Лабораторная работа 2 – Применение модели грид-потоков для внедрения в грид существующего приложения

1. Введение

Напомним, что основной целью лабораторных работ по инструментарию Alchemi является знакомство с базовой моделью программирования данного инструментария – моделью *грид-потоков*. В первой лабораторной работе рассматривалась проблема разработки *нового* распределенного приложения на примере задачи визуализации функций комплексного переменного. Вторая лабораторная работа посвящена внедрению в грид *существующего* приложения и имеет, тем самым, большой практический интерес[1].

Следует заметить, что проблема внедрения приложений в грид в настоящее время стоит достаточно остро[2]. Ситуация усугубляется тем, что широкое распространение получают самые разнообразные реализации промежуточного программного обеспечения, большинство из которых несовместимы между собой. Таким образом, все желающие приобщиться к технологиям грид неизбежно встают перед проблемой выбора некоторой реализации промежуточного программного обеспечения. Причем поддерживать несколько реализаций зачастую оказывается дорого, а ориентироваться лишь на одну – довольно рискованно.

Для решения данной проблемы предлагаются различные подходы. Однако вполне очевидно, что в подобной ситуации преимущества получают инструментарии, позволяющие внедрять существующие приложения в грид с *минимальными* усилиями. Если выбранный инструментарий не получит распространения в будущем, то, во всяком случае, затраты на разработку не окажутся слишком высокими. По этому показателю инструментарий Alchemi заслуживает довольно высокой оценки, подтверждаемой, в частности, и данной лабораторной работой.

В качестве приложения для внедрения в грид была выбрана широко известная программа POV-Ray, предназначенная для создания фотореалистических изображений методом трассировки лучей. Данный пакет предоставляет одновременно стандартную программу трассировки и формат данных для описания сложных трехмерных сцен. Следует также заметить, что пакет POV-Ray является бесплатным и распространяется с открытым исходным кодом. Загрузить пакет можно с официального сайта проекта http://www.povray.org. Широкая популярность пакета POV-Ray привела к созданию множества его модификаций, надстроек и неофициальных сборок, значительно расширяющих функциональность стандартной версии. Для выполнения данной лабораторной работы потребуется неофициальная сборка пакета под названием MegaPOV, которая также является бесплатной и распространяется с открытым исходным кодом. Пакет MegaPOV доступен для загрузки с официального сайта http://megapov.inetart.net. Основным расширением функциональности в пакете MegaPOV является возможность симуляции различных механизмов, однако в состав пакета входит также консольная версия программы трассировки лучей, полностью совместимая с файлами сцен POV-Ray. Именно консольную версию программы лучше всего использовать для организации распределенных вычислений.

2. Цель лабораторной работы

Целью лабораторной работы является внедрение *существующего* приложения в грид, построенную на базе инструментария Alchemi. В качестве такого приложения выбирается известная программа трассировки лучей POV-Ray с предустановленной неофициальной сборкой MegaPOV. Подробнее данные программы обсуждаются ниже.

- *Упражнение 1.*

  Знакомство с программой трассировки лучей MegaPOV.
- *Упражнение 2.*

  Декомпозиция входного набора данных на независимые части.
- *Упражнение 3.*

  Разработка распределенного приложения для инструментария Alchemi.

Примерное время выполнения лабораторной работы: **240 минут**.

---

[1] Под внедрением в грид некоторого приложения здесь подразумевается организация распределенной обработки данных на вычислительных ресурсах грид с помощью данного приложения. После завершения обработки данных пользователь должен получить итоговый результат как единое целое. Таким образом, для пользователя распределенные вычислительные ресурсы выступают в роли *единого виртуального* вычислительного ресурса.

[2] Заметим, что о проблеме наполнения грид приложениями говорят многие исследователи и компании, к числу которых относится и компания Intel. Группа разработчиков инструментария GPE (проект компании Intel, распространяется с открытым исходным кодом на ресурсе SourceForge) уделила данной проблеме особое внимание, разработав специальную технологию GridBean для внедрения существующих приложений в грид. Разработчики полагают, что такой подход (наряду с другими технологиями) должен обеспечить успех данного проекта.

При выполнении лабораторной работы предполагаются знания раздела “Инструментарий Alchemi” учебного курса по технологиям грид, а также базовые знания языка программирования Visual C# и навыки работы в среде Microsoft Visual Studio. Весьма полезным было бы знакомство с первой лабораторной работой, однако возможно и отдельное использование данной лабораторной работы, поскольку необходимые основные положения будут изложены повторно.

3. Упражнение 1 – Знакомство с программой трассировки лучей POV-Ray

Данное упражнение является кратким экскурсом в удивительный мир трассировки лучей и дает базовую информацию, необходимую для использования программ POV-Ray и MegaPOV.

3.1. Метод трассировки лучей

Трассировка лучей – это метод машинной графики, позволяющий создавать фотореалистические изображения любых трехмерных сцен. Трассировка лучей моделирует прохождение лучей света через изображаемую сцену. Фотореализм достигается путем математического моделирования оптических свойств света и его взаимодействия с объектами. Сначала отдельные объекты располагаются в трехмерном пространстве-сцене, а также задаются физические и оптические свойства их поверхностей и их цвет. Затем определяется, где будут расположены источники света и наблюдатель. И, наконец, с помощью программы трассировки лучей создается математическая модель сцены, света и наблюдателя, с помощью которой вычисляется цвет каждого пикселя графического изображения, получаемого на экране дисплея. На рис. 1 показаны некоторые изображения, сгенерированные на компьютере методом трассировки лучей.

Рис. 1. Примеры изображений, сгенерированных методом трассировки лучей в программе POV-Ray. Изображения взяты с официального сайта проекта http://www.povray.org

В качестве хорошей аналогии довольно сложного математического процесса трассировки лучей можно рассмотреть комнату с одним открытым окном. Вообразим, что в ней только что покрасили в разные цвета стены, пол, потолок, а заодно и все, что находится внутри, причем краска еще не высохла. Вообразим теперь, что кто-то бросает в окно этой комнаты очень прыгучий резиновый мячик. Мячик начнет прыгать по комнате, стукаясь о свежевыкрашенные поверхности и собирая на себя их краску. Так будет продолжаться до тех пор, пока этот бесконечно прыгучий мяч не вылетит обратно в окно. Так вот, тот цвет, в который в результате окрасится мячик, и будет соответствовать цвету той точки окна-дисплея, через которую он вылетел из комнаты-сцены. Если же перейти от аналогий к более строгому определению этого математического метода, то происходит обратная трассировка лучей света от воображаемого глаза наблюдателя через каждый пиксель экрана, сквозь который наблюдатель смотрит на сцену, со всеми возможными преломлениями и отражениями внутри сцены к источнику или источникам света. Для каждого луча определяются возможные точки пересечения с поверхностями геометрических объектов, формирующих сцену. Для каждой поверхности, пересекаемой лучом, определяется расстояние от наблюдателя до точки пересечения. Поверхность с минимальным расстоянием считается видимой наблюдателем. Для этой поверхности производится математическое отражение луча, а если поверхность частично или полностью прозрачна, то и преломление луча. Отраженный и преломленный лучи называются вторичными лучами, и для них повторяется та же самая процедура, что и для исходного первичного луча, а именно поиск пересекаемых поверхностей, сортировка по расстоянию и вычисление новых вторичных лучей. Все пересекаемые поверхности с

минимальными расстояниями включаются в так называемое двоичное дерево трассировки, где, например, левые ветви используются для отраженных лучей, а правые для преломленных. Максимальная "глубина" трассировки лучей либо задается пользователем, либо определяется самой программой-трассировщиком в зависимости от наличия свободной памяти. Затем для определения цвета пикселя производится проход дерева снизу вверх, при котором суммируются интенсивности цвета всех поверхностей, пересеченных первичным и всеми вторичными лучами с учетом затухания. Если ни одна из поверхностей не была пересечена первичным лучом, то пикселю присваивается цвет фона. Проблемы вычисления интенсивности цвета поверхностей подробно разбираются, например, в работе [1].

Трассировка лучей используется в компьютерной графике давно и успешно. Программы, выполняющие трассировку, нередко являются составной частью сложных систем геометрического моделирования, таких как, например, системы 3D Studio Max. Однако надо заметить, что, обеспечивая очень высокое качество изображений, трассировка лучей как метод визуализации требует довольно много времени и, как правило, применяется на финальных стадиях подготовки изображений. Зачастую одно изображение может трассироваться на протяжении нескольких часов и даже суток, но результаты оправдывают временные затраты. Совсем недавно трудоемкость выполнения вычислений препятствовала широкому распространению программ трассировки среди пользователей персональных компьютеров. Однако сейчас, когда мощные и сравнительно дешевые системы сильно потеснили дорогие профессиональные графические рабочие станции, трассировка лучей и удивительные миры света и теней могут войти в каждый дом и в каждую лабораторию. Среди широкого семейства программ, в которых, так или иначе, используется трассировка лучей, особое место занимает POV-Ray. Данный пакет за долгий срок своего существования заслужил общее признание и уважение и используется в самых различных научных исследованиях.

Как уже было сказано, задача трассировки лучей является чрезвычайно ресурсоемкой[3]. Естественным путем преодоления этой сложности является применение распределенных вычислений. При этом одна и та же сцена будет разбиваться на блоки и рассчитываться независимо на множестве компьютеров сети. Поскольку сам пакет POV-Ray не поддерживает подобной функциональности, данная задача является особенно актуальной. В данной лабораторной работе предлагается весьма эффективный и простой в реализации способ внедрения существующих приложений в грид на примере программы POV-Ray.

3.2. Краткое описание программы MegaPOV

Поскольку сам по себе пакет POV-Ray не является основным предметом рассмотрения в данной лабораторной работе, а лишь иллюстрирует некоторые общие моменты, мы не будем вдаваться в подробности и детально рассматривать указанную программу. Однако чтобы у читателя сложилось общее представление, некоторые сведения все же могут быть полезными. Данный материал не является обязательным для изучения и может быть пропущен.

Сразу оговоримся, что при выполнении лабораторной работы будет использоваться лишь неофициальная сборка пакета POV-Ray под названием MegaPOV. Дело в том, что официальный пакет POV-Ray содержит лишь графическую версию программы трассировки лучей. Данная версия программы очень удобна в работе, однако большинство операций доступны лишь в виде взаимодействий с графическим интерфейсом. Для нашей задачи гораздо лучше подходит консольная версия программы, которая позволяет специфицировать входные параметры и файл, в который будет сохранен результат. Именно для этой цели будет использоваться неофициальная сборка MegaPOV. Помимо того, что пакет MegaPOV добавляет ряд уникальных возможностей к стандартной версии, в его состав входит необходимая для нас консольная версия программы трассировки, полностью совместимая с файлами сцен POV-Ray.

Заметим, что компьютерные сцены для программы POV-Ray пишутся на специальном *языке описания сцен* (*SDL – Scene Description Language*), синтаксис которого весьма похож на синтаксис языка программирования C. Рассмотрение данного языка выходит за рамки лабораторной работы. Однако интересующиеся читатели смогут без труда познакомиться с ним на многочисленных примерах, поставляемых в комплекте с программой POV-Ray. Примеры файлов сцен содержатся в подкаталоге **scenes** установочного каталога.

Рассмотрим работу программы трассировки MegaPOV на примере произвольной сцены. Прежде всего программу следует установить. Для этого достаточно распаковать дистрибутив, загруженный с сайта проекта (и прилагаемый к данной лабораторной работе), в произвольный каталог на жестком диске. Каких-либо дополнительных настроек системы не требуется. Откройте командную строку с

[3] Интересным примером может служить информация, появившаяся на официальном сайте проекта POV-Ray 19 ноября 2006 года. В этот день была завершена визуализация детализированной карты западной части Северной Америки. Разрешение изображения составило 96000 ×54000 = 5 Гигапикселей, а процесс рендеринга длился *пять месяцев*.

помощью команды **Start | Programs | Standard | Command Prompt**. Далее перейдите в подкаталог **bin** установочного каталога и выполните следующие действия:

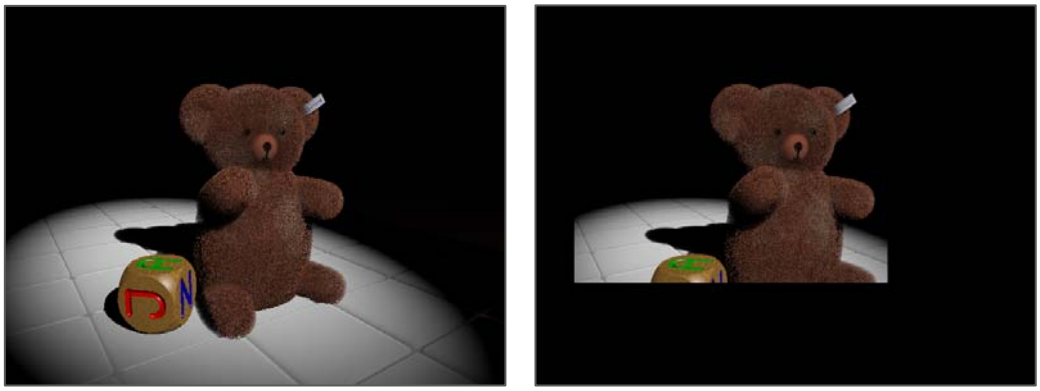
```
megapov +I..\scenes\megapov\bear.pov +L..\include +W320 +H240 +Obear.png +FN16
```

В результате этих действий начнется процесс трассировки изображения, по окончанию которого в подкаталоге **bin** появится графический файл с именем **bear.png**. Сгенерированное изображение показано на рис. 2*а*.

Дадим пояснения к основным параметрам приложения MegaPOV, так как в дальнейшем нам предстоит их использовать. Общий формат команды запуска программы MegaPOV выглядит следующими образом:

```
megapov [+/-]Опция [+/-]Опция [+/-]Опция ...
```

Для активации некоторой новой опции необходимо воспользоваться символом “+”, для отключения опции – символом “-”.



а) *б)*

Рис. 2. Изображение, сгенерированное на основе файла **bear.pov** (*а*). Использование параметров программы MegaPOV для генерации части изображения (*б*)

Все опции, доступные пользователям программы MegaPOV, можно разделить на следующие группы:

- *Опции синтаксического разбора (Parsing Options):*
  - **I<имя файла>** – путь к файлу сцены для трассировки;
  - **L<имя папки>** – путь к каталогу вспомогательных файлов.
- *Опции вывода (Output Options):*
  - **Hn** – высота изображения *n* пикселей;
  - **Wn** – ширина изображения *n* пикселей;
  - **SRn** – начальная строка *n* обрабатываемой части изображения;
  - **ERn** – конечная строка *n* обрабатываемой части изображения;
  - **SCn** – начальный столбец *n* обрабатываемой части изображения;
  - **ECn** – конечный столбец *n* обрабатываемой части изображения;
  - **FN[n]** – формат PNG для выходного файла (*n* бит на пиксель).
- *Опции трассировки (Tracing Options):*
  - **A** – активация сглаживания изображения;
  - **Rn** – глубина сглаживания (использовать $n \times n$ лучей для каждого пикселя).

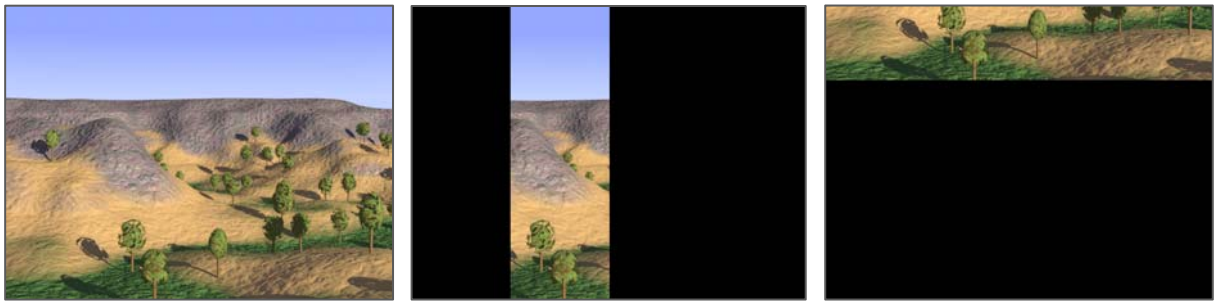
Здесь перечислены лишь самые важные параметры программы MegaPOV, в действительности же их намного больше. Однако указанных здесь параметров вполне достаточно для выполнения лабораторной работы.

Приведем пример использования описанных параметров для обработки определенной прямоугольной части сцены с активированными настройками сглаживания:

```
megapov +I..\scenes\megapov\bear.pov +L..\include +W640 +H480 +SR1 +ER200 +SC50 +EC250 +A +R4
+Obear.png +FN16
```

Результат выполнения данной команды представлен на рис. 2*б*. Легко видеть, что программа MegaPOV обработала не все изображение целиком, а лишь указанную прямоугольную часть. Таким образом, все изображение можно разбить на прямоугольные части (блоки) и обработать независимо на различных компьютерах сети. Подобный подход использовался и в предыдущей лабораторной работе. Различие состоит в том, что в предыдущем случае расчет производился Исполнителями, а теперь все расчеты будут производиться "внешним" приложением, в то время как Исполнители будут ждать завершения его работы.

Отметим одну неприятную особенность параметров **`SRn`**, **`ERn`**, **`SCn`**, **`ECn`**. Опции **`SCn`** и **`ECn`** работают вполне очевидно, вырезая из изображения заданную вертикальную полосу, при этом нулевым столбцом является крайний левый столбец изображения. Как видно из рис. 3*б*, работа данных опций не вызывает нареканий. Аналогично работают и параметры **`SRn`** и **`ERn`**, вырезая из изображения заданную горизонтальную полосу, причем нулевой строкой считается самая верхняя строка изображения. Однако вне зависимости от заданных параметров горизонтальная полоса появляется в верхней части изображения, что иллюстрируется на рис. 3*в*. Трудно сказать, является ли подобное поведение программы ошибочным или его предусмотрели разработчики, но указанный момент следует учитывать при работе с программой MegaPOV.



а) *б)* *в)*

Рис. 3. Исходное изображение размера 640 × 480 пикселей (*а*). Результат применения параметров **`SC160`** и **`EC320`** (*б*) и параметров **`SR360`** и **`ER480`** (*в*)

В заключение заметим, что программы POV-Ray и MegaPOV поставляются с подробной документацией и большим количеством примеров, которые позволяют довольно быстро освоиться и приступить к созданию собственных сцен.

4. Упражнение 2 – Декомпозиция входного набора данных на независимые части

Процесс внедрения существующего приложения в грид с помощью модели грид-потоков можно представить в виде последовательности трех основных шагов:

- анализ входного набора данных и его декомпозиция на *независимые* части;
- реализация класса *грид-потока* для запуска приложения на удаленном Исполнителе с определенной порцией данных;
- организация параллельной работы грид-потоков в вычислительной сети с помощью класса *грид-приложения*.

Таким образом, при внедрении существующего приложения в грид распределенные вычисления сводятся к выполнению однотипной обработки элементов большого набора данных на различных Исполнителях. В этом случае говорят, что имеет место *параллелизм по данным*, и выделение подзадач для грид-потоков сводится к разделению имеющихся данных. Для нашей учебной задачи исходное множество пикселей изображения может быть разделено на отдельные группы строк – *ленточная* схема разделения данных или на прямоугольные наборы пикселей – *блочная* схема разделения данных (рис. 4).

а) *б)* *в)*

Рис. 4. Варианты разделения множества пикселей изображения: исходное изображение (*а*), ленточная схема (*б*) и блочная схема (*в*)

Приведенную схему нельзя признать универсальной, так как она не применима к любому приложению. На пути ее реализации может возникать множество вопросов и проблем. Приведем некоторые из них.

Во-первых, набор входных данных должен допускать декомпозицию на *независимые* части. В нашем случае основными входными данными (которые подлежат разделению) является множество обрабатываемых пикселей. Очевидно, что отдельные группы пикселей можно обрабатывать совершенно независимо. Однако столь простая ситуация встречается не всегда. В качестве примера представим программу, рассчитывающую положение космических тел в некоторые моменты времени. Основными входными данными в этом случае будет являться некоторое множество описаний космических тел (их положения и скорости в начальный момент времени), например, описания всех планет Солнечной системы. Ясно, что простое разбиение множества всех тел на группы и запуск программы для каждой такой группы приведет к неверному расчету их положения, так как отдельные группы тел влияют друг на друга[4].

Во-вторых, должна существовать возможность запуска приложения для обработки определенной порции входных данных. В нашем случае это обеспечивается благодаря специальным параметрам программы MegaPOV, позволяющим указать обрабатываемую часть изображения. Если бы разработчики не предусмотрели данные параметры, параллельная обработка изображения на нескольких компьютерах оказалась бы невозможной.

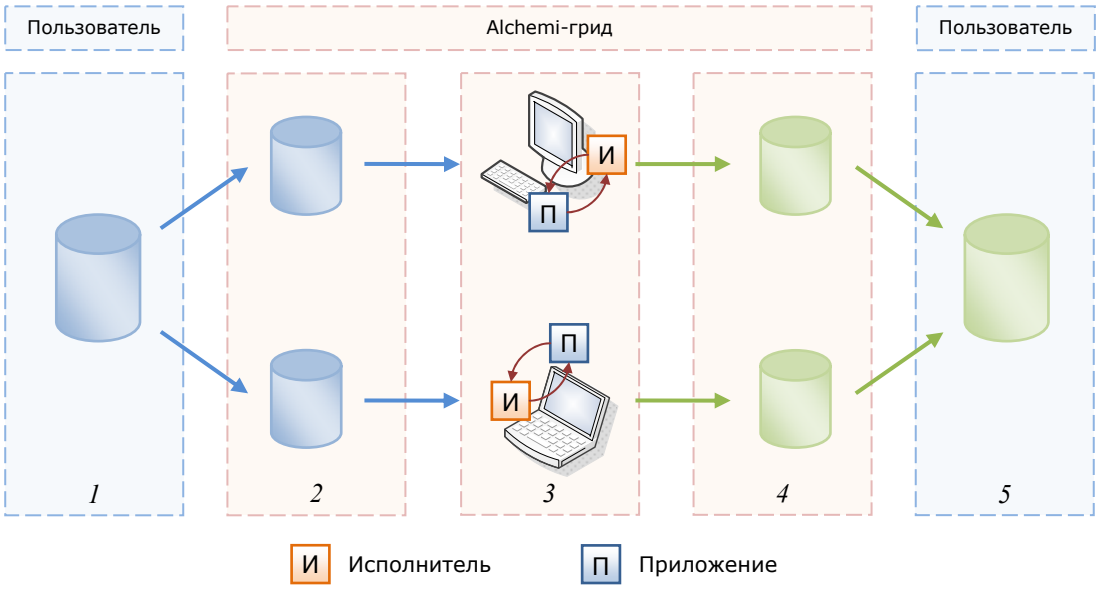


Рис. 5. Процесс распределенной обработки данных в виде последовательности пяти основных шагов

[4] Заметим, что в ряде случаев такое разбиение тел на группы (или *кластеры*) возможно. Для этого необходимо, чтобы силы притяжения между телами различных групп были пренебрежимо малы по сравнению с силами притяжения между телами одной группы. Например, в случае Солнечной системы в роли кластеров могут рассматриваться планеты со своими спутниками. На первом шаге рассчитываются положения кластеров (как будто они являются цельными объектами), затем рассчитываются положения тел внутри каждого кластера.

В-третьих, итоговый результат необходимо конструировать из частичных результатов, полученных при обработке отдельных порций данных. И здесь также существует ряд потенциальных проблем. В нашем случае этот этап очевиден: отдельные части изображения необходимо совместить в нужном порядке.

Следует заметить, что четко формализовать все условия, которым должна удовлетворять программа для внедрения в грид (здесь имеется в виду именно распределенная обработка данных, а не просто запуск приложения на удаленном ресурсе), довольно трудно. Вероятно, выше перечислены далеко не все из них. Однако вряд ли имеет смысл подробно останавливаться на этом вопросе, поскольку для каждого конкретного случая возможность внедрения приложения в грид вполне очевидна.

Основные этапы распределенной обработки данных в грид, построенной на базе инструментария Alchemi, иллюстрируется на рис. 5. На *первом* шаге производится декомпозиция входных данных на независимые части. На *втором* шаге отдельные порции данных загружаются на машины Исполнителей для дальнейшей обработки. На *третьем* шаге производится обработка отельных порций данных на машинах Исполнителей. При этом обработка осуществляется “внешним” приложением, в то время как сами Исполнители простаивают, ожидая окончания его работы. В результате на *четвертом* шаге появляются частичные результаты обработки. Наконец, на *пятом* шаге все частичные результаты объединяются в общий результат, который возвращается пользователю.

Перейдем теперь к разработке класса грид-потока, с помощью которого будет осуществляться запуск “внешнего” приложения MegaPOV для обработки определенной части изображения.

5. Упражнение 3 – Разработка распределенного приложения

Начальный вариант будущей программы представлен в проекте **DistributedMegaPOV**, который содержит некоторую часть исходного кода. В ходе выполнения дальнейших упражнений необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, реализацией описанного алгоритма распределения вычислений и проверкой правильности результатов работы программы.

5.1. Задание 1 – Открытие проекта DistributedMegaPOV

Для открытия проекта **DistributedMegaPOV** выполните следующие шаги:

- Запустите среду Microsoft Visual Studio 2005, если она еще не запущена,
- В меню **File** выполните команду **Open | Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **DistributedMegaPOV**,
- Выберите файл **DistributedMegaPOV.sln** и выполните команду **Open**.

После выполнения описанных шагов в окне **Solution Explorer** (рис. 6) будет отображена структура проекта **DistributedMegaPOV**. В его состав входит приложение **DistributedMegaPOV**, а также одна динамическая библиотека **RenderThread**. Оба проекта подлежат дальнейшей доработке, поэтому опишем их назначение подробнее.

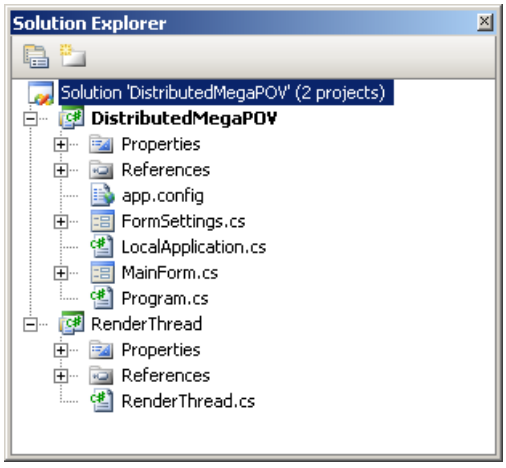


Рис. 6. Состав проекта **DistributedMegaPOV**

Динамическая библиотека **RenderThread** состоит из единственного файла **RenderThread.cs**, в котором объявляется класс грид-потока. Напомним, что для запуска грид-потоков на удаленных Исполнителях инструментарий Alchemi должен произвести копирование модуля, в котором определяется

грид-поток, а также всех вспомогательных модулей, необходимых во время его работы, на машины Исполнителей. Поэтому для минимизации сетевого трафика и времени начала вычислений настоятельно рекомендуется создавать отдельную динамическую библиотеку для реализации грид-потоков.

Проект **DistributedMegaPOV** представляет собой обычное приложение для операционной системы Microsoft Windows. В данном приложении формируются грид-потоки и отправляются для исполнения в вычислительную грид, построенную на базе инструментария Alchemi. Кроме того, предусмотрена также возможность запуска грид-потоков на локальной машине пользователя. В дальнейших заданиях мы подробно рассмотрим реализацию библиотеки **RenderThread** и приложения **DistributedMegaPOV**.

5.2. Задание 2 – Разработка класса грид-потока

Перейдите в проект **RenderThread** и дважды щелкните левой кнопкой мышки на файле **RenderThread.cs**. В результате этих действий на экране появится заготовка кода, предназначенная для дальнейшей доработки. Все фрагменты кода, подлежащие доработке, помечены комментариями, содержащими соответствующие инструкции. На приведенном ниже листинге показана заготовка для класса **`RenderThread`**.

```
using System;
using System.IO;
using System.Drawing;
using System.Diagnostics;
using Alchemi.Core.Owner;

/// <summary>
/// Класс грид-потока. Отвечает за обработку части изображения.
/// </summary>
 [Serializable]
public class RenderThread : GThread
{
    /// <summary>Строка с текстовым представлением сцены для рендеринга.</summary>
    private string inputScene;

    /// <summary>Каталог для хранения временных результатов.</summary>
    private string tempDirectory;

    /// <summary>Рабочий каталог приложения MegaPOV.</summary>
    private string workDirectory;

    /// <summary>Дополнительные аргументы приложения MegaPOV.</summary>
    private string additionalArguments;

    /// <summary>Ширина изображения в пикселях.</summary>
    private int width;

    /// <summary>Высота изображения в пикселях.</summary>
    private int height;

    /// <summary>Начальный ряд пикселей для рендеринга.</summary>
    private int startRow;

    /// <summary>Начальный столбец пикселей для рендеринга.</summary>
    private int startCol;

    /// <summary>Конечный ряд пикселей для рендеринга.</summary>
    private int endRow;

    /// <summary>Конечный столбец пикселей для рендеринга.</summary>
    private int endCol;

    /// <summary>Качесво сглаживания изображения.</summary>
    private AntialisingLevel antialiasLevel;

    /// <summary>Точечный рисунок для хранения части изображения.</summary>
    private Bitmap image;

    /// <summary>
    /// Создает новый экземпляр грид-потока для рендеринга части изображения.
    /// </summary>
    /// <param name="inputScene">строка с текстовым представлением сцены</param>
    /// <param name="tempDirectory">временный каталог</param>
    /// <param name="workDirectory">рабочий каталог</param>
    /// <param name="width">ширина изображения</param>
    /// <param name="height">высота изображения</param>
    /// <param name="startRow">начальный ряд</param>
    /// <param name="startCol">начальный столбец</param>
```

```
    /// <param name="endRow">конечный ряд</param>
    /// <param name="endCol">конечный столбец</param>
    /// <param name="antialiasLevel">настройки сглаживания изображения</param>
    public RenderThread(string inputScene, string tempDirectory, string workDirectory,
                        int width, int height,
                        int startRow, int startCol, int endRow, int endCol,
                        AntialisingLevel antialiasLevel, string additionalArguments)
    {
        ...
    }

    /// <summary>
    /// Основной метод грид-потока. Обрабатывает часть изображения.
    /// </summary>
    public override void Start()
    {
        ...
    }

    /// <summary>Сгенерированная часть общего изображения.</summary>
    public Bitmap Image
    {
        get
        {
            return image;
        }
    }

    /// <summary>Координата левого столбца пикселей.</summary>
    public int StartCol
    {
        get
        {
            return startCol;
        }
    }

    /// <summary>Координата верхнего столбца пикселей.</summary>
    public int StartRow
    {
        get
        {
            return startRow;
        }
    }

    /// <summary>Ширина сгенерированной части изображения.</summary>
    public int Width
    {
        get
        {
            return endCol - startCol;
        }
    }

    /// <summary>Высота сгенерированной части изображения.</summary>
    public int Height
    {
        get
        {
            return endRow - startRow;
        }
    }
}
```

Код данного класса показан в несколько сокращенном виде для концентрации внимания на ключевых моментах. Полный вариант данного кода представлен в прилагаемом к лабораторной работе проекте **DistributedMegaPOV**. В частности, здесь не приводится ряд вспомогательных функций для формирования имен входного и выходного файла, а также аргументов приложения MegaPOV. Поскольку данные функции не требуется дорабатывать, но они используются в ключевых фрагментах кода, мы ограничимся рассмотрением их прототипов.

Перед тем, как приступить к обработке изображения, на узлы Исполнителей следует загрузить выбранную сцену для рендеринга. Поскольку компьютерная сцена для приложения MegaPOV представляет собой обычный текстовый файл, то передавать ее можно в виде текстовой строки. Полученную строку необходимо сохранить на машине Исполнителя в некоторый временный файл,

полное имя которого формируется при помощи функции **`string FormatInput()`**. Полученный таким образом файл сцены будет передан приложению MegaPOV в качестве параметра.

Результат обработки части изображения также следует сохранить в некотором временном файле, полный путь к которому формируется при помощи функции **`string FormatOutput()`** и передается программе MegaPOV в качестве параметра.

Наконец, функция **`string FormatArguments()`** используется для формирования строки параметров для приложения MegaPOV, указывающих имя входного и выходного файла, ширину и высоту изображения, прямоугольную область для обработки, настройки сглаживания, а также формат выходного графического файла. Кроме того, у пользователя есть возможность указать дополнительные параметры, например, путь к необходимым файлам библиотек.

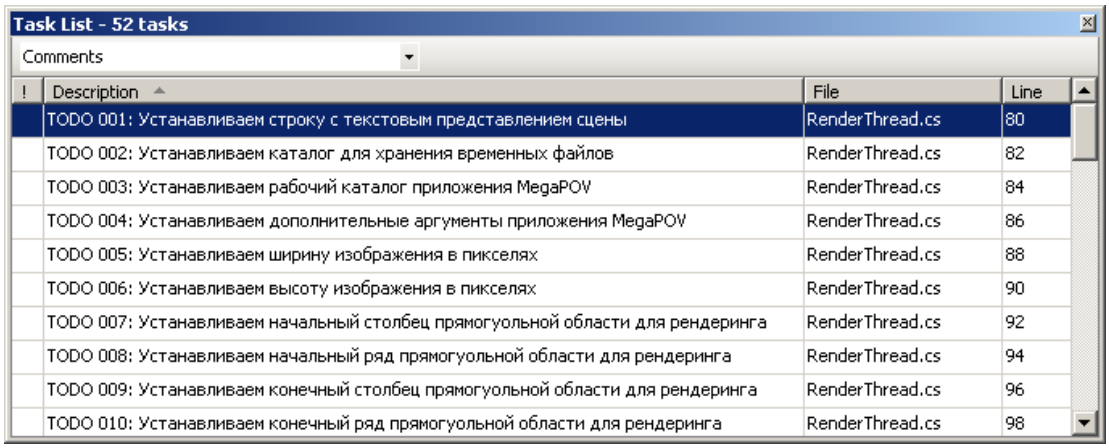


Рис. 7. Окно **Task List** позволяет осуществлять навигацию по назначенным заданиям

Перейдем теперь к доработке незавершенных фрагментов кода. Прежде всего, заметим, что каждое задание сопровождается комментарием, начинающимся со слова **`TODO`**. Все комментарии, начинающиеся с данного слова, среда Microsoft Visual Studio помечает специальным образом и заносит в список заданий. Для того чтобы просмотреть список заданий, нужно выполнить команду **View | Task List**. В результате на экране появится окно **Task List** (рис. 7), в котором будут перечислены задания. С помощью данного окна удобно осуществлять навигацию по программному коду, поскольку щелчок на каждом задании приводит к отображению соответствующего фрагмента. Чтобы убрать выполненное задание из списка достаточно исключить из комментария слово **`TODO`**.

Напомним, что класс грид-потока является производным от *абстрактного* класса **`GThread`**, в котором определяется базовая функциональность для всех грид-потоков. Кроме того, перед объявлением класса необходимо поместить атрибут **`Serializable`**, который указывает на то, что объект может быть отправлен по сети на удаленный компьютер и должен быть снабжен специальными методами *сериализации* (*serialization*) и *десериализации* (*deserialization*). В процессе сериализации состояние объекта преобразуется в форму, которая может быть сохранена или передана по сети. Обратным процессом, при котором некоторый поток данных преобразуется в объект, является десериализация. В совокупности процессы сериализации и десериализации позволяют легко сохранять и пересылать объекты.

Реализацию грид-потока необходимо начинать с определения всех переменных, которые могут потребоваться в процессе работы или для формирования итогового результата. Затем следует создать *конструктор* для класса грид-потока, в котором будет производиться инициализация данных переменных конкретными значениями и выделяться необходимые ресурсы. Для нашей задачи соответствующий фрагмент кода выглядит следующим образом:

```
/// <summary>
/// Создает новый экземпляр грид-потока для рендеринга части изображения.
/// </summary>
/// <param name="inputScene">строка с текстовым представлением сцены</param>
/// <param name="tempDirectory">временный каталог</param>
/// <param name="workDirectory">рабочий каталог</param>
/// <param name="width">ширина изображения</param>
/// <param name="height">высота изображения</param>
/// <param name="startRow">начальный ряд</param>
/// <param name="startCol">начальный столбец</param>
/// <param name="endRow">конечный ряд</param>
/// <param name="endCol">конечный столбец</param>
/// <param name="antialiasLevel">настройки сглаживания изображения</param>
```

```
public RenderThread(string inputScene, string tempDirectory, string workDirectory,
                    int width, int height,
                    int startRow, int startCol, int endRow, int endCol,
                    AntialisingLevel antialiasLevel, string additionalArguments)
{
    // Устанавливаем входной файл со сценой
    this.inputScene = inputScene;

    // Устанавливаем каталог для хранения временных файлов
    this.tempDirectory = tempDirectory;

    // Устанавливаем рабочий каталог приложения MegaPOV
    this.workDirectory = workDirectory;

    // Устанавливаем дополнительные аргументы приложения MegaPOV
    this.additionalArguments = additionalArguments;

    // Устанавливаем ширину и высоту изображения в пикселях
    this.width = width;
    this.height = height;

    // Устанавливаем границы прямоугольной области для рендеринга
    this.startCol = startCol;
    this.startRow = startRow;
    this.endCol = endCol;
    this.endRow = endRow;

    // Устанавливаем настройки сглаживания изображения
    this.antialiasLevel = antialiasLevel;

    // Создаем точечный рисунок для хранения части изображения
    this.image = new Bitmap(endCol - startCol, endRow - startRow);
}
```

Для тех переменных, которые потребуются для обработки завершенного грид-потока и формирования итогового результата, необходимо создать *свойства*. Свойства являются именованными членами класса и предлагают гибкий механизм для чтения, записи и вычисления значений частных полей класса. На приведенном ниже листинге отдельно показаны все свойства, объявленные в классе грид-потока **RenderThread**.

```
/// <summary>Сгенерированная часть общего изображения.</summary>
public Bitmap Image
{
    get
    {
        return image;
    }
}

/// <summary>Координата левого столбца пикселей.</summary>
public int StartCol
{
    get
    {
        return startCol;
    }
}

/// <summary>Координата верхней строки пикселей.</summary>
public int StartRow
{
    get
    {
        return startRow;
    }
}

/// <summary>Ширина сгенерированной части изображения.</summary>
public int Width
{
    get
    {
        return endCol - startCol;
    }
}

/// <summary>Высота сгенерированной части изображения.</summary>
public int Height
```

```
{
    get
    {
        return endRow - startRow;
    }
}
```

Теперь можно приступать к реализации *основного* метода грид-потока, в котором выполняются все необходимые действия на удаленном Исполнителе. Данный метод является обязательным для любого грид-потока, поэтому его объявление присутствует в абстрактном классе **`GThread`**, который является базовым для всех грид-потоков (комментарий к методу переведен на русский язык):

```
/// <summary>
/// Осуществляет исполнение грид-потока на удаленном Исполнителе. Данный
/// метод необходимо реализовывать в производных классах, добавляя
/// в него код, который должен выполняться на удаленных Ипсолнителях.
/// </summary>
public abstract void Start();
```

Таким образом, в каждой конкретной реализации класса грид-потока данный метод необходимо переопределять, заменяя ключевое слово **`abstract`** ключевым словом **`override`**. В нашем учебном примере реализация данного метода естественно разбивается на три части, связанные с сохранением входной сцены в файл, запуском приложения MegaPOV для обработки некоторой ее части и получением результатов работы. Последовательно опишем доработку всех трех фрагментов.

Для записи текстового представления сцены в файл можно воспользоваться стандартным классом **`StreamWriter`**, который является производным от абстрактного класса **`TextWriter`** и предназначен для записи некоторой последовательности символов в файл. Ниже показан соответствующий фрагмент кода:

```
// Сохраняем сцену для рендеринга во временном файле
{
    try
    {
        // Создаем объект StreamWriter для записи текстового файла
        StreamWriter streamWriter = File.CreateText(FormatInput());

        // Сохраняем полученную сцену в файле
        streamWriter.Write(inputScene);

        // Закрываем объект StreamWriter и все связанные с ним ресурсы
        streamWriter.Close();
    }
    catch (Exception e)
    {
        // Сцену не удалось сохранить в файл
        throw new Exception("Error! Can not write a scene to the file.", e);
    }
}
```

В результате выполнения данного кода во временной папке, указанной пользователем, появится файл сцены с уникальным именем, сгенерированным при помощи метода **`string FormatInput()`**. В дальнейшем данный файл будет передан программе MegaPOV в качестве параметра.

Далее следует запустить “внешнее” приложение MegaPOV для обработки заданной части сцены. В инструментарии Microsoft .NET Framework для этой цели предусмотрен специальный класс **`Process`**. Объект класса **`Process`** оказывается чрезвычайно полезным для запуска, остановки, контроля и мониторинга приложений. С помощью данного объекта можно легко получить список активных процессов на компьютере или запустить новое приложение. Следует заметить, что данный объект позволяет также получать информацию об активных процессах. К такой информации относится, например, число запущенных потоков, загруженные модули (файлы динамических библиотек и программ), а также информация о производительности, например, количество используемой оперативной памяти. Ниже показано использование объекта **`Process`** для нашей задачи:

```
// Генерируем заданную часть изображения
{
    // Создаем объект Process для управления локальными процессами
    Process process = new Process();

    // Задаем приложение MegaPOV для запуска
    process.StartInfo.FileName = Path.Combine(workDirectory, "bin/megapov");

    // Задаем аргументы приложения MegaPOV
    process.StartInfo.Arguments = FormatArguments();

    // Скрываем окно приложения MegaPOV
```

```
    process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;

    try
    {
        // Запускаем приложение MegaPOV
        process.Start();

        // Ожидаем завершения работы приложения MegaPOV
        process.WaitForExit();
    }
    catch (Exception e)
    {
        // Приложение MegaPOV не удалось корректно запустить
        throw new Exception("Error! Can not start MegaPOV application.", e);
    }
}
```

При успешном исполнении всех указанных действий во временной папке появится выходной графический файл с уникальным именем, сгенерированным при помощи функции **`string FormatOutput()`**, содержащий обработанную часть изображения.

Далее следует открыть сгенерированный графический файл и извлечь из него полезную часть. Выполнить данное действие можно, например, следующим образом:

```
// Получаем сгенерированное изображение из файла
{
    try
    {
        // Загружаем изображение из файла
        Image output = Bitmap.FromFile(FormatOutput());

        // Создаем объект Graphics для работы с изображением
        Graphics graphics = Graphics.FromImage(image);

        // Вырезаем из загруженного изображения полезную часть
        graphics.DrawImage(output, 0, 0,
                           new Rectangle(startCol, 0, endCol - startCol, endRow - startRow),
                           GraphicsUnit.Pixel);
    }
    catch (Exception e)
    {
        // Графический файл не удалось корректно обработать
        throw new Exception("Error! Can not process output file.", e);
    }
}
```

При успешном выполнении всех этих действий из файла будет загружена обработанная часть изображения. В результате грид-поток завершит свою работу, и инструментарий Alchemi вызовет обработчик события **`ThreadFinish`**.

Заметим, что в реализованном методе может генерироваться большое число исключений, связанных с ошибками чтения и записи, доступом к памяти, безопасностью и т. д. Чтобы избежать абстрактных сообщений об ошибке, мы сгруппировали потенциально опасные фрагменты кода при помощи блоков **`try-catch`** и сгенерировали для каждого такого блока общее исключение, сообщение о котором и будет выводиться пользователю в случае ошибки. Информация об исходном исключении, тем не менее, не теряется, а передается новому исключению в качестве параметра.

При возникновении исключения в методе **`void Start()`** грид-потока инструментарий Alchemi объявляет его неудавшимся (failed) и вызывает обработчик события **`ThreadFailed`**. В данном обработчике можно проанализировать ошибку и принять решение о дальнейших действиях. В простейшем случае о случившейся ошибке необходимо проинформировать пользователя. Однако наилучшим выходом из ситуации является группировка всех неудавшихся грид-потоков и их перезапуск на доступных Исполнителях после того, как грид-приложение завершит свою работу.

На этом доработку класса грид-потока можно считать завершенной. Теперь необходимо произвести тестирование кода, запустив грид-потоки на локальной машине.

5.3. Задание 3 – Разработка класса локального приложения

Формирование грид-потоков и организация их исполнения осуществляется в основном приложении. Для того чтобы открыть заготовку кода, предназначенную для дальнейшей доработки, выберите в окне **Solution Explorer** проект **DistributedMegaPOV**, затем правой кнопкой мышки щелкните на файле **LocalApplication.cs** и в появившемся контекстном меню выберите команду **View Code**.

Для более удобной обработки грид-потоков на локальной машине лучше всего создать вспомогательный класс **LocalAppication**, семантически близкий классу **GApplication**, предназначенному для запуска грид-потоков в вычислительной грид. Данный класс является универсальным и может быть использован для тестирования грид-потоков в любом приложении. Дополнительная выгода от использования данного класса состоит в том, что для запуска грид-потоков в вычислительной грид достаточно будет изменить всего несколько строк кода.

Ознакомимся подробнее с классом **LocalApplication**, заготовка для которого показана на приведенном ниже фрагменте программного кода.

```
using System;
using System.Threading;
using Alchemi.Core.Owner;

/// <summary>
/// Обеспечивает запуск грид-потоков на локальной машине.
/// </summary>
public class LocalApplication
{
    /// <summary>Обработчик успешно завершившихся грид-потоков.</summary>
    private GThreadFinish threadFinish;

    /// <summary>Обработчик неудачно завершившихся грид-потоков.</summary>
    private GThreadFailed threadFailed;

    /// <summary>Коллекция грид-потоков для запуска на локальной машине.</summary>
    private ThreadCollection threadCollection;

    /// <summary>Идентификатор обрабатываемого грид-потока.</summary>
    private int threadID;

    /// <summary>
    /// Создает новое локальное приложение.
    /// </summary>
    public LocalApplication()
    {
        ...
    }

    /// <summary>
    /// Последовательно выполняет грид-потоки на локальной машине.
    /// </summary>
    private void ExecuteThreads()
    {
        ...
    }

    /// <summary>
    /// Запускает локальное приложение в отдельном потоке.
    /// </summary>
    public void Start()
    {
        ...
    }

    /// <summary>Коллекция грид-потоков для запуска на локальной машине.</summary>
    public ThreadCollection Threads
    {
        get
        {
            return threadCollection;
        }

        set
        {
            threadCollection = value;
        }
    }

    /// <summary>Обработчик успешно завершившихся грид-потоков.</summary>
    public GThreadFinish ThreadFinish
    {
        get
        {
            return threadFinish;
        }

        set
```

```
        {
            threadFinish = value;
        }
    }

    /// <summary>Обработчик неудачно завершившихся грид-потоков.</summary>
    public GThreadFailed ThreadFailed
    {
        get
        {
            return threadFailed;
        }

        set
        {
            threadFailed = value;
        }
    }
}
```

Как уже было сказано выше, основное предназначение класса – упрощение обработки грид-потоков на локальной машине пользователя. Работа с данным классом чрезвычайно проста. Сначала следует указать обработчики событий **`ThreadFinish`** и **`ThreadFailed`**, возникающие при успешном и неудачном завершении грид-потока соответственно, затем к свойству **`Threads`** локального приложения необходимо добавить некоторое число грид-потоков, и, наконец, локальное приложение нужно запустить с помощью метода **`void Start()`**. Рассмотрим основные моменты реализации класса.

В первую очередь, необходимо реализовать конструктор класса, действия которого сводятся к созданию нового экземпляра коллекции грид-потоков. Соответствующий фрагмент кода имеет вид:

```
/// <summary>
/// Создает новое локальное приложение.
/// </summary>
public LocalApplication()
{
    // Создаем новую коллекцию грид-потоков
    threadCollection = new ThreadCollection();
}
```

Затем следует реализовать метод **`void ExecuteThreads()`**, который последовательно выполняет грид-потоки на локальной машине. Если грид-поток *успешно* завершает свою работу, то вызывается обработчик события **`ThreadFinish`**. Если же в процессе работы грид-потока возникло исключение, то его работа заканчивается *неудачно* и вызывается обработчик события **`ThreadFailed`**. Обработанные грид-потоки удаляются из коллекции. Именно таким образом ведет себя класс **`GApplication`**. На приведенном ниже листинге показана возможная реализация метода:

```
/// <summary>
/// Последовательно выполняет грид-потоки на локальной машине.
/// </summary>
private void ExecuteThreads()
{
    foreach (GThread thread in threadCollection)
    {
        // Устанавливаем идентификатор грид-потока
        thread.SetId(threadID++);

        try
        {
            // Запускаем грид-поток на выполнение
            thread.Start();
        }
        catch (Exception e)
        {
            // Запускаем обработчик неудачно завершившихся грид-потоков
            if (threadFailed != null)
            {
                threadFailed(thread, e);
            }
        }

        // Запускаем обработчик успешно завершившихся грид-потоков
        if (threadFinish != null)
        {
            threadFinish(thread);
        }
    }
```

```
    // Удаляем обработанные грид-потоки из коллекции
    threadCollection.Clear();
}
```

Обратите внимание, что перед запуском грид-потока ему присваивается уникальный идентификатор – его порядковый номер. Это действие является необходимым, так как идентификатор грид-потока удобно использовать при генерации имени входного и выходного файла программы MegaPOV.

Наконец, можно реализовать основной метод класса **`void Start()`**, который запускает локальное приложение. Соответствующий фрагмент кода имеет вид:

```
/// <summary>
/// Запускает локальное приложение в отдельном потоке.
/// </summary>
public void Start()
{
    // Создаем отдельный поток для выполнения локального приложения
    Thread thread = new Thread(new ThreadStart(ExecuteThreads));

    // Запускаем созданный поток
    thread.Start();
}
```

Обратите внимание, что для обработки грид-потоков на локальной машине используется *отдельный поток*, что позволяет добиться лучшей “отзывчивости” интерфейса приложения. Это особенно актуально для нашей задачи, поскольку исполнение грид-потоков может длиться значительное время (в зависимости от сложности сцены от нескольких секунд до нескольких часов и даже суток).

На этом разработка класса локального приложения завершена, и можно переходить к запуску грид-потоков на локальной машине.

5.4. Задание 4 – Запуск грид-потоков на локальной машине

Запуск грид-потоков осуществляется в основном приложении. Для того чтобы открыть заготовку кода, предназначенную для дальнейшей доработки, выберите в окне **Solution Explorer** проект **DistributedMegaPOV**, затем правой кнопкой мышки щелкните на файле **MainForm.cs** и в появившемся контекстном меню выберите команду **View Code**.

В первую очередь необходимо объявить переменные, которые потребуются во время работы. На приведенном ниже листинге показаны только те переменные, которые потребуются для запуска грид-потоков на *локальной* машине. Смысл данных переменных должен быть ясен из соответствующих комментариев. Обратите также внимание на библиотеки, подключаемые в начале файла.

```
using System;
using System.IO;
using System.Data;
using System.Drawing;
using System.Threading;
using System.Configuration;
using System.Windows.Forms;
using Alchemi.Core.Owner;
using GridThread;

/// <summary>
/// Главная форма приложения.
/// </summary>
public partial class MainForm : Form
{
    ...

    /// <summary>Рабочий каталог приложения MegaPOV.</summary>
    private string workDirectory;

    /// <summary>Каталог для хранения временных файлов.</summary>
    private string tempDirectory;

    /// <summary>Дополнительные аргументы приложения MegaPOV.</summary>
    private string additionalArguments;

    /// <summary>Файл сцены для рендеринга.</summary>
    private string inputScene;

    /// <summary>Точечный рисунок для хранения изображения.</summary>
    private Bitmap image;
```

```
    /// <summary>Время начала рендеринга изображения.</summary>
    private DateTime startTime;

    /// <summary>Локальное приложение для запуска грид-потоков на локальной машине.</summary>
    private LocalApplication localApplication;

    /// <summary>Отвечает за инициализацию локального приложения.</summary>
    private bool localInit;

     ...
}
```

В этом задании мы рассмотрим реализацию метода **`void RenderImageLocal()`**, в котором осуществляется рендеринг изображения на локальном компьютере. Действия данного метода сводятся к инициализации экземпляра локального приложения, получению значений входных переменных, формированию грид-потоков для обработки частей изображения и их запуску с помощью класса локального приложения. Опишем основные этапы реализации данного метода подробнее.

При первом запуске локального приложения необходимо произвести его инициализацию, то есть необходимо создать новый экземпляр локального приложения и назначить обработчики событий **`ThreadFinish`** и **`ThreadFailed`**, возникающие соответственно при успешном и неудачном завершении грид-потока. Соответствующий фрагмент кода имеет вид:

```
// Проверяем, требуется ли инициализация локального приложения
 if (!localInit)
{
    // Создаем локальное приложение
    localApplication = new LocalApplication();

    // Добавляем обработчик успешно завершившихся грид-потоков
    localApplication.ThreadFinish += new GThreadFinish(ThreadFinish);

    // Добавляем обработчик неудачно завершившихся грид-потоков
    localApplication.ThreadFailed += new GThreadFailed(ThreadFalied);

    // Инициализация завершена
    localInit = true;
}
```

Далее необходимо установить значения всех переменных, которые потребуются для формирования грид-потоков. В нашем случае это ширина и высота изображения, количество его горизонтальных и вертикальных разбиений, текстовая строка со сценой для рендеринга, ширина и высота обрабатываемых частей изображения, а также настройки приложения MegaPOV. Значения большинства переменных считываются из элементов управления главного окна приложения. На приведенном ниже листинге показан соответствующий фрагмент кода:

```
// Устанавливаем число горизонтальных и вертикальных разбиений
int hor = (int)spinnerHorCells.Value;
int ver = (int)spinnerVerCells.Value;

// Устанавливаем ширину и высоту изображения в пикселях
int imagewidth = (int)spinnerWidth.Value;
int imageheight = (int)spinnerHeight.Value;

// Вычисляем ширину и высоту ячеек изображения в пикселях
int cellwidth = imagewidth / hor;
int cellheight = imageheight / ver;

// Создаем точечный рисунок для хранения изображения
image = new Bitmap(imagewidth, imageheight);

// Отображаем точечный рисунок на графическом элементе PictureBox
pictureBoxImage.Image = image;

// Устанавливаем настройки сглаживания изображения
AntialisingLevel antialiasLevel = (AntialisingLevel)comboBoxAntialias.SelectedIndex;

// Проверяем, загружен ли файл сцены
if (inputScene == null)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Please load scene file.");

    // Выходим из процедуры
    return;
}
```

Заметим, что дополнительные параметры, необходимые для работы программы MegaPOV, такие как рабочий каталог приложения, каталог для хранения временных файлов и дополнительные опции хранятся в *конфигурационном файле* приложения Distributed MegaPOV. Конфигурационные файлы (configuration files) представляют собой стандартный механизм инструментария Microsoft .NET Framework и являются обычными документами в формате XML, которые могут быть при необходимости изменены. Разработчики могут использовать конфигурационные файлы для того, чтобы изменять настройки приложений без необходимости перекомпиляции. Имя конфигурационного файла приложения обычно строится по формуле **<имя приложения>.exe.config**. В нашем случае конфигурационный файл имеет имя **Distributed MegaPOV.exe.config** и хранится в корневом каталоге приложения Distributed MegaPOV. Структура файла показана на приведенном ниже листинге:

```
<configuration>
  <configSections>
  </configSections>
  <appSettings>
    <add key="Work Directory" value="C:/Program Files/POV-Ray for Windows v3.6" />
    <add key="Temp Directory" value="C:/Temp" />
    <add key="Input File" value="C:/Scenes/woodbox.pov" />
    <add key="Additional Arguments" value="" />
  </appSettings>
</configuration>
```

Использование конфигурационных файлов значительно облегчает работу с приложением, так как при каждом его запуске не приходится заново задавать путь к рабочему каталогу программы MegaPOV, каталогу временных файлов и ряд других настроек. Если при переносе на другой компьютер данные настройки изменились, то достаточно исправить конфигурационный файл, и новые параметры будут автоматически считываться при запуске программы. Считывание параметров из конфигурационного файла можно осуществлять в конструкторе главного окна приложения **`public MainForm()`**. Соответствующий фрагмент кода приведен ниже:

```
// Загружаем настройки приложения из файла
workDirectory = ConfigurationSettings.AppSettings["Work Directory"];
tempDirectory = ConfigurationSettings.AppSettings["Temp Directory"];
additionalArguments = ConfigurationSettings.AppSettings["Additional Arguments"];

// Загружаем файл сцены по умолчанию
LoadFile(ConfigurationSettings.AppSettings["Input File"]);
```

Теперь, когда установлены значения всех необходимых переменных, можно приступать к формированию грид-потоков. Для этого по всем горизонтальным и вертикальным разбиениям нужно организовать двойной цикл, внутри которого следует создать грид-поток и добавить его к локальному приложению. Возможная реализация может выглядеть следующим образом:

```
// Формируем грид-потоки для обработки частей изображения
for (int hornumber = 0; hornumber < hor; hornumber++)
{
    // Вычисляем координату левого ряда пикселей
    int startcol = hornumber * cellwidth;

    for (int vernumber = 0; vernumber < ver; vernumber++)
    {
        // Вычисляем кординату верхнего ряда пикселей
        int startrow = vernumber * cellheight;

        // Создаем грид-поток для обработки части изображения
        RenderThread thread = new RenderThread(inputScene, tempDirectory, workDirectory,
                                               imagewidth, imageheight, startrow, startcol,
                                               startrow + cellheight, startcol + cellwidth,
                                               antialiasLevel, additionalArguments);

        // Добавляем грид-поток к локальному приложению
        localApplication.Threads.Add(thread);
    }
}
```

Прежде чем запускать локальное приложение, следует сохранить время старта вычислений и установить текущее и максимальное значение индикатора хода выполнения (**`ProgressBar`**), равное общему числу сформированных грид-потоков. Данные действия позволят нам в процессе вычислений отслеживать прогресс и затраченное время. Соответствующий фрагмент кода имеет вид:

```
// Обновляем текущее и максимальное значение полосы прогресса
progressBarRenderProgress.Value = 0;
progressBarRenderProgress.Maximum = hor * ver;
```

```
// Сохраняем время начала вычислений
startTime = DateTime.Now;
```

Наконец, все готово для запуска локального приложения. Поскольку в процессе обработки грид-потоков могут возникать исключения, данный код следует заключить в блок **try-catch**:

```
try
{
    // Запускаем локальное приложение
    localApplication.Start();
}
catch (Exception e)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Can not start local application: " + e.ToString());
}
```

В заключение этого задания мы рассмотрим реализацию обработчиков событий **ThreadFinish** и **ThreadFailed**.

Напомним, что все необходимые сервисные операции инструментарий Alchemi выполняет в *отдельном* рабочем потоке, периодически посылая уведомления в *основной* поток приложения. Уведомления могут посылаться при успешном или неудачном завершении грид-потока, а также при завершении работы грид-приложения. Трудность состоит в том, что библиотека пользовательского интерфейса Windows Forms работает с окном и его элементами управления в специально выделенном *основном* потоке. Иными словами, из “внешнего” потока невозможно получить доступ к окну и его элементам управления. Тем не менее, для нормальной работы программы это необходимо. В рассматриваемом примере программа должная получать уведомления (обрабатывать события) об успешном завершении некоторого грид-потока и выводить в главное окно обновленную картинку и значение индикатора хода выполнения. Таким образом, для реализации взаимодействия необходимо использовать средства синхронизации, предоставляемые данной библиотекой. В библиотеке Windows Forms одним из таких средств является метод базового класса **Control**:

```
public virtual IAsyncResult BeginInvoke(Delegate, object[]);
```

Данный метод вызывает заданный делегат с определенным набором параметров из *основного* потока приложения. Как видно из объявления метода, для нормального использования механизма синхронизации необходимо для каждого уведомления определить свой тип делегата и делать достаточно неявные вызовы, что является некоторым неудобством библиотеки Windows Forms.

В нашем случае требуется обрабатывать уведомление об успешном завершении грид-потока. Воспользуемся для этой цели методом **void UpdateProgress(GThread thread)**, реализация которого будет рассмотрена ниже. Поскольку данный метод должен вызываться из внешнего потока, для него следует создать специальный делегат и объявить экземпляр этого делегата:

```
/// <summary>
/// Делегат для обновления прогресса обработки изображения.
/// </summary>
/// <param name="thread">успешно завершенный грид-поток</param>
private delegate void UpdateProgressDelegate(GThread thread);

/// <summary>Экземпляр делегата для обновления прогресса обработки изображения.</summary>
private UpdateProgressDelegate updateProgressDelegate;
```

Экземпляр делегата для обновления прогресса вычислений инициализируется в конструкторе главного окна приложения **public MainForm()**:

```
// Создаем делегат для обновления прогресса вычислений
updateProgressDelegate = new UpdateProgressDelegate(UpdateProgress);
```

Действия обработчика события **ThreadFinish** сводятся, таким образом, к вызову метода **void UpdateProgress(GThread thread)** из *основного* потока приложения:

```
/// <summary>
/// Обработчик события, возникающего при успешном завершении грид-потока.
/// </summary>
/// <param name="thread">успешно завершенный грид-поток</param>
void ThreadFinish(GThread thread)
{
    // Вызываем делегат для обновления прогресса обработки изображения
    BeginInvoke(updateProgressDelegate, new object[] { thread });
}
```

Перейдем к рассмотрению самого метода обновления прогресса обработки **void UpdateProgress(GThread thread)**. В данном методе обновляется время рендеринга, значение

индикатора хода вычислений и добавляется очередной обработанный участок на общее изображение. На приведенном ниже листинге показан код обновления затраченного времени и значения индикатора хода вычислений:

```
// Обновляем время рендеринга
labelRenderTime.Text = "Render Time: " + (DateTime.Now - startTime);

// Обновляем полосу прогресса
if (progressBarRenderProgress.Value < progressBarRenderProgress.Maximum)
{
    progressBarRenderProgress.Value++;
}
```

Для копирования очередного обработанного участка воспользуемся методом объекта **Graphics**, после чего принудительно обновим элемент управления **PictureBox**:

```
// Получаем успешно завершившийся грид-поток
RenderThread renderThread = (RenderThread)thread;

// Создаем экземпляр объекта Graphics для работы с изображением
Graphics graphics = Graphics.FromImage(image);

// Копируем сгенерированную часть изображения
graphics.DrawImage(renderThread.Image, renderThread.StartCol, renderThread.StartRow);

// Обновляем изображение
pictureBoxImage.Invalidate();
```

В обработчике события **ThreadFailed** ограничимся выводом сообщения о неудачном завершении грид-потока и не будем предпринимать каких-либо действий для исправления ситуации. Поскольку в таком обработчике не требуется получать доступ к элементам управления главного окна, то создавать новый делегат необязательно. Соответствующий фрагмент кода приведен ниже:

```
/// <summary>
/// Обработчик события, возникающего при неудачном завершении грид-потока.
/// </summary>
/// <param name="thread">неудачно завершившийся грид-поток</param>
void ThreadFalied(GThread thread, Exception e)
{
    MessageBox.Show("Thread with Id = " + thread.Id + " failed: " + e.ToString());
}
```

Это простейшее поведение программы, однако, правильнее было бы повторно перезапускать неудачно завершившиеся грид-потоки на доступных Исполнителях. Возможность улучшить обработку неудачно завершившихся грид-потоков предоставляется выполнить самостоятельно.

5.5. Задание 5 – Запуск приложения на локальной машине

Теперь приложение можно запустить и проверить работоспособность вычислительного кода грид-потоков. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 8 показано главное окно приложения.

Для устранения потенциальных проблем в первую очередь следует проверить корректность настроек приложения, таких как пути к рабочему каталогу приложения MegaPOV и каталогу временных файлов. Для этого выполните команду **File | Path Settings**, в результате которой на экране появится диалоговое окно с текущими настройками (рис. 9). При необходимости настройки можно занести в конфигурационный файл, так что при следующем запуске программы они будут загружены автоматически.

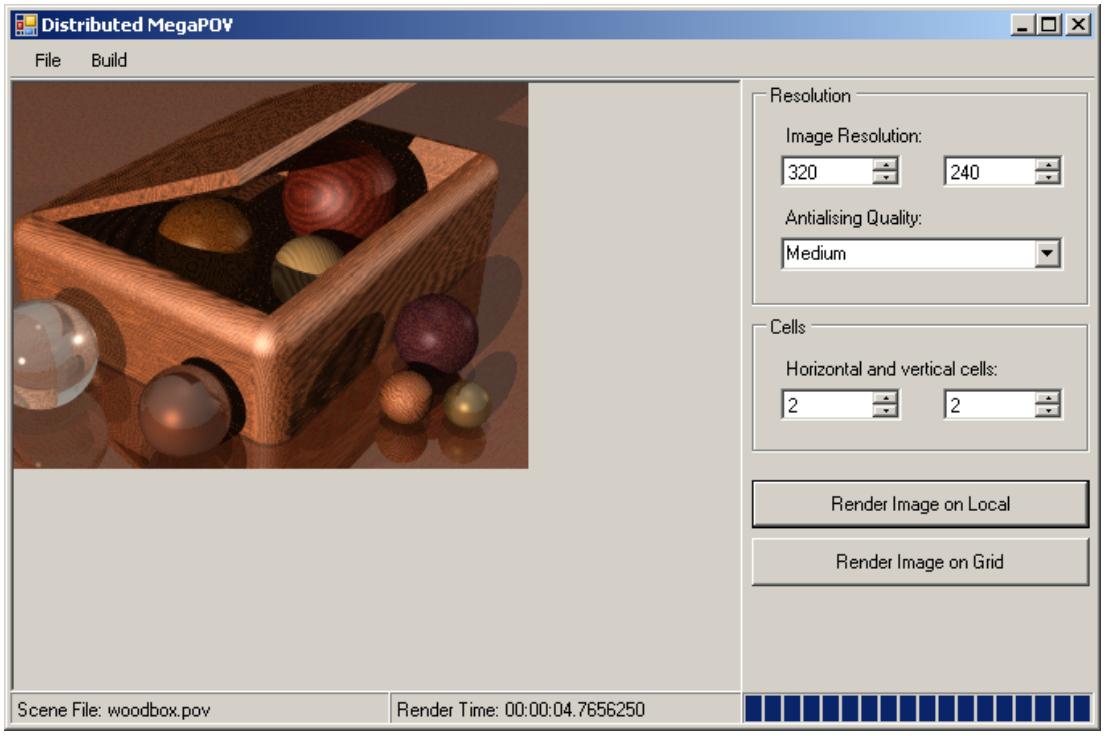


Рис. 8. Главное окно приложения после первого запуска

Убедившись в корректности настроек и внеся при необходимости нужные изменения, выберите файл сцены для рендеринга с помощью команды **File | Open Scene**. Напомним, что множество готовых сцен хранятся в подкаталоге **scenes** установочного каталога программы POV-Ray. Для обработки сцены на локальной машине щелкните на кнопке **Render Image on Local**. В результате на кране появится изображение сгенерированной сцены. Среди доступных сцен есть весьма интересные и красивые, но следует иметь в виду, что генерация некоторых сцен может занимать значительное время.

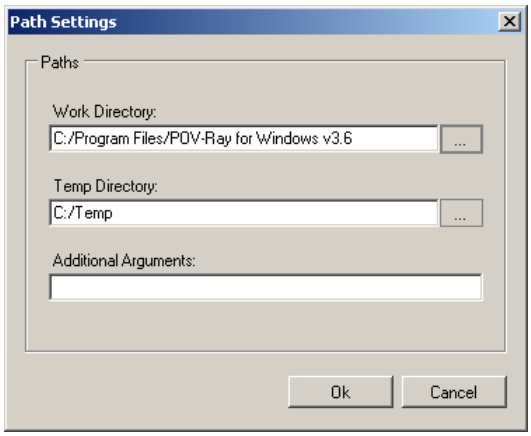


Рис. 9. Диалоговое окно с настройками приложения Distributed MegaPOV

Однако наша цель состоит в том, чтобы запускать грид-потоки в вычислительной грид. В следующем задании мы соответствующим образом доработаем программный код.

5.6. Задание 6 – Запуск грид-потоков в вычислительной грид

Запуск грид-потоков в вычислительной грид потребует минимальной доработки кода. В первую очередь, следует объявить дополнительные переменные, которые потребуются во время работы:

```
/// <summary>Грид-приложение для запуска грид-потоков в грид.</summary>
private GApplication gridApplication;

/// <summary>Отвечает за инициализацию грид-приложения.</summary>
private bool gridInit;
```

В этом задании нам предстоит доработать метод **`void RenderImageGrid()`**, в котором осуществляется запуск грид-потоков в вычислительной грид. Заметим, что реализация данного метода очень близка к реализации соответствующего метода для запуска грид-потоков на локальной машине. Мы рассмотрим только изменившиеся фрагменты кода, так как остальная его часть была рассмотрена ранее.

Подобно локальному приложению, при первом запуске грид-приложение требует инициализации. На этом этапе необходимо создать новый экземпляр грид-приложения, установить подключение к грид, добавить зависимости, необходимые для запуска грид-потоков на удаленных Исполнителях, и назначить обработчики событий **`ThreadFinish`** и **`ThreadFailed`**. На приведенном ниже листинге показан соответствующий фрагмент кода:

```
// Проверяем, требуется ли инициализация грид-приложения
if (!gridInit)
{
    // Создаем диалог подключения к грид
    GConnectionDialog connectionDialog = new GConnectionDialog();

    // Отображаем диалог подключения к грид
    if (connectionDialog.ShowDialog() == DialogResult.OK)
    {
        // Создаем новое грид-приложение многократного использования
        gridApplication = new GApplication(true);

        // Устанавливаем имя созданного грид-приложения
        gridApplication.ApplicationName = "Distributed MegaPOV - Alchemi sample";

        // Устанавлливаем соединение созданного грид-приложения
        gridApplication.Connection = connectionDialog.Connection;
    }

    // Добавляем зависимости, необходимые для запуска грид-потоков
    gridApplication.Manifest.Add(new
                            ModuleDependency(typeof(GridThread.RenderThread).Module));

    // Добавляем событие для обработки успешно завершившихся грид-потоков
    gridApplication.ThreadFinish += new GThreadFinish(ThreadFinish);

    // Добавляем событие для обработки неудачно завершившихся грид-потоков
    gridApplication.ThreadFailed += new GThreadFailed(ThreadFailed);

    // Инициализация грид-приложения завершена
    gridInit = true;
}
```

Затем следует проверить, работает ли в данный момент грид-приложение. Если это так, то грид-приложение необходимо корректно остановить. Следует иметь в виду, что данное действие может генерировать ряд исключений, поэтому данный фрагмент кода нужно заключить в блок **`try-catch`**:

```
// Проверяем, работает ли грид-приложение
if (gridApplication.Running)
{
    try
    {
        // Останавливаем работающее грид-приложение
        gridApplication.Stop();
    }
    catch (Exception e)
    {
        // Выводим сообщение об ошибке
        MessageBox.Show("Can not stop already running grid application: " + e.ToString());
    }
}
```

Дальнейшие действия состоят в получении значений всех переменных, необходимых для работы, и формировании грид-потоков. Поскольку данный код полностью повторяет ранее рассмотренный, мы не будем подробно на этом останавливаться. Отличие от предыдущей реализации состоит в том, что на этот раз грид-потоки добавляются к грид-приложению. Следующий фрагмент кода иллюстрирует данный факт:

```
// Формируем грид-потоки для обработки частей изображения
for (int hornumber = 0; hornumber < hor; hornumber++)
{
    // Вычисляем координату левого ряда пикселей
    int startcol = hornumber * cellwidth;
```

```
    for (int vernumber = 0; vernumber < ver; vernumber++)
    {
        // Вычисляем кординату верхнего ряда пикселей
        int startrow = vernumber * cellheight;

        // Создаем грид-поток
        RenderThread thread =
                    new RenderThread(inputScene, tempDirectory, workDirectory,
                                     imagewidth, imageheight, startrow, startcol,
                                     startrow + cellheight, startcol + cellwidth,
                                     antialiasLevel, additionalArguments);

        // Добавляем грид-поток к грид-приложению
        gridApplication.Threads.Add(thread);
    }
}
```

Непосредственно перед запуском грид-приложения необходимо сохранить текущее время, чтобы впоследствии вычислить время обработки изображения, и установить текущее и максимальное значение индикатора хода выполнения программы. Наконец, грид-приложение нужно запустить. Поскольку данное действия может привести к потенциальным исключениям, его следует заключить в блок **`try-catch`**. Соответствующий фрагмент кода имеет вид:

```
try
{
    // Запускаем грид-приложение на выполнение
    gridApplication.Start();
}
catch (Exception e)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Error trying to run grid application: " + e.ToString());
}
```

Доработка метода завершена. Заметим, что при выходе из программы объект грид-приложения желательно удалить. Данный код можно поместить, например, в обработчик закрытия главного окна приложения:

```
// Уничтожаем грид-приложение
if (gridApplication != null)
{
    gridApplication.Dispose();
}
```

В следующем задании мы протестируем работу грид-потоков в вычислительной грид, построенной на базе инструментария Alchemi.

5.7. Задание 6 – Запуск приложения в вычислительной грид

Теперь приложение можно запустить и протестировать его работу в вычислительной грид, построенной на базе инструментария Alchemi. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 8 показано главное окно приложения.

Перед тем, как работать с программой, на машине разработчика следует создать минимальную “грид” из одного Менеджера и одного Исполнителя. Следует заметить, что при наличии многоядерной или многопроцессорной машины желательно запускать несколько Исполнителей для более эффективного использования вычислительных ресурсов.

Убедившись в корректности настроек и внеся при необходимости нужные изменения, выберите файл сцены для рендеринга с помощью команды **File | Open Scene**. Для обработки сцены в вычислительной грид щелкните на кнопке **Render Image on Grid**. При первом запуске вычислений на экране появится диалоговое окно подключения к грид, в котором следует задать такие параметры, как адрес и порт Менеджера, к которому осуществляется подключение, а также имя пользователя и пароль. После успешного подключения к грид начнется расчет изображения.

6. Заключение

В данной лабораторной работе рассматривался вопрос внедрения в грид существующего приложения на примере инструментария Alchemi. В качестве программы для внедрения был выбран

известный пакет для рендеринга компьютерных сцен POV-Ray с предустановленной надстройкой MegaPOV.

Следует заключить, что инструментарий Alchemi предлагает хорошие возможности для внедрения существующих приложений в грид, которые вместе с тем являются довольно прозрачными в сравнении, например, с инструментарием GPE. Тем не менее, процесс внедрения приложений в грид с помощью модели грид-потоков является низкоуровневым, т.к. требует разработки Во многом это обеспечивается благодаря развитой платформе Microsoft .NET Framework, на которой базируется инструментарий. Говоря о достоинствах и недостатках инструментария в контексте внедрения в грид существующих приложений, следует практически дословно повторить выводы предыдущей лабораторной работы. Однако имеются и некоторые отличия, связанные с различными подходами к использованию инструментария.

- *Простота реализации и компактность кода.*

  Обеспечивается благодаря развитой архитектуре высокоуровневых классов Microsoft .NET Framework. У разработчика имеются в распоряжении высокоуровневые инструменты для выполнения практически любых сервисных операций, связанных с чтением и записью файлов, запуском приложений и контролем их исполнения и т.д.

- *Возможность быстро и удобно проектировать пользовательский интерфейс.*

  Инструментарий Alchemi использует все возможности библиотеки пользовательского интерфейса Windows Forms.

- *Легкость в отладке и использовании приложений.*

  Разработанные классы грид-потоков можно сначала отладить на локальной машине, и лишь затем осуществлять их запуск в грид.

- *Возможность запуска разработанных приложений на локальных компьютерах.*

  Разработанные приложения можно запускать на локальных компьютерах без установленных компонент инструментария Alchemi. Любая программа для инструментария Alchemi может использоваться как обычное приложение для операционной системы Microsoft Windows с установленной платформой Microsoft .NET Framework (при этом, конечно, приложение не получит доступ к ресурсам грид).

Однако при использовании инструментария Alchemi можно обнаружить некоторые недостатки. Отметим на наш взгляд наиболее критичные из них:

- *Отсутствие переносимости кода и ориентация на единственную платформу.*

  К сожалению, на сегодняшний день отсутствует полноценная замена платформы Microsoft .NET Framework для других операционных систем. Данный факт делает невозможным запуск программ, написанных с использованием инструментария Alchemi, под другими операционными системами. Стоит заметить, что прогресс в этом направлении не стоит на месте, и межплатформенная разработка Mono[5] уже сейчас предлагает неплохой уровень совместимости с платформой Microsoft .NET Framework.

- *Слабая масштабируемость инструментария.*

  Инструментарий Alchemi лучше всего подходит для объединения в вычислительную сеть нескольких десятков компьютеров. При подключении большего числа компьютеров могут появиться проблемы со стабильностью системы (в частности, со стабильностью работы Менеджера). Улучшить масштабируемость позволяет организация иерархической структуры из отдельных Менеджеров. Опять же, это скорее особенность, которую следует учитывать, а не недостаток инструментария Alchemi.

7. Вопросы

1. В чем на ваш взгляд состоит проблема наполнения грид приложениями?
2. Каким требованиям должен обладать инструментарий для успешного решения данной проблемы?
3. В чем состоит процесс внедрения в грид некоторого приложения с помощью инструментария Alchemi?
4. Назовите основные этапы данного процесса.

[5] Главной задачей проекта Mono (http://www.mono-project.com) является создание полноценной реализации среды разработки Microsoft .NET Framework для UNIX-подобных операционных систем. Инициативу Mono возглавляет Мигель де Иказа, один из участников проекта Gnome. Спонсирует же группу разработчиков Mono корпорация Novell.

5. Каким требованиям должно отвечать приложение для внедрения в грид?

8. Упражнения

Разработанную программу можно значительно усовершенствовать, добавив в нее ряд новых возможностей.

1. Добавить возможность остановки вычислений при их запуске как на локальном компьютере, так и в вычислительной грид.
2. Реализовать корректную обработку неудачно завершившихся грид-потоков и последующий их перезапуск на доступных Исполнителях.

9. Литература

1. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики. – СПб.: БХВ-Петербург, 2003. – 560 с.
2. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. http://www.gridbus.org/~alchemi/files/alchemi_bookchapter.pdf
3. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Enterprise Grid Computing System. http://www.gridbus.org/%7Eraj/papers/alchemi_icomp05.pdf
4. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids. http://www.gridbus.org/~alchemi/files/alchemi_techreport.pdf
5. http://www.alchemi.net – официальный сайт проекта Alchemi
6. http://www.povray.org – официальный сайт проекта POV-Ray
7. http://megapov.inetart.net – официальный сайт проекта MegaPOV