Witold Jaworski



Programowanie dodatków do Blendera 2.5

Pisanie skryptów w języku Python, z wykorzystaniem Eclipse IDE

wersja 1.01

Programowanie dodatków do Blendera 2.5 - wersja 1.01

Copyright Witold Jaworski, 2011.

wjaworski@samoloty3d.pl http://www.samoloty3d.pl

Recenzent: Jarek Karpiel

Chciałbym także podziękować Dawidowi Ośródce, za jego uwagi.



Ten utwór jest dostępny na <u>licencji Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów</u> zależnych 3.0 Unported.

ISBN: 978-83-931754-1-3

Spis treści

Spis tr	eści	3
Wprow	vadzenie	4
Konwe	ncje zapisu	5
Przygoto	wania	6
Rozdz	ał 1. Instalacja oprogramowania	7
1.1	Python	8
1.2	Eclipse	10
1.3	PyDev	13
Rozdz	ał 2. Pierwsze kroki w Eclipse	17
2.1	Rozpoczęcie projektu	
2.2	Uruchomienie najprostszego skryptu	24
2.3	Debugowanie	27
Tworzen	ie aplikacji Blendera	
Rozdz	ał 3. Skrypt dla Blendera	
3.1	Sformułowanie problemu	
3.2	Dostosowanie Eclipse do API Blendera	
3.3	Opracowanie podstawowego kodu	48
3.4	Uruchamianie skryptu w Blenderze	
3.5	Rozbudowa skryptu: wykorzystanie poleceń Blendera	65
Rozdz	ał 4. Przerabianie skryptu na wtyczkę Blendera (add-on)	74
4.1	Dostosowanie struktury skryptu	75
4.2	Dodanie polecenia (operatora) do menu	
4.3	Implementacja interakcji z użytkownikiem	
Dodatki.		
Rozdz	ał 5. Szczegóły instalacji	
5.1	Szczegóły instalacji Pythona	
5.2	Szczegóły instalacji Eclipse i PyDev	
5.3	Konfiguracja PyDev	
Rozdz	ał 6. Inne	
6.1	Aktualizacja nagłówków API Blendera dla PyDev	
6.2	Importowanie pliku do projektu PyDev	
6.3	Debugowanie skryptu w Blenderze — szczegóły	
6.4	Co się kryje w pliku <i>pydev_debug.py</i> ?	
6.5	Pełen kod wtyczki mesh_bevel.py	
Bibliogra	fia	

Wprowadzenie

Językiem skryptów Blendera jest Python. W tym języku zrealizowano wiele przydatnych dodatków do tego programu. Niestety, w Blenderze brakuje czegoś w rodzaju zintegrowanego środowiska programisty (ang. *integrated development environment* — w skrócie *IDE*). "W standardzie" znajdziesz tylko zaadaptowany do podświetlania składni Pythona edytor tekstowy, oraz konsolę. To wystarcza do tworzenia prostych skryptów, ale zaczyna przeszkadzać, gdy tworzysz większe programy. Szczególnie uciążliwy jest brak "okienkowego" debuggera. W 2007r opracowałem artykuł, w którym proponowałem użycie w tym charakterze dwóch programów: Open Source: **SPE** (edytor) i **Winpdb** (debugger). Ten artykuł został potem opublikowany na *blenderwiki* przez Jeffa Blanka.

W 2009r postanowiono, że przepisywana "od podstaw" nowa wersja Blendera (2.5) będzie miała zupełnie nowe API. Ten Blender ma "wczepionego" Pythona w wydaniu 3.x, podczas gdy poprzednie wersje używały wydań z serii 2.x. W dodatku twórcy Pythona także zdecydowali się w wersji 3.x zerwać wsteczną zgodność kodu. W efekcie biblioteka "okienek", w której napisano SPE i Winpdb — **wxPython**, oparta na **wxWidgets** — działała w Pythonie 2.x, a nie działa w Pythonie 3.x. Co gorsza, jakoś nikt nie pracuje nad jej aktualizacją. Wygląda na to, że w ten sposób obydwa narzędzia stały się niedostępne dla Blendera w wersji 2.5 i następnych (2.6, ...).

Postanowiłem więc zaproponować nowe środowisko programisty, także oparte wyłącznie o oprogramowanie Open Source. Tym razem mój wybór padł na IDE **Eclipse**, wzbogacone o dodatek do pracy ze skryptami Pythona: **PyDev**. Obydwa produkty są rozwijane już od 10 lat, i same w sobie nie zależą od Pythona. (Dzięki temu nie są narażone na taką wsteczną niezgodność kodu, jak SPE i Winpdb). Nim napisałem to opracowanie, zaadaptowałem do API Blendera 2.5 za pomocą Eclipse i PyDev wszystkie moje skrypty. W niektórych przypadkach oznaczało to konieczność przepisania ich od nowa. Na podstawie tych doświadczeń sądzę, że to nowe środowisko jest lepsze od poprzedniego.

Uważam, że narzędzia programisty najlepiej przedstawiać na przykładzie pracy nad jakimś konkretnym skryptem. Zdecydowałem się więc opisać tu proces tworzenia wtyczki Blendera, służącej do fazowania wybranych krawędzi siatki. (Chodzi o odtworzenie działania polecenia *Bevel* z Blendera 2.49). Poziom narracji wymaga przeciętnej znajomości Pythona i Blendera. Do zrozumienia fragmentu o tworzeniu wtyczki (Rozdział 4) trzeba także znać podstawowe pojęcia programowania obiektowego, takie jak: "klasa", "obiekt", "instancja", "dziedziczenie". Gdy jest to potrzebne (pod koniec tego rozdziału) wyjaśniam kilka bardziej zaawansowanych pojęć. (Na przykładzie API dla wtyczek tłumaczę, co to jest "interfejs" i "klasa abstrakcyjna"). Ta książka wprowadza w praktyczne podstawy pisania rozszerzeń Blendera. Nie opisuję tu wszystkich zagadnień. Przestawiam za to metody, które stosuję, by je poznawać. Używając ich, będziesz mógł samodzielnie opanować resztę tego API (np. tworzenie własnych paneli lub menu).

Konwencje zapisu

Wskazówki dotyczące klawiatury i myszki oparłem na założeniu, że masz standardowe:

- klawiaturę w normalnym układzie amerykańskim, 102 klawisze;
- myszkę wyposażoną w dwa przyciski i kółko przewijania (które daje się także naciskać: wtedy działa jak trzeci, środkowy przycisk).

Wywołanie polecenia programu będę zaznaczał następująco:

Menu → Polecenie	- taki zapis oznacza wywołanie z menu "Menu" polecenia "Polecenie". W przypadku
	bardziej zagnieżdżonych menu może wystąpić więcej strzałek!

Panel:Przycisk - taki zapis oznacza naciśnięcie w oknie dialogowym lub panelu "Panel" przycisku "Przycisk".

Naciśnięcie klawisza na klawiaturze:

Alt-K	- myślnik pomiędzy znakami klawiszy oznacza jednoczesne naciśnięcie obydwu klawi-
	szy na klawiaturze. W tym przykładzie trzymając wciśnięty Alt, naciskasz K;
G , X	- przecinek pomiędzy znakami klawiszy oznacza, że je naciskasz (i zwalniasz!) po kolei.

W tym przykładzie najpierw G, a potem X (tak, jak gdybyś chciał napisać wyraz "gx").

Naciśnięcie klawisza myszki:

LPM	- lewy przycisk myszy
PPM	- prawy przycisk myszy
SPM	 środkowy przycisk myszy (naciśnięte kółko przewijania)
KM	- kółko przewijania (pełni tę rolę, gdy jest obracane)

Na koniec: jak mam się do Ciebie zwracać? Zazwyczaj w poradnikach używa się formy bezosobowej ("teraz należy zrobić"). To jednak, mówiąc szczerze, czyni czytany tekst mniej zrozumiałym. Aby ta książka była jak najbardziej czytelna, zwracam się do Czytelnika w krótkiej, drugiej osobie ("teraz zrób"). Czasami używam także osoby pierwszej ("teraz zrobiłem", "teraz zrobimy"). Tak jest mi łatwiej. Podczas pisania i debugowania kodu w tym opracowaniu traktowałem nas — czyli Ciebie, drogi Czytelniku, i siebie, piszącego te słowa — jako jeden zespół. Może trochę wyimaginowany, ale w jakiś sposób prawdziwy. Przecież pisząc tę książkę ja także się wiele nauczyłem, bo wiedziałem, że każde zagadnienie mam Ci porządnie przedstawić!

Przygotowania

6

W tej części opisuję, jak zbudować (zainstalować) odpowiednie środowisko programisty (Rozdział 1). Potem zaznajamiam z podstawami konfiguracji i użycia IDE Eclipse, z dodatkiem PyDev (Rozdział 2).

Rozdział 1. Instalacja oprogramowania

Opisywane w tej książce środowisko pracy programisty wymaga zainstalowania trzech składników:

- "zwykłego" interpretera Pythona (jest to interpreter "zewnętrzny", w stosunku do dostarczanego wraz z Benderem interpretera "osadzonego" w kodzie programu);
- jakiejś odmiany Eclipse;
- dodatku do Eclipse: PyDev;

Ten rozdział opisuje, jak to zrobić.

Zakładam, że masz już zainstalowany Blender. (Podczas pisania tej książki używałem Blendera 2.57. Oczywiście, możesz wykorzystać jakiejkolwiek z jego późniejszych wersji).

1.1 Python

Najpierw sprawdź, której wersji Pythona używa Twój Blender. W tym celu przełącz któryś z jego edytorów na *Python Console*, i odczytaj z jej nagłówka wersję Pythona (Rysunek 1.1.1):

Console Autoco	omplete				
PYTHON INTERACTIVE CO	DNSOLE 3.2 (r32:88445, Mar 4	4 2011,	02:13:51)	[MSC v.1500]	32 b
Cursor:	Left/Right Home/End				
Remove: Execute:	Backspace/Delete Enter				
Autocom Ctrl +/. Python Console	Ctrl+Space				
Builtin Modules: blf, mathutils	bpy, bpy.data, bpy.o Odczytaj Pythona	wersję <mark>s,</mark>	<pre>bpy.types,</pre>	<pre>bpy.context</pre>	, bp
Convenience Imports:	<pre>from mathutils import *; from the second secon</pre>	om math	import *		
/>>>					

Rysunek 1.1.1 Identyfikacja wersji Pythona, używanej przez Twojego Blendera

Blender z ilustracji powyżej (to wersja 2.57) wykorzystuje Pythona 3.2. Zazwyczaj warto instalować jako zewnętrzny program tę samą wersję.

"Zewnętrzny" interpreter Pythona można pobrać z <u>www.python.org</u> (Rysunek 1.1.2):



Rysunek 1.1.2 Pobieranie instalatora Pythona (z http://www.python.org/download)

Jak widzisz, ze względu na brak wstecznej zgodności Python ma obecnie dwie gałęzie rozwoju: 2.x i 3.x. Nas będzie interesować tylko ta druga.

 Nim pobierzesz zewnętrzny interpreter Pythona, nie zaszkodzi sprawdzić, czy przypadkiem nie masz go już zainstalowanego na swoim komputerze. Spróbuj wywołać w linii poleceń następujący program: python --version

Jeżeli masz zainstalowanego Pythona, uruchomi Ci się jego konsola, (taka, jaką pokazuje Rysunek 1.1.1).

W razie czego nie przejmuj się jednak specjalnie drobniejszymi różnicami w numerach wersji Pythona. Może się zdarzyć, że na stronach <u>www.python.org</u> nie znajdziesz dokładnie tej samej wersji, która jest skompilowana z Benderem. (Chodzi mi o różnicę w numerze "po kropce"). Wybierz wówczas wersję o najbliższym wyższym numerze. Blender 2.5 używa wyłącznie swojego "wewnętrznego" interpretera, nawet gdy w systemie jest dostępny taki sam, ale zewnętrzny. Tak więc zazwyczaj wszystko będzie działać, gdy np. zainstalujesz sobie jako zewnętrzny interpreter Pythona w wersji 3.3, zamiast wersji 3.2.

Pobrany ze strony program instalacyjny uruchom tak, jak każdy inny instalator pod Windows (Rysunek 1.1.3). (Musisz mieć tylko uprawnienia Administratora do swojego komputera):

🛃 Python 2.5.2 Setup	X	🛱 Python 2.5.2 Setup
	Customize Python 2.5.2 Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.	Install Python 2.5.2 Please wait while the Installer installs Python 2.5.2. This may take several minutes. Status:
python windows	Python Interpreter and Libraries This feature requires 18MB on your hard drive. It has 5 of 5 subfeatures selected. The subfeatures require 18MB on your hard drive.	
Disk Usage Advanced	< Back Next > Cancel	< Back Next > Cancel

Rysunek 1.1.3 Wybrane ekrany instalacji interpretera Pythona (te pochodzą z wersji 2.5, ale nowsze są takie same)

Wystarczy, jeżeli będziesz w trakcie całego procesu udzielał tylko odpowiedzi domyślnych/potwierdzających. (Szczegółowy opis instalacji — zobacz rozdział 5.1, str. 100).

1.2 Eclipse

Przejdź na stronę, z której można pobrać różne wersje Eclipse (<u>www.eclipse.org/downloads</u> — Rysunek 1.2.1):

Eclipse Downloads - Program Windows Internet Explorer dostarczony przez			
	Http://www. eclipse.org /downloads/		
🚖 Ulubione	Eclipse Downloads		
G+ Ecl	ipse IDE for C/C++ Developers, 87 MB nloaded 408,951 Times Details		
Bown	ipse for PHP Developers , 141 MB nloaded 234,703 Times Details		

Rysunek 1.2.1 Pobieranie Eclipse ze strony tego projektu (ekran z marca 2011)

Gdy przyjrzysz się opisom, zauważysz że to środowisko pracy jest udostępniane w wielu różnych odmianach. Każda z nich jest przygotowana dla określonego języka/języków programowania. (Nie oznacza to, że nie możesz np. w wersji dla PHP tworzyć programu w C++. Wystarczy dograć odpowiednie wtyczki! To, co widać, to po prostu zawczasu przygotowane, typowe "zestawy wtyczek". Odpowiadają najczęściej występującym potrzebom. Nie ma tu "gotowego" zestawu dla Pythona, więc proponuję pobrać *Eclipse for Testers* lub *Eclipse IDE for C/C++*. (Wybieraj zestaw, który ma najmniejszy rozmiar i najmniej specyficznych dodatków). Szczegóły instalacji Eclipse znajdziesz na str. 103.

Twórcy Eclipse nie uznają instalatorów Windows (chwała im za to!) i dostarczają spakowany folder z gotowym do użycia programem. Wystarczy go rozpakować — np. do *Program Files* (Rysunek 1.2.2):



Rysunek 1.2.2 "Instalacja" - czyli po prostu rozpakowanie Eclipse

Po rozpakowaniu możesz dodać skrót do głównego programu na pulpit albo do menu (Rysunek 1.2.3):



Rysunek 1.2.3 Program główny, uruchomiający Eclipse

 Eclipse do działania potrzebuje wirtualnej maszyny Javy. Najprawdopodobniej masz ją już zainstalowaną na swoim komputerze. Jeżeli nie — pobierz ją z <u>www.java.com</u> (szczegóły — p. str. 103).

Po uruchomieniu *eclipse.exe* program wyświetli okno, w którym możesz określić folder dla przyszłych projektów. W Eclipse nazywa się to "przestrzenią roboczą" (*workspace* — por. Rysunek 1.2.4):

🖨 Workspace Launcher	
Select a workspace Eclipse stores your projects in a folder called a workspace. Choose a workspace folder to use for this session.	Indywidualny profil użytkownika. (Uważaj domyślnie to nie są <i>Moje Dokumenty</i> !)
Workspace: C:\Documents and Settings\w4979721\workspace	Browse
Gdy nie chcesz tej lokalizacji później zmieniać, tu możesz wyłączyć to pytanie	OK Cancel



Na projekt składa się parę własnych plików Eclipse, oraz Twoje skrypty. W razie czego, jeżeli skrypt ma być w innym miejscu na dysku, będziesz mógł tu umieścić tylko jego skrót. Każda przestrzeń robocza zawiera, oprócz folderów projektów, własny zestaw preferencji Eclipse, m.in. konfigurację interpretera Pythona. Zwróć uwagę, że domyślnie folder *workspace* jest umieszczony w katalogu głównym profilu użytkownika. (W tym przykładzie to użytkownik *W4979721*). To wcale nie są *Moje Dokumenty* — tylko poziom wyżej. To proste przełożenie konwencji z *Unix/Linux*. Jeżeli jesteś przyzwyczajony, że wszystkie swoje dane trzymasz w *Moje Dokumenty* — zmień tę ścieżkę. Eclipse utworzy odpowiedni folder na dysku. Zazwyczaj do pracy wystarczy Ci jedna przestrzeń robocza.

Potem może się pojawić komunikat **Unable to Launch** (por. sekcja 5.2, Rysunek 5.2.5). Nie przejmuj się nim! Eclipse zawsze stara się otworzyć Twój ostatni projekt, zapisany we wskazanym folderze. Jednak przy pierwszym uruchomieniu tam niczego nie ma, stąd ta wiadomość. Przy pierwszym uruchomieniu okno Eclipse wyświetla ekran *Welcome*, ze skrótami do kilku miejsc w Internecie, związanych z tym środowiskiem (Rysunek 1.2.5):



Rysunek 1.2.5 Wygląd Eclipse przy pierwszym uruchomieniu

Teraz musimy dodać do Eclipse wtyczkę, dostosowującą to środowisko do skryptów Pythona: PyDev.

1.3 PyDev

•

Do instalacji PyDev wykorzystamy wewnętrzny mechanizm Eclipse, przeznaczony do obsługi wtyczek.

UWAGA: aby wykonać opisane w tej sekcji czynności, musisz być podłączony do Internetu

Nowe wtyczki dodaje się do środowiska poleceniem Help→Install New Software (Rysunek 1.3.1):

Help		
🚳 Welcome		
(?) Help Contents		
💯 Search		
Dynamic Help		
Key Assist	Ctrl+Shift+L	
Tips and Tricks		
Report Bug or Enhancement Cheat Sheets	Dod do ś	∣ laje nowe komponenty srodowiska Eclipse
Check for Updates		
Install New Software		
Eclipse Market Aace		
About Eclipse		

Rysunek 1.3.1 Polecenie dodające do środowiska nową wtyczkę

W oknie, które się pojawi, wpisz następujący adres: http://pydev.org/updates (Rysunek 1.3.2):

🖨 Install	
Available Software Select a site or enter the location of a site.	Tu wpisz adres projektu PyDev
Work with: ⁰ http://pydev.org/updates Find more softw	Add
type filter text Name	i naciśnij ten przycisk! Version
	Add Repository
Loca	ation: http://pydev.org/updates Archive

Rysunek 1.3.2 Dodawanie nowego miejsca do okna aktualizacji dodatków

Następnie naciśnij przycisk Add. Spowoduje to otwarcie okna Add Repository, które po prostu zatwierdź.

Po chwili Eclipse odczyta i wyświetli zawartość tego repozytorium (Rysunek 1.3.3):

🖨 Install					
Available Software					
Check the items that you wish to install.					
Work with: http://pydev.org/updates				*	<u>A</u> dd
	Find more softwa	re by working w	ith the <u>"Availat</u>	le Software Sit	<u>es"</u> preferences.
type filter text	Wybierz ten				
Name	komponent		Version		
₩ 🔽 000 PyDev					
🗄 🔄 🚥 PyDev Mylyn Integration (opti	onal)				
Select All Deselect All 1 iter	m selected				
 _ Details					
Show only the latest versions of availab	le software	Hide items	that are alread	ly installed	
Group items by category		What is alrea	ady installed?		
Contact all undate sites during install to find required software					
	nina rodan oa sore				
	i naciśnij Next				
?		< <u>B</u> ack	Next >	Einish	Cancel

Rysunek 1.3.3 Wybór PyDev do zainstalowania

Wybierz z niego komponent PyDev i naciśnij *Next*. Po przejściu następnych ekranów (wyświetlenie dokładnej listy pobieranego oprogramowania oraz umowy licencyjnej — por. sekcja 5.2, str. 103), pojawi się okno postępu instalacji (Rysunek 1.3.4):

🖨 Installing Software	
Installing Software	
Downloading org.eclipse.jdt.de	bug
Always run in background	k,
	Run in Background Cancel Details >>

Rysunek 1.3.4 Postęp pobierania dodatku

Po pobraniu mogą się pojawić dodatkowe pytania o potwierdzenie certyfikatów dla Aptana¹ PyDev. Na koniec pojawi się informacja o konieczności restartu Eclipse (Rysunek 1.3.5):



Rysunek 1.3.5 Okno końcowe

Na wszelki wypadek warto go wykonać.

Dla każdej przestrzeni roboczej (*workspace* — por. Rysunek 1.2.4) Eclipse zapisuje oddzielną konfigurację. Wśród tych parametrów jest też domyślny interpreter Pythona. Warto go od razu ustawić. W tym celu wywołaj polecenie *Window* → *Preferences* (Rysunek 1.3.6):



Rysunek 1.3.6 Przejście do parametrów przestrzeni roboczej (workspce)

¹ PyDev stworzył w 2003 Aleks Totic. Od 2005 roku projekt przejął Fabio Zadrozny. Projekt składał się z części Open Source — PyDev i komercyjnej — PyDev Extensions (zdalny debugger, analiza kodu, itp.). W 2008r PyDev Extensions zostały nabyte przez firmę Aptana. W 2009r Aptana "uwolniła" PyDev Extensions, łącząc je z PyDev (wersji 1.5). W lutym 2011 Aptana została nabyta przez Appcelerator. Portal PyDev jest nadal na serwerach Aptany/Appceleratora, a Fabio Zadrozny przez cały czas czuwa nad jego rozwojem.

W oknie Preferences rozwiń sekcję PyDev i podświetl pozycję Interpreter - Python (Rysunek 1.3.7):

Preferences	
type filter text	Python Interpreters $\Leftrightarrow \circ \Rightarrow \bullet \bullet \bullet$
🖅 General	Python interpreters (e.g.: python.exe)
i Heip ⊡ Tostall/Liodate	Podświetl ten i naciśnij New
inistanjopuate ini∋ilava	element ten przycisk
Builders	Remove
🖅 Debug	
Interactive Console	Down
Interpreter - Iron Python	
Interpreter - Bython	🛋 Libraries 🛛 Forced Builtins 🛛 Predefined 🛛 🌆 Environment 🛛 🗢 String Substitution Variables
Logging	System PYTHONPATH
PyLint	Nau Ealdar
PyUnit	New Folder
Scripting PyDev	New Egg/Zip(s)
Task Tags	
'≝ Run/Debug	Remove

Rysunek 1.3.7 Wywołanie automatycznej konfiguracji Pythona

Następnie wystarczy nacisnąć przycisk *Auto Config*. Jeżeli ścieżka do Twojego Pythona znajduje się w zmiennej środowiskowej (Windows) *PATH*, to Eclipse odnajdzie i skonfiguruje ten interpreter (Rysunek 1.3.8):

Preferences				
type filter text	Python Interpreters $(\neg \neg \neg$			
⊡- General	Python interpreters (e.g.: python.exe)			
≣ ·· Help	Name Location New			
 Install/Update Java PyDev 	Python 3.2 C:\Program Files\Python32\python.exe			
- Builders - Debug - Editor - Interactive Console - Interpreter - Iron Python	Skonfigurowany interpreter Pythona			
Interpreter - Jython Interpreter - Python Logging	Libraries Forced Builtins Predefined To Environment String Substitution Variables System PYTHONPATH			
PyLint PyUnit Scripting PyDev Task Tags Run/Debug Tasks	System libs New Folder Image: C:\Program Files\Python32 New Egg/Zip(s) Image: C:\Program Files\Python32\lib Remove Image: C:\Program Files\Python32\lib\site-packages Remove			
< >>	Restore Defaults Apply Naciśnij OK , by zatwierdzić tę konfigurację OK Cancel			

Rysunek 1.3.8 Skonfigurowany interpreter Pythona

Czasami, może się zdarzyć że masz na swoim komputerze zainstalowane dwie różne wersje Pythona. PyDev wyświetli wówczas ich listę i poprosi byś wybrał spośród nich tę, która ma być skonfigurowana. Jeżeli po naciśnięciu przycisku *Auto Config* pojawi się komunikat o braku możliwości odnalezienia Pythona — wykonaj "ręczną" konfigurację (por. sekcja 5.3, str. 110).

Rozdział 2. Pierwsze kroki w Eclipse

Tutaj zaczyna się nasz projekt. Będzie to adaptacja modyfikatora *Bevel*. Więcej o tym powiem w następnym rozdziale. W tym rozdziale, poza nazwą, nasz projekt nie będzie miał z Benderem nic wspólnego. Na początek chcę pokazać podstawy pracy w środowisku Eclipse. Zrobię to na przykładzie najprostszego skryptu Pythona, piszącego w oknie konsoli napis "Hello". Zakładam, że Czytelnik ma pewne pojęcie o Pythonie, oraz pracował już w jakimś IDE. To nie jest podręcznik żadnego z tych zagadnień. Moim celem jest raczej pokazanie, jak w Eclipse wykonuje się pewne podstawowe czynności, znane każdemu programiście.

2.1 Rozpoczęcie projektu

Nowy projekt zaczynamy poleceniem *File→New→Project...* (Rysunek 2.1.1):

C/C++ - Eclipse				
File Edit Source Refactor	r Navigate Sea	rch Project Run Window Help		
New	Alt+Shift+N	🔂 Makefile Project with Existing Code	1	} (⇒ ⇒ A*
Open File		📬 C++ Project		
Close	Ctrl+W	🔂 C Project		
Close All	Ctrl+Shift+W	📬 Project		
H Save	Ctrl+S	الا Convert to a C/C++ Project		
📓 Save As		😂 Source Folder		
ng Save All	Ctrl+Shift+S	😂 Folder		
Revert		💣 Source File		
Move		💦 Header File		
Rename	F2	File from Template		
🔊 Refresh	F5	Class		
Convert Line Delimiters To) •	👕 Task		
👜 Print	Ctrl+P	📬 Other	Ctrl+N	
Switch Workspace	•			
Restart				
🚵 Import				
🛃 Export				
Properties	Alt+Enter			
Exit				

Rysunek 2.1.1 Polecenie tworzące nowy projekt

Otwiera to okno, w którym rozwiń folder *PyDev* i wybierz kreatora *PyDev Project* (Rysunek 2.1.2):

🖨 New Project	
Select a wizard	
Wizards:	
type filter text	
🕀 🗁 General	
i	
😟 🗁 CVS	
😟 🗁 Java	
🖻 🥭 PyDev	
🚽 🗾 PyDev Django Project	
PyDev Google App Engine Project	Wybierz tego kreatora

Rysunek 2.1.2 Wybór kreatora, odpowiedniego do rodzaju projektu

Następnie naciśnij przycisk Next.

W oknie kreatora projektu wpisz jego nazwę. Proponuję zacząć tu od razu projekt, który wykorzystamy do stworzenia skryptu dla Blendera. Stąd nadaję mu nazwę **Bevel** (Rysunek 2.1.3):

e	
PyDev Project Create a new Pydev Project.	Istal nazwę projektu
Project name: Bevel	
Project contents: Use default	
Directory C:\Documents and Settings\w497972 Project type Choose the project type O Python O Jython Iron Python Trace	21\workspace\Bevel Browse
Grammar Version	rython, składnia wersji 5.x
3.0 Interpreter	`
Default	▼
Click here to configure an interpreter not listed. Create 'src' folder and add it to the PYTHONPA Add project directory to the PYTHONPATH? Ono't configure PYTHONPATH (to be done man	ATH? Możesz także wybrać tę opcję — nie będziesz wtedy musiał od- dzielnie tworzyć tego folderu (por. str. 21)
Gdy to ustawisz, naciśnij ten przycisk —	
? Cack	Next > Finish Cancel

Rysunek 2.1.3 Ekran kreatora, ustalający szczegóły nowego projektu

Wybierz *Python* jako *Project type*, i składnię (*Grammar Version*) na **3.0**. Resztę parametrów pozostaw bez zmian i naciśnij przycisk *Finish*.

Pojawi się jeszcze komunikat (Rysunek 2.1.4):

🖶 Open Associated Perspective?
This kind of project is associated with the PyDev perspective. Do you want to open this perspective now?
Włącz tę opcję, by więcej program o to nie pytał
Remember my decision
Yes No

Rysunek 2.1.4 Ekran kreatora, ustalający szczegóły nowego projektu

Uwaga: jeżeli zapomniałeś skonfigurować interpreter, kreator wyświetla błąd i wyszarza przycisk *Finish* (Rysunek 2.1.5):

	PyDev Project Kreator wyświetla ewentualne błędy w nagłówku okna
	Project interpreter not specified
-	
	Grammar Version
	3.0
	Interpreter
	Please configure an interpreter in the related preferences before proceeding. Gdy zapomniałeś ustawić
	O Create 'src' folder and add it to the PYTHONPATH?
	O Add project directory to the PYTHONPATH?
-	Dopoki w nagłowku jest błąd, przycisk <i>Finish</i> jest wyszarzany
	Cancel

Rysunek 2.1.5 Błąd, zgłaszany przez kreator, gdy interpreter Pythona nie jest jeszcze ustawiony

Skorzystaj wtedy ze skrótu, umieszczonego w oknie, by przejść do okna *Preferences* i uzupełnić tę konfigurację (por. sekcja 5.3). Gdy to zrobisz, wrócisz do okna kreatora, i będziesz mógł stworzyć projekt.

Kreator tworzy w Eclipse pusty projekt PyDev (Rysunek 2.1.6):



Rysunek 2.1.6 Nowy projekt PyDev w Eclipse

To, co widzisz, to domyślny układ zakładek z różnymi panelami projektu. Podobnie jak w Blenderze możesz mieć wiele alternatywnych układów ekranu, tak w Eclipse możesz mieć wiele alternatywnych "perspektyw" (*perspective*) projektu. Nowy projekt zawiera domyślną perspektywę *PyDev*, Przy okazji debugowania dodana zostanie jeszcze inna perspektywa — *Debug*.

Zacznijmy adaptację aktualnej perspektywy od usunięcia niepotrzebnej zakładki *Welcome* (Rysunek 2.1.7):

Kliknij tu zamknąć	by zakładkę		
ə • 🖉	🖻 🥭	PyDev 🗄	c/C++
🛞 Welcome 🏅			
	N 🟠 <	⇒ ⇔ a*	A* 🖺

Rysunek 2.1.7 Zamykamy zakładkę Welcome

Następnie dodajmy do projektu folder na skrypty: zaznacz folder projektu, i z jego menu kontekstowego wybierz *New->Source Folder* (Rysunek 2.1.8):

🖨 PyDev	- Eclipse		
File Edit	Navigate Search Project Pydev Run Window Hel	p	
i 📬 - 🛛] 🗟 🖆 i 🎥 i 🏇 • 💽 • 🏊 • i 🖨 🔗 •	• 1 2 - 7 - 4 - 4 - 4 - 4	E 🔁
🛱 PyDev I	Pack 🛛 🖵 Menu kontekstowe projektu		🗆 🗄 Outline 🛛
	(otwórz, naciskając PPM)		An outline is not a
🗉 😂 Be	vel 📕	-	
	New 🕨	📬 Project	
	Go Into		
	Copy.		
		😕 Source Folder	
		PvDev Module	
	Move		
	Rename		
	Remove from Context Ctrl+Alt+Shift+Down	📑 Other Ctrl+N	

Rysunek 2.1.8 Dodanie nowego folderu na pliki źródłowe

W oknie kreatora folderu wpiszmy mu nazwę — powiedzmy, src (Rysunek 2.1.9):

e	
Create a new Source Folder	
Project Bevel Name src	Browse
Wpisz nazwę folderu na pliki z kodem programu Finish	Cancel



Gdy naciśniesz *Finish*, w projekcie powstanie folder o tej nazwie.

Teraz stworzymy nowy, pusty plik skryptu. Rozwiń menu kontekstowe folderu **src**, i wywołaj polecenie *New→PyDev Module* (Rysunek 2.1.10):

File Edit Na	vigate Search	Project Py	dev Run	Window Help			
i 📬 • 🔛	le 🕒 i 🔗	i 🏇 • (- 🤬	• 🛛 😂 🛷 •	i 🖢 - 🖗 - 🍤	$\Leftrightarrow * \Rightarrow *$	🖹 🛃
増 PyDev Pac	:k 🛛 🗖				-	- 8	🗄 Outline 🛛
D C Dowel		(0	enu konte twórz, nac	ekstowe folderu ciskając PPM)	1		An outline is not av
🔲 🗁 Bevel	-				J		
📃 🗄 🥭 P	New				🎽 📑 Project		
	Go Into				T File		
	📄 Сору				Folder		
	💼 Paste						
	💢 Delete				PyDev Module		
	Move				H PyDev Package	м 	
	Rename				📑 Other	Ctrl+N	
	🕭 Remove fro	m Context	Ctrl+A	Alt+Shift+Down			

Rysunek 2.1.10 Dodanie nowego skryptu do folderu

Otworzy to okno kreatora pliku Pythona. Nadaj plikowi nazwę odpowiednią dla wtyczki Blendera: **mesh_bevel**, i wybierz szablon (*Template*) **<***Empty***>** (Rysunek 2.1.11):

0		
Create a new	Python module	Ż
Source Folder	/Bevel/src	Browse
Package	Tu wpisz nazwę pliku (bez rozszerzenia .py)	Browse
Name Template	mesh_bevel Cempty> Module: Class Module: Main Module: Unittest Module: Unittest with setUp and tearDown	<u>Config</u>
?	Finish	Cancel

Rysunek 2.1.11 Okno kreatora skryptu Pythona

Następnie naciśnij przycisk Finish.

Jeżeli masz na swoim komputerze bardzo restrykcyjny firewall, to mógłbyś teraz zobaczyć prośbę o otwarcie na nim pewnego portu TCP. (To jedna z uwag, jaką otrzymałem od mojego recenzenta). Chodzi o dostęp do pętli zwrotnej 127.0.0.1. W każdym razie sam tego nie widziałem, choć mój firewall także nie jest specjalnie "pobłażliwy". W projekcie pojawi się pierwszy plik Pythona. PyDev domyślnie wstawił w nagłówek komentarz z datą utworzenia i nazwą użytkownika (Rysunek 2.1.12):

PyDev - Bevel/src/mesh_b	evel.py - Eclipse	
File Edit Source Refactoring N	avigate Search ProjeGwiazdka informuje, dow Help	
i 📬 • 🔛 🗟 🗁 i 🔗 i s	🌣 • 💽 • 🂁 • 🕴 że plik zawiera nie 🖓 • 🏷 🗇 •	⇒ - Ê
増 PyDev Pack 🛛 🗧 🗖	P *mesh_bevel ⋈	🗖 🗖 📴 Outline 🖾
 □ 	⊖ Created on 07-06-2011	type filter text
·	Gauthor: W4979721 I VV Nazwa użytk wstawiona w kom	ownika, nentarz

Rysunek 2.1.12 Nowy, pusty skrypt

Podsumowanie

- W tej sekcji stworzyliśmy nowy projekt Pythona, posługując się kreatorem PyDev Project (str. 18);
- Przed dodaniem do projektu plików skryptów, musisz przygotować dla nich odpowiedni folder (*source folder* str. 21);
- Przy tworzeniu nowego skryptu można skorzystać z kilku predefiniowanych wzorów (str. 22). My jednak nie użyliśmy żadnego z nich, wybierając wzorzec "pusty" (<*Empty*>);
- Nazwa projektu jest sprawą dowolną. Projekt w tym przykładzie nazwałem Bevel dlatego, że w dalszych rozdziałach książki posłuży nam do stworzenia w Blenderze 2.5 nowego polecenia: Bevel. Z tego samego powodu nadałem plikowi ze skryptem Pythona nazwę mesh_bevel.py.

2.2 Uruchomienie najprostszego skryptu

Skrypt, który tu napiszemy, ma wyświetlić w konsoli Pythona napis "Hello!". Więc musimy najpierw dodać panel z konsolą do naszego środowiska, bo domyślnie jej tu PyDev nie umieścił. Aby to zrobić, kliknij w zakładkę u dołu ekranu (bo tam dodamy konsolę). Następnie wywołaj: *Window→Show View→Console* (Rysunek 2.2.1):

avigate Search Project Pydev Run	Window Help		
»•• •• • • • • • 🖾	New Window New Editor	🖹 🎅 PyDev 🔤 C/C	2++
P mesh_bevel S	Open Perspective	Code Coverage	
Created on 07-06-2011 @author: W4979721 '''	Customize Perspective Save Perspective As Reset Perspective Close Perspective Close All Perspectives Navigation	■ Console Alt+Shift+Q, C	
Problems 🛛		Other Alt+Shift+Q, Q	- 0
0 items			
Description A	Resource Path	Locat Type	

Rysunek 2.2.1 Dodanie zakładki z konsolą

Domyślnie konsola pokazuje rezultat działania skryptu. W trakcie pisania kodu Pythona bardzo przydatna jest jeszcze tzw. "interaktywna konsola". Dodajmy więc i ją (Rysunek 2.2.2):

🖹 Problems 📮 Console 🛛			[] • 🗆 🗖
PyDev Scripting		🔄 1 CVS	
		2 New Console View	Rozwiń 🚔
		🤪 3 PyDev Console	to
	Wybierz konsolę PyDev		
	•		
<	O Console for currently active editor		>
	• Python console		
Wskaż ten	O Unable to create console for Jython (interp	reter not configured)	
rodzaj konsoli	🔿 Unable to create console for Iron Python (ir	nterpreter not configured)	
	🔘 Jython using VM running Eclipse console		
		OK Cancel	

Rysunek 2.2.2 Zmiana typu konsoli na interaktywną

Po wybraniu z menu rozwijalnego zakładki PyDev Console, wskaż w oknie dialogowym Python console.

I oto masz w panelu uruchomiony interpreter Pythona, w którym można na bieżąco sprawdzać fragmenty kodu (Rysunek 2.2.3):



Rysunek 2.2.3 Interaktywna konsola Pythona

Jedną z bardzo przydatnych funkcji PyDev jest "dopowiadanie kodu" (*autocompletion*). To działa zarówno w oknie edytora skryptu, jak i interaktywnej konsoli (Rysunek 2.2.4):



Rysunek 2.2.4 Przykład dopełniania kodu

Zazwyczaj dopełnianie zaczyna działać, gdy w jakimś wyrażeniu pojawi się kropka (np. wpisz w konsoli "sys."). Dzięki temu nie utrudnia specjalnie pisania "zwykłego" kodu.

No, ale dosyć już gadania. Eclipse jest bardzo bogatym środowiskiem i wszystkich jego funkcji i tak nie zdołam tu opisać. Lepiej przygotujmy nasz najprostszy skrypt (Rysunek 2.2.5):



Rysunek 2.2.5 Nasz skrypt – oczywiście w pierwszej wersji ©

Gdy skrypt jest gotowy, podświetl go i z menu *Run* wybierz *Run As →Python Run* (Rysunek 2.2.6):



Rysunek 2.2.6 Uruchomienie skryptu

W rezultacie konsola przełączy nam się sama na okno rezultatu, i zobaczymy w nim nasz napis "Hello!" (Rysunek 2.2.7):

🖹 Problems	📮 Console	×	Pu PyUnit			X 8	ž 9	5		<u>a</u> !! (₽ (P			· 📬 •	- 8
<terminated></terminated>	C:\Document	s and Se	ttings\w49	79721\wor	kspac	e\Bev(el\src\	mesh_t	evel.p	ру					
Hello!		Rezulta skryptu	ıt działani	a tego							Tu m powr kons	nożes: otem olę	z się na	przełąc interakt	zyć z tywną
<															>



Podsumowanie

- Dodaliśmy do projektu panel z konsolą Pythona (str. 24);
- Poznałeś działanie autokompletacji kodu, oraz wyświetlanie opisu funkcji "w dymkach" (str. 25);
- Uruchomiliśmy najprostszy skrypt i sprawdziliśmy jego rezultat (str. 26);

2.3 Debugowanie

Aby wstawić w jakąś linię kodu punkt przerwania (*breakpoint*), kliknij **PPM** w szary pasek na lewej krawędzi edytora (Rysunek 2.3.1):



Rysunek 2.3.1 Zaznaczenie punktu przerwania (breakpoint)

Z menu kontekstowego, które w ten sposób rozwiniesz, wybierz polecenie *Add Breakpoint*. Oczywiście, kliknij przy linii, w którą chcesz ten punkt przerwania wstawić. Eclipse rysuje w tym miejscu zieloną kropkę z kreską (Rysunek 2.3.1). (W podobny sposób, za pomocą menu kontekstowego, możesz usunąć lub wyłączyć niepo-trzebny punkt przerwania).

Aby uruchomić program w trybie śledzenia, naciśnij ikonę z pluskwą (© Rysunek 2.3.2):



Rysunek 2.3.2 Uruchomienie skryptu w trybie śledzenia

Eclipse wyświetli wówczas informację o zmianie aktualnej perspektywy projektu na *Debug*. (Za pierwszym razem doda ją do Twojego projektu). Pamiętaj, że musisz być w tej perspektywie, by śledzić działanie skryptu! Rysunek 2.3.3 przedstawia układ ekranu, oraz kontrolki i podstawowe skróty klawiszowe dla debugowania programu. Zwróć uwagę, że wykonanie kodu zatrzymało się na naszym *breakpoincie*:

Resume Terminate Step Into Step over Step Return Pydev	Run Window Help
F8 Ctrl -F2 F5 F6 F7 *	🗘 🔹 🔿 🔹 🔛 🔛 🔛 🕆 🗘
Stebug 🖇 📃 🗖	🕪= Variables 🖾 💊 Breakpoints 🗟 🗖 🗖
🔆 🗱 🕩 🗉 🛋 🕺 🕭 🖉 🖉 🐨 🐺 🗡	≝ 🕫 🖻 🖉 💥 🎽 🎽
E e Bevel mesh_bevel.py [Python Run]	Name Value
e 🍪 mesh_bevel.py wywołań (funkcje)	🗄 🔍 Globals 🛛 Global variables
🖃 🔐 MainThread - pid4744_seq1	
main [mesh_bevel.py:5]	zmiennych skryntu
<pre> <module>[mesh_bevel.py:8]</module></pre>	
exectile pydev exectile.py:37	
P mesh_bevel ⋈	
⊖ * * *	▲ 3 [4, 3: 6.1 ▼
The simplest scriptLinia kodu, która ma	hung fillen kende
być obecnie wykonana	cype nicer cexc
⊖def main(): ✓	🔤 👝 🛛 🔍 main
<pre>c = "Hello/"</pre>	
print(c)	
Program zatrzymał się	
main() w ustalonym punkcie.	
🔄 Console 🛛 🧔 Tasks 🔝 Problems 🚺 Executables	- 8
mesh_bevel.py	× 🔆 🗣 🖷 🖹 🗗 🚰 🛃 🕶 🖬 🔹
pydev debugger: warning: psyco not available	e for speedups (the debugger will s🔨
pydev debugger: starting Nie pr.	zejmuj się tym komunikatem o braku opcjonalnego
kompo	nentu (W PyDev 2.1 już się nie pojawia)

Rysunek 2.3.3 Układ ekranu w perspektywie Debug

Zielonkawa linia w oknie kodu źródłowego to linia bieżąca. Gdy teraz naciśniesz **F6** (*Step over*) — nadasz wartość zmiennej **c** i przejdziesz do następnej linii (Rysunek 2.2.4):



Rysunek 2.3.4 Sytuacja po naciśnięciu F6 (Step over)

Gdy znów naciśniesz F6, wykonasz kolejną linię i opuścisz funkcję main() (Rysunek 2.3.5):

🏇 Debug 🛛 🖓 🗖	🝽= Variables 🛛 💁 Breakpoints 🗖 🗖
~	🏝 📲 📄 💣 💥 🔆 🎽
💥 🖉 🕩 🗉 🔳 🖓 D. O. Le 🖚 🔜 (i) 😿	Name Value 🔨
🖃 🍠 Bevel mesh_bevel.py [Python Run] 🛛 🔥	🗄 🔍 Globals 🛛 Global variables
🖨 🔗 mesh_bevel.py	🗄 🔍builtins module: <module 'builtins'="" (built-in)=""></module>
😑 🔊 MainThread - pid4744_seq1	 doc str: \nThe simplest script\n
and the second secon	filestr: C:\\Documents and Settings\
execfile [_pydev_execfile.py:37]	
<pre> P mesh_bevel \(\begin{aligned}</pre>	już funkcję Cutline S Cutline S S S S S S S S S S S S S S S S S S S
Tall Tall Tall Tall Tall Tall Tall Tall	inia została wykonana vięc w konsoli pojawił apis!
mesh_bevel.py	🔳 🗶 💥 🖳 🔚 🕞 🖓 🛃 🚍 🚝 🖆 🖬 🕇 🖬
Hello!	

Rysunek 2.3.5 Sytuacja po zakończeniu funkcji

Zwróć uwagę, że w oknie stosu znikła widoczna dotychczas linia *main() [mesh_bevel.py]* (por. Rysunek 2.3.4). Jednak zielonkawa aktualna linia nadal tkwi na wywołaniu tej funkcji w głównym kodzie modułu (*<module>* [*mesh_bevel.py*]). Tak w Eclipse wygląda każde zakończenie funkcji. Gdy się zagalopujesz i jeszcze raz naci-śniesz F6, znajdziesz się w pomocniczym kodzie PyDev (Rysunek 2.3.6):

Debug X Lepiej jest z cały proces prz Resume (F8)	akończyć 🗖 zyciskiem 🚽	(x)= Variables	23	●o Breakpoints □
💥 🕅 🕩 💷 🔳 🕅 3. 13. 16 - 16 - 1	क 🖬 🐼	Name		Value
🖃 🍠 Bevel mesh_bevel.py [Python Run]	~	🗄 🔍 Glob	als	Global variables
🗐 🞯 mesh_bevel.py	Bo teraz	wylądowali-		int: 1
🖻 🛷 MainThread - pid4744_seq1	śmy w funk	cji jakiegoś	ents	str: ""\nThe simplest script\n""\nde
execfile _pydev_execfile.py:3	PvDev!	ego skryptu _c	oding	NoneType: None
run [pydevd.py:925]	- 1 9000			>
🖻 mesh_bevel 📔 _pydev_execfile 🛛			- 8	E Outline 🛛 🗖 🗖
stream = oper(file)			~	》 🎝 💥 🖓 🏹
try:				hune filter text
contents = stream.re	≘ad()			
finally:				E W exect iego źródło!
stream.close()				- r
	÷			
<pre>exec(compile(contents+"</pre>	n", file,	'exec')	/ 🗸	

Rysunek 2.3.6 Sytuacja po kolejnym naciśnięciu F6 (Step over)

To skrypt, za pomocą którego realizowane jest śledzenie Twojego programu. (PyDev wewnętrznie wykorzystuje zdalny debugger. My także użyjemy go do Blendera). Lepiej naciśnij **F8** (*Resume*), by zakończyć program.

🕴 📬 🕶 🔝 🗟 🖆 🟓 🤌 🗄 🎥 🗄 🏇 ។ 🔕 ។ 🗄 🏷 🔶 ។ 🖘 ។ 🔛 🔛 😰 🌺 Debug 🌏 Py	Dev
Przyciski debugowania riables Breakpoints Przyciski debugowania riables Pora się przełączyć na perspektywę edycji! Pora się przełączyć na perspektywę edycji!	□ □
Tak wygląda stos	>
P mesh_bevel	
stream = open(file)	\$•\$ ▽
Szybkie poprawki możes	z
finally:	,
stream.close()	
<pre>exec(compile(contents+"\n", file, 'exec'),</pre>	
🔄 Console 🛛 🧭 Tasks 🖹 Problems 🜔 Executables	
<terminated>mesh_bevel.py</terminated>	- 📬 -
Hello!	^

Rysunek 2.3.7 pokazuje, jak wygląda ekran po zakończeniu śledzonego procesu:

Rysunek 2.3.7 Sytuacja po naciśnięciu F8 (Resume) — skrypt został wykonany do końca

Drobne poprawki możesz wprowadzać w perspektywie *Debug*, korzystając z dostępnego tu także okna edytora. Gdy jednak chcesz dopisać nowy fragment kodu — przełącz się na perspektywę *PyDev* (Rysunek 2.3.8):

│ 📬 • 🔚 🗟 🗁 │ 🌮 │ 🏇 │ 🖢 - 🖓 - 🏷 🗘 • ↔ -	- 🖸 - 隆 - 🕴 😂 🛷 - 🗈 📅 🏷 Debug 🌏 PyDev	
📕 PyDev Package 🛛 🗖 🗖	🖻 mesh_bevel 🛛 👘 🖓 🖓 🖓	3
Image: Sevel Image: Sevel	<pre> Control Contro Control Control Control Control Control Control Control Cont</pre>	
	🖹 Problems 📮 Console 🛛 🖓	5
	<terminated> mesh_bevel.py 🔳 🗶 🔆 🗞 🖷 🚉 🚮 🗐 😰 📑 🔂</terminated>	•
	pydev debugger: warning: psyco not available for speed pydev debugger: starting Hello!	-

Rysunek 2.3.8 Powrót do perspektywy *PyDev* — czas na dalszą pracę nad kodem

Podsumowanie

- Zaznaczyliśmy w kodzie programu punkt przerwania (breakpoint str. 27);
- Uruchomiliśmy skrypt w debuggerze (str. 27). Przy okazji stworzyliśmy nową perspektywę projektu Debug;
- Poznałeś podstawowe funkcje debuggera: Step Into (F5), Step Over (F6), Resume (F8) (str. 28);
- Poznałeś dodatkowe panele debuggera zmiennych (Variables str. 28) i stosu (Stack str. 29);
- Po zakończeniu skryptu z projektu, debugger przechodzi do linii jakiegoś pomocniczego skryptu PyDev (str. 29). Dlatego lepiej jest w tym miejscu kazać mu wykonać się do końca (*Resume* F8);

32

Tworzenie aplikacji Blendera

To główna część książki. Opisuje proces tworzenia dodatku do Blendera. Najpierw powstaje zwykły skrypt — "linearny" ciąg poleceń, realizujący założoną operację (Rozdział 3). Potem otrzymuje "obudowę" wymaganą dla wtyczek Blendera (Rozdział 4). Rezultatem jest gotowy do użycia dodatek (*add-on*), implementujący nowe polecenie programu.

33

Rozdział 3. Skrypt dla Blendera

W tym rozdziale przygotujemy skrypt, fazujący (*Bevel*) wybrane krawędzie siatki. Posłużyłem się tym przykładem, by pokazać w praktyce wszystkie szczegóły środowiska programisty skryptów Blendera. Opisuję tu także metody rozwiązywania typowych problemów, jakie napotkasz podczas tworzenia kodu. Jednym z nich jest odnalezienie właściwego fragmentu Blender API, obsługującego potrzebne nam zagadnienia! (Na razie chyba nikt, poza jego twórcami, nie ogarnia całości...).

3.1 Sformułowanie problemu

W Blenderze 2.49 po naciśnięciu klawisza w otwierało się menu *Specials*. Można było z niego wybrać polecenie *Bevel*, fazujące zaznaczone krawędzie siatki (Rysunek 3.1.1):



Rysunek 3.1.1 Blender 2.49 — wywołanie polecenia Bevel

Po wywołaniu polecenia ruch myszki sterował szerokością fazy. Jeżeli chciałeś jakąś "okrągłą" wartość, należało dodatkowo wcisnąć klawisz **Ctrl** (Rysunek 3.1.2):



Rysunek 3.1.2 Blender 2.49 — ustalanie szerokości fazki

Drobnym mankamentem tego polecenia był brak możliwości wpisania dokładnej, numerycznej szerokości fazki.

35

Operację kończyło kliknięcie **LPM**. Tam, gdzie trzeba było, Blender dodawał do uzyskanej siatki dodatkowe krawędzie. (Aby wszystkie ściany miały co najwyżej 4 boki) (Rysunek 3.1.3):



Rysunek 3.1.3 Blender 2.49 — rezultat polecenia Bevel

Prawda że prosto i szybko?

Takiego polecenia brakuje obecnie wielu użytkownikom Blendera 2.5. Zrezygnowano w nim z tego "destrukcyjnego" *Bevel*. Pozostawiono tylko, istniejący także w wersji 2.49, modyfikator *Bevel*, który nanosi fazki w sposób "niedestrukcyjny" (Rysunek 3.1.4):





Zaczynamy, oczywiście, od dodania do obiektu modyfikatora **Bevel**. W pierwszej chwili spowoduje to sfazowanie wszystkich krawędzi siatki. Jeżeli jednak przełączysz *Limit Metod* na **Weight**, w panelu modyfikatora pojawi się drugi pasek opcji. Wybierz z nich np. **Largest**. Jednocześnie z siatki znikną wszelkie fazy, bo na razie wszystkie krawędzie mają ustawioną *Bevel Weight* = 0. (To wartość domyślna).

W przypadku modyfikatora *Bevel* można dynamicznie zmieniać szerokości fazek, przesuwając myszką z wciśniętym rokontrolce *Width* (jest suwakiem). Możesz chwilę z tym poeksperymentować, ale potem wpisz tam docelową wartość (np. 0.1 jednostki Blendera — tak, jak pokazuje to Rysunek 3.1.4) Jak zmienić *Bevel Weight* wybranych krawędzi? Otwórz przybornik (**T** – *Toolbox*). W sekcji *Mesh Options* przełącz tryb selekcji krawędzi na *Tag Bevel* (Rysunek 3.1.5):



Rysunek 3.1.5 Zaznaczanie fazowanych krawędzi

Przełącz się jeszcze w tryb selekcji krawędzi. Teraz, trzymając wciśnięty klawisz **Ctrl**, klikaj **PPM** w poszczególne krawędzie. Pod każdą z nich pojawi się fazka (bo tym kliknięciem zmieniasz jej *Bevel Weght* na 1.0).

• Zwróć uwagę że Blender oznacza krawędzie z niezerowym *Bevel Weight* kolorem żółtym. To pomaga zorientować się, co właściwie mamy na siatce ustawione.

Dzięki działaniu modyfikatora *Bevel*, ostateczna siatka jest sfazowana, a jednocześnie oryginalny sześcian nie uległ zmianie. Taki efekt przydaje się w wielu przypadkach, bo pozwala uniknąć nadmiernego komplikowania siatki. Dlatego działanie modyfikatora Bevel określa się często jako "fazowanie niedestrukcyjne".

Aby uzyskać "trwałe" zmiany w siatce, jak w Blenderze 2.49. musimy "utrwalić" ten modyfikator (przyciskiem *Apply* — Rysunek 3.1.6):



Rysunek 3.1.6 "Utrwalenie" (Apply) modyfikatora
37

Po naciśnięciu *Apply* (aha, zrób to w *Object mode*!) modyfikator zniknął, a "fazki" stały się realną siatką. Można ją poddać dalszej edycji (Rysunek 3.1.7):



Rysunek 3.1.7 Usuwanie (na wszelki wypadek) wag fazowania

Właśnie, skoro mowa o edycji: trzeba jeszcze "poklikać" (Ctrl-PPM) w żółte krawędzie, które pozostały po tej operacji. Warto im przełączyć *Bevel Weight* na 0, by przypadkiem nie zostały sfazowane przez następny mody-fikator *Bevel* (Rysunek 3.1.8):



Rysunek 3.1.8 Blender 2.57 — rezultat fazowania

Przyznacie, że sporo tu było "klikania"? I choć modyfikator *Bevel* także ma swoje zalety, to wielu użytkowników Blendera 2.57 chciałoby mieć dodatkowo jego prostą wersję "destrukcyjną".

W tym rozdziale napiszemy skrypt Blendera, który wykorzysta modyfikator *Bevel* do stworzenia "destrukcyjnej" wersji tej operacji. W zasadzie wykona automatycznie to, co na poprzednich stronach zrobiłem ręcznie. W następnym rozdziale przekształcimy ten skrypt w profesjonalny *add-on* Blendera.

Podsumowanie

- W Blenderze 2.5 brakuje polecenia *Bevel*, które trwale fazuje wybrane krawędzie siatki. Takie polecenie istniało w poprzedniej wersji programu (2.49 por. str. 34);
- Ten sam ostateczny efekt można w Blenderze 2.5 uzyskać poprzez "utrwalenie" odpowiedniego modyfikatora (str. 35 - 37). Aby nie powtarzać tych operacji ręcznie, stworzymy skrypt, który wykona je wszystkie za jednym razem. W ten sposób uzupełnimy program o funkcjonalność, której nam brakuje;

3.2 Dostosowanie Eclipse do API Blendera

Aby wygodnie pisać skrypty Blendera, musimy sprawić by PyDev "poznał" jego API. Tylko wtedy będzie nam podpowiadał metody i pola obiektów tak samo, jak to robi dla standardowych modułów Pythona. Na szczęście autorzy tej wtyczki przewidzieli taką możliwość. Wymagają tylko dostarczenia czegoś w rodzaju uproszczonych plików Pythona, zawierających same deklaracje klas, metod i właściwości. Sam pomysł jest podobny do "plików nagłówków" (*header files*), stosowanych w C/C++. Aby odróżnić takie "pliki nagłówków" od zwykłych plików Pythona, PyDev wymaga, by nadać im rozszerzenie *.*pypredef* (od *predefinition*).

Przerobiłem trochę skrypt Campbella Bartona, którym generowana jest dokumentacja API Pythona. Używając go, udało mi się stworzyć odpowiednie pliki **.pypredef* dla całego API Blendera, poza modułem *bge*. Dołączyłem je do tej książki. Pobierz z mojego portalu plik <u>http://airplanes3d.net/downloads/pydev/pydev-blender.zip</u>. Rozpakuj go do folderu, w którym umieściłeś Blendera (Rysunek 3.2.1):



Rysunek 3.2.1 Rozpakowanie materiałów pomocniczych do folderu Blendera

Pliki *.py i folder doc powinny się znaleźć w tym samym folderze, co blender.exe (Rysunek 3.2.2):



Rysunek 3.2.2 Zawartość pliku po rozpakowaniu

PyDev - Bevel/src/mesh_bevel.py - Eclipse File Edit Source Refactoring Navigate Search Project Pydev Help Run Window Open Project 😭 🏇 Debug 🥏 PyDev Podświetl projekt Close Project 긟 l PyDev Package Explorer 🛛 🔜 Build All Ctrl+B 8 📂 Bevel Build Project ₽ 🚊 🗁 进 src script Build Working Set 🗄 🕑 mesh_bevel.py Clean... Python 3.2 (C:\Program Files\Python32\p Build Automatically 110/" 🗄 🥐 C:\Program Files\Python32 Properties 🗄 🛁 System Libs | n kð 2. Przejdź do jego 🛋 Predefined Completions właściwości 🖻 🛋 Forced builtins main()

Gdy pliki są na miejscu, trzeba zmienić konfigurację projektu (*Project → Properties*, Rysunek 3.2.3):

Rysunek 3.2.3 Przejście do konfiguracji projektu

W oknie, które się pojawi, wybierz sekcję *PyDev – PYTHONPATH*, a w niej — zakładkę *External Libraries* (Rysunek 3.2.4):

🖨 Properties for Bevel								
1. Wybierz ilter text	PyDev - PYTHONPATH 2. Przejdź do tej zakładki							
Builders	The final PYTHONPATH used for a launch is composed of the parts defined here, joined with the paths defined by the selected interpreter.							
PyDev - Interpreter/Gra	😕 Source Folders 📓 External Libraries 🔵 String Substitution Variables							
PyDev - PYTHONPATH	External libraries (source folders/zips/jars/eggs) outside of the workspace.							
Task Repository	When using variables, the final paths resolved must be filesystem absolute.							
WikiText	Changes in external libraries are not monitored, so, the 'Force restore inter should be used if an external library changes.							

Rysunek 3.2.4 Przejście konfiguracji PYTHONPATH

Dopisz tu (*Add source folder*) ścieżkę do katalogu *doc\python_api\pypredef* (Rysunek 3.2.5):

😕 Source Folders 🏾 🍒 External Libraries 🛛 👳 String Substitution V	ariables
External libraries (source folders/zips/jars/eggs) outside of the workspa	ace.
When using variables, the final paths resolved must be filesystem abso Changes in external libraries are not monitored, so, the 'Force restore is should be used if an external library changes.	lute. 1. Używając tego przycisku, doda internal ścieżkę do <i>doc\python_api\pypredel</i>
C:\Program Files\Blender\doc\python_api\pypredef	Add source folder Add zip/jar/egg Add based on variable
2. A potem naciśnij ten przycisk	Remove Restore Defaults Apply

Rysunek 3.2.5 Konfiguracja PYTHONPATH

Koniecznie naciśnij potem przycisk *Force restore internal info* (Rysunek 3.2.5). Spowoduje to "przejście" przez pasek stanu Eclipse informacji o postępie aktualizacji (przez sekundę lub dwie)¹.

Od tej chwili, gdy dopiszesz do skryptu odpowiednią deklarację *import*, PyDev zacznie podpowiadać właściwości i metody klas Blendera (Rysunek 3.2.6):

i 📬 🖬 🕼 🗁 i 🗳	🕴 🏇 🔹 🔕 🔹 🕴 🔗 🔹 🏷 🤤 🗢 🖘 🖄 🔛 🔡 🎆 Debug 🚑 PyDev	~
🛱 PyDev 🛛 🗖 🗖	🖻 *mesh_bevel 🗙 🗖 🗖	
E Sevel	Dopisz najpierw tę deklarację! Bez niej nic nie zadziała!	e E
Burter Src Burter Src Burter Powerships Powerships Src Dowerships Src Dowerships Src Src	import bpy A potem już normalnie — po wpisaniu kropki pojawi się lista elementów odpowiedniej klasy	
🖶 🥔 C:)Program File	😢 cube = bpy.data.	
System Libs	print (cube.name) 📀 lattices 🔻	^
in and C:\Program in and C:\Program	main() O libraries	
🗷 🔬 C:\Program	• meshes	
🖻 🚮 C:\Program	• metaballs	
Predefined Co		
In Porcea building		
	PvDev Scripting	\mathbf{v}
	Press Ctrl+Space for templa	ites.

Rysunek 3.2.6 Uzupełnianie kodu dla API Blendera

Podpowiedzi pojawiają się zazwyczaj po wpisaniu kropki. A gdy zatrzymasz na chwilę kursor myszy ponad nazwą metody — PyDev wyświetli jej opis "w chmurce" (Rysunek 3.2.7)

📬 • 🔡 🕼 🇁 🎥	🎄 ▼ 🚺 ▼ 🗄 🔗 ▼ 🗎 🏷 🗘 ▼ 🖘 🔹 📅 📅 Debug 🌏 PyDev
🛱 PyDev 🛛 🗖 🗖	*mesh_bevel 🛛 🗖 🔤
Image: Sevel Image: Sevel	<pre>Ge, The simplest script, import bpy Gef main(): cube = bpy.data.ob_ects["Cube"] print(cube.name) main() main() </pre> Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy ponad właściwościa/metodą — PyDev wyświetli jej opis Gdy zatrzymasz kursor myszy Pyter vyter Pyter vyter vyter Pyt
E S C:\Program	cube = bpy.data.objects["Cube"]
Forced builtins	Możesz także skorzystać z tych skrótów, by przejść do pliku z deklaracją. To się przydaje dla właściwości.

Rysunek 3.2.7 Wyświetlanie opisów elementu

¹ Ten sposób użycia plików *.*pypredef* różni się od tego, który jest opisany na portalu pydev.org. Sęk w tym, że postępując zgodnie z tamtym opisem (dodanie folderu do *Predefined Completions*) nie mogłem zmusić PyDev do poprawnego uzupełniania kodu!

Co prawda te "chmurki" są kwadratowe, ale nazywam je tak ze względu na "ulotność" (znikną, gdy poruszysz myszką). No, chyba że klikniesz w umieszczony w nich skrót (por. Rysunek 3.2.7). Wtedy PyDev przeniesie Cię do definicji tego elementu (Rysunek 3.2.8):

••••	• 🖻	mesh_bevel	🖻 bpy 🛛		
Ħ			Greturns: BlendDataNodeTrees Collection of NodeTree		s E
		obj	ects = types.BlendDataObjects # (read only)		
		I	Object datablocks. Freturns: BlendDataObjects Collection of Object Właściwości mają dodatkowy opis, Właściwości mają dodatkowy opis,		
		par	ticles = types.BlendDataParticles # (read only) Particle datablocks. Greturns: BlendDataParticles Collection of ParticleSe	et 🔽	
		<		>	

Rysunek 3.2.8 Zawartość pliku bpy.pypredef, otwarta za pomocą skrótu z opisu elementu

Taka definicja znajduje się, z punktu widzenia PyDev, w pliku *bpy.pypredef*. W razie potrzeby ten plik zostanie otwarty w edytorze. Ma to sens przy sprawdzaniu właściwości (atrybutów) klasy. W "chmurkach" jest wyświetlany specjalny komentarz (tzw. *docstring*), umieszczony przez programistę poniżej deklaracji każdej metody/funkcji. Ale standard Pythona nie przewiduje stosowania *docstrings* dla zmiennych. (Właściwość obiektu jest, technicznie rzecz biorąc, jego zmiennym polem). Stąd skorzystanie ze skrótu do definicji jest jedynym sposobem na odczytanie opisu, który zazwyczaj towarzyszy każdej właściwości klasy Blender API.

Przy okazji: zwróć uwagę, co pokazuje panel Outline dla otwartego w edytorze pliku bpy (Rysunek 3.2.9):



Rysunek 3.2.9 Fragment struktury modułu bpy w panelu Outline.

43

Zwróć uwagę, że widok *Outline* modułu *bpy* może być czymś w rodzaju "pomocy szkoleniowej". Pozwala się zapoznać z podstawową strukturą API Blendera. Zacznijmy od "zwinięcia" całej hierarchii (Rysunek 3.2.10):



Rysunek 3.2.10 Podstawowa struktura API Blendera

Są to podstawowe elementy API:

udostępnia skryptom dane z aktualnego pliku Blendera. Jego polami są kolekcje róż-
nych rodzajów obiektów (scenes, objects, meshes, itp. — por. Rysunek 3.2.9);
to dane o bieżącym stanie "środowiska": aktywnym obiekcie, scenie, bieżącej selekcji;
zawiera wszystkie polecenia Blendera, udostępnione także użytkownikowi poprzez GUI.
(Dla Pythona każde polecenie to jedna z metod tego obiektu);
zawiera definicje wszystkich klas, używanych w obiektach, które występują w bpy.data i
bpy.context.

Gdy zajrzysz do wnętrza **bpy.types**, zobaczysz alfabetyczną listę wszystkich klas, wykorzystywanych w API. Wyjątkowo na początku umieściłem deklarację **bpy_struct**. To klasa bazowa dla wszystkich pozostałych. Jej metody i właściwości są dostępne zawsze w każdym obiekcie Blendera (Rysunek 3.2.11):



Rysunek 3.2.11 bpy_struct: klasa bazowa wszystkich klas

Inna sprawa, że w konkretnej klasie część właściwości **bpy_struct** może być nie zaimplementowana. Na przykład — **bpy_struct** ma metodę **items()**. Dlatego wszelkie klasy kolekcji — jak chociażby **MeshEdges** (krawędzie siatki — obiekty **MeshEdge**) implementują tylko jakieś dodatkowe metody, jak **add()** (Rysunek 3.2.12):



Rysunek 3.2.12 Klasy potomne – metody, właściwości

Oczywiście, dla wielu klas (np. wierzchołka siatki — *MeshVertex*) metoda *items()* (i wiele innych) jej implementacja jest pusta.

Wykorzystywanie takiej "odziedziczonej" metody *items()* przez każdą kolekcję utrudnia automatyczne uzupełniania kodu. PyDev odczytuje z definicji, że każda z nich zwraca po prostu *bpy_struct*. (Bo tak wynika z ich deklaracji, odziedziczonych po klasie bazowej). Można jednak "zasugerować" interpreterowi odpowiedni typ. Wystarczy umieścić wcześniej linię, przypisującej zmiennej odpowiednią klasę obiektu (Rysunek 3.2.13):



Rysunek 3.2.13 Wyświetlanie opisu funkcji

W zasadzie taką linię powinieneś dopisać tylko na chwilę, gdy potrzebujesz skorzystać z automatycznej kompletacji. Pamiętaj, aby zawsze umieścić ją przed pierwszym nadaniem wartości zmiennej. W ten sposób kod będzie działał poprawnie, nawet gdy o niej zapomnisz.

Zresztą — PyDev wykrywa takie linie, bo są to według niego nie używane zmienne. Umieszcza na nich odpowiednie ostrzeżenia (Rysunek 3.2.14):

Image: Image			📴 Outline δ	3	
		~ =		l ^a , M ini	🏤 ▽
import bpy			type filter tex		
⊖def main():					
cube = bpy.types.Obje cube = bpy.data.objec		🚽 bpy 🚳 main			
print(cube.name)					
main()	Gdybyś zapomniał u linię — wystarczy s	isunąć tę 💌 prawdzić			
📃 Console 🖉 Tasks 🔝 Problems 🔀	zakładkę Problems!			59	~
0 errors, 1 warning, 0 others					
Description 🔺	Resource	Path	Locat	Туре	
🗏 🙆 Warnings (1 item)	↓				
💧 Unused variable: cube	mesh_bevel.py	/Bevel/src	line 5	PyDev Probl	

Rysunek 3.2.14 Ostrzeżenia, związane z "deklarowaniem typu" dla uzupełniania kodu

Dobrą praktyką jest więc zajrzeć co jakiś czas do zakładki *Problems*. Zobaczysz tam wszystkie linie, które zapomniałeś usunąć. Korzystając z tej listy, będziesz mógł to zaraz poprawić.

Aby szybko się zorientować, gdzie w tej całej hierarchii znajduje się konkretne pole lub metoda, podświetl ją w edytorze i rozwiń menu kontekstowe (**PPM**). Wywołaj z niego polecenie **Show In →Outline**, a PyDev podświetli odpowiedni element (Rysunek 3.2.15):

1	•	l 🖻 🔒 i 🔗	☆・○・ ∦・ 🌾	• ⇔ - ⇒ -	😰 🏇 Debug ಿ PyDev
	P *	mesh_bevel 🔃 🖻	bpy 🛛	- 8	Dutline 🛛 🗖 🗖
日 日		111			》↓ª₂ 洪 ⋈ ♡
U		node_group '''Node gr Greturn	os = types.BlendDataN roup datablocks. ns: BlendDataNodeTree	odeTrees # (= s Collection =	type filter text
		objects =	ts # (read o	of armatures of brushes of cameras	
		🞺 Undo Revert File	Ctrl+Z	pllection c	Curves
		L Save	Ctrl+S		
		Show In	Alt+Shift+W	Utline	groups
		Cut Copy Part Copy Context C	Ctrl+X Ctrl+C Qualified Name	육- Navigator ^M 별 PyDev Package Exp III Properties	plorer plorer plorer
		Paste	Ctrl+V	🕴 (read onl 📒	lattices

Rysunek 3.2.15 Odnajdywanie położenia elementu w klasach modułu bpy.

Do tej pory omawialiśmy gałąź **bpy.types**. A operatory? Operatorów jest mnóstwo! Aby się wśród nich od razu nie zagubić, podzielono je na tematyczne grupy: **action**, **anim**, **armature**, … i tak dalej. Rozwińmy chociażby zespół **bpy.ops.brush** (Rysunek 3.2.16):



Rysunek 3.2.16 Operatory: przykład opisu parametrów i ich wartości domyślnych

Każdy zespół operatorów (np. *bpy.ops.brush*) to po prostu taka klasa, która ma same metody. Każdy operator (polecenie) to pojedyncza metoda takiej klasy. Zwróć uwagę, że operator zawsze można wywołać bez żadnych argumentów — zostaną wtedy wykorzystane ich wartości domyślne.

W sumie — wychodzi na to, że głównym tematem tej sekcji stało się poznanie struktury modułów Blender API. W takim razie, aby skończyć temat rozpoczęty na str. 43, wyliczę tu pozostałe moduły. Są o wiele mniejsze od podstawowych (*bpy.data*, *bpy.context*, *bpy.types*, *bpy.ops*) i pełnią rolę pomocniczą:

bpy.app	różne pomocnicze informacje o samym programie: numer wersji, położenie pliku
	<i>blender.exe</i> , flagi kompilacji, itp.;
bpy.path	pomocnicze funkcje do pracy ze ścieżkami i plikami (podobne zagadnienia w standar-
	dowym Pythonie obsługuje moduł os.path);
bpy.props	definicje specjalnych "właściwości" klasy, które Blender potrafi wyświetlać w swoich
	okienkach (gdy zajdzie taka potrzeba). Dla odróżnienia od zwykłych właściwości na-
	zwałbym je "właściwościami Blendera". Będziemy je wykorzystywać w następnym roz-
	dziale, przy okazji tworzenia operatora;
mathutils	pomocnicze klasy matematyczne: Matrix (4x4), Euler, Quaternion (odwzorowanie ob-
	rotu), Vector, Color. Zawiera także submoduł geometry z kilkoma pomocniczymi funk-
	cjami (przecięcie z linii, przecięcie promienia z płaszczyzną, itp);
bgl	procedury pozwalające skryptom rysować w przestrzeni okien Blendera (to w istocie
	większość funkcji OpenGL);
blf	procedury do pisania tekstu na ekranie

O dwóch pozostałych modułach — *aud* (*Audio*) i *bge* (*Blender Game Engine*) mam mgliste pojęcie. Nie będę się więc o nich rozpisywał.

Podsumowanie

- Pliki *.*pypredef*, umożliwiające autokompletację metod i właściwości klas API Blendera, znajdziesz w pliku towarzyszącym tej książce (str. 39);
- Po rozpakowaniu plików *. pypredef, należy ich folder dołączyć do PYTHONPATH projektu (str. 40);
- Autokompletacja Python API zaczyna działać po umieszczeniu na początku skryptu odpowiedniego wyrażenia importu (str. 41);
- Wyświetlane przez Eclipse "chmurki" ze szczegółowym opisem metod można wykorzystać do dalszego poznawania funkcji API (str. 41);
- Skróty do modułu *bpy*, pojawiające się w "chmurkach", można wykorzystać do otworzenia pliku *bpy.pypredef* w edytorze Eclipse (str. 41). Skorzystanie z tego skrótu pozwala odczytać opis właściwości (atrybutów) klasy, których Eclipse nie wyświetla (str. 42).
- Przeglądanie struktury modułu *bpy* w panelu *Outliner* pomaga także poznać strukturę Python API Blendera (str. 44);
- W wielu przypadkach do uzyskania poprawnej autokompletacji należy stosować "deklaracje zmiennych" (str. 44);

3.3 Opracowanie podstawowego kodu

Zazwyczaj w różnych poradnikach znajdujesz od razu gotowy kod, który czasami autor trochę skomentuje. W tej sekcji chciałbym pokazać coś, co musi nastąpić wcześniej: poszukiwanie rozwiązania. Ten etap jest równie ważny jak pisanie programu, a może ważniejszy. Za każdym razem pozwala mi poznać kolejny kawałek API Blendera. (Nawet gdybym chciał, nie jestem w stanie spamiętać wszystkich jego klas, pól i metod).

Przygotuj sobie najpierw plik Blendera do testów. Proponuję wykorzystać do tego celu domyślny sześcian, z ekranem ustawionym tak, jak to pokazuje Rysunek 3.3.1:



Rysunek 3.3.1 Propozycja układu ekranu w testowym pliku Blendera

Zapisz ten plik gdziekolwiek na dysku, a następnie zaimportuj do projektu (poleceniem *Import..* — szczegóły na str. 118), aby go zawsze "mieć pod ręką" (Rysunek 3.3.2):



Rysunek 3.3.2 Plik Blendera, wstawiony w projekt

Celem tej sekcji jest opracowanie kodu Pythona, który sfazuje zaznaczone krawędzie siatki. Szukanie metod programowego dodawania modyfikatora *Bevel* zostawmy sobie na później. Na potrzeby testów w tej sekcji dodajmy go po prostu ręcznie (Rysunek 3.3.3):

User Persp @	E	•	View	Search	Edit	Datablock	s 🗘
· · · · · · · · · · · · · · · · · · ·	0,	Blendf	file Data	a			0
	3 (9 RNA					
	-	Actio	ns				U
) 🛅		3 🗘 🤇	I P	<mark>?</mark> ∀ 🔮 🛿	8 # 🕹
The the second	R	8 →	🥑 Cu	be			
(1) Cube) V M	lodifie	rs				
💽 🗘 💿 View Select Mesh 😰 Edit Mode 🔷 🗘 🗣 🖓 💭	Ado	Modifie	er			_	÷
Current Up/Dow Na razie dodajmy ten			_		7		
Remove: Backspi	┣ ▽		Bevel	🛅 🤇		znacz (
Execute: Enter							COL
Autocomplete: Ctrl+Space		A	Apply			Сору	
Ctrl +/- Wheel: Zoom		Width	n: 0.100	00 - >)	On	lv Vertices	
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.prop		nit Mat	had			,	
s, bpy.types, bpy.context, bpy.utils, bgl, blf, mathu		nit Met	nou:				
tils		None	2	Angl	e	Weight	
convenience imports: trom matnutils import *; trom ma		Avera	ge	Sharp	est	Largest	:]
>>>]	1

Rysunek 3.3.3 Modyfikator Bevel, dodany do testowego sześcianu

Do szukania elementów siatki odpowiedzialnych za efekt *Bevel*, wykorzystamy *Outliner*. W trybie *Datablocks* pokazuje literalnie całą zawartość pliku. To ładnie przedstawiona struktura *bpy.data* (Rysunek 3.3.4):



Rysunek 3.3.4 Odnajdywanie obiektu w oknie Outliner (tryb Datablocks)

Odnajdź w niej kolekcję *Objects*. Gdy ją rozwiniesz, zobaczysz poszczególne obiekty, dostępne w tej scenie. Przejdźmy do obiektu **Cube** — to nasz sześcian (tak wynika z nazwy, wyświetlanej w *3D View*).

Jeżeli z obiektem jest związana siatka, to znajdziesz ją w polu Data (Rysunek 3.3.5):

⊖ View Search Edit Datablocks	iew Search Edit Datablocks 🔷
🕆 🥥 Camera	Bound Box
⊖ 🥑 Cube	Collision Settings
⊕ RNA	Color
Name	Constraints
Fake User	⊖ Data
Library	⊖∵Cube
Tag	P RNA K
Users	Name
⊕ Active Material	Fake User
Active Material Index	Library
Active Shape Key	Tag
Active Shape Key Index znajdziesz	Users
Animation Data siatkę obiektu	All Edges
Animation Visualisation	Animation Data
Axis-Angle Rotation	Auto Smooth
Bound Box	Auto Smooth Angle
Collision Settings	Auto Texture Space
⊕ Color	Clone UV Texture
Constraints	Clone UV Texture Index
⊕ Data	Double Sided
	Draw Bevel Weights

Rysunek 3.3.5 Wewnętrzna struktura obiektu – zawartość pola Data

(Siatka nazywa się tutaj tak samo, jak obiekt). Gdy przyjrzysz się jej właściwościom, znajdziesz najważniejsze kolekcje: *Vertices, Edges, Faces.* Nas interesują krawędzie — elementy kolekcji *Edges* (Rysunek 3.3.6):

•	File	Add	Render	Help	🕀 Bac	k to Previous)	Blender R	ender (•	Ve:0-8 Ed:
Θ	View	Sear	ch Edit	Dat	ablocks	÷ 🔊			- th) N	o Keying Set
		(⊖ Edges ⊖[)	Mesh E	dge		To w które	/ygląda na go szukan	a pole, 1y!		
			Đ	RNA	-		S N	/leshEdge			
				Bevel 1	Weight		(1	0.00	0	P)	
				Crease	!		(1	0.00	oki	Þ	
				Fgon							0
				Hide							Ŭ
				Index			-	0		Þ	
				Loose							
				Seam							
				Select							
				Sharp							
			Œ	Vertice	25						
			• 🚺	Mesh E	dge						
			€	Mesh E	dge						
			€	Mesh E	dge						
111			000								

Rysunek 3.3.6 Pojedyncza krawędź siatki – element kolekcji Edges.

Rozwińmy pierwszy z nich (*MeshEdge*). Co widzimy? Od razu rzuca się w oczy coś, o co nam chodzi: pole *Bevel Weight*. Jego obecna wartość to 0, co zapewne oznacza brak fazowania. Ale jeżeli zmienimy to na 1.0 (więcej i tak nie można) — to na krawędzi powinna się pojawić faza, prawda?

Spróbujmy (Rysunek 3.3.7):



Rysunek 3.3.7 Próba "ręcznej" zmiany Bevel Weight

Ustawiliśmy *Bevel Weight* pierwszej krawędzi na 1.0 — i nic! Co jest grane!? Przecież z opisu tego pola (zatrzymaj na chwilę kursor myszki, by się pojawił) wynika, że ono właśnie do tego służy!

Może po prostu patrzymy ze złej strony? W końcu nie mamy pewności, gdzie właściwie jest ta krawędź nr 0... Sprawdźmy więc z boków, i od spodu (Rysunek 3.3.8):





Żadna krawędź nie jest sfazowana. Poza tym, jak się dobrze przyjrzeć właściwościom tej krawędzi nr. 0, to coś z nimi jest nie tak. Dlaczego właściwie pole *Select* jest zaznaczone!? Przecież kręciliśmy przed chwilą tym sześcianem w oknie *3D View* na wszystkie strony, i żadna z krawędzi nie była zaznaczona!

51

Spróbujmy się przełączyć w Object Mode (Rysunek 3.3.9):

User Persp 💮	💽 😐 View Search Edit	Datablocks 🗘 🔎
	Edge Length	
	⊖ Edges	12 items
	🔍 💭 Mesh Edge	
	⊕ RNA	😤 MeshEdge
	Bevel Weight	0.000
	Crease	0.000
	Fgon	
	Hide	
	Index	(0)
	Loose	
	Seam	
Gdy przełaczysz się w Object	Select	
Mode, ta sama krawędź ma nagle	Sharp	
inne (poprawne) właściwości!	● Vertices	
7	⊕ ↓ Mesh Edge	
Г ₋ у	● [] Mesh Edge	
(1) Cube	● ↓ Mesh Edge	
(1) Cube	●↓ Mesh Edge	
View Select Object 问 Object Mode 🗘		

Rysunek 3.3.9 Te same właściwości po przełączeniu z Edit Mode na Object Mode

O, ciekawe: wartości wyświetlane w *Outlinerze* uległy zmianie: teraz już ta krawędź nie jest wybrana (*Select* jest wyłączone). Także aktualna wartość *Bevel Weight* jest równa 0. Wygląda na to, ze program ignorował wszystko, co wpisywaliśmy w *Edit Mode*. A może by spróbować przestawić jeszcze raz to *Bevel Weight*? Może w tym trybie będzie działać, skoro np. pole *Select* się "urealniło"?

Zmieniłem wartość *Bevel Weight* w *Object Mode* na 1.0 — i jest faza! (Rysunek 3.3.10):



Rysunek 3.3.10 Rezultat zmiany Bevel Weight w Object Mode

• Obecna wersja Blendera (2.57) w trybie *Edit Mode* ignoruje zmiany siatki wykonywane przez program lub *Outliner*. Aby coś zmienić, trzeba się przełączać w *Object Mode*. Ma to się poprawić w którejś z wersji 2.6

Znaleźliśmy już "programowy" sposób fazowania wybranych krawędzi. Musimy jeszcze znaleźć nazwy API Pythona dla pól, z których będziemy korzystać. (*Outliner* wyświetla "ludzkie" nazwy). Nic prostszego. Blender 2.5 wyświetla tę informację w dolnej części opisu pola (Rysunek 3.3.11):



Rysunek 3.3.11 Identyfikacja nazwy pola w Blender API

Kłopot jest tylko ze zidentyfikowaniem kolekcji, bo dla nich nic się nie wyświetla (Rysunek 3.3.12):



Rysunek 3.3.12 Problem ze zidentyfikowaniem nazwy pola kolekcji

Zazwyczaj kolekcje w Pythonie nazywają się tak samo, ale z małej litery. Każda spacja jest zastąpiona podkreśleniem. Jeżeli jednak chcesz się upewnić, możesz to zweryfikować w tzw. *RNA* klasy (Rysunek 3.3.13):



Rysunek 3.3.13 Identyfikacja pola Pythona w strukturze RNA, które odpowiada kolekcji o nazwie Edges.

Wygląda na to, że dostęp do kolekcji krawędzi siatki to *<object>.data.edges*. Sprawdźmy to zaraz w konsoli Pythona (Rysunek 3.3.14):

Command History: Cursor: Remove: Execute: Autocomplete: Ctrl +/- Wheel:	Up/Down Arrow Left/Right Home/E Backspace/Delete Enter Ctrl+Space Zoom	nd
Builtin Modules: bpy.utils, bgl, blf, Convenience Imports:	<pre>bpy, bpy.data, bp mathutils from mathutils im</pre>	Przypisanie do zmiennej cube aktywnego py.context, obiektu (nasz sześcian)
<pre>>>> cube = bpy.contex >>> cube.data.edges[0</pre>	<pre>kt.active_object 0]</pre>	Spróbuj wyświetlić zawartość pierwszej (nr 0) krawędzi tego sześcianu
<pre>bpy.data.meshes["Cube >>></pre>	e"].edges[0]	Python zwrócił w odpowiedzi odpowiedni obiekt — krawędź 0 siatki Cube. Więc możemy wykorzystywać takie odwołania w naszym kodzie.

Rysunek 3.3.14 Sprawdzanie poprawności odnalezionej "ścieżki" do obiektu w konsoli Pythona Blendera

Najpierw pobieram z kontekstu (*bpy.context*) referencję do aktywnego obiektu (*active_object*). Zachowuję ją w chwilowej zmiennej *cube*. Następnie sprawdzam, czy kolekcja *cube.data.edges* ma element nr 0 (zmienialiśmy go w *Outlinerze*). Jak widać, ma. Sprawdźmy, jaka jest *Bevel Weight* tej krawędzi (Rysunek 3.3.15):



Rysunek 3.3.15 Sprawdzanie, czy pole wybranej krawędzi zwraca oczekiwaną wartość

Jak dotąd wszystko nam działa poprawnie — krawędź nr 0 wykazuje wagę = 1.0.

Warto jeszcze zrobić inny test: zmienić za pomocą Pythona wagę kolejnej krawędzi.

Aby ponownie nie wpisywać całej "ścieżki" do tego wyrażenia, wystarczy że w konsoli naciśniesz kursor (A). W linii poleceń podstawi się wówczas poprzednie wyrażenie Pythona. Wystarczy zmienić w nim indeks z [0] na [1] i przypisać wartość 1.0 (Rysunek 3.3.16).

Gdy wykonasz to polecenie, druga krawędź sześcianu także zostanie sfazowana. Czyli to będzie działać!

(Przy okazji warto zapamiętać, że w konsoli Pythona klawisze **m**/♥ pozwalają sekwencyjnie przywoływać coraz wcześniejsze/późniejsze wyrażenia).



Rysunek 3.3.16 Sprawdzanie zmiany wartości bevel_weight w Pythonie

55

Skoro kluczowe wyrażenia Pythona mamy sprawdzone, czas zacząć pisać nasz skrypt. Zacznij od dodania klauzuli importu modułu *bpy* (Blender API). Potem dodaj nagłówek głównej procedury (Rysunek 3.3.17):



Rysunek 3.3.17 Początek pisania właściwego kodu: deklaracja importu i nagłówek głównej procedury

Aby działało nam uzupełnianie kodu, dodaj na początku dwie "deklaracje typu" dla zmiennych (Rysunek 3.3.18):



Rysunek 3.3.18 Wymuszanie działania autouzupełniania za pomocą "deklaracji typu" zmiennych

Wpiszmy w procedurze pętlę, która zmienia każdej zaznaczonej krawędzi (*edge.select*) wagę fazowania (*edge.bevel_weight*) na 1.0 (Rysunek 3.3.19):



Rysunek 3.3.19 Wpisanie kluczowej pętli

I to wszystko! Po napisaniu procedury nie zapomnij skomentować niepotrzebnych już "deklaracji typu", umieszczonych na początku. (Skomentować, a nie usunąć, bo jeszcze mogą się przydać. Jeżeli nic z nimi nie zrobisz, procedura nie będzie działać poprawnie) (Rysunek 3.3.20):



Rysunek 3.3.20 Dodanie wywołania procedury i skomentowanie pomocniczych "deklaracji"

Na końcu modułu dopisz wywołanie procedury **bevel()** dla aktywnego obiektu (a dokładniej jego siatki — **active_object.data**). Na tak wczesnej fazie pisania nie ma co się jeszcze przejmować sprawdzaniem, czy aktywny obiekt ma w ogóle siatkę.

Zwróć uwagę, że PyDev podkreślił jako błąd właściwość *active_object*. A przecież sprawdziliśmy w konsoli Pythona, że takie wyrażenie jest poprawne. To problem ze strukturą *bpy.context*. W zależności od okoliczności wywołania, ten obiekt może udostępniać różne właściwości! Tego nie jestem w stanie zapisać żadną predefiniowaną klasą! Pełen opis pól jej różnych wersji możesz znaleźć <u>tutaj</u>. Musiałem usunąć dokumentujący je fragment kodu ze skryptu *pypredef_gen.py*, bo nie działał¹. Dlatego w wygenerowanych przeze mnie plikach obiekt *bpy.types.Context* zawiera tylko pola wspólne dla wszystkich struktur. Niestety, nie ma wśród nich *active_object* (są konteksty, w których ta właściwość nie jest dostępna).

Z drugiej strony — to nie jest program kompilowany, i mimo tego błędu możemy go bez problemu uruchomić. W praktyce programowania obiekt aktywny odczytujemy w inny sposób — z kontekstu przekazanego jako parametr do określonej procedury. Wtedy Eclipse nie zgłasza błędu (bo właściwie nie wie, co to za obiekt).

¹ Do dokumentacji struktur kontekstu oryginalny skrypt Campbella Bartona użył "hakerskiej" sztuczki. To bezpośrednie odwołanie do kodu wykonywalnego Blendera jak do współdzielonego obiektu (biblioteki *.*dll* w Windows, albo *shared object* — *.so — pod Linuxem) bez nazwy. Potem skrypt czytał z niego bezpośrednio definicje struktur. Niestety, to chyba działa wyłącznie pod Linuxem, bo próby adaptacji pod Windows spełzły na niczym.

Podsumowanie

- Przygotowaliśmy w Blenderze środowisko testowe dla skryptu. To plik *bevel.blend*. Wśród otwartych paneli znajdują się tam m.in. okna *Outliner* i *Python Console* (str. 48);
- Testowy plik Blendera wygodnie jest umieścić w projekcie Eclipse (str. 48);
- Do przeglądania danych Blendera najlepiej jest wykorzystać Outliner w trybie Datablocks (str. 49);
- Do sprawdzania i zmiany siatek Blendera, posługuj się oknem *Outliner* w trybie *Object Mode* (str. 51 52). W obecnej wersji Blendera (2.5) *Outliner* wyświetla w *Edit Mode* k o p i ę danych siatki. Ta kopia jest wy-konana w momencie przełączenia w tryb edycji (gdy *Outliner* już wyświetlał tę siatkę) lub otwarcia okna *Outliner*. Stan tej kopii może nie odpowiadać aktualnemu stanowi siatki. Wszelkie zmiany, które w niej wy-konasz, są ignorowane. To sytuacja przejściowa, która ma zostać poprawiona wraz z nową implementacją obsługi siatek (tzw. projekt **BMesh**) w Blenderze 2.6;
- Nazwy atrybutów Pythona, odpowiadających polom w oknie Outliner (a także w całym GUI Blendera) są wyświetlane w "dymkach" z opisem (str. 53);
- Aby znaleźć nazwę kolekcji Pythona, odpowiadającej kolekcji w Outliner, należy skorzystać z danych RNA (str. 53);
- Zawsze sprawdź w *Python Console*, czy przygotowane wyrażenie Pythona działa tak, jak się tego spodziewasz (str. 54);
- Obiekt *bpy.context* może zawierać więcej pól, niż w deklaracji jego klasy (*bpy.types.Context*). To atrybuty dynamicznie dodawane przez Blender, w zależności od miejsca (okna) w którym wywołano skrypt (str. 56);

3.4 Uruchamianie skryptu w Blenderze

W poprzedniej sekcji napisaliśmy pierwszy kawałek skryptu, który powinien zadziałać w Blenderze. Można byłoby go teraz uruchomić "metodą tradycyjną": otworzyć w oknie *Text Editor* Blendera, uruchomić. Tyle, że w ten sposób nie można śledzić go w debuggerze. W dodatku w projekt wkrada się zamieszanie. (Gdybyś coś zmienił w kodzie skryptu w Blenderze, trzeba pamiętać aby go zapisać z powrotem na dysk).

Proponuję inne, wygodniejsze rozwiązanie. Otwórz w Blenderze plik *Run.py*, dostarczony wraz z tą książką (por. str. 39). Proponuję umieścić go w oknie powyżej konsoli Pythona (Rysunek 3.4.1):



Rysunek 3.4.1 Dodanie do pliku tekstowego kodu Run.py

Ten plik zawiera kilka prostych linii kodu. Aby zaadaptować go do naszego projektu, zmień wartości przypisane do zmiennych **SCRIPT** i **PYDEV_PATH**, umieszczonych na początku skryptu (Rysunek 3.4.2):



Rysunek 3.4.2 Adaptacja kodu Run.py dla projektu

Zmienna **SCRIPT** powinna zawierać pełną ścieżkę do pliku skryptu, a **PYDEV_PATH** — ścieżkę do folderu Py-Dev, zawierającego folder *pysrc*. (To biblioteka z kodem dla klienta zdalnego debuggera — por. str. 124, 129).

Teraz przygotuj model do testów. Kod, który napisaliśmy, ma sfazować zaznaczone krawędzie siatki. Pisząc go założyliśmy, że obiekt ma już dodany modyfikator Bevel — ustawiony tak, jak na str. 49. Zaznacz więc na testowym sześcianie parę krawędzi, a potem się przełącz w *Object Mode* (Rysunek 3.4.3):



Rysunek 3.4.3 Przygotowanie danych do testu - dwie zaznaczone krawędzie

Wstaw w skrypt punkt przerwania w miejscu, gdzie chcesz zacząć debugowanie. W naszym przypadku dodajmy go na sam początek kodu (Rysunek 3.4.4):

🖻 mesh_bevel 🛛 🖻 bpy	- 8
<pre>Gerry The core bevel import bpy Gedef bevel(mesh): Gerry Gerr</pre>	iia na początku buggera mesh

Rysunek 3.4.4 Umieszczenie punkt przerwania gdzieś na początku kodu

Uruchom proces serwera zdalnego debuggera Pythona (więcej o tym - na str. 124) (Rysunek 3.4.5):



Rysunek 3.4.5 Uruchomienie procesu zdalnego debugger

() 🗘 위※] 4 🛙 Η_ Θ File Add Render Help Default Scene User Persp Ð E=≑ ⊖ 💫 Ble 0 RN
 R Act Arr 1) Cub 🕀 Bru 🕀 Ca D÷ 🮯 Object Mode ¢ ۰. ÷ View Select Object ÷ Global #script to run: SCRIPT = "C:/Documents and Settings/w4979721/workspace/Bevel/src/mesh_bevel =+ #path to your org.python.pydev.debug* folder (it may have different version PYDEVD_PATH='C:/Program Files/eclipse/plugins/org.python.pydev.debug_2.1.0 - <mark>- -</mark> -Ŕ import pydev_debug as pydev Naciśnii ten Rend przycisk pydev.debug(SCRIPT, PYDEVD PATH, trace = True) 60 In Run Script 수 💥 🔚 🖬 👪 Register Display: mat Ē Run.py File: *C:\Program R Laye Up/Down Arrow Command History:

Gdy serwer debuggera już działa, możesz uruchomić skrypt w Blenderze (Rysunek 3.4.6):

Rysunek 3.4.6 Uruchomienie skryptu w Blenderze

Po kilku sekundach okna debuggera Eclipse "ożyją". W oknie edytora zostanie otwarty pomocniczy plik *pydev_debug.py*, a wykonywanie skryptu zatrzyma się na jednej z jego linii (Rysunek 3.4.7):

i 📬 🖌 🗟 🕼 🖊 🎽 🍰 🖉 🖓 🕶 🖉 🔹	🎨 🗘 🔹 🗇 🔹 🔛 😰 🎲 Debug 🥭 PyDev				
Debug 🛛 Naciśnij <i>Resume</i> , aby przejść	🗱 🕬 Breakpoints 🔤 🗖 🗖				
do punktu przerwania na po- czątku naszego skryptu (por.	🖾 🏘 🖻 🖉 💥 💥 🎽				
💥 🕪 💷 🔳 str. 59)	Name Value 🛆				
🖃 芦 Debuggerver [Python Server]	🗄 🔍 Globals 🛛 Global variables 👘				
🖃 🔗 unknown	🔹 🔹 pydev_path 🛛 str: C:/Program Files/eclipse/plugi 🛁				
📄 🔐 MainThread - pid2404_seq1	🗄 🔍 pydevd 🛛 module: <module '<="" 'pydevd'="" from="" td=""></module>				
debug [pydev_debug.py:30] 🚤	script str: C:/Documents and Settings/				
= <module>[run.py:9]</module>	Pierwszy krok debuggera				
	jest zawsze w pomocniczym				
P mesh_bevel P bpy P pydev_debug ⊠					
😑 🧧 #Emulating Blender behavior: tr	y to unregister previous version of t🔼				
#(if it has unregister() method	at all:)				
if script_file in sys.modules:					
try:					
sys.modules[script_file].unregister()					
except:					
pass					
<					

Rysunek 3.4.7 Początek pracy zdalnego debuggera

Plik *pydev_debug.py* jest dodatkiem, który napisałem by ułatwić śledzenie skryptów w Pythonie. Zwróć uwagę, ż wykorzystujemy go w kodzie *Run.py* (por. kod, który pokazuje Rysunek 3.4.6). Szczegółowy opis jego funkcji *debug()* znajdziesz na str. 129. W każdym razie debugowanie będzie się zawsze zatrzymywać w tym miejscu. Zawsze naciskaj tu przycisk *Resume* (F8), by "przeskoczyć" do pierwszego punktu przerwania.

Po naciśnięciu *Resume*, wykonany zostanie cały kod aż do punktu przerwania, który przygotowaliśmy. (Gdyby w kodzie nie było żadnego takiego punktu, skrypt wykonałby się do końca). W naszym przypadku zatrzyma się jednak na początku skryptu (po to wstawiliśmy tam *breakpoint* — por. str. 59) (Rysunek 3.4.10):

i 📬 • 🔛 🖻 🖆 i 🖊 🖗 i 🍅 i 🏇 • 🕥 • i i	🄄 🔶 👻 📑 🔂 🔁 PyDev					
🏂 Debug 🛛 🖓 🗖	🕅= Variables 🕱 💁 Breakpoints 🗖 🗖					
▽	🏝 📲 📄 🖉 💥 💥 🏹					
💥 🗱 🕩 🗉 🔳 🔤 🕹 🐼 🔊 🖉 🖬 🕷	Name Value 🔼					
🖻 🔐 MainThread - pid2404_seq1 🛛 🔨	🗄 🔍 Globals 🛛 Global variables 📄					
	🗄 🔍builtins dict: {'bytearray': <class 'bytearr="" td="" 💻<=""></class>					
ebug [pydev_debug.py:38]	 cached str: C:/Documents and Settings/ 					
<pre></pre>	•doc str: \nThe core bevel code\n 💉					
Debug Server						
P mesh_bevel 🛛 P bpy P pydey_debug	estesmy już w naszym module mesh_bevel.py). Zatrzymaliśmy ie na pierwszym punkcie prze-					
e 🗤 r	wania 🦾 🕺 🔼 🔼					
The core bevel code						
import hny						
<pre>def bevel(mesh):</pre>						

Rysunek 3.4.8 Po naciśnięciu Resume – początek właściwego skryptu

Przejdź przez tę i następne linie głównego kodu (*Step Over* — **F6**), dopóki nie dotrzesz do wywołania funkcji *bevel()* (Rysunek 3.4.9):

i 📬 • 🔛 🕼 🗁 i 🖊 🎽 i 🎒 i 🏇 • 💽 • i	🏷 🔶 🔹 🔛 🔛 🔂 🖓 Debug 🌏 PyDev
🏂 Debug 🛛 🗖 🗖	🗱 🗣 Breakpoints 🔤 🗖 🗖
~	🖆 👫 🖻 🖉 💥 🔆 🎽
💥 🗱 🕩 🗉 🔳 🖉 🞝 🖓 . A → 🔜 👘 🈿	Name Value 🛆
🖻 🔐 MainThread - pid24004_seq1 🛛 🔥	🗄 🔍 Globals 🛛 Global variables 🖉 🗐
== <module>[mesh_bevel.py:18]</module>	🗷 🔍builtins dict: {'bytearray': <class 'bytearr="" td="" 🦷<=""></class>
debug [pydev_debug.py:38]	cached str: C:/Documents and Settings/
<pre> = <module>[run.py:9]</module></pre>	•doc str: \nThe core bevel code\n 🛛 🕑
🚽 🚽 Debug Server	2
Naciśnij	Step Into
P mesh_bevel 23 P bpy P pydev_deb ([F5]), by	wejsc do
wnętrza te	j procedury
for edge in mesh.edges:	
<pre>if edge.select:</pre>	
edge.bevel_weight = 1.0	
+	
bevel(bpy.context.active_object.data)
Nie zwracaj na to oznaczenie błędu	
uwagi (por. str. 56)	

Rysunek 3.4.9 Kolejny krok — "wchodzimy" do wnętrza procedury bevel()

Gdy wywołanie procedury jest podświetlone, naciśnij **F5** (*Step Into*), by ją wykonać krok po kroku.

Wykonaj po kolei wszystkie iteracje pętli. Sprawdzaj, czy kod działa tak, jak powinien, tzn. zmienia *bevel_weight* na 1 tylko dla zaznaczonych (*edge.select* = *True*) krawędzi (Rysunek 3.4.8):

🏇 Debug 🛛 🖓)= \	/ari	ables 🖾 🔏 Breakpoints	
	~			*	• 🖻 🖑 🗶 💥 🎽
💥 🖉 🕩 🗉 🔳 🕅 🤁 🖓 . (2 → 元) i⇒	₹	Nar	ne		Value
🚊 🦓 unknown	^	+	•	Globals	Global variables
😑 🛷 MainThread - pid2404_seq1		+	•	edge	MeshEdge: <bpy_struct< th=""></bpy_struct<>
bevel [mesh_bevel.py:14]		+	•	mesh	Mesh: <bpy_struct, mes<="" th=""></bpy_struct,>
<pre> <module> [mesh_bevel.py:18]</module></pre>		1			>
debug [pydev_debug.py:38]					
= <module>[run.py.♀]</module>	W ior	toć	m	())	
Debug Server	odzie	nrc	nny	durv	
P mesh_bevel 🛛 P bpy P ydev_debug b	evel().	pre			
⊖ #mesh = bpy.types.Nesh					~ =
#edge = bpy.types.MeshEdge					
· · · · · · · · · · · · · · · · · · ·					
for edge in mesh.edges:					
<pre>if edge.select:</pre>					
edge.bevel_weight = 1	.0				
bevel(bpy.context.active_object.d	ata)				×
<					>

Rysunek 3.4.10 Po Weryfikacja działania pętli w procedurze bevel()

Do śledzenia wartości pól obiektu edge użyj panelu Expressions (Rysunek 3.4.11 – por. także str. 126):

🗱 Variables 🤏 Breakpoints	ିଙ୍କୁ Expression	ns 🗸	 W zakładce Expressions	(×	k i	~ •	- 6	7
Name	Value		możesz śledzić aktualny stan zmiennej edge				1	~
*+Y "edge.select"	bool: True							
🦉 "edge.bevel_weight"	float: 0.0							
🐈 Add new expression								



Gdy wykonasz całą procedurę, naciśnij przycisk *Resume*, by szybko dokończyć skrypt (Rysunek 3.4.12):

7	Name **** "edge.select") ⇒ti 📄 🕂 🗯 🛪 Value <error(s) during="" t<="" th=""><th>Cou</th><th></th></error(s)>	Cou			
2	Name *** *** ***	Value	Cou	^		
	*** "edge.select"	cerror(s) during t				
		<pre></pre>	F			
	^{x+y} "edge.bevel_weight"	<error(s)_during_t< td=""><td>ŀ</td><td></td></error(s)_during_t<>	ŀ			
	🐈 Add new expression					
gge do	era nadal nasłuchuje kolejnej sesji debu-		< >			
<pre>#edge = bpy.types.MeshEdge for edge in mesh.edges: if edge.select: adge bevel weight = 1.0</pre>						
		Add new expression Igera nadal nasłuchuje to kolejnej sesji debu- Oczywiście, możesz od r zmieniać skrypt w tym okr	Add new expression Igera nadal nasłuchuje do kolejnej sesji debu- Oczywiście, możesz od razu zacząć zmieniać skrypt w tym oknie edytora	Add new expression Igera nadal nasłuchuje to kolejnej sesji debu- Oczywiście, możesz od razu zacząć zmieniać skrypt w tym oknie edytora		

Rysunek 3.4.12 Stan środowiska po naciśnięciu kończącego Resume

Gdyby wystąpił w skrypcie jakiś błąd, debugger także przerwałby wykonywanie kodu. Mógłbyś wtedy od razu nanieść poprawki w oknie edytora, widocznym w perspektywie *Debug*.

Ale nasz kod okazał się, póki co, bezbłędny. Popatrzmy na nasz testowy sześcian (Rysunek 3.4.13):



Rysunek 3.4.13 Rezultat działania naszego skryptu - poprawnie sfazowane krawędzie

Sześcian też ma sfazowane odpowiednie krawędzie. Wygląda więc na to, że nasz skrypt działa.

Aby powtórnie zacząć debugowanie zmodyfikowanego w Eclipse skryptu, po prostu naciśnij jeszcze raz przycisk *Run Script*. Dopóki masz włączony w Eclipse proces serwera zdalnego debuggera, dopóty reszta zrobi się sama. Znajdziesz się ponownie w miejscu, które pokazuje Rysunek 3.4.7 (str. 60). (Najlepiej raz uruchomionego procesu serwera zdalnego debuggera po prostu nigdy nie zamykać).

Proces serwera zdalnego debuggera możesz wyłączyć w Eclipse dopiero po zamknięciu Blendera. Jeżeli nieopatrznie naciśniesz w Blenderze przycisk *Run Script* gdy serwer debuggera nie jest włączony, skrypt "zamrozi" działanie Blendera. W tym stanie będziesz mógł go tylko zamknąć za pomocą *Meneżera Zadań*. Lepiej więc pamiętać o poprawnej kolejności włączania: najpierw serwer debuggera w Eclipse, a potem *Run Script* w Blenderze.

Proponuję zwinąć okno *Text Editor* z kodem *Run.py* "do paska", taj jak to pokazuje Rysunek 3.4.13, i zapisać ten plik Blendera. Zmodyfikowanego pod potrzeby naszego projektu skryptu nie będziemy już w jego edytorze zmieniać. Wystarczy nam tylko dostęp do przycisku *Run Script*, by wygodnie uruchamiać skrypt *mesh_bevel.py* po kolejnej modyfikacji, wykonanej w Eclipse. Następnym razem Blender otworzy ten plik dokładnie z takim samym układem ekranu. To właściwość bardzo ułatwiająca testowanie!

Podsumowanie

- Skrypt uruchamiamy za pomocą pomocniczego kodu *Run.py*, który należy umieścić w edytorze tekstu Blendera (str. 58);
- Przed pierwszym uruchomieniem, w kodzie *Run.py* należy wpisać odpowiednie ścieżki do pliku skryptu i debuggera PyDev (str. 58);
- Przy pierwszym uruchomieniu debuggera należy włączyć w Eclipse *PyDev Debug Server* (str. 59), a następnie nacisnąć w Blenderze przycisk *Run Script* (str. 60); Do każdego następnego wywołania debuggera wystarczy już tylko naciśnięcie *Run Script*;
- Domyślnie wykonanie programu zatrzymuje się w jednej z linii pomocniczego skryptu o nazwie pydev_debug.py (str. 60); Dlatego warto umieszczać na początku kodu naszego programu jakiś punkt przerwania (str. 59), aby do niego szybko "przeskoczyć" poleceniem Resume (F8);
- Do śledzenia zmian wybranych właściwości obiektów używaj okna Expressions (str. 62);

3.5 Rozbudowa skryptu: wykorzystanie poleceń Blendera

Skoro "jądro" skryptu już działa, pora się zająć pisaniem pozostałych operacji, które ma wykonać. Zacznijmy od przełączenia się z *Edit Mode* do *Object Mode*. Powinniśmy je wywołać na samym początku procedury *bevel()*. (A na samym końcu — wrócić do *Edit Mode*, by nie utrudniać użytkownikowi pracy).

Jak to zrobić za pomocą Blender API? Sprawa wydaje się oczywista: jest w kontekście (*bpy.context*) pole o nazwie *mode*, z którego można odczytać aktualny tryb pracy. (W *Object Mode* zwraca '**OBJECT**', a w *Edit Mode* — '**EDIT_MESH**'. Takie rzeczy zawsze najpierw sprawdzam w konsoli Pythona (Rysunek 3.5.1):

Convenience Imports: from mathutils import to from math import *
>>> bpy.context.mode
Tak byłoby najprościej go zmienić, ale nie można!
>>> bpy.context.mode = 'OBJECT'
File " File " blender_console>", line 1, in <module></module>
AttributeError: bpy_struct: attribute "mode" from "Context" is read-only
>>>

Rysunek 3.5.1 Nieudany test przełączenia na Object Mode poprzez zmianę wartości pola mode aktualnego kontekstu

Więc spróbujmy przypisać polu *Context.mode* nową wartość — to powinno przełączyć nas w odpowiedni tryb, prawda? Zazwyczaj takie pomysły się sprawdzają, ale, jak widać (Rysunek 3.5.1), nie w tym przypadku! *Context.mode* jest *read only*. Może tylko zwracać aktualny stan Blendera. I co dalej z tym zrobimy?

Całe GUI Blendera działa, korzystając wyłącznie z API Pythona. Na pewno więc istnieje sposób, by skrypt mógł zmienić tryb pracy. Tylko jak go znaleźć? "Dymek" z opisem tej kontrolki nie wyświetla żadnego identyfikatora Pythona...

W takich przypadkach wygodniej posłużyć jest się oknem *Info*. Przez cały czas widziałeś jego nagłówek, u góry ekranu. (Menu okna *Info* jest jednocześnie głównym menu Blendera). Powiększ teraz jego obszar, przesuwając nagłówek z menu do dołu (Rysunek 3.5.2):



Rysunek 3.5.2 Sprawdzenie za pomocą okna Info, które polecenie Blender API przełącza w Edit Mode.

Przełączysz się teraz z trybu *Edit Mode* w *Object Mode*. Widzisz? W oknie *Info* pojawiło się coś, co wygląda jak wywołanie funkcji Pythona (Rysunek 3.5.2). To polecenie, którego szukaliśmy!

66

Blender wyświetla we wnętrzu okna *Info* kod Pythona dla każdego polecenia, które wywołałeś za pomocą menu lub przycisku. To taki swoisty "log" aktywności użytkownika. W oknie *Info* pojawią się także jakieś ostrzeżenia lub informacje o błędach. Dlatego czasami warto tu zaglądać.

A jak przełączyć się z powrotem? Sprawdź w oknie *Info*...Tak, to nie jest błąd! Powrót z trybu *Object* w tryb *Edit* uzyskasz, wywołując to samo polecenie — *bpy.ops.object.editmode_toggle()*¹. Skoro już wiemy, jak to się robi, zmodyfikujmy odpowiednio nasz skrypt (Rysunek 3.5.3):



Rysunek 3.5.3 Pierwsza poprawka - przełączenie z Edit Mode na Object Mode na czas operacji

Zmianę trybu mamy już opanowaną. Sprawdźmy więc za pomocą nieocenionego okna *Info*, jakie polecenie Blender API dodaje do obiektu modyfikator *Bevel* (Rysunek 3.4.5):



Rysunek 3.5.4 Sprawdzenie polecenia dodającego do obiektu modyfikator Bevel.

Okazuje się, że to bpy.ops.object.modifier_add(). Jest wywoływana z odpowiednim argumentem type.

¹ Wygląda na to, że tryb *Object* jest dla Blendera trybem "bazowym". Blender API zawiera w różnych modułach *bpy.ops* metody (operatory) pozwalające na przełączenie się pomiędzy *Object Mode* i każdym innym trybem: *object.posemode_toggle()*, *paint.vertex_paint_toggle()*, *paint.vertex_paint_toggle()*, *paint.vertex_paint_toggle()*, *sculpt.sculptmode_toggle()*. Recenzent tej książki zwrócił mi uwagę, że istnieje jeszcze jedna, uniwersalna metoda: *bpy.ops.object.mode_set(mode)*. Możesz jej używać zamiast operatorów *_*toggle()*.

Świetnie! Skoro tak nam dobrze idzie, przełączmy teraz ten modyfikator w potrzebny nam tryb działania. Okno *Info* pokaże nam, jak to się robi w Pythonie. Zmień *Limit Metod* modyfikatora *Bevel* na *Weight*, a typ wagi — na *Largest*. I co? Dlaczego w oknie *Info* nic nowego się nie pojawiło?



Rysunek 3.5.5 Problem z wykryciem przestawienia opcji modyfikatora

Niestety, okno *Info* nie pokazuje wszystkiego. Przełączaniu tych opcji nie towarzyszyło wywołanie żadnego polecenia Blendera (tzw. operatora). To była tylko zmiana wartości pojedynczych pól w obiekcie modyfikatora.

Jakie to pola? Musisz to odczytać "z dymków" objaśnień (Rysunek 3.5.6):



Rysunek 3.5.6 Odczytanie nazw pól obiektu

Z objaśnień można odczytać, że wartości z pierwszego wiersza (*None, Angle, Weight*) to trzy możliwe stany pola **BevelModifier.limit_method**. Z kolei drugi wiersz (*Average, Sharpest, Largest*) to trzy możliwe stany pola **BevelModifier.edge_weight_method** (Rysunek 3.5.6).

W GUI Blendera często w ten sposób przedstawiane są możliwe stany pojedynczego przełącznika

68

No dobrze. Znamy już nazwy pól modyfikatora, które należy zmienić, ale jak się "dostać" do tego obiektu? Odpowiedź na to pytanie znajdziesz, przeglądając strukturę danych naszego sześcianu **Cube** w oknie *Outliner* (Rysunek 3.5.7):

⊖ Objects	I Mode	
t Camera	© Modifiers	1 items
Kolekcja modi	fiers 🔎 🖉 Bevel	
nalezy do oblekt	J! → TRNA	👌 BevelModifier
• RNA	Name	Bevel
Name	Apply on spline	
Fake User	Editmode	C
Library	Expanded	M
Tag	On Cage	<u> </u>
Users	Realtime	R
 Active Material 	Render	4
Active Material Index	Type	Revel
Active Shape Key	Apole	20.00
Active Shape Key Index	Edge Weight Method	30.00
Animation Data	Lissit Method	Largest
Animation Visualisation		Weight V 👻
• Axis-Angle Rotation	Only vertices Py	thon: BevelModifier.edge_weight_method
⊕ Bound Box	Width	0.1000
	Motion Path	
▼	Parent	
🔚 🗘 🔍 View Search Edit D	⊃ View Search Edit Datablocks	• P

Rysunek 3.5.7 Znalezienie kolekcji modifiers

Modyfikatorów nie ma w siatce. Znajdziesz je w samym obiekcie. To kolekcja *Object.modifiers***¹**. Jak odnaleźć obiekt już wiesz, więc mamy zidentyfikowaną całą "ścieżkę" prowadzącą do potrzebnych nam pól.

Pozostaje jeszcze upewnić się, jakie wartości mamy nadać tym atrybutom. Opisy w Blender API są na razie bardzo lakoniczne, czasami po prostu wyliczają możliwe wartości bez żadnego komentarza. Dlatego zawsze wolę sprawdzić w konsoli Pythona, co właściwie zwracają te pola (Rysunek 3.5.8):



Rysunek 3.5.8 Sprawdzenie wartości opcji modyfikatora (w Python Console)

¹ Taki podział oznacza, że możesz wykorzystać tę samą siatkę w dwóch różnych obiektach. Każdy z nich może mieć różne zestawy modyfikatorów: np. jeden będzie używać *Subdivision Surface*, a drugi — nie. W rezultacie możesz z tej samej siatki uzyskać dwa obiekty o zupełnie odmiennym kształcie. Warto o takich rzeczach pamiętać — czasami przydają się do usprawnienia pracy nad modelem!

A jaki operator odpowiada przyciskowi *Apply* modyfikatora? Tu także można posłużyć się "dymkiem" z objaśnieniem (Rysunek 3.5.9):

▼ Modifiers		1			
Add Modifier		÷			
🗢 🛛 Bevel		N Lo x		Niby w cenia, l	tym objaśnieniu podano nazwę pole- które wywołuje przycisk <i>Apply</i> …
Apply		Сору			
Width 0.100 Apply n	nodifier and remove fre	om the stack			
None	Anale	Weight			
Average	Sharpest	Largest			ale w oknie <i>Info</i> możesz zobaczyć także parametry tego wywołania!
			/		
bpy.ops.object.m	odifier_apply(ap	oply_as='DATA'	, modifi	er="Be	vel")
🛈 🗘 🗢 File Ad	ld Render Help (Default		+≍)	Scene

Rysunek 3.5.9 Przykład informacji o tej samej operacji dostarczanej przez okno Info i "dymek" objaśnienia

Jednak gdy wykonasz tę operację, zobaczysz w oknie *Info* dokładniejszą informację, bo wraz z wartościami argumentów tej procedury. W dokumentacji argument *apply_as* opisano: *"How to apply the modifier to the geometry*". Można byłoby się długo domyślać, że należy podstawić 'DATA'.

Mamy już wszystkie informacje, możemy rozbudować naszą procedurę (Rysunek 3.5.10):



Rysunek 3.5.10 Rozbudowa skryptu o dodawanie modyfikatora Bevel

69

Jak widzisz z moich objaśnień (Rysunek 3.5.10), podczas dodawania obsługi modyfikatora zdecydowałem się zmienić typ argumentu procedury **bevel()**. (Bo modyfikatory dodaje się do obiektu, a nie do siatki). Tego rodzaju zmiany wprowadzaj bardzo ostrożnie. W takiej sytuacji najłatwiej zapomnieć o jakimś miejscu w kodzie, którego ta zmiana dotyczy. A potem będzie to powodem błędu.

Kod, który przedstawia Rysunek 3.5.10, ma pewną wadę. Został napisany "pod dane testowe". Podczas normalnego używania może się zdarzyć, że wywołasz go dla obiektu, który już ma przypisany jakiś modyfikator. Na przykład — wygładzanie *Subsurf* (Rysunek 3.5.11);

▼ Modifiers	_ Tylko pierwszy modyfikator może być "utrwalony"
Add Modifier	poleceniem <i>Apply</i> bez żadnego wpływu na efekty działania pozostałych modyfikatorów
Image: Subsurf Image: Subsur	Każdy nowy modyfikator jest dodawany na koniec stosu. Musimy więc przesunąć nasz Bevel "do góry"!



Nowy modyfikator, taki jak nasz *Bevel*, jest zawsze dodawany na koniec listy (stosu). To niedobrze, bo do poprawnego efektu operacji *Apply* ("utrwalenia" modyfikatora) musi być na początku. Za pomocą okna *Info* szybko znajdziesz, że do przesuwania modyfikatora "w górę" służy operator *bpy.ops.object.modifier_move_up()*. Musimy go zastosować w pętli, przesuwając nasz modyfikator do góry dopóty, dopóki nie stanie się pierwszym (Rysunek 3.5.12):



Rysunek 3.5.12 Dokończenie implementacji operacji na modyfikatorze

Na koniec dodałem do kodu *bevel()* drugi argument: szerokość fazowania (*width* — por. Rysunek 3.5.12). W testowym wywołaniu procedury jest ustawiony na 0.1 jednostki Blendera. Pozostaje tylko przygotować środowisko do testów (Rysunek 3.5.13):



Rysunek 3.5.13 Dane do nowego testu

Zaznacz parę krawędzi siatki, i naciśnij przycisk *Run Script*. Oczywiście, jak każde pierwsze uruchomienie po poważnych zmianach, śledziłem wykonanie skryptu w debuggerze. Szczęśliwie zakończył się bez błędów. Rysunek 3.5.14 przedstawia uzyskany rezultat:



Rysunek 3.5.14 Rezultat działania nowej wersji skryptu

Wygląda to tak, jak powinno: zaznaczone krawędzie zostały sfazowane. Warto jednak jeszcze sprawdzić, jak wygląda rozkład wartości *Bevel Weight* na nowych krawędziach siatki (Rysunek 3.5.15):



Rysunek 3.5.15 Efekt propagacji Bevel Weight na nowe krawędzie

W tej chwili na siatce zaznaczone są nowo dodane krawędzie fazki (por. Rysunek 3.5.14). Wygląda na to, że tylko zaznaczone krawędzie (ale nie wszystkie!) mają wartości *Bevel Weight* różne od zera. Nie znalazłem żadnej nie zaznaczonej krawędzi w testowym sześcianie, która miałaby *Bevel Weight* > 0.. Zakładam więc, że Blender przenosi wartości *Bevel Weight* (= 1.0) oryginalnych krawędzi na krawędzie rezultatu. To mogłoby przeszkodzić w wykonaniu kolejnego fazowania. (Bo zostałyby zmodyfikowane nie tylko zaznaczone krawędzie, ale także inne, które odziedziczyły *Bevel Weight* z poprzednich wywołań). Na koniec działania skryptu należy więc wyzerować *Bevel Weight* dla zaznaczonych krawędzi siatki (Rysunek 3.5.16):

```
def bevel(obj, width):
    """Bevels selected edges of the mesh
      Arguments:
            Cobj (Object): an object with a mesh.
                           It should have some edges selected
            Qwidth (float):width of the bevel
       This function should be called in the Edit Mode, only!
    .....
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier
   bpy.ops.object.editmode toggle() #switch into OBJECT mode
    #adding the Bevel modifier
   bpy.ops.object.modifier add(type = 'BEVEL')
   bevel = obj.modifiers[-1] #the new modifier is always added at the end
   bevel.limit_method = 'WEIGHT'
   bevel.edge weight method = 'LARGEST'
   bevel.width = width
    #moving it up, to the first position on the modifier stack:
    while obj.modifiers[0] != bevel:
        bpy.ops.object.modifier move up(modifier = bevel.name)
    for edge in obj.data.edges:
        if edge.select:
            edge.bevel weight = 1.0
   bpy.ops.object.modifier apply(apply as = 'DATA', modifier = bevel.name)
    #clean up after applying our modifier: remove bevel weights:
    for edge in obj.data.edges:
        if edge.select:
                                                Usuwanie wag z zaznaczonych
            edge.bevel weight = 0.0
                                                krawędzi
   bpy.ops.object.editmode toggle() #switch back into EDIT MESH mode
```

Procedura **bevel(object, width)** jest już gotowa. Zwróć uwagę, że na razie nie zawiera żadnego kodu sprawdzającego poprawność danych wejściowych. To pojawi się w następnym rozdziale, gdzie nasz skrypt przerobimy w dodatek (*add-on*) Blendera.

Rysunek 3.5.16 Ostateczna postać procedury bevel()
Podsumowanie

- Kod Pythona operatorów, wywoływanych przez GUI Blendera można śledzić w oknie *Info* (str. 66). To okno nie pokazuje jednak wszystkich działań użytkownika np. zmiany wartości atrybutów obiektu (str. 67);
- Nazwy pól (atrybutów) obiektów Pythona, odpowiadających elementom GUI, należy odczytywać z wyświetlanych przez Blender podpowiedzi (w "dymkach" — str. 67);
- Wartości, przypisane atrybutom obiektu, najlepiej jest sprawdzić w Python Console (str. 68);
- Wywołanie operatora to zwykłe wywołanie metody jakiegoś obiektu *bpy.ops*. Podstawową techniką programisty jest łączenie ich z kodem, sprawdzającym stan odpowiednich danych Blendera. Na przykład w ten sposób zaimplementowaliśmy przesuwanie nowo dodanego modyfikatora na początek stosu (str. 70);

Rozdział 4. Przerabianie skryptu na wtyczkę Blendera (add-on)

Zapewne parę razy zaglądałeś do okna *User Preferences* Blendera. Przypuszczam, że już zwróciłeś uwagę na zakładkę *Add-Ons*:

Blender User Preferences						
Interface Ed	liting	Input	Add-Ons	Themes	File	System
All Enabled Disabled 3D View Add Curve Add Mesh Animation	 Mesh: Bsur Description: Location: Author: Version: Warning: Internet: 	faces GPL Editi Draw mesh View3D > E Eclectiel 0.9 M Beta M Lin	ion es and re-topologies wi EditMode > ToolShelf k to the Wiki	th Grease Pencil	Tu kliknij, by włączyć / wył wtyczkę	ączyć
Development	Mesh: Fake	e Knife				
Import-Export Mesh	D Mesh: Inter	rsection				
Object	D Mesh: Loop	Tools				× -
Render Rigging	D Mesh: Rela	x				<u>ا</u> چ
System				⊳		
Official Community						

Add-on to dodatek (wtyczka) do Blendera, napisana w Pythonie. To okno pozwala wybrać zestaw dodatków, z których obecnie chcesz korzystać. Wtyczki podczas inicjalizacji dodają do GUI Blendera nowe elementy: przyciski, polecenia menu, panele. Zresztą cały interfejs użytkownika Blendera jest napisany w Pythonie, z użyciem tych samych poleceń API, których używają *add-ons*.

W tym rozdziale pokażę, jak przerobić naszą procedurę na dodatek do Blendera. Ten *add-on* będzie dodawał do menu *Specials* polecenie (operator) o nazwie *Bevel*.

4.1 Dostosowanie struktury skryptu

Do tej pory nasz skrypt był "linearny": wykonywało się to, co było wpisane w kodzie głównym, i koniec. Wtyczki Blendera działają inaczej, o czym przekonasz się w tej sekcji. W związku z tym ich skrypt musi mieć określoną strukturę. Poznasz ją, przy okazji przerabiania naszego kodu na wtyczkę.

Zacznijmy od nagłówka. Każda wtyczka Blendera musi zawierać globalny słownik o nazwie **bl_info**. Kluczami tego słownika są ściśle określone teksty: **"name"**, **"autor"**, **"location"**, itp. Blender wykorzystuje tę strukturę do wyświetlenia opisu wtyczki w oknie *Add-Ons* (Rysunek 4.1.1):



Rysunek 4.1.1 Struktura nagłówka wtyczki i jej reprezentacja w oknie User Preferences.

Niektóre wartości słownika możesz pozostawić puste — np. odsyłacze do opisu i raportu błędów ("wiki_url", "tracker_url"). Istotnym polem jest "category": używaj tu wyłącznie jednego z tekstów, które widzisz na liście kategorii w zakładce *Add-Ons*. Jeżeli wpiszesz coś, czego tam nie ma — Twój dodatek będzie widoczny tylko po wybraniu kategorii *All*.

Wtyczka wstawi naszą metodę **bevel()** do listy poleceń Blendera. Aby to było możliwe, musimy naszą procedurę "obudować" prostą klasą operatora (Rysunek 4.1.2):

<pre># ### Operator class Bevel(bpy.types.Operator): ''' Bevels selected edges of the mesh''' bl idname = "mesh.bevel"</pre>	Tu ustalasz, pod jaką nazwą operator będzie wi- doczny w Blender API: bpy.ops.mesh.bevel() . Pisz ten tekst małymi literami!
bl_label = "Bevel"	Nazwa polecenia (np. do wyświetlenia w menu)
<pre>bl_description = "Bevels selected edges" def everyte(self context):</pre>	Opis polecenia, wyświetlany "w dymku"
bevel(context.active_object,0.1)	Na razie zakładamy stałą szerokość: 0.1



Nadałem tej klasie nazwę **Bevel** (nazywaj je jak chcesz). Nowy operator musi koniecznie pochodzić od abstrakcyjnej klasy **bpy.types.Operator**. Inaczej nie będzie działał. 76

W klasie operatora koniecznie zdefiniuj pola *bl_idname*, *bl_label* (Rysunek 4.1.2). Możesz także dodać pole *bl_description*. (Jeżeli go nie ma, Blender wyświetla w "dymku" z opisem polecenia komentarz typu *docstring*, umieszczony pod nagłówkiem klasy). Na razie nasza klasa będzie miała jedną metodę, o ściśle określonej nazwie i liście parametrów: *execute(self, context)*. Umieść w niej wywołanie naszej procedury *bevel()*, na razie z szerokością fazowania wpisaną na stałe. Na tym etapie prac jeszcze nie dodaję żadnego kodu sprawdzającego poprawność danych wejściowych (kontekstu wywołania).

Aby Blender zarejestrował wszystkie takie klasy specjalne, zdefiniowane w Twoim module, musisz dodać do skryptu odpowiednie funkcje odpowiedzialne za "rejestrację". Ten kod praktycznie zawsze wygląda tak samo: na początku skryptu dodaj import z *bpy.utils* dwóch pomocniczych funkcji. Następnie na końcu skryptu wykorzy-staj je w procedurach o nazwie (koniecznie!) *register()* i *unregister()* (Rysunek 4.1.3):



Rysunek 4.1.3 Kod rejestrujący w środowisku Blendera klasy zdefiniowane w skrypcie.

Sprawdźmy teraz działanie tak przygotowanego skryptu. Upewnij się, że serwer debuggera jest włączony. Ustaw odpowiednie środowisko w Blenderze, a potem naciśnij przycisk *Run Script* (Rysunek 4.1.4):



Rysunek 4.1.4 Uruchomienie wtyczki w debuggerze.

I co? W Eclipse wygląda na to, że skrypt się zakończył, a krawędzie sześcianu nie są sfazowane? Upewnij się raz jeszcze: dodaj do procedury *Bevel.execute()* punkt przerwania, i uruchom skrypt jeszcze raz. Nadal to samo (Rysunek 4.1.5):

🏇 Debug 🖾 Wygląda na to, że skrypt się zakończył	🕪= Variables 🔍 Breakpoint 🖓 Expression 🛛 🖓 🗖
	┤ <u></u>
💥 🗱 🕪 🗉 🔳 🖓 R. 👁 🖛 🠺 i⊅ 😿	Name Value
🖃 🏓 Debug Server [Python Server]	Add new expression
🖨 🞯 unknown	
MainThread - pid5076_seq1	ale serwer debuggera
Debug Server	od klienta, uruchomionego
	w Blenderze!
P mesh_bevel 🛛 🕐 pydev_debug 🗳	a metoda execute()
Class Bevel (bpy.types.Operator) :	jeszcze nie została ani przez 🛒 💽 😜 🏹
''' Bevels selected edges of the	mess type filter text
bl_idname = "mesh.bevel"	
bl_label = "Bevel"	 bl info
bl_description = "Bevels selected	a eag ### Imports
	, bpy
er execute(self, context):	📄 🚽 register_module,unregister_module (bpy.
bever(concext.active_object,c	# ### Core operation
# ### Register	🗸 👘 🧶 bevel

Rysunek 4.1.5 Debugger po naciśnięciu przycisku Resume (F8)

Rzecz polega na tym, że obecnie główny kod skryptu wcale nie wywołuje procedury **bevel()**. On tylko rejestruje nowe polecenie (operator) Blendera, o takiej nazwie, jaką wpisałeś w **Bevel.bl_idname**. W naszym przypadku to **"mesh.bevel**" (por. str. 75, Rysunek 4.1.2). Sprawdź w konsoli Pythona, czy istnieje procedura **bpy.ops.mesh.bevel** (Rysunek 4.1.6):

<pre>>>> bpy.ops.mesh.bevel # Bevels selected edges of the mesh</pre>	Wpisz nazwę procedury bez nawiasu "()"	
<pre>bpy.ops.mesh.bevel() >>></pre>	A Python wyświetli jej opis i przykład wywołania!	
💽 🗘 🗢 Console Autocomplete		

Rysunek 4.1.6 Sprawdzenie, czy w Blenderze pojawiło się nowe polecenie (operator)

Taki operator można teraz dodać do jakiegoś menu, albo umieścić gdzieś jako przycisk. Integracją z GUI zajmiemy się jednak w następnej sekcji tego rozdziału. Na razie po prostu wywołajmy to polecenie "z ręki" — w konsoli Pythona (Rysunek 4.1.7):



Rysunek 4.1.7 Testowe wywołanie operatora...

Ekran Blendera uległ zamrożeniu, a w naszym Eclipse włączył się debugger. Aktualna pozycja w kodzie to wstawiony wcześniej punkt przerwania wewnątrz procedury *execute()* (Rysunek 4.1.8):



Rysunek 4.1.8 ... i proces debuggera zatrzymuje się na punkcie przerwania

Widzisz? Zasymulowaliśmy tutaj to, co będzie robił z naszym operatorem Blender. Gdy wywołasz polecenie **bpy.ops.mesh.bevel()** (zazwyczaj poprzez polecenie z menu lub przycisk), Blender utworzy nową instancję klasy **Bevel**. Zrobi to tylko po to, by wywołać jej metodę **Bevel.execute()**. Po wykonaniu tej procedury obiekt będzie zaraz zwolniony (tzn. usunięty). Taki sposób działania ("nie wołaj nas, to my wywołamy ciebie") jest typowy dla większości środowisk graficznych.

Przy okazji: zwróć uwagę na parametry procedury, dostępne w oknie *Variables*. Rozwiń np. parametr *context*, aby przekonać się, jak od strony programu wygląda kontekst wywołania naszego operatora (Rysunek 4.1.9):

(×)= 1	/ari	ables 🛛 🗣 Breakpoints	🚱 🕫 🕞 🦨 💥		5
Nar	me		Value		^
±	•	Globals	Global variables		
	•	context	Context: <bpy_struct, 0x01f55d40="" at="" context=""></bpy_struct,>		
	÷	active_base	ObjectBase: bpy_struct, ObjectBase at 0x0C988488>		
		 active_bone 	NoneType: None		
	÷	active_object	Object: <bpy_struct, object("cube")=""> Wygląda na t</bpy_struct,>	to, że w	/ tym
		 active_pose_bone 	NoneType: None tylko active	st podaj	e nie ale
	÷	🔍 area	Area: http://www.area.at/0x0CB17A90> także dwa in	nne pol	a —
	÷	bl_rna	Context: <bpy_struct, struct("context")=""> edit_object i</bpy_struct,>	object	
	÷	blend_data	BlendData: truct, BlendData at 0x0CB06AA0>		
	÷	edit_object	Object: <bpy_struct, object("cube")=""></bpy_struct,>		
		 editable_bones 	NoneType: None		
		image_paint_object	NoneType: None		
		• mode	str: EDIT_MESH		
	+	object	Object: <bpy_struct, object("cube")=""> 🚽</bpy_struct,>	J	

Rysunek 4.1.9 Podgląd zawartości kontekstu wywołania operatora

Struktura *context* może mieć różne pola dla różnych trybów pracy Blendera. Przeglądając ją, zawsze można odkryć jakiś ciekawy szczegół. Na przykład — co to za pola *object* i *edit_object*? Niestety, na razie na <u>stronach</u> <u>dokumentacji Blender API</u> żadne pole kontekstu (moduł *bpy.context*) nie ma jakiegokolwiek opisu.

Przyjrzyjmy się jeszcze w oknie *Variables* samemu obiektowi **self**. Zwróć uwagę, że to klasa **Bevel** ma tutaj wpisane inne klasy bazowe. Zmieniła się także wartość jej pola **bl_idname** (Rysunek 4.1.10):

(×)= Variab	les 🛛 🔍 💁 Breakpoints	1 🙀 Expressions 👘 🖘 🍇 🏹 🗖	8		
Name		Value			
🗉 🔍 Globals		Global variables			
🗉 🔍 context		Context: ty_struct, Context at 0x01F55D40>			
🖃 🔍 self		Bevel: by_struct, MESH_OT_bevel("MESH_OT_bevel")>			
•	bl_description	str: Bevels selected edges			
•	bl_idname	str: MESH_OT_bevel			
•	bl_label	str: Bevel Zmienione klasy bazowe i			
. E (bl_options	set: set() wantosc bi_funame:			
. E (▶ bl_rna	Bevel: <bpy_struct, struct("mesh_ot_bevel")=""></bpy_struct,>			
•	cancel	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
•	check	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
•	' draw	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
•	has_reports	bool: False			
•	invoke	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
•	layout	NoneType: None			
•	modal	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
•	name	str: Bevel			
± C	🕨 order	list: ['module', 'doc', 'bl_idname', 'bl_label', 'bl_description', 'exec			
•	poll	str: Traceback (most recent call last):\n File "C:/Program Files/eclipse/plug			
± C	properties	MESH_OT_bevel: <bpy_struct, 0x0cb63dd8="" at="" mesh_ot_bevel=""></bpy_struct,>			
± C	rna_type	MESH_OT_bevel: <bpy_struct, struct("mesh_ot_bevel")=""></bpy_struct,>			
str: B	evel		~		
			V		
<		>			

Rysunek 4.1.10 Podgląd zawartości naszej klasy

Od razu uspokajam: to normalne. Wygląda na to, że Blender kierując się pierwszym członem oryginalnej wartości *bl_idname* ("<u>mesh.bevel</u>") stworzył na potrzeby naszego operatora klasę *MESH_OT_bevel*. (Słowo "mesh." jest zastępowane w nazwie klasy przez "*MESH_OT_*". Wygląda na to, że Blender podstawia na miejsce kropki (".") w operatorze symbol "*OT_*"). Wyświetl zawartość przestrzeni nazw *bpy.types* (np. wpisz polecenie *dir(bpy.types)* w konsoli Pythona). Zobaczysz wtedy masę nie udokumentowanych klas, zawierających w nazwie "_OT_", "_MT_", albo "_PT_". To wszystkie menu i panele GUI Blendera!

Przy okazji: zerknij także na aktualny stan stosu, na którym jest wykonywany nasz skrypt (Rysunek 4.1.11). Porównaj go np. ze stosem przedstawianym przez przez Rysunek 3.4.7 (str. 60), albo Rysunek 3.4.10 (str. 62).

U dołu stosu są funkcje obsługujące *Python Console* (jak widać, duża jej część jest także napisana w Pythonie). Potem jest wywołanie pierwszej linii w chwilowym skrypcie "*<blender console>*". To polecenie, które wpisaliśmy. Jak widać, spowodowało wywołanie modułu *ops.py*, który z kolei stworzył instancję klasy *Bevel* i wywołał jej metodę *execute()*.



Rysunek 4.1.11 Stos wywołania z konsoli Pythona

Gdy wykonasz linię z wywołaniem *bevel()* (*Step Over* — **F6**) debugger przeniesie Cię do skryptu Blendera *ops.py*, który ją wywołał (Rysunek 4.1.12):



Rysunek 4.1.12 Polecenie wykonywane po opuszczeniu procedury Bevel.execute()

Specjalnie doszedłem tutaj, by pokazać Ci zachowanie się debuggera PyDev w przypadku wystąpienia błędu. W takiej sytuacji znika zielonkawe podświetlenie aktualnej linii (Rysunek 4.1.13):



Rysunek 4.1.13 Stan debuggera po wystąpieniu błędu

Jednocześnie w konsoli Blendera debugger wypisuje komunikat o nazwie skryptu i numerze linii, w której błąd wystąpił. Mimo to wykonywanie skryptu nie zostało jeszcze zakończone. W panelu *Debug* widzisz nadal zawartość stosu. W panelu *Variables* możesz sprawdzić aktualny stan zmiennych — lokalnych i globalnych. Zazwyczaj dzięki analizie ich zawartości będziesz mógł szybko zdeterminować przyczynę problemu. Tylko jednego elementu brakuje mi wśród tych informacji: tekstu komunikatu o błędzie! Przyznam się, że do tej pory nie znalazłem miejsca, w którym PyDev mógłby go wyświetlić. Gdy nie wiemy, co się nie zgadza, trudno jest szukać przyczyny... W każdym razie gdy chcesz zakończyć tak przerwany skrypt — naciśnij przycisk *Resume* (F8). Wtedy dopiero zobaczysz informację o błędzie (Rysunek 4.1.14):

Convenience Imports: from mathutils import *; from math import *
>>> bpy.ops.mesh.bevel()
rror: TypeError: calling class function: Function.result expected a strin enum or a set of strings in ('RUNNING_MODAL', 'CANCELLED', 'FINISHED', ' PASS_THROUGH'), not NoneType
.ocation:C:\PROGRA~1\Blender\2.57\scripts\modules\bpy\ops.py:179
Console Autocomplete

Rysunek 4.1.14 Komunikat o błędzie

To z kolei trochę "musztarda po obiedzie", bo teraz, gdy znamy przyczynę, możesz chcieć obejrzeć stan jakiejś zmiennej. W tym momencie jest to jednak niemożliwe – skrypt zakończył już działanie (por. stan stosu, który pokazuje Rysunek 4.1.15). W praktyce zazwyczaj pracuję tak, że za pierwszym wystąpieniem błędu pozwalam się skryptowi "dokończyć", aby mógł wyświetlić komunikat. Dysponując już jego opisem, wstawiam w linię, w której wystąpił, punkt przerwania. Potem ponownie wywołuję skrypt, by błąd wystąpił powtórnie. Za drugim razem analizuję stan zmiennych, i dochodzę do przyczyny problemu.

Komunikaty o błędzie polecenia wywołanego z konsoli Pythona (*Python Console*) pojawią się poniżej wywołania, tak jak pokazuje to Rysunek 4.1.14. Komunikaty o błędzie poleceń wywoływanych z GUI Blendera — menu, przycisku, itp. — pojawią się w konsoli Blendera (*System Console* — por. str. 127, Rysunek 6.3.8).

W tym konkretnym przypadku taka złożona analiza nie była konieczna. Blender wyraźnie napisał, że nie podoba mu się wartość zwracana przez funkcję (chodzi zapewne o **Bevel.execute()**). Istotnie, w pośpiechu pisania kodu zupełnie zapomniałem, że **execute()** musi zwracać jedną z wartości wyliczenia, podanego w komunikacie. Zazwyczaj chodzi o '**FINISHED**'. Poprawmy od razu nasz skrypt (Rysunek 4.1.15):

🏇 Debug 🛿 🖓 🖓	🗱 Variable 🤷 Breakpo 🙀 Express 🕅	
🌺 🗱 🕩 🔲 🔳 🕺 🔍 🖘 🐟 🖶 i> 😿 🎽	🌆 🛋 🖨 🕹	× 🙀 🏱
🖃 🏓 Debug Server [Python Server]	Name	Value
🖃 👸 unknown	*** "sys.exc_info()"	<error(s)_durin< th=""></error(s)_durin<>
MainThread - pid5076_seq1	🕂 Add new expression	
🔤 Debug Server		
	<u><</u>	>
	Could not resolve variable	x
	<	>
🖻 *mesh_bevel 🛛 🖻 bpy		- 8
		<u>^</u>
der execute(sell, context):		
<pre>bevel(context.active_object,U.)</pre>	1) Funkcia avaguta() musi zuragaá	
return { 'FINISHED' }	kolekcie z jedna z dopuszczala	
# ### Register	nych wartości wyliczenia	
⊖def register():	,	_
register_module(name)		
		<u> </u>

Rysunek 4.1.15 Szybka poprawka kodu - od razu, w perspektywie Debug

Teraz wystarczy, abyś zapisał zmodyfikowany skrypt na dysku. Potem naciśnij przycisk *Run Script*, aby go "przeładować" w Blenderze. Na koniec wywołaj powtórnie to polecenie w konsoli Pythona (Rysunek 4.1.16):



Rysunek 4.1.16 "Przeładowanie" wtyczki i powtórne uruchomienie polecenia

Jak widać, po wprowadzeniu poprawki nasz operator zaczął działać. Teraz można go dodać do menu *Specials* (por. str. 34). W następnej sekcji pokażę, jak to zrobić.

Podsumowanie

- Każda wtyczka musi zawierać słownik *bl_info* (str. 75). To jej "metryczka", używana do wyświetlania informacji w oknie User Preferences:Add-Ons;
- Procedurę, która coś zmienia w danych Blendera (jak nasz bevel()) należy przekształcić w operator. Polega to na stworzeniu klasy pochodnej bpy.types.Operator. Procedurę należy wywołać w metodzie execute() tej nowej klasy (str. 75);
- Wtyczka musi implementować procedury register() i unregister() (str. 76);
- Uruchomienie wtyczki oznacza tylko jej zarejestrowanie (wykonanie metody *register()* skryptu str. 77).
 Operator, który implementuje, trzeba wywołać np. z konsoli Pythona (str. 77 78). Powoduje to stworzenie przez Blender nowej instancji klasy operatora, i wywołanie jego metody *execute()*;
- Przycisk *Run Script* służy wyłącznie do załadowania aktualnej wersji wtyczki. (Wywołuje procedurę unregister() dla starej wersji i register() dla nowej por. str. 82, 129);
- Informacji o otoczeniu (kontekście) wywołania operatora selekcji, aktualnej scenie, itp. dostarcza metodzie *execute()* argument *context* (str. 78);
- Gdy w programie wystąpi błąd (tzn. sygnalizowany jest wyjątek *exception*), debugger Eclipse zatrzymuje wykonywanie kodu (str. 80). Można w tym momencie sprawdzić stan zmiennych. Niestety, nie znalazłem nigdzie miejsca, gdzie można byłoby już zobaczyć komunikat o błędzie. Ten tekst zostanie wyświetlony w konsoli Blendera gdy pozwolisz skryptowi "wywalić się do końca" (poleceniem *Resume* str. 81). (Konsola Blendera to tzw. *System Console* por. str. 127. Nie myl jej z *Python Console*!);

4.2 Dodanie polecenia (operatora) do menu

Gdy dodajemy operator do menu, czas wreszcie zadbać o sprawdzenie poprawności danych wejściowych. Poprawne działanie procedury *bevel()* wymaga od siatki spełnienia dwóch warunków:

- 1. musi być włączony tryb *Edit Mode*;
- 2. muszą być zaznaczone jakieś krawędzie;

Zajmijmy się na razie pierwszym warunkiem. Co prawda menu *Specials*, do którego zamierzamy dodać nasze polecenie, jest dostępne wyłącznie gdy siatka jest w trybie *Edit Mode*. Nigdy jednak nie możesz być pewnym, czy ktoś w przyszłości nie doda wywołania naszego operatora do innego menu lub panelu. Dlatego zawsze warto umieścić w klasie każdego polecenia metodę *poll()* (Rysunek 4.2.1):



Rysunek 4.2.1 Dodanie podstawowego "testu na widoczność" - procedury poll()

Blender wywołuję tę funkcję, gdy się dowiedzieć czy "w obecnej sytuacji" polecenie może być w ogóle wyświetlone. "Obecną sytuację" opisuje argument *context*. To instancja klasy *bpy.types.Context* (poznaliśmy już ten rodzaj obiektu — por. str. 78, 56, 54). Kod *poll()* po analizie otrzymanego obiektu *context* ma zwrócić *True*, jeżeli operator może być użyty. W przeciwnym razie powinien zwrócić *False*.

W tym miejscu raczej należy umieszczać "zgrubne" testy, właśnie takie, jak warunek 1. W naszej implementacji funkcja *poll()* zwraca *True*, jeżeli dla siatki włączono *Edit Mode*. (Takie znaczenie ma stała '**EDIT_MESH**'. Gdybyśmy byli w trybie edycji np. armatury, *context.mode* zwróciłby inną wartość).

• W kodzie metody *poll()* nie umieszczaj nigdy żadnych poleceń, które coś zmieniają w Blenderze. (Chodzi np. o zmianę trybu pracy, albo zawartości sceny). Program nie pozwoli im się w tym miejscu wykonać

Zwróć uwagę na wyrażenie @*classmethod*, poprzedzający definicję metody *poll()*. (W slangu programistów to tak zwany "dekorator"). To deklaracja, że metoda jest metodą klasy, a nie obiektu (instancji)¹.

 Pamiętaj, żeby nigdy nie zapomnieć o wpisaniu "dekoratora" @classmethod przed definicją funkcji poll()! Jeżeli go pominiesz, Blender nigdy nie wywoła tej metody.

Swoją drogą, czy zwróciłeś uwagę że w przypadku wtyczki API Blendera wymaga od Ciebie implementacji ściśle określonych metod? To taki "kontrakt" pomiędzy Twoim skryptem a resztą systemu. Ty zobowiązujesz się przygotować klasę o określonych funkcjach. Blender zobowiązuje się wywoływać je w ściśle określonej sytuacji.

¹ To zapewne dla poprawienia wydajności pracy całego środowiska Blendera. Metodę *poll()* implementują wszystkie elementu GUI, a system co chwila je wywołuje. (Funkcje *poll()* są wywoływane gdy cokolwiek ulega zmianie — tryb pracy, zawartość obiektu, itp.). Gdyby *poll()* była zwykłą metodą obiektu, tak jak *execute()*, Blender musiałby za każdym razem tworzyć nowe instancje klasy, wywołać ich funkcje *poll()*, i zaraz potem je zwalniać. Przypuszczam, że wtedy wszystko działało by wolniej, być może nawet zbyt wolno. Do wywołania metody klasy nie trzeba tworzyć instancji (nowych obiektów), i w związku z tym nie obciąża to tak bardzo procesora.

Listę tak uzgodnionych funkcji i właściwości nazywa się w programowaniu obiektowym "interfejsem". Aby ułatwić Ci nieco zadanie, w Blender API znajduje się gotowy "wzorzec" klasy operatora: *bpy.types.Operator*¹. W języku programowania obiektowego *Operator* jest tzw. "klasą abstrakcyjną". Sama niczego specjalnego nie robi, dostarcza co najwyżej domyślnych, pustych implementacji wszystkich metod, przewidzianych w interfejsie wtyczki. Nasz operator dziedziczy te implementacje po klasie bazowej (właśnie *bpy.types.Operator*). Dlatego możemy w naszej klasie *Bevel* implementować (nadpisywać) tylko te z metod klasy *Operator*, które nie mają działać w sposób domyślny.

Warunku 2 ("siatka musi mieć jakieś zaznaczone krawędzie") nie będziemy sprawdzać w metodzie *poll()*. Jest zbyt szczegółowy. Pomyśl, co to za polecenie, które pojawiałoby się w menu tylko wtedy, gdy coś jest na siatce zaznaczone? Połowa użytkowników nie miałaby szczęścia go w ogóle zobaczyć, i doszłaby do wniosku, że wtyczka nie działa. Już lepiej pozwolić, aby polecenie *Bevel* było w menu *Specials* przez cały czas. Jeżeli przed jego wywołaniem użytkownik nie zaznaczył żadnych krawędzi, lepiej wyświetlić o tym odpowiedni komunikat. W ten sposób będzie wiedział, co powinien następnym razem zrobić.

Takie "zaawansowane sprawdzanie" można byłoby teoretycznie dodać do metody *execute()*. Jednak, w pewnych sytuacjach, ta metoda może być wołana wielokrotnie, raz za razem, dla tego samego kontekstu i różnych parametrów wejściowych. (Przekonasz się o tym w następnej sekcji). Dlatego nie warto umieszczać w niej takich weryfikacji, a już na pewno — wyświetlać komunikatów dla użytkownika. Lepszym miejscem jest inna metoda, przewidziana w interfejsie **Operator**: **invoke()** (Rysunek 4.2.2):



Rysunek 4.2.2 Dodatkowe testy danych wejściowych - procedura invoke()

Blender oczekuje od *invoke()* tego samego, co od *execute()*: kodu informacji zwrotnej. Nasza implementacja zaczyna od policzenia zaznaczonych krawędzi siatki. Jeżeli nie ma żadnych, wyświetla komunikat i zwraca 'CANCELLED'. Jeżeli jednak jest co fazować — zwraca rezultat wywołania metody *execute()* ('FINISHED').

¹ Oprócz interfejsu **Operator**, API Blendera zawiera jeszcze dwa inne interfejsy (klasy abstrakcyjne): **Menu** i **Panel**. Służą oczywiście do implementacji GUI. Wszystkie trzy znajdziesz w opisie modułu **bpy.types**, a także w podpowiedziach uzupełniania kodu, które dodaliśmy do PyDev. Żałuję, że na razie opis tych interfejsów w dokumentacji Blendera jest bardzo ubogi. Wiele szczegółów, które tu opisuję, jest opartych na analizie przykładów i moich własnych obserwacjach!

86

Metoda *invoke()* oprócz kontekstu (*context*) otrzymuje także obiekt *event*. To informacja o "wydarzeniu" — przesunięciu myszki i stanie klawiatury. Można ją wykorzystać do budowy zaawansowanych operatorów (patrz przykłady w opisie klasy Operator). Do sprawdzenia zawartości obiektu *event* używaj w debuggerze okna *Expressions*. Wpisuj w nie konkretne nazwy pól *bpy.types.Event*, np. "event.type", albo "event.value". Próba rozwinięcia pól obiektu *event* w oknie *Variables* kończy się błędem i zakończeniem działania Blendera!

Chciałbym tylko Cię jeszcze uczulić na pewne zagadnienie: pętle w Pythonie. W kodzie, który pokazuje Rysunek 4.2.2, napisałem pętlę liczącą zaznaczone krawędzie w sposób jak najbardziej czytelny. (Nazwałbym go "*visual basicowym*"). Przeglądając kod na przykładach umieszczonych gdzieś w sieci, możesz się natknąć na inne, "jednoliniowe" rozwiązanie tego samego zagadnienia (Rysunek 4.2.3):



Rysunek 4.2.3 Alternatywny sposób liczenia zaznaczonych krawędzi w metodzie invoke()

Ten styl nazwałbym "*pythonowym*", a może nawet "*lispowym*". Funkcja *filter()* zwraca tzw. *iterator*, który funkcja *list()* przekształca w listę. Potem pozostaje sprawdzić, jaka jest długość tej listy. W funkcji *filter()* wykorzystałem bezimienną, chwilową *funkcję lambda*. Funkcja *lambda* otrzymuje od filtru jeden argument — element listy (*e*). Zwraca wartość swojego ostatniego (tutaj: jedynego) wyrażenia — czyli *True*, gdy krawędź *e* jest zaznaczona. (Opis szczegółów funkcji *filter()* znajdziesz w dokumentacji Pythona). Czytelność kodu zależy od zaawansowania czytającego. Dla doświadczonego programisty wyrażenie wykorzystujące funkcję *filter()* jest tak samo czytelne, jak pętla którą pokazuje Rysunek 4.2.2. Z drugiej strony tacy programiści wolą pisać bardziej "zwarty" kod, stąd warto się przyzwyczaić do takich wyrażeń.

No dobrze, to mamy ulepszony, gotowy do użycia operator. Ale jak go dodać do standardowego menu Blendera, takiego jak *Specials* (Rysunek 4.2.4)?



Rysunek 4.2.4 Menu Specials. To do niego dodamy polecenie Bevel.

Standardowe menu Blendera są tworzone w ten sam sposób, w jaki Twój dodatek dopisze swoje: za pomocą API. Trzeba tylko odkryć, jak się nazywa klasa implementująca menu *Specials*. Zacznijmy od znalezienia pliku, który powinien ją zawierać. Skrypty, które implementują całe GUI Blendera, znadziesz w katalogu *scripts\startup\bl_ui* (Rysunek 4.2.5):



Rysunek 4.2.5 Wyszukiwanie pliku z kodem do obsługi okna View 3D.

Pliki o nazwach zaczynających się od *properties_**, zawierają klasy różnych grup paneli okna *Properties.* Te w tej chwili pomijamy. Druga grupa plików ma nazwy postaci *space_<nazwa obszaru>.py*. To tam znajduje się kod odpowiedzialny za wygląd nagłówków i menu poszczególnych okien Blendera. Menu Specials występuje w obszarze (oknie) *View 3D*, więc plikiem, w którym powinno być zdefiniowane, jest *space_view3d.py*.

Otwórzmy ten plik w swoim ulubionym pomocniczym edytorze (to może być standardowy Notepad, albo popularny Notepad++). Zacznij w nim szukać nazwy menu — tekstu "**Specials**" (Rysunek 4.2.6):

📮 space_view3d.p	y - Notatnik 📃 🗖	×			
Plik Edycja Format	Widok Pomoc				
# ##### BEGIN G	FL LICENSE BLOCK #####	^			
# This program # modify it un # as published # of the Licen # This prog <mark>Zna</mark>	n is free software; you can redistribute it and/or oder the terms of the GNU General Public License d by the Free Software Foundation; either version 2 ose or (at your optic Szukaj jakiegoś charakterystycznego tekstu, który widziała ajdowanie na menu. W naszym przypadku to tekst <i>Specials</i> .	eś			
# but WITHO # MERCHANTA Zn # GNU Gener	najdź: Specials Znajdź <u>następny</u>	_			
# # You shoul # along wit # Inc., 51	Uwzględnij wielkość liter				
# ##### END GPL LICENSE BLOCK #####					
# <pep8 complia<br="">import bpy</pep8>	ant>	~			

Rysunek 4.2.6 Wyszukiwanie tekstu "Specials" w pliku space_view3d.py

88

Podczas wyszukiwania zachowaj jednak czujność — ten tekst może wystąpić w różnych menu! Tak jest i w tym przypadku. Napierw znalazłem klasę implementującą menu *Specials* dla trybu *Object Mode* (Rysunek 4.2.7):

📮 space_view3d.py	r - Notatnik	
Plik Edycja Format V	Znajdowanie 💦 🛛 💽	
class VIEW3D_MT_ bl_label = "	Znajdź: Specials Znajdź <u>n</u> astępny	
def draw(sel layout =	Uwzględnij wi <u>e</u> lkość liter OW górę ⊙W <u>d</u> ół	_
layout.c layout.c layout.c layout.c	operator("object.rotation_clear", text="Rotation") operator("object.scale_clear", text="Scale") operator("object.To jest menu <i>Specials</i> , ale z <i>Object Mode</i> !	
class VIEW3D_MT bl_label =	object <mark>specials</mark> (bpy.types.Menu): 'Specials"	
@classmethod def poll(cls # add mo	d s, context): pre special types	~
<		≥:

Rysunek 4.2.7 Jedno z niewłaściwych trafień: menu dla innego trybu (Object Mode)

Skąd wiedziałem, że to nie menu, którego szukam? Choć nazwę (wartość pola *bl_label*) miało odpowiednią, to jego pozycje (linie *layout.operator(<nazwa operatora>, text* = <*nazwa wyświetlana>*)) były inne!

Po znalezieniu drugiej i trzeciej definicji menu *Specials* zorientowałem się, że częścią ich nazwy jest symbol trybu Blendera, w którym są stosowane: "object", "particle"... Za trzecim razem znalazłem właściwe: "edit_mesh" (Rysunek 4.2.8):



Rysunek 4.2.8 Odnaleziona definicja menu Specials z Edit Mode.

Gdy znalazłem już nazwę klasy menu, mogłem dopisać do skryptu wtyczki kod dodający ją do menu *Specials* (Rysunek 4.2.9):



Rysunek 4.2.9 Obsługa rejestracji operatora w menu Specials

Pierwsze testy tych rozbudowanych metod register() i uregister() wypadły pomyślnie (Rysunek 4.2.10):





Także test "zerowy" — wywołanie bez zaznaczonych krawędzi — dało oczekiwany wynik (Rysunek 4.2.11):



Rysunek 4.2.11 Rezultat wywołania polecenia gdy siatka nie ma zaznaczonych krawędzi

90

Jednak gdy zaznaczyłem niektóre krawędzie siatki i ponownie wywołałem W, *Bevel* — zamiast faz zobaczyłem znowu ten sam komunikat (Rysunek 4.2.12):



Rysunek 4.2.12 Rezultat wywołania polecenia gdy siatka ma zaznaczone krawędzie

Wygląda na to, że tak się rozgadałem na str. 84—86, że popełniłem głupi błąd. Zapomniałem o tym, o czym sam pisałem w poprzednim rozdziale (por. str. 52, 53). Przed rozpoczęciem liczenia krawędzi należy się przełączyć w *Object Mode*, a po policzeniu — powrócić do *Edit Mode* (Rysunek 4.2.13):



Rysunek 4.2.13 Poprawka w programie: każde odwołanie do danych siatki trzeba wykonywać w Object Mode!

Teraz procedura *invoke()* znajduje zaznaczone krawędzie, i całość działa poprawnie (Rysunek 4.2.14):



Rysunek 4.2.14 Działanie obecnej wersji wtyczki

Osiągnęliśmy już efekt zbliżony do polecenia *Bevel* z Blendera 2.49 (por. str. 34). Barkuje nam tylko jeszcze interaktywnej zmiany szerokości fazy. Tę funkcjonalność dodamy w następnej sekcji.

Podsumowanie

- Operator można "wyposażyć" w opcjonalną metodę *poll()*. Ta funkcja jest używana przez Blender do sprawdzania, czy w aktualnym kontekście operator w ogóle ma być dostępny. (Na przykład —. widoczny w menu). Możesz w niej sprawdzać aktualny tryb pracy programu (str. 84);
- Dalszą, bardziej szczegółową weryfikację danych wejściowych najlepiej jest umieścić w innej metodzie: *invoke()* (str. 85). GUI Blendera zazwyczaj wywołuje tę metodę operatora gdy tylko jest dostępna. Sterują tym odpowiednie właściwości obiektu *bpy.types.Menu* (por. str. 89). To samo dotyczy (tego tu nie pokaza-łem) paneli (klas pochodnych *bpy.types.Panel*);
- Polecenie dodajemy do menu Blendera w metodzie *register()*, a usuwamy w *unregister()* (str. 89). Aby napisać ten fragment kodu, musimy znać nazwę klasy Pythona, implementującej menu;
- Aby odnaleźć nazwę klasy standardowego menu, trzeba zajrzeć do skryptów, implementujących GUI Blendera (str. 87 - 88);

4.3 Implementacja interakcji z użytkownikiem

W Blenderze 2.5 implementacja interakcji operatora z użytkownikiem stała się bardzo prosta. Przynajmniej jeżeli chodzi o realizację pewnego podstawowego schematu. Pozwala on użytkownikowi dynamicznie zmieniać (np. ruchem myszki) parametry naszego polecenia i obserwować na bieżąco rezultat na ekranie.

Dodaj najpierw do klasy wtyczki pole *width*, utworzone za pomocą funkcji z modułu *bpy.props* (Rysunek 4.3.1):

Imports
import bpy
from bpy props import FloatProperty
niującej atrybut (kontrolkę) typu Float
Pozoslaty kou skryplu
Operator
class Bevel (bpy.types.Operator):
''' Bevels selected edges of the mesh''' Twisted Torus, czy coś podobnego
bl_idname = "mesh.bevel"
bl_label - Bevel bl description = "Bevels selected edges"
bl_options = { 'REGISTER', 'UNDO'} #Set this options, if you want to update
parameters of this operator interactively
(in the Tools pane)
width = FloatProperty(name="Width", description="Bevel width",
subtype = 'DISTANCE', default = 0.1, min = 0.0,
step = 1, precision = 2)
Blender interface methods
Blender interface methods @classmethod width
Blender interface methods @classmethod def poll (cls,context):
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH')</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event):</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected?</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle()</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bry.ops.object.data.edges()</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle()</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0:</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context)</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context) else: selef_report(type='EREOR', message="No edges_selected")</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context) else: self.report(type='ERROR', message="No edges selected") return {'CANCELLED'}</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? byy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) byy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context) else: self.report(type='ERROR', message="No edges selected") return {'CANCELLED'}</pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context) else: self.report(type='ERROR', message="No edges selected") return {'CANCELLED'} def execute(self, context): Wykorzystanie wartości berel/certert_skiett/certert_adata.edge </pre>
<pre># Blender interface methods @classmethod def poll(cls,context): return (context.mode == 'EDIT_MESH') def invoke(self, context, event): #input validation: are there any edges selected? bpy.ops.object.editmode_toggle() selected = list(filter(lambda e: e.select, context.object.data.edges)) bpy.ops.object.editmode_toggle() if len(selected) > 0: return self.execute(context) else: self.report(type='ERROR', message="No edges selected") return {'CANCELLED'} def execute(self,context): bevel(context.object,self.width) return ('FINISHED') </pre>

Rysunek 4.3.1 Zmiany w definicji klasy

Pole, utworzone w ten sposób, Blender będzie wyświetlał jako kontrolkę na ekranie. Moduł *bpy.props* zawiera klasy odpowiadające wszystkim elementarnym typom: *Bool*, Float*, Int*, String**. Dodatkowo znajdziesz w nim także jednowymiarowe tablice (klasy **Vector**) każdego z tych typów. W naszym skrypcie szerokość fazy jest liczbą zmiennoprzecinkową (*float*). Dlatego z *bpy.props* importuję tylko jedną klasę — *FloatProperty*. W jej konstruktorze ustala się wszystko, co jest potrzebne kontrolce: etykietę (*name*), opis (*description*) do "dymka", wartość domyślną, dopuszczalny zakres. W parametrze *step* ustalasz krok, o który będzie się zmieniać kontrol-ka podczas klikania strzałek. Parametr *precision* dotyczy sposobu wyświetlania. To liczba cyfr po kropce dzie-siętnej.

Po dodaniu do klasy parametrów (właściwości), koniecznie musisz jeszcze dodać do niej pole o nazwie *bl_options*. (Dotychczas go nie używaliśmy, występuje w klasie bazowej — *bpy.types.Operator*). Musisz mu przypisać opcje {'**REGISTER**', '**UNDO**'} (Rysunek 4.3.1). Wpisz dokładnie taką kombinację. Jeżeli przypiszesz samą wartość '**REGISTER**' albo samo '**UNDO**', nie uzyskasz efektu, który pokazuje Rysunek 4.3.2:



Rysunek 4.3.2 Interaktywna zmiana szerokości fazowania

Wywołaj teraz naszą wtyczkę (*Specials → Bevel*). Zaznaczone krawędzie sześcianu zostały sfazowane, tak jak poprzednio. Naciśnij jednak klawisz , by otworzyło się okno przybornika (po lewej stronie ekranu). W obszarze *Tool properties* pojawi się panel o takiej samej nazwie, jak operator (*Bevel*). Takie panele zawierają kontrolki z prametrami polecenia — w naszym przypadku to szerokość fazy *Width*. Gdy wpiszesz tu nową wartość, zmienisz natychmiast szerokość fazowanych krawędzi. A jeżeli przeciągniesz po niej myszką (wciśnięty) — szerokość faz będzie się zmieniać dynamicznie, podobnie jak w Blenderze 2.49 (por. str. 34, Rysunek 3.1.2).

Jak Blender uzyskuje ten efekt? Do wyśledzenia takich interaktywnych zdarzeń czasami najlepiej się nadaje nie debugowanie, a prosty wydruk jakiegoś tekstu w konsoli programu. Umieśćmy na chwilę odpowiednie polecenia *print()* w obydwu procedurach operatora: *invoke()* i *execute()* (Rysunek 4.3.3):



Rysunek 4.3.3 Dodanie do kodu komunikatów diagnostycznych (tylko na chwilę)

Załaduj tę nową wersję wtyczki, i jeszcze raz wywołaj Specials →Bevel (Rysunek 4.3.4):



Rysunek 4.3.4 Stan bezpośrednio po wywołaniu polecenia Specials →Bevel

Bezpośrednio po tym wywołaniu w konsoli Blendera pojawiły się dwie linie (Rysunek 4.3.4). Wygląda na to, że została tu wywołana metoda *invoke()*, która z kolei (por. Rysunek 4.3.1) wywołała *execute()*, z domyślną wartością parametru *width*.

Teraz zacznij zmieniać wartość kontrolki *Bevel:Width* w obszarze *Tool Properties*. Zdecydowałem się nacisnąć dziesięć razy "strzałkę" z prawej strony pola, zwiększając szerokość fazy od 0.1 do 0.2 (Rysunek 4.3.5):



1	ī	work correc	ctly, b	ut	; a ĥi	t slower)
li	n	invoke()				
li	n	execute(),	width	=	0.10	
i	n	execute(),	width	=	0.11	
i	n	execute(),	width	=	0.12	
i	n	execute(),	width	=	0.13	
li	n	execute(),	width	=	0.14	
li	n	execute(),	width	=	0.15	
li	n	execute(),	width	=	0.16	
li	n	execute(),	width	=	0.17	
li	n	execute(),	width	=	0.18	
li	n	execute(),	width	=	0.19	
i	n	execute(),	width	=	0.20	

Rysunek 4.3.5 Stan bezpośrednio po interaktywnym poszerzeniu szerokości faz do 0.2

Widzisz? Wygląda na to, że po każdym moim kliknięciu Blender wykonywał *Undo*, a następnie po prostu ponownie wołał operator. Tyle, że tym razem wywoływał bezpośrednio jego metodę *execute()*, dla odpowiedniej wartości parametru *width*, odczytanej z kontrolki.

Przypuszczam, że z punktu widzenia Blendera wywoływany był operator: **bpy.ops.mesh.bevel(width = <wartość z kontrolki>)**. Po dodaniu właściwości **width** do klasy **Bevel**, w funkcji odpowiadającej temu operatorowi pojawił się argument o takiej samej nazwie (Rysunek 4.3.6):



Rysunek 4.3.6 Blender traktuje pole klasy jak nazwany argument metody bpy.ops.mesh.bevel()

Myślę, że podział ról pomiędzy procedurami invoke() i execute() można przedstawić następująco:

 Procedura *invoke()* jest wołana, gdy operator ma zostać wykonany z domyślnymi wartościami swoich parametrów. Procedura *execute()* jest wykonywana dla konkretnych wartości parametrów (podanych jawnie w liście argumentów wywołania).

Wyborem wywoływanej przez GUI metody można sterować np. za pomocą pewnych flag (por. str. 89).

Na koniec został do zaimplementowania ostatni szczegół: nasze polecenie ma zapamiętywać ostatnio użytą szerokość fazowania. Powinno ją podstawiać jako wartość domyślną przy kolejnym wywołaniu. To bardzo ułatwi użytkownikowi pracę.

Jak to zaimplementować? Od razu zapomnij o przechowywaniu czegokolwiek w klasie **Bevel**. Wygląda na to, że Blender za każdym razem tworzy jej nową instancję. Dlatego obsługę każdego wywołania klasa operatora rozpoczyna w swoim stanie "domyślnym". Właściwym miejscem do przechowywania informacji pomiedzy wywołaniami są dane Blendera. Większość elementów sceny można używać tak, jak gdyby były słownikiem (*dictionary*) Pythona (Rysunek 4.3.7):

```
context.object["our data"] = 1.0 #Store
stored_data = context.object["our data"] #Retrieve
```

Rysunek 4.3.7 Najprostszy przykład przechowywania danych skryptu w obiekcie Blendera

Z odczytywaniem wartości należy jednak trochę uważać. Zazwyczaj nie możesz być pewien, czy Twoje dane zostały wcześniej tam umieszczone. Dlatego lepiej do ich odczytu użyć standardowej metody *get()*, która nie generuje wyjątku gdy żądanego elementu brakuje w słowniku. A oto modyfikacja, który implementuje zapamiętywanie ostatniej szerokości fazownia (Rysunek 4.3.8):



Rysunek 4.3.8 Implementacja zapamiętywania ostatniej użytej szerokości

Zwróć uwagę, że do przechowywania ostatniej szerokości fazowania zdecydowałem się wykorzystać bieżącą scenę (*context.scene*), a nie konkretny obiekt (Rysunek 4.3.8). Gdybym szerokość fazy zapisywał w bieżącym obiekcie, wówczas przy edycji każdej siatki mogłaby podpowiadać się coraz to inna wartość. Myślę, że byłoby to bardzo mylące dla użytkownika. Dlatego wolę zapamiętywać jedną szerokość dla wszystkich wywołań — a takie rzeczy najlepiej jest przechowywać w bieżącej scenie. Zapisywanie parametrów w danych Blendera ma także tę zaletę, że są one zapamiętywane podczas zapisu pliku na dysk.

Istnieje jeszcze inny problemem z takimi danymi, który może narosnąć wraz z upływem czasu. Ten sam klucz słownika (nazwa danej) może być zapisany w tej samej scenie przez dwie różne wtyczki. W efekcie jedna "nadpisze" parametry drugiej, i skończy się to najprawdopodobniej błędem pierwszego z tych skryptów. Dlatego należy używać jako kluczy jak najbardziej specyficznych (i długich!) nazw.

Implementacja przechowywania ostatniej użytej wartości fazy była "końcową kosmetyką". Akurat demonstracja tej funkcjonalności na obrazkach jest niemożliwa, więc ich tu już nie umieszczam (③). Nasza wtyczka *mesh_bevel.py* jest już gotowa. Gdy umieścisz ten plik wśród innych wtyczek, w folderze Blendera *scripts\addons*, wtyczka stanie się widoczna w oknie konfiguracji (Rysunek 4.3.9):

Interface	Editing	Input	Add-Ons	Themes	File	System
All Enabled Disabled 3D View Add Curve	Mesh: Description: Location: Author: Version:	Bevel Bevels se View3D Witold Ja 1.0.0	elected edges >Specials (W-ke worski	₂y)		۳ چ
Add Mesh Animation	Mesh:	Bsurfaces G	PL Edition	ologics with Cross	sa Pancil	≍ _
Development Game Engine	Location:	View3D	> EditMode > To	bolShelf	se Pencii	
Import-Export Mesh	Author: Version:	Eclectiel		Takie przyc ku bl_info adresy "wił	iski pojawią się skryptu wtyc ki_url" i "tracke	, gdy w słowni- zki uzupełnisz r_url"
Object Render	Warning:	U Beta	to the Wiki	🕼 Report a B	Bug	

Rysunek 4.3.9 Gotowa wtyczka Bevel w oknie Add-Ons.

Pozostało tylko jeszcze założyć na *blenderwiki.org* stronę z opisem tego dodatku oraz jego *bug tracker*, na ewentualne zgłoszenia od użytkowników¹. Nie jest to jednak tematem tej książki. Pełną wersja skryptu, który tu opracowaliśmy, znajdziesz na str. 131.

¹ Właśnie w wyniku takich zgłoszeń dodałem później do tego skryptu dalsze modyfikacje. Jedną z nich jest dynamiczna adaptacja szerokości fazowania. Chodziło o to, by proponowana domyślnie wartość pasowała także do bardzo dużych lub bardzo małych obiektów. W tym celu dodałem do procedury *invoke()* kolejne linie, "oceniające" rozmiar obiektu. Na tej podstawie program podejmujące decyzję, czy nie trzeba zignorować ostatnio używanej szerokości fazowania. Jeżeli tak — to proponuje użytkownikowi nową szerokość fazy, dopasowaną do rozmiaru aktualnie edytowanej siatki. Takie "dopiski" to naturalny rozwój każdego programu. Sądzę jednak, że opis implementacji takiej dodatkowej funkcjonalności niepotrzebnie skomplikowałyby nasz przykład, stanowiący podstawę tej książki. Jeżeli chcesz przeanalizować pełen kod obecnej wersji wtyczki *mesh_bevel.py*, możesz ją pobrać ze strony: <u>http://airplanes3d.net/scripts-253_p.xml</u>.

Podsumowanie

- Parametry polecenia (*properties*) tworzymy w klasie operatora za pomocą funkcji z modułu *bpy.props* (str. 92). Pola, utworzone w ten sposób, stają się automatycznie nazwanymi parametrami procedury, za pomocą której Blender wywołuje operator (str. 94);
- Jeżeli do klasy dodamy pole bl_options = {'REGISTER','UNDO'}, wówczas nasze polecenie stanie się "interaktywne". Gdy je wywołasz, w oknie przybornika (*Tool Properties*) pojawią się kontrolki, odpowiadające parametrom (*properties*) operatora. Można je tutaj dynamicznie zmieniać, np. za pomocą myszki. Każdej zmianie towarzyszy odpowiednia aktualizacji rezultatu działania operatora. Możesz ją na bieżąco śledzić na ekranie (str. 94);
- Wartości parametrów operatora, użyte w ostatnim poleceniu, warto zapamiętać w aktualnej scenie. Można ich użyć jako wartości domyślnych przy kolejnym wywołaniu (str. 95).

Dodatki

W tej części umieściłem różne opcjonalne materiały pomocnicze. Mogą Ci się przydać, gdy czegoś nie jesteś pewien w trakcie czytania tekstu głównego.

Rozdział 5. Szczegóły instalacji

W tym rozdziale umieściłem opis szczegółów instalacji Pythona, Eclipse i PyDev. Zrobiłem to na wszelki wypadek, gdybyś "utknął" na jakimś drobiazgu.

5.1 Szczegóły instalacji Pythona

Przebieg instalacji interpretera Pythona nie zmienia się już od lat, więc pozwolicie, że pokażę go Wam na przykładzie wersji 2.5.2, dla której mam zrobione ilustracje.

Zaczynamy od wejścia na portal tego projektu: <u>www.python.org</u> (Rysunek 5.1.1):



Rysunek 5.1.1 Strona główna projektu Pythona

Przejdź na niej do sekcji DOWNLOAD (Rysunek 5.1.2):

💽 🗢 🥐 http://www.python.org/do	wnload/ 🛛 💌 🐼 😽 🗙 🚼 Google	2
Jlubione 🌏 Pownload Python	🚵 🔻 🔂 👻 🖃 🖶 🛪 Strona 👻 Bezpieczeństwo 👻 Narzędzia 👻	•
Source	Download Python	
	The current production versions are Python 2.7.1 and Python 3.2.	
FOUNDATION >>	Start with one of these versions for learning Python or if you want the most stability; they're both considered stable production releases.	
Python Wiki Python Insider Blog	If you don't know which version to use, start with Python 2.7; more existing third party software is compatible with Python 2 than Python 3 right now.	
Python 2 or 3? Help Maintain Website	For the MD5 checksums and OpenPGP signatures, look at the detailed Python 2.7.1 page:	
	 Python 2.7.1 Windows Installer (Windows binary does not include source) Python 2.7.1 Windows X86-64 Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] does not include source) 	
Non-English Resources	 Python 2.7.1 Mac OS X 32-bit i386/PPC Installer (for Med OS X 10.2 through 10.6 [2]) Python 2.7.1 Mac OS X 64-bit/32-bit x86-64/138 Python 2.7.1 compressed source tarball (for Lift) 	
Release Schedule	Python 2.7.1 bzipped source tarbal/(for Linux, Unix or Mac OS X, more compressed) Also look at the detailed Python 3.2 page:	
Python 2.6.7rc1	Python 3.2 Windows x86 MSI Installer (Windows binary does not include source)	
niedziela, 15 maj Python 3.2.1rc1	Python 3.2 Windows X86-64 MSI Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] does not include source)	

Wybierz z niej taką wersję Pythona, której używa Twój Blender. (Jeżeli nie możesz znaleźć identycznej – wybierz jak najbardziej zbliżoną).

Kliknij w wybrany link, i wybierz opcję Uruchom (Rysunek 5.1.3):

Pobieranie pliku - ostrzeżenie o zabezpieczenia	ch 🔀
Czy chcesz uruchomić plik, czy zapisać go?	
Nazwa: python-2.5.2.msi Ukończono: Typ: Pakiet Instalatora W Od: www.python.org	3% z python-2.5.2.msi
Uruchom Otwieranie: python-2.5.2. Pliki pochodzące z Internetu mog może być potencjalnie szkotliwy zaufania do źródła, nie uruchami oprogramowania. Jakie jest zatro	Internet Explorer - ostrzeżenie o zabezpieczeniach Image: Composition of the system of the syste
	Ten plik nie ma prawidłowego podpisu cyfrowego weryfikującego jego wydawcę. Należy wyłącznie uruchamiać oprogramowanie pochodzące od zaufanych wydawców. Jak zdecydować, czy oprogramowanie można uruchomić?

Rysunek 5.1.3 Pobieranie programu instalacyjnego z portalu Pythona

(Oczywiście, jeżeli wolisz, możesz zamiast bezpośredniego uruchamiania wcześniej zapisać ten plik na dysku).

Upewnij się, że masz pełne uprawnienia do tego komputera (administracyjne), i uruchom program instalacyjny. Przejdź przez kolejne ekrany (Rysunek 5.1.4 — możesz na nich niczego nie zmieniać):

🔂 Python 2.5.2 Setup		🕼 Python 2.5.2 Setup
	Select whether to install Python 2.5.2 for all users of this computer.	Select Destination Directory
		Please select a directory for the Python 2.5.2 files.
	 Install for all users 	Python25 Up New
python	O Install just for me	python Filetter
windows		windows
	Back Next > Cancel	< Back Next > Cancel
🛱 Python 2.5.2 Setup	X	S 1₽ Python 2.5.2 Setup
🛃 Python 2.5.2 Setup	Customize Python 2.5.2	Ill Python 2.5.2 Setup Install Python 2.5.2
Python 2:5.2 Setup	Customize Python 2.5.2 Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed. Register Extensions Trd/Tk Documentation Utility Scripts Test suite	Python 2.5.2 Setup Install Python 2.5.2 Please wait while the Installer installs Python 2.5.2. This may take several minutes. Status:
Python 2.5.2 Setup Python python windows	Customize Python 2.5.2 Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.	Install Python 2.5.2 Setup Install Python 2.5.2 Please wait while the Installer installs Python 2.5.2. This may take several minutes. Status:

Rysunek 5.1.4 Kolejne ekrany instalacji

Na zakończenie program wyświetli taki ekran (Rysunek 5.1.5):



Rysunek 5.1.5 Ostatni ekran instalacji Pythona

Naciskasz Finish i to kończy całą instalację.

5.2 Szczegóły instalacji Eclipse i PyDev

Na początku sprawdź, czy masz na swoim komputerze dostępny Java Runtime Environment (Java JRE).
 W systemie Windows powinieneś w panelu sterowania mieć ikonę "Java". Jeżeli jej tam nie ma — pobierz ją z java.com i zainstaluj¹.

Zacznijmy od pobrania Eclipse. Wejdź na stronę http://www.eclipse.org/downloads (Rysunek 5.2.1):

🖉 Eclipse Downloads - Program Windows Internet Explorer dostarczony przez TADMAR	
C C C T T T T T T T T T T T T T T T T T	🔽 🛃 🛃 🔀 🔽
🖕 Ulubione 🖉 Eclipse Downloads	🐴 🔹 题 👻 🖃 🖶 🔹 St
Indigo is Coming! Join the Indigo 500	Visit other Eclipse Sites
Home Downloads Users Members Committers Resources Projects Abou	It Us Google" Cu
Eclipse Downloads	
Packages Developer Builds Projects	
Compare Packages Older Versions Eclipse Helios (3.6.2) Packag	ges for Windows 💌
Composed 1,685,306 Times Details	Windows 32 Bit Windows 64 Bit
Eclipse IDE for Java EE Developers, 206 MB Downloaded 1,247,215 Times Details	Windows 32 Bit Windows 64 Bit
Eclipse Classic 3.6.2, 171 MB Downloaded 822,124 Times Details Other Downloads	Windows 32 Bit Windows 64 Bit
Image: Second system Promoted Download Redeploying sucks. Try JRebel: Code without interruption. Save 1 hour a day.	Download
G+ Eclipse IDE for C/C++ Developers, 87 MB Downloaded 408,951 Times Details	Windows 32 Bit Windows 64 Bit
Bownloaded 234,703 Times Details Wybierz taką najmniejszy ro	odmianę, która ma ozmiar!

Rysunek 5.2.1 Wybór odmiany Eclipse

W istocie, Eclipse to coś w rodzaju "ramowego" środowiska programisty. Te "ramy" można przystosować do pracy z konkretnym językiem/językami programowania za pomocą odpowiednich dodatków. Na portalu Eclipse są udostępniane pewne typowe, najczęściej pobierane konfiguracje. Nie ma wśród nich gotowej wersji dla Pythona, więc złożymy ją sobie sami. Przy wyborze odmiany kieruj się rozmiarem pliku — powinien być jak najmniejszy. Od czerwca 2011r, w wersji 3.7 ("Indigo") pojawiła się "prawie pusty" zestaw *Eclipse for Testers* (87 MB). Pisząc ten podręcznik korzystałem z poprzedniej wersji 3.6 ("Helios"), gdzie najmniejszym "pakietem" był *Eclipse IDE for C/C++ Developers* (także 87 MB). Na *Eclipse for Testers* PyDev na będzie się instalował trochę dłużej, ale potem *eclipse.exe* szybciej będzie uruchamiać całe środowisko.

W każdym razie linki z tej strony prowadzą do plików *.*zip*. Pobrany plik zapisz gdzieś na dysku. (Podczas pisania tego podręcznika pobrałem plik o nazwie *eclipse-cpp-helios-SR2-win32.zip*).

103

¹ Uwaga: w niektórych odmianach Linuxa, jak Ubuntu, domyślną maszyną wirtualną (VM) Javy jest GCJ. Eclipse działa na niej wolniej niż na JVM z <u>www.java.com</u>. Co więcej, nawet po pobraniu i wgraniu do Ubuntu tej nowej Javy, nie jest ona "maszyną" domyślną! To trzeba poprawiać "ręcznie". Szczegółowe instrukcje dot. instalacji Javy i Eclipse na Ubuntu — patrz <u>https://help.ubuntu.com/community/EclipseIDE</u>.



W pobranym pliku znajduje się folder *eclipse*, z gotowym do uruchomienia programem (Rysunek 5.2.2):

Rysunek 5.2.2 Rozpakowanie folderu z programem

Wystarczy go rozpakować do folderu *Program Files*. (Tak! Tu nie ma żadnego instalatora, który robi coś, czego nie wiemy! Eclipse nie ma żadnych zewnętrznych zależności poza JRE, nie wgrywa niczego do rejestru. Możesz dzięki temu np. mieć wgranych kilka alternatywnych wersji tego programu jednocześnie, i nie będzie pomiędzy nimi żadnych konfliktów).

Eclipse uruchamia program eclipse.exe (Rysunek 5.2.3):

🖙 C:\Program Files\eclipse		
Plik Edycja Widok Ulubione Narzędzia Pomoc		A 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997 - 1997
Adres 🗁 C:\Program Files\eclipse		💌 🄁 Przejdź
Foldery ×	Nazwa 🔺	Rozmiar Typ
🗉 🛅 Common Files 🛛 🗛	Configuration	File Folde
ComPlus Applications	🗀 dropins	File Folde
	Features	File Folde
🗉 🧰 configuration	☐ p2	File Folde
👝 dropins	🛅 plugins	File Folde
E 🛅 features	🚞 readme	File Folde
■ P ² uruchamia Eclipse	🔤 .eclipseproduct	1 KB Plik ECLIF
	artifacts.×ml	63 KB Dokumen
	eclipse.exe	52 KB Applicatio
🗉 🧰 eclipse.cpp	eclipse.inik	1 KB Configura
🗉 🧰 eclipse.org	eclipsec.exe	24 KB Applicatio
🗉 🧰 FileZilla	epl-v10.html	17 KB HTML Do
🗉 🧰 GIMP 📲	Enotice.html	9 KB HTML Do



Możesz wstawić skrót do tego pliku w swoje ulubione menu, lub umieścić na pulpicie.

105

Przy uruchomieniu najpierw pojawi się okno dialogowe, w którym należy wskazać (wystarczy potwierdzić) położenie foldera z projektami (Rysunek 5.2.4):

🖨 Workspace Launcher	
Select a workspace	
Eclipse stores your projects in a folder called a workspace. Choose a workspace folder to use for this session.	Indywidualny profil użytkownika. (Uważaj domyślnie to nie są <i>Moje Dokumenty</i> !)
Workspace: C:\Documents and Settings\w4979721\workspace	Browse
Use this as the default and do not ask again	
Gdy nie chcesz tej lokalizacji później zmieniać, tu możesz wyłączyć to pytanie	OK Cancel

Rysunek 5.2.4 Pytanie o folder dla przyszłych projektów

Każdy projekt to oddzielny folder, zawierający parę własnych plików Eclipse, oraz pliki z Twoim kodem. (Jeżeli Twój skrypt ma być w innym miejscu na dysku, będziesz mógł tu umieścić w folderze projektu tylko jego skrót). Zwróć uwagę, że domyślnie folder *workspace* jest umieszczony w katalogu głównym profilu użytkownika. (W tym przykładzie to użytkownik *W4979721*). To wcale nie są *Moje Dokumenty* — tylko poziom wyżej. To proste przełożenie konwencji z *Unix/Linux*. Jeżeli jesteś przyzwyczajony, że wszystkie swoje dane trzymasz w katalogu *Moje Dokumenty* — zmień ścieżkę, wyświetloną w tym oknie. Eclipse utworzy odpowiedni folder na dysku.

Po wskazaniu tego miejsca, Eclipse stara się zawsze wyszukać w nim i otworzyć ostatni używany projekt. Za pierwszym razem to niemożliwe, bo nowo utworzony folder jest pusty. Dlatego pojawia się odpowiednie okno dialogowe z ostrzeżeniem (Rysunek 5.2.5):



Rysunek 5.2.5 Ostrzeżenie (zawsze się pojawi przy pierwszym uruchomieniu)

Oczywiście, w tym przypadku nie ma się nim co przejmować.

Gdy nie można pokazać ostatniego projektu, Eclipse wyświetla stronę z linkami do miejsc w Internecie (Rysunek 5.2.6):



Rysunek 5.2.6 Okno Eclipse przy pierwszym uruchomieniu

Do zainstalowania PyDev najlepiej jest posłużyć się samym Eclipse. Wywołaj polecenie *Help→Install New Software* (Rysunek 5.2.7).

(Położenie takiego polecenia w menu *Help* może trochę dziwić stałych użytkowników Windows. Już prędzej spodziewałbym się go w *Edit* lub *File*. Ale do wszystkiego można się przyzwyczaić)



Rysunek 5.2.7 Instalacja dodatków do środowiska Eclipse

W oknie, które się pojawi, wpisz następujący adres: http://pydev.org/updates (Rysunek 5.2.8):

🖨 Install		
Available Software Select a site or enter the location of a site.	Tu wpisz adres projektu PyDev	
Work with: Http://pydev.org/updates	nore software by working with the <u>"Available Software Sites"</u> i naciśnij ten przycisk!	Add
Name	Version	
Tu możesz nadać temu źródłu bardziej elegancką nazwę, zamiast url.	Name: tocation: http://pydev.org/updates	Local Archive

Rysunek 5.2.8 Dodawanie do listy dostawców oprogramowania strony projektu PyDev

Następnie naciśnij przycisk *Add*. Spowoduje to otwarcie okna *Add Repository*. Gdy je zatwierdzisz, Eclipse odczyta komponenty, wyeksponowane pod podanym adresem (Rysunek 5.2.9):

🖶 Install	
Available Software Check the items that you wish to install.	
Work with: http://pydev.org/updates Add Find more software by working with the " <u>Available Software Sites</u> " preferences.	
Type ritter text Name Version Image: Constraint of the state o	
Select All Deselect All ■ ■ ■	Version
Show only the latest versions of available software Group items by category Contact all update sites during install to find required software PyDev i naciśnij Next	

Rysunek 5.2.9 Wybór dodatku Eclipse ze strony dostawcy oprogramowania

Zaznacz na tej liście komponent PyDev i naciśnij przycisk Next.

Eclipse wyświetli jeszcze dodatkową listę, zawierającą szczegóły instalowanych komponentów (Rysunek 5.2.10):

🖨 Install				3	
Install Details Review the items to be i	nstalled.				
Name		Version	Id		
PyDev for Eclipse	e 	2.1.0.2011052613	org.python.pydev.feature.feature.g.		
	Y				
	Install Details				Tutai widać no
	Review the items to be	e installed.			numer aktualnej wersji
	Name			Version	
Size: Upknown	🖗 PyDev for Eclip	se		2.1.0.2011052613	
	U		×		1
?		<u>Back</u>	Einish Cancel]	

Rysunek 5.2.10 Potwierdzenie szczegółów komponentów do zainstalowania

Po naciśnięciu kolejnego *Next* wyświetli się umowa licencyjna, do zaakceptowania (Rysunek 5.2.11):

🗧 Install
Review Licenses Licenses must be reviewed and accepted before the software can be installed.
License text (for PyDev for Eclipse 2.1.0.2011052613):
Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.
1. DEFINITIONS
"Contribution" means:
 a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and b) in the case of each subsequent Contributor:
i)changes to the Program, and
ii)additions to the Program;
where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.
"Contributor" means any person or entity that distributes the Program.
"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale
 I accept the terms of the license agreement ○ I do not accept the terms of the license agreement
Cancel

Rysunek 5.2.11 Potwierdzenie umowy licencyjnej

A po naciśnięciu przycisku Finish rozpocznie się instalacja.
109

Podczas instalacji Eclipse pobiera z Internetu wskazane komponenty. Postęp procesu pokazuje standardowe okno dialogowe (Rysunek 5.2.12):

🖨 Installing Software	
Installing Software	
Downloading org.eclipse.jdt.de	bug
Always run in background	ß
	Run in Background Cancel Details >>

Rysunek 5.2.12 Postęp instalacji

Po zakończeniu pobierania, Eclipse poprosi w kolejnym oknie o potwierdzenie certyfikatu (Rysunek 5.2.13):

Selection Needed	X	
Do you trust these certificates?		
Aptana Pydev; Pydev; Aptana		
	W przyszłości się pojawić <i>Aj</i> pił <i>Aptanę</i> na wątpliwości sp	zamiast firmy <i>Aptana</i> może tu ppcelarator. (<i>Appcelarator</i> zaku- początku 2011 roku). W razie prawdź w Wikipedii hasło PyDev.
Select All	Deselect All	
Aptana Pydev; Pydev; Aptana Aptana Pydev; Pydev; Aptana		
Details		
(?) ОК	Cancel	

Rysunek 5.2.13 Potwierdzenie certyfikatu

Po potwierdzeniu certyfikatu pojawi się ostatnie okno, kończące instalację dodatku (Rysunek 5.2.14):

🖶 Software Updates 🛛 🔀
You will need to restart Eclipse for the installation changes to take effect. You may try to apply the changes without restarting, but this may cause errors.
Restart Now Not Now Apply Changes Now



Uważam, że zawsze warto się zgodzić na domyślnie proponowany restart Eclipse.

5.3 Konfiguracja PyDev

Po zainstalowaniu PyDev należy skonfigurować domyślny interpreter Pythona. To ustawienie jest zapisywane w aktualnej przestrzeni roboczej (*workspace* — por. Rysunek 1.2.4). Aby je zmienić, wywołaj polecenie *Window → Preferences* (Rysunek 5.3.1):



Rysunek 5.3.1 Przejście do parametrów przestrzeni roboczej (workspce)

W oknie Preferences rozwiń sekcję PyDev i podświetl pozycję Interpreter - Python (Rysunek 5.3.2):

Preferences	
type filter text	Python Interpreters $(\neg \neg \neg$
⊞- General	Python interpreters (e.g.: python.exe)
■ C/C++	
	Name Location New
i Install/Update	element
Hin Java	
Builders	Remove
- Interactive Console	Down
- Interpreter - Iron Python	
Interpreter - Jython 🖌	🛋 Libraries Forced Builtins Predefined 🚾 Environment 🜻 String Substitution Variables
Interpreter - Python	Curkey DVTLIONDATI
Logging	System PT HONPATH
PyLint	New Folder
PyUnit	
Scripting PyDev	New Egg/Zip(s)
im Task Tags	
En Tacks	Remove
I I I I I I I I I I I I I I I I I I I	

Rysunek 5.3.2 Wywołanie automatycznej konfiguracji Pythona

Następnie naciśnij przycisk *Auto Config*. Jeżeli ścieżka do folderu Pythona została dodana do zmiennej środowiskowej (Windows) *Path*, to powinno zadziałać poprawnie. (Eclipse otworzy wówczas takie okno, jakie pokazuje Rysunek 5.3.5). Jeżeli jednak program nie mógł znaleźć Pythona — wyświetli taki komunikat (Rysunek 5.3.3):



Rysunek 5.3.3 Komunikat o braku możliwość automatycznej konfiguracji interpretera

Naciśnij wówczas w oknie *Properties* przycisk *New* (por. Rysunek 5.3.2). Spowoduje to otwarcie okna "ręcznego" wyboru interpretera (Rysunek 5.3.4):

🖨 Select interpreter 🛛 🗙				
Enter the name and executable of your interpreter		Tu wpisz nazwę, pod jaką ma się pojawiać na listach PyDev		
Interpreter Name:	Python 3.2			
Interpreter Executable:	C:\Program Files\Python32\python.exe Browse			
		Tu wpisz ścieżkę do pliku <i>python.exe</i>		
		OK Cancel		



Jeżeli się nie pomyliłeś, to po naciśnięciu **OK** pojawi się okno z propozycją ścieżek, które mają zostać umieszczone w systemowej zmiennej **pythonpath** (Rysunek 5.3.5). Zaakceptuj je bez zmiany:

🖨 Selection Needed
Select the folders to be added to the SYSTEM pythonpath! IMPORTANT: The folders for your PROJECTS should NOT be added here, but in your project configuration. Check:http://pydev.org/manual_101_interpreter.html for more details.
✓ ≦ C:\Program Files\Python32 ✓ ≦ C:\Program Files\Python32\DLLs ✓ ≦ C:\Program Files\Python32\lib ✓ ≦ C:\Program Files\Python32\lib\site-packages ✓ ≦ C:\Program Files\eclipse\plugins\org.python.pydev_2.1.0.2011052613\PySrc ✓ ≦ C:\WINDOWS\system32\python32.zip
Select All Deselect All
OK Cancel

Rysunek 5.3.5 Okno wyboru folderów do systemowej ścieżki PYTHONPATH

W rezultacie w oknie *Preferences* pojawi się skonfigurowany interpreter Pythona (Rysunek 5.3.6):

Preferences				
type filter text	Python Interpreters $\Leftrightarrow * \Rightarrow * *$			
⊞- General	Python interpreters (e.g.: python.exe)			
≣∾ C/C++ ≣∾ Help	Name Location New			
⊡ Install/Update ⊡ Java	Python 3.2 C:\Program Files\Python32\python.exe			
■ PyDev Builders Debug	Remove			
Editor	Skonfigurowany interpreter Pythona			
Interpreter - Python	Libraries Forced Builtins Predefined T Environment String Substitution Variables System PYTHONPATH			
Logging PyLint PyUnit Scripting PyDey	System The New Folder System libs C:\Program Files\Python32 C:\Program Files\Python32 New Folder New Folder			
Task Tags ⊕-Run/Debug ⊕-Tasks	C:\Program Files\Python32\lib C:\Program Files\Python32\lib C:\Program Files\Python32\lib\site-packages			
<	Restore Defaults Apply			
?	tę konfigurację			

Rysunek 5.3.6 Skonfigurowany interpreter Pythona

Po naciśnięciu w tym oknie przycisku **OK**, Eclipse przeanalizuje wszystkie pliki umieszczone w folderach wymienionych w *PYTHONPATH*. Przygotuje sobie w ten sposób indeksy do autokompletacji kodu i innych pomocy (Rysunek 5.3.7):



Rysunek 5.3.7 Okno przygotowujące pliki znalezione na PYTHONPATH

 Uwaga: Podczas instalowania PyDev w wersji 2.2.1 (2.2.1.2011071313) na Eclipse 3.7 ("Indigo") w wersji *Eclipse for Testers* podczas przygotowywania indeksów pojawił mi się komunikat o wyjątku (błędzie) w programie. Mimo to przygotowywanie dalej działało (pod spodem okna z komunikatem), i po kilkunastu sekundach się zakończyło. Wystąpienie tego błędu nie spowodowało żadnych zauważalnych nieprawidłowości w działaniu programu.

Podczas wcześniejszych instalacji PyDev 2.1.0 (2.1.0.2011052613) z Eclipse 3.6 ("Helios") w wersji *Eclipse for C/C++ Developers*, takiego błędu nie było.

Rozdział 6. Inne

W tym rozdziale umieściłem wszystkie pozostałe materiały dodatkowe. (To trochę taki "groch z kapustą". Trudno by było go dzielić na kolejne rozdziały).

6.1 Aktualizacja nagłówków API Blendera dla PyDev

API Blendera zmienia się wraz z kolejnymi wersjami programu. Dlatego w folderze *doc*, dostarczanym wraz z tą książką (por. str. 39), umieściłem skrót, który je uaktualni (Rysunek 6.1.1):

Adres 🛅 C:\Program Files\Blender\doc				✓ →
Foldery	×	Nazwa 🔺	Rozmiar	Тур
 ■ image: Blender ● image: 2.57 ■ image: 0 doc □ image: 0 python_api image: 0 pypredef image: 0 pypredef-tmp 	~	python_api refresh_python_api.bat Tym_sk *.pypred	1 KB rótem możesz u lef do aktualnej we	File Folder MS-DOS Batch Fi Iaktualnić pliki Irsji Blendera



Ten plik wsadowy uruchamia skrypt pypredef_gen.py, umieszczony w folderze doc\python_api (Rysunek 6.1.2):





Ten skrypt to przerobiona wersja pliku *sphinx_doc_gen.py*, opracowanego przez Campbella Bartona do generowania opisu API Blendera. (Tak, tego samego które jest wyświetlane na portalu *blender.org*). Dzięki temu kodowi, opisy wszystkich funkcji i metod w plikach dla Eclipse są takie same, jak w oficjalnej dokumentacji API. Dodatkowo umożliwiło to także umieszczenie listy parametrów każdej procedury oraz ich opisów. Jedynym modułem, który nie jest udokumentowany w ten sposób, jest *bge*. Oprócz niego także *bpy.context* może mieć pewne luki, gdyż jego zawartość zależy od typu (*3D View, Python Console*, itp.) aktualnego obszaru ekranu.

Rezultat działania skryptu — pliki *.*pypredef* poszczególnych modułów API — są umieszczone w folderze *doc\python_api\pypredef* (Rysunek 6.1.3):





To właśnie ten folder należy wskazać w konfiguracji projektu PyDev jako bibliotekę zewnętrzną (external library).

Gdy wgrasz do tego folderu nową wersję Blendera (lub przeniesiesz folder *doc* do nowej — wszystko jedno) koniecznie uruchom plik wsadowy *doc\refresh_python_api.bat* (Rysunek 6.1.4):

🔤 refresh_python_api 💦 🗖 🎽
NA Warning: Current value "0" matches no enum in 'EnumProperty', 'modifier', 'de ' 'EnumProperty', 'scene', 'defaul 'EnumProperty', 'scene', 'defaul 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defaul 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defaul 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defaul 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'NA Warning: Current value "0" matches no enum in 'EnumProperty', 'scene', 'defau 'SenumProperty', 'scene', 'defau 'SenumProperty'

Rysunek 6.1.4 Aktualizacja nagłówków Python API Blendera

I to wszystko. Jeżeli jednak zaczynasz nowy projekt w PyDev, zawsze musisz dodać ścieżkę *doc\python_api\pypredef* do jego konfiguracji. Przejdź do właściwości projektu (Rysunek 6.1.5):



Rysunek 6.1.5 Przejście do właściwości projektu

W oknie, które się pojawi, wybierz sekcję *PyDev – PYTHONPATH*, a w niej — zakładkę *External Libraries* (Rysunek 6.1.6):

Properties for Bevel				
1. Wybierz tę pozycję Builders Project Raferences PyDev - Interpreter/Gra PyDev - PYTHONPATH Run/Debug Settings	PyDev - PYTHONPATH 2. Przejdź do tej zakładki			
	The final PYTHONPATH used for a launch is composed of the pacing defined here, joined with the paths defined by the selected interpreter.			
	🕮 Source Folders 📓 External Libraries 💿 String Substitution Variables			
	External libraries (source folders/zips/jars/eggs) outside of the workspace.			
Task Repository	When using variables, the final paths resolved must be filesystem absolute.			
WikiText	Changes in external libraries are not monitored, so, the 'Force restore internal info'			

Rysunek 6.1.6 Przejście do edycji PYTHONPATH

Początkowo projekt nie ma żadnych dodatkowych bibliotek (lista w tej zakładce jest pusta). Naciśnij przycisk *Add source folder*, aby dodać nową, i wskaż folder *doc\python_api\pypredef* (Rysunek 6.1.7):

	PyDev - PYTHONPATH	↓ → → ▼	
	The final PYTHONPATH used for a launch is composed of the paths defined here, joined with the paths defined by the selected interpreter.		
	🗁 Source Folders 📓 External Libraries 🕒 String Substitution Variables		
	External libraries (source folders/zips/jars/eggs) outside of the workspace.		Tym przyciskiem dodasz nowa
	When using variables, the final paths resolved must be filesystem absolute.		
	Changes in external libraries are not monitored, so, the 'Force restore internal info' should be used if an external library changes.		
Na razie lista zewnętrznych		Add source folder	
bibliotek jest pusta -		Adda legg	
	Przeglądanie w poszukiwaniu folder	u	? 🛛
	Force restore internal info		
	I Biender		2
I	🖬 🧰 doc		<u> </u>
	□ 📄 python_api ◯ pypredef		
	🗀 pypredef-t	mp	
	E 🛅 Blender-2.49		
	⊞ Diender-2,56 ⊞ C Biender-2,57		
	■ 🖬 Blender-2.57a		✓
	Folder: pypredef		
	Utwórz nowy folder		nuluj

Rysunek 6.1.7 Dodanie nagłówków API jako "biblioteki zewnętrznej"

Gdy folder z deklaracjami Blender API jest już na liście, naciśnij przycisk *Force restore internal info*. Jak wynika z opisu w oknie, tak należy robić tu zawsze, po wprowadzeniu zmiany (Rysunek 6.1.8):

PyDev - PYTHONPATH	⇔ • ⇒ • •
The final PYTHONPATH used for a launch is composed of the paths defined here, joined with the paths defined by the selected interpreter.	
😕 Source Folders 📓 External Libraries 🕒 String Substitution Variables	
External libraries (source folders/zips/jars/eggs) outside of the workspace.	
When using variables, the final paths resolved must be filesystem absolute.	
Changes in external libraries are not monitored, so, the 'Force restore internal info' should be used if an external library changes.	
C:\Program Files\Blender\doc\python_api\pypredef	Add source folder
	Add zip/jar/egg
	Add based on variable
Naciśnij ten przycisk	Remove
Force restore internal info	

Rysunek 6.1.8 Wymuszenie odświeżenia wewnętrznych danych projektu

Po zatwierdzeniu zmian w konfiguracji projektu, dodaj na początek skryptu deklarację "import bpy". Dzięki niej PyDev będzie wiedział, że ma wyszukać deklaracje z modułu *bpy* (Blender API). Potem wystarczy, że w czasie pisania kodu postawisz kropkę po nazwie obiektu. Edytor wyświetli wówczas listę jego metod i właściwości (Rysunek 6.1.9):



Rysunek 6.1.9 Autokompletacja kodu — po wpisaniu kropki

Więcej o posługiwaniu się funkcjami autokompletacji znajdziesz na str. 41 i następnych.

UWAGA: Na oficjalnej stronie <u>pydev.org</u> znajdziesz opis innego sposobu korzystania z plików predefinicji (*.*pypredef*)¹. Problem w tym, że opisane tam dodawanie do sekcji *Predefined* nie chciało mi działać. Przedstawiłem więc metodę sprawdzoną i skuteczną, choć "nieortodoksyjną".

Uważam zresztą, że przypisywanie takiego powiązania z Blender API do projektu, a nie interpretera, jest lepsze. Możesz przecież równolegle pisać inny projekt w "zewnętrznym Pythonie". W takim projekcie podpowiedzi związane z Blenderem mogłyby tylko przeszkadzać.

¹ Dokładny adres tego artykułu: <u>http://pydev.org/manual_101_interpreter.html</u>

6.2 Importowanie pliku do projektu PyDev

Do projektu PyDev można dołączyć jakieś istniejące pliki. Z menu kontekstowego folderu, w którym mają się znaleźć, wybierasz polecenie *Import...* (Rysunek 6.2.1):

File Edit Source Refactoring Navigate Search P	oject Pydev Run Window Help
📬 • 🔚 🕼 🗁 🎓 🏇 • 💽 • 🔗	• 1 🎨 🔶 • 🔿 •
🛱 PyDev Package E 🛛 🦳 🖻 mesh_bev	el 🛛 🖻 bpy 📃 🗖 🗖
□ \$ \$ 0	<u>^</u>
Bevel 1. Podświetl do otwórz jego men	zelowy folder i ^{ript} u kontekstowe
in P mesh_bevel.py → blender file → blender file	ain():
E C Python 3 New	bujecca[cane]
Go Into	
Сору	
👔 Paste	
🔀 Delete	
Move	
Rename 2. V	/ywołaj to polecenie
& Remove from Context	trl+Alt+Shift+Down
import	
🛃 Export	

Rysunek 6.2.1 Import istniejącego elementu do folderu projektu

To otworzy okno kreatora importu. Na pierwszym ekranie wskaż *General → File System* jako rodzaj źródła (Rysunek 6.2.2):

🖨 Import	
Select Import resources from the local file system into an existing project.	Ľ
Select an import source: type filter text	
General Archive File Existing Projects into Workspace File System Preference C/C++ CVS	
Back Next > Finish	Cancel

Rysunek 6.2.2 Wybór rodzaju źródła

W kolejnym oknie kreatora należy najpierw wybrać folder, w którym znajduje się plik/pliki (Rysunek 6.2.3):

🖨 Import	\mathbf{X}
File system	1,
Source must not be empty. Kliknij tutaj, by wskazać folder z plikami	
From directory:	
Jedyne wypełnione domyślnie pole	
Filter Types Select All Deselect All to folder docerowy (projektu) Into folder: Bevel/blender file Browse	7
Options Overwrite existing resources without warning Create complete folder structure Advanced >>	
Formularz jest niewypełniony, dlatego <i>Finish</i> jest niedostępny	
(?) < Back	

Rysunek 6.2.3 Puste okno kreatora importu

Po naciśnięciu przycisku Browse... możesz wskazać istniejący katalog, lub utworzyć nowy (Rysunek 6.2.4):

Import from directory		? 🗙
Select a directory to import from.	Wskaż folder, który zawiera plik.	
Folder: Python		
Utwórz nowy folder		nuluj

Rysunek 6.2.4 Wybór folderu

Po wybraniu folderu, w liście po prawej pojawią się jego pliki (Rysunek 6.2.5):

🖨 Import		
File system		
🔞 There are no resources cur	rently selected for import.	
· · · · · · · · · · · · · · · · · · ·	Na razie nie mamy wybra- nego żadnego pliku	
From directory: C:\Documen	nts and Settings\w497972‡\My Documents	Blender\2.5 🔽 Browse
Pliki we v folderze	wskazanym	ch_v0_6_alpha.zip nd nd1 nd2
Filter Types Select A	I Deselect All	

Rysunek 6.2.5 Wyświetlanie zawartości folderu

Zaznacz co najmniej jeden z nich do importu (Rysunek 6.2.6):

🖨 Import		
File system Import resources From directory:	from the local file system. Gdy zaznaczysz jakiś plik – zniknie komunikat o błędzie C:\Documents and Settings\w4979721\My Documents\Blender\2.5 V Brows	e
🔳 🥟 Pytho	on ArraySketch_v0_6_alpha.zip ArraySketch_v0_6_alpha.zip ArraySketch_v0_6_alpha.zip bevel.blend bevel.blend1 bevel.blend2 P Bevel.py Select All Deselect All	~
Into folder: Bev Options Overwrite ex Create compl Advanced >>	vel/blender file Brows isting resources without warning lete folder structure	e
	a przycisk <i>Finish</i> stanie się dostępny	
?	< Back Next > Finish Car	ncel

Rysunek 6.2.6 Wybór pliku do importu

Gdy to zrobisz, kreator dojdzie do wniosku, że ma już komplet danych, i "odszarzy" przycisk *Finish*. Naciśnij go, a wybrane pliki zostaną skopiowane do wybranego folderu projektu.

Poprzez import możesz wstawić do projektu np. testowy plik Blendera. Wystarczy potem kliknąć, by go otworzyć (Rysunek 6.2.7). W ten sposób masz wszystko "w jednym miejscu":



Rysunek 6.2.7 Otwieranie testowego pliku Blendera

• PyDev domyślnie tworzy w folderze projektu kopie plików, które wskazałeś. Oznacza to, że wprowadzone do nich zmiany nie mają wpływu na pierwowzory.

Jeżeli jednak potrzebujesz włączyć do projektu plik, który musi leżeć gdzieś w innym katalogu — jest to także możliwe. Tak się może zdarzyć, gdy będziesz chciał bezpośrednio poprawiać np. wtyczkę Blendera. Skrypty wtyczek muszą się znajdować w folderze *addons*. Zaczyna się tak samo jak poprzednio (Rysunek 6.2.8):

🛱 PyDev Pack	age E 🛛 🗖 🗖	🖻 mesh_bevel 🛛 🕑 bpy	/
🖃 📂 Bevel		⊖,,, The simplest sc.	ript
🖻 🕮 📴			7
⊡ … P	New	•	
🖻 🗁 ble	Go Into		lata.objects["
🕀 🥐 Py	📄 Сору		(ame)
	💼 Paste		
	💢 Delete		
	Move		
	Rename		
	🕭 Remove from Context	Ctrl+Alt+Shift+Down	
	占 Import		
	🛃 Export は		

Rysunek 6.2.8 Dodawanie skrótu do pliku - początek jak poprzednio...

W oknie kreatora wskaż folder wtyczek Blendera i zaznacz tę, którą włączasz do projektu (Rysunek 6.2.9):

🖨 Import				
File system Import resources from the loca	l file system.	Fo	lder z wtyczk endera	ami
From directory: C:\Program F	Files\Blender\2.57\scrip	ts\addons_contr	ib 🗸	Browse
■		mesh_ir mesh_k P mesh_k P scene_t space_v	itersection.py nife.py nandle_panel.py view3d_adjust_pr Zaznaczam skr	ojection.py ypt, do którego
Into folder: Bevel/src			chcę często zac	Browse
Options Overwrite existing resource Create complete folder structure Advanced >	es without warning ucture i nacisk	am ten przycis	k	

Rysunek 6.2.9 Dodawanie do projektu skrótu do wtyczki Blendera

Następnie naciśnij przycisk Advanced>>, by rozwinąć dodatkowe opcje kreatora (Rysunek 6.2.10):

Options ✓ Overwrite existing resources without warning Create complete for zaznaczam, że to ma być skrót ✓ Create links in workspace ✓ Create virtual folders ✓ Create link locations relative to: ECLIPSE_HOME	dniesienia" ki do pliku
Pack Next > Finish C	Cancel

Rysunek 6.2.10 Wybór położenia i otoczenia skryptu

Najpierw zaznacz opcję *Create links in workspace*. Spowoduje to, że PyDev nie będzie tworzył lokalnej kopii pliku w swoim folderze. Zamiast tego umieści tam tylko referencję do oryginalnego skryptu. W ten sposób możesz np. wygodnie zmieniać kod wtyczki Blendera, umieszczonej w jego katalogu.

PyDev pozwala Ci także określić, czy ma zapamiętać pełną ścieżkę do wskazanego pliku, czy też ścieżkę względną. Służy do tego opcja *Create link locations relative to*. Osobiście zawsze używam ścieżek względnych, bo ułatwiają ewentualne przenoszenie projektu. PyDev umożliwia także określenie miejsca w strukturze katalogów, do którego ma się odnosić taka ścieżka względna. W przypadku wtyczek Blendera proponuję używać *ECLIPSE_HOME*. (Najprawdopodobniej i Blender, i Eclipse znajdują się u Ciebie w tym samym katalogu).

Gdy naciśniesz w oknie kreatora przycisk Finish, skrót zostanie dodany do projektu (Rysunek 6.2.11):



Rysunek 6.2.11 Skrót do pliku, dodany do projektu

Jak widzisz, ikony skrótów do plików są oznaczane w Eclipse dodatkową strzałką w prawym dolnym rogu. Gdy zajrzysz do właściwości takiego skrótu, możesz z nich odczytać lub zmienić położenie "oryginału" (Rysunek 6.2.12):

Resource		Tym przyciskiem możesz	$\Leftrightarrow \neg \Rightarrow \neg \bullet$
Path:	/Bevel/src/mesh_knife.py	przypisać do skrótu inny plik	
Туре:	Linked File		
Location:	ECLIPSE_HOME\\Blender\	2.57\scripts\addons_contrib\me	esh_knife.pyEdit
Resolved location:	C:\Program Files\Blender\2	.57\scripts\addons_contrib\mes	:h_knife.py
Size: Last modified:	18 182 Dytes 21 marca 2011 15:46:12		Aktualne położenie pliku

Rysunek 6.2.12 Przykładowa informacja o powiązanym pliku. (To fragment okna jego właściwości - Properties)

6.3 Debugowanie skryptu w Blenderze — szczegóły

Blender wykonuje skrypty Pythona posługując się swoim własnym interpreterem. Dlatego można je debugować za pomocą wbudowanego, standardowego debuggera. Niestety, to narzędzie działa wyłącznie w trybie "konwersacyjnym", w konsoli. Nie jest przez to zbyt wygodne w użyciu.

Szansę na śledzenie skryptu w jakimś "okienkowym" debuggerze, takim jakim dysponuje IDE Eclipse, daje tylko tzw. debugger zdalny. To rozwiązanie wymyślone oryginalnie do śledzenia programów wykonywanych na innym komputerze (Rysunek 6.3.1):



Rysunek 6.3.1 Śledzenie wykonywania skryptu w Blenderze: działanie zdalnego debuggera

IDE (takie jak Eclipse) uruchamia proces serwera, który zaczyna "nasłuchiwać" żądań od ewentualnych debugowanych skryptów. Informacje o tym będzie wysyłał klient zdalnego debuggera, włączony w kodzie śledzonego skryptu. W naszym przypadku kod tego klienta znajduje się w pakiecie (*package*) o nazwie *pydevd*. Ten kod jest importowany i inicjowany w pomocniczym module *pydev_debug.py* (por. str. 129), który jest wykorzystany z kolei w pliku *Run.py*. (To plik, który wywołuje nasz skrypt — por. str. 58). Komunikacja pomiędzy klientem i serwerem zdalnego debuggera odbywa się poprzez sieć. Już dawno temu ktoś spostrzegł, że nic nie stoi na przeszkodzie, by te dwa procesy działały na tej samej maszynie. Wymieniają wtedy między sobą dane korzystając z lokalnej karty sieciowej komputera. Koncepcyjnie odpowiada to sytuacji, jak gdyby dwóch ludzi siedzących w tym samym pokoju rozmawiało przez telefon. Ale programy "są głupie" i nie narzekają, że muszą się porozumiewać tak okrężną drogą. A całość działa poprawnie, i to się tylko liczy.

Proces serwera w PyDev uruchamiasz poleceniem *PyDev→Start Debug Server* (Rysunek 6.3.2):

File Edit Source Refacto	ring Navigate Search	Project Pydev	Run Window	Help	
: 📬 • 🔛 🕼 🗁 🗄 🖡	📕 🌮 i 🎓 i 🏇 •	- 🜔 - 📑 End	Debug Server	📑 🔁 🕏	🏂 Debug 🥭 PyDev
The Debug		🗖 🔁 Sta	rt kjebug Server	reaknoints	
	Nie pomyl te	ao skró-	bals Browser	x ⇒	
🍇 🛤 🕩 💷 🕹	tu z "nor debugowanie	rmalnym" em!			

Rysunek 6.3.2 Polecenia PyDev i ich skróty, uruchamiające i wyłączające serwer zdalnego debuggera Pythona w Eclipse

A co zrobić, gdy na pasku przycisków ani w menu *Pydev* nie ma poleceń¹, które pokazuje Rysunek 6.3.2?

Czasami polecenia PyDev *Start/End Debug Server* mogą być po prostu wyłączone z perspektywy *Debug*! Aby je włączyć, przejdź do okna dostosowywania perspektywy: *Window→Customize Perspective* (Rysunek 6.3.3).

W oknie *Customize Perspective* przejdź do zakładki *Command Groups Availability* (Rysunek 6.3.4). Odszukaj na liście *Available command groups* (po lewej) grupę poleceń o nazwie *PyDev Debug*. Wystarczy ją zaznaczyć, by polecenia *Start Debug Server* i *End Debug Server* pojawiły się w menu i na pasku przybornika!

edef	\bpy.pypredef - Eclipse	
Run	Window Help	
•	New Window New Editor	😭 🏇 Debug 🌏 PyDev
(×)= \	Open Perspective Show View	
	Customize Perspective Save Perspective As Reset Perspective Close Perspective	
<	Close All Perspectives	>
	Preferences	ا• ﷺ

Rysunek 6.3.3 Przejście do dostosowywania perspektywy projektu

Customize Perspective - Debu	ug	
Tool Bar Visibility Menu Visibility Co Select the command groups that you w menu items and/or toolbar items are a	mmand Groups Availability Shortcuts want to see added to the current perspective dded to the perspective by the selected com	e (Debug). The details field identifies which mand group.
Available command groups: Make Actions Open Files Profile PyDev Debug PyDev Navigation Reverse Debugging Search	Menubar details: Pydev Menubar details: Pydev End Debug Server Start Debug Server Ta grupa poleceń musi być włączona!	Toolbar details: PyDev Debug PyDev: stop the debugger serv PyDev: start the pydev server
?		OK Cancel

Rysunek 6.3.4 Włączenie wyświetlania kontrolek zdalnego debuggera Pythona

Gdy przygotowywałem prezentację na potrzeby tej książki, przyciski *Start/End Debug Server* były od razu na swoim miejscu. Nie musiałem niczego poprawiać w konfiguracji perspektywy. Przypuszczam, że może być to związane z okolicznościami, w których została w projekcie stworzona perspektywa *Debug*.

🛛 Przy okazji dowiedziałeś się, jak można dostosowywać perspektywę projektu Eclipse do swoich potrzeb 😊

¹ Gdy instalowałem PyDev po raz pierwszy, zdarzyło mi się właśnie coś takiego. Spędziłem wtedy chyba cały dzień na wertowaniu wszelkiej dokumentacji i uwag użytkowników, dostępnych w Internecie. Równolegle ciągle wchodziłem w różne menu Eclipse, w poszukiwaniu tych dwóch kluczowych drobiazgów. W końcu je znalazłem. Abyś oszczędzić Wam tej samej męki, podaję tutaj rozwiązanie tego problemu

Podczas debugowania skryptu będziesz często sprawdzał aktualny stan zmiennych. W Eclipse umożliwia to panel *Variables* (Rysunek 6.3.5):

(×)= Variables	🛛 🗣 Breakpoints			4		Na żółto są zaz		zmier	nne,		
Name		Value				ktorym w poprzednim kroku nada no nowa wartość					
🗄 🔍 Globa	als	Global variables						_			
😑 🔍 edge	•	MeshEdge: <bpy< td=""><td>_struct, M</td><td>eshEdge at 0x0E9A</td><td>430C</td><td>></td><td></td><td></td><td></td></bpy<>	_struct, M	eshEdge at 0x0E9A	430C	>					
• 6	bevel_weight	float: 0.0						_			
	ol_rna	MeshEdge: <bpy_< td=""><td>_struct, St</td><td>ruct("MeshEdge")></td><td>•</td><td></td><td></td><td>_</td><td></td></bpy_<>	_struct, St	ruct("MeshEdge")>	•			_			
* c	rease	float: 0.0					Wortoćć	zmio	nnoi		
• h	nide	bool: False			\wedge		można	zmi	ienić		
• ir	ndex	int: 1	• c	rease	\sim	1	wpisując	tutaj	j jej		
• is	s_fgon	bool: False	• h	ide		bool: False	nową				
• is	s_loose	bool: False	• ir	ndex		int: 1		_			
_	ey.	tuple: (0, 3)									
	• 0	int: 0									
	• 1	int: 3									
	• _len Okno	pokazujące wa	rtość					_			
± • n	na_type podśw	vietlonej zmiennej	i, St	ruct("MeshEdge")>	•			~			
float: O								^			
1											
	Assign Value	Ctrl+S 🔸		W tym ol	knie 1	także możesz z	mieniać	~			
	The Content Assist	Ctrl+Space		wartość z polecenia	mien	nej — za pomo	cą tego	>			
<	Cut	Ctrl+X									

Rysunek 6.3.5 Panel śledzenia zmiennych skryptu (globalnych i lokalnych)

Panel jest podzielona na listę z nazwami i wartościami zmiennych globalnych i lokalnych, oraz na obszar szczegółów (*details*). W obszarze szczegółów pokazywana jest wartość zmiennej, którą podświetliłeś na liście. Sądzę, że obszar *details* przydaje się do sprawdzania jakichś dłuższych ciągów znaków (wartości typu *str*). Gdy wartością zmiennej jest referencja do obiektu, Eclipse wyświetla przy nich ikonki [+]/[–], za pomocą których możesz rozwinąć listę pól tych obiektów.

W oknie *Variables* możesz także zmieniać aktualne wartości zmiennych. Zazwyczaj będziesz je po prostu wpisywał w kolumnie *Value* (Rysunek 6.3.5). Możesz je jednak także zmienić w obszarze *details* (poleceniem *Assign Value* z menu kontekstowego).

Do śledzenia pojedynczego pola obiektu wygodniej jest korzystać z panelu *Expressions*. Możesz go dodać do perspektywy poleceniem *Window→Show View→Expressions* (Rysunek 6.3.6):

Window Help						
New Window New Editor	E 🕄	🌣 Debug 🍓 PyDev				
Open Perspective	Sreakpoints	<u> </u>	$\diamond \diamond \bullet \bullet$	*	🛛 🖻 🏂	Debug
Show View 🕨 🕨	©₀ Breakpoints	Alt+Shift+Q, B	(×)= Variables	ら Breakpoin	ିଙ୍କୁ Expressio	×
Customize Perspective		Alt+Shift+Q, ⊂			🏝 🏘 🗖	+ >
Save Perspective As	Sebug		Name		Value	
Close Perspective			🔶 🔶 Add n	ew expression		
Close All Devenestives	🥺 Error Log	Alt+Shift+Q, L				
Close All Perspectives	🜔 Executables	N				
Navigation 🕨	🔣 Expressions		\geq			
Preferences	📋 Memory 🛛 😽		1			

Rysunek 6.3.6 Dodawanie panelu *Expressions*

Układ panelu Expressions przypomina układ Variables: tu także mamy listę z nazwą i wartością wyrażenia. Jest tu także pole details, pokazujące na większym obszarze wartość zaznaczonej pozycji. To, co różni je od Variables to możliwość wpisania dowolnego wyrażenia, które ma być wartościowane po każdym kroku debuggera (Rysunek 6.3.7):



Rysunek 6.3.7 Dodawanie śledzonych wyrażeń do listy Expressions

W oknie Expressions można wpisać po prostu nazwę zmiennej. Częściej jednak jest wykorzystywane do śledzenia wybranych pól jakiegoś obiektu. Można także wpisać tu odwołanie do konkretnego elementu listy (np. selected[0]). W odróżnieniu od okna Variables, w oknie Expressions nie można zmieniać wartości wyświetlonych wyrażeń.

Innym elementem, który jest przydatny przy debugowaniu skryptu jest konsola Blendera. To dodatkowe okno tekstowe programu, otwierane obok okna głównego. Podczas uruchomienia Blendera pierwsza pojawiała się jego konsola, a dopiero później okno główne. W wersji 2.57 musisz ją jednak ręcznie wyświetlać poleceniem Help→Toggle System Console (Rysunek 6.3.8).

Konsola Blendera to domyślne wyjście (standard output) wszystkich skryptów. (Nie myl jej z oknem Python Console! Tam widzisz tylko rezultaty wpisywanych ręcznie poleceń). W konsoli Blendera zobaczysz wszystkie teksty pochodzące z wywołań procedury print() w skryptach. Gdy podczas wykonywania wystąpi jakiś błąd, to właśnie tutaj znajdziesz jego dokładny opis.

🔤 C:\Program Files\Blender\blender.exe



Rysunek 6.3.8 Włączenie konsoli Blendera

l.blend

Ten komunikat pochodzi z wywołania debuggera PyDev (por. ze str. 28, Rysunek 2.3.3)

Podczas debugowania skryptu (tzn. w chwili, gdy wykonujesz pojedyncze kroki w kodzie programu) Blender jest "zamrożony". Dzieje się tak dlatego, że czeka na zakończenie całej operacji, którą rozpocząłeś naciskając przycisk *Run Script*.

Mimo to, jeżeli wpiszesz jakieś wyrażenie w konsoli *Debug Server* w Eclipse, to jego wartość zostanie wyświetlona w konsoli Blendera (Rysunek 6.3.9):



Rysunek 6.3.9 Współpraca konsoli Eclipse i konsoli Blendera podczas debugowania skryptu

Możesz to potraktować jako metodę "ad hoc" do sprawdzania wartości różnych wyrażeń — na przykład pola jakiegoś obiektu (Rysunek 6.3.10):



Rysunek 6.3.10 Współpraca konsoli Eclipse i konsoli Blendera — inne przykłady

Oczywiście, to samo można sprawdzić za pomocą panelu *Expressions*. W konsoli serwera możesz jednak więcej — na przykład wywołać metodę jakiegoś obiektu.

6.4 Co się kryje w pliku pydev_debug.py?

Właściwie do uruchomienia śledzenia skryptu Blendera w zdalnym debuggerze PyDev wystarczyłyby takie dwie linie kodu (Rysunek 6.4.1):

```
import pydevd
pydevd.settrace() #<-- debugger stops at the next statement</pre>
```

Rysunek 6.4.1 Kod, ładujący i uruchamiający klienta zdalnego debuggera PyDev

Oczywiście, aby ten kod zadziałał, należałoby wcześniej dodać do **PYTHONPATH** folder, w którym znajduje się pakiet (*package*) *pydev.py*. Zresztą ten warunek nie wyczerpuje jeszcze listy wszystkiego, co trzeba lub warto wykonać podczas inicjalizacji. Stąd te dwie linie "obrosły" w całą procedurę **debug()** (Rysunek 6.4.2):



Rysunek 6.4.2 Skrypt pydev_debug.py

Zdecydowałem się wydzielić główny kod uruchamiający w Blenderze skrypt z projektu Eclipse w oddzielny moduł *pydev_debug.py*. Ten moduł zawiera tylko jedną procedurę: *debug()* (Rysunek 6.4.2). Pozwoliło to na maksymalne uproszczenie skryptu *Run.py* — wzorca wywołania, dostosowywanego do nowego projektu (por. str. 58). *Run.py* to po prostu wywołanie procedury *debug()* z następującymi parametrami:

- script: ścieżka do pliku skryptu, który ma być uruchomiony;
- *pydev_path*: scieżka PyDev, w której znajduje się plik *pydevd.py*;
- trace: opcjonalny. True, gdy program ma być śledzony w debuggerze, False gdy ma być wykonany bez śledzenia. (Tylko wtedy, gdy trace = False, możesz wywoływać tę procedurę z wyłączonym procesem serwera PyDev w Eclipse por. 124);

Zwróć uwagę (Rysunek 6.4.2), że procedura **debug()** wczytuje podany moduł **script** klauzulą *import*. To pozwala używać jej także do debugowania wtyczki (*add-on*) Blendera¹. Przed importem program próbuje potraktować załadowany moduł jako *add-on* i wyrejestrować jego poprzednią wersję. Jak się nie uda — nie ma błędu (nie każdy skrypt musi być wtyczką). Po załadowaniu skryptu próbuje z kolei go zarejestrować.

Gdy piszesz kod wtyczki Blendera (*add-on*), zawsze dopisz na jej końcu obydwie wymagane metody: *register()* i *unregister()*. Pozwoli to na poprawne inicjowanie kodu za każdym wywołaniem w Blenderze. (Tzn. po każdym naciśnięciu przycisku *Run Script* w oknie z kodem *Run.py* — por str. 60).

¹ Wtyczka implementuje jedną lub więcej klas pochodnych *bpy.types.Operator* lub *bpy.types.Panel*. Oprócz tego musi zawierać dwie metody: *register()* i *unregister()*, zawierające polecenia związane z rejestracją klas wtyczki "do użytku" w Blenderze. Podczas wczytywania modułu środowisko ma wywołać metodę *register()*, a gdy moduł jest wyłączany — *unregister()*. Potem Blender sam sobie tworzy, kiedy to uzna za stosowane, instancje klas wtyczki. (To model działania "nie wołaj mnie, to ja do ciebie wywołam". Takim samym zasadom podlegają np. wszystkie programy w środowisku Windows). Dlatego podczas debugowania wtyczek koniecznie trzeba w ich kodzie umieścić punkty przerwania. Zadziałają poprawnie i zatrzymają w debuggerze wykonywanie skryptu klasy, "wywołanej" przez Blendera.

6.5 Pełen kod wtyczki mesh_bevel.py

W kolejnych rozdziałąch tej książki stopniowo rozbudowywałem kod jednego skryptu: *mesh_bevel.py*. W tekście przytaczałem często dodawane fragmenty kodu. Jednak po tylu modyfikacjach warto jest pokazać ostateczny rezultat w całości. Gdybyś chciał skopiować ten tekst do schowka — uważaj na wycięcia! Podczas takiego kopiowania z PDF wszystkie są usuwane. Lepiej chyba będzie po prostu pobrać ten plik z <u>mojej strony</u>.

Skrypt nie mieści się na jednej stronie, więc zdecydowałem się go podzielić na trzy części. Pierwsza część to nagłówek, z licencją GPL, strukturą informacyjną, i deklaracjami importu (Rysunek 6.5.1):

```
##### BEGIN GPL LICENSE BLOCK #####
  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
#
  as published by the Free Software Foundation; either version 2
#
#
  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
#
  GNU General Public License for more details.
#
  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software Foundation,
#
#
# ##### END GPL LICENSE BLOCK #####
. . .
Bevel add-on
A substitute of the old, 'destructive' Bevel command from Blender 2.49
...
#--- ### Header
bl info = {
    "name": "Bevel",
    "author": "Witold Jaworski",
    "version": (1, 0, 0),
    "blender": (2, 5, 7),
    "api": 36147,
    "location": "View3D > Specials (W-key)",
    "category": "Mesh",
    "description": "Bevels selected edges",
    "warning": "",
    "wiki url": "",
    "tracker url": ""
    }
#--- ### Imports
import bpy
from bpy.utils import register module, unregister module
from bpy.props import FloatProperty
                                             Ciąg dalszy na następnej stronie...
```

Rysunek 6.5.1 Skrypt mesh_bevel.py, cz. 1 (deklaracje)

Kolejny fragment skryptu to procedura bevel(), implementująca właściwą operację fazowania (Rysunek 6.5.2):

```
###
def bevel(obj, width):
    """Bevels selected edges of the mesh
       Arguments:
            @obj (Object): an object with a mesh.
                           It should have some edges selected
            @width (float):width of the bevel
       This function should be called in the Edit Mode, only!
    .....
    #edge = bpy.types.MeshEdge
    #obj = bpy.types.Object
    #bevel = bpy.types.BevelModifier
    bpy.ops.object.editmode toggle() #switch into OBJECT mode
    #adding the Bevel modifie:
    bpy.ops.object.modifier add(type = 'BEVEL')
    bevel = obj.modifiers[-1] #the new modifier is always added at the end
    bevel.limit method = 'WEIGHT'
    bevel.edge_weight_method = 'LARGEST'
    bevel.width = width
    #moving it up, to the first position on the modifier stack:
    while obj.modifiers[0] != bevel:
        bpy.ops.object.modifier move up(modifier = bevel.name)
    for edge in obj.data.edges:
        if edge.select:
            edge.bevel weight = 1.0
    bpy.ops.object.modifier_apply(apply_as = 'DATA', modifier = bevel.name)
    #clean up after applying our modifier: remove bevel weights:
    for edge in obj.data.edges:
        if edge.select:
            edge.bevel weight = 0.0
    bpy.ops.object.editmode toggle() #switch back into EDIT MESH mode
                                        Ciąg dalszy na następnej stronie...
```

```
Rysunek 6.5.2 Skrypt mesh_bevel.py, cz. 2 (procedura fazująca)
```

Ostatnią częścią skryptu jest implementacja operatora Bevel i kodu rejestrującego wtyczkę (Rysunek 6.5.3):

```
#--- ### Operato
class Bevel(bpy.types.Operator):
    ''' Bevels selected edges of the mesh'''
   bl idname = "mesh.bevel"
   bl label = "Bevel"
   bl description = "Bevels selected edges"
   bl options = { 'REGISTER', 'UNDO' } #Set this options, if you want to update
    #--- parameters
   width = FloatProperty(name="Width", description="Bevel width",
                          subtype = 'DISTANCE', default = 0.1, min = 0.0,
                                                     step = 1, precision = 2)
    #--- other fields
   LAST WIDTH NAME = "mesh.bevel.last width" #name of the custom scene property
    #--- Blender interface methods
    @classmethod
   def poll(cls,context):
        return (context.mode == 'EDIT MESH')
   def invoke(self, context, event):
        #input validation: are there any edges selected?
       bpy.ops.object.editmode_toggle()
        selected = list(filter(lambda e: e.select, context.object.data.edges))
       bpy.ops.object.editmode toggle()
        if len(selected) > 0:
            last width = context.scene.get(self.LAST WIDTH NAME,None)
            if last width:
                self.width = last width
            return self.execute(context)
        else:
            self.report(type='ERROR', message="No edges selected")
            return { 'CANCELLED' }
   def execute(self, context):
       bevel(context.object, self.width)
        context.scene[self.LAST WIDTH NAME] = self.width
       return { 'FINISHED'}
def menu draw(self, context):
   self.layout.operator context = 'INVOKE REGION WIN'
   self.layout.operator(Bevel.bl idname, "Bevel")
#--- ### Register
def register():
   register module( name )
   bpy.types.VIEW3D_MT_edit_mesh_specials.prepend(menu_draw)
def unregister():
   bpy.types.VIEW3D MT edit mesh specials.remove(menu draw)
   unregister module( name )
#--- ### Main code
if __name__ == '__main__':
   register()
```

Rysunek 6.5.3 Skrypt mesh_bevel.py, cz. 3 (kod wtyczki)

Bibliografia

Książki

[1]	Thomas Larsson,	Code	snippets.li	ntroduction	to	Python	scripting	for	Blender	2.5x,	free	e-book,
	2010.											

[2] Guido van Rossum, *Python Tutorial*, (part of Python electronic documentation), 2011

Internet

- [1] <u>http://www.blender.org</u>
- [2] <u>http://www.python.org</u>
- [3] <u>http://www.eclipse.org</u>
- [4] <u>http://www.pydev.org</u>
- [5] http://wiki.blender.org



Jeżeli masz już pewne doświadczenie w programowaniu, i zamierzasz napisać jakiś dodatek do programu Blender 3D, to ta książka jest dla Ciebie!

Pokazuję w niej, jak zestawić wygodne środowisko do pisania skryptów Blendera. Wykorzystuję do tego oprogramowanie Open Source: pakiet Eclipse, rozbudowany o wtyczkę PyDev. To dobra kombinacja, udostępniająca użytkownikowi wszystkie narzędzia, pokazane na ilustracjach wokół tego tekstu.

Książka zawiera także praktyczne wprowadzenie do API Blendera. Tworzę w niej od podstaw wtyczkę z nowym poleceniem programu. Omawiam szczegółowo każdą fazę implementacji. Pokazuję w ten sposób nie tylko same narzędzia, ale także metody, którymi się posługuję. Ten opis pozwoli Ci nabrać wprawy, potrzebnej do samodzielnej pracy nad kolejnymi skryptami.



ISBN: 978-83-931754-1-3

Bezpłatna publikacja elektroniczna

