SOA grid design patterns for computer graphics animation

Using Alchemi to render POV-Ray animations on a grid

Skill Level: Intermediate

Michael Head (head@acm.org)
Consultant
##

10 Jul 2007

Ray tracing produces high-quality images with realistic reflections, shading, and perspective. Computational efficiency of rendering is achieved through design patterns for a grid service model fitted into a Service-Oriented Architecture (SOA). Learn how to configure and run Alchemi — a grid services model for rendering — build a simple distributed scene-animation-rendering application using the Alchemi framework, deploy it with a Web services interface, and test your application with a simple animated scene.

Section 1. Before you start

About this tutorial

Service-Oriented Architecture (SOA) provides a convenient cross-platform Internet-scale mechanism for composing software components. Alchemi is a Microsoft® .NET-based system and framework for building and deploying grid applications with C# that provides a Web services interface to the application. This tutorial provides step-by-step instructions for building a Web services-enabled grid application that renders 3-D computer graphics (CG) animations using Alchemi and the Persistence of Vision Raytracer (POV-Ray). In addition, it discusses these concepts in the context of grid computing and SOA.

Objectives

You will learn how to configure and run Alchemi, build a simple distributed scene animation-rendering application using the Alchemi framework, deploy it with a Web services interface, then test your application with a simple animated scene.

Prerequisites

This was written for those with an introductory-level familiarity with Microsoft Visual Studio® 2005, C#, and Microsoft ASP.NET.

System requirements

To complete all the steps, you need a computer with Microsoft Windows® XP Professional and Internet Information Services (IIS), along with Visual Studio 2005 and the Microsoft .NET V2.0 framework, including ASP.NET V2.0. You need at least 500 MB of free disk space (depending on how many animation frames you want to render), Internet access to download Alchemi, and access to install and run applications on the machine.

Section 2. Introduction

Before diving in, I'd like to discuss the topics in a bit more depth, so you'll have a better ground in the actions you'll be taking.

The SOA style

The SOA style has gained momentum in recent years. Solutions adhering to this style consist of loosely coupled services that can be combined and recombined simply, hiding details like data access and internal software models behind interface contracts, such as the Web Services Description Language (WSDL) document. The major benefit is that services within an SOA can more easily interoperate. In this way, higher-level tools, such as business process modelers, which know nothing about the underlying services other than their contract, can take advantage of all the software within an enterprise or even between enterprises. While an SOA solution

might use any number of underlying technologies, for the purposes of this tutorial, I consider SOA to mean a solution based on Web services and SOAP.

Grid computing

Grid computing has been another subject of interest lately. Like SOA, grid computing has no hard and fast definition. But in general, it refers to combining large numbers of computers that may or may not be owned and managed by a single entity. In some communities, this means connecting clusters, supercomputers, and data centers owned by different universities and research laboratories. In other contexts — and in the context of this tutorial — it means combining workstations and desktop computers to achieve a single virtual high-performance computer.

Computer graphics

CG is another broad topic. It can involve interactive 3-D virtual environments, computer-aided design (CAD) systems, and CG-animated movie rendering, among many other use cases. This tutorial deals with the case of rendering prepared scenes with a ray tracer.

Section 3. Design patterns for SOA and grid applications

Design patterns capture common solutions to recurring problems. There are many well-known patterns, such as singleton and factory, that apply across many domains because they solve general programming problems. Within a domain, however, there may be recurring problems that may not be apparent in other domains. The grid domain includes problems related to node management (how to divide work across all those machines, for example), as well as to general programming. This section describes a few patterns.

Embarrassingly parallel

There is a class of computational problems that takes a large amount of computation, but can trivially be divided into very small chunks. These sorts of problems are affectionately referred to as *embarrassingly parallel*. A famous example is the SETI@Home workload, which divides the task of searching for signs

of extraterrestrial life in vast amounts of radio telescope data across millions of computers with owners willing to donate. In a manner analogous to searching for the figurative needle in a haystack, small portions of the haystack are given to many people to search through.

Because it's so easy to break up these problems, they are amenable to grid solutions. CG rendering is another task that, as you will see, is easily broken up into small chunks.

CPU scavenging

There are many ways to build large computer systems that can process massive quantities of data. Such systems can involve specialized hardware, such as is used in Cray supercomputers, they can involve high-performance networking devices and dedicated servers in a data center, or they can involve gathering all the computers in an organization — or even on the Internet — and using all the spare CPU cycles. Systems built using the latter model are called *CPU scavenging* systems. CPU scavenging systems tend to be easy to install and are useful for building programs that solve embarrassingly parallel problems. Alchemi is one of these systems, although it can be used with dedicated servers, too.

Grid Web services

Grid Web services can refer to a number of technologies and standards. Here, I introduce a simple Web services interface to a grid application, but it's important to mention other definitions. Indiana University's Extreme! Computing Laboratory (see Resources) started early work on combining Web service and grid computing, making SOAP and XML more amenable to large-scale scientific computing. Still, the most common definition of grid Web services is the Web Services Resource Framework (WSRF) specification, defined by the Globus Alliance and IBM® in 2004 (see Resources).

Section 4. Setting up Alchemi, POV-Ray, MegaPOV, and the sample code

In this section, you install the software demonstrated in this tutorial. To prepare for the installation, download the code from the Sample code section, along with the

following software:

- Alchemi Manager, Executor, and SDK — This tutorial uses V1.0.5 for Microsoft .NET 2.0.
- POV-Ray scene renderer — This tutorial uses V3.6.1c for Windows.
- MegaPOV, a command-line interface for POV-Ray on Windows — This tutorial uses V1.2.1 for Windows.

You should have Visual Studio 2005 Professional installed with the ASP.NET software development kit (SDK).

Install the software

I had some problems with Alchemi V1.0.6 (the release from 16 Oct 2006), so I used V1.0.5. Specifically, I downloaded the following files:

- Alchemi.Executor-1.0.5-net.2.0.msi
- Alchemi.Manager-1.0.5-net-2.0.msi
- Alchemi-1.0.5-sdk-net-2.0.zip

Note: These instructions prepare the software for one node. To add more computers to your grid, install each one with Alchemi Executor, POV-Ray, and MegaPOV.

Install Alchemi Manager

To install Alchemi Manager, accept all the installation defaults with the exception of the **Install Alchemi Manager for yourself, or for anyone who uses this computer** option. Choose to install it for **Everyone**, instead. Then, in the Install Database window, click **Install Database** and close the window.

Install the Alchemi Executor

To install the Alchemi Executor, again accept all the defaults with the exception of installing for **Everyone**.

Install the Alchemi SDK

Next, extract and install the Alchemi SDK. You can place the contents of this file anywhere (I left them on the desktop). The compressed file contains a grid monitoring application and a set of example grid applications and the

Alchemi.Core.dll file, which is the assembly required to write grid applications that take advantage of Alchemi.

Install POV-Ray

The Windows download of POV-Ray (povwin36.exe) is a standard executable installer. You can accept all the default options.

Install MegaPOV

MegaPOV is packaged into megapov-1.2.1-windows.zip. Extract this file into C:\Program Files\megapov-1.2.1-windows.

Launch Alchemi

The Alchemi Manager and Alchemi Executor should appear in the Windows Start menu under Alchemi/Manager/Alchemi Manager and Alchemi/Executor/Alchemi Executor, respectively. To launch Alchemi:

1. Launch Alchemi Manager so Alchemi Executor has one to connect to.
2. Note the settings and the port, then click **Start**.
3. Launch Alchemi Executor. If no Alchemi Executor window appears, it may have created a task bar entry with an **E** icon. Double-click this icon to display the Executor window.
4. The Executor should automatically connect to the Manager. If it doesn't, click **Connect**.
   Note: If you are installing the Executor on a separate node, type the host name of the Manager in the **Manager Node** box.
5. By default, the Executor starts in nondedicated mode, so you must click the **Manage Execution** tab in the Executor window, and click **Start Executing**.

At this point, you should be able to run the sample applications in the SDK.

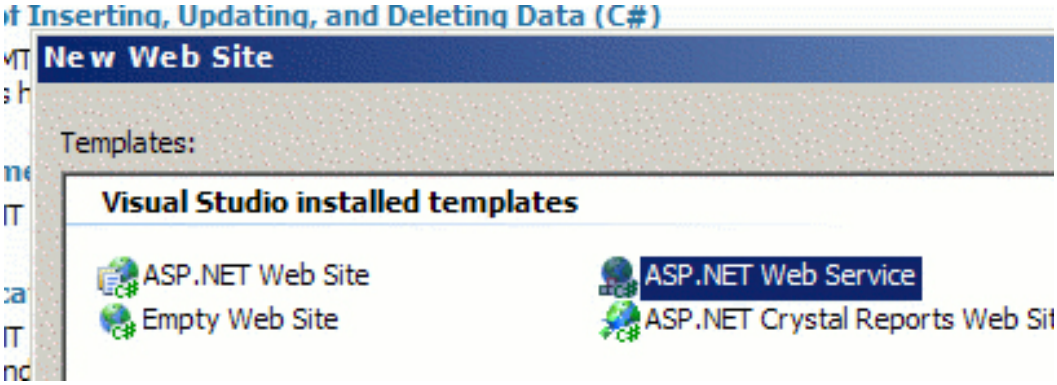
Extract and use the sample code

The sample code package, SampleCode.zip, contains four files:

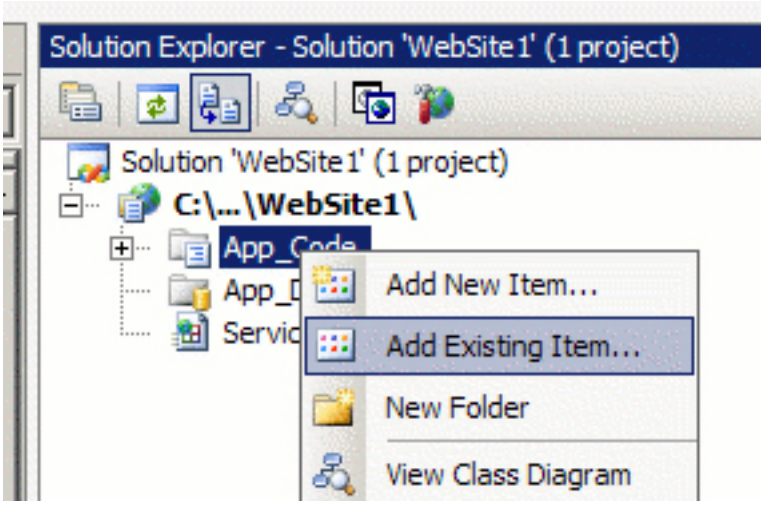
- hello.pov
- SceneRenderer.cs
- Service.cs
- Web.Config

You can place the hello.pov file anywhere as long as you remember the location. The other files should be loaded into an ASP.NET solution in Visual Studio.

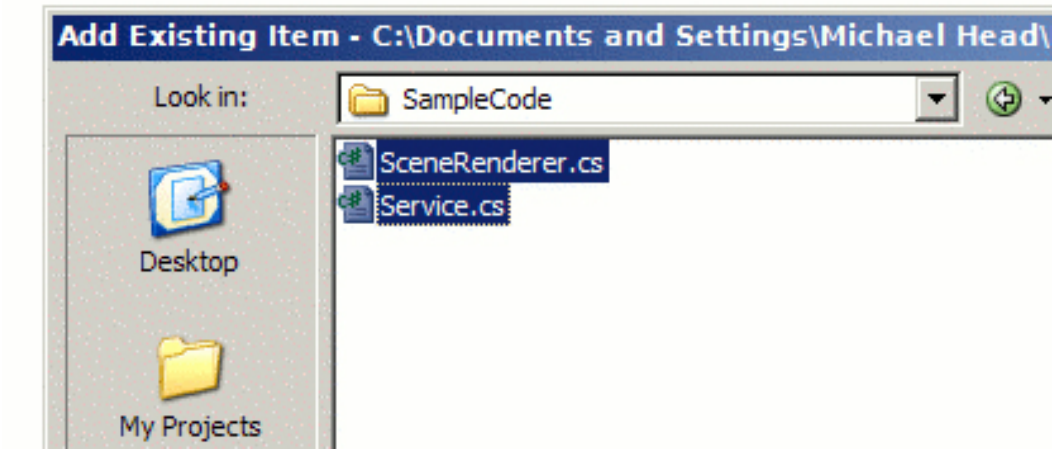
To load the Alchemi DLL file and the sample source code:

1. Start Visual Studio 2005.
2. Create a new Web site by clicking **File > New > Web Site**.
3. Select **ASP.NET Web Service**, as shown below.
   Figure 1. Create a new Web site
   

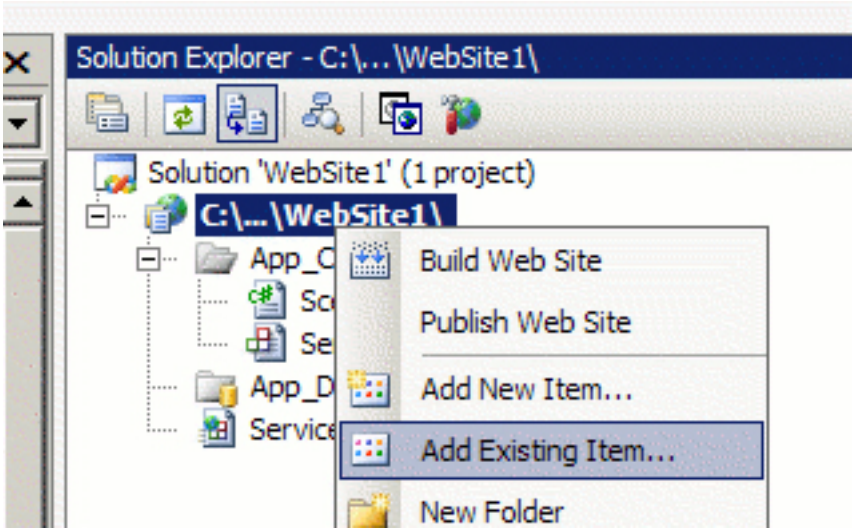
4. Right-click the App_Code folder in the Solution Explorer, then click **Add Existing Item**.
   Figure 2. Add an item to the App_Code folder



5. Browse to the folder in which you extracted the sample code, then select Service.cs and SceneRenderer.cs, as shown below. **Note:** Accept any prompts that appear during the import process.
Figure 3. Select the .cs files



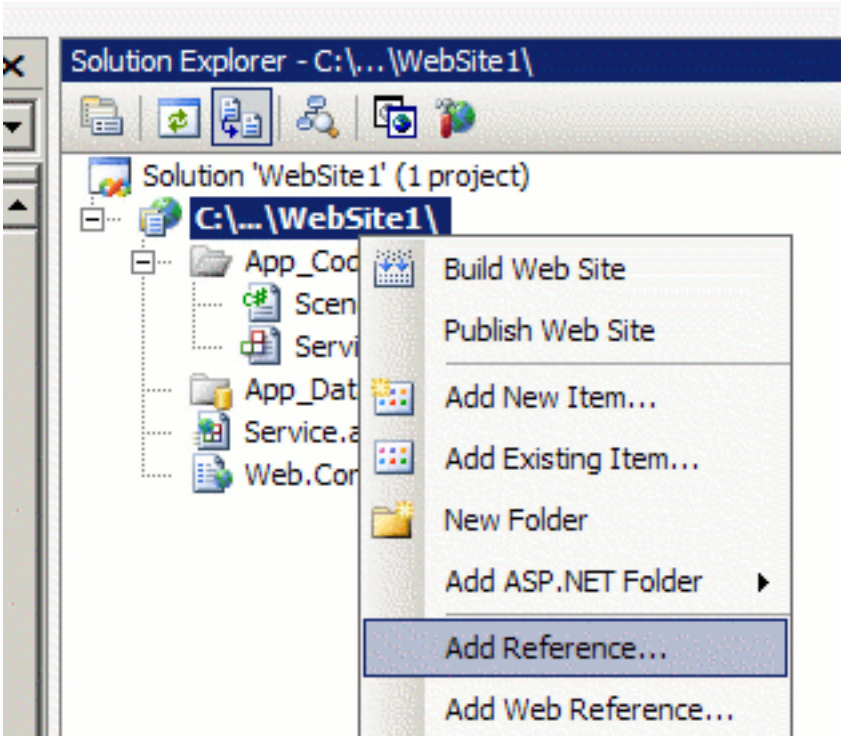
6. Right-click the C:\...\WebSite1\ entry in the Solution Explorer, then click **Add Existing Item**.
Figure 4. Add an item to the WebSite1 folder



7. Select the Web.Config file.

8. Right-click the C:\...\WebSite1\ entry in the Solution Explorer, then click **Add Reference**.

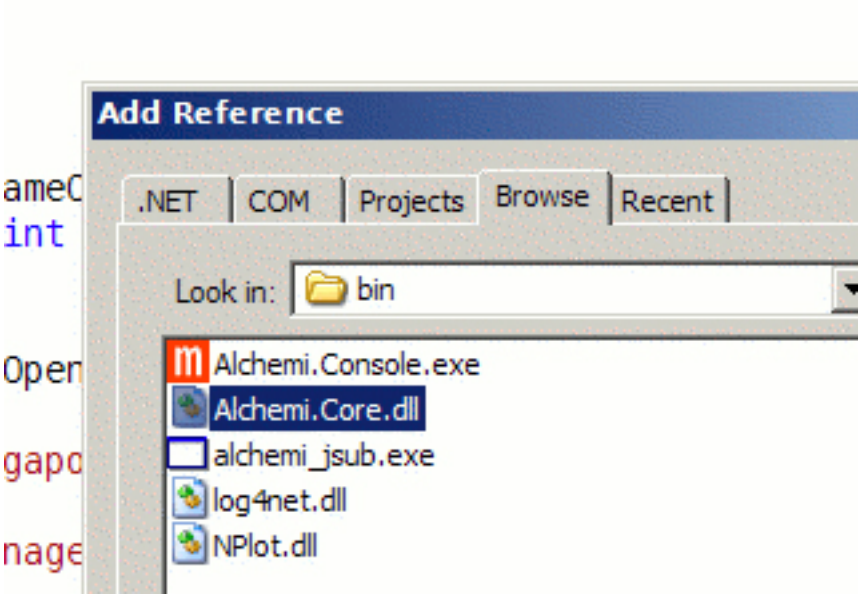
Figure 5. Add a reference to the Web site



9. Click the **Browse** tab.

10. Navigate to the bin folder in the directory into which you extracted the Alchemi SDK files.

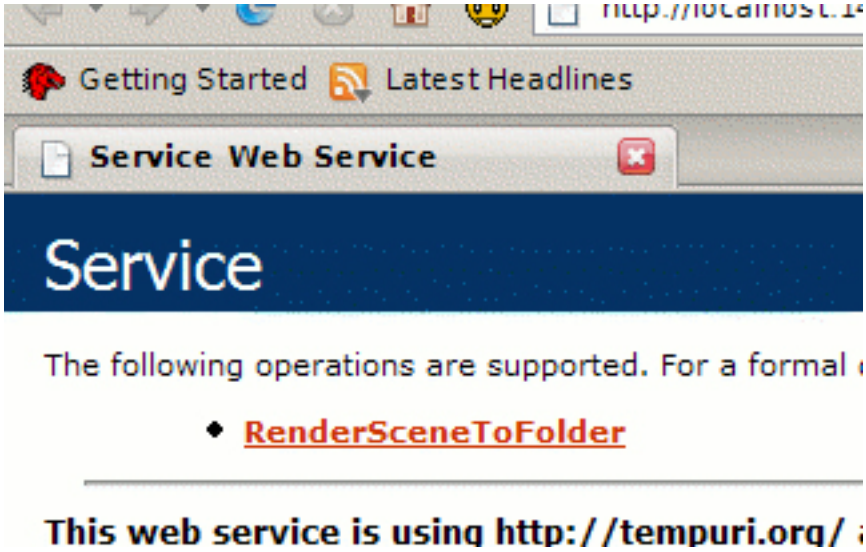
11. Select Alchemi.Core.dll, then click **OK**.
Figure 6. Select the Alchemi DLL



12. Select Service.asmx in the Solution Explorer.

13. Click **Debug > Start Debugging** from the main menu bar.

14. In the Web browser window that appears, click the **RenderSceneToFolder** link.
Figure 7. Rendering the scene to a folder



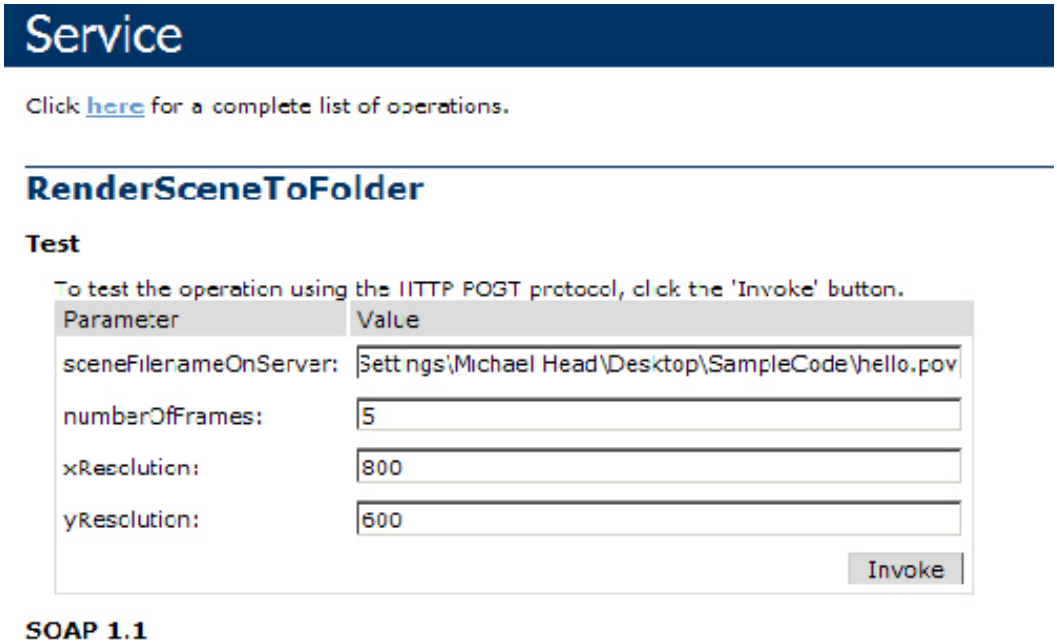
15. Type the path to the hello.pov file on your computer in the first box.

16. Type the number of frames to render. (Five should be enough for a test run.)

17. Type an X resolution, such as **800**, then type a Y resolution, such as **600**.

18. Click **Invoke**. After a short wait, the service should return a URL similar to Figure 8.

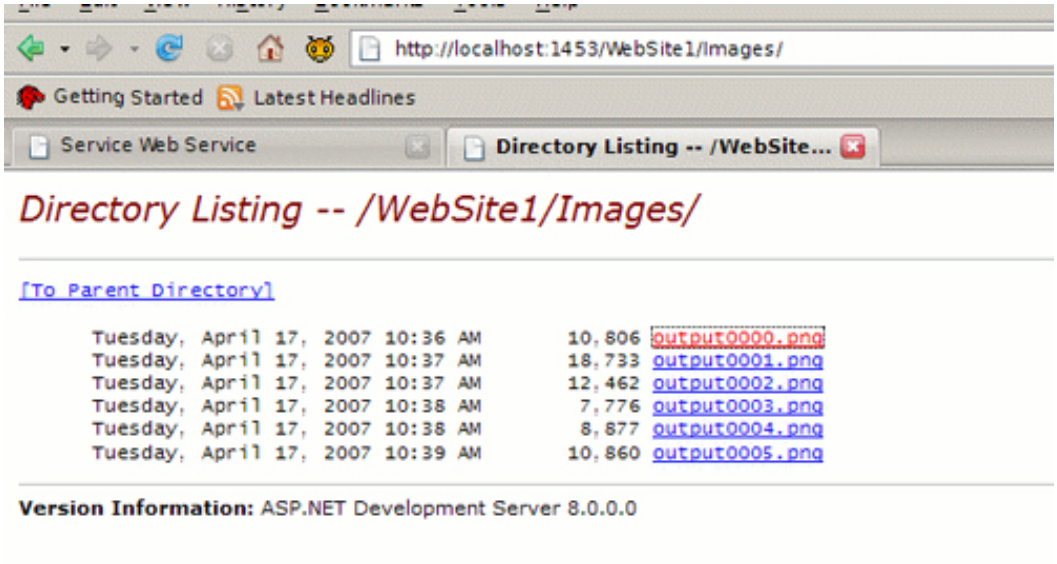
Figure 8. The returned URL



19. Paste the URL in the entry in the browser.

20. Click the links to see the resulting frames of the animation.

Figure 9. The list of animation frames



Section 5. Using POV-Ray for scene construction and animation

POV-Ray is a language and a tool for creating and rendering 3-D scenes. The rendering process can take a long time, particularly for very complex scenes with a lot of objects. This section describes a bit about how POV-Ray works and provides a sample (and simple) animated scene.

POV-Ray's rendering model

POV-Ray renders 3-D scenes using ray tracing techniques. This means that light sources and a camera, representing the viewer's eye, are modelled in the scene along with the objects themselves. During the rendering process, for each pixel in the image, a ray is drawn from the eye through the screen at the pixel location into the scene. The ray reflects and refracts against and through the objects until it hits one or more light sources. The colors generated by the light sources and those

found in the scene objects are combined to determine the pixel's value. When this process has been completed for each pixel, the image is complete.

Animated scenes

You can create animations by tying scene object and light source movements to a clock. The clock is a number you can use to move, spin, or deform objects. For example, you can move an object horizontally through the scene by fixing its X-coordinate to the clock. There are many ways to manipulate the clock, but you'll be using a simple method: as a number ranging from 0 to 1 equal to the current frame number being rendered divided by the total number of frames to be rendered. This makes it easy to render animations during development that use a low number of frames for faster rendering times, then to switch to a high number of frames for production while starting and finishing on the same scene.

A sample animated scene

I've put together a simple animated scene to render throughout the tutorial. The scene has two blocks of text and a flying sphere. The two blocks of text say "Welcome" in green and "To POV-Ray" in red, and spin around the center of the scene as the animation unfolds. The sphere is blue and flies across the scene from right to left, casting a shadow on the text. Two light sources illuminate the scene: one from behind the scene, one from above the camera.

Listing 1. Hello.pov, a sample POV-Ray scene

```
#declare DisplayFont = "C:\\WINDOWS\\Fonts\\arial.ttf"

camera {location<0,0,-10> look_at<0,0,0> }

text {ttf DisplayFont "Welcome", 0.25,<0,0,0>
        pigment {rgbf<0,1,0,0.5>}
        translate <0.66,0,0>
        rotate <clock*720,clock*360,0>
}

text {ttf DisplayFont "To POV-Ray", 0.25,0
        translate <0,-1,0>
        pigment {rgbf<1,0,0,0.5>}
        rotate <clock*-720,clock*360>
}

sphere { <2.5-clock*5,0.5,-5>, 0.25
        pigment{ rgbf<0,0,1,0.25>}
}

light_source { <0,2,10>,  rgb<0,0,1> }
light_source { <0,-2,-10>,  rgb<1,1,0> }
```

Sample rendered images

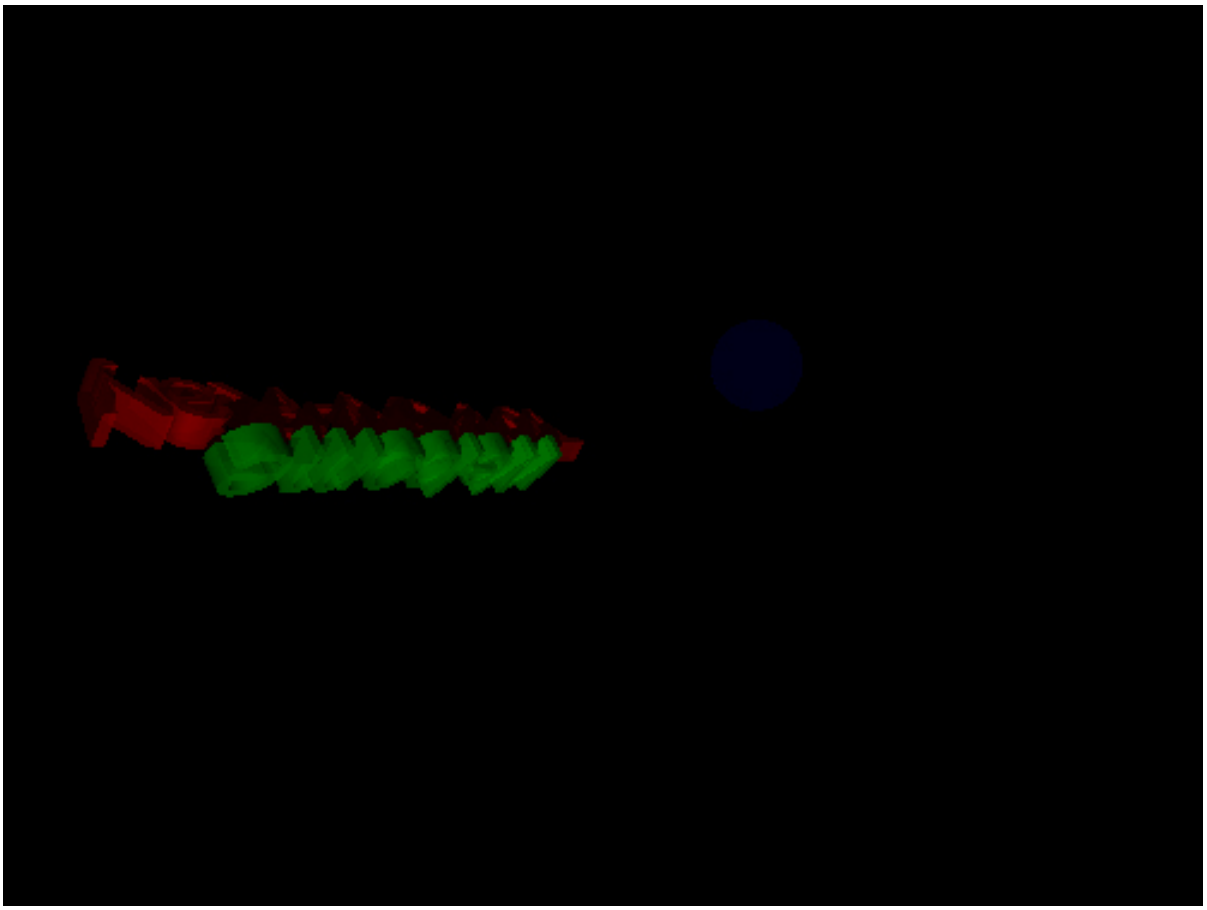
Figure 10 shows the initial frame in the animation. As specified in Listing 1, "Welcome" appears green: `pigment {rbgf=<0,1,0,0.5>}` means the base color of the object has no red component, 100-percent green component, no blue component, and a filter component of 50 percent — a measure of transparency. Similarly, "To POV-Ray" appears in red, because its *color vector* is `<1,0,0,0.5>`. A dark blue sphere appears just above and to the right of the letter e in the word "Welcome."

All the objects appear in their initial positions because the initial frame is rendered with the `clock` variable set to zero. For example, the sphere's location is specified as `<2.5-clock*5,0.5,-5>`. If you substitute the value `0` for the variable clock, you come up with the location `<2.5,0.5,-5>`, which means that the sphere is located 2.5 units to the right of the horizontal center of the scene, 0.5 units above the vertical center of the scene, and 5 units along the Z-axis of the scene. In other words, it's halfway between the camera (with Z-axis set to `-10`) and the words, which have Z-axes set to `0`.

Figure 10. Sample scene rendered with the frame clock set at 0

Figure 11 shows the scene when rendered at clock time 0.33, or one-third of the way through the animation. Note how the words have rotated around the center of the scene and twirled around a spot between them. This behavior is the result of the `rotate <clock*720,clock*360,0>` and `rotate <clock*-720,clock*360,0>` elements. The two text elements have both spun around the vertical axis of the scene by `0.33*360=118.8` degrees. At the same time, they have spun around an imaginary line between them by `0.33*720=237.60` and `-237.60` degrees, respectively, which puts them about one-third of the way around the scene and puts "Welcome" just underneath "To POV-Ray."

Figure 11. Sample scene rendered with the frame clock set at 0.33



Notice that the blue sphere has traveled one-third of the way through the scene. You can calculate its location again. Recalling that its position is defined as `<2.5-clock*5,0.5,-5>`, and `clock` is now `0.33`, its position should be `<0.85,0.5,-5>` — slightly to the right of the center of the scene at the same vertical and Z-axis height as before.

Figures 1 and 2 above were rendered using MegaPOV at two different clock times to show how the frames of the animation shift as time changes. The figures were generated using the commands in Listing 2. Note that the options use the plus sign (`+`) rather than a minus sign (`-`) or a forward slash (`/`). Using `-` often disables a given option. Here's what each option means:

- `+Ihello.pov` sets the input file to hello.pov.
- `+W500 +H375` sets the dimensions of the image to 500 pixels wide and 375 pixels high.
- `+FN` sets the output file format to PNG.

- `+K0.33` sets the clock value to 0.33.
- `+Ofigure1.png` sets the output file to figure1.png.

Listing 2. MegaPOV command-line arguments used to generate the images

```
megapov.exe +Ihello.pov +W500 +H375 +FN +Ofigure1.png
megapov.exe +Ihello.pov +W500 +H375 +FN +K0.33 +Ofigure2.png
```

Section 6. Building the grid application

Alchemi makes it easy to develop grid applications by presenting the *grid thread* abstraction. Grid threads work similar to regular Microsoft .NET threads, except that you can schedule them on any Executor machine in the cluster.

The design of this grid application is simple: You simply package the scene file, decide which frame or part of the scene to render in each thread, then launch the threads you need. When they're all complete, you gather the results and put together the final scene or animation.

Create the grid application

You're going to build the grid application in two classes: an application manager (`SceneRenderer`), of which there's typically one instance, and the grid thread class (`RenderThread`), of which there are many instances. The task of the `SceneRenderer` is to connect to the Alchemi Manager, authenticate, launch the threads, and collect the results. Listing 3 shows a snippet with this boilerplate.

Listing 3. Snippet from SceneRenderer.RenderScene showing the grid setup code

```
                    GApplication GridApp = new GApplication();
  GridApp.ApplicationName = "POV-Ray Render Grid Application";
  GConnection gc = new GConnection(hostname, port, username, password);
  GridApp.Connection = gc;
  GridApp.Manifest.Add(new ModuleDependency(typeof(SceneRenderer).Module));
  GridApp.ThreadFinish += new GThreadFinish(GridApp_ThreadFinish);
  GridApp.ApplicationFinish += new GApplicationFinish(GridApp_ApplicationFinish);

  byte[] sceneBytes = File.ReadAllBytes(sceneFileName);
  // Create a RenderThread for each frame and dispatch them
```

```
   for (int i = 0; i <= nFrames; i++)
   {
      GridApp.Threads.Add(new RenderThread(sceneBytes, i, nFrames,
                                           xResolution, yResolution,
                                           megapovLocation));
   }
   GridApp.Start();
   //  Will wait here until application is complete
   while (GridApp.Running)
   {
       System.Threading.Thread.Sleep(2000);
   }
```

The `GThreadFinish` delegate, shown in Listing 4, extracts the data from the execution threads. As you can see in Listing 5, the `RenderThread` worker packages its results (the actual bits of the image rendered) into a `byte[]` array. The `GridApp_ThreadFinish` delegate must read this array and write those bits out to files on the disk. It names the output file something like output0001.png by getting the thread's `FrameNumber` parameter and setting that number at the end of the file name.

Listing 4. GridApp_ThreadFinish delegate

```
void GridApp_ThreadFinish(...)
 {
   RenderThread rt = (RenderThread)thread;
   string filename = String.Format("output{0:D4}.png", rt.FrameNumber);
   DirectoryInfo info = new DirectoryInfo(OutputDirectory);
   if (!info.Exists)
   {
     info.Create();
   }
   File.WriteAllBytes(Path.Combine(OutputDirectory, filename), rt.OutputImageBytes);
 }
```

Build the grid thread workers

The worker thread is also rather easy to implement. When it's created and its constructor is called, the thread simply copies all the needed configuration information. When the thread is actually scheduled and run on an Executor node, the `Start()` method, shown in Listing 5, is called.

The most difficult part is getting MegaPOV to launch with the right command-line options and ensuring that it doesn't display a window on the Executors that would annoy users who have decided to contribute their computers to the grid. One of the trickier problems is realizing that MegaPOV doesn't understand file paths with backslashes in them. As a result, you must call `povFileName.Replace("\\","/")` before passing the input file to MegaPOV.

The bytes of the POV input file are written to a temporary file so you can pass that file to MegaPOV. For simplicity, the output file name is chosen to be the input file name with the extension changed to .png. MegaPOV is called with a somewhat complex command line, although it ends up being similar to what you saw in Listing 2. Note that you set the clock option (+K) by dividing the `frameNumber` by the total number of frames, `nFrames`. After the process is run, the bytes are packaged into a `byte[]` array for processing by the `GridApp_ThreadFinish` delegate.

Listing 5. RenderThread.Start() method showing the interaction with the Microsoft .NET process-launching facility

```
public override void Start()
 {
   string povFileName = Path.GetTempFileName();
   File.WriteAllBytes(povFileName, sceneBytes);
   string pngFileName = Path.ChangeExtension(povFileName, ".png");

   string megapovPath = Path.Combine(megapovLocation, "megapov.exe");
   string arguments = String.Format("-W{0} -H{1} -I\"{2}\" -O\"{3}\" +FN +K{4}",
                       XResolution, YResolution, povFileName.Replace("\\", "/"),
                       pngFileName.Replace("\\", "/"),
                       (double)frameNumber / (double)nFrames);
   Process process = new Process();
   process.StartInfo.FileName = megapovPath;
   process.StartInfo.Arguments = arguments;
   process.StartInfo.WorkingDirectory = Path.GetDirectoryName(povFileName);
   process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
   process.StartInfo.UseShellExecute = false;
   process.StartInfo.CreateNoWindow = true;
   process.StartInfo.RedirectStandardError = false;
   process.StartInfo.RedirectStandardOutput = false;
   process.Start();
   process.WaitForExit();

   outputImage = File.ReadAllBytes(pngFileName);
   File.Delete(povFileName);
   File.Delete(pngFileName);
 }
```

Note: The packaging and unpackaging of files is necessary to enable network transparency for the grid worker threads. Passing a file path as is — say, by passing the string `C:\temp\input.pov` — wouldn't work, because that file might not exist on the machine on which the worker thread is scheduled. The code makes the assumption that megapov.exe exists in C:\Program Files\megapov-1.2.1-windows to simplify coding and because it is reasonable to assume that a program that the renderer service requires is installed on all machines in the grid.

Section 7. Building the Web service and submitting the

animation scene

Alchemi does not provide a SOAP wrapper for grid applications, so it's necessary to implement one. This section describes the Web service I created for the scene animation application described earlier.

Build the SOAP interface

To submit the animation scene to the grid service, you must build a SOAP interface to the grid service. Doing so requires a bit of ASP.NET coding. The task involves getting all the parameters from the user and the local configuration in place so the grid service can run. I chose to put the Alchemi authentication and connection information in the Web.Config file to minimize the number of parameters the SOAP interface requires. In production code, it probably makes more sense to manage the authentication information more rigorously.

To simplify the service, the scene file must reside in the Web server's file system (or potentially, a network share). It wouldn't be too difficult to add it as a SOAP parameter — perhaps as a base64-encoded byte array containing the file — but that would just complicate the sample code. The service lets the client specify the path to the scene, the number of frames to render, and the resolution of the animation. In addition, the service directs the `SceneRenderer` to output to a directory inside the Web site; as its only result, it returns the URL that lists all the images so that the client can then download them.

The service can combine the images and encode and compress them into an MPEG video file, then return this file as a Direct Internet Message Encapsulation (DIME) or Multipurpose Internet Mail Extensions (MIME) SOAP attachment, but that's beyond the scope of this tutorial. The bulk of the code representing the service is shown below.

Listing 6. The SOAP implementation that manages the parameters for the SceneRenderer

```
[WebMethod]
 public string RenderSceneToFolder(string sceneFilenameOnServer, int numberOfFrames,
                                                    int xResolution, int yResolution)
 {
    SceneRenderer renderer = new SceneRenderer();
    Configuration config = WebConfigurationManager.OpenWebConfiguration("~");
    string megapovLocation =
           config.AppSettings.Settings["renderer.megapov.location"].Value;
```

```
    string managerHostname =
           config.AppSettings.Settings["renderer.manager.hostname"].Value;
    int managerPort = Int32.Parse(
           config.AppSettings.Settings["renderer.manager.port"].Value);
    string username =
           config.AppSettings.Settings["renderer.manager.username"].Value;
    string password =
           config.AppSettings.Settings["renderer.manager.password"].Value;

    string outputDirectoryForImages = Path.Combine(
           Context.Server.MapPath(
            ((WebContext)config.EvaluationContext.HostingContext).ApplicationPath),
           "Images");
    renderer.RenderScene(sceneFilenameOnServer, numberOfFrames, xResolution,
                         yResolution, megapovLocation, outputDirectoryForImages,
                              managerHostname, managerPort, username, password);
    string baseUrl = Context.Request.Url.GetLeftPart(UriPartial.Authority);
    Uri result = new Uri(new Uri(baseUrl),
          ((WebContext)config.EvaluationContext.HostingContext).ApplicationPath
           + "/Images");
    return result.ToString();
 }
```

Submit the job

After you've deployed this service into an ASP.NET server, submitting a new task is easy. ASP.NET provides SOAP V1.1, SOAP V1.2, and Representational State Transfer (REST)-style HTTP POST interfaces for Web methods. As such, all that's needed to submit the job is to put the scene file on the server, put together some XML code (as shown in Listing 7), and post it to the service URL with the appropriate SOAP HTTP headers.

Listing 7. XML input for the service

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
   <soap:Body>
      <RenderSceneToFolder xmlns="http://tempuri.org/">
         <sceneFilenameOnServer>C:\temp\hello.pov</sceneFilenameOnServer>
         <numberOfFrames>5</numberOfFrames>
         <xResolution>800</xResolution>
         <yResolution>600</yResolution>
      </RenderSceneToFolder>
   </soap:Body>
</soap:Envelope>
```

Section 8. Performance, extensibility, and limitations

The sample animation application is not applicable for all possible scenarios. As a result, this section briefly describes some of the ways in which you could modify it to resolve some of those deficiencies.

Strategies for dividing the workload

For this application, rendering 3-D scene animations, I've chosen to use the single frame as the smallest chunk of work and have one grid thread for each animation frame. This setup works well because reasonably sized frames of this simple animation render rather quickly — in a matter of seconds on relatively modern hardware. More complicated scenes with lots of refraction and translucent materials or frames rendered at very high resolutions would take much longer to render. In these cases, you may want to break down the work further, particularly if the scene being rendered isn't animated.

It is possible to divide the task of rendering a single frame further with POV-Ray, although doing so would require modifications to the grid application. You can subdivide the scene into stripes and dedicate separate threads to rendering these stripes. For example, when rendering a 1600x1200 image, you could render four stripes of, say, 1600x300 by including additional arguments to the MegaPOV command line.

Keep in mind that dividing the task too much could hurt overall performance. The more chunks the more time will be spent on the overhead of communicating between nodes and scheduling the tasks. It's also a good idea to consider the number of computing nodes that will be available and to be aware of how they're configured (dedicated or not). The number of grid threads should probably stay within a few multiples of the number of nodes; otherwise, the overhead of scheduling the threads won't be offset by being able to schedule them on more nodes. However, if the grid is made up of mostly nondedicated nodes that might be in active use for other tasks, it might be sensible to increase the number of thread to give the scheduler more choices for distributing the workload.

Extensibility to NVIDIA's Cg Toolkit

NVIDIA makes widely deployed high-performance 3-D graphics adapters. These 3-D adapters are very powerful, but most of the time, this capacity goes unused. To help developers take advantage of this extra computing facility and to develop real-time 3D applications, NVIDIA created the Cg Toolkit. Unfortunately, this technology doesn't quite apply to POV-Ray because it would require that POV-Ray be rewritten to take advantage of Cg.

Extensibility to other platforms

While the Microsoft .NET Platform is for Windows only, Novell has developed an open source project called Mono that compiles and runs Microsoft .NET applications on Mac OS X and various distributions of Linux®. I've read that it's possible to run Alchemi with Mono, although the sample code here uses Windows-specific paths, so it wouldn't run on Linux or Mac OS X without changes.

Limitations of the sample code

In the interest of keeping the code simple, I left out a lot of functionality that would be desirable or even required for serious use. Here are a few things to note:

- Remember that the scene file must reside on the SOAP server before submission.
- You should work out some policy about authentication and authorization.
- The grid application should more intelligently divide the task to take into account very complex scenes and nonanimated tasks.
- Consider working out a better mechanism for gathering the rendered images, converting them into a video — which could be done on the grid as its own application — then returning them to the client.
- The threads assume that the Executor nodes all have MegaPOV installed to the same location. More importantly, the grid threads don't check for error conditions when executing MegaPOV, which could be problematic if errors do occur.
- Most critically, the client could timeout waiting for a result for longer-running rendering tasks.
- SOAP over HTTP was never intended to be used with long-running services, so creating two services — one to initiate the rendering task and one to obtain the result if the task is complete — could be the way to go.

Section 9. Summary

In this tutorial, you learned about SOA and grid computing, as well as how to use Alchemi to implement Web services for grid applications. Further, you learned about the POV-Ray and MegaPOV tools for scene rendering. The sample application uses Alchemi to distribute the rendering task around a grid of Windows workstations. Finally, you saw some of the limitations of this solution as well as how you could extend it to resolve these limitations.

Downloads

Description	Name	Size	Download method
Sample code and POV-ray scene	gr-soacg-SampleCode.zip	4KB	HTTP

Information about download methods

Resources

Learn

- "Grid and SOA" defines and describes how grids and SOAs are merging.
- "Build grid applications based on SOA" describes the concepts behind SOA and how to move grid applications to an SOA model.
- In the developerWorks SOA and Web services zone, get the resources you need to advance your knowledge and skills with Web services.
- Browse the Alchemi documentation for more tutorials, screenshots, demos, and presentations.
- Visit the Safari bookstore for books on these and other technical topics.
- Check out the Indiana University Extreme! Computing Lab and its Grid Web Services projects.
- Study the WS-Resource Framework and related standards and technologies from the Globus Alliance.
- Learn about NVIDIA's Cg Toolkit.
- Learn more about Novell's open source Mono project.
- Browse all the grid computing content on developerWorks.
- To listen to interesting interviews and discussions for software developers, check out check out developerWorks podcasts.
- Stay current with developerWorks' Technical events and webcasts.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Grid computing zone for more information.

Get products and technologies

- Download the Alchemi Manager, Executor, and SDK. This tutorial uses V1.0.5 for Microsoft .NET 2.0.
- Download the POV-Ray scene renderer. This tutorial uses V3.6.1c for Windows.
- Download MegaPOV, a command-line interface for POV-Ray on Windows. This tutorial uses V1.2.1 for Windows.
- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®,

Tivoli®, and WebSphere®.

- Innovate your next development project with IBM trial software, available for download or on DVD.

Discuss

- Check out the Grid computing forum on developerWorks.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Michael Head
Michael R. Head has been a doctoral student at Binghamton University since 2004, where he specializes in grid computing. He has worked as an advisory IT specialist with IBM and has published several academic articles on GridFTP, XML, and Web services.