesił i spróbować zabić nasz proces. Dlatego pobieramy wiadomości z kolejki komunikatów i przekazujemy do dalszego przetwarzania. Do pobierania komunikatów używamy nieblokującej funkcji PeekMessage (nie GetMessage).

Jako pętla główna silnika służy funkcja [render\_frame](#_Engine::render_frame). Jest ona wywoływana tylko pod warunkiem, że wcześniejsza inicjalizacja directXa w pełni powiodła się. Ważnym punktem jest inicjacja zmiennej time\_previous. Niezainicjowanie tej zmiennej, mogłoby spowodować, że w pierwszej klatce nastąpiłaby nieprzewidywalna zmiana położeń i orientacji wszystkich obiektów na scenie.

Wyjście z aplikacji następuje kiedy w kolejce komunikatów windowsa znajdzie się komunikat WM\_QUIT. Z wnetrza silnika można to osiągnąć przez wywołanie funkcji [end\_aplication](#_Engine::end_aplication).

### Potok przetwarzania obiektów

Definicja:

" Engine\_window\_functions.cpp"

void Engine::render\_frame()

{

float time\_interval;

time\_controller( time\_interval );

//muszą się wykonać w także w trybie pauzy

ui\_engine->proceed\_input( time\_interval );

while ( lag >= FIXED\_MOVE\_UPDATE\_INTERVAL )

{

physic\_engine->proceed\_physic( FIXED\_MOVE\_UPDATE\_INTERVAL );

controllers\_engine->proceed\_controllers( FIXED\_MOVE\_UPDATE\_INTERVAL );

movement\_engine->proceed\_movement( FIXED\_MOVE\_UPDATE\_INTERVAL );

collision\_engine->proceed\_collisions( FIXED\_MOVE\_UPDATE\_INTERVAL );

fable\_engine->proceed\_fable( time\_interval );

lag -= FIXED\_MOVE\_UPDATE\_INTERVAL;

}

//Renderujemy scenę oraz interfejs użytkownika

directX\_device->Clear(0, NULL, D3DCLEAR\_TARGET, D3DCOLOR\_XRGB(0, 0, 0), 1.0f, 0);

directX\_device->Clear( 0, nullptr, D3DCLEAR\_ZBUFFER, D3DCOLOR\_XRGB( 0, 0, 0 ), 1.0f, 0 );

directX\_device->BeginScene(); // begins the 3D scene

display\_engine->display\_scene(time\_interval);

ui\_engine->draw\_GUI(time\_interval);

directX\_device->EndScene(); // ends the 3D scene

directX\_device->Present(NULL, NULL, NULL, NULL); // displays the created frame

}

Ogólne informacje:

Za potok przetwarzania obiektów odpowiada funkcja [Engine::render\_frame](#_Engine::render_frame). Funkcja ta jest wywoływana w [pętli głównej silnika](#_Pętla_główna_silnika), konkretniej w funkcji [Engine](#_Engine)::[main\_loop](#_Engine::main_loop), w każdym możliwym momencie, kiedy nie są przetwarzane komunikaty z kolejki komunikatów WinApi.

Funkcja wywołuje w odpowiedniej kolejności poszczególne moduły silnika. Moduły odpowiedzialne za ruch obiektów na scenie i ich interakcje wywoływane są ze stałą częstotliwością określoną przez stałą [FIXED\_MOVE\_UPDATE\_INTERVAL](#_FIXED_MOVE_UPDATE_INTERVAL). Oznacza to, że w niektórych obiegach pętli mogą nie zostać wywołane, jeżeli odległość w czasie od odstatniego wywołania jest za mała. Taka architektura ma na celu spowodowanie, żeby gra działała tak samo zarówno na szybkim sprzęcie jak i na wolnym. Zauważmy, że jedno przesunięcie obiektów po czasie t, nie jest równoważne dwóm przesunięciom po czasie t/2. Dzieje się tak ze względu na niedokładność obliczeń wykonywanych na floatach. O ile w tym przypadku różnice nie muszą być duże, to różnica zachowania sztucznej inteligencji w tych dwóch sytuacjach jest nie do przewidzenia.

Za pomocą tego samego mechanizmu realizowane jest pauzowanie gry. Wtedy czas nie przyrasta, więc odpowiednie bloki nie zostają wywołane.

Moduły silnika odpowiedzialne za renderowanie obiektów, GUI oraz przechwytywanie urządzeń wejściowych są wywoływane w każdym obiegu pętli.

Obsługa eventów wywoływana jest według schematu dwukrotnie. Jest to konieczne, gdyż chcemy obsługiwać eventy także wtedy, kiedy jesteśmy w trybie pauzy. Nie możemy zabrać obsługi eventów z bloku wykonywanego warunkowo, ponieważ po każdym przeliczeniu położeń i parametrów obiektów, mogą zostać wygenerowane nowe zdarzenia, które trzeba obsłużyć. Blok może zostać wtedy ponownie wykonany, z powodu opóźnienia. W takiej sytuacji obsługa eventów poza pętlą odbyłaby się za późno.

Z tych względów musimy wywołać obsługę zarówno po każdym obiegu bloku warunkowego jak i po wyjściu z niego. Nie powoduje to żadnego spowolnienia. Nadmiarowa procedura natrafia w takim wypadku na pustą kolejkę eventów i natychmiast powraca.

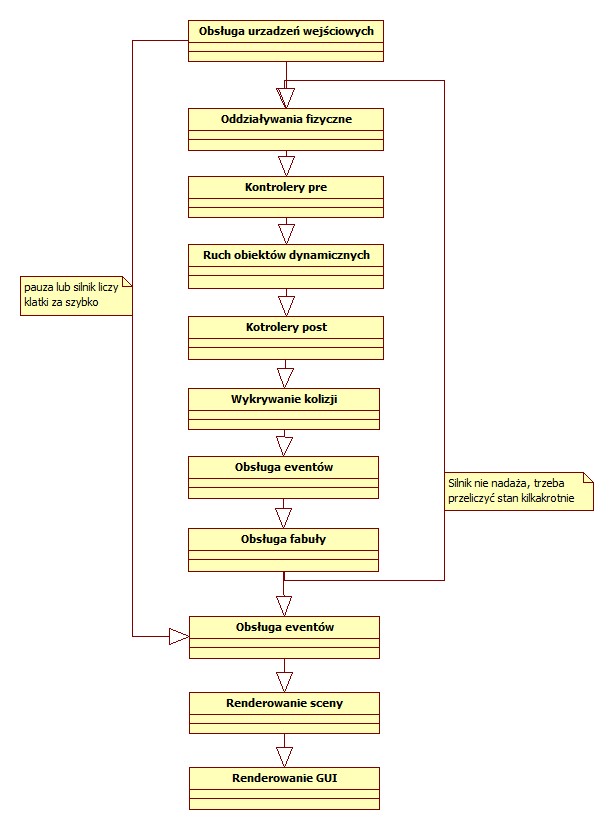
Jak widać ze schematu obsługa kontroleró podzielona jest na dwie części: kontrolery pre i kontrolery post. Dla niektórych grup kotrolerów wymagane jest, aby pozycje pozostałych obiektów dynamicznych był już znane, inne mają wpływ na późniejsze przeliczenie pozycji, dlatego muszą być wywołane wcześniej. Aby przeczytać więcej o tym temacie odsyłam do [kontrolery pre i post](#_Kontrolery).

Renderowanie klatki podzielone jest na dwa etapy: pierwszym jest renderowanie sceny, drugim renderowanie graficznego interfejsu użytkownika. Aby dowiedzieć się więcej zalecam przeczytać tematy: [Renderowanie sceny](#_Renderowanie_sceny) i [Renderowanie GUI](#_Renderowanie_GUI). Ze względu na to, że funkcje BeginScene i EndScene mogą być wywołane tylko raz podczas renderowania jednej klatki, dzieje się to na tym poziomie, nie wenątrz jednej z tych funkcji. Podobnie na tym poziomie realizowane jest czyszczenie bufora kolorów oraz z-bufora.

Funkcja [Engine](#_Engine)::[time\_controller](#_Engine::time_controller) wywoływana na początku robi dwie rzeczy. Po pierwsze liczy czas jaki upłynął od ostatniego wywołania i ustawia zmienną lag.Przy ustawianiu bierze pod uwagę czy jesteśmy w trybie pauzy czy nie, w związku z czym na tym poziomie nie musimy się o nic martwić.

Drugim zadaniem jest generowanie eventów związanych z czasem i opóźnieniami.

Schemat potoku przetwarzania obiektów:



### Renderowanie GUI

### Renderowanie sceny

### Układ współrzednych

Podobnie jak directX, silnik używa lewoskrętnego układu współrzędnych. Oś x skierowana jest w prawą stronę ekranu, oś y w górę ekranu, a oś z w kierunku ekranu.

Domyślne ustawienie kamery, która nie została przekształcona żadną macierza widoku jest takie, że z punktu ( 0, 0, 0 ) patrzy w głąb ekranu, czyli patrzy wzdłuż wektora ( 0, 0, 1 ). Kierunek do góry wyznaczony jest przez wektor ( 0, 1, 0 ).

## Podstawowe obiekty i struktury danych

### event\_mapping

#include "Engine\UI\_Engine\*InputAbstractionLayer.h*"

Definicje:

"Engine\UI\_Engine\*InputAbstractionLayer.h*"

struct event\_mapping

{

short virtual\_index;

char down\_event : 4;

char up\_event : 4;

char value;

};

Opis klasy:

Struktura przechowująca informacje o przyciskach, których wciśnięcie lub puszczenie ma powodować wysłanie eventu. Virtual\_index jest indeksem elementu w tablicy virtual\_button klasy [InputAbstractionLayer\_base](#_InputAbstractionLayer_base). Wartość pola value jest przechowywana jedynie roboczo w trakcie wykonania funkcji odpowiedzialnych za wypełnienie wartstwy abstrakcji danymi.

### InputAbstractionLayer

#include " Engine\UI\_Engine\*InputAbstractionLayer.h*"

Definicje:

Engine\ Engine\UI\_Engine\*InputAbstractionLayer*.cpp

Dziedziczenie:

[InputAbstractionLayer\_base](#_InputAbstractionLayer_base)

↓

[InputAbstractionLayer](#_ModelsManager)

class InputAbstractionLayer : public InputAbstractionLayer\_base

{

private:

bool active\_changed;

std::vector<input\_mapping> buttons\_mapping\_table;

std::vector<input\_mapping> axis\_mapping\_table;

std::vector<event\_mapping> event\_mapping\_table;

public:

InputAbstractionLayer();

~InputAbstractionLayer();

void inline set\_active(bool activate) { active = activate; if(active) active\_changed = true; };

void inline set\_mouse\_position(short X, short Y) { mouseX = X; mouseY = Y; };

void begin\_event\_collection();

void send\_events(Engine\*);

void update\_keyboard\_device( short device\_nr, char\* keyboard\_state );

void update\_mouse\_device( short device\_nr, DIMOUSESTATE2\* mouse\_state );

void update\_joystick\_device( short device\_nr, DIJOYSTATE\* joystick\_state );

void demand\_down\_event( short v\_index );

void demand\_up\_event( short v\_index );

void delete\_up\_event( short v\_index );

void delete\_down\_event( short v\_index );

void setup\_layer( input\_mapping\* mapping, unsigned int length );

int setup\_layer\_from\_file( const std::string& file\_name );

};

Opis klasy:

Klasa stanowi interfejs umożliwiający definiowanie i modyfikowanie wastwy abstrakcji. Zalecam przeczytanie opisu klasy [InputAbstractionLayer\_base](#_InputAbstractionLayer_base), który zawiera wszystkie potrzebne informacje.

Klasa bazowa stanowi interfejs uniemożliwiający nadpisanie danych w klasie i tylko w takim celu istnieje to rozdzielenie. Nie powinny istnieć żadne obiekty klasy [InputAbstractionLayer\_base](#_InputAbstractionLayer_base), ponieważ nie ma to żadnego sensu. Taki obiekt nie mógłby zostać w żaden sposób uzupełniony danymi.

Funkcje publiczne:

|  |  |
| --- | --- |
| [begin\_event\_collection](#_InputAbstractionLayer::begin_event_) | Przygotowuje aktualizację stanu przycisków i inicjuje zbieranie eventów. |
| [delete\_down\_event](#_InputAbstractionLayer::delete_down_) | Kasuje żądanie eventu przy wciśnięciu. |
| [delete\_up\_event](#_InputAbstractionLayer::delete_up_ev) | Kasuje żądanie eventu przy puszczeniu przycisku. |
| [demand\_down\_event](#_InputAbstractionLayer::demand_down_) | Żądanie wysłania eventu przy wciśnięciu przycisku. |
| [demand\_up\_event](#_InputAbstractionLayer::demand_up_ev) | Żądanie wysłania eventu przy puszczeniu przycisku. |
| [send\_events](#_InputAbstractionLayer::send_events) | Wysyła zebrane eventy o przyciskach. |
| [set\_active](#_InputAbstractionLayer::set_active) | Ustawia warstwę abstrakcji jako aktywną. |
| [set\_mouse\_position](#_InputAbstractionLayer::set_mouse_po) | Ustawia wewnętrzną pozycję myszy. |
| [setup\_axis\_layer](#_InputAbstractionLayer::setup_axis_l) | Tworzy tablicę mapowania osi. |
| [setup\_buttons\_layer](#_InputAbstractionLayer::setup_button) | Tworzy tablicę mapowania przycisków. |
| [setup\_layer\_from\_file](#_InputAbstractionLayer::setup_layer_) | Tworzy tablicę mapowania z zawartości pliku. |
| [update\_joystick\_device](#_InputAbstractionLayer::update_joyst) | Ustawia stan wejść o interfejsie joysticka. |
| [update\_keyboard\_device](#_InputAbstractionLayer::update_keybo) | Ustawia stan wejść o interfejsie klawiatury. |
| [update\_mouse\_device](#_InputAbstractionLayer::update_mouse) | Ustawia stan wejść o interfejsie myszy. |

Opis funkcji:

#### InputAbstractionLayer::begin\_event\_collection

Funkcja zapisuje wszystkie wartości przycisków, które potencjalnie mogą wysłać eventy. Następnie tablice są zerowane.

Po dostarczeniu klasie wszystkich informacji o urzadzeniach wejściowych, nastąpi sprawdzenie, gdzie zmieniły się stany przycisków i wysłanie eventów. Do tego należy wywołać funkcję end\_events.

Wszystkie wywołania funkcji ustawiających stan przycisków i osi powinny znajdować się pomiędzy wywołaniami begin\_event\_collection i send\_events. Inaczej nic nie zadziała, ponieważ funkcje zakładają, że tablice są wyzerowane.

Składnia:

void InputAbstractionLayer::begin\_event\_collection()

Parametry:

brak

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::delete\_down\_event

Składnia:

void InputAbstractionLayer::delete\_down\_event(

[in] unsigned short v\_index )

Parametry:

v\_index [in]

Typ: unsigned short

Indeks wirtualnego przycisku w tablicy, dla którego ma nastąpić modyfikacja eventu.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::delete\_up\_event

Składnia:

void InputAbstractionLayer::delete\_up\_event(

[in] unsigned short v\_index )

Parametry:

v\_index [in]

Typ: unsigned short

Indeks wirtualnego przycisku w tablicy, dla którego ma nastąpić modyfikacja eventu.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::demand\_down\_event

Składnia:

void InputAbstractionLayer::demand\_down\_event(

[in] unsigned short v\_index )

Parametry:

v\_index [in]

Typ: unsigned short

Indeks wirtualnego przycisku w tablicy, dla którego ma nastąpić modyfikacja eventu.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::demand\_up\_event

Składnia:

void InputAbstractionLayer::demand\_up\_event(

[in] unsigned short v\_index )

Parametry:

v\_index [in]

Typ: unsigned short

Indeks wirtualnego przycisku w tablicy, dla którego ma nastąpić modyfikacja eventu.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::send\_events

Składnia:

void InputAbstractionLayer::send\_events)

[in] Engine\* engine)

Parametry:

engine [in]

Typ: Engine\*

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::set\_active

Ustawia daną wartwę abstrakcji jako aktywną lub ją dezaktywizuje. należy pamiętać, że tylko jedna wartwa powinna byc w danym momencie aktywna i pilnowanie tego należy do klasy [UI\_Engine](#_UI_Engine).

Składnia:

void inline set\_active(

[in] bool activate)

Parametry:

activate [in]

Typ: bool

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::set\_mouse\_position

Składnia:

unsigned int Model3DFromFile::add\_material(

[in] const D3DMATERIAL9 &material)

Parametry:

material [in]

Typ:

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::setup\_axis\_layer

Ustawia mapowanie między osiami fizycznymi, a osiami wirtualnymi na podstawie podanej w parametrze tablicy mapowania.

Funkcja alokuje nową tablicę virtual\_axis klasy [InputAbstractionLayer\_base](#_InputAbstractionLayer_base).

Składnia:

void InputAbstractionLayer::setup\_axis\_layer(

[in] input\_mapping\* mapping,

[in] unsigned int length )

Parametry:

mapping[in]

Typ: input\_mapping\*

Wskaźnik na tablicę zawierającą struktury mapujące przyciski fizyczne na wirtualne

*length* [in]

Typ: *unsigned int*

Długość tablicy podanej w pierwszym parametrze.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::setup\_buttons\_layer

Ustawia mapowanie między przyciskami fizycznymi, a przyciskami wirtualnymi na podstawie podanej w parametrze tablicy mapowania.

Funkcja alokuje nową tablicę *virtual\_button* klasy [InputAbstractionLayer\_base](#_InputAbstractionLayer_base).

Składnia:

void InputAbstractionLayer::setup\_buttons\_layer(

[in] input\_mapping\* mapping,

[in] unsigned int length )

Parametry:

mapping[in]

Typ: input\_mapping\*

Wskaźnik na tablicę zawierającą struktury mapujące przyciski fizyczne na wirtualne

*length* [in]

Typ: *unsigned int*

Długość tablicy podanej w pierwszym parametrze.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::setup\_layer\_from\_file

Wczytuje ustawienia warstwy abstrakcji z pliku.

{do zaimplementowania}

Składnia:

int InputAbstractionLayer::setup\_layer\_from\_file(

[in] const std::string& file\_name )

Parametry:

file\_name [in]

Typ: const std::string&

Nazwa pliku, z którego zostaną pobrane dane o mapowaniu.

Wartość zwracana:

Typ: unsigned int

#### InputAbstractionLayer::update\_joystick\_device

Funkcja wywoływana w każdej klatce do ustawienia stanu wirtualnych przycisków oraz osi na podstawie wejścia o intefejsie joysticka. Zazwyczaj wywołuje się ją dla joysticka i wszystkich innych urządzeń, które nie sa klawiaturą albo myszą, ale może być też tak, że directX reprezentuje jakieś inne urządzenie takim samym interfejsem.

Składnia:

void InputAbstractionLayer::update\_joystick\_device(

[in] short device\_nr,

[in] const DIJOYSTATE\* joystick\_state )

Parametry:

*device\_nr* [in]

Typ: short

Numer urządzenia w tablicach mapping\_table klasy. Klawiatura zajmuje indeks 0, mysz indeks 1, pozostałe numery są przeznaczone dla innych urządzeń, które moga używać takich interfejsów. Dokładniejsze informacje w opisie klasy.

*mouse\_state*[in]

Typ: const DIJOYSTATE\*

Struktura zwracana przez DirectXa.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::update\_keyboard\_device

Funkcja wywoływana w każdej klatce do ustawienia stanu wirtualnych przycisków na podstawie wejścia o intefejsie klawiatury. Zazwyczaj wywołuje się ją dla klawiatury, ale może być też tak, że directX reprezentuje jakieś inne urządzenie takim samym interfejsem.

Składnia:

void InputAbstractionLayer::update\_keyboard\_device(

[in] short device\_nr,

[in] const char\* keyboard\_state )

Parametry:

*device\_nr* [in]

Typ: short

Numer urządzenia w tablicach mapping\_table klasy. Klawiatura zajmuje indeks 0, mysz indeks 1, pozostałe numery są przeznaczone dla innych urządzeń, które moga używać takich interfejsów. Dokładniejsze informacje w opisie klasy.

*mouse\_state*[in]

Typ: const char\*

Struktura zwracana przez DirectXa.

Wartość zwracana:

Typ: void

#### InputAbstractionLayer::update\_mouse\_device

Funkcja wywoływana w każdej klatce do ustawienia stanu wirtualnych przycisków oraz osi na podstawie wejścia o intefejsie myszy. Zazwyczaj wywołuje się ją dla myszy, ale może być też tak, że directX reprezentuje jakieś inne urządzenie takim samym interfejsem.

Składnia:

void InputAbstractionLayer::update\_mouse\_device(

[in] short device\_nr,

[in] const DIMOUSESTATE2\* mouse\_state )

Parametry:

*device\_nr* [in]

Typ: short

Numer urządzenia w tablicach mapping\_table klasy. Klawiatura zajmuje indeks 0, mysz indeks 1, pozostałe numery są przeznaczone dla innych urządzeń, które moga używać takich interfejsów. Dokładniejsze informacje w opisie klasy.

*mouse\_state*[in]

Typ: const DIMOUSESTATE2\*

Struktura zwracana przez DirectXa.

Wartość zwracana:

Typ: void

### InputAbstractionLayer\_base

#include " Engine\UI\_Engine\*InputAbstractionLayer.h*"

Definicje:

Engine\ Engine\UI\_Engine\*InputAbstractionLayer*.cpp

Dziedziczenie:

[InputAbstractionLayer\_base](#_InputAbstractionLayer_base)

class InputAbstractionLayer\_base

{

protected:

short mouseX;

short mouseY;

char\* virtual\_button;

float\* virtual\_axis;

short num\_buttons;

short num\_axis;

bool active;

public:

InputAbstractionLayer\_base();

~InputAbstractionLayer\_base();

inline bool is\_active() { return active; };

inline const char\* get\_buttons\_table() { return virtual\_button; };

inline const float\* get\_axis\_table() { return virtual\_axis; };

inline short get\_mouseX() { return mouseX; };

inline short get\_mouseY() { return mouseY; };

inline unsigned short get\_num\_axis() { return num\_axis; };

inline unsigned short get\_num\_buttons() { return num\_buttons; };

};

Opis klasy:

Klasa reprezentująca warstwę abstrakcji między wartościami zwracanymi z directInputa, a wejściem aplikacji. W grze może istnieć wiele wartstw abstrakcji, z których w danym momencie aktywna jest tylko jedna. Warstwa abstrakcji umożliwia stworzenie zestawu własnych przycisków oraz kontrolerów posiadających osie, do których następnie zostaną przypisane odpowiednie przyciski fizyczne. Wiele przycisków może jednocześnie być mapowanych na jeden przycisk wirtualny.

[InputAbstractionLayer\_base](#_InputAbstractionLayer_base) odpowiada za interakcję z [kontrolerami](#_Kontrolery) i zapewnia ograniczony dostęp do zasobów klasy, a co najważniejsze, nie pozwala na edycję danych.

[Klasa InputAbstractionLayer](#_InputAbstractionLayer) służy jako interfejs do komunikacji dla obiektu [UI\_Engine](#_UI_Engine). Umożliwia modyfikowanie zawartości tej klasy. Zostaje w każdej klatce uzupełniona danymi, które będą potem widziane przez [kontrolery](#_Kontrolery).

Do klasy bazowej nie należy dodawać żadnych funkcji, umożliwiających modyfikację danych o wejściu.

Użytkownik tworzący warstwę abstrakcji powinien stworzyć obiekt klasy pochodnej, ponieważ tylko on umożliwia zdefiniowanie takiej warstwy (czyli zdefiniowanie mapowania przycisków oraz podanie, które z nich mają wysyłać eventy przy wciśnięciu lub puszczeniu). Aby przeczytać więcej o wartwach abstrakcji: [Obsługa myszy, klawiatury](#_Obsługa_klawiatury,_myszy).

[Kontrolery](#_Kontrolery) dostają wskaźnik na warstwę abstrakcji, z którą mają współpracować. Aby odczytać dane, należy wywołać funkcje get\_buttons\_table oraz get\_axis\_table. Zwracają one wskaźnik na tablice, pierwsza z przyciskami, a druga z położeniem na osi.

Przyciski są opisywane zmiennymi typu char, gdzie wartość 1 oznacza, że przycisk jest wciśnięty, a 0, że nie jest. Kolejne przyciski znajdują się pod kolejnymi indeksami w tablicy. Zasadniczo nie można się dowiedzieć, jakie jest znaczenie poszczególnych pól w tablicy, ponieważ wartwa abstrakcji jest definiowana przez użytkownika, który następnie pisze dopasowany do tej wartwy kontroler. Wyjątek stanowią predefiniowane warstwy abstrakcji, które są opisane w tej dokumentacji.

{do zrobienia}

Położenie na osi opisuje wychylenie jakiegoś urządzenia, np. joysticka. Jego wartość jest typu float i zawiera się w przedziale [-1;1].

Uwaga!! Każdy kontroler przed pobraniem danych z tablic powinien się zorientować, czy warstwa abstrakcji, na której pracuje jest aktywna. Robi się to wywołując funkcję is\_active. Kontrolery powinny sterować przypisanymi im obiektami jednie wtedy, gdy ich wartwa jest aktywna. W przeciwnym razie będą odczytywać dane, które są od dawana nieaktualne, ponieważ aktualizowana jest tylko aktywna warstwa.

Ponadto kontrolery nie powinny być przypisywane do niekompatybilnych z nimi wartw abstrakcji. W szczególności wypadałoby przynajmniej sprawdzić czy zakres tablic wirtualnych jest większy, niż wymagany przez kontroler.

### input\_mapping

#include "Engine\UI\_Engine\*InputAbstractionLayer.h*"

Definicje:

"Engine\UI\_Engine\*InputAbstractionLayer.h*"

struct input\_mapping

{

short device\_nr;

short virtual\_index;

int button\_axis\_id;

};

Opis klasy:

Struktura opisująca, które przyciski fizyczne powinny zostać zmapowane na przyciski wirtualne.

* device\_nr - urządzeniem, które mapujemy
* virtual\_index - index w tablicy wirtualnych przycisków (osi)
* button\_axis\_id - identyfikator przycisku lub osi na urządzeniu fizycznym

Aby przeczytać więcej o temacie warstw abstrakcji urządzeń wejścia polecam temat [Obsługa myszy, klawiatury itp](#_Obsługa_klawiatury,_myszy). oraz opisy klas [InputAbstractionLayer\_base](#_InputAbstractionLayer_base) lub[InputAbstractionLayer](#_InputAbstractionLayer_base).

### MaterialObject

### MeshObject

### MeshPart

### Model3DFromFile

#include "Engine\ModelsManager\meshes\_textures\_materials.h"

Definicje:

Engine\ModelsManager\*Model3DFormFile*.cpp

Dziedziczenie:

[Model3DFromFile](#_ModelsManager)

class Model3DFromFile

{

friend class ModelsManager;

private:

static ModelsManager\* models\_manager;

unsigned int references; //liczba obiektów, które się odwołują

unsigned int unique\_id; //unikalny identyfikator pliku (mozna się odwoływać po nazwie pliku lub po tym id)

std::string file\_path;

//tekstura i materiał odpowiadają meshowi spod danego indeksu

std::vector<MeshPart\*> mesh\_parts;

std::vector<TextureObject\*> textures;

std::vector<MaterialObject\*> materials;

public:

Model3DFromFile(int id);

~Model3DFromFile();

unsigned int add\_material(const D3DMATERIAL9 &material);

unsigned int add\_texture(const std::string& path);

unsigned int add\_mesh(Vertex\_Normal\_TexCords1\* vertices, unsigned int vert\_num, const DirectX::XMFLOAT4X4& matrix);

MaterialObject\* add\_material(unsigned int id);

TextureObject\* add\_texture(unsigned int id);

MeshPart\* add\_mesh(unsigned int id, DirectX::XMFLOAT4X4& matrix);

void add\_null\_material();

void add\_null\_texture();

unsigned int get\_parts\_count();

MaterialObject\* get\_material(unsigned int index);

TextureObject\* get\_texture(unsigned int index);

MeshPart\* get\_mesh\_part(unsigned int index);

//zarządzanie odwołaniami

bool can\_delete();

void add\_reference();

void delete\_reference();

};

Opis klasy:

Klasa służy do przechowywania zawartości pliku z modelem. W momencie wczytywania modelu, odpowiedni [Loader](#_Loader) dostaje wskaźnik na ten obiekt i wypełnia go danymi. Do pliku można się odwoływać za pomocą unikalnego identyfikatora pliku, przechowywanego w zmiennej unique\_id.

Klasa przechowuje w zmiennej references liczbę obiektów, które odwołują się do zapisanego modelu. W momencie przypisania modelu do obiektu [Dynamic\_mesh\_object](#_Model_object), funkcje, które to robią, maja obowiązek pamiętać o modyfikowaniu odpowiednich referencji za pomocą funkcji [add\_reference](#_Model3DFromFile::add_reference) i [delete\_reference](#_Model3DFromFile::delete_reference).

Struktura modelu:

Dane modelu przechowywane są w tablicach mesh\_parts, materials i textures. Ponieważ do meshy może być przypisanych wiele materiałów i tekstur, to nie ma możliwości przechowywania całych meshy, bo nie moglibyśmy potem przypisać materiałów do trójkątów. Z tego względu meshe przechowywane są w kawałkach (obiekty [MeshPart](#_MeshObject)). Obiektowi [MeshPart](#_MeshObject) znajdującemu się na n-tej pozycji tablicy mesh\_parts, odpowiadają znajdujący się pod tym samym indeksem materiał i tekstura.

Nie każdy [MeshPart](#_MeshObject) musi mieć przypisaną teksturę i materiał. Jednego z tych elementów może brakować i w takim wypadku należy odpowiednie miejsce w tablicy wypełnić nullptrem.

Materiały są przechowywane w klasach [MaterialObject](#_MaterialObject), a tekstury w klasach [TextureObject](#_TextureObject_1). Za zachowanie niepowtarzalności tych obiektów odpowiada klasa [ModelsManager](#_ModelsManager_1), dlatego nie wolno tworzyć tych obiektów samodzielnie, lecz należy to robić za pomocą funkcji Model3DFromFile, które są do tego przeznaczone.

Poszczególne części mesha przechowywane są w obiektach [MeshPart](#_MeshObject). Klasa ta składa się ze wskaźnika na klasę [MeshObject](#_TextureObject), zawierającej bufor wierzchołków, oraz macierzy przekształcenia, które należy zastosować na tych wierzchołkach. Taka struktura pozwala na wielokrotne użycie tych samych wierzchołków w różnych miejscach modelu, w różnej orientacji, a także w różnej skali.

Za niepowtarzalność i zwalnianie pamięci po obiektach [MeshObject](#_TextureObject) odpowiada klasa [ModelsManager](#_ModelsManager_1), a za obiekty [MeshPart](#_MeshObject) odpowiada klasa Model3DFromFile.

Konstruktory i destruktor:

|  |  |
| --- | --- |
| Model3DFromFile(int id) | Inicjuje obiekt unikalnym identyfikatorem podanym w parametrze. |
| ~Model3DFromFile() |  |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**add\_material**](#_Model3DFromFile::add_material)**(const D3DMATERIAL9&)** | Dodaje materiał na koniec tablicy |
| [**add\_material(unsigned int id)**](#_Model3DFromFile::add_material(unsig) | Dodaje materiał na koniec tablicy |
| [**add\_mesh**](#_Model3DFromFile::add_mesh)**(unsigned int, DirectX::XMFLOAT4X4&)** | Dodaje mesh na koniec tablicy |
| [**add\_mesh**](#_Model3DFromFile::add_mesh)**(Vertex\_Normal\_TexCords1\*, unsigned int,**  **const DirectX::XMFLOAT4X4&)** | Dodaje mesh na koniec tablicy |
| [**add\_null\_material**](#_Model3DFromFile::add_null_material) | Wstawia pusty materiał |
| [**add\_null\_texture**](#_Model3DFromFile::add_null_texture) | Wstawia pusta teksturę |
| [**add\_reference**](#_Model3DFromFile::add_reference) | Dodaje odwołanie obiektu do tego pliku |
| [**add\_texture**](#_Model3DFromFile::add_texture)**(const std::string&)** | Dodaje teksturę na koniec tablicy |
| [**add\_texture**](#_Model3DFromFile::add_texture_1)**(unsigned int id)** | Dodaje teksturę na koniec tablicy |
| [**can\_delete**](#_Model3DFromFile::can_delete) | Sprawdza czy nie ma odwołań |
| [**delete\_reference**](#_Model3DFromFile::delete_reference) | Kasuje odwołanie obiektu do tego pliku |
| [**get\_material**](#_Model3DFromFile::get_material) | Pobiera materiał na n-tej pozycji wektora |
| [**get\_mesh\_part**](#_Model3DFromFile::get_mesh_part) | Pobiera mesha z n–tej pozycji wektora |
| [**get\_parts\_count**](#_Model3DFromFile::get_parts_count) | Pobiera ilość pozycji w wektorach |
| [**get\_texture**](#_Model3DFromFile::get_texture) | Pobiera texturę z n–tej pozycji wektora |

Opis funkcji:

#### Model3DFromFile::add\_material(const D3DMATERIAL9&)

Dodaje nowy materiał na koniec wektora materiałów na podstawie podanej w parametrze struktury. Jeżeli materiał nie istniał, zostaje dodany do [ModelsManagera](#_ModelsManager_1).

Składnia:

unsigned int Model3DFromFile::add\_material(

[in] const D3DMATERIAL9 &material)

Parametry:

material [in]

Typ: const D3DMATERIAL9

Struktura opisująca materiał.

Wartość zwracana:

Typ: unsigned int

Funkcja zwraca unikalny indeks materiału.

#### Model3DFromFile::add\_material(unsigned int)

Dodaje nowy materiał na koniec wektora materiałów na podstawie podanego w parametrze indeksu materiału

Składnia:

MaterialObject\* Model3DFromFile::add\_material(

[in] unsigned int id)

Parametry:

id [in]

Typ: unsigned int

Identyfikator materiału.

Wartość zwracana:

Typ: [MaterialObject](#_MaterialObject)\*

Wskaźnik na obiekt materiału, który został dodany do listy.

#### Model3DFromFile::add\_mesh(unsigned int, DirectX::XMFLOAT4X4&)

Dodaje na koniec tablicy nowego mesha o podanym identyfikatorze i przypisuje mu macierz przekształcenia.

Składnia:

MeshPart\* add\_mesh(

[in] unsigned int id,

[in] DirectX::XMFLOAT4X4& matrix);

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_mesh(Vertex\_Normal\_TexCords1\*, unsigned int, const DirectX::XMFLOAT4X4&)

Dodaje na koniec tablicy mesha podanego w tablicy wierzchołków. Jeżeli mesh jeszcze nie istniał, zostaje dodany do [ModelsManagera](#_ModelsManager_1).

Składnia:

unsigned int add\_mesh(

[in] Vertex\_Normal\_TexCords1\* vertices,

[in] unsigned int vert\_num,

[in] const DirectX::XMFLOAT4X4& matrix);

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_null\_material

Dodaje na koniec tablicy materiałów wskaźnik null.

Składnia:

void add\_null\_material();

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_null\_texture

Dodaje na koniec tablicy tekstur wskaźnik null.

Składnia:

void add\_null\_texture();

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_reference

Inkrementuje wskaźnik referencji do danego modelu.

Składnia:

void add\_reference();

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_texture(const std::string&)

Dodaje teksturę na koniec tablicy tekstur. Jeżeli tekstura wczesniej nie istniała zostaje dodana do [ModelsManagera](#_ModelsManager_1).

Składnia:

unsigned int add\_texture(

[in] const std::string& path);

Parametry:

Wartość zwracana:

#### Model3DFromFile::add\_texture(unsigned int id)

Dodaje teksturę na koniec tablicy tekstur na podstawie identyfiaktora tekstury.

Składnia:

TextureObject\* add\_texture(

[in] unsigned int id);

Parametry:

Wartość zwracana:

#### Model3DFromFile::can\_delete

Sprawdza czy są jakieś odwołania do podanego pliku. Jeżeli nie ma to znaczy, że obiekt może

zostać bezpiecznie usunięty.

Składnia:

bool can\_delete(

[out] unsigned int& ref);

Parametry:

Wartość zwracana:

#### Model3DFromFile::delete\_reference

Dekrementuje wskaźnik modeli do danego modelu.

Składnia:

void delete\_reference();

Parametry:

Wartość zwracana:

#### Model3DFromFile::get\_material

Pobiera materiał z n-tej pozycji wektora.

Składnia:

MaterialObject\* get\_material(

[in] unsigned int index);

Parametry:

index [in]

Typ: unsigned int

Index materiału w tablicy materiałów. Nie mylić z identyfikatorem materiału.

Wartość zwracana:

Typ: MaterialObject\*

Zwraca obiekt zawierający materiał lub nullptr, jeżeli index wykracza poza tablicę.

#### Model3DFromFile::get\_mesh\_part

Pobiera [MeshPart](#_MeshObject) z n-tej pozycji wektora.

Składnia:

MeshPart\* get\_mesh\_part(

[in] unsigned int index);

Parametry:

index [in]

Typ: unsigned int

Index mesha w tablicy meshy. Nie mylić z identyfikatorem mesha.

Wartość zwracana:

Typ: MeshPart\*

Zwraca obiekt zawierający mesh lub nullptr, jeżeli index wykracza poza tablicę.

#### Model3DFromFile::get\_parts\_count

Pobiera ilość elementów w wektorach zawierających materiały textury i meshe. W prawidłowym przypadku wielkości te są takie same. Jeżeli z jakiegoś powodu tablice zostały zbudowane nieprawidłowo i nie zostało to usunięte do tej pory, to funkcja zwróci najmniejszy rozmiar tablicy.

Składnia:

unsigned int get\_parts\_count();

Parametry:

Brak

Wartość zwracana:

Typ: unsigned int

Liczba elementów w tablicach.

#### Model3DFromFile::get\_texture

Pobiera wskaźnik na obiekt zawierający teksturę z pod podanego indeksu.

Składnia:

TextureObject\* get\_texture(

[in] unsigned int index);

Parametry:

index [in]

Typ: unsigned int

Index tekstury w tablicy tekstur. Nie mylić z identyfikatorem tekstury.

Wartość zwracana:

Typ: TextureObject\*

Wskaźnik na obiekt zawierający teksturę.

### referenced\_object

#include "Engine\ModelsManager\meshes\_textures\_materials.h"

Definicje:

Engine\ModelsManager\*Model3DFormFile*.cpp

Dziedziczenie:

[referenced\_object](#_ModelsManager)

class referenced\_object

{//definicja w pliku Model3DFormFile

friend ModelsManager;

private:

unsigned int file\_references; //liczba plików, które sie odwołują

unsigned int object\_references; //liczba modeli, które się odwołują

unsigned int unique\_id; //unikalny identyfikator materiału

public:

referenced\_object(int id);

virtual ~referenced\_object();

//sprawdza czy można zwolnić zmienną

bool can\_delete(unsigned int& file\_ref, unsigned int& other\_ref);

//zarządzanie odwołaniami

void inline add\_file\_reference();

void inline add\_object\_reference();

void inline delete\_file\_reference();

void inline delete\_object\_reference();

};

Opis klasy:

Klasa bazowa dla obiektów, do których istnieją wielokrotne odwołania (np. obiektów [MaterialObject](#_MaterialObject), [MeshObject](#_TextureObject) i [TextureObject](#_TextureObject_1)). Odwołania dzielimy na odwołania plikowe w zmiennej file\_references oraz odwołania obiektów – zmienna object\_references. Więcej o mechanizmie odwołań można przeczytać w dalszym opisie.

Klasa posiada też zmienną unique\_id, która przechowuje unikalny identyfikator wewnątrz danej klasy. Czyli np. wewnątrz klasy [MaterialObject](#_MaterialObject) identyfikator pozostaje uniklany, ale ta sama liczba może wystąpić w którejś z klas [MeshObject](#_TextureObject) lub [TextureObject](#_TextureObject_1).

Mechanizm odwołań:

Aby jak najoszczędniej korzystać z pamięci operacyjnej, chcemy, aby tekstury, materiały i meshe nie były duplikowane. Za zapewnienie, że żadne z tych obiektów nie zostaną wczytane dwukrotnie, odpowiada klasa [ModelsManager](#_ModelsManager). Takie założenie powoduje, że do jednego obiektu możemy się odwoływać wielokrotnie, co utrudnia zarządzanie tymi obiektami. Z pomoca przychodzi mechanizm odwołań i zmienne file\_references oraz object\_references.

Pierwszy typ odwołania, to odwołanie plikowe. Wczytując plik, tworzymy obiekt klasy [Model3DFromFile](#_Model3DFromFile), który reprezentuje zawartość pliku z modelem 3D. Obiekt ten przechowuje tylko wskaźniki, faktyczne instancje są przechowywane w tablicach [ModelsManager](#_ModelsManager)a (to rozróżnienie istnieje oczywiście tylko na poziomie logicznym, fizycznie [ModelsManager](#_ModelsManager) posiada też tylko wskaźniki). Jeden plik może się wielokrotnie odwoływać do tego samego obiektu (zachęcam do poczytania o strukturze [Model3DFromFile](#_Model3DFromFile)), jak również może istnieć wiele plików korzystających z tego samego obiektu. Każde odwołanie do [referenced\_object](#_referenced_object) przez [Model3DFromFile](#_Model3DFromFile) inkrementuje wartość file\_references. (Zwracam uwagę, że nie zliczamy tego, ile klas się odwołuje, tylko zliczamy ile razy klasy się odwołują. Każde odwołanie inkrementuje wartość o jeden. Taka struktura ułatwia kasowanie modeli i ewentualną modyfikację – napotykając wskaźnik możemy wywołać na nim funkcję [delete\_file\_reference](#_referenced_object::delete_file_refe) i nie sprawdzamy czy w innym miejscu tego pliku jakieś inne odwołania jeszcze istnieją.)

Drugi typ odwołania, to odwołanie bezpośrednie obiektu, który zamierza dany mesh, teksturę czy materiał wyświetlać. Dotyczy to obiektów dziedziczących po [Dynamic\_mesh\_object](#_Model_object), reprezentujących instancję obiektu, posiadającego własnego mesha. Takiemu obiektowi możemy przypisać mesha wczytanego z jakiegoś pliku. Operacja ta oznacza:

* przepisanie danych z tablicy w klasie [Model3DFromFile](#_Model3DFromFile) do własnych tablic wewnętrznych [Dynamic\_mesh\_object](#_Model_object)
* zapisanie wskaźnika na [Model3DFromFile](#_Model3DFromFile) w klasie [Dynamic\_mesh\_object](#_Model_object)
* zinkrementowanie referencji do klasy [Model3DFromFile](#_Model3DFromFile)
* zinkrementowanie referencji do każdego obiektu z tablic [Model3DFromFile](#_Model3DFromFile)

Głównym powodem dla którego chcemy zapisywać referencje obiektów oddzielnie od referencji plikowych jest to, że moglibyśmy chcieć w trakcie gry podmieniać tekstury i materiały jednej instancji obiektu niezależnie od wszystkich pozostałych obiektów posiadających ten sam model. Jednocześnie chcemy zachować pierwotną, niezmodyfikowaną zawartość [Model3DFromFile](#_Model3DFromFile). Każde odwołanie do referenced\_object wewnątrz klasy [Dynamic\_mesh\_object](#_Model_object) wymaga inkrementowania zmiennej object\_references, a usuniecie odwołania dekrementacji tej zmiennej. Oczywiście modyfikacja tekstur i materiałów musi być obudowana takimi funkcjami, żeby użytkownik silnika nie musiał się przejmować odwołaniami.

Podkreślam wiec jeszcze raz, że przypisując obiektowy [Dynamic\_mesh\_object](#_Model_object) jakiś plik z modelem, modyfikujemy także odwołania bezpośrednie. Jeżeli plik odwoływał się do obiektu referenced\_object n razy, to tyle razy musimy inkrementować object\_references.

Jak napisałem wyżej, każdy [Dynamic\_mesh\_object](#_Model_object) posiada wskaźnik na model [Model3DFromFile](#_Model3DFromFile), który mu przypisano. Wskaźnik ten nie ulega zmianie, jeżeli zmieniamy jakieś pojedyncze tekstury czy materiały. Modyfikujemy go jedynie, jeżeli przypisaliśmy do obiektu zupełnie nowy plik lub kasujemy [Dynamic\_mesh\_object](#_Model_object). W takim przypadku pamiętamy o kasowaniu referencji do [Model3DFromFile](#_Model3DFromFile), oraz referencji bezposrednich.

Powodem dla którego trzymamy tę referencję, mimo że wyświetlany model nie odpowiada już temu z pliku jest to, że odwołujemy się do obiektów, za których zwalnianie odpowiada [Model3DFromFile](#_Model3DFromFile). O ile za tekstury i materiały odpowiada [ModelsManager](#_ModelsManager), to z meshami sprawa jest troszkę bardziej skomplikowana.

Jeden model może używać danego mesha wielokrotnie, ale w inaczej zorientowanego w przestrzeni. Dlatego tablice wierzchołków zarządzane są przez [ModelsManager](#_ModelsManager), a [Model3DFromFile](#_Model3DFromFile) zawiera obiekty typu [MeshPart](#_MeshPart), które oprócz wskaźnika na tablice wierzchołków, zawierają macierz przekształcenia.

Oznacza to, że nie możemy podmienić tylko jakiejś części mesha w klasie [Dynamic\_mesh\_object](#_Model_object) , możemy go tylko wymienić w całości.

Do modyfikowania odwołań zaleca się stosować funkcje [add\_file\_reference](#_referenced_object::add_file_referen), [add\_object\_reference](#_referenced_object::add_object_refer), [delete\_file\_reference](#_referenced_object::delete_file_refe) i [delete\_object\_reference](#_referenced_object::delete_object_re).

Konstruktory i destruktor:

|  |  |
| --- | --- |
| referenced\_object(int id) | Inicjuje obiekt unikalnym identyfikatorem podanym w parametrze. |
| virtual ~referenced\_object() |  |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**can\_delete**](#_referenced_object::can_delete) | Sprawdza czy nie ma odwołań do obiektu |
| [**add\_file\_reference**](#_referenced_object::add_file_referen) | Inkrementuje referencje plikowe |
| [**add\_object\_reference**](#_referenced_object::add_object_refer) | Inkrementuje referencje bezpośrednie |
| [**delete\_file\_reference**](#_referenced_object::delete_file_refe) | Dekrementuje referencje plikowe |
| [**delete\_object\_reference**](#_referenced_object::delete_object_re) | Dekrementuje referencje bezpośrednie |

Opis funkcji:

#### referenced\_object::can\_delete

Składnia:

Parametry:

Wartość zwracana:

#### referenced\_object::add\_file\_reference

Składnia:

Parametry:

Wartość zwracana:

#### referenced\_object::add\_object\_reference

Składnia:

Parametry:

Wartość zwracana:

#### referenced\_object::delete\_file\_reference

Składnia:

Parametry:

Wartość zwracana:

#### referenced\_object::delete\_object\_reference

Składnia:

Parametry:

Wartość zwracana:

### TextureObject

### Vertex\_Normal\_TexCords1

### Vertex\_Normal\_TexCords2

### Vertex\_TexCords1

### VertexUI

## Klasy predefiniowane

## Klasy bazowe dla obiektów predefiniowanych

### Animation\_object

### Base\_AI\_controller

### Base\_input\_controller

### Camera\_object

### Collision\_object

### Complex\_object

#include "Interfaces\basic\_interfaces.h"

Definicje:

"Interfaces\basic\_interfaces.cpp”

Dziedziczenie:

{jeszcze nie wiadomo}

Opis klasy:

{Klasa do napisania}

Klasa pozwala na tworzenie obiektów złożonych, które możemy dowolnie dodawać lub kasować. Pozycja obiektów podrzędnych obliczana jest na podstawie położenia obiektu Complex\_object i pewnego offsetu. Składnikiem klasy nie powinny być obiekty dynamiczne (w tym w szczegolności takie, którym przypisane są jakieś kontrolery). Ponieważ pozycja obliczana jest na podstawie obiektu nadrzędnego, to położenie obiektu dziecka musi być stałe względem rodzica. Przypisanie obiektu dynamicznego spowoduje, że nie będzie się on poruszał zgodnie z oczekiwaniami.

Obiekty złożone istnieją po to, aby umożliwić związanie z obiektem takich rzeczy jak kamery, triggery lub inne obiekty funkcyjne. (Innym sposobem realizowania takich rozwiazań jest zastosowanie odpowiednich [kontrolerów](#_Kontrolery)). Klasa Complex\_object nie jest po to by realizować złożone hierarchie obiektów.

### Controller

### Dynamic\_mesh\_object

#include "Interfaces\basic\_interfaces.h"

Definicje:

"Interfaces\basic\_interfaces.cpp”

Dziedziczenie:

[Object](#_Object)

↓

[Static\_object](#_Static_object)

↓

[Collision\_object](#_Collision_object)

↓

[Dynamic\_object](#_Dynamic_object)

↓

[Physical\_object](#_Physical_object)

↓

[Dynamic\_mesh\_object](#_ModelsManager)

class Dynamic\_mesh\_object : public Physical\_object

{

friend class DisplayEngine;

protected:

Model3DFromFile\* model\_reference;

std::vector<MeshPart\*> mesh\_parts;

std::vector<TextureObject\*> textures;

std::vector<MaterialObject\*> materials;

public:

Dynamic\_mesh\_object();

virtual ~Dynamic\_mesh\_object();

int set\_model( Model3DFromFile\* model );

};

Opis klasy:

Klasa bazowa dla wszystkich obiektów posiadających mesha. Struktura danych zapisanych w tablicach mesh\_parts, textures i materials jest taka sama jak analogiczne struktury w klasie [Model3DFromFile](#_Model3DFromFile).

Przechowujemy obiekt modelu, do którego się odwołujemy w zmiennej model\_reference. Więcej o mechaniźmie odwołań znajduje się tutaj: [referenced\_object](#_referenced_object).

Konstruktory i destruktor:

|  |  |
| --- | --- |
| Dynamic\_mesh\_object(); |  |
| virtual ~Dynamic\_mesh\_object(); | Kasuje referencje do przechowywanych obiektów |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**set\_model**](#_Dynamic_mesh_object::set_model) | Przypisuje obiektowy plik z modelem |

Opis funkcji:

#### Dynamic\_mesh\_object::set\_model

Przypisuje do obiektu nowy plik z modelem. Funkcja modyfikuje wewnątrz referencje plikowe i bezpośrednie, nie należy ich modyfikować samemu.

Składnia:

int set\_model(

[in] Model3DFromFile\* model );

Parametry:

model [in]

Typ: [Model3DFormFile](#_Model3DFromFile)\*

Struktura opisująca materiał.

Wartość zwracana:

Typ: int

Funkcja zwraca 0, jeżeli wszystko poszło dobrze. Inaczej zwraca inną wartość. Jedyny możliwy błąd to podanie wskaźnika nullptr.

### Dynamic\_object

### Event

### Instanced\_mesh

### Object

#include "Interfaces\basic\_interfaces.h"

Definicje:

”Interfaces\basic\_interfaces.cp”

Dziedziczenie:

[Object](#_ModelsManager)

class Object

{

friend class Engine;

private:

static Engine\* engine;

protected:

short class\_type;

short object\_type;

void inline event(Event\*);

inline engine\_interface\* get\_interface(){ return static\_cast<engine\_interface\*>(engine); }

public:

virtual ~Object() {};

virtual void init( engine\_interface\* engine ){};

};

Opis klasy:

Klasa bazowa dla większości obiektów w silniku w szczególności tych, które może zdefiniować użytkownik. {do\_zrobienia} Wszystkie obiekty dziedziczące po Object są alokowane z przedefiniowanym operatorem new i delete w celu polepszenia zarządzania pamięcią.

#### Funkcja event

Klasa posiada funkcję event, która wstawia do kolejki komunikat (zobacz: [eventy](#_Eventy)). Umożliwia to definiowanie przez użytkowników własnych eventów i wysyłanie ich w zdefiniowanych przez użytkownika funkcjach. Używając funkcji event należy uważać, żeby nie doprowadzić do zapętlenia. Wywołanie jej w obsłudze tego samego eventu spowoduje natychmiastowe wstawienie do kolejki i obsługę w tym samym obiegu pętli. Powoduje to nieskończoną pętlę.

Aby uniknąć wywoływania funkcji event z dowolnego miejsca kodu, jest ona zadeklarowana jako protected, żeby tylko metoda klasy mogła się do niej odwołać. Poprawne wywołanie funkcji event może nastąpić we wszystkich funkcjach oprócz tych, które są odpowiedzialne za obsługę komunikatów. (Oczywiście użytkownik może to obejść pisząc metode, która wyśle event. Przed tym się nie zabezpieczymy, dlatego trzeba w dokumentacji wszedzie użytkowników uświadamiac o zagrożeniach.)

#### Funkcja init

Klasa bazowa deklaruje wirtualną funkcję init, która zostanie wywołana przez silnik w momencie dodania obiektu. Czasami dodanie obiektu może wymagać użycia wielu funkcji typu: przypisanie modelu, dodanie do odpowiednich modułów, zdefiniowanie położeń itp. Jeszcze wiecej pracy potrzeba, gdy mamy doczynienia z obiektem złożonym, ponieważ wszystkie podobiekty też powinny zostać dodane w odpowiednie miejsca i każdy wymaga indywidualnego potraktowania, bo może być zupełnie innego typu.

Ułatwieniem jest funkcja init, którą można zaimplementować, żeby wykonywała te czynności automatycznie po dodaniu do silnika. Ułatwia to sytuację, w której mamy wiele takich samych obiektów.

Przy implementacji należy użyć funkcji get\_interface, która zwróci nam interfejs silnika.

W klasach bazowych dla innych obiektów funkcja init jest zdefiniowana jako pusty kod. W klasach predefiniowanych może ona być zaimplementowana, więc jeżeli dziedziczymy po takiej klasie, powinniśmy się zastanowić czy przedefiniowanie tej funkcji jest konieczne. Zalecana praktyka jest taka, żeby przedefiniowywać tylko dodawane przez nas funkcjonalności, a następnie wywoływać funkcję init dla klasy stojącej w hierarchii dziedziczenia dokładnie ponad naszą klasą. (Robi się to używając operatora :: z nazwą klasy bazowej przed nim).

### Physical\_object

### Static\_object

### Trigger\_object

## Interfejsy

### engine\_interface

#include "EngineInterface.h"

Definicje: "Engine\EngineInterface.cpp "

Dziedziczenie:

engine\_interface

class engine\_interface

{

friend class ControllersEngine;

friend class ModelsManager;

friend class PhysicEngine;

friend class FableEngine;

friend class MovementEngine;

friend class DisplayEngine;

friend class CollisionEngine;

friend class UI\_Engine;

friend class Object;

protected:

//directX and windows variables

LPDIRECT3D9 directX\_interface; // the pointer to our Direct3D interface

LPDIRECT3DDEVICE9 directX\_device; // the pointer to the device class

bool directX\_ready; // jeżeli zmienna jest niepoprawna, nie renderujemy

bool full\_screen;

HWND window\_handler;

HINSTANCE instance\_handler;

int window\_width;

int window\_height;

//Engines

ControllersEngine\* controllers\_engine; //kontroluje AI (klawiaturę trzeba przenieść)

MovementEngine\* movement\_engine; //przelicza pozycję obiektów w następnej klatce

DisplayEngine\* display\_engine; //wyświetla obiekty na scenie

CollisionEngine\* collision\_engine; //liczy kolizje

PhysicEngine\* physic\_engine; //liczy oddziaływania fizyczne

ModelsManager\* models\_manager; //zarządza modelami, teksturami i materiałami

FableEngine\* fable\_engine; //zarządza fabułą gry, interakcjami obiektów itd.

UI\_Engine\* ui\_engine; //interfejs użytkownika (także graficzny)

//Objects

std::vector<IndexPrediction> index\_predictor;

//time control

DWORD time\_previous; //będziemy mierzyć różnicę miedzy czasami w kolejnych

DWORD elapsed\_time; //do licznika FPSów, czas od ostatniej aktualizacji frames\_per\_sec

float lag; //opóźnienie względem normalego tempa updatowania sceny

unsigned int frames;

float frames\_per\_sec; //najważniejszy licznik w grze

//queue

std::queue<Event\*>\* events\_queue; //wskaźnik na kolejke komunikatów w celu szybszego dostępu

protected:

~engine\_interface(){};

public:

};

Opis klasy:

Taki interfejs będzie widział użytkownik silnika z wnętrza swoich klas odpowiedzialnych za fabułę. Użytkownik musi mieć ograniczone możliwości, żeby przypadkiem nie namieszał. Szczególnie nie powinien mieć możliwości wysyłania eventów, bo mogłoby to doprowadzić do zapętlenia. Wszystkie funkcje jakie powinien mieć dostępne należy deklarować w tej klasie w sekcji publicznej. Wszystkie pozostałe funkcje powinny się znaleźć w deklaracji klasy [Engine](#_Engine).

Wszystkie zmienne silnika powinny być deklarowane na tym poziomie, aby również metody tej klasy dostępne dla użytkownika miały do nich dostęp.

Konstruktory i destruktor:

|  |  |
| --- | --- |
| engine\_interface() | Konstruktor nic nie robi. |
| protected ~engine\_interface() | Obiekt może zostać skasowany tylko przez wskaźnik na klasę pochodną. |

Funkcje prywatne:

|  |  |
| --- | --- |
|  |  |
|  |  |

Funkcje publiczne:

|  |  |
| --- | --- |
|  |  |
|  |  |

Opis funkcji:

### GamePlay

### Loader

#include "Engine\ModelsManager\Loaders\loader\_interface.h"

Definicje:

Dziedziczenie:

[Loader](#_ModelsManager)

class Loader

{

protected:

ModelsManager\* models\_manager;

public:

Loader(ModelsManager\* models\_manager) : models\_manager(models\_manager){};

virtual ~Loader(){};

virtual bool can\_load(const std::string& name) = 0;

virtual int load\_mesh(Model3DFromFile\* new\_file\_mesh, const std::string& name) = 0;

//virtual void load\_animation(const std::string& name) = 0;

};

Opis klasy:

Konstruktory i destruktor:

|  |  |
| --- | --- |
| Loader(ModelsManager\* ) | Zapisuje wskaźnik na obiekt ModelManagera, do którego należy. |
| virtual ~Loader() | Konstruktor wirtualny. Trzeba pamiętać, żeby go zdefiniować w klasie pochodnej. |

Funkcje publiczne:

|  |  |
| --- | --- |
| virtual [**bool can\_load**](#_Loader::can_load) | Zwraca informacje czy może wczytać plik. |
| virtual [**int load\_mesh**](#_Loader::load_mesh) | Wczytuje podany mesh. |

Opis funkcji:

#### Loader::can\_load

Funkcja sprawdza czy dany Loader potrafi otworzyć plik podany w parametrze. Sprawdzenie może opierać się wyłącznie na rozszerzeniu pliku lub może go otwierać i sprawdzać nagłówek. Kwestia ta zależy od implementacji konkretnego leadera, należy jednak uważać, żeby nie robić w tej funkcji za dużo, gdyż podczas ładowania pliku, [ModelsManager](#_ModelsManager_1) odpytuje po kolei Loadery ze swojej listy i zbyt rozbudowana funkcja może znacznie spowolnić ten proces.

Składnia:

virtual bool can\_load(

[in] const std::string& name) = 0;

Parametry:

name [in]

Typ: const std::string&

Ścieżka do pliku, którego testujemy pod względem otwieralności.

Wartość zwracana:

Typ: bool

Zwraca wartość true, jeżeli plik nadaje się do otwarcia, w przeciwnym razie zwraca false.

#### Loader::load\_mesh

Funkcja służy do wczytywania pliku do [ModelsManagera](#_ModelsManager_1).

Składnia:

virtual int load\_mesh(

[out] Model3DFromFile\* new\_file\_mesh,

[in] const std::string& name) = 0;

Parametry:

new\_file\_mesh [out]

Typ: [Model3DFromFile](#_Model3DFromFile)\*

Obiekt, do którego należy zapisać wszystkie potrzebne informacje o pliku.

Implementując funkcję load\_mesh należy używać funkcji udostępnianych przez [Model3DFromFile](#_Model3DFromFile). Klasa wywołuje wewnątrz funkcje [ModelsManagera](#_ModelsManager_1), służące do dodawania nowych obiektów, co jest konieczne, aby były one niepowtarzalne w silniku. Nie wolno tworzyć obiektów na własną rękę.

name [in]

Typ: const std::string&

Ścieżka do pliku, który ma zostać otwarty.

Wartość zwracana:

Typ: int

Zwraca wartość [MESH\_LOADING\_OK](#_MESH_LOADING) w przypadku powodzenia lub jedną z innych wartości enumeracji [MESH\_LOADING](#_MESH_LOADING), w przypadku błędu.

## Moduły silnika

### CollisionEngine

### ControllersEngine

### DisplayEngine

#include "Engine\DisplayEngine\ DisplayEngine.h"

Definicje:

Engine\ DisplayEngine \ DisplayEngine.cpp

Dziedziczenie:

[DisplayEngine](#_ModelsManager)

class DisplayEngine

{

private:

Engine\* engine;

XMFLOAT4X4 projection\_matrix;

Camera\_object\* current\_camera;

std::vector<Dynamic\_mesh\_object\*> meshes; //modele nieanimowane

std::vector<Camera\_object\*> cameras;

public:

DisplayEngine(Engine\* engine);

~DisplayEngine();

void display\_scene(float time\_interval);

void set\_projection\_matrix(float angle, float X\_to\_Y, float near\_plane, float far\_plane);

};

Opis klasy:

Konstruktory i destruktor:

|  |  |
| --- | --- |
| DisplayEngine(Engine\*) | Pobiera wskaźnik na modu ł główny silnika. |
| ~DisplayEngine() |  |

Funkcje prywatne:

|  |  |
| --- | --- |
| [**set\_view\_matrix**](#_DisplayEngine::set_view_matrix) | Ustawia macierz widoku na podstawie aktualnej kamery. |
|  |  |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**display\_scene**](#_DisplayEngine::display_scene) | Główna funkcja renderująca programu |
| [**set\_projection\_matrix**](#_DisplayEngine::set_projection_matri) | Ustawia macierz projekcji. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Opis funkcji:

#### DisplayEngine::display\_scene

Główna funkcja renderująca programu. Wyświetla wszystkie obiekty, które zostały dodane do klasy.

Wymagania implementacyjne:

Ponieważ klasa [UI\_Engine](#_UI_Engine) także wykonuje renderowanie (wyświetla elementy interfejsu graficznego), funkcja display\_engine nie może sama wywoływać BeginScene() i EndScene() z bilbioteki directX, aby nie było podwojnego wywołania. Z tego względu powyższe funkcje są wywoływane zewnetrznie w pętli głównej przez klasę [Engine](#_Engine).

Funkcja display\_engine ma obowiązek za każdym razem od nowa ustawić macierz widoku i projekcji, ponieważ mogą być zmodyfikowane przez [UI\_Engine](#_UI_Engine). Innymi słowy nie można założyć, że jak się raz ustawiło macierze, to przy następnym wywołaniu będą takie same.

Składnia:

void display\_scene(

[in] float time\_interval)

Parametry:

time\_interval [in]

Typ: float

Czas jaki minął od poprzedniego wykonania pętli głównej programu.

Wartość zwracana:

Typ: void

#### DisplayEngine::set\_projection\_matrix

Tworzy macierz projekcji i zapamiętuje ją w polu projection\_matrix klasy. W każdym wywołaniu funkcji [display\_scene](#_DisplayEngine::display_scene) ustawiana jest macierz zapisana w tym polu. Domyślne wartości macierzy projekcji są ustawione w funkcji Engine::init\_directX.

Składnia:

void set\_projection\_matrix(

[in] float angle,

[in] float X\_to\_Y,

[in] float near\_plane,

[in] float far\_plane);

Parametry:

angle [in]

Typ: float

Kąt widzenia kamery.

X\_to\_Y [in]

Typ: float

Stosunek szerokości ekranu do jego wysokości.

near\_plane [in]

Typ: float

Bliższa powierzchnia obcinania.

far\_plane [in]

Typ: float

Dalsza powierzchnia obcinania.

Wartość zwracana:

Typ: void

#### DisplayEngine::set\_view\_matrix

Używa aktualnie ustawionej kamery, żeby stworzyć macierz widoku. Macierz widoku jest ustawiana w directX. Funkcja jest prywatna, ponieważ jest wywoływana podczas renderowania sceny. Aby ingerować z zewnątrz w ustawienie kamery należy zmienić aktualnie ustawioną kamerę na jedną z innych zapisanych w silniku. Ta funkcja do tego nie służy.

Składnia:

void set\_view\_matrix();

Parametry:

Brak

Wartość zwracana:

Typ: void

### Engine

#include "Engine\Engine.h"

Definicje:

Engine.cpp

Engine\_window\_functions.cpp

Dziedziczenie:

[engine](#_ModelsManager)\_interface

class Engine : public engine\_interface

{

private:

//window functions

ATOM MyRegisterClass();

BOOL InitInstance(int);

//DirectX functions

void clean\_DirectX();

//rendering

void render\_frame();

public:

Engine(HINSTANCE instance);

~Engine();

//DirectX

int init\_directX();

int init\_directXinput();

//window fuctions

BOOL init\_window(int width, int height, BOOL full\_screen, int nCmdShow);

void end\_aplication();

int main\_loop();

//Engine functions

void time\_controller(float& time\_interval);

//event functions

void send\_event(Event\* new\_event);

};

Opis klasy:

Klasa Engine jest nadrzędnym obiektem zarzadzającym wszystkimi modułami silnika. W aplikacji powinien istnieć tylko jeden obiekt podanego typu. O tym, jak się inicjuje działanie silnika, można przeczytac w temacie [Inicjowanie silnika](#_Inicjowanie_silnika).

W tej klasie powinny być umieszczane wszystkie funkcje, które nie powinny być dostępne dla użytkownika silnika. Interfejsem dla użytkownika jest klasa [engine\_interface](#_engine_interface) i tam należy umieszczać pozostałe funkcje. Co do zmiennych klasy, wszystki powinny się znaleźć w klasie [engine\_interface](#_engine_interface), aby wszystkie funkcje mogły mieć do nich dostęp.

Konstruktory i destruktor:

|  |  |
| --- | --- |
| Engine(HINSTANCE instance); | Tworzy obiekt silnika. Trzeba podać uchwyt na instację naszego procesu. |
| ~Engine(); | Destruktora nie można wywołać z poziomu engine\_interface, trzeba mieć wskaźnik na Engine. |

Funkcje prywatne:

|  |  |
| --- | --- |
| [**clean\_DirectX**](#_Engine::clean_DirectX) | Zwalnia interfejsy DirectXa. |
| [**InitInstance**](#_Engine::InitInstance) | Tworzy okno aplikacji. |
| [**MyRegisterClass**](#_Engine::MyRegisterClass) | Rejestruje klasę okna (WinApi) |
| [**render\_frame**](#_Engine::render_frame) | Główna pętla przetwarzania obiektów silnika. |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**end\_aplication**](#_Engine::end_aplication) |  |
| [**init\_directX**](#_Engine::init_directX) |  |
| [**init\_directXinput**](#_Engine::init_directXinput) |  |
| [**init\_window**](#_Engine::init_window) |  |
| [**main\_loop**](#_Engine::main_loop) |  |
| [**send\_event**](#_Engine::send_event) |  |
| [**time\_controller**](#_Engine::time_controller) |  |
|  |  |

Opis funkcji:

#### Engine::clean\_DirectX

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::end\_aplication

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::init\_directX

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::init\_directXinput

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::InitInstance

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::init\_window

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::main\_loop

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::MyRegisterClass

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::render\_frame

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::send\_event

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

#### Engine::time\_controller

Składnia:

Parametry:

model [in]

Typ:

Wartość zwracana:

Typ:

### FableEngine

### ModelsManager

#include "Engine\ModelsManager\ModelsManager.h"

Definicje:

Engine\ModelsManager\ModelsManager.cpp

Dziedziczenie:

[ModelsManager](#_ModelsManager)

class ModelsManager

{

private:

Engine\* engine;

unsigned int materials\_count;

unsigned int textures\_count;

unsigned int meshes\_count;

std::vector<MeshObject\*> meshes;

std::vector<TextureObject\*> textures;

std::vector<MaterialObject\*> materials;

/\*wczytane pliki\*/

std::vector<Model3DFromFile\*> models;

/\*loadery dla różnych formatów plików z modelami\*/

std::vector<Loader\*> loaders;

public:

ModelsManager(Engine\* engine);

~ModelsManager();

MaterialObject\* add\_material(const D3DMATERIAL9 &material, unsigned int& id);

MaterialObject\* get\_material(unsigned int id);

MeshObject\* add\_mesh(Vertex\_Normal\_TexCords1\* vertices, unsigned int vert\_num, unsigned int& id);

MeshObject\* get\_mesh(unsigned int id);

TextureObject\* add\_texture(const std::string& path, unsigned int& id);

TextureObject\* get\_texture(unsigned int id);

private:

bool is\_equal(LPDIRECT3DVERTEXBUFFER9 vertex\_buffer,

Vertex\_Normal\_TexCords1\* vertices, unsigned int vert\_num);

Loader\* find\_loader(const std::string path);

};

Opis klasy:

ModelsManager jest obiektem służącym do zarządzania pamięcią tekstur, meshy oraz materiałów. Tablice meches, materials i textures służą do przechowywania obiektów [MaterialObject](#_MaterialObject), [MeshObject](#_TextureObject) i [TextureObject](#_TextureObject_1). Mamy tu pełny spis wszystkich tych obiektów występujących w całym silniku (oczywiście obiekty w tablicach występują jednokrotnie). Do wskaźników tu zapisanych istnieją odwołania w innych miejscach program jak choćby w obiektach [Model3DFromFile](#_Model3DFromFile) oraz [Dynamic\_mesh\_object](#_Model_object). Z tego względu, aby uniknąć błędów, tylko ta klasa ma prawo niszczyć istniejące obiekty, inne klasy mogą tylko kasować swoje odwołania do nich.

Aby zapewnić, że używane obiekty nie zostaną skasowane, istnieje mechanizm zliczania odwołań do poszczególnych meshy, tekstur i materiałów. Każdy obiekt posiada liczniki odwołań plików i innych obiektów do siebie (więcej na temat mechanizmu odwołań: [referenced\_object](#_referenced_object)). Aby skasować obiekt, należy się upewnić, że nikt się do niego nie odwołuje. Można to zrobić za pomocą funkcji [referenced\_object::can\_delete](#_referenced_object::can_delete).

Pisząc jakiekolwiek funkcje, które zmieniają liczbę odwołań należy pamiętać o modyfikowaniu tej liczby!!!

Do obiektów zarządzanych przez ModelsManager można odwoływać się za pomocą unikalnego identyfikatora, który każdy obiekt otrzymuje w momencie stworzenia. Identyfikatory są unikatowe jedynie wewnątrz swojej klasy, tzn. że mogą istnieć tekstury i meshe, które mają taki sam identyfikator, ale dwa meshe już nie. Do nadawania identyfikatorów używane są zmienne materials\_count, textures\_count, models\_count i meches\_count, zawierające identyfikator, który zostanie przypisany kolejnemu obiektowi danego typu.

Wszystkie tablice są z natury wykonywanych operacji posortowane rosnąco po unikalnych identyfikatorach. Wynika to z tego, ze każdy nowy element jest dodawany na koniec tablicy. Nie należy naruszać tej prawidłowości, ponieważ funkcje wyszukujące używają wyszukiwania binarnego.

Konstruktory i destruktor:

|  |  |
| --- | --- |
| ModelsManager(Engine\* engine) | Otrzymuje wskaźnik na obiekt główny silnika. |
| ~ModelsManager() |  |

Funkcje prywatne:

|  |  |
| --- | --- |
| [**is\_equal**](#_ModelsManager::is_equal) | Porównuje bufor wierzchołków directX z tablicą wierzchołków podaną w parametrze. |
| [**find\_loader**](#_ModelsManager::find_loader) | Znajduje Loader pasujący do podanego pliku. |

Funkcje publiczne:

|  |  |
| --- | --- |
| [**add\_material**](#_ModelsManager::add_material) | Dodaje materiał do tablicy materiałów. |
| [**get\_material**](#_ModelsManager::get_material) | Zwraca wskaźnik na materiał na podstawie indeksu. |
| [**add\_mesh**](#_ModelsManager::add_mesh) | Dodaje mesh do tablicy meshy. |
| [**get\_mesh**](#_ModelsManager::get_mesh) | Zwraca wskaźnik na mesh na podstawie indeksu. |
| [**add\_texture**](#_ModelsManager::add_texture) | Dodaje teksturę to tablicy tekstur. |
| [**get\_texture**](#_ModelsManager::get_texture) | Zwraca wskaźnik na teksturę na podstawie identyfikatora. |
| add\_model | Dodaje nowy plik z modelem do tablicy. |
| get\_model | Zwraca wskaźnik na plik z modelem na podstawie identyfikatora. |

Opis funkcji:

#### ModelsManager::is\_equal

Porównuje podany w pierwszym parametrze obiekt bufora wierzchołków directX, z tablicą podaną w drugim parametrze o liczbie wierzchołków vert\_num.

Składnia:

bool ModelsManager::is\_equal(

[in] LPDIRECT3DVERTEXBUFFER9 vertex\_buffer,

[in] Vertex\_Normal\_TexCords1\* vertices,

[in] unsigned int vert\_num)

Parametry:

vertex\_buffer[in]

Typ: LPDIRECT3DVERTEXBUFFER9

Interfejs bufora wierzchołków DirectX.

vertices[in]

Typ: [Vertex\_Normal\_TexCords1](#_Vertex_Normal_TexCords1)\*

Wskaźnik na tablicę wierzchołków typu [Vertex\_Normal\_TexCords1](#_Vertex_Normal_TexCords1).

vert\_num[in]

Typ: unsigned int

Liczba wierzchołków w tablicy podanej w parametrze vertices.

Wartość zwracana:

Typ: bool

Zwraca wartość true, jeżeli bufory są takie same w przeciwnym razie zwraca false.

#### ModelsManager::find\_loader

Znajduje [Loader](#_Loader) pasujący do pliku podanego w parametrze. Wewnątrz wywoływana jest funkcja [can\_load](#_Loader::can_load) interfejsów [Loaderów](#_Loader), które są przechowywane w ModelsManagerze.

Składnia:

Loader\* find\_loader(

[in] const std::string& path);

Parametry:

path [in]

Typ: const std::strin&

Ścieżka do pliku, dla którego szukamy [Loader](#_Loader)a.

Wartość zwracana:

Typ: Loader\*

Funkcja zwraca wskaźnik na interfejs [Loader](#_Loader)a, który móże otworzyc plik lub nullptr, jeżeli taki się nie znalazł.

#### ModelsManager::[**add\_material**](#_ModelsManager::add_material)

Dodaje materiał do ModelsManagera. Materiał jest dodawany tylko, jeżeli jeszcze nie istniał. W przeciwnym razie zwracany jest wskaźnik na materiał wcześniej stworzony.

Składnia:

MaterialObject\* ModelsManager::add\_material(

[in] const D3DMATERIAL9 &material,

[out] unsigned int& id)

Parametry:

material [in]

Typ: const D3DMATERIAL9&

Referencja na strukturę materiału w formacie directXa.

id [out]

Typ: unsigned int&

Unikalny identyfikator materiału, którego można używać do odwoływania się. Jest niezmienny do końca gry.

W przypadku błędów zwracana jest wartość [WRONG\_MATERIAL\_ID](#_WRONG_MATERIAL_ID).

Wartość zwracana:

Typ: [MaterialObject](#_MaterialObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający materiał.

#### ModelsManager::[get\_material](#_ModelsManager::get_material)

Na podstawie indeksu materiału zwraca skojarzony z tym indeksem obiekt. Funkcja używa przeszukiwania binarnego, które zakłada, że tablica jest posortowana rosnąco. Z tego względu żadna funkcja nie ma prawa dodawać elementów inaczej niż na koniec listy.

Składnia:

MaterialObject\* ModelsManager::get\_material(

[in] unsigned int id)

Parametry:

id [in]

Typ: unsigned int

Unikalny identyfikator materiału, do którego chcemy się odwołać.

Wartość zwracana:

Typ: [MaterialObject](#_MaterialObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający materiał.

#### ModelsManager::add\_mesh

Dodaje nowy mesh do ModelsManagera. Mesh jest dodawany tylko, jeżeli jeszcze nie istniał. W przeciwnym razie zwracany jest wskaźnik na wcześniej stworzony obiekt.

Składnia:

MeshObject\* ModelsManager::add\_mesh(

[in] Vertex\_Normal\_TexCords1\* vertices,

[in] unsigned int vert\_num,

[out] unsigned int& id)

Parametry:

vertices [in]

Typ: [Vertex\_Normal\_TexCords1](#_Vertex_Normal_TexCords1)\*

Wskaźnik na tablicę wierzchołków typu [Vertex\_Normal\_TexCords1](#_Vertex_Normal_TexCords1).

vert\_num [in]

Typ: unsigned int

Liczba wierzchołków w podanej w pierwszym parametrze tablicy.

id [out]

Typ: unsigned int&

Unikalny identyfikator mesha, którego można używać do odwoływania się. Jest niezmienny do końca gry.

W przypadku błędów zwracana jest wartość [WRONG\_MESH\_ID](#_WRONG_MESH_ID).

Wartość zwracana:

Typ: [MeshObject](#_MeshObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający mesh.

#### ModelsManager::get\_mesh

Na podstawie indeksu mesha zwraca skojarzony z tym indeksem obiekt. Funkcja używa przeszukiwania binarnego, które zakłada, że tablica jest posortowana rosnąco. Z tego względu żadna funkcja nie ma prawa dodawać elementów inaczej niż na koniec listy.

Składnia:

MeshObject\* ModelsManager::get\_mesh(

[in] unsigned int id)

Parametry:

id [in]

Typ: unsigned int

Unikalny identyfikator mesha, do którego chcemy się odwołać.

Wartość zwracana:

Typ: [MeshObject](#_MeshObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający mesh.

#### ModelsManager::add\_texture

Dodaje teksturę do ModelsManagera. Tekstura jest dodawana tylko, jeżeli jeszcze nie istniała. W przeciwnym razie zwracany jest wskaźnik na wcześniej stworzoną teksturę.

Składnia:

TextureObject\* ModelsManager::add\_texture(

[in] const std::string& path,

[out] unsigned int& id)

Parametry:

path [in]

Typ: const std::string& path

Referencja na ścieżkę do pliku zawierającego teksturę.

id [out]

Typ: unsigned int&

Unikalny identyfikator tekstury, którego można używać do odwoływania się. Jest niezmienny do końca gry.

W przypadku błędów zwracana jest wartość [WRONG\_TEXTURE\_ID](#_WRONG_TEXTURE_ID).

Wartość zwracana:

Typ: [TextureObject](#_TextureObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający teksturę.

#### ModelsManager::get\_texture

Na podstawie indeksu tekstury zwraca skojarzony z tym indeksem obiekt. Funkcja używa przeszukiwania binarnego, które zakłada, że tablica jest posortowana rosnąco. Z tego względu żadna funkcja nie ma prawa dodawać elementów inaczej niż na koniec listy.

Składnia:

TextureObject\* ModelsManager::get\_texture(

[in] unsigned int id)

Parametry:

id [in]

Typ: unsigned int

Unikalny identyfikator tekstury, do której chcemy się odwołać.

Wartość zwracana:

Typ: [TextureObject](#_TextureObject)\*

Funkcja zwraca wskaźnik na obiekt zawierający teksturę.

#### ModelsManager::add\_model

Dodaje plik z modelem. Jeżeli model już istaniał, to nie jest tworzony nowy, tylko zwracane dane starego. Modele nie są porównywane, liczy się tylko nazwa pliku.

Składnia:

Model3DFromFile\* add\_model(

[in] const std::string& file\_name,

[out] unsigned int& id);

Parametry:

file\_name [in]

Typ: const std::string&

Nazwa pliku z modelem.

id [out]

Typ: unsigned int

Zwraca wartość unikalnego identyfikatora pliku lub [WRONG\_MODEL\_FILE\_ID](#_WRONG_MODEL_FILE_ID), jeżeli coś pójdzie nie tak.

Wartość zwracana:

Typ: [Model3DFromFile](#_Model3DFromFile)\*

Funkcja zwraca wskaźnik na obiekt zawierający dane pliku.

#### ModelsManager::get\_model

Na podstawie indeksu pliku z modelem zwraca skojarzony z tym indeksem obiekt. Funkcja używa przeszukiwania binarnego, które zakłada, że tablica jest posortowana rosnąco. Z tego względu żadna funkcja nie ma prawa dodawać elementów inaczej niż na koniec listy.

Składnia:

Model3DFromFile\* get\_model(

[in] unsigned int id);

Parametry:

id [in]

Typ: unsigned int

Unikalny identyfikator pliku z modelem, do którego chcemy się odwołać.

Wartość zwracana:

Typ: [Model3DFromFile](#_Model3DFromFile)\*

Funkcja zwraca wskaźnik na obiekt zawierający dane pliku.

### MovementEngine

### PhysicEngine

### UI\_Engine

## Stałe i makrodefinicje

### \_\_UNUSED

Definicja:

Configuration properties -> C/C++ -> Preprocesor -> Preprocesor Definitions

Zdefiniowanie tej stałej powoduje usunięcie z kodu źródłowego wszystkich funkcji, które zostały otoczone dyrektywą preprocesora #ifndef \_\_UNUSED. Funkcje potraktowane w ten sposób okazały się niepotrzebne w trakcie pisania programu, mogą być nieskończone lub niedziałające.

Domyślnie dyrektywa jest dodana do dyrektyw preprocesora w opcjach projektu.

### EVENT\_TYPE

Zawiera wszystkie predefiniowane eventy, które służą do wypełnienia pola type klasy [Event](#_Event).

Przeczytaj więcej o [mechanizmie eventów](#_Eventy).

### FIXED\_MOVE\_UPDATE\_INTERVAL

Definicja:

"Engine\Engine.cpp"

const float FIXED\_MOVE\_UPDATE\_INTERVAL = ((float)1 / (float)56);

Przeliczanie nowych położeń obiektów, fizyki, sztucznej inteligencji oraz kolizji odbywa się w stałym interwale czasowym. Ten interwał określony jest przez stałą FIXED\_MOVE\_UPDATE\_INTERVAL wyrażoną w sekundach. Wynosi ona 0.0178571429. Wartośc ta daje się łatwo zapisać w systemie binarnym, dlatego jest taka dziwna. Przy zmianie polecam podawać zawsze takie ułamki, które są łatwe w zapisie binarnym, co ogranicza występowanie niedokładności wynikających z przybliżania stałych. Praktyka ta jest zalecana, ale w tym przypadku nie wymagana.

Wartość ta oznacza, że położenia obiektów są przeliczane 56 razy na sekundę.

(Zobacz temat: [Pętla główna silnika](#_Pętla_główna_silnika), [Potok przetwarzania obiektów](#_Potok_przetwarzania_obiektów)).

### FRAMES\_PER\_SEC\_UPDATE

Definicja:

"Engine\Engine.h"

Stała oznaczająca czas, co który aktualizowana jest wartość frames\_per\_sec. Wynosi ona 10 000 milisekund, czyli aktualizacja licznika następuje co 10 sekund.

### MESH\_LOADING

Definicja:

"Engine\ModelsManager\Loaders\loader\_interface.h"

### Modules

Definicja:

Engine\Engine.h

typedef enum

{

E\_CONTROLLERS = 0,

E\_PHYSIC = 1,

E\_FABLE = 2,

E\_MOVEMENT\_DYNAMIC = 3,

E\_MOVEMENT\_COMPLEX = 4,

E\_DISPLAY = 5,

E\_COLLISION = 6,

E\_UI = 7,

E\_MODELS\_MANAGER = 8

} Modules;

Opis:

Stałe definiujące numery poszczególnych modułów silnika. Służą głównie jako indeksy dostępu do tablicy index\_predictor klasy [Engine](#_Engine). Tablica ta przechowuje najbardziej prawdopodobne pozycje obiektów w poszczególnych modułach silnika. (Więcej na temat przyspieszania dostępu do obiektów w opisie klasy [Engine](#_Engine))

E\_CONTROLLERS

Klasa [ControllersEngine](#_ControllersEngine).

E\_PHYSIC

Klasa [PhysicEngine](#_PhysicEngine).

E\_FABLE

Klasa [FableEngine](#_FableEngine).

E\_MOVEMENT\_DYNAMIC

Klasa [MovementEngine](#_MovementEngine). Dostęp do tablicy przechowującej obiekty dynamiczne, ale nie złożone.

E\_MOVEMENT\_COMPLEX

Klasa [MovementEngine](#_MovementEngine). Dostęp do tablicy przechowującej obiekty złożone.

E\_DISPLAY

Klasa [DisplayEngine](#_DisplayEngine).

E\_COLLISION

Klasa [CollisionEngine](#_CollisionEngine).

E\_UI

Klasa [UI\_Engine](#_UI_Engine).

E\_MODELS\_MANAGER

Klasa [ModelsManager](#_ModelsManager_1). Uwaga! Nie posiada swojego miejsca w tablicy index\_predictor, ponieważ przechowuje obiekty innego typu, do których nie potrzeba przyspieszania dostępu w ten sposób.

### STANDARD\_ABSTRACTION\_LAYER

Definicja:

Engine\UI\_Engine\InputAbstractionLayer.h

typedef enum

{

GAME

} STANDARD\_ABSTRACTION\_LAYER;

Opis:

Zestaw stałych służących do wybierania jednej ze standardowych [wartw abstrakcji](#_Obsługa_klawiatury,_myszy) urządzeń wejścia wbudowanych w silnik.

### STANDARD\_ABSTRACTION\_LAYER\_COUNT

#define STANDARD\_ABSTRACTION\_LAYER\_COUNT 1

Stała opisująca liczbe wbudowanych wartw abstrakcji istniejących w silniku. Należy pamiętać, żeby ją modyfikować, jeżeli zmieniamy coś w enumeracji [STANDARD\_ABSTRACTION\_LAYER](#_STANDARD_ABSTRACTION_LAYER). Niemodyfikowanie stałej może umożliwić skasowanie wbudowanej wartwy abstrakcji, albo inne niepoprawne działania silnika.

### WRONG\_ID

### WRONG\_MATERIAL\_ID

### WRONG\_MESH\_ID

### WRONG\_MODEL\_FILE\_ID

### WRONG\_TEXTURE\_ID

# API dla użytkowników silnika