Розділ 16
Морфи

Графічний інтерфейс Pharo називають *Morphic*[1]. Він підтримує два головні аспекти: з одного боку, Morphic визначає всі низькорівневі графічні сутності та відповідну інфраструктуру (події, перемальовування тощо), з іншого – він визначає всі доступні у Pharo графічні елементи (віджети). Morphic написаний на Pharo, тому без обмежень працює в усіх операційних системах. Як наслідок, Pharo виглядає однаково в Unix, MacOS та Windows. На відміну від більшості інших графічних інструментів Morphic не має окремих режимів для *компонування* і *виконання* інтерфейсу: користувач може у будь-який момент зібрати або розібрати кожен графічний елемент. Ми вдячні Ілеру Фернандесу (Hilaire Fernandes) за дозвіл побудувати цей розділ з використанням його оригінальної статті французькою.

16.1. Історія створення

Morphic розробили Джон Мелоні (John Maloney) і Ренді Сміт (Randy Smith) для мови програмування Self десь на початку 1993 року. Пізніше Мелоні написав нову версію Morphic для Squeak, але головні ідеї початкової версії досі живі-здорові та працюють у Morphic для Pharo – це *безпосередність* і *активність*. Безпосередність означає, що фігури на екрані є об'єктами, які можна дослідити та змінити напряму, просто клацнувши на них мишкою. Активність означає, що графічний інтерфейс здатний щомиті реагувати на дії користувача, інформація на екрані оновлюється, щойно зміниться стан, який вона відображає. Простий приклад використання цих ідей: можемо продублювати пункт меню і перетворити його на кнопку, що діятиме так само.

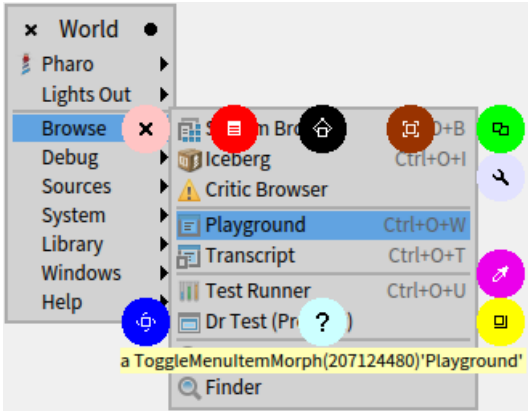

Рис. 16.1. Меню-ореол графічного елемента використовують для маніпуляцій з ним

Відкрийте Головне меню, оберіть якийсь з його пунктів, наприклад, «*Playground*» з розділу «*Browse*», і метаклацніть на ньому, щоб відкрити його меню-ореол, як на рис. 16.1. Клацніть на зеленому маніпуляторі «*Duplicate*», щоб створити копію пункту меню, і перенесіть копію в довільне місце на екрані (рис. 16.2). Ще одне клацання зафіксує

[1] У англійській мові суфікс *-morphic* означає «той, що має форму, вигляд, структуру». Таке значення якнайкраще підходить для вікон графічного інтерфейсу (*прим. – Ярошко С.*).

кнопку в обраному місці. Тепер можна випробувати її дію і переконатися, що вона працює так само, як початковий пункт меню. Примітно, що створена кнопка завжди залишатиметься видимою, виринаючи поверх усіх вікон Pharo. Щоб прибрати її з екрана, знову скористайтеся меню-ореолом.

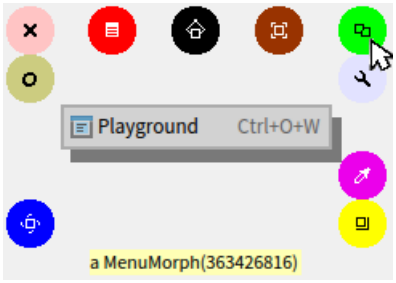

Рис. 16.2. Копію рядка меню можна перетворити на окрему кнопку

Цей приклад демонструє, що ми називаємо *безпосередністю* і *активністю*. Вони надають широке коло можливостей під час розробки альтернативного інтерфейсу користувача та прототипування альтернативних взаємодій.

Morphic потроху старіє, і спільнота Pharo вже кілька років працює над можливою заміною. Заміна Morphic означає розробку нової низькорівневої інфраструктури та нових наборів графічних елементів. Проєкт називається Block, уже виконано кілька ітерацій. Block – це інфраструктура, а Brick – набір побудованих поверх неї графічних елементів. Але давайте отримувати задоволення від Morphic.

16.2. Морфи

Усі об'єкти, які видно на екрані запущеного Pharo, – це "морфи", тобто екземпляри підкласів *Morph*. Сам собою *Morph* – це великий клас з багатьма методами, що дає змогу похідним класам реалізувати цікаву поведінку малою кількістю коду.

Від перекладача. Надалі в тексті використовуватимемо слово *морфа* як термін. Для цього є кілька причин. Видимі елементи інтерфейсу користувача – екземпляри класу *Morph*, морфи. З давньогрецької μορφη – вид, зовнішність, форма, що добре відображає сутність цих об'єктів. Слово «морфа» – анаграма слова «форма», а формами часто називають графічні елементи інтерфейсу користувача в ОС Windows, проте говоритимемо саме «морфа», бо інтерфейс Pharo однаковий для різних операційних систем.

Можна створити морфу, яка відображає довільний об'єкт, хоча отриманий результат залежить від об'єкта! Щоб створити морфу для зображення рядка, виконайте в Робочому вікні такий код.

```
'Morph' asMorph openInWorld
```

Він створить морфу для відображення рядка 'Morph' і відкриє її (тобто відобразить) на екрані, або «*у світі*», бо екран у Pharo називають *world*. Ви мали б отримати графічний елемент (екземпляр *Morph*), яким можна маніпулювати за допомогою метаклацання.

Звісно, можна визначати морфи з цікавішим графічним зображенням, ніж ми щойно бачили. Реалізований за замовчуванням у класі *Object class* метод *asMorph* лише створює *StringMorph*. Наприклад, «*Color tan asMorph*» поверне екземпляр StringMorph надписаний результатом виконання «*Color tan printString*». Щоб отримати кольоровий прямокутник, виконайте інший код.

```
(Morph new color: Color orange) openInWorld
```

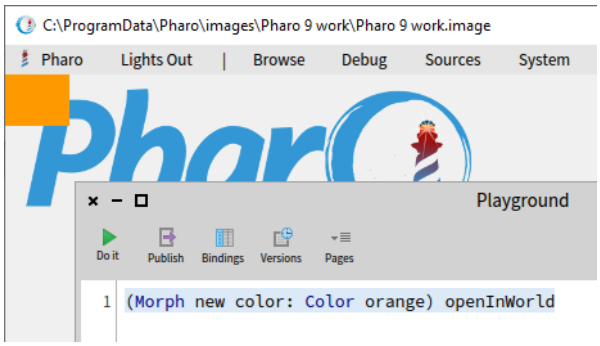

Рис. 16.3. За замовчуванням морфа прямокутна, розташована у лівому верхньому кутку екрана

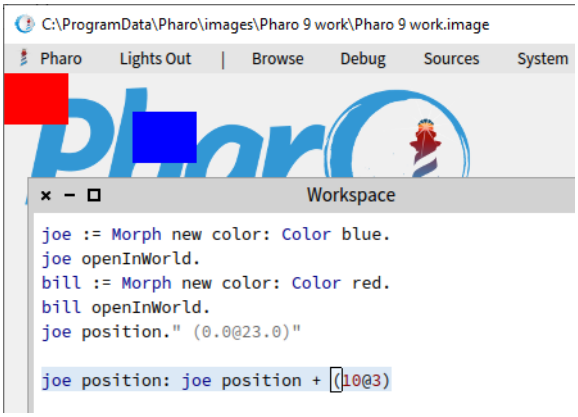

Рис. 16.4. *bill* і *joe* після десяти переміщень

Замість морфи у вигляді рядка отримано оранжевий прямокутник (рис. 16.3).

16.3. Маніпулювання морфами

Морфи – це об'єкти, тому ними можна маніпулювати як усіма іншими об'єктами у Pharo: за допомогою повідомлень можна змінювати їхній стан, створювати нові підкласи *Morph* тощо.

Кожна морфа, навіть невидима в поточний момент, має позицію і розмір. За домовленістю всі морфи займають прямокутну область екрана. Якщо морфа неправильної форми, то її позицію і розмір задає найменший описаний прямокутник, так званий обмежувальний прямокутник, або *межа*.

- Метод *position* повертає екземпляр *Point*, який описує розташування верхнього лівого кута морфи або його обмежувального прямокутника. Початком системи координат є верхній лівий кут головного вікна Pharo, координата *y* зростає донизу, а *x* – праворуч.
- Метод *extent* також повертає точку, але ця точка визначає не координати, а ширину та висоту морфи.

Наберіть наведений нижче код у Робочому вікні і виконайте його командою «*Do it*»[2].

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red.
bill openInWorld.
```

Тоді спробуйте виконати за допомогою «*Print it*» вираз «*joe position*», щоб довідатися поточне розташування першої морфи. Ви мали б отримати щось схоже на «*(0.0@23.0)*».

Тепер змінимо це розташування. Надрукуйте і виконайте кілька разів «*joe position: joe position + (10@3)*». Результат переміщень зображено на рис. 16.4.

Подібно можна взаємодіяти з розміром і іншими властивостями морфи.

```
joe extent. "повідомляє розмір морфи joe, <Print it>"
```

[2] Локальні змінні у Робочому вікні Pharo 9.0 можна не оголошувати (*прим. – Ярошко С.*).

```
>>> (50.0@40.0)
joe extent: joe extent * 1.1. "збільшує розмір морфи joe, <Do it>"
joe color: (Color orange alpha: 0.5). "задає колір морфи joe, <Do it>"

"задає відносне розташування морфи bill, <Do it>"
bill position: joe position + (100@0)
```

Щоб пропорційно збільшити морфу, виконайте за допомогою «*Do it*» вираз «*joe extent: joe extent * 1.1*». Щоб змінити колір морфи, надішліть їй повідомлення *color:* з бажаним екземпляром *Color* як аргумент, наприклад, «*joe color: Color orange*». Щоб додати прозорість, спробуйте «*joe color: (Color orange alpha: 0.5)*».

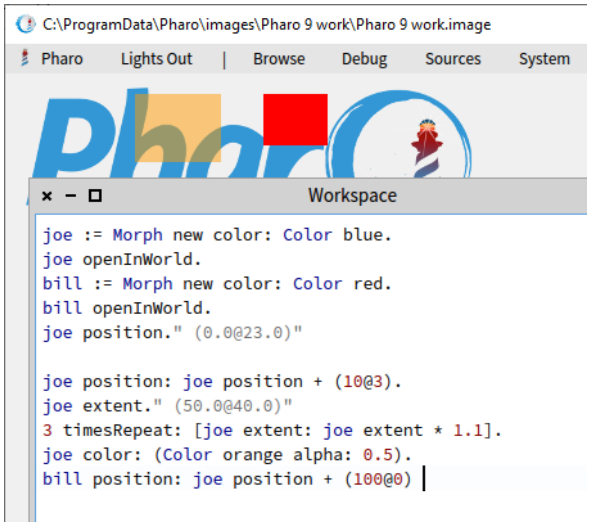

Рис. 16.5. *bill* слідує за *joe*

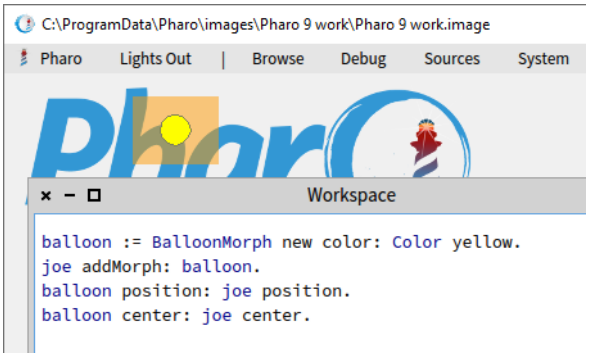

Рис. 16.6. *balloon* вбудовано в *joe* – оранжеву напівпрозору морфу

Для того, щоб задати розташування однієї морфи стосовно іншої, можна використати повідомлення «*bill position: joe position + (100@0)*». Виконуйте його щоразу після переміщення *joe*, і *bill* слідуватиме за нею. Наприклад, перетягніть *joe* мишею, виконайте код і побачите, що *bill* розташується на 100 цяток праворуч від *joe*. Результати виконання всіх повідомлень зображено на рис. 16.5.

Створені морфи можна вилучити:

- надсиланням повідомлення *delete*, наприклад, «*bill delete*»;
- за допомогою маніпулятора з хрестиком меню-ореолу.

16.4. Композиція морфів

Один зі способів створення нового графічного подання полягає у додаванні однієї морфи до структури іншої. Його називають *композицією*. Глибина композиції не обмежена. Щоб додати вкладену морфу до морфи-контейнера, контейнерові надсилають повідомлення *addMorph: anEnclosedMorph*.

Спробуємо додати нову морфу до створеної раніше, як у наведеному коді.

```
balloon := BalloonMorph new color: Color yellow.
joe addMorph: balloon.
balloon position: joe position.
```

Останній рядок розташовує кульку за тими самими координатами, що й *joe*. Зауважте, що координати вкладеної морфи відраховують стосовно екрана, а не стосовно контейнера. Використовувати абсолютні координати для позиціонування морфів, насправді, не дуже зручно. Ця особливість робить програмування морфів трохи дивним. Але є багато методів для задання позиції морфи, перегляньте протокол *geometry* класу *Morph* і відшукайте деякі з них. Наприклад, щоб розташувати *balloon* посередині *joe*, виконайте «*balloon center: joe center*» (див. рис. 16.6).

Якщо тепер спробувати перетягнути мишкою *balloon*, то виявиться, що мишка захопила і *joe*, і дві морфи перетягаються разом: кульку *вбудовано* всередину прямокутника. У *joe* можна вбудувати більше морфів. Це можна зробити і програмно, і вручну.

16.5. Створення і малювання власних морф

Хоча за допомогою композиції можна створити багато цікавих і корисних графічних представлень, іноді виникає потреба створити щось цілковито інше.

З цією метою визначають підклас класу *Morph* і перевантажують метод *drawOn:*, щоб змінити спосіб відображення морфи.

Середовище Morphic надсилає повідомлення *drawOn:* морфі, коли потрібно перемалювати її на екрані. Параметром повідомлення є різновид *Canvas*, очікувана поведінка – морф намалює себе на цьому полотні у його межах. Давайте використаємо ці знання, щоб створити хрестоподібну морфу.

Рис. 16.7. Екземпляр *CrossMorph* з меню-ореолом. Жовтим маніпулятором можна змінити його розмір

За допомогою Системного оглядача визначимо новий клас *CrossMorph*, що наслідує *Morph*.

```
Morph subclass: #CrossMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Метод *drawOn:* можна визначити як записано нижче.

```
CrossMorph >> drawOn: aCanvas
   | crossHeight crossWidth horizontalBar verticalBar |
   crossHeight := self height / 3.
   crossWidth := self width / 3.
   horizontalBar := self bounds insetBy: 0 @ crossHeight.
   verticalBar := self bounds insetBy: crossWidth @ 0.
   aCanvas fillRectangle: horizontalBar color: self color.
   aCanvas fillRectangle: verticalBar color: self color
```

У відповідь на повідомлення *bounds* морф повертає обмежувальний прямокутник, який є екземпляром класу *Rectangle*. Прямокутники розуміють велику кількість повідомлень для створення інших прямокутників схожих розмірів. Тут використано повідомлення *insetBy:* з аргументом екземпляром *Point*, щоб створити спочатку прямокутник зі зменшеною висотою, а потім – прямокутник зі зменшеною шириною, причому виміри змінюються симетрично з обох сторін, змінюючи розташування фігури.

Щоб протестувати новий морф, виконайте *CrossMorph new openInWorld*.

Результат мав би бути схожим на зображений на рис. 16.7. Це те, що й планували, проте можна зауважити невелику помилку, якщо спробувати захопити морфу мишкою: чутливою зоною є весь обмежувальний прямокутник, включно з незафарбованими ділянками, а не лише хрест. Давайте це виправимо.

Коли середовище намагається визначити, які морфи розташовані під курсором, воно надсилає повідомлення *containsPoint:* до усіх морфів, чиї обмежувальні прямокутники перебувають під вказівником мишки. Тому, щоб зменшити чутливу зону нашої морфи до самого лише хреста, потрібно переписати метод *containsPoint:*. Визначимо його в класі *CrossMorph*.

```
CrossMorph >> containsPoint: aPoint
  | crossHeight crossWidth horizontalBar verticalBar |
  crossHeight := self height / 3.
  crossWidth := self width / 3.
  horizontalBar := self bounds insetBy: 0 @ crossHeight.
  verticalBar := self bounds insetBy: crossWidth @ 0.
  ^ (horizontalBar containsPoint: aPoint) or:
     [ verticalBar containsPoint: aPoint]
```

Цей метод використовує таку саму логіку як *drawOn:*, тому можна бути впевненим, що *containsPoint:* повертає *true* тільки для тих точок, які замальовує *drawOn:*. Зверніть увагу на те, як використали метод *containsPoint:* класу *Rectangle*, щоб зробити важку роботу.

У написаних методах приховано дві проблеми.

Найочевиднішим є дублювання коду. Це принципова помилка: якщо виявиться, що потрібно змінити спосіб обчислення *horizontalBar* або *verticalBar*, то легко можна забути змінити один з двох випадків його використання. Розв'язок проблеми – винести ці обчислення у два окремі методи у протоколі *private*.

```
CrossMorph >> horizontalBar
  | crossHeight |
  crossHeight := self height / 3.
  ^ self bounds insetBy: 0 @ crossHeight
```

```
  CrossMorph >> verticalBar
  | crossWidth |
  crossWidth := self width / 3.
  ^ self bounds insetBy: crossWidth @ 0
```

Тоді можна визначити *drawOn:* і *containsPoint:*, що використовують ці методи.

```
CrossMorph >> drawOn: aCanvas
  aCanvas fillRectangle: self horizontalBar color: self color.
  aCanvas fillRectangle: self verticalBar color: self color
```

```
CrossMorph >> containsPoint: aPoint
  ^ (self horizontalBar containsPoint: aPoint) or: [
     self verticalBar containsPoint: aPoint ]
```

Такий код набагато зрозуміліший, значною мірою завдяки осмисленим іменам приватних методів. Вони такі прості, що легко помітити другу проблему: ділянка в центрі хреста, яка належить обом прямокутникам, замальовується двічі. Це не важливо, коли колір непрозорий, але ця вада стає помітною, коли для заповнення використовують напівпрозорий колір (рис. 16.8.)

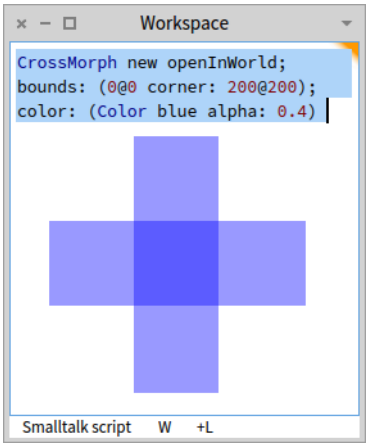

Рис. 16.8. Середину хреста замальовано двічі

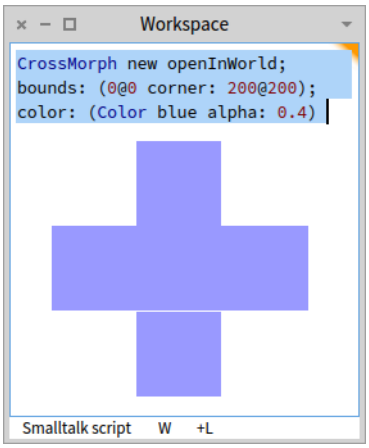

Рис. 16.9. Хрестоподібна морфа містить незамальовану лінію

Щоб переконатися, виконайте такий код у Робочому вікні.

```
CrossMorph new openInWorld;
   bounds: (0@0 corner: 200@200);
   color: (Color blue alpha: 0.4)
```

Виправити помилку можна поділом вертикального прямокутника на три частини з наступним замальовуванням лише двох з них: верхньої та нижньої. І знову можна знайти в класі *Rectangle* метод, який виконає складну роботу: *rect1 areasOutside: rect2* повертає масив прямокутників, які частинами *rect1* поза межами *rect2*. Нижче наведено виправлений код.

```
CrossMorph >> drawOn: aCanvas
     | topAndBottom |
     aCanvas fillRectangle: self horizontalBar color: self color.
     topAndBottom := self verticalBar areasOutside: self horizontalBar.
     topAndBottom do: [ :each |
        aCanvas fillRectangle: each color: self color ]
```

Цей код виглядає робочим, але якщо випробувати його на різних екземплярах, розтягаючи їх, то можна зауважити, що для окремих розмірів на хресті з'являється біла лінія товщиною в цятку (рис. 16.9). Так відбувається через похибки заокруглення, коли розмір прямокутника для замальовування не ціле число, метод *fillRectangle:color:* заокруглює його по-різному, залишаючи один рядок цяток незамальованим. Помилку можна виправити, виконавши заокруглення обчислених розмірів у методах побудови вертикального і горизонтального прямокутників, як подано нижче.

```
CrossMorph >> horizontalBar
  | crossHeight |
  crossHeight := (self height / 3) rounded.
  ^ self bounds insetBy: 0 @ crossHeight
```

```
CrossMorph >> verticalBar
| crossWidth |
crossWidth := (self width / 3) rounded.
^ self bounds insetBy: crossWidth @ 0
```

16.6. Взаємодія через події мишки

Щоб побудувати з морфів живий користувацький інтерфейс, потрібно вміти взаємодіяти з ними за допомогою мишки та клавіатури. Ба більше, морфи мусять уміти реагувати на дії користувача зміною свого вигляду та місця розташування, виконаного за допомогою анімації.

Одразу після натискання кнопки мишки Morphic надсилає кожній морфі, розташованій під вказівником мишки, повідомлення *handlesMouseDown:*. Якщо вона відповідає *true*, то Morphic невідкладно надсилає їй повідомлення *mouseDown:*. Середовище також надішле повідомлення *mouseUp:*, коли користувач відпустить кнопку. Якщо всі морфи відповіли *false*, то Morphic ініціює операцію перетягування (drag-and-drop). Щоб переконатися в цьому, відобразіть на екрані екземпляр *CrossMorph* і клацніть на ньому мишкою – хрестик причепиться до курсора мишки і мандруватиме за ним до наступного клацання.

Згодом побачимо, що повідомлення *mouseDown:* і *mouseUp:* надсилають з аргументом, екземпляром *MouseEvent*, який містить опис події мишки.

Давайте розширимо клас *CrossMorph* так, щоб він зміг обробляти події мишки. Спочатку впевнимося, що усі екземпляри *CrossMorph* відповідають *true* на повідомлення *handlesMouseDown:*. Для цього оголосимо в *CrossMorph* відповідний метод.

```
CrossMorph >> handlesMouseDown: anEvent
  ^ true
```

Припустимо, що під час натискання лівої кнопки мишки потрібно змінити колір хреста на червоний, а після натискання правої – на жовтий. Це можна реалізувати методом *mouseDown:* як показано нижче.

```
CrossMorph >> mouseDown: anEvent
  anEvent redButtonPressed
     ifTrue: [ self color: Color red ]. "клацання мишкою"
  anEvent yellowButtonPressed
     ifTrue: [ self color: Color yellow ]. "контекстне клацання"
  self changed
```

Важливо, що, крім зміни кольору морфи, цей метод також надсилає повідомлення «*self changed*». Завдяки цьому середовище своєчасно надішле морфі повідомлення *drawOn:*.

Потрібно також зауважити, що як тільки морфа опрацьовує події мишки, її вже не вдасться перетягати, як раніше. Натомість потрібно відкрити меню-ореол морфи і перетягнути її за чорний маніпулятор, розташований над нею. Скористайтеся нагодою і змініть колір морфи за допомогою бузкового маніпулятора, розташованого праворуч. Далі можна експериментувати з клацанням на морфі.

Аргумент *anEvent* повідомлення *mouseDown:* є екземпляром класу *MouseEvent*, підкласу *MorphicEvent*. *MouseEvent* визначає методи *redButtonPressed* і *yellowButtonPressed*. Перегляньте цей клас і знайдіть інші методи для опитування події мишки.

16.7. Події клавіатури

Для перехоплення подій клавіатури потрібно зробити три кроки.

1. Передати *фокус клавіатури* певній морфі. Наприклад, можна передати фокус морфі, коли вказівник мишки розташований над нею.
2. Обробити подію клавіатури за допомогою методу *keyDown:*. Середовище надсилає відповідне повідомлення морфі, що має фокус клавіатури, коли користувач натискає клавішу.
3. Зняти фокус клавіатури з морфи, коли курсор більше не розташований над нею.

Давайте розширимо *CrossMorph* так, щоб він реагував на натискання клавіш. Спочатку потрібно організувати інформування морфи про те, що курсор мишки перебуває над нею. Середовище надсилає відповідні повідомлення морфі, яка відповідає *true* на повідомлення *handlesMouseOver:*, тому потрібно оголосити метод, наведений нижче.

```
CrossMorph >> handlesMouseOver: anEvent
  ^ true
```

Це повідомлення подібне до *handlesMouseDown:* для позиції мишки. Коли вказівник мишки заходить на морфу чи покидає її, то морфі надсилаються повідомлення *mouseEnter:* і *mouseLeave:*, відповідно.

Визначимо два методи так, щоб *CrossMorph* захоплював і звільняв фокус клавіатури, третій метод задавав інформування про натискання, а четвертий – обробляв натискання клавіш.

```
CrossMorph >> mouseEnter: anEvent
  anEvent hand newKeyboardFocus: self

CrossMorph >> mouseLeave: anEvent
  anEvent hand releaseKeyboardFocus: self

CrossMorph >> handlesKeyDown: anEvent
  ^ true

CrossMorph >> keyDown: anEvent
  | key |
  key := anEvent key.
  key = KeyboardKey up ifTrue: [ self position: self position - (0 @ 10) ].
  key = KeyboardKey down ifTrue:[self position: self position + (0 @ 10) ].
  key = KeyboardKey right ifTrue:[self position: self position + (10 @ 0)].
  key = KeyboardKey left ifTrue:[self position: self position - (10 @ 0) ]
```

Метод написано так, щоб можна було пересувати морфу за допомогою клавіш зі стрілками. Зауважте, що коли мишка не розташована над морфою, то повідомлення *keyDown:* не надсилається, і морфа не реагує на стрілки. Щоб побачити значення натиснутих клавіш, додайте вираз «*Transcript show: anEvent keyValue*.» до методу *keyDown:* і відкрийте вікно *Transcript*. Тепер спробуйте керувати хрестиком клавішами і спостерігайте за виведенням у консоль.

Аргумент *anEvent* методу *keyDown:* є екземпляром класу *KeyboardEvent*, підкласу *MorphicEvent*. Перегляньте *KeyboardEvent*, щоб більше дізнатися про події клавіатури.

Якщо хочете переміщувати морфу комбінаціями клавіш вигляду [*Ctrl+Key*], то в методі *keyDown:* можна використати розпізнавання, як описано нижче.

```
anEvent controlKeyPressed ifTrue: [
   anEvent keyCharacter == $d ifTrue: [
      self position: self position + (0 @ 10) ] ]
```

Від перекладача. Експерименти з переміщенням *CrossMorph* засвідчують, що перемальовування фігур неправильної форми не найсильніша сторона Morphic. У наведеному вище методі *keyDown:* крок переміщення задано невеликим – 10 цяток. Може виявитися, що під час переміщення хреста клавішами на екрані залишатимуться зафарбовані ділянки в попередньому розташуванні фігури: середовище не завжди витирає прозорі частини морфи. Проблем не виникає, якщо переміщати морфу з розгорнутим меню-ореолом. Можна також збільшити крок переміщення: хоча б 27 по вертикалі і 33 по горизонталі (ці величини пов'язані з розмірами морфи за замовчуванням).

Не зайвим буде додати, що Morphic підтримує три різні події клавіатури: *keystroke*, *keydown* і *keyup*. У цьому параграфі опрацьовували подію *keydown*, а в наступному буде використано *keystroke*.

16.8. Анімація морф

Morphic надає просту систему анімації з двома основними повідомленнями: *step* надсилається морфі через постійні проміжки часу, а *stepTime* визначає інтервал у мілісекундах між двома такими надсиланнями. Саме *stepTime* визначає *мінімальний* час між двома *step*. Якщо ви захочете, щоб *stepTime* задавав одну мілісекунду, то не дивуйтеся, що Pharo буде занадто зайнятий виконанням кроків анімації вашої морфи аж так часто. Додамо, що метод *startStepping* вмикає механізм анімації, а *stopStepping* вимикає його. Щоб довідатися, чи виконується анімація морфи, можна запитати її *isStepping*.

Навчимо *CrossMorph* миготіти, визначивши методи, як зображено нижче.

```
CrossMorph >> stepTime
  ^ 100

CrossMorph >> step
  (self color diff: Color black) < 0.1
     ifTrue: [ self color: Color red ]
     ifFalse: [ self color: self color darker ]
```

Щоб побачити анімацію в дії, відкрийте інспектор морфи маніпулятором налагодження меню-ореолу (маніпулятор лавандового кольору з гайковим ключем), у панелі редактора коду введіть вираз «*self startStepping*» і виконайте його командою «*Do it*». Так само можна зупинити анімацію: «*self stopStepping*».

Зауваження. Якщо створення морфи виконувати кнопкою ***Do it*** у вікні Playground, то вікно інспектора відкриється автоматично.

Для керування анімацією можна використати клавіатуру, наприклад, [+] та [–] для вмикання і вимикання, відповідно.

Зазвичай для опрацювання тексту використовують подію *keystroke*, а для опрацювання гарячих клавіш – *keydown* і *keyup*. Про *keydown* йшлося в попередньому параграфі, тому

використаємо опрацювання *keystroke*. Потрібно оголосити два методи: перший вмикає інформування про виникнення події, а другий – опрацьовує її.

```
CrossMorph >> handlesKeyStroke: anEvent
   ^ true

CrossMorph >> keyStroke: anEvent
   | keyValue |
   keyValue := anEvent keyCharacter.
   keyValue == $+ ifTrue: [ self startStepping ].
   keyValue == $- ifTrue: [ self stopStepping ]
```

Зауважимо, що подія *keystroke* морфи стається тільки тоді, коли вона має фокус уведення клавіатури.

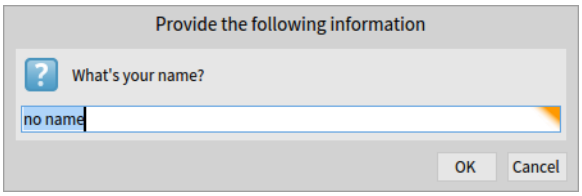

Рис. 16.10. Уведення рядка

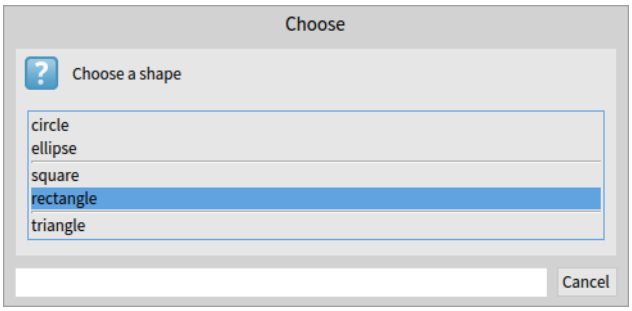

Рис. 16.11. Діалог-меню

16.9. Діалоги

Клас *UIManager* підтримує велику кількість готових панелей діалогу, щоб можна було попросити користувача ввести певні дані. Наприклад, метод *request:initialAnswer:* повертає введений користувачем рядок (див. рис. 16.10). Відкрити діалог досить просто.

```
UIManager default
   request: 'What''s your name?'
   initialAnswer: 'no name'
```

Щоб відкрити діалог, схожий на виринаюче меню, використовують різні варіанти методу *chooseFrom:* (див. рис. 16.11).

```
UIManager default
   chooseFrom: #('circle' 'ellipse' 'square' 'rectangle' 'triangle')
   lines: #(2 4)
   message: 'Choose a shape'
```

Перегляньте клас *UIManager* і випробуйте інші його методи для взаємодії з користувачем. Використайте в діалогах українську. Перевірте, яку відповідь поверне діалог, якщо користувач натисне кнопку ***Cancel***.

16.10. Перетягування

Середовище Morphic підтримує взаємодію морф за допомогою операції перетягування. Дослідимо простий приклад взаємодії двох морф: морфи-приймача і переміщеної морфи. Переміщену морфу тягнуть мишкою і скидають на приймач, приймач може

перевірити, чи задовольняє скинутий об'єкт задану умову, і або прийняти його, або відхилити. Припустимо, що скинута морфа має бути синього кольору. Відхилена морфа сама вирішує, що зробити далі.

Спочатку визначимо морфу-отримувач.

```
Morph subclass: #ReceiverMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Звичайно визначимо її метод ініціалізації.

```
ReceiverMorph >> initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 100 @ 100
```

Хто має вирішити, чи приймач схвалить, чи відхилить скинуту морфу? В загальному випадку, обидві морфи мають бути готовими взаємодіяти. Приймач робить це, відповідаючи на повідомлення *wantsDroppedMorph:event:*. Його перший аргумент – скинута морфа, а другий – подія мишки. Так приймач може, наприклад, побачити, чи натискали якісь командні клавіші в момент скидання. Скинута морфа також має шанс перевірити і подивитися, чи її влаштовує морфа, на яку її скидають, відповідаючи на повідомлення *wantsToBeDroppedInto:*. Реалізація за замовчуванням цього методу (у класі *Morph*) повертає *true*.

```
ReceiverMorph >> wantsDroppedMorph: aMorph event: anEvent
  ^ aMorph color = Color blue
```

Що відбудеться зі скинутою морфою, якщо приймач не хоче її прийняти? Поведінка за замовчуванням – не робити нічого, тобто залишатися поверх нього, але не взаємодіяти з ним. Інтуїтивно зрозумілішою поведінкою було б повернутися на початкову позицію. Цього можна досягти, отримавши від приймача ствердну відповідь на повідомлення *repelsMorph:event:*, коли він не хоче приймати скинуту морфу.

```
ReceiverMorph >> repelsMorph: aMorph event: anEvent
  ^ (self wantsDroppedMorph: aMorph event: anEvent) not
```

Це все, що потрібно зробити в класі приймача.

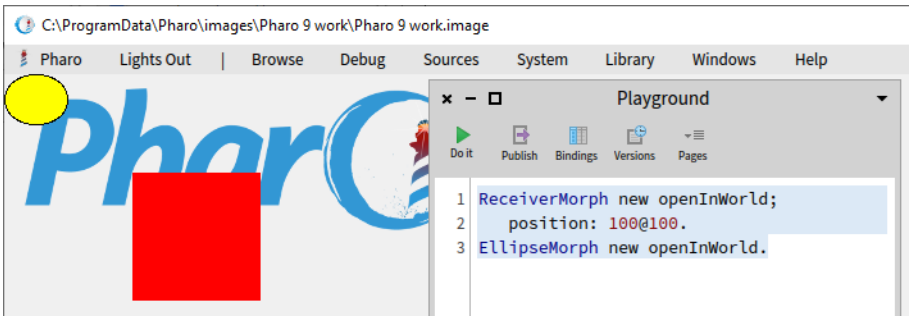

Рис. 16.12. Екземпляри класів *ReceiverMorph* та *EllipseMorph*

Створіть у Робочому вікні екземпляри *ReceiverMorph* та *EllipseMorph* (див. рис. 16.12).

```
ReceiverMorph new openInWorld;
   position: 100@100.
EllipseMorph new openInWorld.
```

Спробуйте перетягнути та скинути жовтий *EllipseMorph* на приймача. Приймач його відхилить і еліпс повернеться на свою початкову позицію.

Щоб побачити іншу поведінку, змініть колір еліпса на синій (надішліть йому повідомлення «*color: Color blue;*» одразу після *new*, перед *openInWorld*). Перетягніть синю морфу на червоний квадрат і відпустіть – скинута морфа стане частиною приймача. Можете створити кілька еліпсів і перетягнути усі на квадрат.

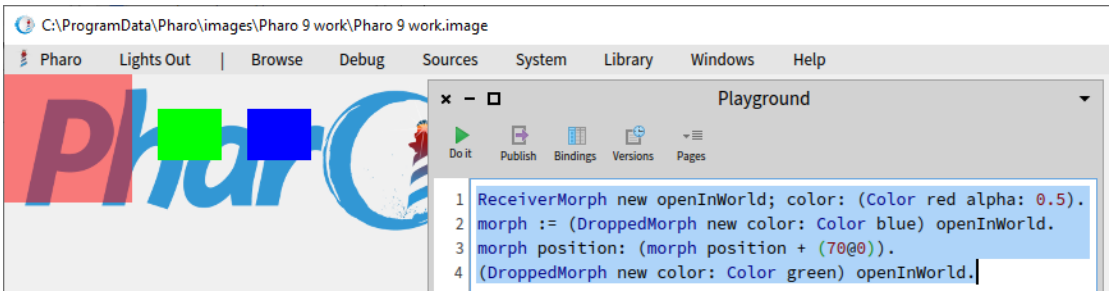

Рис. 16.13. Створення *DroppedMorph* і *ReceiverMorph*

Щоб продовжити експерименти, створимо підклас класу *Morph* і назвемо його *DroppedMorph*.

```
Morph subclass: #DroppedMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'

DroppedMorph >> initialize
   super initialize.
   color := Color blue.
   self position: 120 @ 50
```

Тепер можна задати, як поводитиметься скинута морфа, коли приймач відкине її. В описаному нижче випадку вона залишатиметься прикріпленою до вказівника мишки.

```
DroppedMorph >> rejectDropMorphEvent: anEvent
   | h |
   h := anEvent hand.
   WorldState addDeferredUIMessage: [ h grabMorph: self ].
   anEvent wasHandled: true
```

Тут об'єкт *anEvent* – подія мишки, у відповідь на повідомлення *hand* вона повертає «руку», екземпляр класу *HandMorph*, який представляє вказівник мишки та все, що він тримає. Тут сказано об'єктові *World*, що рука має захопити *self* – відкинуту морфу.

Створіть два екземпляри класу *DroppedMorph* різних кольорів (рис. 16.13), перетягніть і скиньте їх на приймача.

```
ReceiverMorph new openInWorld; color: (Color red alpha: 0.5).
morph := (DroppedMorph new color: Color blue) openInWorld.
morph position: (morph position + (70@0)).
```

```
(DroppedMorph new color: Color green) openInWorld.
```

Приймач відкине зелену морфу, і вона залишиться прикріпленою до вказівника мишки.

16.11. Завершений приклад

Давайте розробимо морфу, яка відображає і обертає ігральну кісточку. Клацання на ній запускатиме почергове відображення різних граней, а наступне клацання – зупинятиме анімацію.

Клас кісточки наслідуємо від *BorderedMorph* замість *Morph*, бо використовуватимемо її краї.

```
BorderedMorph subclass: #DieMorph
   instanceVariableNames: 'faces dieValue isStopped'
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Змінна екземпляра *faces* міститиме кількість граней ігральної кісточки. Найбільша кількість граней буде дев'ять! Змінна *dieValue* записуватиме значення видимої в цей момент грані, а *isStopped* відповідатиме за стан анімації: значення *true* означає, що анімацію зупинено. Щоб створити екземпляр ігральної кісточки, визначимо *метод класу*: *DieMorph class >> faces: n* створює кісточку з *n* гранями.

```
DieMorph class >> faces: aNumber
  ^ self new faces: aNumber
```

Визначимо метод *initialize* на стороні об'єкта звичним способом. Пам'ятаємо, що *new* автоматично надсилає повідомлення *initialize* новоствореному об'єкту.

```
DieMorph >> initialize
  super initialize.
  self extent: 50 @ 50.
  self
     useGradientFill;
     borderWidth: 2;
     useRoundedCorners.
  self setBorderStyle: #complexRaised.
  self fillStyle direction: self extent.
  self color: Color green.
  dieValue := 1.
  faces := 6.
  isStopped := false
```

Щоб надати кісточці гарного вигляду, використано кілька методів *BorderedMorph*: задано товстий опуклий край, заокруглені кути і градієнт кольору на видимій грані. Метод екземпляра *faces:* визначено так, щоб перевіряти правильність параметра.

```
DieMorph >> faces: aNumber
  "Задає кількість граней"
  (aNumber isInteger and: [ aNumber > 0 and: [ aNumber <= 9 ] ])
     ifTrue: [ faces := aNumber ]
```

Було б добре переглянути порядок надсилання повідомлень під час створення кісточки. Наприклад, під час виконання «*DieMorph faces: 9*».

- Метод класу *DieMorph class >> faces:* надсилає повідомлення *new* метакласові *DieMorph class*.
- Метод *new* (успадкований *DieMorph class* від *Behavior*) створює новий об'єкт і надсилає йому повідомлення *initialize*.
- Метод *initialize* у *DieMorph* задає *faces* початкове значення 6.
- *DieMorph class >> new* повертає керування методу класу *DieMorph class >> faces:*, який надсилає повідомлення *faces: 9* новоствореному екземпляру.
- Виконується метод екземпляра *DieMorph>>faces:* і задає змінній *faces* значення 9.

Перш ніж визначати *drawOn:*, потрібно оголосити кілька приватних методів, які повідомляють розташування крапок на видимій грані.

```
DieMorph >> face1
  ^ {(0.5 @ 0.5)}

DieMorph >> face2
  ^{0.25@0.25 . 0.75@0.75}

DieMorph >> face3
  ^{0.25@0.25 . 0.75@0.75 . 0.5@0.5}

DieMorph >> face4
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}

DieMorph >> face5
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}

DieMorph >> face6
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}

DieMorph >> face7
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5}

DieMorph >> face8
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5 . 0.5@0.25}

DieMorph >> face9
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5 . 0.5@0.25 . 0.5@0.75}
```

Ці методи визначають колекції координат для кожної грані. Координати розраховано для квадрата розміром 1×1, достатньо їх масштабувати, щоб розташувати крапки на грані справжнього розміру.

Метод *drawOn:* виконує дві дії: спочатку малює фон грані за допомогою звертання до *super*, а потім малює крапки.

```
DieMorph >> drawOn: aCanvas
   super drawOn: aCanvas.
   (self perform: ('face', dieValue asString) asSymbol)
      do: [ :aPoint | self drawDotOn: aCanvas at: aPoint ]
```

Друга частина цього методу використовує можливості рефлексії Pharo. Малювання крапок на грані – це простий перебір колекції, наданої методом *faceX* для цієї грані, задля надсилання повідомлення *drawDotOn:at:* з кожною координатою. Щоб викликати правильний метод *faceX*, сконструйовано відповідне повідомлення виразом «*('face', dieValue asString) asSymbol*» і використано метод *perform:*, щоб надіслати його.

```
DieMorph >> drawDotOn: aCanvas at: aPoint
   aCanvas
      fillOval: (Rectangle
         center: self position + (self extent * aPoint)
         extent: self extent / 6)
      color: Color black
```

Оскільки координати нормовані до інтервалу [0; 1], то їх масштабують відповідно до розміру грані: *self extent * aPoint*. Тепер можна створити в Робочому вікні екземпляр кісточки (рис. 16.14).

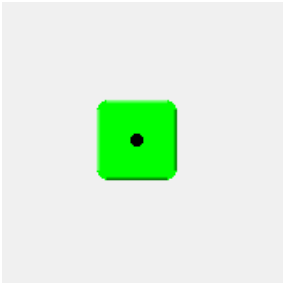

Рис. 16.14. Нова кісточка створена виразом *(DieMorph faces: 6) openInWorld*

Рис. 16.15. Результат виконання *(DieMorph faces: 6) openInWorld; dieValue: 5* і меню-ореол

Щоб мати змогу змінювати видиму грань, створимо відповідний метод доступу. Його можна буде використовувати як ключове повідомлення *myDie dieValue: 5*.

```
DieMorph >> dieValue: aNumber
   (aNumber isInteger and: [ aNumber > 0 and: [ aNumber <= faces ] ])
      ifTrue: [
         dieValue := aNumber.
         self changed ]
```

Щоб швидко змінювати грані, використаємо анімацію.

```
DieMorph >> stepTime
   ^ 100

DieMorph >> step
   isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

Створіть *DieMorph* і побачите, що кісточка обертається! Принаймні, змінює кількість крапок на видимій грані.

Навчимо кісточку запускати або зупиняти анімацію клацанням мишки. Використаємо для цього здобуті знання про події мишки. Спочатку активуємо повідомлення про них.

```
DieMorph >> handlesMouseDown: anEvent
  ^ true
```

Потім оголосимо метод опрацювання клацання: він альтернативно вмикає-вимикає анімацію.

```
DieMorph >> mouseDown: anEvent
  anEvent redButtonPressed ifTrue: [isStopped := isStopped not]
```

Тепер кісточка починатиме або переставатиме обертатися після кожного клацання на ній.

16.12. Більше про полотно малювання

Єдиний аргумент методу *drawOn:* – екземпляр класу *Canvas*, полотно. Це ділянка, на якій морфа відображає себе. За допомогою графічних методів полотна можна створювати такий вигляд морфи, якого забажаєте. Якщо переглянути ієрархію класу *Canvas*, то легко бачити, що він має кілька підкласів. Підкласом за замовчуванням є *FormCanvas*, і більшість ключових графічних методів міститься у *Canvas* та *FormCanvas*. Ці методи можуть малювати точки, лінії, ламані, прямокутники, еліпси, тексти та зображення, а також повертати їх та масштабувати.

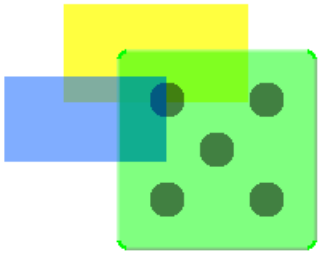

Рис. 16.16. Напівпрозоре відображення грального кісточки

Також можливо використовувати інші види полотна, наприклад, щоб отримати прозорі морфи, використати більше графічних методів, згладжування тощо. Щоб використати ці засоби, потрібний буде *AlphaBlendingCanvas* або *BalloonCanvas*. Але як можна одержати таке полотно у методі *drawOn:*, якщо він отримує своїм аргументом екземпляр класу *FormCanvas*? На щастя, можна перетворити полотно одного типу в інший.

Щоб використати у *DieMorph* полотно з коефіцієнтом прозорості 0.5, перевизначимо *drawOn:*, як показано нижче.

```
DieMorph >> drawOn: aCanvas
| theCanvas |
theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
super drawOn: theCanvas.
(self perform: ('face', dieValue asString) asSymbol)
   do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

Це все, що потрібно зробити! Змінене зображення кісточки видно на рис. 16.16.

16.13. Підсумки до розділу

Morphic – це графічне середовище, в якому можна динамічно компонувати графічні елементи інтерфейсу.

- Будь-який об’єкт можна перетворити на морфу і відобразити його на екрані за допомогою повідомлень *asMorph openInWorld*.
- Морфою можна керувати за допомогою маніпуляторів меню-ореола, яке відкривають метаклацанням на морфі. (Маніпулятори мають виринаючі підказки, які пояснюють їхнє призначення).
- Морфи можна компонувати, вставляючи їх одна в одну перетягуванням або за допомогою повідомлення *addMorph:*.
- Клас морфи можна наслідувати та перевизначити ключові методи, такі як *initialize* та *drawOn:*.
- Можна контролювати взаємодію морфи з мишкою та клавіатурою, перевизначивши такі методи, як *handlesMouseDown:*, *handlesMouseOver:* тощо.
- Морфу можна анімувати, перевизначивши методи *step* (що робити) та *stepTime* (кількість мілісекунд між кадрами).