ПРИКЛАДНЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 004.921+004.415.2

О.Г. Казанцева, Е.А. Барановский, Е.А. Ландарский

ШЕЙДЕРНАЯ МЕТАМОДЕЛЬ КАК КОМПОНЕНТ АРХИТЕКТУРЫ ИНТЕРАКТИВНОГО ПРИЛОЖЕНИЯ

Предлагается метамодель рендеринга в качестве основы для архитектуры сложных графических приложений. Данная модель позволяет отделить абстрактные логические и алгоритмические подходы, используемые в теории современной компьютерной графики, от вопросов управления вычислительными ресурсами и предоставляет большие возможности по управлению сложностью программного кода.

Введение

Одними из ключевых элементов компьютерной графики стали программируемые графические шейдеры – программы, которые могут быть загружены для исполнения на графическом процессоре [1].

Использование таких устоявшихся парадигм, как объектно-ориентированное и процедурное программирование, сегодня не может в полной мере удовлетворить потребность в управлении постоянно растущей сложностью программного обеспечения. Новые подходы и технологии наподобие аспектно-ориентированной парадигмы и концепции IoC(Inversion of Control)-контейнера уже доказали свою эффективность в сфере заказных программных продуктов (интернет-магазинов, систем автоматизации документооборота и т. д.). Между тем опасения потерь производительности замедлили внедрение этих технологий при разработке программных ядер для приложений, насыщенных графикой: игр, систем автоматизации проектирования и моделирования, что является серьезным заблуждением, так как мощность даже мобильных устройств сегодня значительно возросла. Более того, любой инструмент, упрощающий контроль над программным обеспечением, всегда можно использовать разумно, чтобы обойти его недостатки.

В работе анализируются существующие подходы к управлению программируемого сценария рендеринга (programmable graphics pipeline) и проектируются более гибкие механизмы рендеринга с учетом лучших их черт.

1. Стратегия управления шейдерами

Управление шейдерами порождает целый ряд проблем при разработке приложений, использующих аппаратное ускорение графики. При этом многие из них являются общими для всех графических API (application programming interface), используемых на сегодняшний день как на мобильных, так и на настольных платформах. Общий сценарий работы с шейдерами представляется как следующая последовательность шагов, теоретически не вызывающих явных трудностей [2–4]:

– инициализация:

- подготовка исходного кода составляющих подпрограмм (вершинный шейдер, пиксельный шейдер, геометрический шейдер и т. д.);
- компиляция каждой подпрограммы;
- объявление параметров, используемых в каждой подпрограмме;
- связывание при необходимости подпрограмм в единую программу;

– использование в цикле рендеринга:

- активация ресурсов (например, текстуры) в памяти GPU (graphics processing unit);
- передача параметров в шейдерные подпрограммы;

- активация шейдерного кода;

– освобождение ресурсов, занимаемых шейдерным кодом или ассоциированных с ним, когда шейдер не нужен.

Следует отметить, что порядок перечисленных выше шагов зависит от специфики используемого API. Как видно из общего сценария работы с шейдерами, с применением шейдеров сопряжено достаточно интенсивное использование вызовов методов графического API, а также резервирование нескольких различных ресурсов в памяти GPU.

Предположим, что у нас имеется некоторая сцена, состоящая, как минимум, из нескольких десятков объектов, обладающих набором различных визуальных характеристик – состояний визуализации (Render State). Далее будем называть данное понятие *RS*. Предположим, что эти объекты представляют несколько категорий, которые различаются конфигурациями их *RS*. Объекты могут иметь различные материалы (разную степень прозрачности, реакцию на освещение, излучение света, текстуру и т. п.). Различия могут быть и более существенными. Например, для некоторых категорий объектов определенный набор эффектов, таких как динамическое затенение, может быть проигнорирован из соображений достижения приемлемого баланса производительности и качества. Более того, в целях оптимизации вышеперечисленные свойства могут изменяться в зависимости от расстояния до точки наблюдателя независимо от категории объекта. Для визуализации такой сцены подходят шейдеры. Будучи далеко не самой сложной, данная ситуация уже порождает ряд проблем. Так как программная логика в этом случае допускает динамические изменения *RS* любого из объектов сцены, может возникнуть необходимость переключиться на другой шейдерный код. Вполне возможно, что данный код может понадобиться не сразу или не понадобиться вообще. Также не исключено, что уже загруженный шейдерный код не будет использоваться достаточно длительный промежуток времени. Здесь возникает вопрос о времени инициализации и загрузки шейдеров в память GPU, вопрос их совместного использования несколькими объектами, а также эффективного отслеживания неиспользуемого кода. Вместе с общей нетривиальностью приложений, пользующихся аппаратным ускорением графики, и необходимостью изолировать всю полезную для пользователя логику от управления ресурсами системы, эти вопросы представляют достаточно серьезный барьер для проектирования и написания качественных и гибких программ.

Для решения вышеописанных проблем можно выбирать различные пути: от применения усложненных структур управления на стороне GPU до организации специальной логики на стороне CPU.

Два самых простых и распространенных подхода:

– использование так называемых убершейдеров;

– подготовка всевозможных вариантов шейдеров по отдельности и их предварительное кэширование.

Первый подход подразумевает написание одной или нескольких больших шейдерных программ, содержащих множество операторов ветвления и макроопределения. Каждая такая программа затем используется для работы с несколькими вариантами *RS* в зависимости от значений достаточно обширного набора входных параметров. Преимущества такого подхода состоят в существенном сокращении числа дескрипторов ресурсов, выделенных в памяти GPU, в централизации описания шейдерной модели, а также в реализации достаточно простой логики конечного автомата для управления сценарием рендеринга. Однако здесь имеется и ряд серьезных недостатков. В первую очередь это удар по производительности при работе с убершейдерами из-за необходимости выполнения большого числа проверок на стороне GPU, что наиболее заметно на этапе вычисления цвета фрагментов изображения для пиксельных шейдеров и для любых шейдеров в целом при работе с мобильными платформами. Кроме того, при необходимости радикальных изменений в сценарии рендеринга приходится существенно усложнять либо переписывать весь шейдерный код.

Второй подход состоит в разбиении каждой из шейдерных подпрограмм на секции, отвечающие за тот или иной этап вычисления, написании реализации каждой из таких секций для всех частных случаев и сборки всевозможных комбинаций из полученных составляющих кода. На этапе выполнения программы в этом случае все вариации, которые могут потребо-

ваться, предварительно компилируются и загружаются в память GPU. Преимущество данного подхода состоит в естественности разбиения сценария рендеринга на логические блоки, которые легко изменять, добавлять и удалять, а также в простоте этапа инициализации. В остальном это решение привносит избыток использования памяти GPU, а следовательно, и снижение производительности (за счет дополнительных действий, предотвращающих исчерпание ресурсов GPU). Кроме того, оно обычно требует использования дополнительных инструментов для проверки всех фрагментов кода и генерации из них готовых шейдеров.

Описанные подходы действуют абсолютно противоположно, но на практике чаще всего требуется компромисс. Более того, они слишком прямолинейны и обладают целым рядом серьезных недостатков.

Наличие отдельной иерархии классов, инкапсулирующих работу с конфигурируемой шейдерной моделью, основанной на метаданных, позволило бы упростить разработку сложных сценариев рендеринга и гибко контролировать баланс производительности и качества на каждом его этапе.

2. Общий сценарий рендеринга сцены

Опишем общие концепции логики рендеринга сцены с учетом возможности программирования на стороне GPU.

Интерфейс *Skippable* определяет поведение классов, экземпляры которых имеют аспект логического состояния, позволяющий определить, следует ли пропустить данный объект на итерации некоторого алгоритма.

Интерфейс *Stateful* определяет поведение классов, экземпляры которых имеют составное состояние (куда и входит *RS*) и предоставляют сервисы для его изменения.

Интерфейс *Transformable* определяет поведение классов, экземпляры которых имеют положение и границы в пространстве и предоставляют сервисы по их изменению.

Интерфейс *Renderable* расширяет интерфейсы *Skippable, Stateful* и *Transformable* и определяет поведение классов, экземпляры которых поддаются процессу рендеринга.

Интерфейс *Node* расширяет интерфейсы *Skippable, Stateful* и *Transformable* и определяет поведение классов, экземпляры которых могут быть добавлены в граф сцены с целью группировки нескольких экземпляров типа *Renderable*.

Интерфейс *Controller* определяет поведение классов, экземпляры которых отвечают за обновление определенных аспектов состояния у наблюдаемых экземпляров типа *Stateful*.

Интерфейс *Renderer* определяет поведение классов, экземпляры которых осуществляют рендеринг экземпляров типа *Renderable*.

Интерфейс *GraphicsPipeline* определяет поведение класса, экземпляр которого предоставляет сервис по рендерингу графа сцены, представленного экземпляром типа *Node*. Этот экземпляр является корневым узлом графа сцены.

Общая логика инициализации *GraphicsPipeline* представлена на диаграмме последовательности (рис. 1). Объекты *models*, загружаемые в самом начале, соответствуют набору так называемых *метамоделей*. Под метамоделью далее будем понимать конфигурируемую сущность, соответствующую некоторым правилам описания и содержащую метаданные, по которым приложение во время исполнения может восстановить состояние набора объектов, связанного общей целью функционирования. Одной из таких сущностей является шейдерная модель, которую рассмотрим ниже. Момент восстановления необходимого состояния объектов, скрытых под абстракцией *GraphicsPipeline* и *Renderer*, обозначен событием *configurePipeline*(..).

Далее рассмотрим логику итерации рендеринга (рис. 2). Вначале поток приложения, выделенный под рендеринг, инициирует итерацию над заданным графом сцены, передавая объекту *graphicsPipeline* его корневой узел (экземпляр *Node*). Далее *graphicsPipeline* собирает коллекцию всех контроллеров *controllers*, ассоциированных со всеми элементами графа сцены, после чего обновляет состояние этих элементов. Затем следует этап предварительной обработки графа сцены (например, сортировки и фильтрации), в результате чего получается упорядоченный список объектов, которые должны быть видны в текущем кадре.

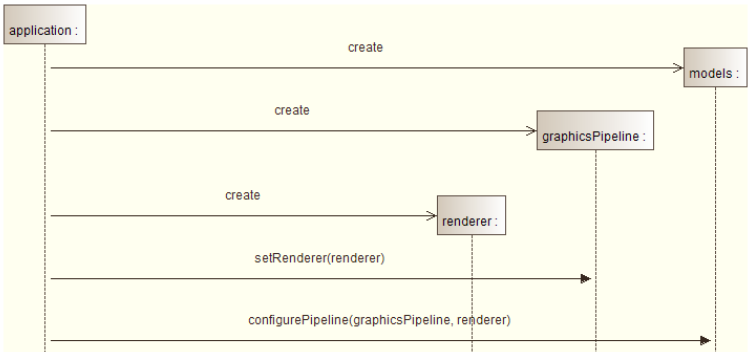

Рис. 1. Общая логика инициализации сценария рендеринга

Одна итерация рендеринга с учетом таких графических техник, как отложенное освещение и эффект адаптации глаз, может состоять из нескольких субитераций – проходов (render passes). Поэтому завершающий этап рендеринга одного кадра состоит в выполнении всех таких проходов в требуемой последовательности. В свою очередь, для каждой субитерации выполняются следующие действия:

– учет параметров графического контекста – объекта, через который осуществляется связь с GPU;

– определение набора из отобранных ранее объектов, который должен быть отображен на текущем проходе;

– подготовка экземпляра *Renderer*, связанного с *graphicsPipeline*. Это необходимо, если, например, некоторые аспекты *RS* должны быть проигнорированы на текущем проходе;

– запрос у экземпляра *Renderer* на заполнение целевого буфера текущего прохода.

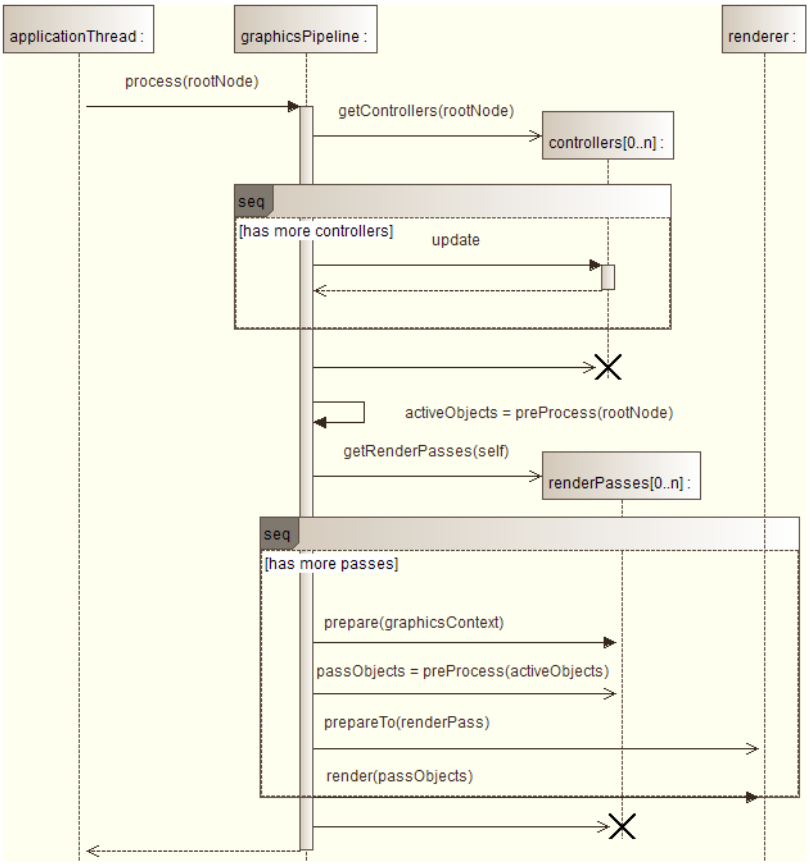

Рис. 2. Итерация рендеринга

Рассмотрим более детально действия, выполняемые экземпляром *Renderer* (далее будем называть его *визуализатор*) в рамках одного прохода. Из диаграммы (рис. 3) видно, что визуализатор для каждого объекта из переданной для отображения коллекции перебирает все активные на текущем проходе аспекты *RS* и обрабатывает их с помощью подходящих аппликаторов – вспомогательных объектов, инкапсулирующих работу с конкретным типом сущности состояния.

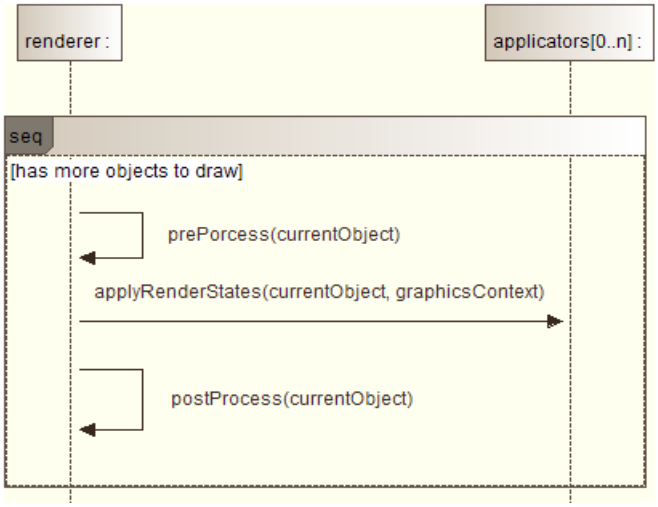

Рис. 3. Действия *Renderer* в рамках одной субитерации

Объекты, которые нужно отобразить в текущем кадре, передаются визуализатору с информацией о *RS* в такой форме, что все параметры, которые нужны при выполнении или подготовке шейдера, содержатся в одной сущности состояния – *shader state*.

Рассмотрим теперь возможную логику действия аппликатора для *shader state* (рис. 4). Вначале данный аппликатор проверяет, была ли изменена сущность *shader state* так, что для нее необходимо сменить шейдерную программу. В зависимости от результата проверки он будет вынужден обратиться к менеджеру шейдеров за подходящей программой, иначе будет использована текущая. После разрешения шейдерной программы аппликатор активирует ее и применяет к ней все параметры *shader state* в соответствии со спецификой используемого графического API.

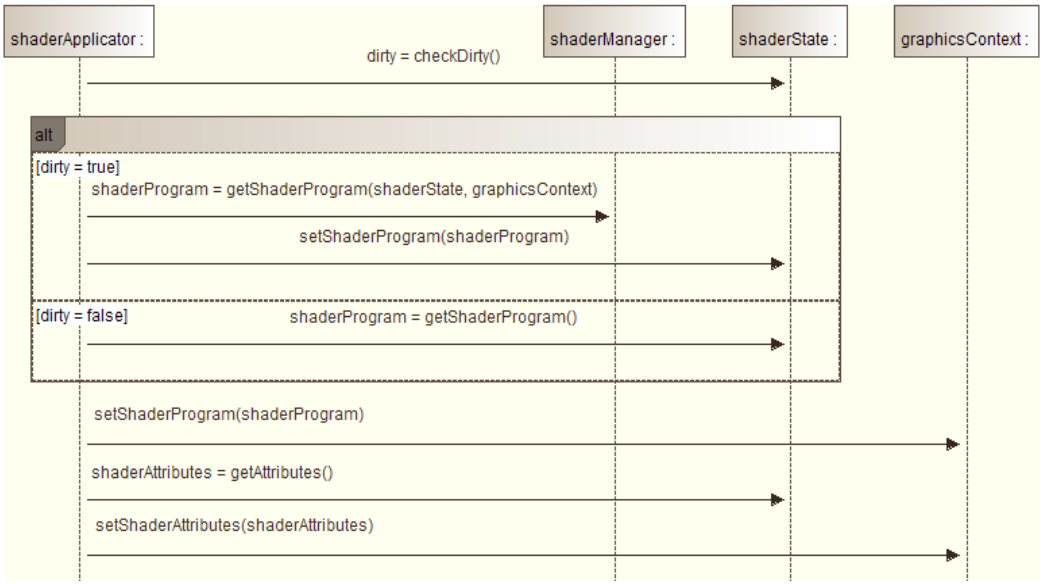

Рис. 4. Диаграмма последовательности действий аппликатора *shader state*

Примерный алгоритм работы менеджера шейдеров (рис. 5) состоит из следущих шагов:

– извлечение из сущности *shader state* ключа, идентифицирующего подходящую программу;

– проверка на существование скомпилированной программы, соответствующей данному ключу;

– в случае отсутствия такой программы:

- построение кода программы;
- его компиляция;
- сохранение скомпилированной программы в реестре;

– возврат программы.

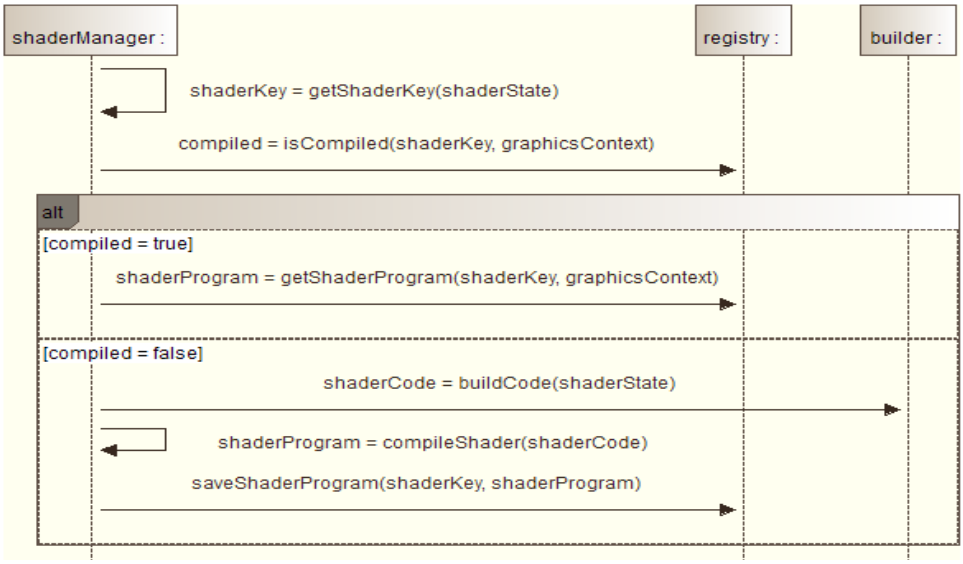

Рис. 5. Принцип действия менеджера шейдеров

3. Шейдерная модель

Ранее было отмечено, что на этапе инициализации приложения значительная часть действий по восстановлению объектной структуры и связыванию компонентов выполняется в соответствии с металогикой и метаданными, содержащимися в предварительно загруженных моделях. В случае организации сценария рендеринга одной из таких моделей является шейдерная модель, отвечающая за его программируемую часть (programmable pipeline). Как и любая другая метасущность, шейдерная модель имеет персистентное представление, предназначенное для возможности хранения между запусками приложения и конфигурирования, и представление во время исполнения приложения. Персистентное представление логично реализовать в одном из структурированных форматов, таких как XML, YAML или JSON.

Представление шейдерной модели времени исполнения приложения удобно разбить на две составляющие: ранее упомянутую сущность *shader state*, являющуюся частью состояния визуализации отображаемого объекта, и вспомогательные объекты, упрощающие работу для классов, реализующих логику рендеринга (т. е. для *GraphicsPipeline*, *Renderer*, аппликатора и менеджера шейдеров).

Сущность *shader state* для классов логики рендеринга должна быть представлена простым интерфейсом, состоящим из методов доступа к следующей информации:

– данным, идентифицирующим загруженную в память GPU шейдерную программу (например, в случае OpenGL это целочисленный идентификатор);

– значениям параметров, передаваемым в шейдерную программу.

Следует отметить, что предоставляемые данным интерфейсом значения перечисленных атрибутов должны быть готовы к непосредственному использованию с конкретным графическим API. Однако скрытая под ним реализация может иметь дополнительные возможности, упрощающие работу с сущностью для вспомогательных классов, описанных ниже.

После загрузки из персистентного представления шейдерная модель создает все необходимые вспомогательные объекты и производит их связывание с общей логикой рендеринга на основе своих метаданных.

Типичным набором вспомогательных объектов в данном случае являются:

– объекты, описывающие все требуемые для построения каждого кадра проходы (render passes);

– объект-валидатор, инъектируемый в аппликатор сущности *shader state* и определяющий, нужно ли сменить шейдерную программу, привязанную к данной сущности;

– объект, инъектируемый в менеджер шейдеров и вычисляющий по полям сущности *render state* ключ, идентифицирующий необходимую для нее шейдерную программу в рамках объекта – хранилища шейдеров;

– объект, инъектируемый в менеджер шейдеров и отвечающий за сборку и кэширование кода требуемой шейдерной программы в случае ее отсутствия в памяти GPU.

Следует отметить, что наиболее удобной и гибкой реализацией связывания любой из метамоделей, в том числе и шейдерной модели, с остальной логикой приложения является инъекция зависимостей в среде, подобной IoC-контейнеру.

Как было отмечено ранее, любые конфигурируемые метамодели, описывающие позднее связывание компонентов приложения, удобно хранить в структурированном представлении. Ниже приведем примерный шаблон персистентного представления простейшей шейдерной модели освещения по Фонгу для OpenGL ES 2.x-3.x в формате XML.

Листинг-шаблон XML-представления шейдерной модели освещения по Фонгу имеет следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<shaderModel>
  <shader type="vertex">
    <declarations>
      <declaration
        name="u_MVMatrix"
        category="uniform"
        dataType="mat4"/>
      <declaration
        name="u_MVPMatrix"
        category="uniform"
        dataType="mat4"/>
      <declaration
        name="a_Position"
        category="attribute"
        dataType="vec4"/>
      <declaration
        name="v_Position"
        category="varying"
        dataType="vec3"/>
      <if target="lightingEnabled" value="true">
        <declaration
          name="a_Normal"
          category="attribute"
          type="vec4"/>
        <declaration
          name="v_Normal"
          category="varying"
          type="vec4"/>
      </if>
    </declarations>
    <body>
```

```
        <if target="lightingEnabled" value="true">
          v_Normal = vec3(u_MVMatrix * vec4(a_Normal, 0.0));
        </if>
        v_Position = vec3(u_MVMatrix * a_Position);
        gl_Position = u_MVPMatrix * a_Position;
      </body>
    </shader>
    <shader type="fragment">
      <!-- аналогичные объявления declarations и body -->
    </shader>
</shaderModel>
```

Заключение

В качестве компромиссного решения в работе предложена метамодель рендеринга, позволяющая достигать требуемого в конкретной ситуации баланса в отношении объема потребляемой памяти, расхода машинного времени, централизации управления родственными компонентами и обеспечения модульности. Концепция этой метамодели легко распространяется на любую функциональность при наличии минимальной поддержки ядром приложения возможностей IoC-контейнера. Данное решение упрощает контроль над архитектурой сложных приложений, не имеет привязки к конкретной платформе и языку программирования. Что касается производительности, то здесь следует отметить возможность разработки дополнительных инструментов, которые на определенном этапе жизненного цикла приложения (например, перед интеграционным тестированием или перед релизом) генерируют обычный код, не использующий металогику.

Список литературы

1. Bailey, M. Graphics shaders: Theory and practice / M. Bailey, S. Cunningham. – 2nd ed. – A K Peters/CRC Press, 2011. – 518 p.

2. Zink, J. Practical rendering and computation with Direct3D 11 / J. Zink, M. Pettineo, J. Hoxley. – CRC Press, 2011. – 637 p.

3. Sellers, G. OpenGL superBible: comprehensive tutorial and reference / G. Sellers, R.S. Wright, N. Haemel. – 6th ed. – Crawfordsille : Addison-Wesley, 2013. – 848 p.

4. Smithwick, M. Pro OpenGL ES for Android / M. Smithwick, M. Verma. – N.Y. : Apress, 2012. – 308 p.

Поступила 08.04.2014

Витебский государственный университет
им. П.М. Машерова,
Витебск, пр. Московский, 33
e-mail: o.g.kazantseva@gmail.com,
judzin.baranovsky@gmail.com

V. Kazantsava, Y. Baranouski, Y. Landarski

SHADER METAMODEL AS A COMPONENT OF AN INTERACTIVE APPLICATION ARCHITECTURE

This article introduces the rendering metamodel as an architecture basis for complicated graphical applications. The model separates abstract logical and algorithmic approaches used in modern computer graphics theory from problems of computational resource management. It also yields wide possibilities for program code complexity management.