helion.pl

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Blender. Od planowania, modelowania oraz teksturowania do animacji i renderingu. Praktyczne projekty

Autor: Jarosław Kolmaga
ISBN: 83-246-1465-6
Format: 168x237, stron: 328

Poznaj tajniki Blendera i twórz wyjątkowe projekty w technologii 3D

- Jak wykonywać rendery statyczne?
- Jak przygotować bazę modelu?
- Jak wykorzystać skrypty języka Python?

Blender to znakomity, darmowy program do modelowania i renderowania obrazów oraz animacji trójwymiarowej. Posiada niekonwencjonalny interfejs użytkownika, bez zachodzących na siebie i blokujących się okien, własny silnik graficzny oraz wiele funkcji do edycji obiektów, pozwalających uzyskać praktycznie dowolny kształt. Istnieje kilka wersji Blendera – na różne platformy sprzętowe i programowe. Za pomocą tej aplikacji można tworzyć przestrzenne projekty techniczne, gry komputerowe, reklamy telewizyjne i profesjonalne logotypy firm. To wszystko sprawia, że Blender jest niezwykle popularny i coraz szerzej wykorzystywany.

„Blender. Od planowania, modelowania oraz teksturowania do animacji i renderingu. Praktyczne projekty" to nietypowy podręcznik, pokazujący działanie programu i jego zaawansowane funkcje na przykładzie budowania konkretnego projektu – czołgu M1A2 Abrams. Dzięki temu możesz nauczyć się, na czym polega modelowanie, teksturowanie, rozpakowywanie siatek, rigging oraz rendering. Taka konstrukcja to wielki atut książki, bowiem pozwala Ci podjąć pracę nad projektem w dowolnym miejscu, przyjrzeć się sposobowi rozwiązania danego problemu, poznać nie tylko narzędzia i funkcje, ale także sposoby organizacji pracy nad projektem.

- Bluprints
- Skrypty języka Python
- Przygotowanie projektu
- Modelowanie
- Rewaloryzacja
- Pędzle, desenie i efekty ze zdjęć
- Malowanie modelu hi-poly oraz low-poly
- UV Unwrapping
- Rigging
- Rendering i środowisko renderingu Kerkythea
- Typy świateł i oświetlenia

Cała teoria i praktyka, których potrzebujesz, aby tworzyć niesamowite projekty 3D

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wstęp 7

Rozdział 1. Przygotowanie projektu 9

Część 1.1. Materiały 9
1.1.1. Zasoby referencyjne 9
1.1.2. Blueprints 10
1.1.3. Szkice 11
Część 1.2. Warsztat 12
1.2.1. Opis skryptów języka Python 12
1.2.2. Skrypt Geom Tool 13
1.2.3. Skrypt UV Tool 15
1.2.4. Przygotowanie miejsca pracy 15
Część 1.3. Czołg M1A2 Abrams 16
1.3.1. Historia serii M1 16
1.3.2. Opancerzenie 17
1.3.3. Uzbrojenie główne, dodatkowe i urządzenia celownicze 17

Rozdział 2. Modelowanie 19

Część 2.1. Przygotowanie bazy modelu 20
2.1.1. Bryła korpusu 20
2.1.2. Wieżyczka 24
2.1.3. Lufa i boczne płyty pancerza 30
2.1.4. Koła prowadzące i napędowe 34
Część 2.2. Detale 39
2.2.1. Przód pojazdu 40
2.2.2. Przednia płyta pancerza 52
2.2.3. Boki korpusu 57
2.2.4. Tył pojazdu 62
2.2.5. Płyty pancerza bocznego 75
2.2.6. Lufa 78
2.2.7. Wieżyczka: boki i tył 84
2.2.8. Wieżyczka: część górna 90
2.2.9. Układ jezdny 110
Część 2.3. Rewaloryzacja 116
2.3.1. Sharp Modifier i F-Gony 116
2.3.2. Dalsze detale 122

Rozdział 3. Teksturowanie 127
Część 3.1. UV Unwrapping 128
3.1.1. Teoria 128
3.1.2. Siatki lufy i gniazda 131
3.1.3. Siatki M2HB i M240 138
3.1.4. Siatki podstawowej bryły wieży 167
3.1.5. Siatki systemu trakcji 175
3.1.6. Siatki korpusu 177
Część 3.2. Przygotowanie materiałów 183
3.2.1. Logistyka 183
3.2.2. Pędzle, desenie i efekty ze zdjęć 185
3.2.3. Opracowanie materiałów dla czołgu M1A2 188
Część 3.3. Malowanie modelu hi-poly 189
3.3.1. Przygotowania i radzenie sobie ze szwami 189
3.3.2. Col BKG z maskowaniem leśnym 190
3.3.3. Highlighting 192
3.3.4. Mapy bazowe (kompozycja Color) 192
3.3.5. Mapy rozchodzenia się światła (kompozycja Diff) 195
3.3.6. Mapy wypukłości (kompozycja Bump) 196
3.3.7. Mapy odbijania światła (kompozycja Spec), twardości odbić (kompozycja Hard) oraz odbić lustrzanych (kompozycja Mirror) 198
Część 3.4. Malowanie modelu low-poly 199
3.4.1. Proste wypukłości i wgłębienia 200
3.4.2. Wycięcia 201
3.4.3. Sztuczne światło 203
Część 3.5. Tekstury w praktyce 205
3.5.1. Siatki UV 206
3.5.2. Malowanie tekstur 206

Rozdział 4. Rigging 213
Część 4.1. Podstawowy szkielet czołgu 214
Część 4.2. Podstawowa funkcjonalność szkieletu 219
Część 4.3. Zaawansowana funkcjonalność szkieletu 223
4.3.1. Dualny system namierzania 223
4.3.2. Animacja ruchu gąsienic 225

Rozdział 5. Rendering 229
Część 5.1. Kompozycja 229
5.1.1. Kadr 230
5.1.2. Perspektywa 235
5.1.3. Kolor, kontrast, równowaga… 242
Część 5.2. Oświetlenie 243
5.2.1. Typy świateł i oświetlenia 244
5.2.2. Sposoby oświetlenia modelu 245
5.2.3. Kompozycja świetlna w różnych sytuacjach 249

Część 5.3. Silnik renderujący Yaf(a)Ray 251
5.3.1. Instalacja i uruchomienie renderera Yaf(a)Ray 251
5.3.2. Zakładka Object/Light/Camera dla obiektów Mesh oraz kamery 252
5.3.3. Zakładka Object/Light/Camera dla świateł 255
5.3.4. Zakładka Material 257
5.3.5. Zakładka Render 260
Część 5.4. Środowisko renderingu Kerkythea 267
5.4.1. Rozpoczęcie pracy z Kerkytheą 268
5.4.2. Ustawienia sceny 276
5.4.3. Ustawienia nieba 282
5.4.4. Ustawienia materiałów 284
5.4.5. Instancing Brush 311
5.4.6. Rendering 314

Skorowidz 317

Rozdział 4.
Rigging

Niektórzy spośród grafików 3D uważają rigging za zbędny podczas wykonywania renderów statycznych. Po części jest to prawda — można pracować bez kośćca, a jego nieobecność wcale nie zmieni wyniku końcowego. Wyobraź sobie jednak, że chciałbyś przetestować różne sposoby ustawienia czołgu — bez kośćca każdorazowo konieczne byłoby ręczne ustawianie kursora 3D na osi obrotu danej bryły, zaznaczanie wierzchołków przynależnych danej transformacji i wreszcie modyfikacja ustawienia kształtu. Sytuacja mogłaby wyglądać znacznie gorzej — utworzenie szkieletu i jego skonfigurowanie pozwala bowiem w bardzo łatwo modyfikować sposób ustawienia gąsienic, a więc na przykład odwzorowanie ich zniekształcenia po wjechaniu pojazdu na nierówny teren. Bez wykorzystania armatur odpowiednie zmodyfikowanie gąsienic byłoby dużo trudniejsze, zwłaszcza jeśli konieczne okazałoby się ich kilkukrotne poprawianie. Jeśli zaś chodzi o animację, trudno sobie wyobrazić, jak bez kośćca mogłaby ona przebiegać.

Kiedy mowa o szkielecie czołgu, naturalnie koncentrujemy się na riggingu czołgowej gąsienicy. Sam z tematem zetknąłem się po raz pierwszy podczas modelowania i riggingu futurystycznego pojazdu z uniwersum znanej gry bitewnej. Po dniu pełnym pracy, rozczarowań i sukcesów udało mi się stworzyć funkcjonalny system, na tyle skuteczny, by animacja czołgu stała się czystą przyjemnością, i na tyle skomplikowany, że pod koniec pracy potrafiłem tylko jasno określić funkcje kilku zewnętrznych kontrolerów — do nich też odnosiłem się, dobudowując kolejne elementy szkieletu — zupełnie nie wiedziałem jednak, „jak to działa” w środku. Specjalnie na okazję pracy z czołgiem M1 odświeżyłem stary system, posprzątałem go nieco i uprościłem, wykonałem też dziesiątki testów mających na celu odnalezienie „jedynych słusznych rozwiązań”. Nie ukrywam, że montaż tego szkieletu może być trudny, nie tylko ze względu na wielość elementów, ale też pewne drobne niuanse, jak chociażby konieczność wykorzystywania pośredników dla relacji dziecko – rodzic. Nie zrażaj się więc, jeśli coś nie wychodzi — z pomocą poniższego tekstu oraz dostępnego na płycie CD modelu powinieneś móc stworzyć własną wersję systemu sterowania trakcją czołgu.

Zachęcam też do eksperymentowania na własną rękę — poniższy tekst jest w końcu najlepszym dowodem na to, że blenderowy system modyfikatorów posiada olbrzymie możliwości!

Kościec wykorzystywany dla renderów statycznych oraz ten dla animacji są niemal identyczne. Różnica polega na pewnych usprawnieniach, które wymagane są podczas animowania obiektów — jak na przykład kontrolery poruszania się gąsienic czy inny system namierzania wykorzystywany dla działa oraz karabinów maszynowych. W części poświęconej riggingowi opiszę, w jaki sposób stworzyć intuicyjny interfejs użytkownika z wykorzystaniem systemu kości i ograniczników Blendera.

Część 4.1. Podstawowy szkielet czołgu

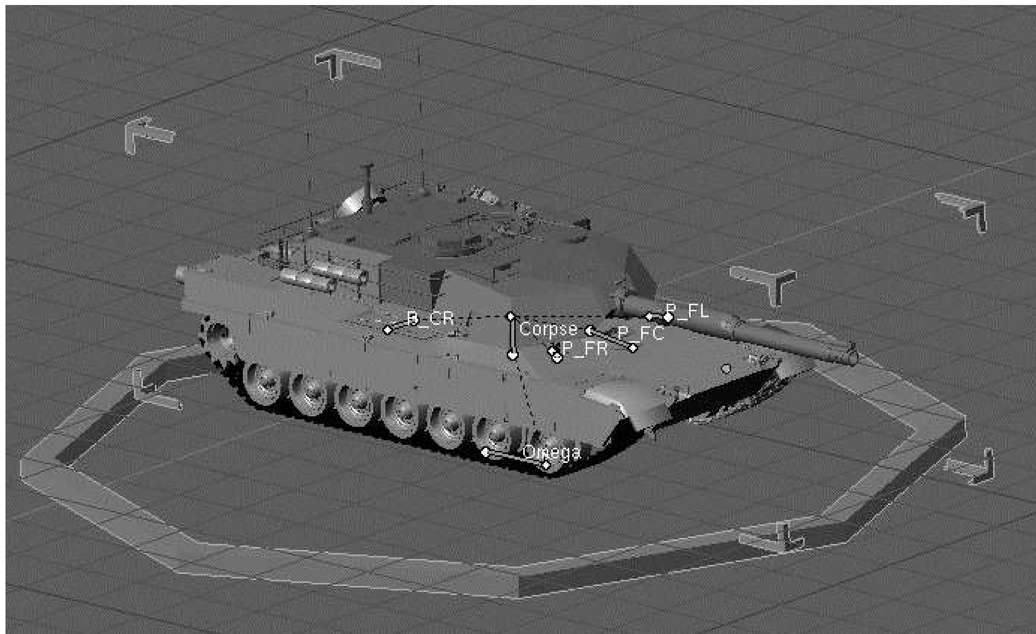
Podstawą dobrego szkieletu jest planowanie. Musimy więc się zastanowić, jakie grupy obiektów będą nam potrzebne, i odpowiednio podzielić model. Podział ten może przebiegać na dwa sposoby. Jeśli posiadasz wyjątkowo szybki komputer lub skupiłeś się na nisko poligonowym modelu, wybierzesz zapewne najbardziej „naturalną" metodę podziału poprzez tworzenie grup wierzchołków. W przeciwnym razie świetnym pomysłem jest stworzenie kilku obiektów oraz kilku szkieletów, współpracujących z sobą jak jeden, lecz pozwalających na odseparowanie poszczególnych systemów. W wypadku naszego czołgu wystarczą trzy takie systemy: korpusu, trakcji oraz uzbrojenia. Pierwszy z kośćców będzie służył do kontroli położenia czołgu, jego przechyłu oraz elementów ruchomych takich jak kalpy; będzie też systemem głównym, do którego będą odwoływać się pozostałe kośćce. System trakcji powinien odpowiadać za kontrolę gąsienic oraz kół — w wersji statycznej ograniczy się on do ustalania położenia poszczególnych kół oraz określenia, w jaki sposób na nie zostanie nałożona gąsienica. Ostatni system będzie kontrolował obrót oraz pochylenie armaty oraz obu karabinów.

Zanim przejdę do opisu kośćca korpusu, słów kilka o kwestiach nazewnictwa. Podczas tworzenia kośćca niezwykle dużo zależy od odpowiedniego dobrania i przestrzegania nazewnictwa — zarówno kości jak i całych obiektów. Pracując nad ogranicznikami, co chwilę będziemy musieli odnosić się do jakiegoś obiektu, stąd też lepiej, żebyśmy nie zastanawiali się, jak właściwie nazywa się kość odpowiadająca za czwarte koło z lewej strony. W miarę opisywania kolejnych systemów będę także tłumaczył wykorzystane przeze mnie nazewnictwo. W swoich pracach z przyzwyczajenia używam nazw angielskich (w końcu całe środowisko programu bazuje na tym języku), takich też będę używał podczas pracy nad czołgiem Abrams. Kwestią pochodną od nazewnictwa, o nieco tylko mniejszym znaczeniu, jest używanie siatek *Mesh* podstawianych jako sposób wyświetlania kości (*custom draw type*) — ich wykorzystanie potrafi zmienić uciążliwą pracę

w zabawę. Co jednak jest znacznie ważniejsze, kościec wykorzystujący dobre siatki podstawione pod kości może być używany bez samego modelu — wystarczy bowiem rzut oka na szkielet, by dowiedzieć się, co w danej chwili dzieje się z czołgiem. Jest to niezwykle ważne, gdy animujemy obiekty o dość dużej złożoności.

System sterowania korpusem, nazwany przeze mnie *Main Sys* musi się opierać na kości bazowej. Wbrew pozorom nie chodzi tu o kość sterującą korpusem, ale o kość-matkę, sterującą wszystkimi kośćmi wszystkich systemów. Kość taką zazwyczaj nazywam mianem *Omega* i umieszczam w centrum obiektu — w tym wypadku wzdłuż dolnej krawędzi kadłuba. Siatka reprezentująca kość *Omega* powinna być możliwie duża i łatwo dostrzegalna — świetnie sprawdza się foremny wielościan, nadający modelowi wygląd jednostki z gier strategicznych. Dopiero w drugiej kolejności pojawia się kość odpowiedzialna za korpus (*Corpse*), podłączona oczywiście do kości *Omega* i umieszczona nad nią. Wizualizacja kontrolera korpusu może przebiegać na dwa sposoby. Jednym z nich jest umieszczenie niewielkiego znacznika w dowolnym kształcie ponad czołgiem, drugim zaś stworzenie trójwymiarowych obejm, znajdujących się na wierzchołkach prostopadłościanu, w jakim można by nasz czołg zamknąć. W dalszej kolejności należy określić wszystkie ruchome części modelu — w naszym wypadku będzie to sześć otwieralnych (bądź przesuwalnych) pokryw. Oczywiście nie wymodelowaliśmy wnętrza czołgu, więc ich pełne otworzenie nie jest być może najlepszym pomysłem, jednak samo ich uchylenie może znacznie zmienić wygląd modelu. Nie ma też problemu, by samą klapą lub innym elementem zasłonić otwór — w takim wypadku stwarzamy pozory wielowymiarowości przekraczającej nawet to, co wymodelowaliśmy. Świetnie sprawdza się tu modułowy system nazewnictwa — wykorzystamy przedrostek *P_* oznaczający ruchomą pokrywę oraz dwie litery oznaczające jej położenie — *F* i *C* oznaczające odpowiednio front i centrum oraz *R*, *L*, *C* oznaczające prawą stronę obiektu, lewą stronę obiektu lub jego środek. Zgodnie z tym nazewnictwem przednia środkowa klapa będzie nazywać się *P_FC*, środkowa prawa zaś *P_CR*. O ile same nazwy są być może mało intuicyjne, to stosując taki system, nie musimy ich wcale znać — możemy je bowiem wyprowadzić w każdej chwili. Podczas tworzenia kości dla klap pamiętaj, że musisz odpowiednio wycentrować kursor! W wypadku klap *P_FR* i *P_FL* jest konieczne dwukrotne centrowanie kursora. Za pierwszym razem określamy miejsce, w którym znajduje się oś obrotu obiektu i umieszczamy w nim trzon kości, za drugim — szukamy takiego punktu, by kość przyjęła dobry zwrot, centrując do niego koniec kości. Drugie centrowanie kursora można przeprowadzić względem pręta po przeciwnej stronie zawiasu. Wygląd armatury *Main Sys* prezentuje rysunek 4.1.

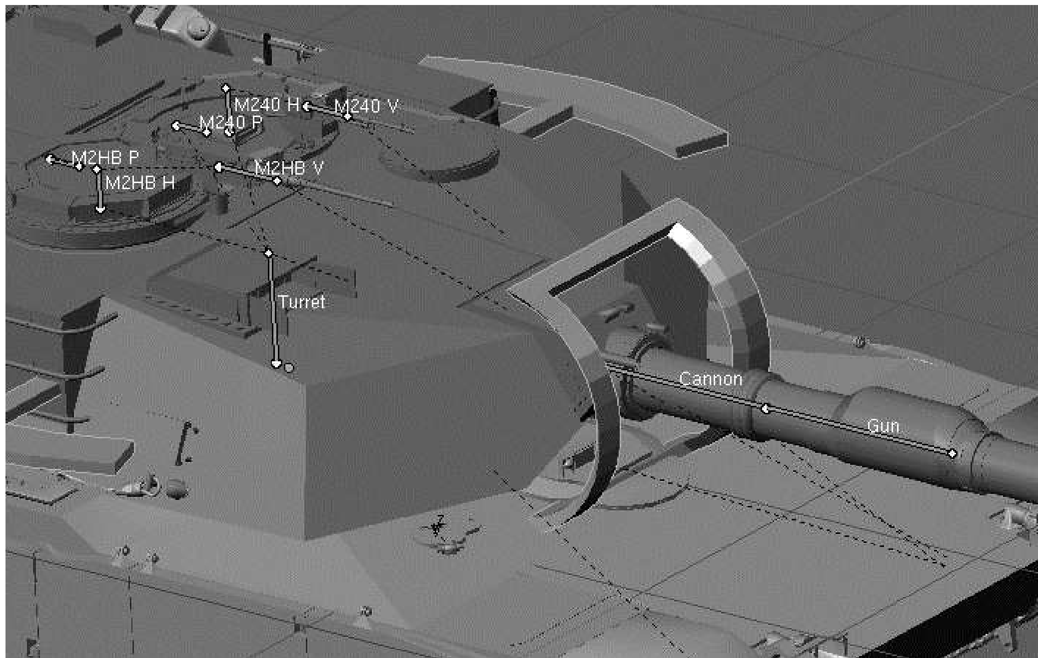
Następny z systemów, *Armanent Sys*, składa się w sumie z ośmiu kości, tworzących połączenia nieco bardziej skomplikowane niż w wypadku *Main Sys*. Podstawą będzie oczywiście kość odpowiadająca za całość wieży, która musi się znaleźć



Rysunek 4.1. *Armatura Main Sys*

na jej osi obrotu — *Turret*. Kość ta ukazuje doskonale, jak wygodne może okazać się tworzenie siatek zastępczych dla kości — tutaj warto stworzyć okrągły kształt o dużym promieniu lub krzyż, którego ramiona będą przenikać przez ścianki wieży. Oczywiście można by też przenieść kość wzwyż na osi obrotu, rozwiązanie to jest jednak niepraktyczne — chcąc wykorzystać je dla wszystkich kości znacznie zwiększymy panujący wśród nich chaos. Następna w kolejności jest kość *Cannon*, która musi odpowiadać za unoszenie armaty. W tym wypadku jako siatkę zastępczą zwykłem wykorzystywać okrągły kształt otaczający armatę — jest on prosty do odnalezienia i dość dobrze komponuje się z resztą szkieletu, nie wprowadzając bałaganu. Zaraz za kością *Cannon* musi też znajdować się zależna od niej kość *Gun*, która w przyszłości pozwoli nam na symulowanie odrzutu wynikłego z wystrzałów. Do kości *Cannon* podłączone powinno być wyłącznie gniazdo lufy, sama lufa zaś musi być przyporządkowana kości *Gun*. Kolejne kości dotyczą karabinów maszynowych oraz ich pokryw — w tym wypadku zastosowałem system nazewnictwa wykorzystujący nazwę karabinu oraz litery *H*, *V* oraz *P* dla oznaczenia kości sterujących kątem w poziomie (od angielskiego *horizontal*), pionie (od *vertical*) oraz unoszeniem pokryw włazów. Sposób tworzenia kości w dużej mierze zależy od karabinów. W wypadku M2HB kości *V* i *P* są dziećmi kości *H*, jako że ostatnia z nich porusza całym włazem; z kolei w wypadku karabinu M240 istnieje zależność pomiędzy kośćmi *V* oraz *H*; kość *P* jest już zależna tylko od kości-matki całego systemu, *Turret*. Dodatkowo zwróć uwagę na to, by kość *M240 H* była dobrze umiejscowiona — musisz ją wyrównać do szyny otaczającej właz ładowniczego, jednak szyna ta jest nieco obcięta — dlatego

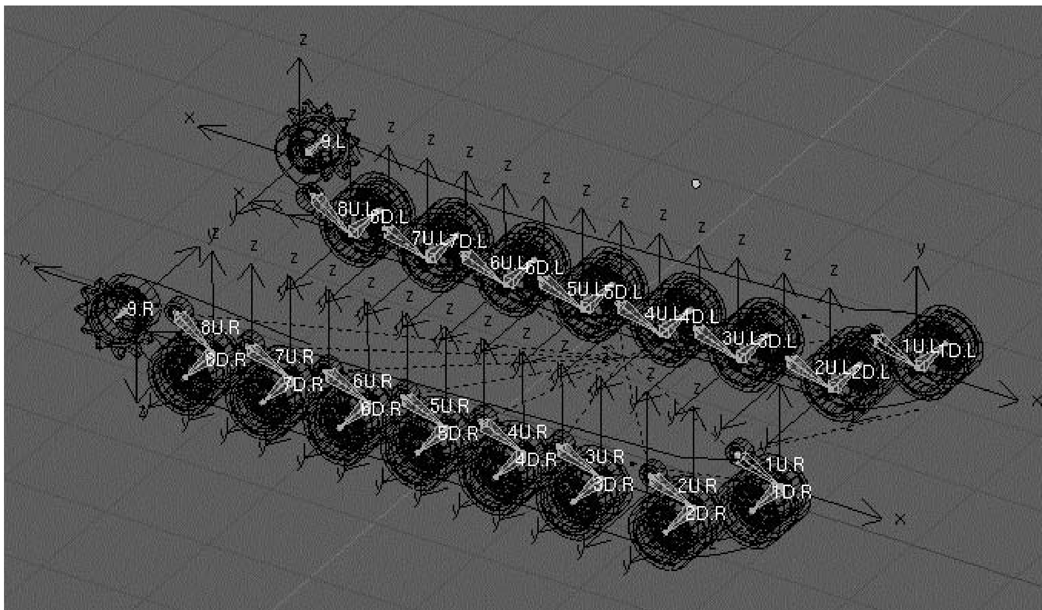
nie zaznaczaj jej całej, lecz dwa naprzeciwległe wierzchołki, krawędzie lub powierzchnie! Armaturę należy jeszcze uzależnić od kości *Corpse* z kośćca *Main Sys*. W jaki sposób to zrobić? Procedura jest bardzo prosta — wystarczy, że włączysz tryb *Pose Mode* dla kośćca *Main Sys*, zaznaczysz kościec *Armanent Sys*, następnie kość *Corpse* i wywołasz menu tworzenia związku rodzic – dziecko standardowym skrótem *Ctrl+P*. Z menu kontekstowego wybierz opcję *Bone* — dzięki temu cały system *Armanent Sys* zostanie uzależniony od kości *Corpse*. Kościec *Armanent Sys* prezentuje rysunek 4.2.



Rysunek 4.2. *Armatura Armanent Sys*

System trakcji (*Traction Sys*) nie jest być może wybitnie skomplikowanym kośćcem, wymaga jednak sporo pracy ze względu na liczbę kół. Wszystkie koła nośne należy wyposażyć w parę kości, jedną odpowiadającą za ramię zaczepu, drugą zaś, połączoną z nią bezpośrednio, określającą obrót i położenie koła — kości te nie powinny przejmować zmian rotacji po swoim rodzicu, toteż należy przypisać im własność *Hinge*. Koło napędowe potrzebuje wyłącznie jednej kości, odpowiadającej za obrót, którą podczas animowania będzie też określać obrót wszystkich innych kół. Oczywiście nie ma sensu tworzyć wszystkich kości ręcznie — wystarczy jedna para, którą potem będziesz kopiował i centrował do kursora przesuwanego pomiędzy kolejnymi ramionami. Pamiętaj przy tym, że w wypadku każdego zaczepu-ramienia, kursor 3D musi znaleźć się dokładnie w tym samym miejscu (z perspektywy danego ramienia, rzecz jasna)! Po przygotowaniu wszystkich kości po jednej stronie pojazdu, można nadać im nazwy składające się z ich numeru (licząc od frontu czołgu), kolejności w parze (*U* od angielskiego

Up dla kości bazowej oraz *D* od angielskiego *Down* dla kości-dziecka) oraz (po kropce) inicjału angielskiego słowa, odpowiadającego lewej lub prawej stronie — *L* dla lewej (*left*) oraz *R* dla prawej (*Right*). Zapis taki, w formie na przykład *1U.R*, pozwala wykorzystać funkcje odnoszące się do stron w kośćcach. W naszym wypadku możemy skopiować armaturę, wykonać odbicie lustrzane, wszystkim kościom zmienić nazwy na odpowiadające „drugiej stronie" za pocą funkcji *Flip Left-Right Names* dostępnej z menu kontekstowego *Specials* [*W*] w trybie *Edit Mode* i scalić obie armatury — mimo dość długiego opisu procedura zajmuje kilka sekund, co jest dość miłą odmianą wobec monotonnego nazywania kolejnych kości dla każdego z kół. Należy jeszcze cały kościec uzależnić od kości *Main Sys Omega*, zanim przejdziemy do etapu następnego — a jest nim przygotowanie krzywej, na której „zawinięta" jest gąsienica. Krzywa ta musi zostać pocięta tak, by jej dolny fragment posiadał liczbę wierzchołków równą dwukrotnej liczbie kół nośnych w dolnej części pomniejszonej o jeden — innymi słowy, każde koło musi posiadać „swój" wierzchołek, dodatkowe wierzchołki muszą też znaleźć się pomiędzy każdą parą kół. W efekcie w dolnej części gąsienicy powinno znajdować się 13 wierzchołków. Każdy z nich należy następnie przyporządkować pojedynczemu kontrolerowi *Hook* — można to zrobić, zaznaczając kolejno każdy z 13 wierzchołków i z menu kontekstowego *Hooks* (*Ctrl+H*) wybierając *Add/To New Empty*. Zostanie stworzonych 13 dodatkowych obiektów *Empty*, które wykorzystamy podczas pracy nad ogranicznikami systemu trakcji, a na razie możemy je pominąć. Podobnie należy także przygotować drugą krzywą, odpowiadającą drugiej gąsienicy. System *Traction Sys* prezentuje rysunek 4.3.



Rysunek 4.3. *Armatura Traction Sys*

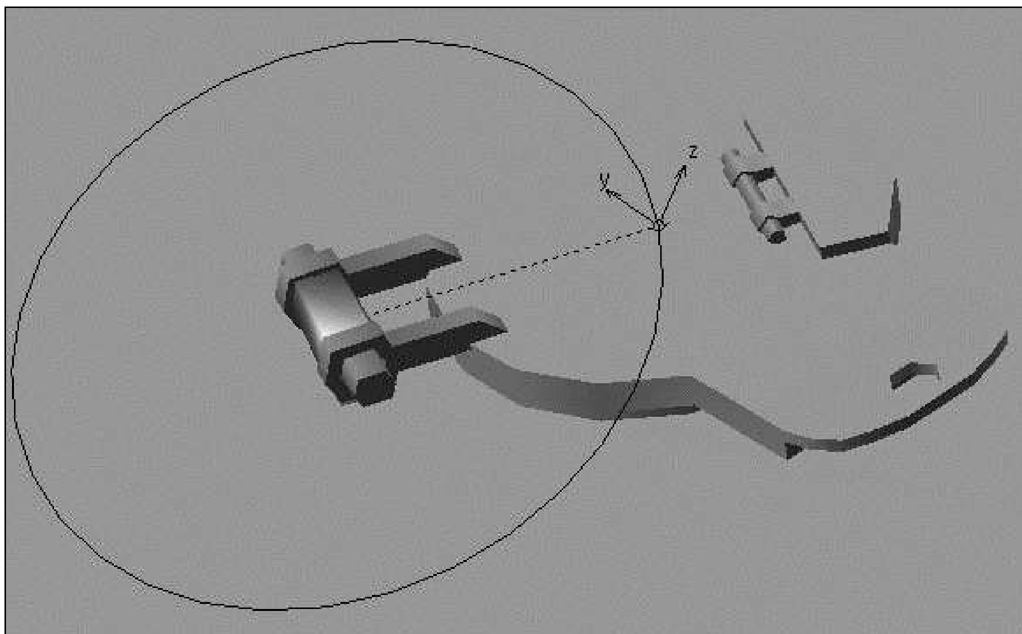
Część 4.2. Podstawowa funkcjonalność szkieletu

Teoretycznie szkielet jest już w pełni funkcjonalny — poza systemem trakcji, rzecz jasna. Wieża, karabiny, pokrywy włazów — wszystkimi tymi elementami można poruszać wedle woli. Ostatnie słowa nie mają jednak wydźwięku pozytywnego, wciąż jest bowiem możliwe takie modyfikowanie poszczególnych elementów, które absolutnie nie byłoby możliwe w rzeczywistości. Aby problem ten naprawić, potrzebne będą ograniczniki.

Zacznijmy od kośćca *Armanent Sys*. Wykonać należy tutaj serię ograniczeń. Przede wszystkim należy uniemożliwić przemieszczanie się wszystkich elementów z wyjątkiem działa, jemu zaś należy umożliwić jedynie niewielkie przesunięcie, mające odwzorowywać odrzut lufy po wystrzale. Stworzenie odpowiednich ograniczników jest w tym wypadku banalne — wystarczy, że w trybie *Pose Mode* każdej kości dodasz *Limit Location Constraint*, włączając wszystkie z podanych przycisków: *MinX*, *MinY*, *MinZ*, *MaxX*, *MaxY*, *MaxZ* i *Local*. Modyfikując wartości liczbowe przy każdym z przycisków, możesz ustalić pewien zakres ruchu dla kości, pamiętaj jednak, że wartości przyporządkowane przyciskom *Min* muszą być ujemne! W podobny sposób działa ogranicznik obrotu *Limit Rotation* — w jego wypadku należy zaznaczyć przyciski *LimitX*, *LimitY*, *LimitZ* oraz *Local*. Oczywiście ograniczników nie musisz dodawać pojedynczo — jeśli zaznaczysz wszystkie kości tak, by wyróżnić jedną z nich, kombinacją *Ctrl+C* uzyskasz dostęp do menu kontekstowego pozwalającego kopiować właściwości kości wyróżnionej do reszty zaznaczonych kości. Wybierając opcję *Constraints…* albo *Constraints (All)*, możesz skopiować także wybrane lub wszystkie ograniczniki. Dobrym pomysłem jest więc zdefiniowanie „pustych" ograniczników dla jednej kości i skopiowanie ich do pozostałych — potem zaś modyfikowane będą tylko poszczególne wartości. W wypadku kości *Turret* ograniczenie musi zakazywać obrotu na dwóch osiach, pozwalać zaś na pełen obrót względem osi Z, co jednak niekoniecznie musi oznaczać odznaczenie przycisku *LimitZ*. Kiedy kość jest ustawiona pionowo, będzie konieczne odznaczenie przycisku *LimitY*, jako że interesują nas osie lokalne, oś Y zaś biegnie wzdłuż kości i w tym wypadku pokrywa się z osią globalną Z. W wypadku określania ograniczeń, nie zaś zakazów, musimy jeszcze podać wartości maksymalnego obrotu bądź przesunięcia dla kości, także posługując się osiami lokalnymi. Jedyną kością wymagającą stworzenia ogranicznika (nie zaś zakazu) położenia jest oczywiście kość *Cannon*.

W wypadku kośćca *Main Sys* także należy przygotować kilka ograniczników, w tym jeden pozwalający na niewielki obrót (w zakresie do kilku stopni) kości *Corpse* — w animacji odpowiada ona za odwzorowanie bezwładności korpusu np. podczas gwałtownego hamowania. Kapy *P_FC* oraz *P_CR* należy jednak ograniczać nie tak, by działały na zasadzie zawiasów, lecz by dało się je „wyjąć" z ich wgłębień i „odłożyć" na bok. W wypadku klap *P_FR* oraz *P_FL* sytuacja

jest nieco bardziej skomplikowana — specyficzny system obsługiwania kości w Blenderze sprawia, że osie kości ustawionych pod kątem innym niż 0, 90, 180 lub 270° na globalnej osi Z… nie odpowiadają oczekiwaniom w nich pokładanych. Możemy więc zapomnieć o używaniu ograniczników czy rotowaniu kości w ogóle, chyba że w trybie *Pose Mode* obrócimy kości względem ich osi Y tak, by osie dopasować do naszych oczekiwań, potem zaś w trybie *Edit Mode* siatki *Mesh* korpusu odpowiednie kształty obrócimy „z powrotem". Innym sposobem jest wykorzystanie zewnętrznego rotatora. W takim wypadku tworzymy nowy okrąg *Mesh Circle*, który ustawiamy tak, by sam jego kształt odwzorowywał ścieżkę, po jakiej klapa powinna się poruszać. Można to zrobić, wykorzystując boczne powierzchnie zawiasów dla centrowania kursora 3D oraz jako koordynaty dla funkcji *Align View*. Następnie centrujemy kursor 3D do jednego z wierzchołków okręgu i w tym właśnie punkcie tworzymy obiekt *Empty*, który łączymy z okręgiem jako jego dziecko. Wracamy do kości *P_FR* (*P_FL*) i dodajemy nowy ogranicznik *Constraint — IK Solver*. W polu *Target* wpisujemy nazwę obiektu *Empty* (w moim wypadku jest to odpowiednio *P_FR Target* oraz *P_FL Target*) i wartość *ChainLen* ustawiamy na *1* — w przeciwnym razie nie tylko interesująca nas kość, ale także kości *Corpse* oraz *Omega* będą dostosowywane tak, by pokrywa znalazła się w odpowiednim miejscu! Wyłącz także przycisk *Stretch*, w końcu nie chcemy, by metalowy element rozciągał się jak guma. Po wykonaniu wszystkich tych operacji, obracając okrąg względem osi *Local Z*, możemy kontrolować stopień otwarcia klapy. Kontroler *P_FR* prezentuje rysunek 4.4.



Rysunek 4.4. *Kontroler P_FR pozwalający obejść specyficzne ograniczenia Blendera w zakresie określania osi lokalnych pojedynczych kości*

W wypadku *Traction Sys* sytuacja jest dużo bardziej skomplikowana. Na początek możemy zrobić to, co najprostsze — a więc zdefiniować i przypisać wszystkim „dolnym" kościom (zawierającym znacznik *D*) ograniczenie obrotu takie, które pozwoli im obracać się wyłącznie względem jednej osi. Ograniczenie położenia nie jest konieczne — tym zajmują się rodzice kości (wyjątkiem są oczywiście koła napędowe). Owi rodzice (kości *U*) przysporzą nam nieco więcej problemów — musimy ograniczyć stopień ich rotacji, a jednocześnie umożliwić sobie ich swobodną kontrolę. Najlepszym sposobem byłoby wykorzystanie ogranicznika *Track To* lub *IK Solver*, jednak kiedy tylko ogranicznik taki zaczyna działać, wszystkie ograniczniki rotacji i lokacji są ignorowane. Rozwiązaniem jest wykorzystanie ograniczników nie dla kości bazowych (*U*), ale dla celów do których będą się one odwoływać. Cele te muszą więc być kośćmi — obiektom *Empty* nie można niestety przypisywać ograniczników bazujących na przestrzeni lokalnej. Kości te należy stworzyć w miejscu łączenia się „górnych" (*U*) oraz „dolnych" (*D*) kości kół nośnych. Następnym etapem jest przypisanie nowo powstałym kościom ograniczników położenia, umożliwiających im niewielki ruch w górę oraz w dół, a także dodanie do „górnych" (*U*) kości kół nośnych ograniczników *Track To*, wykorzystujących jako cel jedną z nowo utworzonych kości — pamiętaj, że najpierw w polu *Target* musisz wpisać nazwę armatury, a więc *Traction Sys*! W ten sposób utworzony zostaje kontroler kąta pochylenia zaczepów na koła oraz poziomu kół, jednak daleko jeszcze do końca prac. W chwili obecnej możliwe jest symulowanie bezwładności pojazdu, jednak jeśli skorzystasz z tej możliwości, zaczepy kół oderwą się od korpusu — wynika to z faktu, że cała armatura kopiuje położenie i rotację kości *Main Sys Omega*, ale nie uwzględnia przesunięć kości *Corpse*! Musimy więc kolejne kości *U.L* (*U.R*) oraz kości kół napędowych (*9.L*, *9.R*) uzależnić od korpusu relacją rodzic – dziecko. Nie jest to możliwe bezpośrednio pomiędzy dwiema armaturami, my zaś nie chcemy ich łączyć — co zrobić w takiej sytuacji? W kośćcu *Traction Sys* utwórz kość *Corpse Controller*, położoną dokładnie w tym samym miejscu co kość *Main Sys Corpse*, nadając jej jednocześnie dla odróżnienia inny zwrot. Do kości tej dołącz relacją rodzic – dziecko kolejne kości *U.L* (*U.R*), następnie zaś dodaj trzy ograniczniki — *Copy Location*, *Copy Rotation* oraz *Limit Rotation*. Ostatni z ograniczników skopiuj tak, by kość *Corpse Controller* realizowała dokładnie taki sam zakres obrotu, co kość *Corpse*. Ogranicznik *Limit Rotation* jest wymagany ze względu na działanie tego typu ograniczników: jeśli ograniczymy kość do obrotu o wartość 45° i nadamy jej obrót 90°, kość „zapamięta" drugą wartość wizualnie zostanie zaś obrócona tylko o 45°. Odwołując się do rotacji takiej kości, otrzymujemy więc wartość 90° z pominięciem wpływu ogranicznika. Ograniczniki *Copy Location* oraz *Copy Rotation* należy skonfigurować tak, by wskazywały na armaturę *Coprse Sys* i jej kość *Corpse*, zaznaczając jednocześnie opcję *Local*. Tak skonfigurowana para kości pozwala przenieść modyfikację z kości w jednej armaturze do kości w drugiej armaturze tak, jakby były jedną kością. W tej chwili system powinien już działać dość efektownie — spróbuj „pobawić się" kością *Corpse*. Cały czołg powinien

zmieniać nieco swój przechył, jednak koła zareagują nieco odmiennie, starając się zrównoważyć obciążenie, a ich zaczepy same dostosują się do zaistniałej sytuacji. Do zakończenia pracy pozostała jeszcze najżmudniejsza część — wszystkim obiektom *Empty* kontrolującym położenie gąsienic należy utworzyć ograniczniki *Copy Location*, uzależniające je od kości odpowiadających im kół. Oczywiście efektem takiego działania będzie nagłe przeniesienie w przestrzeni haka, co z kolei skutkuje niechcianą modyfikacją krzywej. Aby temu zaradzić, wystarczy, że po zaznaczeniu krzywej w panelu *Modifiers* znajdziesz odpowiedni hak i wybierzesz opcję *Reset* — sprawi to, że obecne położenie i rotacja obiektu *Empty* staną się wartościami bazowymi. Do specyficznej sytuacji dochodzi, gdy edytujemy obiekt *Empty* ustawiony pomiędzy dwoma kołami — w takim wypadku dodajemy dwa ograniczniki, ustawiając je tak, by ich wpływ rozłożył się po połowie, ustawiając dla pierwszego (dolnego) parametr *Influence* na `0,5`, dla drugiego pozostawiając zaś *Influence* ustawione na `1`. Na pierwszy rzut oka może się to wydawać błędem — w końcu interesuje nas, by obiekt ten pobierał położenie zależnie od obu kości, na każdej opierając się jednakowo mocno, my jednak wprowadzamy różne wartości parametru *Influence* — dlaczego? Odpowiedź jest prosta — w obliczaniu siły wpływu Blender działa na wartościach względnych, nie zaś globalnych. Oznacza to, że parametr *Influence* drugorzędnego modyfikatora równy `100%` (`1,0`) oznacza nie 100% wpływu całkowitego, ale 100% wpływu pozostawionego przez pozostałe modyfikatory. A zatem sposób ustawienia modyfikatorów sprawia teraz, że pierwszy z nich wpływa na obiekt w 50%, drugi zaś w 100% pozostałych 50%, czyli po prostu w 50%. Także koła o numerach 1 i 9 wymagają indywidualnego podejścia — dla kości kontrolującej dane koło należy dodać pośrednik oraz nowy obiekt *Empty*, który posłuży jako hak. Pośrednik, o którym mowa, to kość, która stanie pomiędzy kośćmi *Corpse Controller* oraz kością odpowiadającą za koło — jest ona konieczna, by pozostawić sobie możliwość obracania kołem, inaczej obrót koła spowoduje też obrót krzywej! Kość ta musi być ustawiona w tym samym miejscu, co kość odpowiadająca kołu, należy ją połączyć relacją *Parent* z kością *Corpse Controller*, do niej zaś należy tą samą relacją połączyć kość odpowiadającą za obrót koła. Obiekt *Empty* musi być połączony z tym właśnie pośrednikiem za pomocą modyfikatora *Copy Location* oraz *Copy Rotation*. Do obiektu tego należy jeszcze dołączyć te wierzchołki krzywej, które nie zostały połączone z żadnym z pozostałych kół. W tym wypadku należy zaznaczyć najpierw obiekt *Empty*, potem zaś krzywą i w trybie *Edit Mode*, z zaznaczonymi odpowiednimi wierzchołkami, wybrać opcję *Add/To Selected Object* z menu kontekstowego *Hooks*. Istnieje też bardzo duże prawdopodobieństwo, że modyfikator *Copy Rotation* spowoduje błędne przekształcenie krzywych — w tym jednak wypadku wystarczy wykorzystać funkcję *Reset* dostępną w opcjach modyfikatora *Hook*. Możliwe też, że w dalszym ciągu obrót kości *Main Sys Omega* na jednej z osi spowoduje nieoczekiwane efekty, kiedy końcowe elementy krzywej nie obrócą się tak jak cała jej reszta. Wówczas należy modyfikator *Copy Rotation* zmienić tak, by odwrócić rotację na danych osiach — służą do tego przyciski „–”

przy przyciskach odpowiadających poszczególnym osiom. Inne z haków mogą „masowo" pobierać obrót z kości *Main Sys Omega* za pomocą relacji dziecko – rodzic, nie jest więc potrzebne żmudne definiowanie modyfikatorów *Copy Rotation*. Na koniec musimy zaradzić jeszcze jednemu problemowi — jeśli przechylisz czołg tak, by jedna gąsienica znalazła się poniżej drugiej, zauważysz, że gąsienice nie zmieniają swojej rotacji — co jest efektem co najmniej interesującym, niemniej jednak niechcianym. Aby problem zlikwidować, będziemy potrzebowali dwóch dodatkowych obiektów *Empty*, umieszczonych w środkach obu krzywych. Obiekty te, nazwane przeze mnie `LT` i `RT Driver` (*Left\Right Track Driver*) należy połączyć relacją rodzic – dziecko z kością *Main Sys Omega*, krzywymi oraz gąsienicami tak, by krzywe oraz gąsienice uzależnić od obiektu *Driver*, a sam obiekt od kości *Omega*. Zwróć uwagę, że bezpośrednie połączenie gąsienic z kością *Omega* nie przyniesie spodziewanych efektów! W idealnej sytuacji nasza praca kończy się w tym momencie, możliwe jednak, że do jej zakończenia potrzeba wciąż kilku drobnych modyfikacji. Weź też pod uwagę, że jeśli do stworzenia gąsienic wykorzystałeś krzywą *Bezier*, to wierzchołki w jej spodniej części, podłączone do poszczególnych kół nośnych, muszą zostać przeskalowane do zera! Inaczej podczas zmiany położenia kół krzywa może zostać wyrysowana w zły sposób! Ważne jest także, by krzywa kończyła się od strony przeciwnej niż ta z kołem napędowym — jeśli natomiast tak jest, należy odwrócić jej kierunek! Każda modyfikacja położenia kół zmienia nieco długość krzywej, a co za tym idzie, także sposób „nawinięcia" na nią gąsienicy — my natomiast nie możemy dopuścić, by precyzyjnie dopasowana do koła napędowego gąsienica nagle z niego „zjechała".

Część 4.3. Zaawansowana funkcjonalność szkieletu

4.3.1. Dualny system namierzania

System przypisany szkieletowi wieży można na pewno nazwać funkcjonalnym — w końcu pozwala wycelować zarówno działo jak i karabiny w dowolne miejsce na scenie. W tym właśnie tkwi problem — spróbuj teraz wycelować lufę czołgu w pewien daleko położony punkt. Nie jest to aż tak proste, jak być powinno, to pewne. Wyobraź sobie teraz rotację wieży czołgu podczas animacji — byłoby to niezmiernie czasochłonne i trudne, trzeba więc wyciągnąć asa z rękawa i rozbudować system celowniczy naszego czołgu.

Systemy namierzania używane w riggingu można podzielić na dwa rodzaje nazywane przeze mnie systemami pierwotnymi i wtórnymi. Systemy pierwotne to dokładnie to, co już zamontowaliśmy na naszym Abramsie — system celowania

zależny od kąta obrotu i uniesienia lufy podanych przez użytkownika programu. System ten cechuje się wysoką precyzją i prostotą w uzyskiwaniu obrotu o dokładnie założony kąt, stąd też ten właśnie system jest idealny, gdy chcemy obrócić wieżę do pozycji spoczynku. Jak już zauważyliśmy, nie jest on jednak najlepszą opcją, gdy chcemy mierzyć do jakiegoś konkretnego celu. W tej chwili na scenę wkraczają systemy wtórne — użytkownik podaje wyłącznie koordynaty celu, najczęściej przesuwając nań specjalnie do tego celu przygotowany celownik, wieża zaś sama przyjmuje taki obrót, by lufa wskazywała dokładnie zadany punkt. „Montaż" systemu wtórnego w Abramsie jest banalnie prosty — wystarczy, że kościom *Turret* i *Cannon* (lub *M240 H* i *V* oraz *M2HB H* i *V*) kośćca *Armanent Sys* przypiszemy modyfikatory *Locked Track*, dobierając odpowiednie osie *To* i *Loc*. Bardzo przydaje się włączenie opcji wyświetlania osi dla kości, dzięki czemu łatwiej jest określić, która z nich ma nakierowywać się na cel (*To*) oraz która ma pozostać niezmieniona (*Loc*). W polu *Target* należy wpisać nazwę obiektu *Empty* lub jego siatkowego odpowiednika — obiektu bez powierzchni *faces*, widocznego w strefie *3D View*, lecz niewidocznego na renderze. W moim wypadku nazwy poszczególnych „celowników" to *Cannon Target*, *M2HB Target* oraz *M240 Target*. Niestety, modyfikator *Track To* znosi ograniczenia obrotu, stąd też należy używać go ostrożnie! Możemy także wykorzystać małą sztuczkę, dzięki której zyskamy dostęp do obu systemów jednocześnie — starczy, że na osi obrotu wieży stworzymy nowy obiekt *Empty* lub (co jest lepszym rozwiązaniem) „pustą" siatkę — obiekt ten posłuży nam za kontroler, który należy połączyć relacją dziecko – rodzic z użytym wcześniej celem tak, by cel uzależnić od obrotu kontrolera. Dzięki temu, obracając kontroler, możemy kontrolować wieżę z użyciem systemu pierwotnego; przemieszczając cel, zyskujemy natomiast dostęp do systemu wtórnego. Sam kontroler należy też połączyć relacją rodzic – dziecko lub modyfikatorem *Copy Location* z kością *Main Sys Omega*. W pierwszym przypadku wieża czołgu będzie obracać się razem z nim, w drugim — pozostanie niezależna od obrotów korpusu. Dobrym rozwiązaniem jest też wykorzystanie modyfikatorów *Copy Location* i *Copy Rotation*. Modyfikując parametr *Influence* drugiego z nich, zyskamy możliwość kontrolowania sposobu zachowania się wieży. Niestety, opcja taka wymaga użycia kolejnego pośrednika, do którego przypiszemy ograniczniki, połączonego za pomocą modyfikatorów z kością *Main Sys Omega* oraz relacją rodzic – dziecko z kontrolerem wieży — modyfikator *Copy Rotation* uniemożliwia bowiem ręczną kontrolę rotacji!

Jako ciekawostkę wspomnę jeszcze, że w ramach zabawy udało mi się z pomocą modyfikatorów uzyskać dynamiczny obiekt nazwany przeze mnie „pełzaczem" — składał się on z krzywej oraz podróżującego po niej obiektu *Empty*. Oba te obiekty były skonfigurowane tak, że z początku obiekt *Empty* przemieszczał się na koniec krzywej, potem zaś krzywa przemieszczała się tak, by znaleźć się w miejscu wskazanym przez obiekt *Empty*. Powodowało to nieustanny ruch, przy kilku dodatkowych ogranicznikach pozwalający na automatyczne ograniczenie szybkości obracania się wieży — nawet jeśli przemieściliśmy cel z jednego końca

sceny na drugi, mijał pewien czas, zanim wieża była w stanie weń wycelować. Ostatecznie jednak pełzacz okazał się bardzo problematyczny — bazował w końcu na niedoskonałości architektury programów komputerowych analizujących dane w pewnej kolejności, w przeciwnym razie stworzenie dwóch obiektów nawzajem zmieniających swoje położenie nie byłoby możliwe.

Jeśli użyjesz modyfikatorów Python, wymagających oczywiście znajomości tego języka, możesz znacznie udoskonalić system celowniczy. „Pełzacza" można więc zastąpić znacznie mniej zawodnym systemem, wtórny system celowania rozbudować zaś tak, by brał pod uwagę trajektorię pocisku, szybkość poruszania się czołgu czy nawet wcześniej zdefiniowany wiatr... Teoretycznie jest nawet możliwe odwzorowanie komputera balistycznego zamontowanego w prawdziwych czołgach M1!

4.3.2. Animacja ruchu gąsienic

Posiadamy więc pierwszej jakości system kontroli gąsienic — nie posiadamy jednak możliwości poruszania nimi. Błąd ten należy oczywiście czym prędzej naprawić. Wykorzystamy w tym celu pewną sztuczkę — chodzi mianowicie o to, że jeśli przemieścimy na odpowiedniej osi obiekt powielony na krzywej modyfikatorami *Array* i *Curve*, efekt tego powielenia również będzie podlegać przesunięciu.

Jednak na początek rzeczy pierwsze — musimy stworzyć dwa nowe, dość nietypowe kośćce, składające się z pojedynczych kości. Odpowiednio dla lewej i prawej gąsienicy będą to *Track Sys.L* oraz *Track Sys.R*, kości zaś określiłem mianem *Controler*. Do kości tych należy przyłączyć ogniwo gąsienicy (obiekt, nie siatkę), używając w tym celu relacji rodzic – dziecko (*Make Parent to Bone*), nie zaś typowej zależności *Armature*! Oba kośćce należy z kolei połączyć z odpowiadającymi im krzywymi gąsienic, używając w tym celu modyfikatorów *Copy Location* oraz *Copy Roatation* — relacja typu *Parent* nie zadziała, jako że modyfikator *Curve* wykorzystany dla członu gąsienicy i wykorzystujący tę właśnie krzywą jako cel wymaga, by była ona interpretowana jako ścieżka. W takim zaś wypadku dowolny obiekt połączony z krzywą relacją rodzic – dziecko będzie się przemieszczał wzdłuż tejże ścieżki — co jest oczywiście efektem jak najbardziej niepożądanym w naszej sytuacji.

Czas zająć się częścią nieco bardziej skomplikowaną — dla każdego z kośćców musimy dodać nową akcję, prezentującą przesunięcie się gąsienicy o jedną jednostkę. Pisząc „jedną jednostkę" mam na myśli tyle członów gąsienicy, ile stanowi podstawowy obiekt, pomijając powtórzenia wynikające z działania modyfikatora *Array*. Dlatego też, jeśli za pomocą wspomnianego modyfikatora na całej długości gąsienicy powtarzany jest jeden człon, jego długość będzie jedną jednostką, jeśli cztery człony — jednostka będzie równa długości czterech członów. Konieczność określenia wielkości takiej jednostki wynika z faktu,

że nie będziemy przemieszczać całej gąsienicy, to bowiem wymagałoby systemu tak skomplikowanego, by każdy człon gąsienicy był osobnym elementem, którego położenie określałby jego własny, miniaturowy kościec. Oczywiście stworzenie takiego systemu byłoby możliwe, jednak eksperymenty pozostawiam Tobie — my zaś będziemy „oszukiwać", wykonując za każdym razem to samo przejście gąsienicy. W praktyce będzie to wyglądać tak, jakby po przesunięciu się o jedną jednostkę gąsienica teleportowała się z powrotem do pozycji bazowej i powtarzała swój ruch. Rzecz jasna widz nie będzie miał szans tej teleportacji zauważyć — gąsienica jest przecież powtarzalna, a więc jej nagłe przesunięcie się z powrotem do pozycji bazowej będzie wyglądało zupełnie tak samo jak kontynuacja ruchu.

Wracając jednak do części praktycznej — kości *Controler* należy nadać dwa klucze *IPO Loc*, jeden z nich w pierwszej klatce w jej pozycji bazowej, drugi zaś w klatce dziesiątej po przesunięciu kości o odpowiednią wartość. Wartość tę, stanowiącą długość jednej jednostki, można uzyskać w bardzo prosty sposób — jest przecież wpisana w jednym z pól *Constant Offset* modyfikatora *Array* członu gąsienicy! Także oś, pod którą owa wartość widnieje, ma znaczenie — to właśnie na niej przesunięcie powinno się odbyć. W oknie *Action Editor* możemy teraz nadać takiej akcji odpowiednią nazwę, na przykład *Move*. Weź pod uwagę, że wystarczy nam jedna akcja dla obu gąsienic — w końcu kości kontrolujące przesunięcie w obu kośćcach mają tą samą nazwę. Należy jeszcze wybrać odpowiedni rodzaj interpolacji — obecnie ruch nie jest jednostajny, lecz przyśpiesza z początku i zwalnia na końcu trasy. Gdyby taką akcję wykorzystać do „napędu" czołgu, cała gąsienica poruszałaby się ze zmienną prędkością, co poza fatalnym i nierealistycznym wyglądem stanowiłoby koszmar synchronizacji ruchu gąsienicy i kół. Dlatego też należy zaznaczyć oba klucze i z rozwijanego menu *Key* panelu narzędziowego okna *Action Editor* wybrać podmenu *Interpolation Mode*, tam zaś wskazać na interpolację liniową (*Linear*). Na koniec usuń akcję *Move* — nie zostanie ona oczywiście całkowicie usunięta, a jedynie wyłączona, dzięki czemu nie będzie wpływać na przemieszczenie gąsienic, dopóki sami jej nie wywołamy.

Kolejnym elementem będzie uzależnienie przemieszczenia członu gąsienicy od kontrolera. Nie będziemy jednak (jeszcze) tworzyć żadnej zewnętrznej kontrolki — wykorzystamy inną metodę, która pozwoli nam zgrabnie zamknąć kwestię przesuwania gąsienicy w innym mechanizmie. Sztuczka, o której mowa, polega na uzależnieniu ruchu gąsienicy od koła napędowego — co jest oczywiście odwzorowaniem realnego stanu rzeczy, jednak stanowi to raczej efekt uboczny. Najważniejszą zaletą takiego podejścia jest całkowite wyeliminowanie problemu synchronizacji pomiędzy kołami a gąsienicą — raz przeprowadzona przestanie być istotna! Aby taką synchronizację przeprowadzić, trzeba najpierw akcję *Move* kości *Track Sys Controler* uzależnić od obrotu koła napędowego. W tym celu do kości tej dodajemy ogranicznik *Action*, w pole *OB:* wpisując nazwę kośćca — `Traction Sys`, w pole *BO:* — `9.L` (`9.R`), w pole *AC:* — `Move`, pola *Start* i *End* uzupełniamy

za pomocą wartości `1` i `9`, a w pola *Min* i *Max* wprowadzamy wartości `-180` i `180`. Pierwszym zaskoczeniem może być wartość `9` w polu *End* — przecież akcja kończyła się w dziesiątej klatce? Jest to oczywiście prawda, jednak jej pierwsza i ostatnia klatka są identyczne, dlatego też należy jedną z nich pominąć. Wartości *Min* i *Max* odpowiadają oczywiście wartości obrotu, na jaki reagować będzie akcja — nas interesuje oczywiście, by dowolny obrót koła aktywował przemieszczenie gąsienicy. Zaznacz jeszcze przycisk *Local* i wybierz oś, która odpowiada obrotowi koła. Następnym krokiem jest przygotowanie synchronizacji — obecnie pełen obrót koła powoduje przejście jednego członu gąsienicy, co jest oczywiście błędem, ponieważ przemieszczenie powinno wynosić 11 członów. Wystarczy jednak zmodyfikować akcję *Move* tak, by realizowane przez nią przemieszczenie dotyczyło właśnie 11 członów gąsienicy. Wartość takiego przemieszczenia obliczysz na podstawie prostego wzoru: *11*(A/C)*, gdzie *C* oznacza liczbę członów w jednostce długości, *A* oznacza zaś wartość przesunięcia wpisaną w modyfikator *Array*. Na koniec należy jeszcze uzależnić motorykę pozostałych kół od koła napędowego — wystarczy skopiować jego rotację za pomocą modyfikatorów *Copy Rotation*.

Tak przygotowany system trakcji jest już praktycznie gotów do użycia — można go oczywiście dalej rozbudować, dodając zewnętrzne kontrolery rotacji kół, a nawet wykorzystując kolejne akcje dla zautomatyzowania animacji gąsienic podczas poruszania się czołgu po ścieżce — to jednak pozostawiam już Twojej inwencji, jako że w porównaniu do samego systemu trakcji będzie to dziecinnie proste.