Автоматизация отмены и повтора пользовательских действий в системах интерактивной графики

Дерябин Н.Б., Хлупина А. А., ИПМ им. М.В. Келдыша РАН
dek@keldysh.ru

Аннотация

Рассматриваются подход к реализации отмены и повтора действий пользователя, реализованный авторами в ряде систем интерактивной графики и оптического моделирования. Особенностью подхода является то, что от разработчика содержательного контента не требуется добавления специальных объектов и / или операций – поддержка повтора / отмены пользовательских действий при добавлении нового контента выполняется инфраструктурой системы автоматически.

1 Введение

В процессе работы пользователь интерактивной графической системы постоянно изменяет те или иные параметры моделируемой сцены. Дружественный графический интерфейс пользователя должен предоставлять возможность отмены последних сделанных изменений, а при необходимости и повторного применения этих изменений. Без аппарата отмены / повтора пользовательских действий современная графическая система просто не дееспособна.

В настоящее время принято рассматривать два основных способа реализации отмены / повтора (undo / redo) действий пользователя

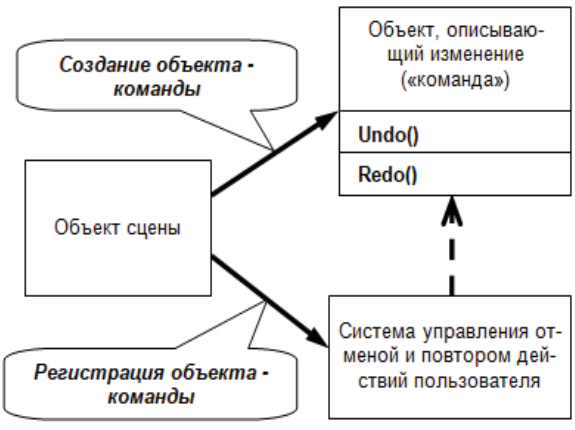

Рис. 1. Традиционная реализация отмены / повтора операций.

[Фасман, 2016]. Первый из них основан на создании специальных объектов-команд для каждого действия пользователя (шаблон программирования «команд»). Команды имеют специальные методы для отмены (`Undo()`) и повтора (`Redo()`) действия (Рис. 1). Система управления отменой и повтором помещает команды в undo/redo стек и вызывает в них соответствующие методы для выполнения откатов и повторов.

Именно этот подход обычно используется в современных графических системах. В качестве примера можно привести продукты компании Autodesk – 3D SUDIO MAX, Maya и др. Данный метод считается более гибким, но при его использовании приходится для каждого нетривиального действия писать новый производный класс команды с двумя специальными функциями, то есть автоматизация в принципе исключается. Недостаток подхода – при ошибочной реализации данных функций логика повтора / отмены ломается. Кроме того, этот метод вступает в противоречие с возможностью создания в приложении новых пользовательских объектов на языке сценариев, где важна простота, минимум кода (нет места специальным функциям).

Второй способ, на практике чаще используемый в относительно простых приложениях типа текстовых редакторов, основан на протоколировании изменений. В применении к графическим системам, состояние сцены определяется совокупностью свойств составляющих ее объектов, поэтому для сохранения истории изменений можно просто хранить историю изменения свойств объектов сцены. Для отмены действия просто восстанавливаются последние измененные свойства.

Преимуществом этого подхода является его простота и потенциальная возможность надежной автоматической поддержки отмены / повтора пользовательских действий. Препятствием к его использованию в графических системах является сложность реализации «скрытых» зависимостей. При выполнении действий пользователя, помимо изменения основных свойств объектов сцены, выполняется изменение (создание, удаление)

других объектов приложения, например, специального представления сцены для целей оптического моделирования. При отмене / повторе действий необходимо каким-то образом приводить эти «другие» объекты к корректному состоянию. Решить эту непростую задачу, по нашему мнению, можно только построением соответствующей инфраструктуры графической системы.

При создании инфраструктуры для наших графических комплексов, предназначенных для моделирования оптических процессов (смотрите, например, [Жданов, 2011]), мы – среди прочего – изначально поставили задачу добиться автоматического выполнения отмены и повтора пользовательских действий, то есть освободить разработчиков графической системы от написания для этого какого-либо дополнительного кода. Задача оказалась не простой, но все же была успешно решена. При этом за основу был взят второй из описанных подходов с соответствующей инфраструктурной поддержкой. Первый способ тоже используется, но в ограниченном контексте.

Для упрощения изложения в данной статье мы не будем углубляться в архитектуру специальных возможностей наших комплексов компьютерной графики и оптического моделирования (информацию по этому вопросу можно найти, например, в [Галактионов, 2009]), а ограничимся типовой проблемой визуализации сцены. Для визуализации используется специальное представление сцены в формате, удобном для графической системы (такой, как, OpenGL). Выполнение пользовательских действий, состоящее в изменении каких-либо свойств объектов сцены, влечет соответствующее изменение визуального представления сцены и, в конечном счете, модификацию изображения сцены в основном окне приложения. При отмене одного или нескольких действий производится восстановление предыдущих свойств объектов сцены, и мы покажем, как инфраструктура графической системы поддерживает соответствующее изменение визуального представления и соответствующую перерисовку сцены.

2 Целевое представление сцены

Предметной областью систем интерактивной графики является моделируемая сцена. Под **целевым представлением сцены** мы понимаем представление сцены, отображающее ее вид с точки зрения *пользователя*. В сильно упрощенном виде целевое представление сцены, используемое нами, показано на Рис. 2.

Мы исходим из типового иерархического представления сцены с использованием аппарата свойств и действий. Целевое представление сцены представляет собой иерархию **целевых объектов** (прямоугольники на рисунке), создаваемую **целевыми ссылками** (стрелки на рисунке). Каждый объект в целевом представлении, кроме корневого, имеет только одного предка, но может бать потомком нескольких объектов.

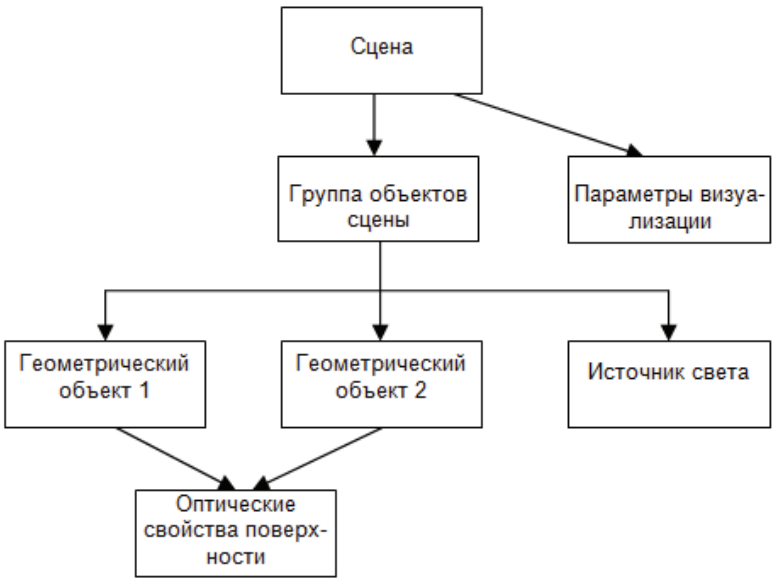

Рис. 2. Целевое представление сцены.

Все целевые объекты имеют общий **целевой интерфейс**, реализуемый базовым классом `Entity` (используется язык программирования C++). Целевой интерфейс предоставляет унифицированный набор операций, позволяющий работать с объектами в терминах **свойств** и **действий**.

2.1 Свойства

Примерами свойств являются связи между целевыми объектами в представлении сцены, реализуемые целевыми ссылками. Например, «Геометрический объект» (Рис. 2) имеет

свойство `surf_attrs`, значением которого является объект типа «Оптические свойства поверхности».

К свойствам относится также способ доступа к внутренним подобъектам (атрибутам) целевого объекта. Например, целевой объект «Оптические свойства поверхности» может иметь свойства «цвет диффузного рассеяния», «коэффициент спектрального отражения» и т.п.

Аппарат свойств реализован на базе аппарата ссылок. Ссылка – это «умный» указатель на объект (управляющий временем жизни объектов и выполняющий сборку мусора), который в нашей инфраструктуре выполняет ряд других важных функций, в частности – поддержку свойств. Внутреннее атрибуты целевых объектов также реализованы при помощи ссылок на подобъекты, хотя это и не показано на Рис. 2. При этом используются специальные объекты для представления значений элементарных типов, таких как скаляры и массивы. Эти объекты также поддерживают базовый интерфейс `Entity` и в этом смысле являются целевыми. К элементарным типам относятся целочисленные, вещественные, логические и строковые значения, а также вектора, массивы и матрицы целых или вещественных чисел.

Для простоты (но без ограничения общности) мы будем считать объекты элементарных типов неизменяемыми. Если требуется изменить, например, значение скаляра, создается новый элементарный объект, который присваивается ссылке как новое значение свойства.

Помимо «простых» свойств, имеющих одно значение, в целевом представлении сцены используются «списочные» свойства, которые могут иметь несколько значений одного базового типа. Например, «Группа объектов сцены» на Рис. 2 имеет свойство `children`, значением которого является список из трех объектов сцены (два геометрических объекта и источник света). Списочные свойства опираются на аппарат списочных ссылок, что по существу является списком однотипных ссылок.

Целевой интерфейс предоставляет специальный метод типа для определения свойств целевых объектов. Атрибутами свойства, задаваемыми при его определении, являются имя свойства, тип значений свойства, используемая для свойства ссылка и специальные флаги. Более подробную информацию об определении свойств с фрагментами C++ кода можно найти в [Дерябин, 2017].

Целевой интерфейс `Entity` предоставляет методы `GetProperty()` для извлечения значения свойства и `SetProperty()` для установки нового значения свойства целевого объекта Конкретное свойство может задаваться его именем. Перегруженные версии этих методов с добавочным параметром – номером элемента списка служат для извлечения и установки элемента списочного свойства. Для списочных свойств также имеются методы `InsertProperty()` для добавления и `ExcludeProperty()` для удаления элементов списочного свойства.

Важно, что состояние моделируемой сцены полностью определяется значением свойств составляющих ее целевых объектов и ничем другим. Так, при сохранении сцены в файле (сериализации) сохраняются только свойства целевых объектов, составляющих сцену. Как следствие, любое изменение состояния сцены может быть выполнено путем изменения свойств целевых объектов. Более того, наша инфраструктура требует от разработчиков графической системы использования для изменения свойств исключительно соответствующих методов унифицированного целевого интерфейса `Entity`. Это **правило изменения состояния** необходимо для автоматизации отмены и повтора пользовательских действий.

2.2 Действия

Действия – это более высокий уровень оперирования с целевыми объектами. Действие реализуется функцией (методом) класса целевого объекта. Действия могут иметь параметры и возвращать результат. Параметрами и результатами действия могут быть любые целевые объекты, в том числе объекты, представляющие элементарные типы данных.

Целевой интерфейс предоставляет специальный метод типа для определения действий. Атрибутами действия являются имя действия, типы параметров действия, тип результата (если действие возвращает результат) и функция (метод класса), исполняющая действие. Более подробную информацию об определении действий с фрагментами C++ кода можно найти в [Дерябин, 2017].

Для изменения состояния сцены, функции, реализующие действия, должны – в соответствии с правилом изменения состояния – ис-

пользовать метода изменения свойств целевого интерфейса `Entity`.

Отметим, что действие, выполняемое в одном целевом объекте, может изменять свойства не только в данном объекте, но и в других (как правило, подобъектах этого объекта в иерархии сцены), в том числе и путем исполнения других (вложенных) действий. В качестве примера приведем действие «Замена свойств поверхности», вызываемое в целевом объекте «Группа объектов сцены». У этого действия два параметра типа «оптические свойства поверхности» – те, которые надо заменить, и новые, которые нужно установить взамен. Реализация действия просматривает список прямых потомков данной группы объектов сцены (свойство `children`) и вызывает такое же действие (если оно есть) с теми же параметрами в них. Если потомок является группой объектов сцены, мы получаем рекурсивное выполнение того же действия. Потомок типа «Геометрический объект» должен сравнить значение своего свойства `surf_attrs` с первым параметром действия (сравнение указателей) и, при совпадении, установить новое значение этого свойства, заданное вторым параметром действия.

Одной из причин для применения действий служит поддержка корректного состояния сцены. Представим, к примеру, целевой объект, содержащий в качестве свойств три ортогональных единичных вектора. Прямое изменение любого из этих векторов как свойства приведет к нарушению (хотя бы временному) условия ортогональности. Логичнее реализовать действие, которое корректно переустановит все три вектора за раз. Параметрами такого действия могут быть, например, углы поворота системы координат.

Возможность прямой установки одного орта в приведенном примере остается. Для решения этой проблемы используются специальные флаги, устанавливаемые для свойств – `PUBLIC`, `READONLY`. Напрямую (даже для опроса значения) разрешается использовать только свойства с флагом `PUBLIC`. Свойства с флагом `READONLY` запрещается изменять прямым способом. В приведенном примере полезно установить оба флага – тогда можно извлекать значение любого орта напрямую методом `GetProperty()`, но устанавливать их можно только совместно используя соответствующее действие.

Для выполнения действий целевой интерфейс `Entity` предоставляет унифицированные методы `ExecFunc()` для действий, возвращающих результат, и `ExecProc()` для действий без результата. Конкретное действие задается его именем. Задаются также фактические параметры для выполнения действия.

2.3 Использование целевого интерфейса

Итак, целевое представление сцены состоит из иерархии целевых объектов, реализующих базовый целевой интерфейс `Entity`. Методы целевого интерфейса позволяют опрашивать и изменять состояние целевых объектов на основе аппарата свойств и действий. Конкретные классы целевых объектов и их типов наследуют базовый интерфейс `Entity` (`EntityType` для типов) и реализуют его, определяя конкретный набор свойств и действий конкретных целевых объектов.

В наших системах оптического моделирования целевой интерфейс используется непосредственно для реализации графического интерфейса пользователя. При этом мы используем специальную концепцию (шаблон программирования) управляющего элемента (Рис. 3).

Концепция управляющего элемента предельно проста, но она позволяет решить ряд архитектурных задач, например, независимость ядра графической системы от пользовательского интерфейса. Управляющий элемент (элемент графического интерфейса

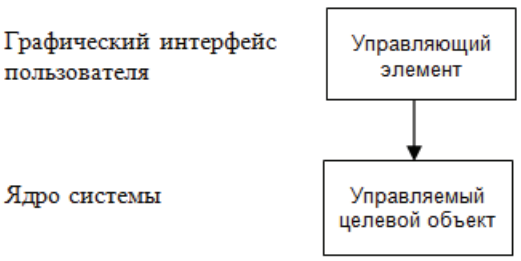

Рис. 3. Концепция управляющего элемента.

пользователя, виджет) имеет ссылку на управляемый элемент (целевой объект из иерархии сцены). Имея эту ссылку, он может показывать и изменять состояние управляемого целевого объекта, используя целевой интерфейс (то есть в терминах свойств и действий).

В данном случае установка свойства или исполнение действия в целевом объекте будет иметь немедленный эффект, который при необходимости должен быть отменен и повторен. В более сложном варианте, когда виджетом является панель с кнопками «Применить», «ОК», управляющий элемент создает копию управляемого объекта, и делает все изменения на этой копии. При нажатии кнопки «Применить» или «ОК» исходный объект приводится к состоянию копии. При этом отмене / повтору подлежит только суммарный эффект внесенных изменений.

Для создания копии объекта используются специальный метод `Clone()` целевого интерфейса, а для приведения объекта к состоянию модифицированного – специальное непубличное действие `Copy()`. Оба метода копируют все свойства соответствующего объекта, используя при этом целевой интерфейс.

Второй стороной использования целевого интерфейса являются языки сценариев, ставшие обязательной компонентой современных графических систем. Мы успешно внедрили язык сценариев (в данном случае, Python) в наши комплексы оптического моделирования, переложив при этом целевой интерфейс на язык сценариев непосредственно [Дерябин, 2017].

В связи с этим хотелось бы отметить важный методологический аспект проектирования целевого интерфейса (свойств, действий) конкретных объектов. В связи с использованием языков сценариев целевой интерфейс выходит на конечного пользователя. Поэтому целевой интерфейс должен разрабатываться не на основе, скажем, достаточности для реализации графического интерфейса, а с точки зрения конечного пользователя. Он должен быть, удобным, полным и непротиворечивым, понятным и естественным (дружественным) для *человека*.

3 Взаимодействие с другими объектами

Итак, любое изменение сцены делается через целевой интерфейс. Но пока мы видим только изменение состояния сцены, и остается открытым вопрос, как же при этом изменяются другие объекты, а именно (в нашей архитектуре), объекты специальных представлений сцены. Так как мы для простоты решили ограничиться проблемой визуализации, вопрос состоит в том, как происходит перерисовка сцены при ее изменении.

Ответом является использование аппарата уведомлений об изменениях, который мы сейчас и рассмотрим.

3.1 Обратимость ссылок

Аппарат уведомлений об изменениях в нашей инфраструктуре базируется на ссылках. Ссылки в нашей инфраструктуре имеют принципиальную особенность: они фактически являются двусторонними. Каждая ссылка содержит не только указатель на ссылаемый объект, то и обратный указатель на объект, содержащий эту ссылку, или, как мы говорим, объект, владеющий данной ссылкой (Рис. 4).

Имея указатель на объект, можно найти все ссылки на него, а по ним – их владельцев, Эту функциональность мы называем **обратимостью ссылок**.

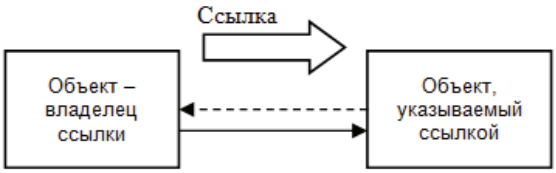

Рис. 4. Ссылка на объект.

Обратимость ссылок позволяет, например, найти все объекты типа «Геометрический объект» в сцене, использующие конкретный объект типа «Оптические свойства поверхности» (Рис. 2).

3.2 Уведомления об изменениях

Аппарат уведомлений об изменениях в нашей инфраструктуре базируется на ссылках и их свойстве обратимости. Объект `A` программного комплекса объявляет о своем изменении путем вызова метода `NotifyChange()`. Результатом является вызов виртуального метода `OnChange()` во всех объектах (B, C…), владеющих ссылками на объект `A` (Рис. 5).

Реализация метода `OnChange()` в объекте `B` анализирует причину изменения, выполняет соответствующие действия (если требуется), и своим результатом сообщает, завершена ли обработка изменения, или ее следует продолжить. В последнем случае уведомление об изменении распространяется дальше вверх по иерархии ссылок, то есть вызываются методы

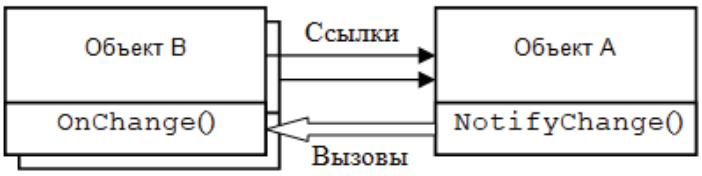

Рис. 5. Уведомление об изменении.

`OnChange()` в объектах, владеющих ссылками на объект `B`.

Такой механизм позволяет, например, распространять определенные уведомления от любого целевого объекта в иерархии сцены до корневого объекта сцены и дальше (Рис. 2). Как мы увидим в следующем разделе, такое распространение уведомлений об изменениях используется механизмом отмены и повтора действий пользователя. При этом система уведомлений распространяет уведомление только один раз даже при наличии нескольких путей распространения (это имеет место для объекта «Оптические свойства поверхности» на Рис. 2).

При реализации (переопределении) виртуального метода `OnChange()` в производном классе должно соблюдаться следующее **правило переопределения метода**: новая реализация, выполнив необходимую обработку, должна вызвать реализацию того же метода в базовом классе. Это позволяет базовым классам выполнить свою предопределенную обработку уведомлений. Касательно механизма отмены и повтора, именно базовая реализация метода `OnChange()` в интерфейсе `Entity` позаботится о распространении соответствующих уведомлений об изменениях вверх по иерархии сцены.

Аргументами вызова `NotifyChange()` являются причина уведомления[1] и любые дополнительные параметры, полезные для обработки уведомления. Аргументами вызова `OnChange()` являются причина уведомления, объект-источник уведомления, объект, от которого непосредственно пришло уведомление (не совпадает с источником в случае распространения уведомления), и дополнительные параметры уведомления, заданные при вызове `NotifyChange()`. Разумное использование этой информации позволяет упростить обработку уведомлений и сделать ее более гибкой.

В качестве примера использования механизма уведомлений об изменениях вернемся к концепции управляющего элемента (Рис. 3). Здесь управляющий элемент, используя ссылку на управляемый целевой объект, отслеживает изменения последнего, которые, возможно, вносятся третьей стороной. Если такие изменения происходят, состояние виджета соответствующим образом корректируется.

3.3 Перерисовка сцены при изменениях

Читатель уже должен догадаться, что для изменения «прочих» объектов, в частности, для визуализации, используется аппарат уведомлений об изменениях. Для этого в классы целевых объектов вводятся уведомления о производимых изменениях.

Концептуальная схема визуализации схемы представлена на Рис. 6.

Пусть пользователь, используя графический интерфейс, внес изменения в какой-то объект (объекты) целевого представления

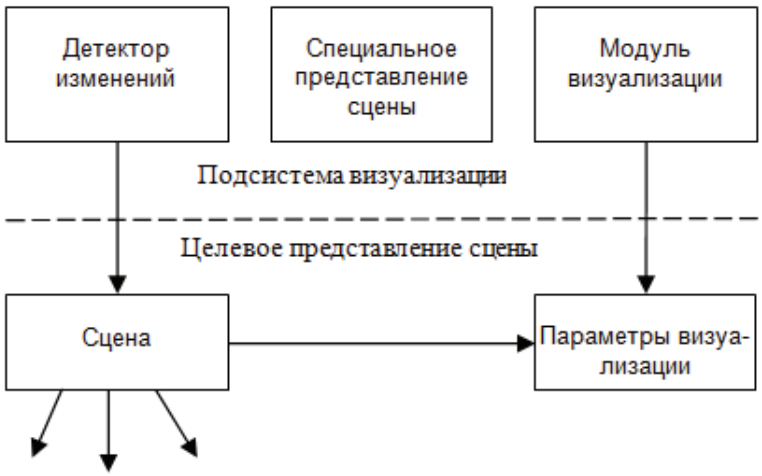

Рис. 6. Концептуальная схема визуализации сцены.

сцены. Для простоты мы постулируем, что эти изменения распространяются в корневой объект («Сцена») и далее. В конечном счете, они будут получены детектором изменений

[1] В целях структуризации причины уведомлений задаются конкретными классами в виде констант перечисления и поэтому передаются парой {класс, причина класса}

подсистемы визуализации через ссылку на корневой объект сцены. Детектор изменений обрабатывает эти изменения, в простейшем случае – просто запоминает факт изменения геометрий, свойств поверхностей и т.п.

По окончании акта пользователя по изменению сцены дается команда модулю визуализации на перерисовку сцены. Модуль визуализации обращается к детектору изменений для приведения специального представления сцены (используемого для визуализации) в новое состояние, соответствующее изменениям пользователя. При этом детектор изменений может делать обход целевого представления сцены или использовать какую-либо другую технику. По завершении, модуль визуализации перерисовывает сцену в новом состоянии, возможно, с измененными параметрами визуализации.

Работа со специальными представлениями сцены – это большая особая тема, выходящая за рамки данной статьи. Нам важен только сам факт достаточности использования для этого механизма уведомления о событиях.

3.4 Правило уведомления об изменениях сцены

Изменения, о которых сообщают целевые объекты для цели перерисовки сцены и модификации других специальных представлений, могут быть примерно следующих типов: добавление нового геометрического объекта или источника света; удаление объекта сцены; изменение геометрии объекта; изменение свойств поверхности и т.п.

Согласно правилу изменения состояния, все изменения сцены должны производиться через методы изменения значений свойств интерфейса `Entity` (`SetProperty()` и т.п.) Для корректной работы системы отмены и повтора действий, разработчики системы интерактивной графики должны соблюдать следующее **правило уведомления об изменениях сцены**: уведомления об изменениях должны выполняться также исключительно в этих методах, сразу после изменения свойства. Нельзя, например, вызывать уведомления из методов, реализующих действия пользователя.

Методы изменения свойств являются виртуальными, и к ним также должно применяться правило переопределения метода: метод переопределяется в целевом классе и вызывает базовый метод собственно для изменения свойства.

4 Система управления отменой и повтором действий пользователя

Во введении было приведено два способа реализации системы отмены и повтора действий пользователя – использование шаблона команд и протоколирование изменений. Для автоматизации повторов и отмен был выбран второй способ, при котором автоматизация может быть сделана очень легко, но который требует решения сложной задачи по доведению изменений до «других» объектов интерактивной графической системы. Эта задача была успешно решена в разработанной нами инфраструктуре, как описано в предыдущих разделах. Также была подготовлена вся необходимая база для системы управления отменами и повторами действий. Теперь рассмотрим, наконец, как же работает эта система.

Система управления отменами и повторами состоит из объекта-экземпляра класса `UndoRedo` и соответствующего управляющего элемента графического интерфейса пользователя (в простейшем случае представляющего пару кнопок «Отменить» / «Повторить»). Объект `UndoRedo` имеет ссылку на корневой целевой объект (Рис. 7).

Здесь управляющий элемент просто вызывает методы `Undo` / `Redo`.

Все! Это будет работать, при условии, что целевые объекты созданы с учетом простых

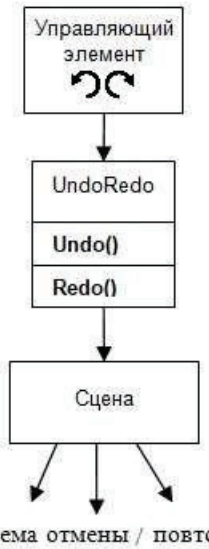

Рис. 7. Система отмены / повтора действий.

правил, приведенных в предыдущих разделах.

Объясним, как это работает.

Согласно правилу изменения состояния, все изменения сцены, в конечном счете, делаются путем изменения значения свойств,

поэтому система должна протоколировать изменения свойств. Рассмотрим метод `SetProperty()` целевого интерфейса, устанавливающий новое значение для простого свойства. Согласно правилу переопределения метода, для установки свойства, в конечном счете, будет вызвана реализация этого метода в базовом классе `Entity`. Она и присвоит свойству новое значение. Но после этого, она вдобавок создаст объект-команду `SetPropCmd`, и выполнит метод `NotifyChange()`, уведомляя об изменении с именем `ACT_DO`, причем команда передается (по адресу) в качестве параметра уведомления. Другие методы изменения свойств (имеющие дело со списочными свойствами) работают аналогично, за исключением того, что создают команду, специфичную для «своего» изменения.

Уведомление `ACT_DO` распространяется вверх по иерархии сцены вплоть до объекта `UndoRedo`. За это ответственна реализация метода `OnChange()` в базовом классе `Entity`, вызываемая в соответствии с правилом переопределения метода.

Объект `UndoRedo` обрабатывает уведомление `ACT_DO`. Он работает по традиционному первому способу, изложенному во введении, и просто помещает команду-параметр на вершину поддерживаемого им стека команд.

Команда содержит всю необходимую информацию для отмены и повтора действия (для `SetPropCmd`: объект, свойство которого менялось; само свойство; старое и новое значение свойства) и методы `Undo()` / `Redo()` для отмены и повтора действия. Эти методы устанавливают, соответственно, старое или новое значение данного свойства в данном объекте. При этом используется соответствующий метод целевого интерфейса. А так как согласно правилу уведомления об изменениях, вспомогательные уведомления об изменениях локализуются в этих методах, эти уведомления тоже будут выполнены и при отменах и повторах, что приведет к соответствующей модификации специальных представлений сцены и перерисовке сцены.

Методы `Undo()` / `Redo()` объекта `UndoRedo`, вызываемые управляющим элементом, когда пользователь хочет отменить или повторить команду, выполняют соответствующий метод в текущей команде стека команд и передвигают указатель текущей команды в стеке назад или вперед, соответственно.

В итоге целевые объекты, исполняющие при изменении сцены методы изменения значений свойств, даже не могут отличить, приходят ли эти изменения из графического интерфейса пользователя или от системы управления отменой и повтором действий. В действительности они даже не имеют представления, о наличии системы управления отменой / повтором – эта система может быть добавлена как абсолютно независимая компонента.

Теперь о специфике обработки действий. Эффект действия должен отменяться или повторяться системой управления отменами и повторами за один шаг. Это достигается следующим образом.

Методы целевого интерфейса `ExecProc()` и `ExecFunc()`, выполняющие действия, генерируют уведомление `ACT_BEGIN` перед выполнением действия и уведомление `ACT_END` после. Аналогично уведомлению `ACT_DO`, эти уведомления распространяются по иерархии сцены и доходят до объекта `UndoRedo`. Получив уведомление `ACT_BEGIN`, объект `UndoRedo` помещает в стек команд специальную команду-контейнер `CompoundCmd`, и последующие команды – вплоть до уведомления `ACT_END` – записываются уже в этот контейнер, а не в основной стек. Методы `Undo()` / `Redo()` составной команды исполняют все команды в контейнере в соответствующем (обратном или прямом) порядке. Так как в главном стеке для действия находится только одна (составная) команда, то отмена и повтор действий выполняются за один шаг.

Пример стека команд приведен на Рис. 8.

5 Применение на других уровнях

Описанная система управления отменами и повторами может применять и на уровне от-

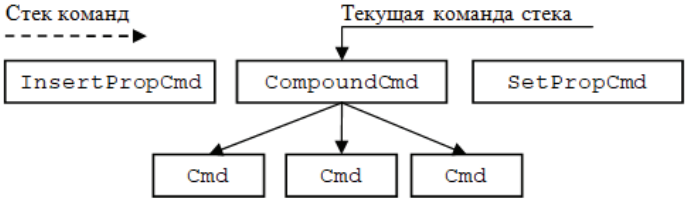

Рис. 8. Пример стека команд.

дельных виджетов. Представим себе панель редактирования какого-нибудь сложного объекта (например, таблицы) с кнопками «Отменить», «Повторить», «Применить», «ОК», первые две из которых позволяют отменить или повторить одно пользовательское действие, выполненное в этой панели. Задача состоит в реализации функций этих кнопок.

Это реализуется следующим способом (Рис. 9).

Как было сказано в подразделе 2.3, в этом случае редактируется копия исходного целевого объекта. Управляющий элемент имеет ссылку на исходный целевой объект. Начиная

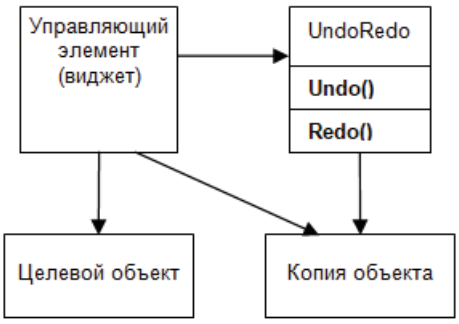

Рис. 9. Отмена / повтор для отдельного виджета.

редактирование, управляющий элемент создает копию исходного объекта и новый экземпляр класса `UndoRedo`, которому в качестве контролируемого объекта задается созданная копия. Дальнейшие действия пользователя в панели приводят к изменению копии исходного объекта. При нажатии кнопок «Отменить», «Повторить» управляющий элемент вызывает методы `Undo()` / `Redo()` объекта `UndoRedo`, что приводит к отмене / повтору действий, произведенных с копией. При нажатии кнопок «Применить» или «ОК» исходный объект приводится к состоянию копии.

Эта схема работает автоматически, как и предыдущая. Отличие толь в том, что любые уведомления об изменениях, исходящие от редактируемой копии, игнорируются (они распространяются только в виджет). Окончательные уведомления об изменениях, на которые могут прореагировать «другие» объекты графической системы, будут возбуждены методами изменения значений свойств при приведении исходного объекта к состоянию копии.

6 Заключение

Представлена инфраструктура интерактивной графической системы, которая – при соблюдении разработчиками содержательного контента нескольких несложных правил – обеспечивает полностью автоматическое управление отменами и повторами действий пользователя и тем самым снижает трудозатраты на создание и сопровождение графических систем.

Данная инфраструктура реализована и используется в комплексах оптического моделирования, применяемых в Институте прикладной математики РАН для научных исследований.

Благодарности

Благодарим коллег по разработке комплексов оптического моделирования за использование изложенной инфраструктуры и сделанные при этом полезные замечания.

Список литературы

Фасман С. 2016. *Undo и Redo – анализ и реализации.* URL: https://habrahabr.ru/post/306398/ (дата обращения: 20.02.2018).

Жданов Д.Д., Потемин И.С., Галактионов В.А., Барладян Б.Х., Востряков К.А., Шапиро Л.З. 2011. *Спектральная трассировка лучей в задачах построения фотореалистичных изображений* // Программирование, 2011, № 5, с. 13-26.

Галактионов В.А., Дерябин Н.Б., Денисов Е.Ю. 2009. *Объектно-ориентированный подход к реализации систем компьютерной графики* // Информационно-измерительные и управляющие системы, № 6, 2009, с. 96-108.

Дерябин Н.Б., Жданов Д.Д., Соколов В.Г. 2017. *Внедрение языка сценариев в программные комплексы оптического моделирования* // Программирование, 2017, № 1, с. 40-53.