Learn by doing: less theory, more results

Irrlicht 1.7 Realtime 3D Engine

Create complete 2D and 3D applications with this cross-platform, high performance engine

Beginner's Guide

Aung Sithu Kyaw
Johannes Stein

[PACKT] PUBLISHING open source community experience distilled

Irrlicht 1.7 Realtime 3D Engine

Beginner's Guide

Create complete 2D and 3D applications with this cross-platform, high performance engine

Aung Sithu Kyaw

Johannes Stein

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Irrlicht 1.7 Realtime 3D Engine
Beginner's Guide

First published: October 2011

Production Reference: 1181011

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-398-2

www.packtpub.com

Cover Image by Asher Wishkerman (a.wishkerman@mpic.de)

Credits

Authors

Aung Sithu Kyaw

Johannes Stein

Reviewers

Christian Stehno

Nikolaus Gebhardt

Acquisition Editor

Usha Iyer

Development Editor

Maitreya Bhakal

Technical Editors

Joyslita D'Souza

Snehal Gawde

Project Coordinator

Michelle Quadros

Proofreader

Linda Morris

Indexer

Hemangini Bari

Graphics

Geetanjali Sawant

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

About the Authors

Aung Sithu Kyaw is originally from Myanmar (Burma). He has been a developer in the software development industry for over seven years and has a great passion for graphics programming, creating video games, writing, and sharing knowledge with others. He holds a M.Sc (Digital Media Technology) from Nanyang Technological University (NTU), Singapore. Aung is currently the CEO of a digital media company, Rival Edge Pte Ltd, Singapore, which he cofounded in 2011 with two other engineers. Visit `http://rivaledge.sg` for more information.

He can be followed on twitter `@aungsithu` and his LinkedIn profile `http://www.linkedin.com/in/aungsithu`.

Acknowledgement

It took me over eight months to reach this point writing an acknowledgment section. It was really an exciting journey. I owe these people in my endeavour to write this book and I'd like to take this opportunity to express my gratitude.

Thanks to Nikolaus 'niko' Gebhardt and Ambiera e.U. for initiating this wonderful Irrlicht engine project and making it open-source, sharing with the community, Christian Stenho for reviewing the book and giving lots of useful feedback and all the folks from the Irrlicht forum. And the following people from Packt Publishing; Newton Sequeira (Author Relation) for contacting me about this opportunity to write a book on Irrlicht, Usha (Acquisition Editor), Maitreya Bhakal (Development Editor), and Ashwin Shetty (Project Manager) for taking this project into reality, Michelle Quadros (Project Coordinator) for working and coordinating with me along the way, chapter-by-chapter, and Snehal Gawde (Technical Editor) for making sure everything fits together.

I'd like to thank my big brother and my parents, who've been supporting me all these years. Without all of your support I wouldn't have been here today.

And finally a big thank to my wife, Muriel. If she hadn't let me sit in front of my computer for hours, late nights, and weekends, writing this book, it wouldn't have been possible. Thank you for all your support. I love you.

Johannes Stein is an independent game developer and former student of the university of Augsburg, Germany. He started showing an interest in computer science at the early age of twelve and has expanded his knowledge ever since. After learning Visual Basic and Delphi, he got into C/C++ and C#, while he learned Objective-C and JavaScript, in the last few years. He specializes on cross-platform development, which means software that runs on Windows, Linux, and Mac OS X for example. Apart from cross-platform development, he has developed mobile games and web applications using the latest technologies such as HTML5 and CSS3.

On several occasions, he worked as a freelance journalist and reported from yearly events such as the gamescom in Cologne.

Working on a book was a completely new and exciting experience for me and first of all I would like to thank everyone who had the patience to put up with me and helped me during the process. I would like to thank my parents who helped trying to motivate me when I was feeling overwhelmed and frustrated. A special thanks goes to my father.

I would also like to thank Nikolaus Gebhardt and Christian Stehno, who have not only been technical reviewers of this book, but shaped the Irrlicht engine into what it is now.

And last, but not least I would like to thank everyone at Packt Publishing, who made this book possible and a simple "Thank you" does not cover how grateful I am.

About the Reviewer

Christian Stehno has been in touch with Irrlicht since 2004. Being a researcher at that time, he led a group of students to develop a 3D visualization tool. Since 2006, he has been part of the development team. Today he is responsible for the general coordination of the project and its development. When it comes down to coding, he likes working on the video drivers, but does the bug hunting in all parts of the engine. In real life, Christian is married (longer with his wife than with Irrlicht) and has two children. He is the founder and general manager of a small company, developing embedded systems and software for industrial use.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Irrlicht 1.7 Realtime 3D Engine Beginner's Guide will teach you to master all that is required to create 2D and 3D applications using Irrlicht, beginning right from installation, and proceeding step-by-step to deployment.

Beginning with installation, this book guides you through creating a basic template application, followed by using meshes, creating overlays, and UI. You will then scan through data types, nodes, scenes, camera, lights, and particle systems. Finally, you will learn about some advanced concepts such as handling data, files, and shaders, followed by the last stage—deployment—in an appendix.

This book is a step-by-step guide to Irrlicht that starts at an easy level for beginners and then gradually works to more advanced topics through clear code examples and a number of demos, which illustrate theoretical concepts.

What this book covers

Chapter 1, *Installing Irrlicht*, shows how to get and set up Irrlicht across different platforms such as Windows, Max, and Linux.

Chapter 2, *Creating a Basic Template Application*, explains how to set up and create a basic Irrlicht application using different Integrated Development Environments (IDEs) such as Microsoft Visual Studio, XCode, CodeBlocks, and so on.

Chapter 3, *Loading Meshes*, shows how to add 3D meshes to our application, apply materials, and animate them.

Chapter 4, *Overlays and User Interface*, explores the 2D capabilities of Irrlicht, creating graphical user interface (GUI), loading, and rendering 2D images.

Chapter 5, *Understanding Data Types*, provides a primer on C++ templates as well as some mathematical concepts such as vectors.

Chapter 6, *Managing Scenes*, introduces using CopperCube, which is a visual 3D scene editor for Irrlicht available for both free and commercial use.

Chapter 7, *Using Nodes—The Basic Objects of the Irrlicht 3D engine*, explains what the nodes in Irrlicht are and how to use animators to do some basic animations.

Chapter 8, *Managing Cameras*, shows how to use camera scene nodes and touches a little bit on creating a simple terrain to test the walkthrough camera.

Chapter 9, *Lightening the Scene*, introduces lighting concepts, materials, shadows, and using CopperCube/IrrEdit.

Chapter 10, *Creating Eye Candy Effects with Particle Systems*, shows how to implement particle systems both in the visual editor and from the code.

Chapter 11, *Handling Data and Files*, explains how to save and load game-related data files such as player save files.

Chapter 12, *Using Shaders in Irrlicht*, introduces how the graphics rendering pipeline works and practical usage of shaders in Irrlicht.

The *Appendix*, *Deploying an Irrlicht Application*, explains the procedures and essential dependencies to be included while packaging an Irrlicht application for different platforms such as Windows, Mac, or Linux.

What you need for this book

You should have a working knowledge and understanding of programming in C\C++ and a compiler to compile and run the sample code. Your computer should also have a graphic card with 3D capabilities with supports for DirectX and OpenGL. We'll start with setting up Irrlicht on different platforms, so you don't need to have installed Irrlicht on your computer.

Who this book is for

If you have C++ skills and are interested in learning Irrlicht to create 2D/3D applications, this book is for you. Absolutely no knowledge of Irrlicht is necessary for you to follow this book!

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "we load our mesh file and store it in an `IMesh` object".

A block of code is set as follows:

```
if (node)
{
   node->setMaterialFlag(EMF_LIGHTING, false);
   node->setMaterialTexture(0, driver->getTexture("sydney.bmp"));
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "click on **Add Images** to add images".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Installing Irrlicht

This chapter will explain what we need to consider when we take our first step with the Irrlicht 3D graphics engine.

In this chapter, we will:

- Learn about how the Irrlicht library is structured
- Set up Irrlicht on Windows using either Visual Studio or CodeBlocks
- Set up Irrlicht on Linux using the command-line
- Set up Irrlicht on Mac OS X using XCode
- Compile the Irrlicht library for ourselves

So let's get started.

Irrlicht license

Irrlicht is available for free and open source, which means that its source can be viewed and modified by anyone. It uses the zlib license that allows you to use the engine free of charge for private, educational, and even for commercial use. Unlike other open source licenses such as GPL (GNU Public License), you are allowed to modify Irrlicht without publishing your changes under the same license or make your changes not available to the public at all. It is recommended to give credit to the Irrlicht development team in your released application, although this is not necessary with an exception of JPEG support. Irrlicht uses libjpeg for JPEG support and you must credit JPEG group in your release application, according to their license requirement.

System requirements

While the Irrlicht 3D graphics engine should work on pretty much any computer or notebook, there are a few steps to ensure that your Irrlicht development environment, and your created applications, will run smoothly. First of all, ensure that you have installed the latest driver for your graphics card. If you want to be able to use all features of Irrlicht, your graphics card needs to support at least OpenGL 1.5. Also, an equivalent DirectX 9.0c graphics card will suffice, if you are planning to use DirectX and Microsoft Windows exclusively. If you want to modify and recompile Irrlicht, you'll have to install the necessary SDKs such as the latest DirectX SDK for Direct3D 9 support and the May 2006 version of DirectX SDK for Direct3D 8 support. We'll discuss how to set them up in the following sections.

Checking for graphics card support

To check if your graphics card supports OpenGL 1.5, check the specifications of your graphics card on the official homepage of your graphics card vendor. Or alternatively, you can use a tool like DoctorGL that can be downloaded from `http://ononesoft.cachefly.net/support/DoctorGL.exe.zip`.

If you want to make use of Irrlicht's DirectX interface under Windows, you need to have DirectX installed. DirectX runtime files come pre-installed with Windows by default.

Installing the latest DirectX Runtimes

DirectX is an application programming interface, specifically designed for multimedia and games developed by Microsoft.

You can download the latest version of DirectX at the following web address: `http://msdn.microsoft.com/en-us/directx/default.aspx`. On this site click on **DirectX End-User Runtimes (June 2010) Full Download** in the right sidebar listed under **Recent Downloads** to be redirected to the download page.

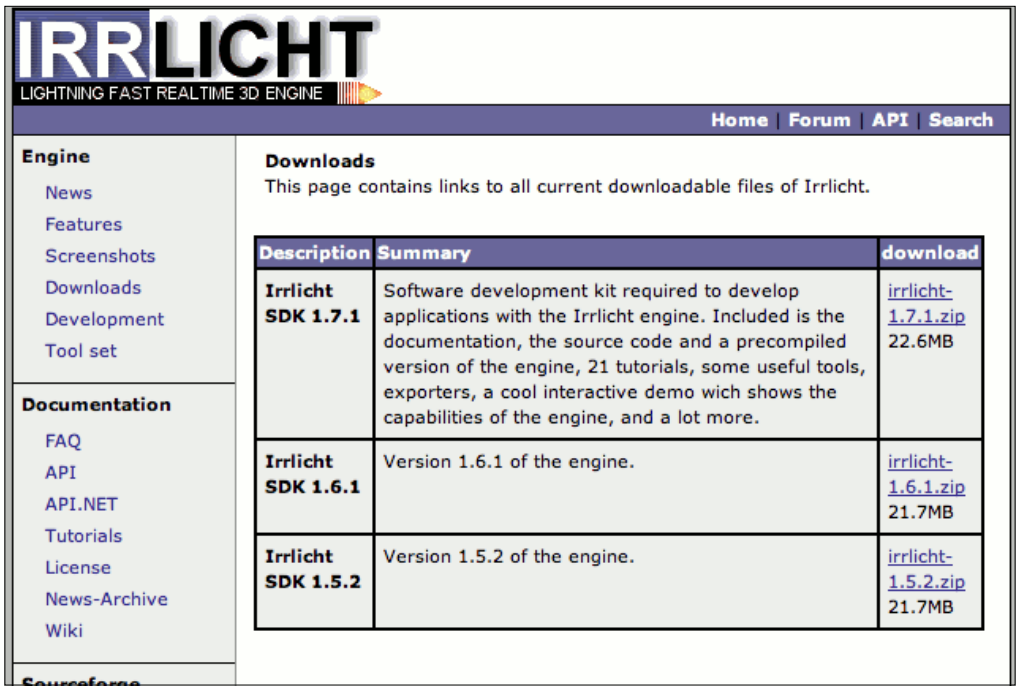
Since Irrlicht is cross-platform and can be used on Windows, Linux, and Mac OS X, we will discuss how to set up Irrlicht on those different platforms with different development environments and compilers. We will create a skeletal application in the next chapter, which can be used as a starting point for the upcoming exercises. Use a compiler or an integrated development environment of your choice. Eventually, it will not matter which IDE or compiler you ultimately choose, because each code sample of this book will work on any IDE and compiler described in this chapter. You'll learn more about platform-specific requirements when you actually try to set up Irrlicht on different platforms and development environments.

Time for action – downloading Irrlicht

The first step is to download Irrlicht from their official homepage. As of the release of this book the latest version of Irrlicht was 1.7.2. The examples in this book should work fine for that version and above.

Now, we are going to download Irrlicht and take a look at what the package has to offer out of the box:

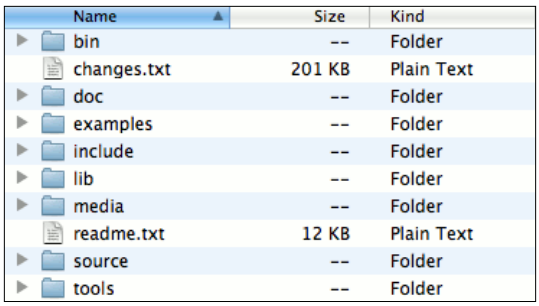
1. Go to the official Irrlicht homepage at `http://irrlicht.sourceforge.net`. Choose **Downloads** in the left sidebar and then choose the latest stable release:



2. As with all projects hosted on sourceforge, you will then be redirected to a site where you can choose from a number of mirrors and the closest mirror to your location is selected by default. The download will start in a few seconds.

Contents of the Irrlicht package

After the download has finished, extract the file at a location of your choice. To extract this file you need an unpacker. Windows, Linux, and Mac OS X have built-in applications for this task. Another solution would use a tool like 7zip (`http://www.7-zip.org/`), which is an open source cross-platform file archiver:



bin folder

The **bin** folder, which splits itself into different folders named after platforms, contains pre-compiled examples. Only 32-bit binaries for Windows, compiled with Visual Studio, can be found here and the examples should give you a general idea of what is possible when using Irrlicht. If you compile an example from the included package in another platform or environment, the compiled executables will be placed here.

doc folder

This contains a complete documentation of the Irrlicht 3D graphics engine, generated from the source code through Doxygen. This documentation is very detailed, but may be a bit overwhelming for beginners. Since there's no function to search in these documents, it's more convenient to Google a specific function of Irrlicht that in turn will point to the online version of this API documentation.

examples folder

The folder **examples** contains a number of examples to demonstrate various features of Irrlicht. We will compile an example after we set up our development environment to test if everything is correctly set up before we move on to the next chapter, where we will create our own template application for the examples of this book.

lib folder

The directory **lib** contains the Irrlicht library and the files that are needed to work with the Irrlicht library. Precompiled Irrlicht library files are stored in this directory so that the application linking to Irrlicht doesn't need to recompile the whole engine every time.

include folder

This folder contains all header files that are necessary to connect with the Irrlicht library itself.

media folder

This folder contains all graphics, models, sounds, and other data needed to run the examples of the Irrlicht engine. Take a note that some of these assets are copyrighted, you should seek further permissions if you plan to use it in commercial applications.

source folder

The folder **source** contains the complete source code of the Irrlicht engine. I will elaborate on that part when I explain how to build the Irrlicht engine from source.

tools folder

As the name suggests this folder contains a number of tools that integrate very well into Irrlicht. For example, a GUI editor or a tool for creating new fonts.

Text files

As you probably noticed, there are two text files in the root directory of the extracted package file, namely **readme.txt** and **changes.txt**. The `readme` file offers short instructions on how to configure and work with Irrlicht, while the `changes` file contains information on what has changed since the last versions of Irrlicht, for example, bug fixes or new features.

Irrlicht on Windows with Visual Studio

Visual Studio is a compiler and an IDE for Windows, developed by Microsoft, and is currently available in its 2010 version. While there are many high-priced versions of Visual Studio (**Professional** and **Enterprise**), there is also a free version called **Express Edition**. Example projects in this book used Visual Studio 2008 Professional, though they should also be able to run in Visual Studio Express Editions. To run any executable, compiled with Visual Studio, you need the Visual C++ Redistributable package that can be downloaded from the Microsoft Download Center.

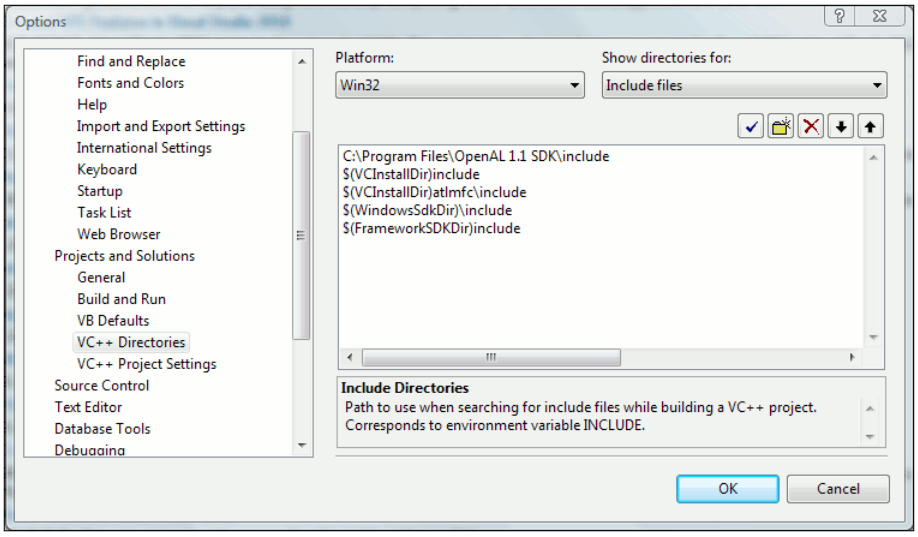
To use Visual Studio Express, you first need to download it from `http://www.microsoft.com/express/downloads/#2010-Visual-CPP`. You will have to register your copy of Visual Studio with your MSN account within 30 days after the installation.

More information on that aspect, as well as how to distribute the Visual C++ Redistributable with your application, is available in the next chapter.

Time for action – adding file references

After you have successfully installed Visual Studio on your computer, we need to add reference paths, where the Irrlicht-related files are located, so that Visual Studio knows where to look, whenever we refer to such files.

1. Open Visual Studio.
2. Go to **Tools | Options**.
3. Expand the menu item **Projects and Solutions** and select **VC++ Directories** from the list.
4. Select **Include files** from the drop-down menu below **Show directories for** as shown in the following screenshot:

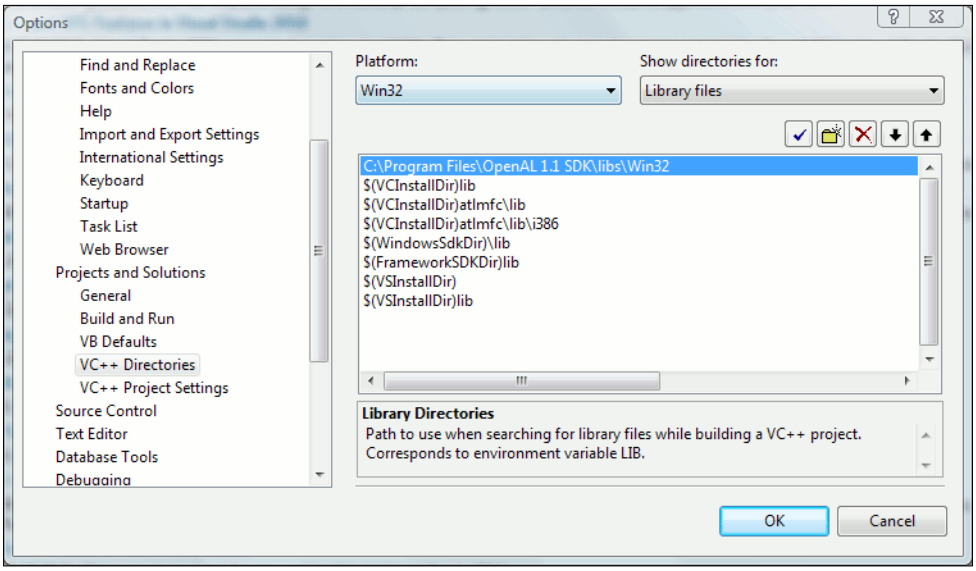


5. Add a new line to add another `include` directory to Visual Studio.

6. Click on the browse button and select the `include` folder from where you have extracted the Irrlicht package.

7. Now select **Library files** from the combo box.

8. Add a new line like you did in step 5 and click on the button with three dots to select the `lib/Win32-visualstudio` folder. Confirm by clicking on **OK**:



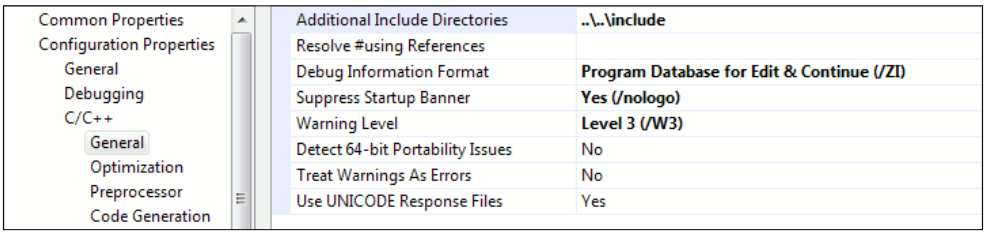
What just happened?

We just added the header files of Irrlicht to the global search path of Visual Studio. That means for every new project you create with Visual Studio, the IDE will first check if necessary files are listed in the global `include` directory and then move on to the local directory. We did the same thing with the library files.

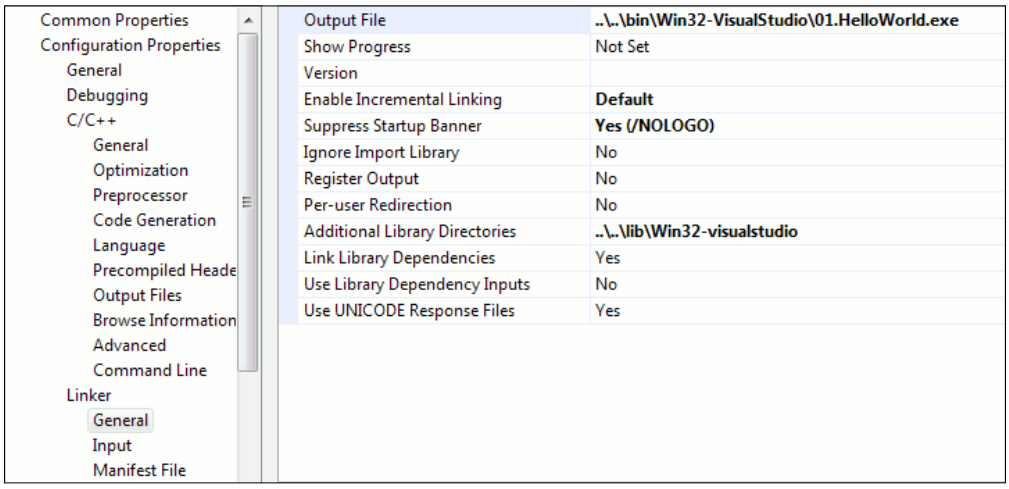
This step is necessary, because the `include` files provide us with the `source` files necessary to interact with the Irrlicht library, while the `Irrlicht.lib` file is the connection between the Visual Studio and the Irrlicht dynamic library.

Project-specific configuration

Another way would be to add these references only in the current active project. Right-click on the **Project file** and select **Properties** from the context menu. If you select **Configuration Properties | C/C++ | General**, you can add the Irrlicht `include` folder in **Additional Include Directories**:



In **Linker | General**, you can add the `lib` directory of Irrlicht to **Additional Library Directories** as shown in the following screenshot:



Remember, if you want to use the alternative configuration method, you need to repeat this step for every new Irrlicht project you set up. On the other hand, the configuration of the global search path works for every project and does not require additional steps after the project is created.

Compiling Irrlicht as a dynamic library using Visual Studio

This, in fact, may not be necessary since the downloaded package already contains pre-compiled libraries for Windows and both Visual Studio and GCC. Those libraries were compiled for 32-bit operating systems only, so if you want to develop natively on a 64-bit system, recompiling the library is recommended. They are also compiled in release mode and contain no debug symbols. So if we want to debug and fix the problems inside Irrlicht DLL, we will need to recompile Irrlicht in debug mode.

For 64-bit Windows

Keep in mind that if you want to compile the Irrlicht library for 64-bit Windows, you need to install the Windows Platform SDK from `http://msdn.microsoft.com/en-us/windows/bb980924.aspx`. While installing, make sure you install the x64 and IA64 compilers from **Developer Tools | Visual C++ Compilers**.

Irrlicht is able to use Direct3D 8 or Direct3D 9 for rendering. In order to compile the library unmodified, having the DirectX SDK installed is absolutely necessary.

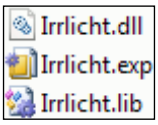
Installing Microsoft's DirectX SDK

The latest version of DirectX SDK can be downloaded from `http://msdn.microsoft.com/en-us/directx/`. But since DirectX 8 SDK is no longer included in the latest DirectX SDK releases, you need to install an older release from summer 2004 if you need to use DirectX 8's features. It is available to download at `http://bit.ly/oPI8AP`. It is also possible to install more than one DirectX SDK on your system. If you wish to install a newer release of the DirectX SDK, go to `http://msdn.microsoft.com/en-us/directx/default.aspx` and select the DirectX SDK of your choice. You should close all the instances of Visual Studio currently running, so that the installer can set up all the necessary configurations with the IDE. Otherwise, you will need to add the `include` and `lib` folders to the global search path of Visual Studio manually, before you will be able to use DirectX to compile Irrlicht.

Time for action – compiling Irrlicht as a dynamic library

Let's get started with compiling Irrlicht as a dynamic library:

1. Open the `source` folder and open the sub-folder `Irrlicht`.
2. Open the solution file corresponding to your Visual Studio version: If you are using Visual Studio 2003 then open `Irrlicht7.1.sln`, if you are using Visual Studio 2005 then open `Irrlicht8.0.sln`, or if you are using Visual Studio 2008 or higher open `Irrlicht9.0.sln`.
3. Make sure Microsoft's DirectX SDK, as well as the Windows Platform SDK are installed on your system. Otherwise you will get errors in the next step.
4. Click on **Build | Build Solution**:



What just happened?

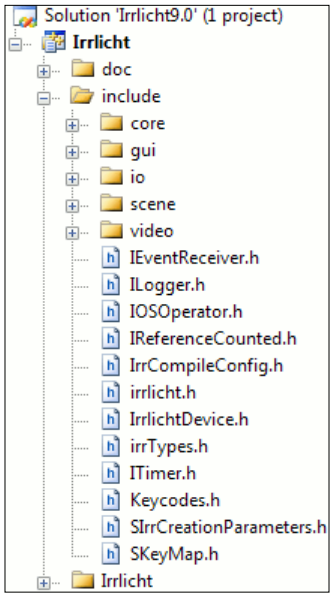
We just built Irrlicht as a dynamic library on Windows. While the dynamic library itself is being automatically copied to `bin/Win32-VisualStudio`, the `lib` and `exp` files for the library can be found in `lib/Win32-visualstudio`.

In case you do not want the DirectX SDK from Summer 2004 or have another reason for not wanting to include DirectX 8 in the Irrlicht, follow these instructions to rebuild the library:

Time for action – compiling the Irrlicht dynamic library with modifications

Let's recompile the Irrlicht library without DirectX 8:

1. Open the `source` folder and open the sub-folder `Irrlicht`.
2. Open the solution file corresponding to your Visual Studio version: If you are using Visual Studio 2003 then open `Irrlicht7.1.sln`, if you are using Visual Studio 2005 then open `Irrlicht8.0.sln`, or if you are using Visual Studio 2008 or higher open `Irrlicht9.0.sln`.
3. Expand the folder **include** in your solution explorer as follows:



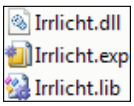
4. Double-click on `IrrCompileConfig.h` to open the file.

5. Go to line 120:

```
#if defined(_IRR_WINDOWS_API_) && (!defined(__GNUC__) || defined(IRR_COMPILE_WITH_DX9_DEV_PACK))

//! Only define _IRR_COMPILE_WITH_DIRECT3D_8_ if you have an appropriate DXSDK, e.g. Summer 2004
#define _IRR_COMPILE_WITH_DIRECT3D_8_
#define _IRR_COMPILE_WITH_DIRECT3D_9_

#endif

//! Define _IRR_COMPILE_WITH_OPENGL_ to compile the Irrlicht engine with OpenGL.
/** If you do not wish the engine to be compiled with OpenGL, comment this
define out. */
#define _IRR_COMPILE_WITH_OPENGL_

//! Define _IRR_COMPILE_WITH_SOFTWARE_ to compile the Irrlicht engine with software driver
/** If you do not need the software driver, or want to use Burning's Video instead,
comment this define out */
#define _IRR_COMPILE_WITH_SOFTWARE_

//! Define _IRR_COMPILE_WITH_BURNINGSVIDEO_ to compile the Irrlicht engine with Burning's video d
/** If you do not need this software driver, you can comment this define out. */
#define _IRR_COMPILE_WITH_BURNINGSVIDEO_
```

6. Comment out the define `#define _IRR_COMPILE_WITH_DIRECT3D_8_`.

7. Click on **Build | Build Solution**:



What just happened?

We have now built Irrlicht, which will take a while depending on your system, without DirectX 8 support and it will automatically be copied to the correct directories. If you would now launch one of the examples and select DirectX 8.1 as the rendering device the example would obviously not start.

Additional configurations

As you probably have seen when scrolling over the `IrrCompileConfig.h` file, there are a lot possible options to configure the Irrlicht library. You can switch different renderers on or off, if don't need them. All the setting flags are well commented. So this should be the first place to look at if you want to compile Irrlicht in a different way, rather than the default settings.

Remember that you have to recompile the complete Irrlicht library, if you change a define in the configuration file.

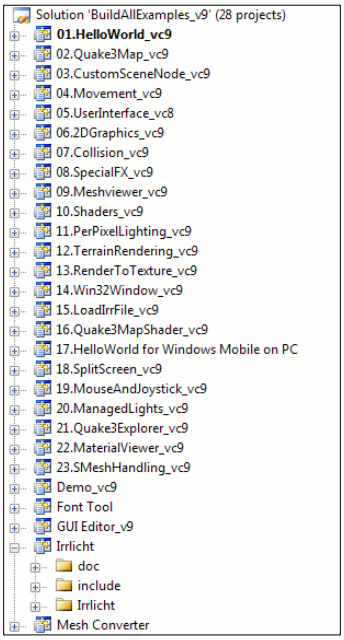
Building an example with Visual Studio

Now that we have set up Irrlicht with Visual Studio and know how to recompile and slightly modify the configuration file, let's see if everything is set up correctly and compile the first Irrlicht example.

Time for action – building an Irrlicht example

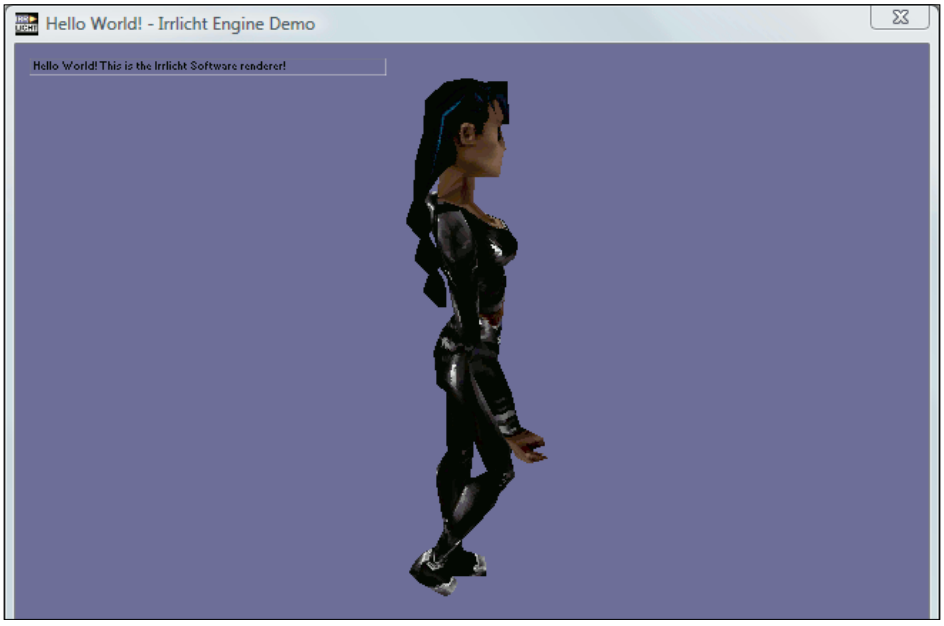
Let's compile the "Hello World" example:

1. Open the `examples` folder.
2. Open the solution file corresponding to your Visual Studio version: If you are using Visual Studio 2003 then open `BuildAllExamples_v7.sln`, if you are using Visual Studio 2005 then open `BuildAllExamples_v8.sln`, or if you are using Visual Studio 2008 or higher open `BuildAllExamples_v9.sln`.
3. Make sure `01.HelloWorld` is checked as the startup project. Right-click on `01.HelloWorld` and check **Set as Startup Project**:



4. Right-click on **01.HelloWorld_vc9** and click on **Rebuild**.

5. Press *Ctrl* + *F5* to start the example application without the debugger:



What just happened?

Because the Irrlicht examples are already pre-compiled for Visual Studio, just building the example will not suffice. If you would start the application from Visual Studio without step 4 the pre-compiled application would launch. That is why we have to rebuild the example.

Using Irrlicht with CodeBlocks

CodeBlocks is a free open source integrated development environment for C++, which uses the **GNU Compiler Collection** (**GCC**) by default.

Installing CodeBlocks on Windows

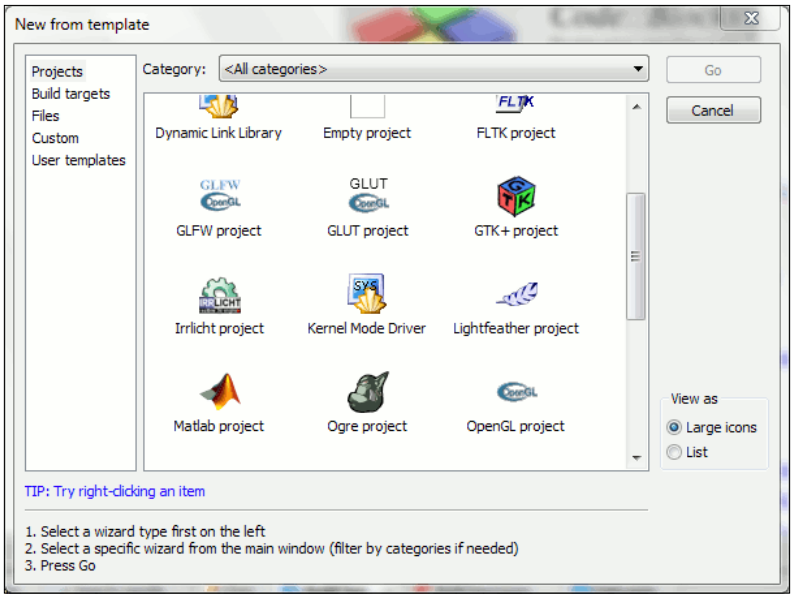
The latest release of CodeBlocks is 10.05. Go to `http://www.codeblocks.org/`, select **Downloads** from the top navigation bar. Then click on **Download the binary release** and download `codeblocks-10.05mingw-setup.exe` by clicking on **BerliOS**. Double-click on the downloaded file and follow the instructions of the installer.

If you don't feel comfortable using Visual Studio, CodeBlocks may be the right solution for you. While CodeBlocks may not offer as many advanced features as Visual Studio does, it has the advantage of being cross-platform and projects created with CodeBlocks can be easily used and shared on Windows, Linux, and Mac OS X. And there's no need for Redistributable files and so the application compiled with CodeBlocks won't necessarily need an installer and the `redist` packages. For example, Visual C++ needs Microsoft Visual C++ Redistributable package to install runtime components to run the applications created using Visual C++.

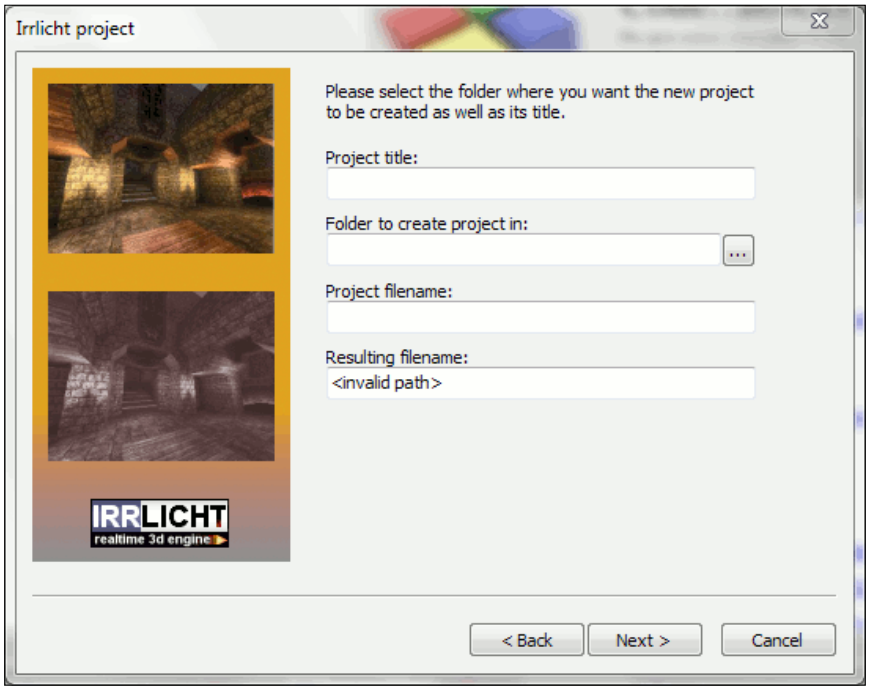
Time for action – creating an application using the CodeBlocks wizard

Let's create a new project with CodeBlocks:

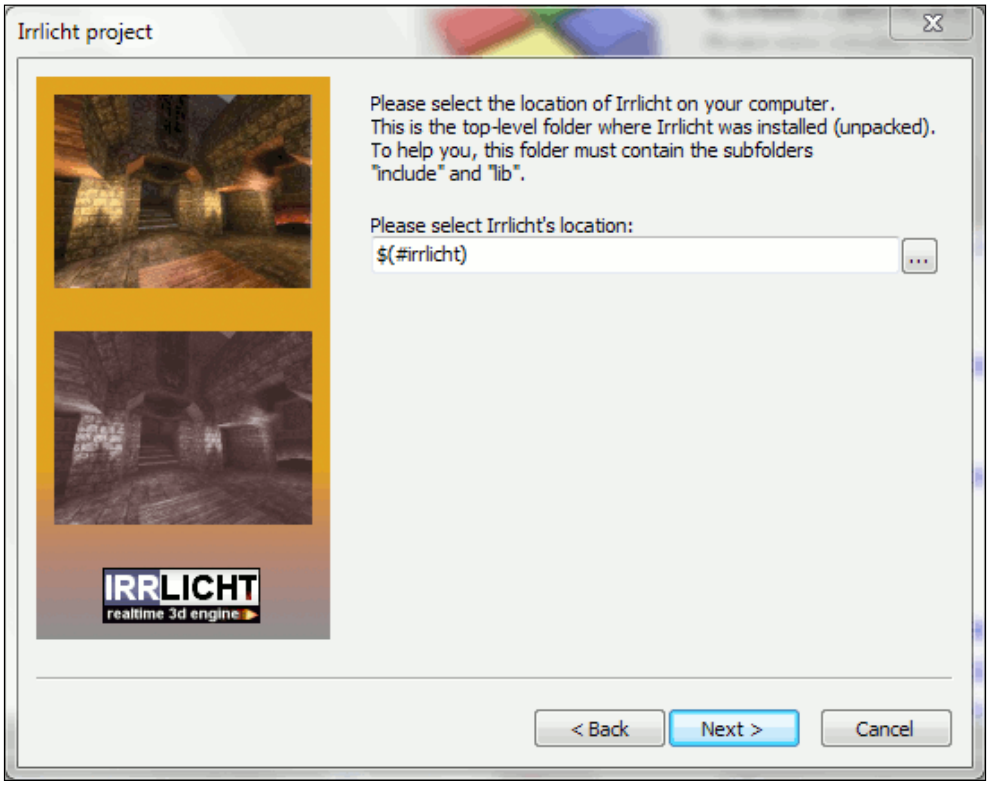
1. Click on **File | New | Project**.
2. Select **Irrlicht Project** and click on **Go**:



3. In this window, enter a project title and select a directory where the project should be created. If you are done, click on **Next** as follows:

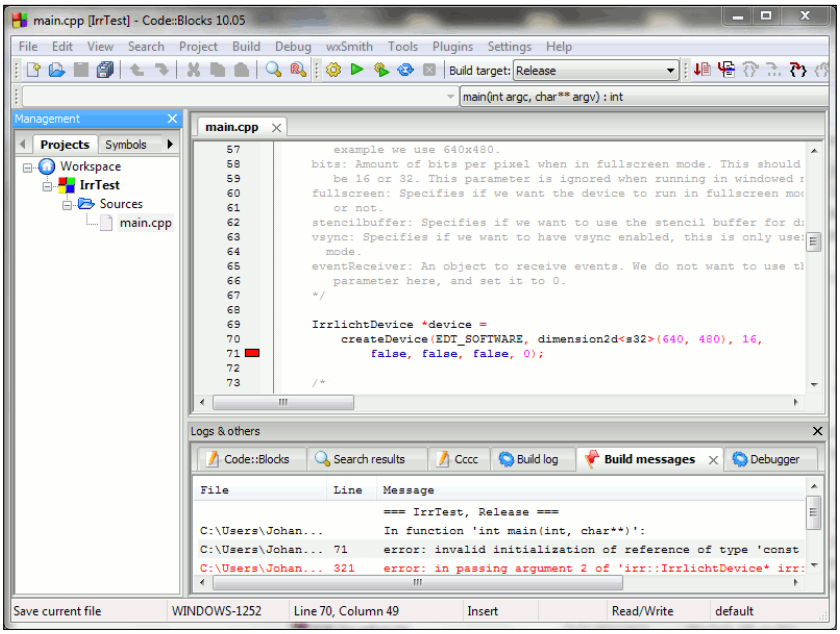


4. Next, we'll need to select the folder where you extracted Irrlicht. If you click on the browse button, a window called **Global Variable Settings** will pop up. Click **Close** to ignore this window. Now, we can select the directory where the Irrlicht engine is located:



5. The project has now been set up. If you try to compile and run the example using **Build | Build and run**, the example will not be built. You may get an error saying something as follows:

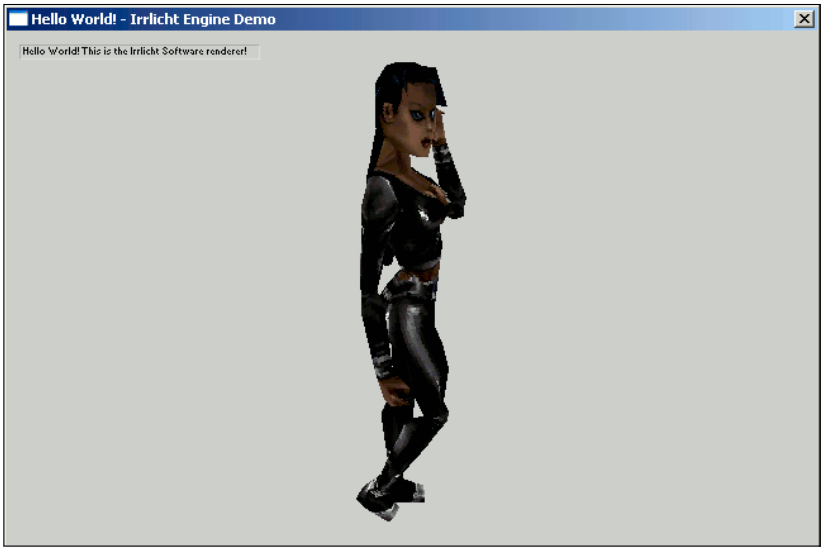
```
"error: in passing argument 2 of 'irr::IrrlichtDevice*
irr::createDevice(irr::video::E_DRIVER_TYPE, const
irr::core::dimension2d<unsigned int>&, irr::u32, bool, bool, bool,
irr::IEventReceiver*)'"
```



6. Change `dimension2d<s32>` to `dimension2d<u32>` in line 70.

```
IrrlichtDevice *device = createDevice(EDT_SOFTWARE,
                        diension2d<u32>(640, 480), 16, false,
                        false, false, 0);
```

7. The example will now compile successfully:

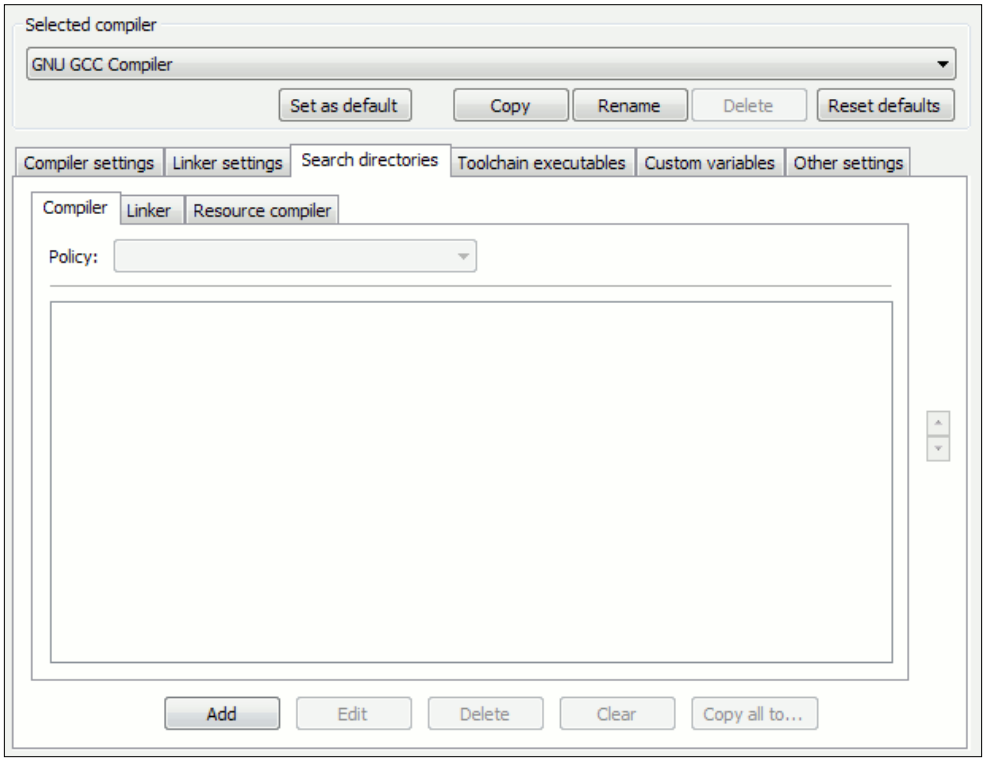


What just happened?

Using the project wizard is the easiest way to set up a new example project with CodeBlocks. The CodeBlocks Irrlicht template is the same as the "Hello World" example from the Irrlicht `examples` folder. Since Irrlicht 1.6, there have been some changes concerning data types of Irrlicht, which in this example is the change from a platform-independent signed integer type to a platform-independent unsigned integer type. Since the CodeBlocks template has not been updated to reflect this change, we have to modify this for ourselves.

CodeBlocks under Linux

If you wish to use CodeBlocks under Linux, the project wizard may not work properly. The better way is to add the `include` and `library` directories to the global search path. Click on **Settings | Compiler and debugger...**:



Switch to the **Search directories** tab. In the **Compiler** tab click on **Add** to add Irrlicht's `include` folder. Switch to the **Linker** and add the `lib` folder of Irrlicht. You still have to link against the Irrlicht library directly. This is done by right-clicking on the project file and opening the menu item **Build options...**. Switch to the tab **Linker settings** and add Irrlicht under **Link libraries**.

Irrlicht on Linux

Unlike on Windows, there is no precompiled library of Irrlicht for Linux.

Before you can actually build the library or develop with Irrlicht, you should make sure all necessary development packages are installed on your system. Which development packages you install strongly depends on the distribution you are using.

Getting started using Ubuntu

If you are using Ubuntu, you need to have the following packages installed:

`build-essential`
`xserver-xorg-dev`
`x11proto-xf86vidmode-dev`
`libxxf86vm-dev`
`mesa-common-dev`
`libgl1-mesa-dev`
`libglu1-mesa-dev`
`libxext-dev`
`libxcursor-dev`

To install those packages use either the graphical package manager called **Synaptic** or the command-line utility `apt-get`. You will need `superuser` privileges to perform this action.

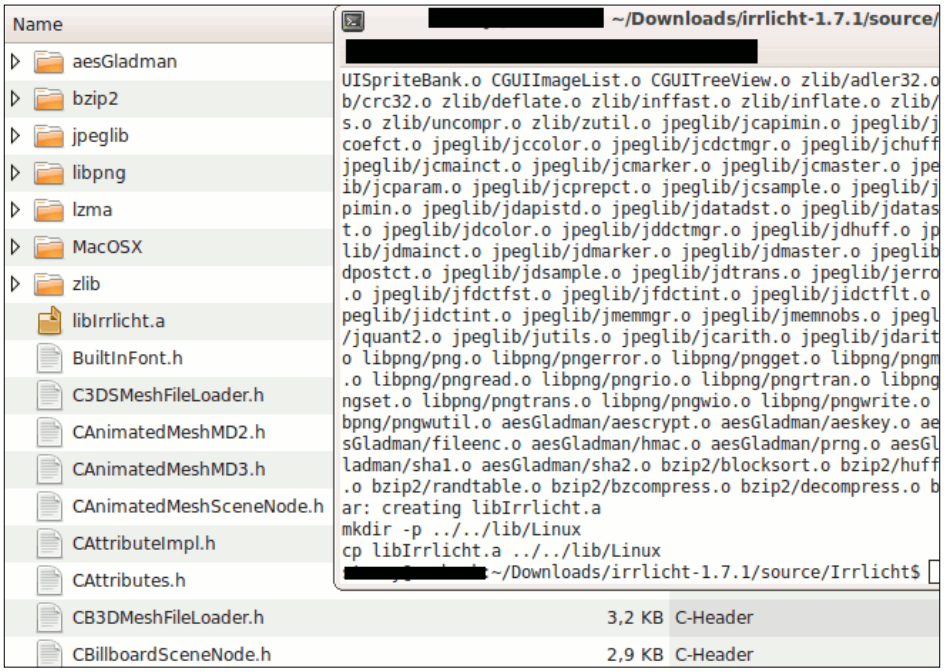
Make sure you have the latest video card drivers installed or you may have errors when building or executing your applications like black screen or missing graphics.

Because there is no DirectX implementation for Linux, you only have software renderers and the OpenGL renderer at your disposal. There is a way around this, that is to use Windows version of CodeBlocks under Linux using Wine. But that topic is out of the scope of this book.

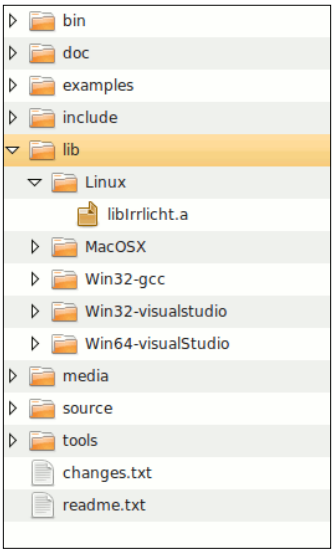
Time for action – compiling the static library

Let's get started and build the static library:

1. Go to the `source` folder of your extracted Irrlicht package.
2. Open the sub-folder `Irrlicht` and open the command line.
3. Type in `make` and after a few moments the static library will be built in this folder:



4. The `make` file automatically copies the static library to `lib` folder of Irrlicht:



What just happened?

The `make` file in the `source` folder contains a series of commands that will build the static Irrlicht library on Linux.

Making Irrlicht available on the whole system

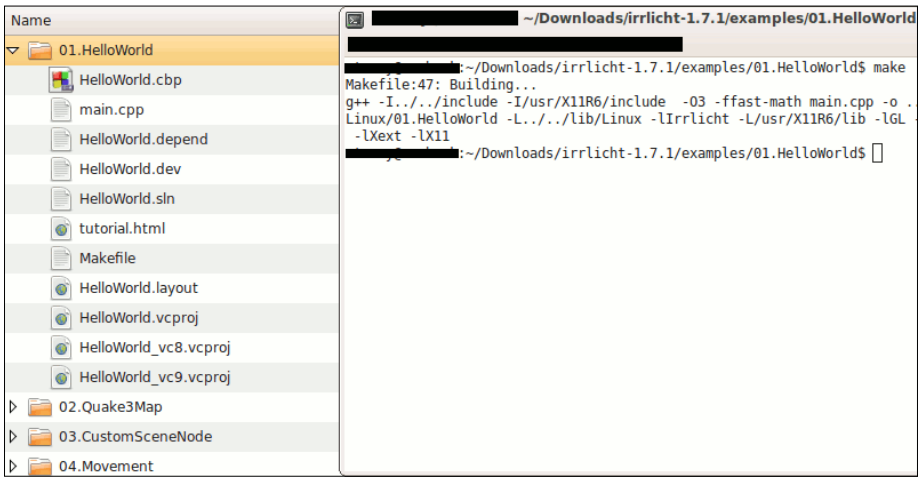
Although this is not necessary, it is recommended to copy `libIrrlicht.a` to `/usr/lib` and the `include` files from Irrlicht to `/usr/include`. This will require `root` or `superuser` privileges. If this step is done, you don't need to set any additional global search paths in any IDE.

Time for action – compiling "Hello World" on Ubuntu

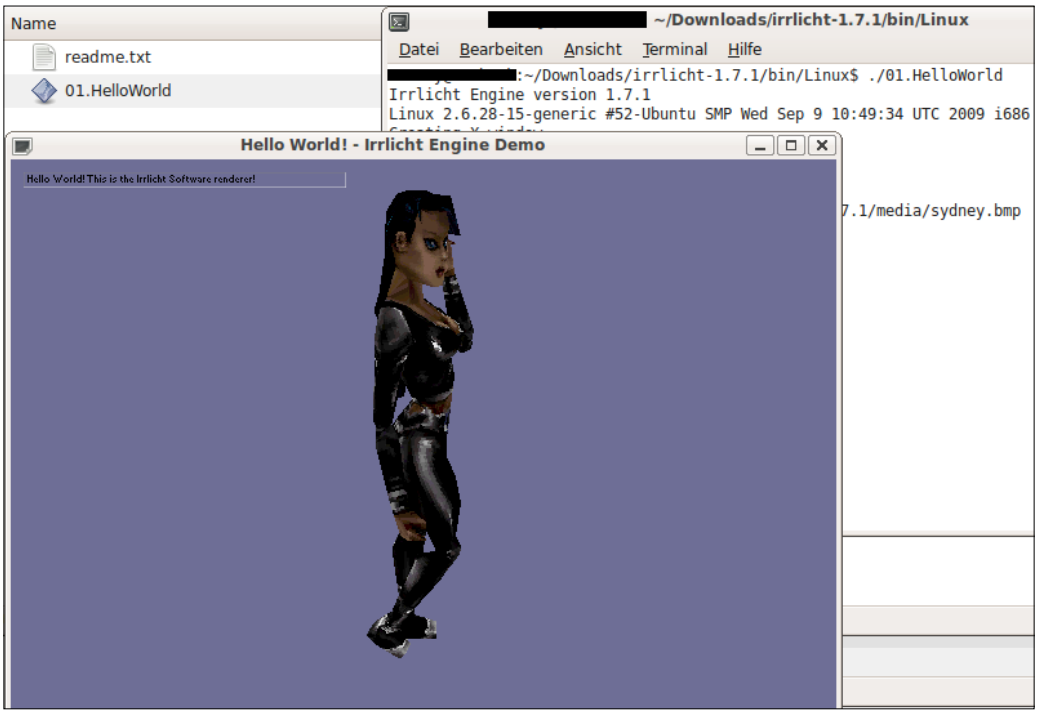
Now let's see if everything is working and compile the "Hello World" example from Irrlicht.

1. Go to the `examples` folder of your extracted Irrlicht package.
2. Open the sub-folder `01.HelloWorld` and open the command line.

3. Type in `make` and after a few moments the example will be built:



4. The binary is copied to the `bin/Linux` folder of the extracted Irrlicht package. If you launch the application, we see that everything is working:



What just happened?

After we have built the static library on Linux, we are now able to compile any example from the Irrlicht `examples` folder. To test if everything is working, we built the "Hello World" example. It is recommended that you launch the examples from the command line because some Irrlicht examples require the user to choose a renderer before the application itself starts.

Irrlicht on Mac OS X with Xcode

Before we start, make sure that you have the latest software updates installed that also include updates to the graphics drivers. To be able to develop with Irrlicht on Mac OS X, you need Xcode, the standard integrated development environment for Mac OS X that also uses GCC as its default compiler.

Getting Xcode

To use Xcode, you need to register as an Apple Developer on their site `http://developer.apple.com`.

After you are a fully registered Apple developer—which should only take a few minutes—you will be able to download Xcode on `http://developer.apple.com/technologies/xcode.html`. To be able to download Xcode, you need be logged in with your Apple Developer account.

Warning: The download is about 2.3 gigabytes large

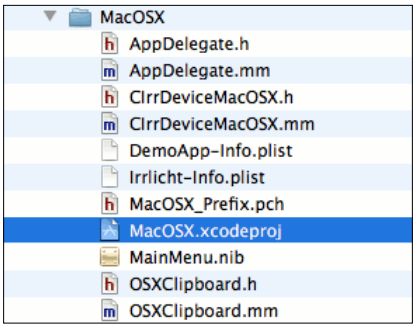
Like on Linux, Irrlicht is not able to use DirectX 8 or 9, because that is not available for this platform. Unfortunately, the software renderer has a few issues on Mac OS X, so we will have to change this manually.

But first things first.

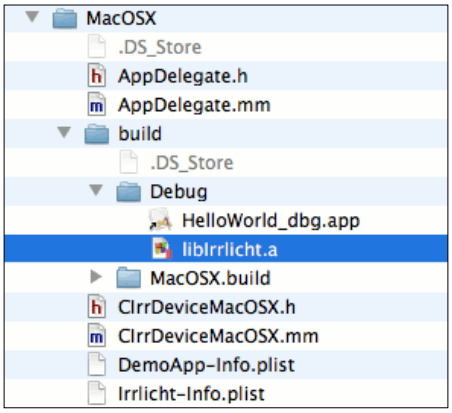
Time for action – compiling the static library on Mac OS X

We need the static library to be able to use Irrlicht:

1. Go to the `source` folder and open the sub-folder `MacOSX`.
2. Double-click on `MacOSX.xcodeproj`:



3. Xcode will open. Change **Active Target** to `libIrrlicht.a`. Click on **Build | Build** to create the static library, which will then show up in the source directory:



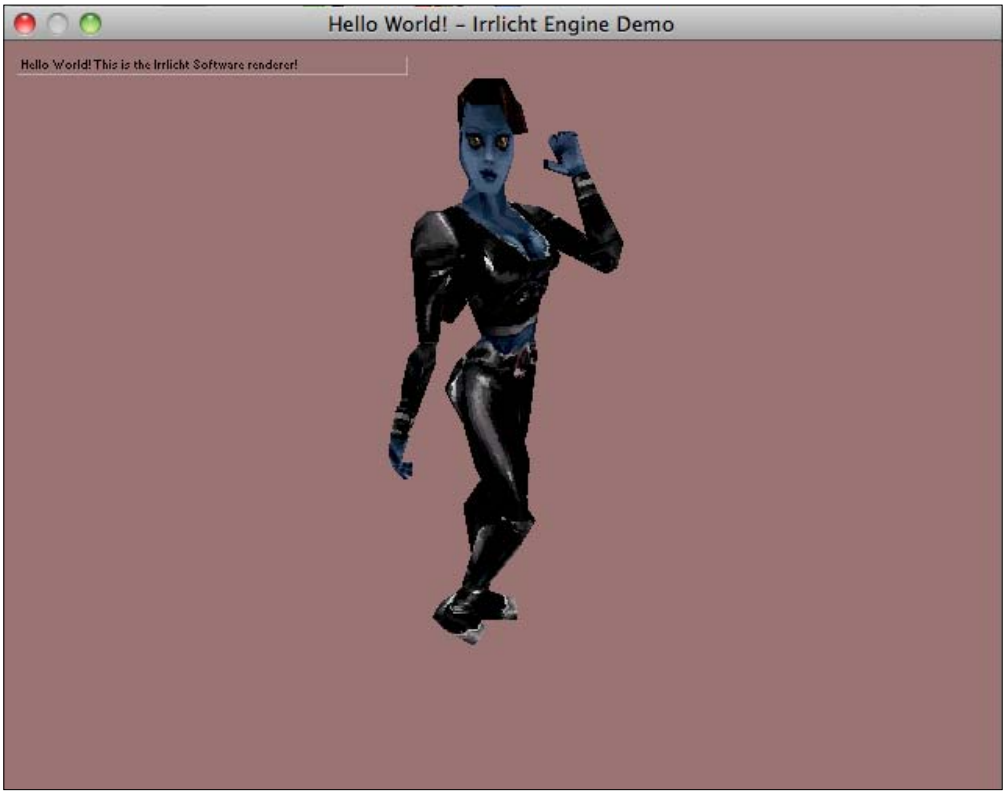
What just happened?

The library has just been compiled. It is recommended to keep the library at a place where you find it easier, for example, copy the library to `lib/MacOSX` of your extracted Irrlicht package.

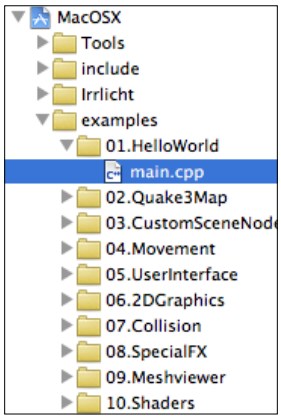
Time for action – compiling "Hello World" project with Xcode

Let's try to compile one of the Irrlicht examples to see if everything is working correctly.

1. Switch back your Xcode window and change **Active Target** to `01.HelloWorld`.
2. If you are developing on an Intel Mac (for example, every Mac produced since 2006), check if `i386` is selected in **Active Architecture**.
3. If you already tried **Build & Run**, you will have noticed that the colors were distorted or maybe even a blank screen:

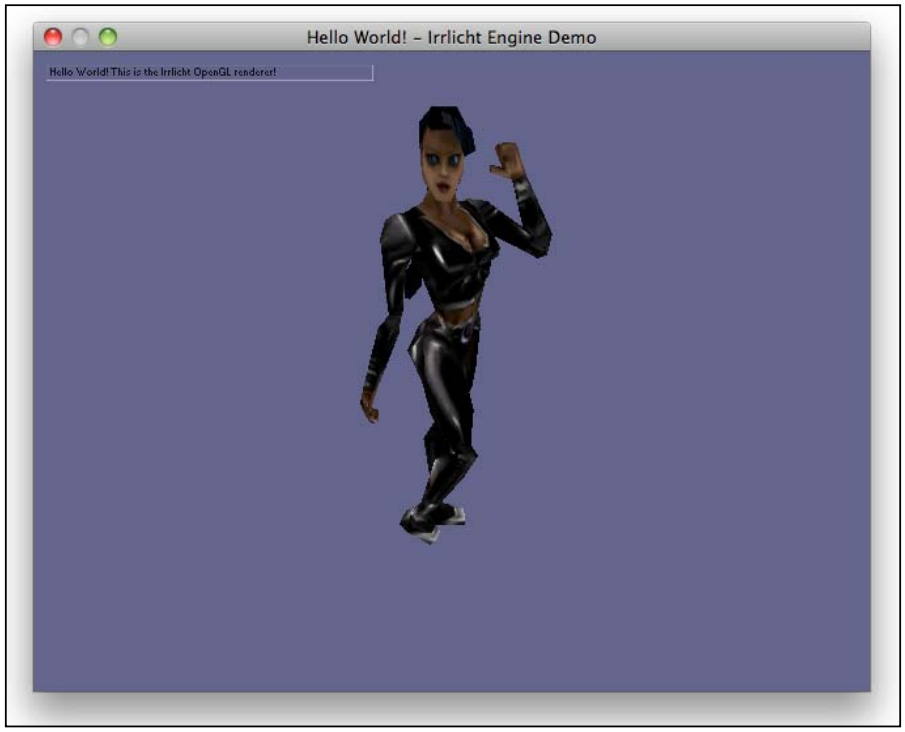


4. Expand `examples` and expand its sub-folder `01.HelloWorld`. Double-click on `main.cpp` to open this file:



5. Go to line 125 and change `video::EDT_SOFTWARE` to `video::EDT_OPENGL`.

6. Click on **Build & Run**. The colors are not distorted any more and the example will run smoothly:



What just happened?

To test if Irrlicht is working correctly on Mac OS X, we compiled and ran an example. There are some problems concerning the software renderer from Irrlicht in combination with Mac OS X, so we had to modify the device creation and force Irrlicht to use the OpenGL renderer instead of the software renderer.

Summary

We learned about how the Irrlicht engine is organized and what the contents inside the package are. We also got a first glance at the Irrlicht application by running the example project that comes with the package.

Specifically, we covered:

- Where and how to get Irrlicht
- The portability of Irrlicht—Setting up Irrlicht on different platforms, compilers, and integrated development environments
- Compiling the Irrlicht library

Now that we've learned about setting up our development environment with Irrlicht, we're ready to create a skeletal template application for our upcoming exercise projects.

2
Creating a Basic Template Application

Now that we know how to set up our Irrlicht development environment, it is time to get started with programming our very first application with Irrlicht.

In this chapter, we will:

- Create a new empty project
- Explain the use of an Irrlicht device
- Use the "game loop"

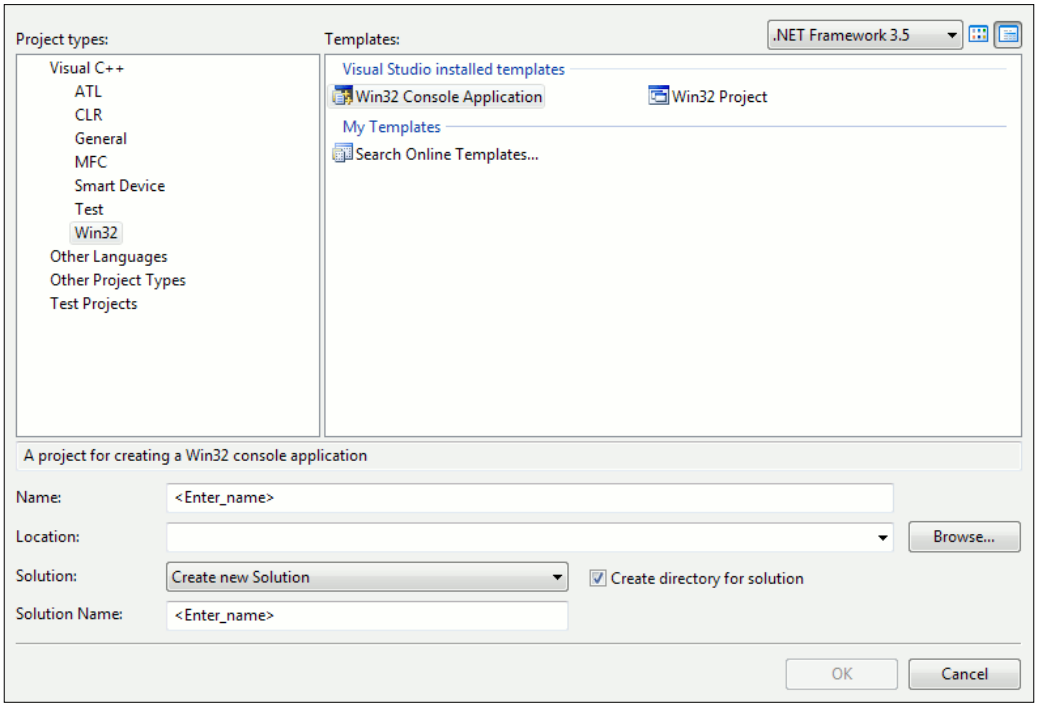
So let's get on with it...

Creating a new empty project

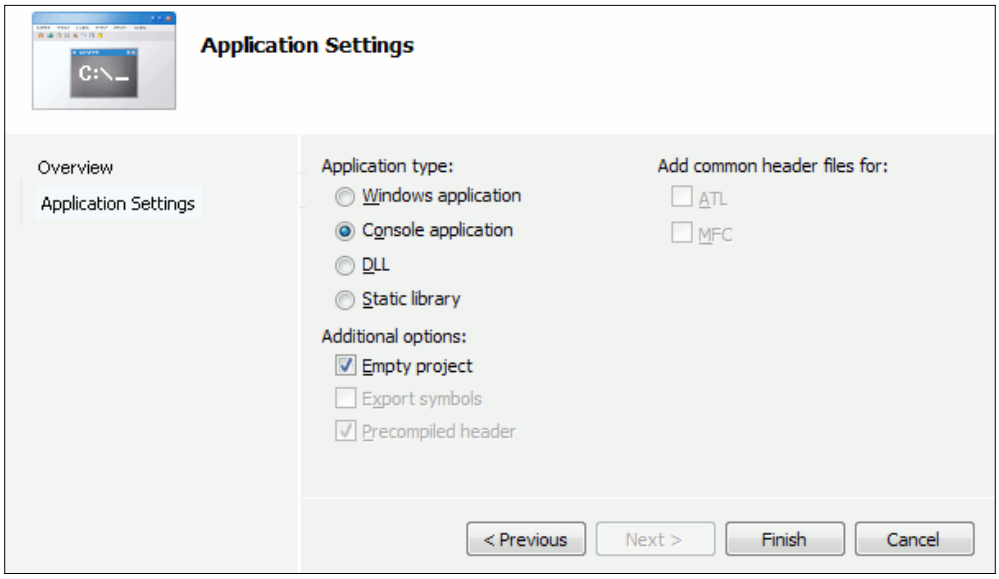
Let's get started by creating a new project from scratch. Follow the steps that are given for the IDE and operating system of your choice.

Visual Studio

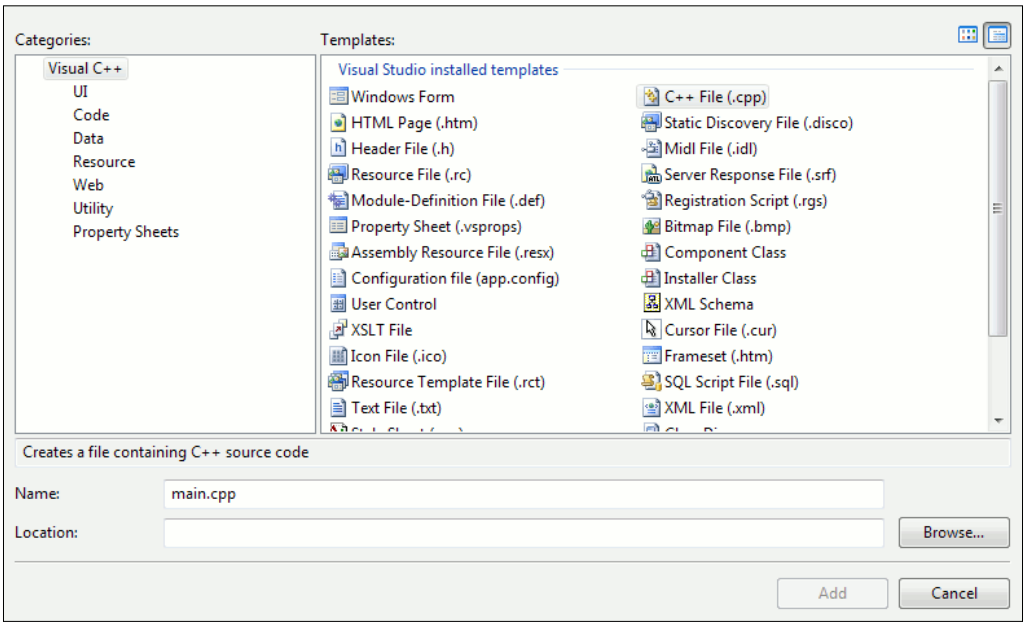
Open Visual Studio and select **File | New | Project** from the menu. Expand the **Visual C++** item and select **Win32 Console Application**. Click on **OK** to continue:



In the project wizard click on **Next** to edit the **Application Settings**. Make sure **Empty project** is checked. Whether **Windows application** or **Console application** is selected will not matter. Click on **Finish** and your new project will be created:



Let's add a main source file to the project. Right-click on **Source Files** and select **Add | New Item...**. Choose **C++ File(.cpp)** and call the file `main.cpp`:



CodeBlocks

Use the CodeBlocks project wizard to create a new project as described in the last chapter. Now double-click on `main.cpp` to open this file and delete its contents. Your main source file should now be blank.

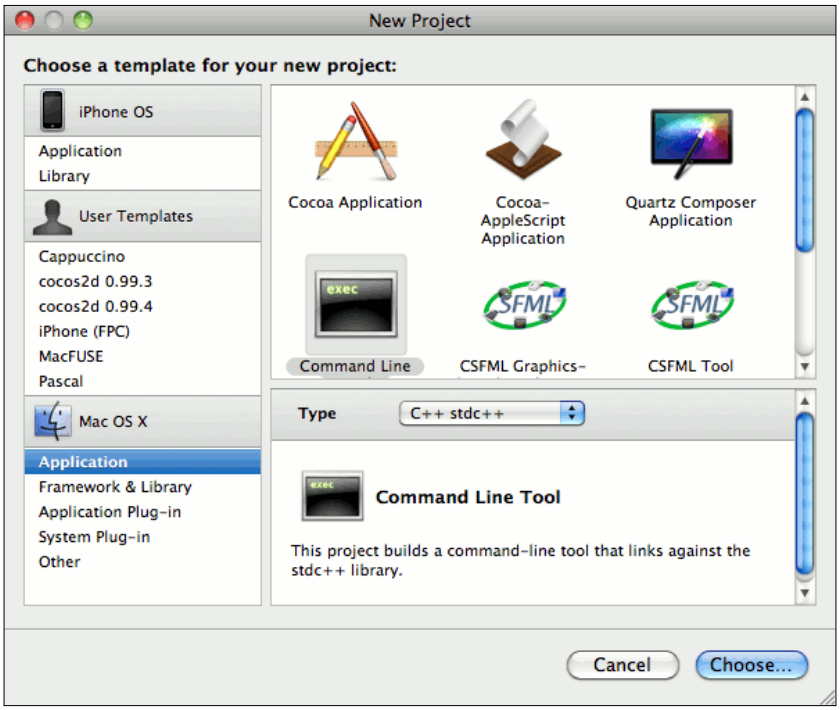
Linux and the command line

Copy the `make` file of one of the examples from the Irrlicht `examples` folder to where you wish to create your new project. Open the `make` file with a text editor of your choice and change, in line 6, the target name to what you wish your project to be called.

Additionally, change, in line 10, the variable `IrrlichtHome` to where you extracted your `Irrlicht` folder. Now create a new empty file called `main.cpp`.

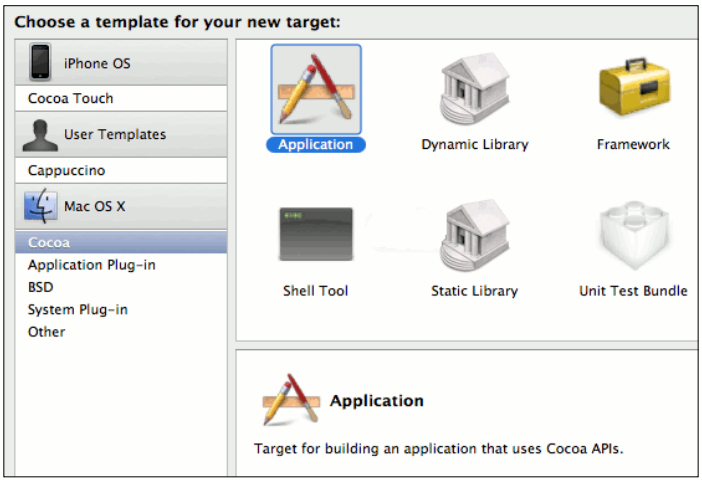
Xcode

Open Xcode and select **File | New Project**. Select **Command Line Tool** from **Application**. Make sure the type of the application is set to **C++ stdc++**:

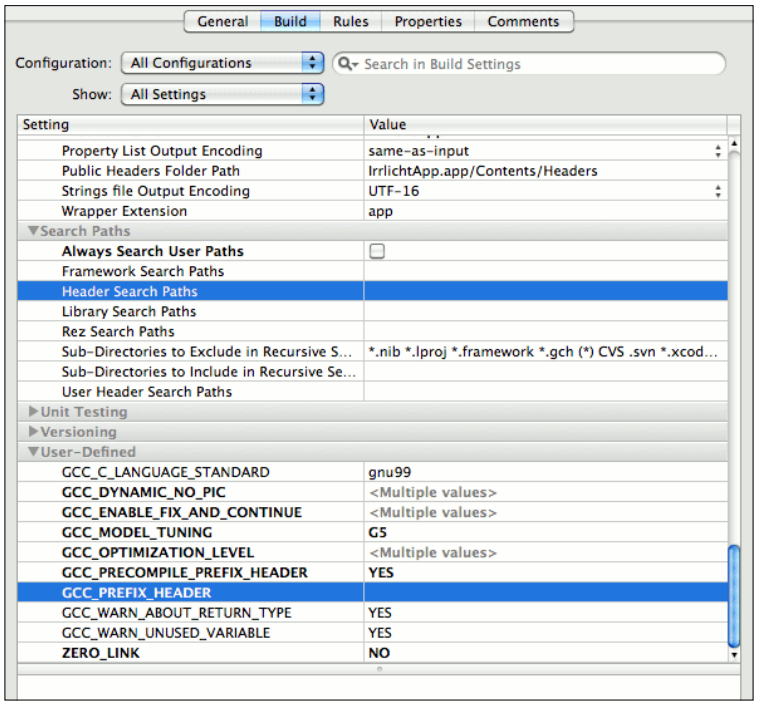


When your new project is created, change **Active Architecture** to **i386** if you are using an Intel Mac, or **ppc** if you are using a PowerPC Mac.

Create a new target by right-clicking on **Targets**, select **Application** and click on **Next**. **Target Name** represents the name of the compiled executable and application bundle:



The target info window will show up. Fill in the location of the `include` folder of your extracted `Irrlicht` package in the field **Header Search Path** and make sure the field **GCC_PREFIX_HEADER** is empty:



Right-click on the project file and add the following frameworks by selecting **Add | Existing Frameworks...**:

- `Cocoa.framework`
- `Carbon.framework`
- `IOKit.framework`
- `OpenGL.framework`

Now, we have to add the static library named `libIrrlicht.a` that we compiled in Chapter 1, *Installing Irrlicht*. Right-click on the project file and click on **Add | Existing Frameworks...**. Now click on the button **Add Other...** and select the static library.

Delete the original compile target and delete the contents of `main.cpp`.

Time for action – creating the main entry point

Now that our main file is completely empty, we need a main entry point. We don't need any command-line parameters, so just go ahead and add an empty `main()` method as follows:

```
int main()
{
    return 0;
}
```

If you are using Visual Studio, you need to link against the `Irrlicht` library. You can link from code by adding the following line of code:

```
#pragma comment(lib, "Irrlicht.lib")
```

This line should be placed between your include statements and your `main()` function.

If you are planning to use the same codebase for compiling on different platforms, you should use a compiler-specific define statement, so that this line will only be active when compiling the application with Visual Studio.

```
#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif
```

Using Irrlicht namespaces

The Irrlicht 3D graphics engine is structured into different namespaces. Let's take a look at them:

Namespace	Description
`irr`	Everything in the Irrlicht 3D graphics engine is to be found in the `irr` namespace.
`core`	The `irr::core` namespace contains basic data types like vectors, lists, and arrays.
`scene`	The `scene` namespace—as the name suggests—covers everything that has to do with scene management.
`video`	Anything that has to do with 2D and 3D rendering or accessing the video driver is found in the `video` namespace.
`io`	If you would need to load or save data from files such as XMLs or packaged archives from your local hard drive, you would need to use methods from the `io` namespace.

Make sure to add: `#include <irrlicht.h>` at the top of your main source file.

And to make the function calls shorter, we'll add the following using directives to the previously mentioned namespaces. This way we don't need to specify the namespace whenever we want to call a function from that namespace:

```
using namespace irr;
using namespace core;
using namespace video;
```

But you should be aware that using such a `using namespace` is considered bad practice according to some coding standards. Because exposing all the names from the root namespace can cause some confusions, for example, what if two different namespaces contain a function with the same name? That's why using only the top-level namespace is encouraged and the low-level namespace developers should use the full qualified name. But in this chapter, we'll just use this way to make our code shorter.

Irrlicht device

The Irrlicht device is the core object that we need to interact with the Irrlicht engine. It is created using the device driver of choice such as Direct3D, OpenGL, and so on. Basically, it's an interface to the actual underlying graphics hardware. So, we need to create a device first to be able to draw anything on the screen.

Time for action – creating an Irrlicht device

Creating a device in Irrlicht is as easy as calling a single function. If you have prior experience with graphics APIs such as Direct3D or OpenGL, you will know there's a lot of code involved to set up and create a device. So let's create an Irrlicht device.

1. Go into your `main()` function.
2. Define a variable called `device` by writing `IrrlichtDevice* device`.
3. Create our Irrlicht device by calling:

```
createDevice(EDT_OPENGL, dimension<u32>(640, 480), 16, false,
             false, false, 0);
```

4. Add the condition `if (!device) return 1;` immediately after that.
5. Finally we need to release our device with `device->drop();`.

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);
    if (!device)
       return 1;

    device->drop();

    return 0;
}
```

What just happened?

At the moment, a new window with the specified parameters of `createDevice` will be created, but will be instantly closed. If there is a problem with the created device, the application will receive the `return 1` code.

The Irrlicht device handles everything that has to do with creating and using the window that we need to draw anything on the screen.

The createDevice method

The `createDevice` has seven parameters. The first one is the renderer that we will be using. Because we want our application to be platform-independent, we are going to use OpenGL, which is available for Windows, Linux, and Mac OS X.

You can also use the DirectX 8 renderer with `EDT_DIRECT3D8` or the DirectX 9 renderer with `EDT_DIRECT3D9` on Windows. Irrlicht also provides two software renderers that are `EDT_SOFTWARE` and `EDT_BURNINGSVIDEO`. If the first parameter is set to `EDT_NULL`, there won't be any window created.

The second parameter sets the window size, while the third parameter sets the color depth, which should be set to either 16 bit or 32 bit. If the application is going to be run in windowed mode, this parameter might be ignored and the color depth of the created window will be set to the depth of the desktop.

The fourth parameter determines whether or not the application should run in windowed or fullscreen mode. If the parameter is set to `true` the application will launch in fullscreen mode. If this is the case the sixth parameter is a boolean flag that can switch vertical synchronization (vsync) on or off.

The fifth parameter specifies whether to use the stencil buffer or not. If we would need to draw shadows, we need to set this parameter to `true`, but for now, we can leave this to `false`.

The seventh parameter should be used if we are going to use an event receiver; that does not interest us yet.

The "game loop"

You already know `while` and `for` loops, so what's the deal with this alleged "game loop"? In game development, we usually have a scene that needs to be updated in every frame. We would need to check for game logic events, like checking for collisions or if the player has won or lost the game then update the scene accordingly. The Irrlicht 3D graphics engine uses the same approach. Assume your application is like an interactive movie, each frame redraws the complete screen.

Time for action – creating the "game loop"

Let's create our game loop:

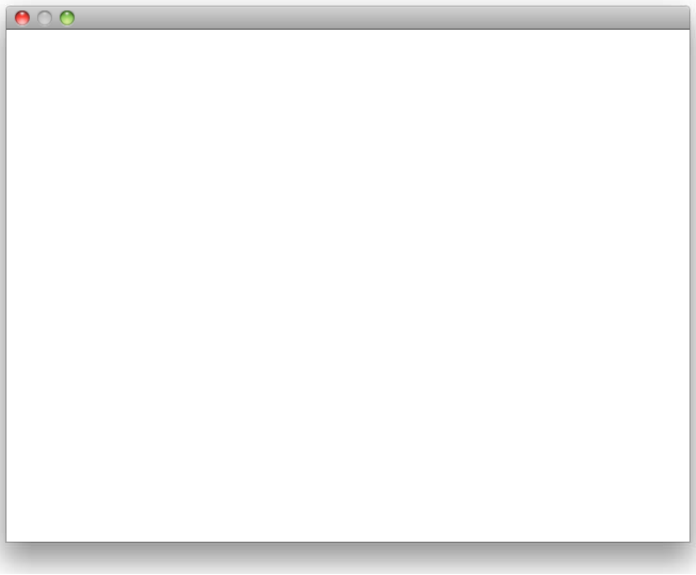
1. The "game loop" is a simple `while` loop. We will have to redraw each frame as long as the device is running.
2. Define the variable `IVideoDriver* driver` and allocate `device->getVideoDriver()`.
3. Now jump to the "game loop".
4. Add `driver->beginScene(true, true, SColor(255, 255, 255, 255));` at the beginning of the loop.
5. Add `driver->endScene();` at the end of the loop.

   Your source code should look something like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();

    while (device->run())
    {
         driver->beginScene(true, true, SColor(255, 255, 255,
                            255));
         driver->endScene();
    }
```

```
    device->drop();

    return 0;
}
```

You should see a blank window created as follows, if it runs successfully:



What just happened?

We just succeeded in creating our first application with Irrlicht that displays a window with a plain white background. To be able to draw within our "game loop", we need to get information about the video driver we are using. To save us some work, by not having to write `device->getVideoDriver()` each time, we would need something from the video driver; we define a variable that gets an instance from the video driver. You may have noticed the `I` in front of the type `IVideoDriver` tells you that this type is an interface. Irrlicht relies heavily on interfaces to be able to use different renderers under a common API.

beginScene

Drawing graphics to the display screen directly, every time, can cause the flickering artifact. To avoid this problem a method called double buffering is widely used. The idea is to use a secondary back buffer before drawing to the actual screen. So the renderer will draw on that back buffer first. Only once the drawing is finished, it'll flip the back buffer with the front buffer. The first parameter of `beginScene` method specifies if the back buffer should be cleared and set to `true` by default. The second parameter specifies whether to clear the Z buffer.

The third parameter is the color that will be used to clear the back buffer. The color type consists of four values alpha, red, green, and blue ranging from 0 to 255.

This method has to be called first before any rendering occurs.

endScene

The `endScene` method flips the screen buffers and draws everything on the screen.

Pop quiz – namespaces and beginScene

1. Irrlicht is organized into different namespaces. Why is this useful?
    a. To have an easy and neat structure of the engine
    b. To use as many header files as possible
2. What would happen if we set the first two parameters of the `driver->beginScene` method to `false`?
    a. Nothing
    b. The current frame will be drawn over the last frame
    c. Only a black empty screen will appear

Have a go hero – creating the fade-in/fade-out effect

Now that we have set up our base application with a white background, let's try to integrate a fade-in and fade-out effect into our application. If the background is white, we want to slowly fade to a black background and vice versa. So basically, you'll have to play around with the color values inside your game loop.

You figured it out? Compare it with this approach:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
```

```
IrrlichtDevice *device = createDevice(EDT_OPENGL,
                         dimension2d<u32>(640, 480), 16, false,
                         false, false, 0);

if (!device)
    return 1;

IVideoDriver* driver = device->getVideoDriver();

f32 bg_r = 255.0f;
f32 bg_g = 255.0f;
f32 bg_b = 255.0f;

bool fadeOut = true;

int lastFPS = -1;

u32 then = device->getTimer()->getTime();

const f32 fadeRate = 0.1f;

while(device->run())
{
     const u32 now = device->getTimer()->getTime();
     const f32 frameDeltaTime = (f32)(now - then);
     then = now;

   if (bg_r <= 0.0f) fadeOut = false;
   else if (bg_r >= 255.0f) fadeOut = true;

   if (fadeOut)
   {
      bg_r-= fadeRate * frameDeltaTime;
      bg_g-= fadeRate * frameDeltaTime;
      bg_b-= fadeRate * frameDeltaTime;
   }
    else
    {
      bg_r+= fadeRate * frameDeltaTime;
      bg_g+= fadeRate * frameDeltaTime;
      bg_b+= fadeRate * frameDeltaTime;
    }
```

```
            driver->beginScene(true, true, SColor(255, (u32)bg_r,
                               (u32)bg_g, (u32)bg_b));

            driver->endScene();
        }

        device->drop();

        return 0;
    }
```

However, the preceding approach contains a new concept that is called frame independent update. Before starting any rendering, we calculate the total time (ticks) difference between each frame, and multiply that with the rate we want to use for fade-in, fade-out. This way we can achieve a smooth transition across different PCs. Otherwise, if you just use a constant value to subtract or add to the color values, your program will fade-in/out very fast on fast computers.

Summary

We set up our development environment and created our first application.

Specifically, we covered:

- Creating an empty application using different platforms and IDEs
- Irrlicht namespaces
- Creating and using a device
- Introduction of the frame independent update concept

Now that we've created a base template application, let's dig a bit deeper into the Irrlicht 3D engine and learn more about polygon meshes—which is the topic of the next chapter.

3
Loading Meshes

Now that we know how to set up our Irrlicht development environment, it is time to get started with programming our first application with Irrlicht.

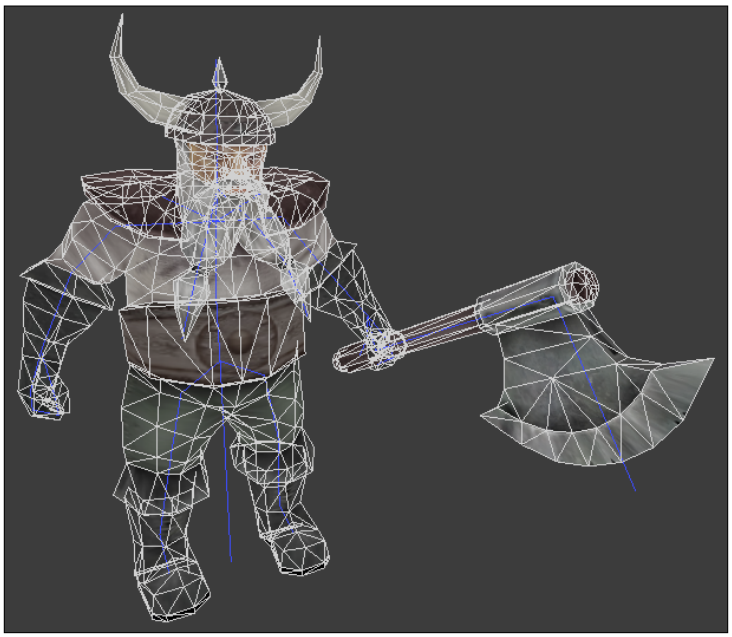
In this chapter, we shall:

- Learn what a polygon mesh is
- Load a mesh from Irrlicht's `media` folder
- Apply a texture to our mesh
- Load and access our mesh in our application
- Animate our mesh
- Model a mesh using Blender

So, let's take a look at it...

What is a mesh?

A polygon mesh is basically a construct of vertices, faces, and edges that defines the shape of an object which are then rendered on the screen by one of the chosen renderers that Irrlicht provides:



You might already know that there are different file formats for storing mesh data, for example, OBJ, MD2, MD3, and so on. Irrlicht supports 15 different file formats out of the box, so you should not run into problems when exporting your own model. These formats are 3DS, B3D, BSP, IRRMESH, LMTS, LWO, OBJ, MD2, MD3, MS3D, MY3D, OGRE, PLY, STL, and the X file format. Frequently used file formats are explained later in this chapter.

If you want to import a format that Irrlicht doesn't support, you can always write your own mesh file loader by extending the `IMeshLoader` interface that Irrlicht has provided. And hopefully, you can contribute back to the community again by distributing your newly-created loader class for Irrlicht. It's always a good idea to involve yourself in forums if you need that kind of support. Somebody probably must have already written what you need and posted it on the forum.

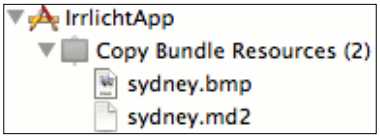
Time for action – loading a mesh

Now, we are going to load and display a mesh in our application. Let's take the template application, we made in Chapter 2, *Creating a Basic Template Application* and take it a bit further:

1. Open the `media` folder of your extracted Irrlicht package.
2. Copy `sydney.md2` and `sydney.bmp` to where the executable of your new application is:

Note for Xcode users:

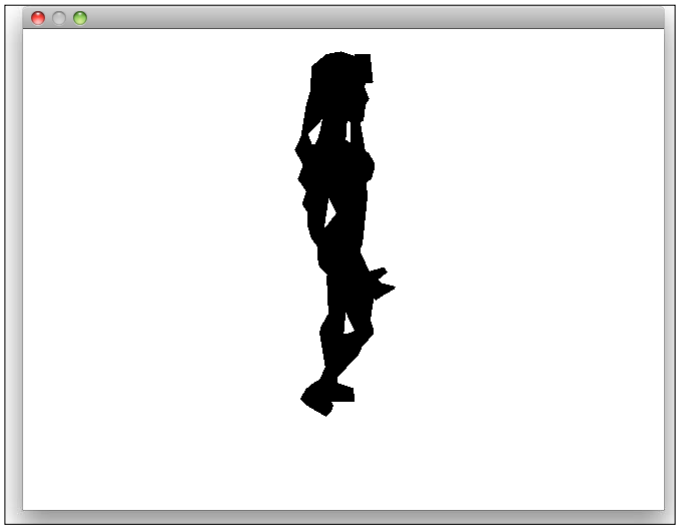
Right-click on the Xcode project and **Add | Existing Files...** and select `sydney.md2` and `sydney.bmp`. Check the option Copy items into destination group's folder (if needed). Make sure those two files are listed in **Copy Bundle Resources** in your build target.



3. Add the line `ISceneManager* smgr = device->getSceneManager();` directly after where we create our instance to the video driver.
4. To actually load a mesh, insert `IMesh* mesh = smgr->getMesh("sydney.md2");`.
5. Now that our mesh is loaded, we still can't see our mesh. We need a node to display the mesh in our scene. Let's create one by adding this code: `IMeshSceneNode* node = smgr->addMeshSceneNode(mesh);`.
6. Our node is added to the scene, but we need to adjust the camera to be able to actually see the mesh. Add this line: `smgr->addCameraSceneNode(0, vector3df(0, 30, -40), vector3df(0, 5, 0));`.
7. Add the line `smgr->drawAll();` into our "game loop" between the `beginScene()` and `endScene()` function.

Your complete source code should look like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace scene;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
       return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    IMesh* mesh = smgr->getMesh("sydney.md2");
    IMeshSceneNode* node = smgr->addMeshSceneNode(mesh);

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
                             vector3df(0, 5, 0));

    while (device->run())
    {
          driver->beginScene(true, true, SColor(255, 255, 255,
                                                255));

        smgr->drawAll();

        driver->endScene();
    }

    device->drop();

    return 0;
}
```



What just happened?

We just copied a ready-to-go mesh from the Irrlicht `media` folder to demonstrate some of the possibilities of what we can do with meshes.

In step 3, we retrieved the instance of a scene manager from our Irrlicht device object. A scene manager is responsible for everything inside our scene, including the meshes we want to render. It holds several child nodes, some of which are drawable objects while others are used just for references such as camera nodes.

After that we load our mesh file and store it in an `IMesh` object. But, to actually draw this mesh on screen, we need to add this mesh as a child node under our scene manager so that it knows about the existence of our mesh node. Step 5 shows how to add that node to the scene manager with the initial position and rotation vectors.

In step 6, we set up a camera to be able to see the mesh. The first parameter is the parent scene node, which we leave at `0` at the moment, because this camera does not have a parent node. We could attach our mesh scene node as the parent to make our camera follow wherever our mesh goes. The second parameter is the position of the camera in the scene, while the third parameter is the point where the camera looks at.

The `drawAll()` function is the most important line in this code snippet, it makes sure that everything we have added to the scene manager will be drawn.

Differences between mesh formats

In this example, we added a mesh with the extension MD2. Let's take a quick look at the most popular mesh formats and their advantages and disadvantages:

OBJ

The OBJ file format has been developed by Wavefront Technologies. It is one of the simplest mesh formats and stores a list of vertices, normals, and texture coordinates. There can be a reference to a material file, but the material or any other data, besides the object geometry itself, is not stored in this file format. An OBJ file can be viewed and edited in any text editor.

MD2/MD3

These file formats were created by ID Software for Quake 2 and Quake 3 respectively. They are very common in game development and can store additionally to the geometry itself as an animation data.

COLLADA

COLLADA is a relatively new mesh format that uses XML file specifications and can be easily edited using any text editor. COLLADA files can store not only a single mesh, but also the whole scene with multiple meshes together with lights, cameras, and so on. Irrlicht loader for COLLADA files can set up a scene as described in the file.

X

This is the mesh file format developed by Microsoft along with DirectX. It can be saved as text or binary and can also store animations and skeleton information as well, in addition to the geometry data.

Using textures

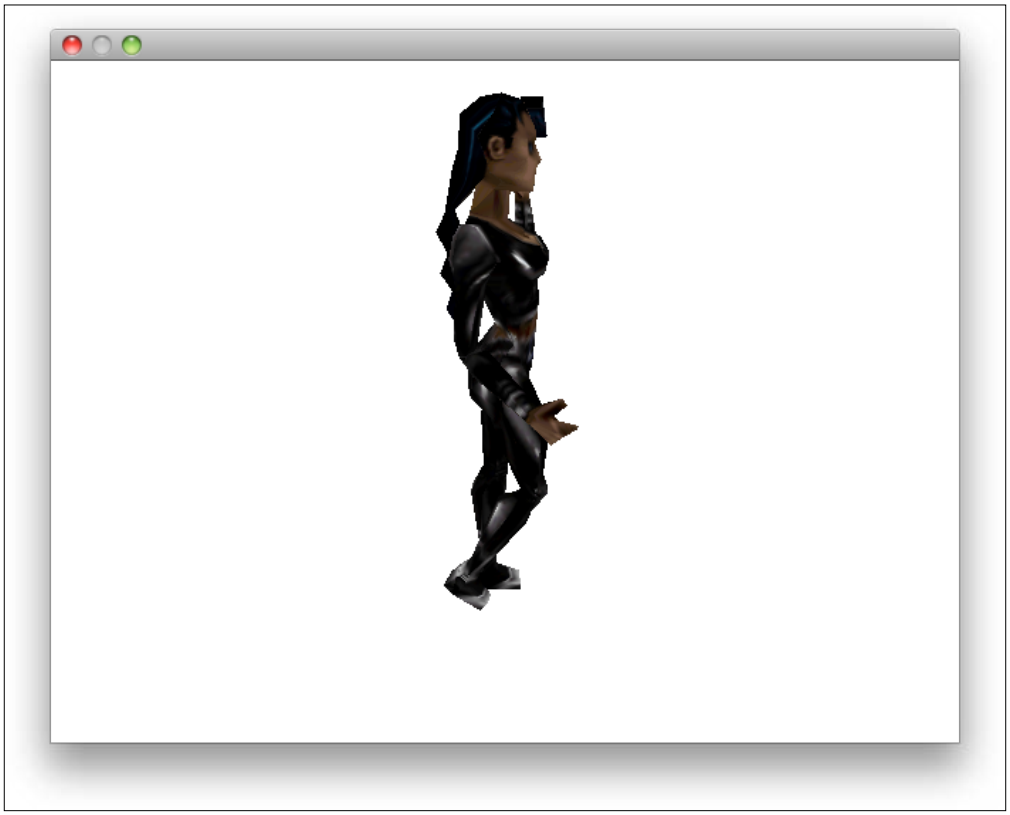
At the moment, we just have a blackish mesh on a white background. This is because MD2 and MD3 formats don't have material information on which textures to load. So let's try to change that by loading the required texture manually. If we use a mesh file format that contains material information such as `ninja.b3d` from `media` directory, it would load the texture automatically with the mesh.

Time for action – applying texture to a mesh

Our mesh looks a bit dull at the moment, now we are going to try to add a texture. Think of textures as clothes for meshes.

1. Go to the line where you added the mesh to the scene node.
2. Add the following code directly after that:

```
if (node)
{
node->setMaterialFlag(EMF_LIGHTING, false);
node->setMaterialTexture(0, driver->getTexture("sydney.bmp"));
}
```



And this is what it should look like:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace scene;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    IMesh* mesh = smgr->getMesh("sydney.md2");
    IMeshSceneNode* node = smgr->addMeshSceneNode(mesh);

    if (node)
    {
       node->setMaterialFlag(EMF_LIGHTING, false);
       node->setMaterialTexture(0, driver-
                                >getTexture("sydney.bmp"));
    }

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
                             vector3df(0, 5, 0));

    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 255, 255, 255));

       smgr->drawAll();

       driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

At first, we are going to check if the mesh node object is successfully loaded and if so continue with the actual code.

The first line sets a material flag. Because we don't have any lights set in the scene, we are going to disable lighting on materials. If this flag would still be set to `true`, the whole model would still be displayed in black.

In the second line, we assign a texture as a material for this node. The first parameter stands for the layer of the texture and as we don't have any layered textures, we just set it to `zero`. The second parameter requires a texture interface pointer.

Time for action – manipulating our mesh

Now that we have a textured mesh object on the screen, let's try to rotate, scale, and update the position of our mesh:

1. Go to the line where we check if our node has been assigned.
2. Add `node->setRotation(vector3df(0.0f, -70.0f, 0.0f));` after we set the material of the mesh.
3. Add `node->setPosition(vector3df(10.0f, -10.0f, 0.0f));`.
4. Add the line `node->setScale(vector3df(0.5f, 1.4f, 1.0f));`:

If you have had any problem, take a look at the full source code of this example:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace scene;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    IMesh* mesh = smgr->getMesh("sydney.md2");
    IMeshSceneNode* node = smgr->addMeshSceneNode(mesh);

    if (node)
    {
      node->setMaterialFlag(EMF_LIGHTING, false);
      node->setMaterialTexture(0, driver-
                               >getTexture("sydney.bmp"));

      node->setRotation(vector3df(0.0f, -70.0f, 0.0f));
      node->setPosition(vector3df(10.0f, -10.0f, 0.0f));
      node->setScale(vector3df(0.5f, 1.4f, 1.0f));
    }

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
                             vector3df(0, 5, 0));

    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 255, 255, 255));

       smgr->drawAll();

       driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

Now we have mesh that rotated -70 degrees around the Y-axis, hideously scaled, and moved 10 units on the X-axis and -10 units on the Y-axis.

The most important thing here is that we don't actually manipulate the mesh itself, but the scene node with the mesh.

Each of these functions we have been introduced to require a 3D vector as a parameter. Create one using the `vector3df()` function, whose parameters are the X, Y, and Z values.

Time for action – animating our mesh

Let's take our code that we used before we accessed and manipulated our mesh. Fortunately, the exemplary `sydney.md2` file already contains some animations for us to play with. We are going to use an animated mesh instead of our static mesh and add an animation:

1. Change all references from `IMesh` to `IAnimatedMesh`.
2. Change all references from `IMeshSceneNode` to `IAnimatedMeshSceneNode`.
3. Change all references from `addMeshSceneNode` to `addAnimatedMeshSceneNode`.
4. Add `node->setMD2Animation(scene::EMAT_ATTACK);` after the place where we check if the node has been assigned.
5. If you think the animation is running too fast, you can adjust it by using `node->setAnimationSpeed(25);` method and passing the speed you want to use.

6. Compile and run the application to enjoy our first animated mesh:

The complete source code should look something like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace scene;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);
```

```
    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    IAnimatedMesh* mesh = smgr->getMesh("sydney.md2");
    IAnimatedMeshSceneNode* node = smgr-
                                   >addAnimatedMeshSceneNode(mesh);

    if (node)
    {
       node->setMD2Animation(EMAT_ATTACK);
       node->setAnimationSpeed(25);
       node->setMaterialFlag(EMF_LIGHTING, false);
       node->setMaterialTexture(0, driver-
                                >getTexture("sydney.bmp"));
    }

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
                             vector3df(0, 5, 0));

    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 255, 255, 255));

       smgr->drawAll();

       driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

An animated mesh is a derived object from `IMesh`. To be able to use animations, we have to use an animated scene node.

We need to set the animation we want to play using `setMD2Animation`. The Quake 2 mesh file is able to store up to nearly a dozen different animations from standing and running to attacking and dying. If you use a different mesh format than MD2, you need to call the `node->setFrameLoop(0, 15);` method specifying the position of frames for the particular animation of that mesh file.

Have a go hero – switching animations

In the above example, we play the attack animation with an infinite loop. Now try to play the stand animation. Disable the infinite loop and switch to run animation once the stand animation has finished playing.

Hint: you'll need to use `setLoopMode()` and `setAnimationEndCallback()` methods of the animated mesh scene node. Plus, you'll also need to implement your own `AnimationEndCallback` class extending from `IAnimationEndCallback`.

Pop quiz

1. You want to use a mesh file format that Irrlicht doesn't support currently. To import this format into the engine, you can implement your own reader class deriving from:
    a. `IMeshLoader`
    b. `IAnimatedMesh`
    c. `ISceneManager`
2. You have an animated mesh loaded into the scene. Which of the following is responsible for manipulating that animated mesh's attributes such as position, rotation, scale, and so on?
    a. `IMesh`
    b. `IMeshSceneNode`
    c. `IAnimatedMesh`

Summary

In this chapter, we explored different types of mesh and how to work with them in Irrlicht.

In particular, we've covered:

- Loading and manipulating a mesh in Irrlicht
- Dressing up meshes with textures
- File formats
- Animating a mesh

Now that we've learned how to load meshes, we should get deeper into 3D graphics programming. To do so, we need to understand more about vectors, points, and other data types available in Irrlicht. We'll look into that in the next chapter.

4
Overlays and User Interface

We already know how to create an Irrlicht application, how to load and display a mesh on the screen. Before we move on to more advanced 3D mechanics, let's take a look at Irrlicht's 2D capabilities.

In this chapter, we shall:

- Learn what the overlay and user interface are
- Loading and displaying an image
- Displaying text on the screen
- Building our own user interface
- Adding events to our user interface

So, let's get started.

What is an overlay?

You might be wondering why we would need 2D objects, because Irrlicht is primarily a 3D graphics engine.

Let's assume the case that you would want to create a 3D action game, where you would discover caverns and battle against evil monsters you might find in those caves. Now, we would need something to display how many hit points our character has left, which weapon he is currently using, some kind of inventory to store loot from the monsters we killed and so on. Using 3D objects for those tasks would be too time-consuming as this is exactly the situation where we should use overlays.

Overlays are any 2D objects, be it images, buttons, text, or user interfaces of any kind. The term overlay was coined because usually the 3D scene is rendered first and those 2D objects will be rendered afterwards and laid upon this scene.

Time for action – drawing a 2D image

Let's use the template application from Chapter 2, *Creating a Basic Template Application* as a base for this exercise. We'll load an image from Irrlicht's `media` folder and draw it:

1. After the line `IVideoDriver* driver = device->getVideoDriver();` add `ITexture* image = driver->getTexture("../../media/irrlichtlogo2.png");`.

You can use another image from the `media` folder or any other image of your choice. Just make sure the filename and path are correctly passed to the getTexture() method.

2. In the "game loop" between the `beginScene()` and `endScene()` methods add `driver->draw2DImage(image, position2d<s32>(50, 50), rect<s32>(0, 0, 128, 128), 0, SColor(255, 255, 255, 255), true);`.

   Your full source code should look like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ITexture* image = driver-
                    >getTexture("../../media/irrlichtlogo2.png");
```

```
    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 255, 255, 255));

        driver->draw2DImage(image, position2d<s32>(50, 50),
                            rect<s32>(0, 0, 128, 128), 0,
                            SColor(255, 255, 255, 255), true);

        driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

First of all, we need to create a texture much like we did when we needed a texture for our mesh in Chapter 3, *Loading Meshes*. Remember that a texture just stores the information of an image, but does not actually draw an image. Think of it as a picture stored in a digital camera that only contains data about the image itself, but to access the image it must be downloaded on another storage device. Transferred to the Irrlicht engine, we have to call the `draw2DImage()` method.

This method can be overloaded with different parameters. Let's take a look at the specific method we used here that has parameters split up as follows:

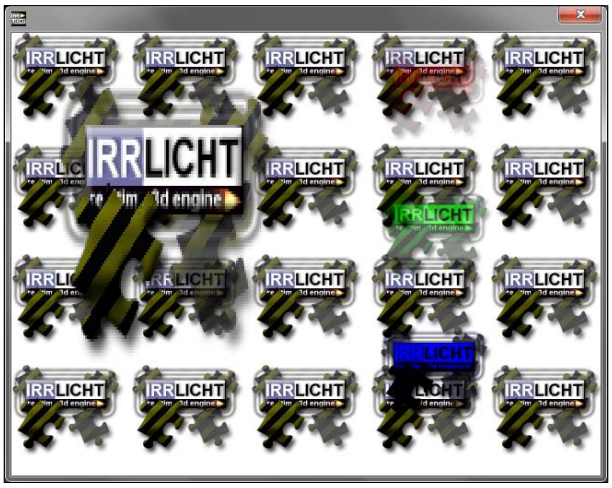
- Texture object requires a pointer of the type `ITexture`.

- **Position**: This can either be of the type `position2d<s32>` or `rect<s32>`. The coordinate system has its origin in the top-left corner and so does the image. Note that there is no Z-coordinate to order the drawn images, so the first image you draw will be in the background and every additional image will be drawn over the last image.
- **Source rect**: It needs to be a `rect<s32>` with the coordinates of how big the image is, you want to show on the screen.
- **Clip rect**: It is a pointer to a rectangle where the image should be clipped to, on the screen. This is `0` or `NULL` by default.
- **Color**: It is the color of the image and has the color white by default.
- **Use alpha channel**: If this is set to `true`, it uses the alpha channel provided by the image. The default value is `false`.

The other method has only two parameters, the first one for the texture object and the second for the position that has to be from the type `position2d<s32>`.

Have a go hero – drawing an image

Now that you know how to load textures and display images, try experimenting with the `draw2DImage()` method. Scale the image, tint the image, or draw more than one image at a time. These are just some suggestions:



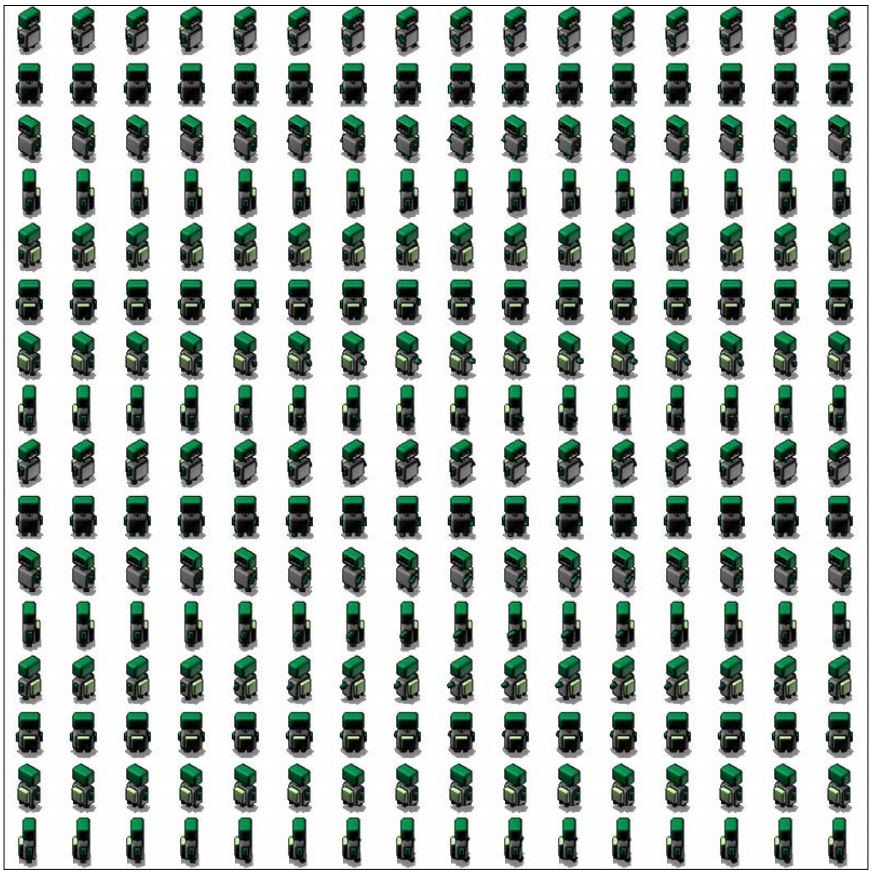
Example code:

```
#include <irrlicht.h>
using namespace irr;
```

```
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 32, false,
                             false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ITexture* image = driver-
                      >getTexture("../../media/irrlichtlogo2.png");

    while (device->run())
    {
        driver->beginScene(true, true, SColor(255, 255, 255, 255));

        for (int i = 0; i < 5; i++)
        {
            for (int j = 0; j < 5; j++)
            {
                 driver->draw2DImage(image, position2d<s32>(i * 130,
                                    j * 120),rect<s32>(0, 0, 128,
                                    128), 0, SColor(255, 255, 255,
                                                 255),true);
            }
        }

        driver->draw2DImage(image, position2d<s32>(400, 20),
                            rect<s32>(0, 0, 128, 128), 0, SColor(85,
                            255, 0, 0), true);
        driver->draw2DImage(image, position2d<s32>(400, 170),
                            rect<s32>(0, 0, 128, 128), 0, SColor(170,
                            0, 255, 0), true);
        driver->draw2DImage(image, position2d<s32>(400, 320),
                            rect<s32>(0, 0, 128, 128), 0, SColor(255,
                            0, 0, 255), true);

        driver->draw2DImage(image, rect<s32>(50, 50, 300, 450),
                            rect<s32>(0, 0, 128, 128),0 , 0, true);

        driver->endScene();
    }

    device->drop();

    return 0;
}
```

Using a sprite sheet

If you have taken a closer look at some 2D games, you may have noticed that usually animations are assembled into a sprite sheet (also called **texture atlas**). It is always better to load one single texture instead of multiple small textures, because having a lot of texture switches in your application will slow down the performance considerably:



This sprite sheet is from an open-source game called *A Practical Survival Guide for Robots* that can be downloaded from `http://www.ludumdare.com/compo/ludum-dare-17/?uid=321`. The graphics and all the media of this game are licensed under the Creative Commons 3.0 SA-BY license, which means that you can use this media for anything, even for commercial applications.

This specific robot sprite sheet is located at `/resources/images/robot.png` in the `game` folder.

Time for action – using a sprite sheet

Let's get this little robot guy animated. The best way is to use the example of our last exercise:

1. Copy the robot sprite sheet to your `application` folder.
2. Change `../../media/irrlichtlogo2.png` in `ITexture* image = driver->getTexture("../../media/irrlichtlogo2.png");` to `robot.png`.
3. After that line add `driver->makeColorKeyTexture(image, position2d<s32>(0, 0));`.
4. Add a variable called `currentFrame` having the `int` type and initialize this variable with `0`.
5. Add another `int` variable called `row` initialized with `0`.
6. Directly in the "game loop" even before the `beginScene()` method add the following piece of code:

```
u32 time = device->getTimer()->getTime();
if ((time % 25) == 0) currentFrame++;
if (currentFrame >= 31)
{
    currentFrame = 0;
    row++;
}
if (row >= 7) row = 0;
```

7. Change the `draw2DImage()` method to this: `driver->draw2DImage(image, position2d<s32>(200, 200), rect<s32>(currentFrame * 64, row * 64, (currentFrame + 1) * 64, (row + 1) * 64), 0, SColor(255, 255, 255, 255), true);`.
8. Compile the example and our little robot guy is animated.

   Your complete source code of this example should look like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
```

```
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ITexture* image = driver->getTexture("robot.png");
    driver->makeColorKeyTexture(image, position2d<s32>(0, 0));

    int currentFrame = 0;
    int row = 0;

    while (device->run())
    {
        u32 time = device->getTimer()->getTime();
        if ((time % 25) == 0) currentFrame++;
        if (currentFrame >= 31)
        {
            currentFrame = 0;
            row++;
        }
        if (row >= 7) row = 0;

        driver->beginScene(true, true, SColor(255, 200, 200,
                                              200));

        driver->draw2DImage(image, position2d<s32>(200, 200),
                            rect<s32>(currentFrame * 64, row * 64,
                            (currentFrame + 1) * 64, (row + 1) *
                            64), 0, SColor(255, 255, 255, 255),
                            true);

        driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

At first we need to copy the sprite sheet so our application can find it and change the filename to represent that we want to load the robot texture.

In the last example, we had an image with an alpha channel that is not the case now. As you may have already noticed the sprite sheet has a white background. There are two ways to add an alpha channel from within the Irrlicht 3D engine with the `makeColorKeyTexture()` method. The first parameter takes the texture object, the second parameter can either be the color itself of the `SColor` type or a point of the `position2d` type, which tells where the color can be found that will turn transparent. Be advised that this function just adds an alpha channel to this image. You need to call `draw2DImage()` with the `Use alpha channel` parameter set to `true` or the specified color areas won't show up as transparent.

We will need to set up two variables to represent which frame is currently being used. As you can see the sprite sheet is not just a simple strip of sprites. On the X-axis, there are different frames of a walk animation. On the Y-axis, there are different directions of the robot. Each frame has a width and height of 64 pixels.

Irrlicht has a built-in timer that should be used every time we are doing a time-sensitive operation. For example, if you need a timer shown in your screen that counts down each second, you can call this timer with `device->getTimer()->getTime();`. The quickest way to get an action would be to use a mathematical mod operation on the result of the Irrlicht timer method. If we would use `if (((device->getTimer()->getTime()) % 1000) == 0)`, the condition will be called every second. Back to our code: each 25 milliseconds we count the current frame up, if we reach the maximum frame—which is 32, because the image is 2048 pixels wide and each frame is 64 pixels big as mentioned previously—we switch to the next row of the sprite sheet. Think of this animation as a flip-book that changes the current frame so rapidly that it would create the illusion of an animation.

In step 7, we need to modify our `draw2DImage()` method to reflect what we did in the step before. We need to adjust the source `rect` parameter. You have to remember that the third and fourth parameters of the source `rect` are not actually the width and height respectively, but a position. The difference between the first and second position will then be drawn on the screen.

Making sprite sheets

Alright, we know how to use sprite sheets, but how do we make a sprite sheet for ourselves? There are several applications that can assemble different sprites into a big texture atlas.

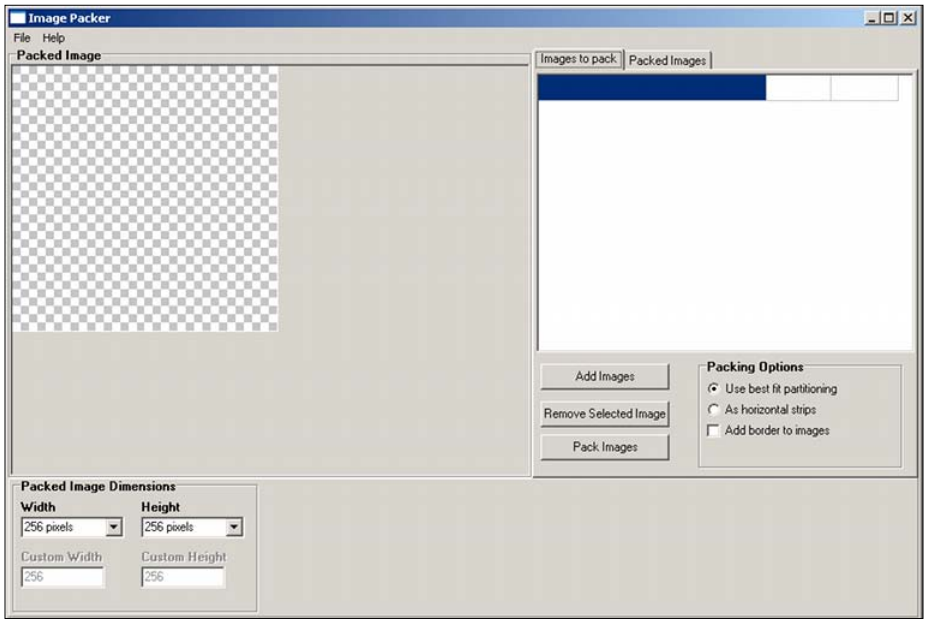
We will be using Paul Nicholls' open-source **Image Packer** utility for creating a texture atlas. This program can be downloaded from `http://files.freeze-dev.com/ImagePacker.zip`.

Unfortunately, Image Packer is, at the moment, only available for Windows.

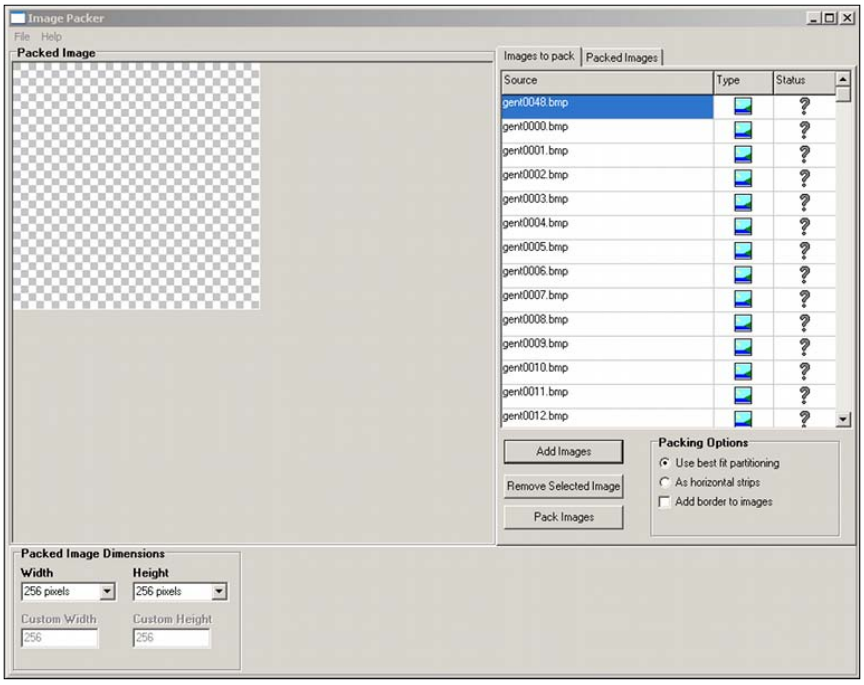
Time for action – making sprite sheets

As you have downloaded and extracted the file, let's get started:

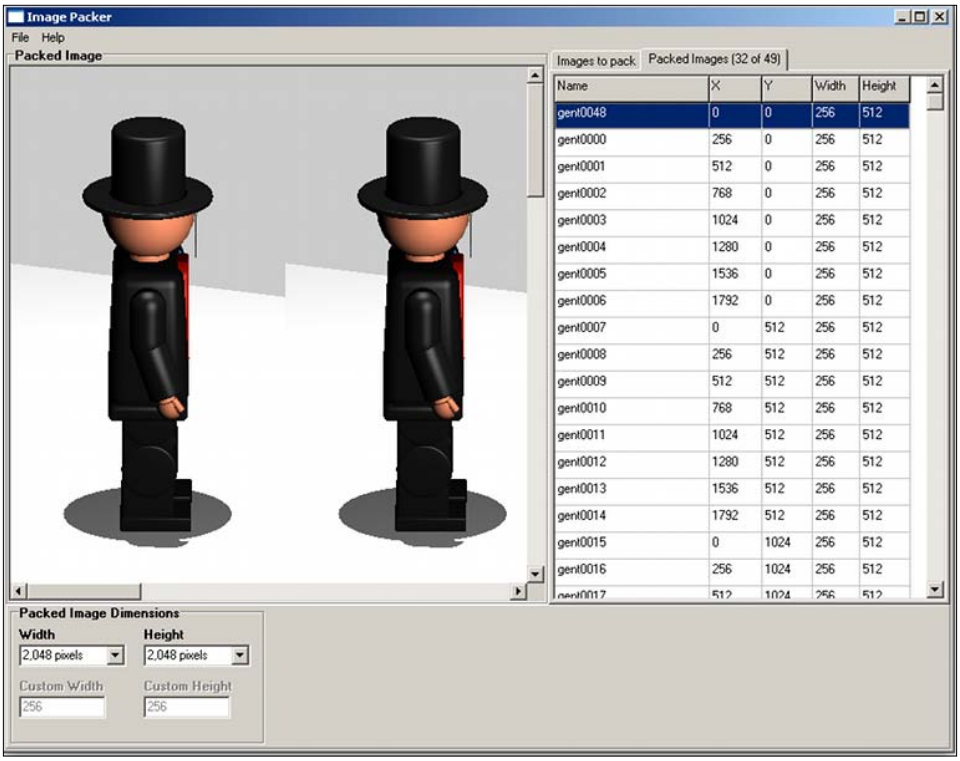
1. Open `imagepacker.exe` from the extracted package:



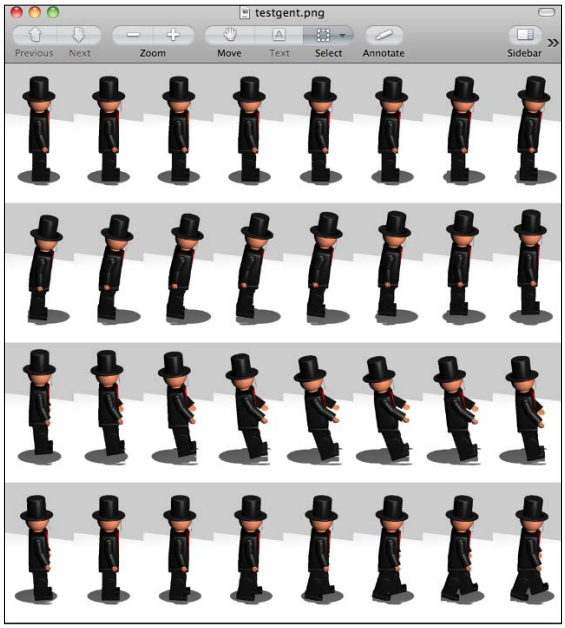
2. Click on **Add Images** to add images. The added images will be shown up in the right sidebar:



3. Adjust the size of the texture atlas under **Packed Image Dimensions** and click on **Pack Images** to create your sprite sheet. Make sure **Horizontal Strips** is checked under **Packing Options**:



4. Right-click on the image and save it in the file format of your choice:



What just happened?

Now, we know how to create our own sprite sheets. We just used it for animating a character, but of course you can assemble sprite sheets from any sprites even with each one having a different width and height. If you don't have an animation strip, but a couple of unrelated sprites instead, using the packing option **Use best fit partitioning** is the best solution. If you use this option on an animation strip, you will not get the expected result and the animation will appear somewhat chaotic.

When importing images through **Add Images**, you have the option to split them into imported images that allows you to import previously-created sprite sheets and assemble them into a new texture atlas.

If you want to delete images from the texture atlas, select the image or images and click on **Remove Selected Images**.

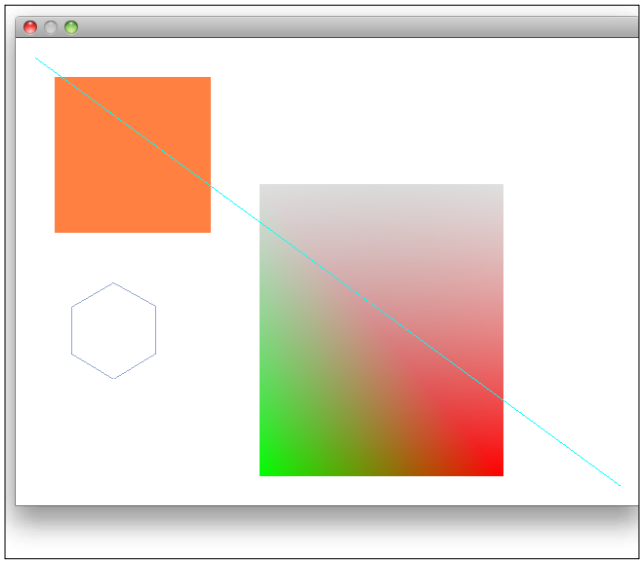
Drawing primitives

Drawing primitives like rectangles, octagons, and similar geometrical shapes can also be done in Irrlicht. In almost any application, you need some kind of geometrical shapes—if you want to write a function plotter, a Geometry Wars styled game, or just a smoother user interface experience.

Time for action – drawing primitives

We'll use the template application from Chapter 2, *Creating a Basic Template Application* for this exercise.

1. Go to the "game loop".
2. Between the `beginScene()` and `endScene()` methods type `driver->draw2DRectangle(SColor(255, 255, 128, 64), rect<s32>(40, 40, 200, 200));`.
3. Draw another rectangle with `driver->draw2DRectangle(rect<s32>(250, 150, 500, 450), SColor(64, 128, 128, 128), SColor(128, 192, 192, 192), SColor(255, 0, 255, 0), SColor(255, 255, 0, 0));`.
4. Let's draw a hexagon using `driver->draw2DPolygon(position2d<s32>(100, 300), 50.0f, SColor(128, 40, 80, 160), 6);`.
5. Draw a line all across the screen with `driver->draw2DLine(position2d<s32>(20, 20), position2d<s32>(620, 460), SColor(255, 0, 255, 255));`:



What just happened?

Irrlicht is able to draw the following types of primitives:

Rectangles

Rectangles can be drawn in three different ways. You can either draw an outlined or a filled rectangle. A filled rectangle can be drawn in two different ways: either you draw a rectangle in a single color or you specify each color of each vertex of your rectangle. Remember that when you use the `rect` data type the last two values are not width and height but vertices.

Much like the `draw2DImage()` method, you can also specify a clipping `rect` that is an optional parameter and set to `0` by default.

Polygons

You can draw polygons with the `draw2DPolygon()` method. The first parameter is the center of the shape, the second is the radius, the third is the color of the shape, and the last parameter is the number of edges. If this vertex count is three, a triangle will be drawn, if the vertex count is six, a hexagon will be drawn.

This method can also be used for drawing circles as long as the vertex count is set high enough.

Lines

A line can be drawn by defining the start, then the end point that both have to be of the `position2d` type, and the last parameter is the line color.

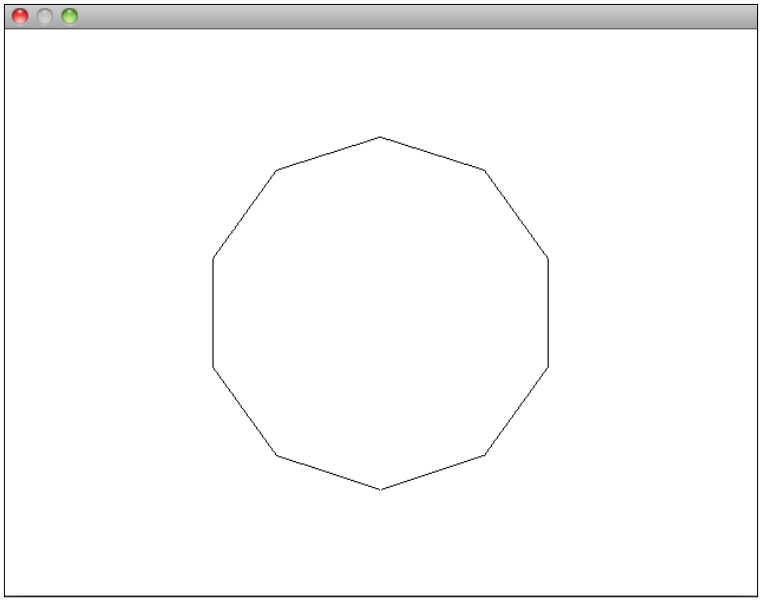
There is also the possibility to draw primitives using the `draw2DVertexPrimitiveList()` method, which is more optimized if you need to draw hundreds of primitives in each frame.

After this exercise your "game loop" should look like this:

```
driver->beginScene(true, true, SColor(255,200,200,200));

    driver->draw2DRectangle(SColor(255, 255, 128, 64), rect<s32>(40,
                            40, 200, 200));
    driver->draw2DRectangle(rect<s32>(250, 150, 500, 450), SColor(64,
                            128, 128, 128), SColor(128, 192, 192,
                            192), SColor(255, 0, 255, 0), SColor(255,
                            255, 0, 0));
    driver->draw2DPolygon(position2d<s32>(100, 300), 50.0f,
                          SColor(128, 40, 80, 160), 6);
    driver->draw2DLine(position2d<s32>(20, 20), position2d<s32>(620,
                       460), SColor(255, 0, 255, 255));

driver->endScene();
```

Have a go hero – drawing primitives

Now that you know how to draw primitives, try to create an application in which a polygon is being drawn and it changes itself in any interval (for example, one second) from a triangle to a rectangle, to a pentagon, and so on. The shape should also scale up in that interval:



Graphical user interface

In this part of this chapter, we will take a closer look at the user interface and how we can design a graphical user interface with the Irrlicht 3D engine.

A graphical user interface (or GUI) allows the user to interact with an application by providing graphical elements for different tasks. For example, a button that pops up in another window, an edit box to enter some text, or radio buttons to give the user the possibility to choose between different options.

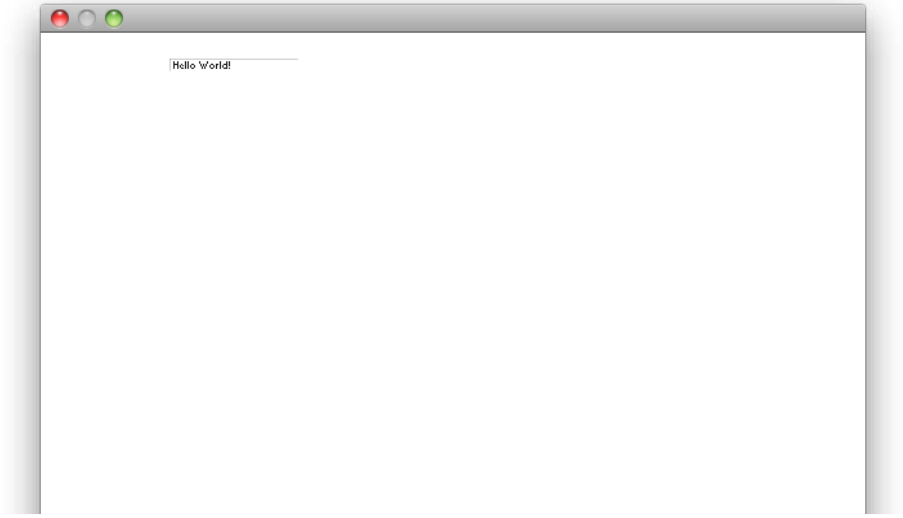
Displaying text on the screen

We know how to draw images at primitives on the screen. Let's write something on the screen.

Time for action – displaying text on the screen

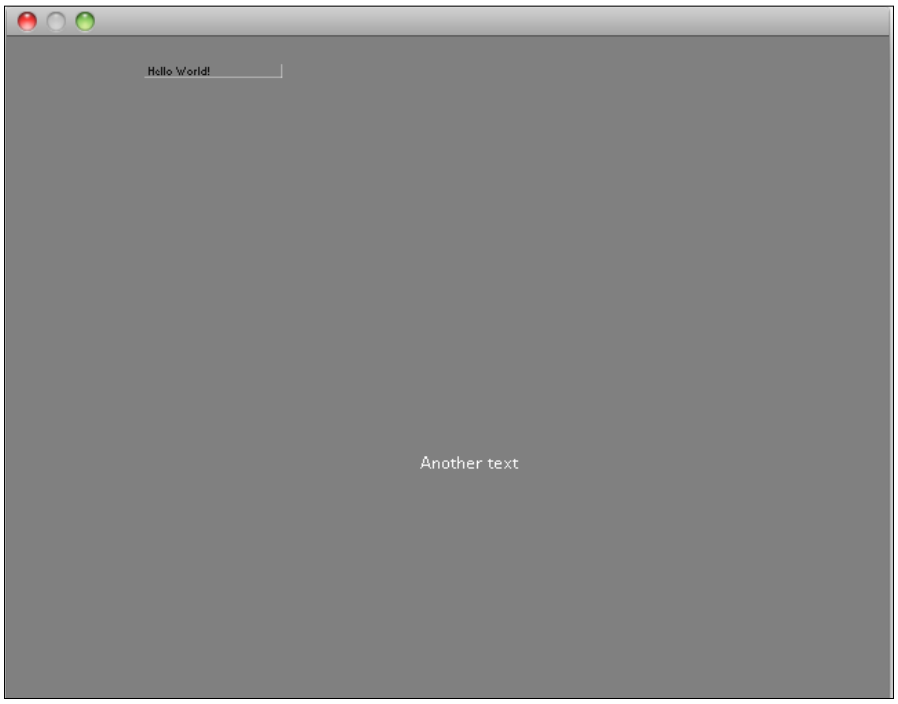
We'll use the template application from Chapter 2, *Creating a Basic Template Application* once again for this exercise.

1. Add `using namespace gui;` directly together with all the other "using namespaces".
2. After the line `IVideoDriver* driver = device->getVideoDriver();`, add `IGUIEnvironment* guienv = device->getGUIEnvironment();`.
3. Add `guienv->addStaticText(L"Hello World!", rect<s32>(100, 20, 200, 30), true);` immediately after that.
4. In the "game loop", add `guienv->drawAll();` so that the GUI can be drawn:



5. Copy the files `lucida.xml` and `lucida0.png` from Irrlicht's `media` folder to your application's directory.
6. Under the the line `IGUIEnvironment* guienv = device->getGUIEnvironment();`, add `IGUIFont* font guienv->getFont("lucida.xml");`.
7. In your "game loop", call `font->draw(L"Another text", rect<s32>(300, 300, 300, 50), SColor(255, 255, 255, 255));`.

8. Adjust the background color to a slightly darker color:



Here is the full source code of this example:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace gui;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
```

```
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    IGUIEnvironment* guienv = device->getGUIEnvironment();
    IGUIFont* font = guienv->getFont("lucida.xml");

    guienv->addStaticText(L"Hello World!", rect<s32>(100, 20, 200,
                          30), true);

    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 128, 128, 128));

       guienv->drawAll();

       font->draw(L"Another text", rect<s32>(300, 300, 300, 50),
                  SColor(255, 255, 255, 255));

       driver->endScene();
    }

    device->drop();

    return 0;
}
```

What just happened?

There are basically two ways to get text displayed on the screen. The first option is to add a static text through the GUI environment class. The GUI environment class is a factory pattern, which means that it should not create GUI-relevant classes like fonts or buttons directly, but use the GUI environment class for those tasks.

If you add text with `addStaticText()` the Irrlicht built-in font is used and can only be changed if you create a custom skin for your GUI. The first parameter of this method is the text as a char pointer. Remember that this method expects a wide char pointer, so don't forget to add `L` before your string. The second parameter is a `rect` in which the text will be drawn. The third parameter is a Boolean value, which determines if a border will be drawn around the specified `rect`.

There are some additional optional parameters that are not important for us at the moment.

The second option is to create a font object using the GUI environment class and specifying a font that we want to be loaded into our application. After that we draw a text on the screen using the `draw()` method of the font object. The parameters are quite similar to the `addStaticText()` method, the first is a wide char pointer, the second is the `rect` in which the text will be drawn, and the third is the color of the text. The forth and fifth parameters are to align the text horizontally and vertically respectively. The sixth parameter is for a clipping `rect`. The last three parameters are optional. This is what we did in step 4 to step 6.

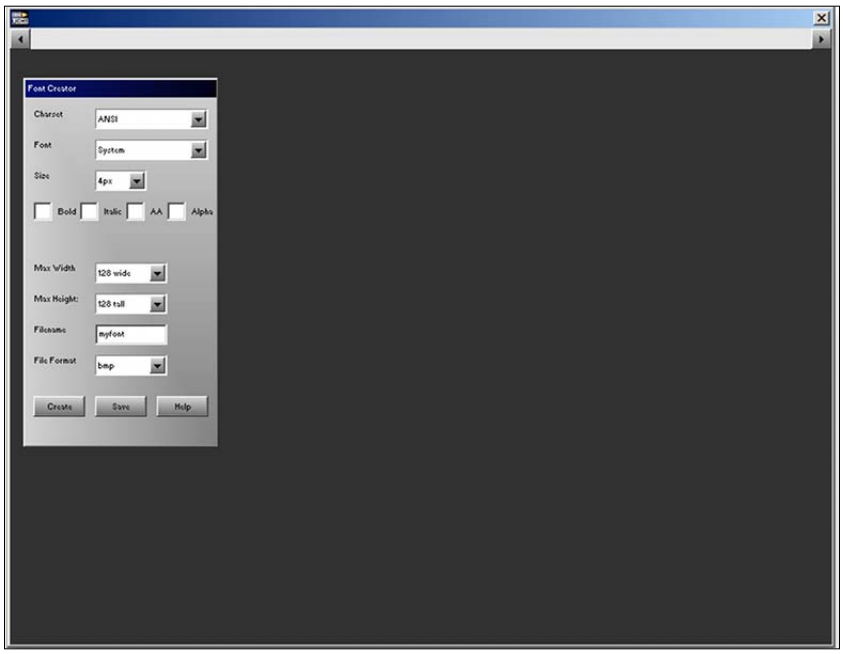
Using the Irrlicht font tool

Irrlicht offers a small tool for creating your own fonts, which you will now learn how to use.

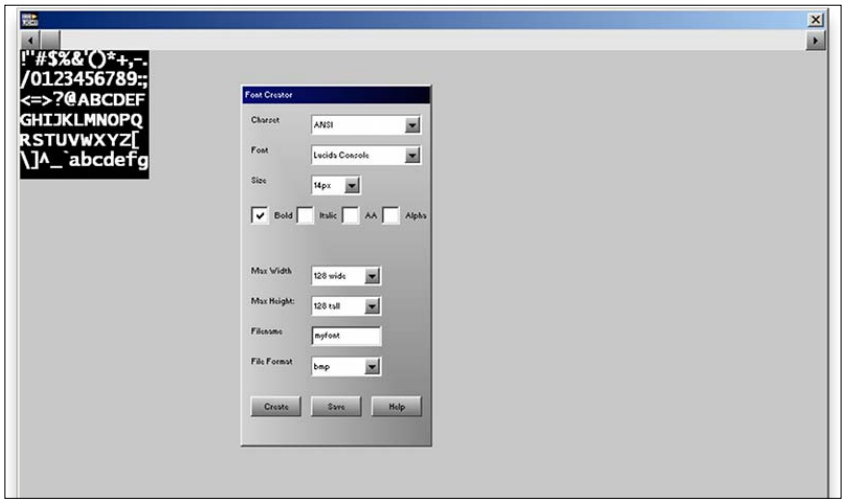
Time for action – using the Irrlicht font tool

The font tool can be found in `/bin/Win32-VisualStudio` if you are on a Windows machine; if you are using Linux, or Mac OS X, you need to compile the font tool for yourself.

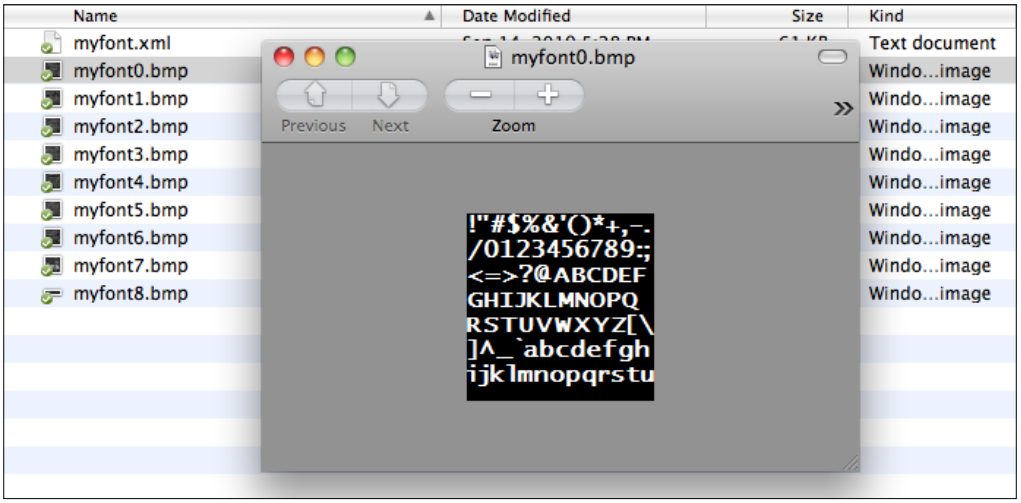
1. Open `FontTool.exe` from the extracted package or your self-compiled application if you are using Linux or Mac OS X:



2. Set the the font dialog box to your preferred settings and click on **Create** to continue:



3. You can change the default filename under which the font will be saved. If you click on **Save**, the files will be written in the path of the Irrlicht font tool unless you did change that by changing the default font file name:



What just happened?

If the specified width and height won't fit all the characters, your font will be split up into multiple files. Because of texture switches, it is recommended to set a higher width and height.

Adding buttons to our GUI

We are going to add two buttons on the screen and add events to those buttons: one button will close the application, while the other one will spawn new windows on the screen.

Time for action – adding buttons to your GUI

We are going to use the example from the last exercise.

1. Add a struct called `SAppContext` that contains `IrrlichtDevice* device`.
2. Add an enum with two states. One called `GUI_ID_QUIT_BUTTON`, which should be initialized with the value `101` and the other should be called `GUI_ID_NEW_WINDOW_BUTTON`.
3. We need our own event receiver class that inherits from `IEventReceiver`, which is going to handle the events if we press one of our buttons:

```
class MyEventReceiver: public IEventReceiver
{
public:
MyEventReceiver(SAppContext & context) : Context(context) { }

    virtual bool OnEvent(const SEvent& event)
    {
       if (event.EventType == EET_GUI_EVENT)
       {
          s32 id = event.GUIEvent.Caller->getID();
          IGUIEnvironment* guienv = Context.device-
                                    >getGUIEnvironment();

          if (event.GUIEvent.EventType == EGET_BUTTON_CLICKED)
          {
                 if (id == GUI_ID_QUIT_BUTTON)
                 {
                    Context.device->closeDevice();
                    return true;
                 }
```

```
                    if (id == GUI_ID_NEW_WINDOW_BUTTON)
                    {
                        IGUIWindow* window = guienv-
                        >addWindow(rect<s32>(100, 100, 300, 200),
                        false, L"New window");
                        return true;
                    }

                }
            }

            return false;
        }

    private:
        SAppContext & Context;
    };
```

4. Add `guienv->addButton(rect<s32>(250, 20, 250 + 120, 50), 0, GUI_ID_QUIT_BUTTON, L"Exit", L"Exits Program");` and `guienv->addButton(rect<s32>(400, 20, 400 + 120, 50), 0, GUI_ID_NEW_WINDOW_BUTTON, L"New Window", L"Launches a new Window");` directly after the line `guienv->addStaticText(L"Hello World!", rect<s32>(100, 20, 200, 30), true);`.

5. Create an instance of the `SAppContext` struct called `context` and set `context.device` to `device`.

6. Then we need to create an instance of `MyEventReceiver` and call its constructor with `context`.

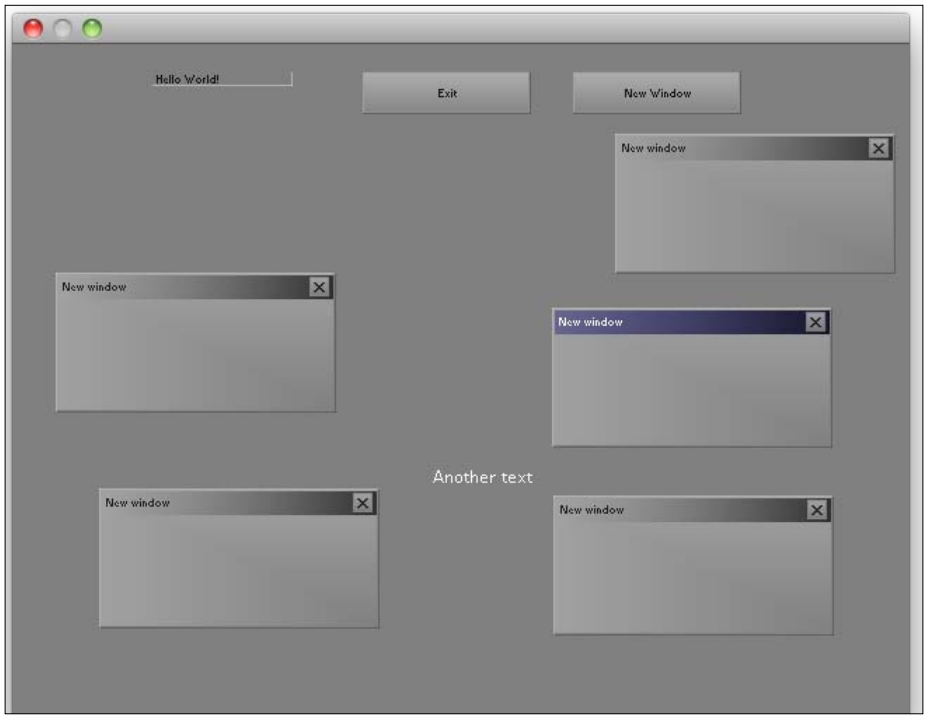
7. We are now going to connect the event receiver to our device by calling `device->setEventReceiver(&receiver);`.

Your full source code of this example should look like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;
using namespace gui;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif
```

```
struct SAppContext
{
    IrrlichtDevice *device;
};

// Define some values that we'll use to identify individual GUI
controls.
enum
{
    GUI_ID_QUIT_BUTTON = 101,
    GUI_ID_NEW_WINDOW_BUTTON
};

class MyEventReceiver: public IEventReceiver
{
public:
    MyEventReceiver(SAppContext & context) : Context(context) { }
    virtual bool OnEvent(const SEvent& event)
    {
       if (event.EventType == EET_GUI_EVENT)
       {
          s32 id = event.GUIEvent.Caller->getID();
          IGUIEnvironment* guienv = Context.device-
                                    >getGUIEnvironment();
          if (event.GUIEvent.EventType == EGET_BUTTON_CLICKED)
          {
                 if (id == GUI_ID_QUIT_BUTTON)
                 {
                    Context.device->closeDevice();
                    return true;
                 }
                 if (id == GUI_ID_NEW_WINDOW_BUTTON)
                 {
                     IGUIWindow* window = guienv-
                     >addWindow(rect<s32>(100, 100, 300, 200),
                     false, L"New window");
                     return true;
                 }
          }
       }
       return false;
    }
private:
```

```
    SAppContext & Context;
};

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16,
                             false, false, false, 0);

    if (!device)
       return 1;

    IVideoDriver* driver = device->getVideoDriver();
    IGUIEnvironment* guienv = device->getGUIEnvironment();
    IGUIFont* font = guienv->getFont("lucida.xml");

    guienv->addStaticText(L"Hello World!",
                          rect<s32>(100, 20, 200, 30), true);
    guienv->addButton(rect<s32>(250, 20, 250 + 120, 50), 0,
                      GUI_ID_QUIT_BUTTON, L"Exit", L"Exits
                      Program");
    guienv->addButton(rect<s32>(400, 20, 400 + 120, 50), 0,
                      GUI_ID_NEW_WINDOW_BUTTON, L"New Window",
                      L"Launches a new Window");

    SAppContext context;
    context.device = device;

    MyEventReceiver receiver(context);

    device->setEventReceiver(&receiver);

    while (device->run())
    {
       driver->beginScene(true, true, SColor(255, 128, 128, 128));

       guienv->drawAll();

       font->draw(L"Another text", rect<s32>(300, 300, 300, 50),
                  SColor(255, 255, 255, 255));

       driver->endScene();
    }

    device->drop();

    return 0;
}
```



What just happened?

Every button, or GUI object for that matter, can be identified by an ID, which we need to set. We need a struct, because we cannot use the device directly and need to set up a bridge between our application device and our own event receiver class.

Let's take a closer look at our own event receiver class— specifically at what happens in the `OnEvent()` method. First we check if there has been a GUI event. If there has, we check if any of our two buttons have been pressed and close the application if **Exit** has been clicked or launch a new window if **New Window** has been clicked.

To add our buttons to the screen, we need the method `addButton()` that has the following parameters: the first is the size of the button, the second is a GUI element parent, then the ID, and finally the caption of this button and a tooltip that will show up if the mouse is hovering over the button.

In steps 5 to 7, we are simply connecting `MyEventReceiver` to our application or the GUI environment of our application, to be precise.

Pop quiz

1. Which of the following resolutions could be most suitable to form a sprite sheet containing animations separated in three rows and eight columns?
    a. 640 x 360
    b. 640 x 320
    c. 650 x 360
2. When the user clicks on a button, what method is being triggered?
    a. `OnEvent()` method of GUI environment class
    b. `OnEvent()` method of the event receiver class
    c. `OnClick()` method of the button

Summary

We learned about what you can do with Irrlicht if you are looking for adding some 2D objects into your application.

Specifically, we covered:

- Drawing images
- Showing text on the screen
- Adding buttons to our user interface
- Adding events to buttons

We also touched upon how to create sprite sheets and Irrlicht fonts.

Now that we've learned how to handle the basics of 3D and 2D graphics programming, we should get deeper into 3D graphics programming. But in order to do so, we need to understand more about vectors, points, rectangles, and arrays. We are going to learn about them in the next chapter.

5
Understanding Data Types

We already know how to create an application with both Irrlicht's 2D and 3D capabilities or just one of those two possibilities. But before we create more advanced examples we need to take a closer look at Irrlicht's data types.

In this chapter we shall:

- Freshen up our knowledge about C++ templates
- Explain type definitions
- Learn how to use Irrlicht arrays
- Learn about vectors

So let's get on with it.

Using C++ templates

You probably already noticed the angle brackets when we declared the positions of vectors and rectangles in our examples in previous chapters. You need to do that because these rectangle and vector classes are implemented as template classes.

What are templates?

Templates (also called parameterized types) are a more advanced topic in C++ programming language. Templates are used if you need a function or class to handle multiple data types, instead of having to write a function or class for each different data type. Templates were quite unique and have been introduced into other programming language just in the last few years. Although the concept of generics in Java, C#, or FreePascal are very similar to templates they technically work differently. Generic type information is only available at compile time, while template type information is available at run time.

Using templates

Let's try to implement a class template example to understand more about how templates should be used.

Time for action – using templates

Our template class will have two methods to compare two values with each other and return the maximum and the minimum of those two values.

1. Start a new class with `template <class T> class Diff`.
2. Add the following public method:

```
T minVal(T a, T b)
{
  return (a < b) ? a : b
}
```

3. Add another public method:

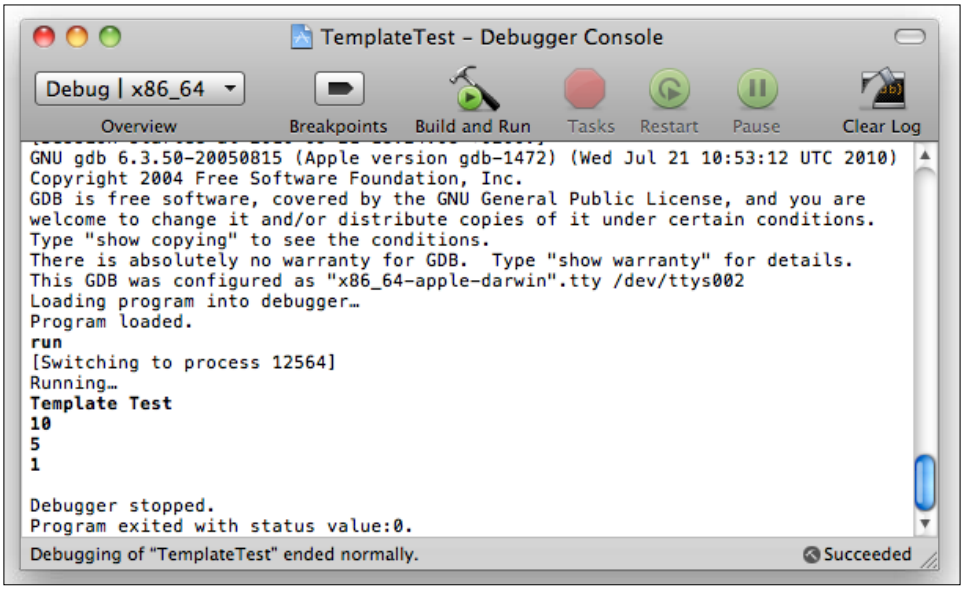
```
T maxVal(T a, T b)
{
  return (a > b) ? a : b
}
```

4. Add a third public method:

```
bool equals(T a, T b)
{
  return (a == b);
}
```

5. In the main method add some classes and test the implemented functions. For example like the following:

```
Diff<int> diffInt;
Diff<float> diffFloat;
Diff<double> diffDouble;

std::cout << diffInt.minVal(10, 20) << std::endl;
std::cout << diffFloat.maxVal(3.0f, 5.0f) << std::endl;
std::cout << diffDouble.equals(6.7f, 6.7f) << std::endl;
```



What just happened?

To specify a template class we need the keyword `template`, and in angle brackets the class and the generic type. It has been common practice to name the generic type just plain and simple `T`, but you can call it anything you want.

You may have noticed our example template class doesn't have a constructor which is unnecessary in our case. If you want to make a more complete template class, or inherit from an already existing template class, a constructor would be advisable.

It would also be possible to use C macros or void pointers to achieve what we did in this example but then we won't be able to use the object oriented design and methodologies and you won't also be able to get real types and debug info.

The example code would look like this:

```
#include <iostream>

template <class T> class Diff
{
  public:
    T minVal(T a, T b)
    {
      return (a < b) ? a : b;
    }

    T maxVal(T a, T b)
    {
      return (a > b) ? a : b;
    }

    bool equals(T a, T b)
    {
      return (a == b) ? true : false;
    }
};

int main()
{
  Diff<int> diffInt;
  Diff<float> diffFloat;
  Diff<double> diffDouble;

  std::cout << "Template Test" << std::endl;

  std::cout << diffInt.minVal(10, 20) << std::endl;
  std::cout << diffFloat.maxVal(3.0f, 5.0f) << std::endl;
  std::cout << diffDouble.equals(6.7f, 6.7f) << std::endl;

  return 0;
}
```

Type definitions

Because of Irrlicht's cross-platform design, basic types have been redefined. All of these types can be accessed when using the `irr` namespace.

Data type	Description
u8	This 8-bit type represents an unsigned char which has a range of values between 0 and 255.
s8	This 8-bit large signed char has a range between -128 to 127.
c8	This is the same type as any regular char. It has been abbreviated to fit the Irrlicht's naming convention. Its size is 8 bit.
u16	As the name implies this unsigned type is 16-bit large and with a range from 0 to 65535.
s16	The signed 16-bit version can hold any values from -32768 to 32767.
u32	The unsigned 32-bit type has a range from 0 to 4294967295.
s32	The signed 32-bit type can contain any values between -2147483648 and +2147483648. This is also the default int type on 32-bit machines.
f32	A 32-bit floating point type and basically a redefined float.
f64	A 64-bit floating point type which is more precise than the 32-bit type. On most platforms it maps to the double type of C++.

Class types

Additionally to the type definitions there are some template classes which need further explanation. All of these classes support operator overloading, so classes of the same type can, for example, be compared with each other or merged together.

video::SColor

A class which stores a color in an ARGB format, which is an alpha, red, green, and blue component in that order. Each component has a range from 0 to 255. There is an alternative constructor and a setter which takes a hex color in the format 0xAARRGGBB.

Furthermore, the `video::SColor` class provides methods to interpolate the current color which returns a new `video::Color` class and get the average, lightness, and luminance of the current color.

video::SColorf

This class is pretty similar to `video::SColor` but instead of taking unsigned char values it handles float values from 0.0 to 1.0.

There is a method to convert a `video::SColorf` class into a `video::SColor` class.

core::rect

This is a template class which requires one of the Irrlicht types in its angle brackets. Remember that the parameters are actually the edges of the rectangle and the last two parameters are actually not width and height. If you have had experience with other graphics engines, especially 2D graphics engines, this may be an adjustment.

Besides having methods to return the width, height, or area of a rectangle, the `rect` class can also check for a bounding box collision with another rectangle or check if a certain point is within the rectangle. The latter is useful when trying to do simple collision detection between two objects or to implement a mouse over effect for buttons, and so on.

core::dimension2d

This template class creates a 2-dimensional size.

Graphics cards are optimized to load images that are power of two in size which means image sizes of 256 pixels, 512 pixels, 2048 pixels, and so on. The image's size does not need to be a square; a width of 2048 pixels and a height of 512 pixels would also suffice. If a texture does not fit that criteria it will either be stretched or the texture size will be internally updated to the next-higher power of two size.

With the `getOptimalSize` method the `core::dimension2d` class returns the optimal power of two size.

core::array

The Irrlicht array is similar to the vector from the C++ STL but has additional features like the ability to perform a binary or linear search through the whole array.

Here is an example on how to use an Irrlicht array:

```
// Create an array
array< rect<s32> > myArray = array< rect<s32> >();

// Add some items
myArray.insert(rect<s32>(0, 0, 64, 64));
myArray.insert(rect<s32>(64, 64, 128, 128));
myArray.insert(rect<s32>(128, 128, 256, 256));

// Erase element at position 1
myArray.erase(1);

// Get width from the rectangle of array's last element
s32 width = myArray.getLast().getWidth();

// Clear the whole array
myArray.clear();
```

core::list

This class template provides us with a double-linked list which resembles the list template from the C++ STL.

Vectors

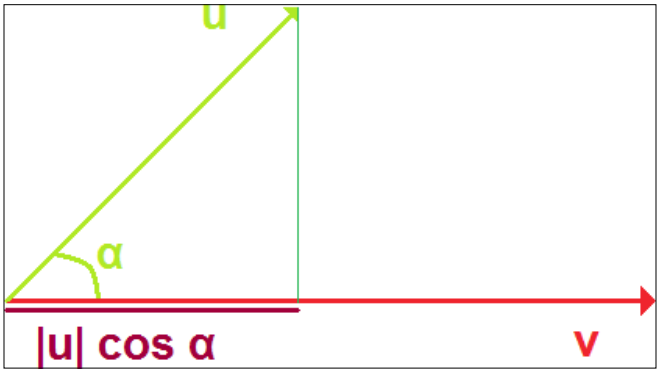
According to the mathematical definition a vector is an element defined by both magnitude and direction. This is also the case in a graphics engine, but a vector can also define a path. Irrlicht has implementations for both 2D and 3D vectors. Here's how to create a 2D vector called "position" with f32 values: vector2d<f32> position.

There are two operations that will become important later on:

Dot product

Using the dot product on just one vector will result in the length of this vector $u \cdot u = |u|$.

If the dot product is performed on two different vectors this will result in the length of both vectors multiplied with the cosine of the angle between those two vectors or mathematically written $u \cdot v = |u| \ |v| \cos \alpha$:



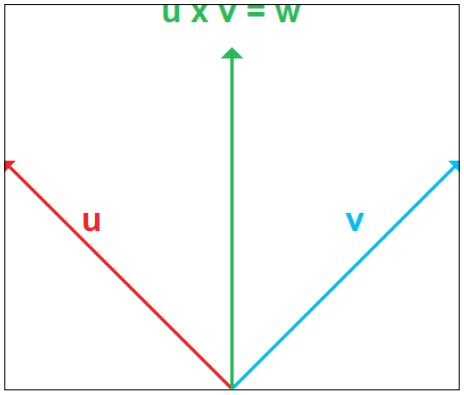
Assume that you have two f32 vectors called u and v, this is how you do dot product in Irrlicht:

```
f32 dot = u.dotProduct(v);
```

Mostly we use dot product to get the angle between two vectors. Irrlicht provides some helper functions for such tasks. For example, to get the angle between u and v vectors you can simply call `u.getAngleWith(v)`; and it'll return a degree value between these two vectors.

Cross product

The cross product between the two vectors u and v is notated as `u × v`. The result is the vector which is perpendicular to both u and v:



Magnitude (length)

The length of a vector is also known as the magnitude of that vector. It can be calculated using simple Pythagoras' theorem. For example, for 3D vectors:

$$r = \sqrt{x^2 + y^2 + z^2}$$

You can get the magnitude of a vector, let's say 'u', in Irrlicht by calling `getLength()` method like this:

```
f32 len = u.getLength();
```

The return value type will depend on the type of vector since vectors are implemented as template classes.

Unit vector

A vector with the magnitude of one is called a unit vector. Unit vectors are normally used to represent the direction vectors where the scalar values are not a particular interest.

Normalization

Vector normalization refers to making the magnitude of a vector to become one. It is important to normalize vectors in some cases because multiplying a vector will also multiple with its scalar values. If we don't want to alter the magnitude of the original vector we need to normalize the other vector before multiplication so that its magnitude becomes one. Normalizing a vector involves two steps, first calculation of the length of the vector and second, dividing each scalar component of the vector with that length. In Irrlicht, it can be done by calling `normalize()` method like this:

```
u.normalize();
```

And vector '`u`' will be normalized.

Direction vector

You can simply subtract the source vector from the target vector to get the direction vector from source to target. This is quite useful since we can then move an object from one point to another in that direction. We can even use this technique to implement simple AI stuffs. For example, creating an enemy AI that follows wherever the player goes. By just subtracting the enemy position vector from the player position vector we can get that direction vector where the enemy should follow.

Time for action – moving a ball

With the knowledge acquired in this chapter and drawing 2D examples from previous chapters, we'll try to write a simple program where we'll have a ball moving towards wherever you click on the screen.

1. Let's define some global variables first:

```
ITexture* image;

vector2d<f32> position;
vector2d<f32> target;
vector2d<f32> direction;

f32 speed = 0.2f;
s32 screenWidth = 640;
s32 screenHeight = 480;
Next we need to implement event receiver for mouse events.
class MyEventReceiver: public IEventReceiver
{
  public:
    virtual bool OnEvent(const SEvent& event)
```

```
    {
      if (event.EventType == EET_MOUSE_INPUT_EVENT)
      {
        if (event.MouseInput.isLeftPressed())
        {
          target.X = (f32)event.MouseInput.X;
          target.Y = (f32)event.MouseInput.Y;

          direction = target - position;
          direction.normalize();
        }
      }
      return false;
    }
};
```

2. Then an `update` method to move the ball and checking window's size boundary:

```
void update(f32 deltaTime)
{
  position += (speed * deltaTime) * direction;
  if (position.X > screenWidth - image->getSize().Width ||
    position.X < 0)
    direction.X *= -1;

  if (position.Y > screenHeight - image->getSize().Height ||
    position.Y < 0)
    direction.Y *= -1;
}
```

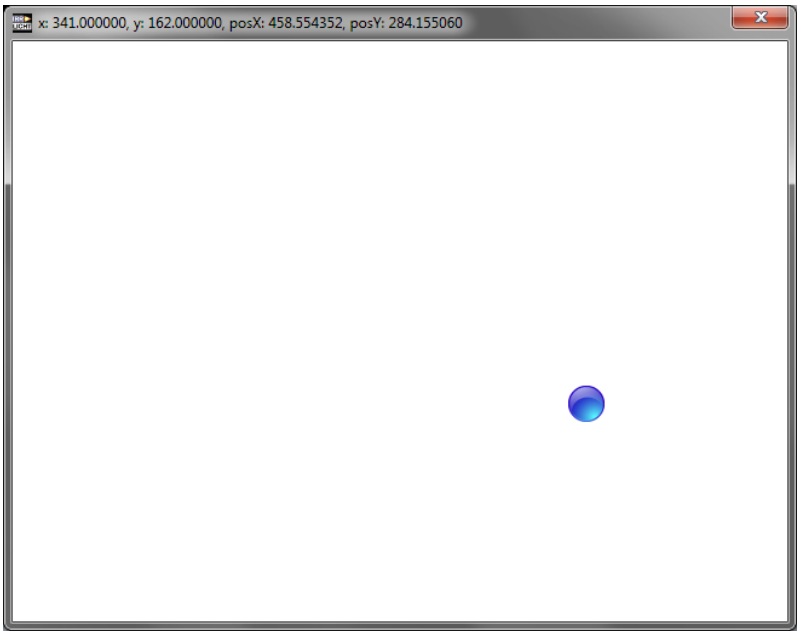
3. Copy the `ball.png` to your project folder. Load the ball image and initialize the position and direction vector:

```
image = driver->getTexture("../../media/ball.png");
  position.set(0,0);
  direction.set(1,1);
```

4. Calculate the delta time difference inside "game loop" and pass that delta time to `update` method. Draw the ball image at the updated position:

```
update(deltaTime);
..
  driver->draw2DImage(image,
    position2d<s32>((s32)position.X, (s32)position.Y),
    rect<s32>(0, 0, 32, 32), 0, SColor(255, 255, 255, 255),
    true);
```

Build and run the application. You should see a ball moving in your window. Click somewhere inside your window and the ball should approach there:



What just happened?

Unlike previous exercises, we declared some variables outside the main function because we want to access them from the whole program scope. Later we can wrap them inside a class but for the sake of simplicity let's keep things like this for now. We declared the position vector to hold the current position of our ball and the target vector to hold the position where the mouse click occurs. And then we implemented our event receiver to receive a mouse click event. Now when a mouse click occurs we capture the X and Y values to our target vector. Then we subtract the current position vector of our ball from the target vector resulting in the direction vector. Since we only need the direction we normalize our direction vector to become a unit vector. Irrlicht vector class has already overloaded the operators and thus, we can just "-" them together.

Then we implement our update method. It expects a delta time value to move the ball in constant speed across different PCs. The idea is on the very fast PCs the time difference between each frame will be very small. So we multiply that delta time with our speed value and the ball will move just a little bit of distance. But since the machine runs very fast it'll get higher FPS and the movement code will be called many times higher than on a slow PC. On a slower PC we'll get a lower FPS but the delta time, time difference between each frame, will be higher.

So, if we multiply that higher delta time with our speed value, our ball will move more distance, resulting in approximately the same amount of distance travelled across different PCs. Then we multiplied that scalar value with our direction vector and add up to the current position of the ball. The latter part is just checking whether the ball hits the borders of our window. If it does, we'll just reverse the direction by multiplying that component with -1.

And in the game loop between beginScene and endScene we draw our ball at the updated position.

Your complete code should look something like this:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
  #pragma comment(lib, "Irrlicht.lib")
#endif

ITexture* image;

vector2d<f32> position; //current position
vector2d<f32> target; //mouse click positions
vector2d<f32> direction;

f32 speed = 0.2f;
s32 screenWidth = 640;
s32 screenHeight = 480;

class MyEventReceiver: public IEventReceiver
{
public:
  virtual bool OnEvent(const SEvent& event)
  {
    if (event.EventType == EET_MOUSE_INPUT_EVENT)
    {
      if (event.MouseInput.isLeftPressed())
      {
        target.X = (f32)event.MouseInput.X;
        target.Y = (f32)event.MouseInput.Y;

        direction = target - position;
        direction.normalize();
```

```
        }
      }
      return false;
    }
  };

  void update(f32 deltaTime)
  {
    position += (speed * deltaTime) * direction;
    if (position.X > screenWidth - image->getSize().Width ||
      position.X < 0)
      direction.X *= -1;

    if (position.Y > screenHeight - image->getSize().Height ||
      position.Y < 0)
      direction.Y *= -1;
  }

  int main()
  {
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
      dimension2d<u32>(screenWidth, screenHeight), 16, false, false,
      false, 0);

    if (!device)
      return 1;

    IVideoDriver* driver = device->getVideoDriver();

    MyEventReceiver receiver;
    device->setEventReceiver(&receiver);

    image = driver->getTexture("../../media/ball.png");

    position.set(0,0);
    direction.set(1,1);

    u32 then = device->getTimer()->getTime();

    while (device->run())
    {
      const u32 now = device->getTimer()->getTime();
      const f32 deltaTime = (f32)(now - then);
      then = now;

      update(deltaTime);

      driver->beginScene(true, true, SColor(255,255,255,255));

      driver->draw2DImage(image,
        position2d<s32>((s32)position.X, (s32)position.Y),
        rect<s32>(0, 0, 32, 32), 0, SColor(255, 255, 255, 255),
        true);
```

```
        driver->endScene();
    }
    device->drop();
        return 0;
}
```

Have a go hero – normalization

In the above exercise, try to remove the normalize function call after calculating the direction vector. Examine what happens.

Pop quiz

1. For which purpose are templates used for?
    a. Having multiple instances of one object
    b. Handling multiple data types with just one function or class
    c. Templates are the same as interfaces
2. Which of these options would be a valid template declaration?
    a. `template <class T> class ClassName { ... };`
    b. `template <T> class ClassName { ... };`
    c. `template <class T> ClassName { ... };`
3. What is the range of a u8 type?
    a. 127 to 128
    b. 0 to 255
    c. It does not have any range because it's a pointer
4. Which of these is the correct order for the color components of `video::SColor`?
    a. RGBA
    b. RGB
    c. ARGB
5. In which order are the edges of `core::rect` constructor declared?
    a. Clockwise
    b. Counter-clockwise
    c. Zigzag (First top-left position, then bottom-right position)

6. Which of these texture sizes would be the most optimized texture to load into an Irrlicht application?

    a. Width of 258 pixels, height of 258 pixels

    b. Width of 1024 pixels, height of 512 pixels

Summary

We learned about the most important data types that are present in the Irrlicht 3D graphics engine and also the practical application of vectors.

Specifically, we covered:

- C++ Templates
- Irrlicht's own type definitions and class types
- Manipulating vectors
- Frame rate independent update

We also touched upon relevant methods by Irrlicht's classes.

Now that we've learned which data types exist and how to use them, we will move on to more sophisticated topics and our first step is scene management in Irrlicht, which will be covered in the next chapter.

6
Managing Scenes

After our short introduction into the 2D capabilities of the Irrlicht graphics engine and learning more about Irrlicht's data types, we will now return to create and learn more about 3D graphics applications.

In this chapter, we shall:

- Learn the definition of scenes
- Create a scene with IrrEdit and load this scene into our application
- Work with our scene

So, let's take a look at it...

What is a scene?

We already have used scenes before, when we loaded the `sydney.md2` model that comes with Irrlicht. Remember the variable of the type `ISceneManager` that stored an instance through the `getSceneManager()` method of the Irrlicht device. This scene manager can load, as well as save scenes, and gives information about the current scene.

A scene may have several children objects, meshes (static or animated meshes), lights, graphical user interface elements, and cameras or scene nodes. We will learn more about the scene nodes in the next chapter.

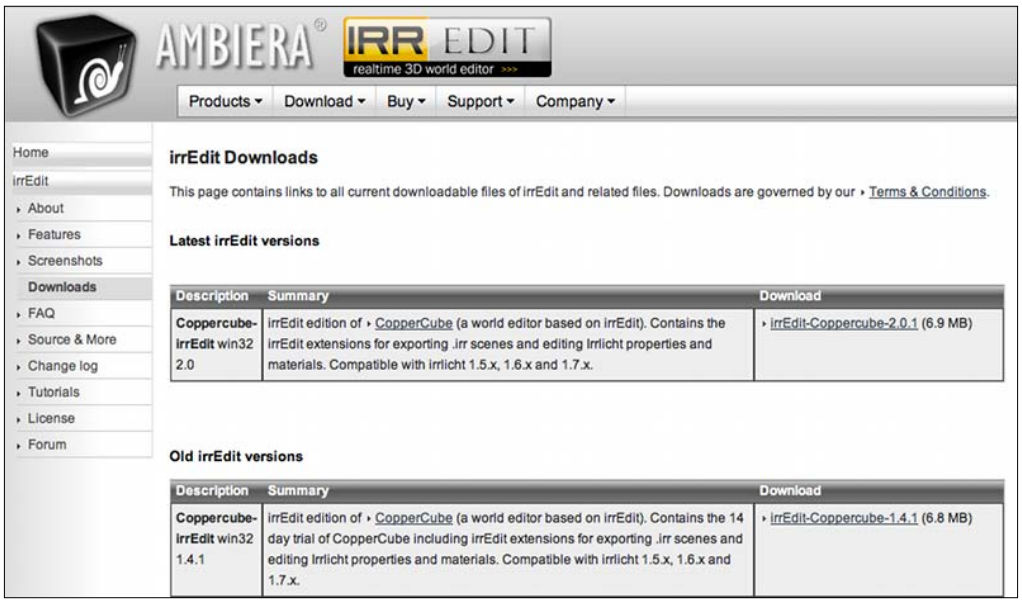
If you would want to create a game in Irrlicht, your game will consist of different scenes, one scene for the main menu, a scene for each level, and so on.

Getting CopperCube/IrrEditCopp

Nikolaus Gebhardt, the creator of the Irrlicht 3D engine has developed an editor to facilitate creating scenes for Irrlicht. This editor, called IrrEdit, is only compatible up to Irrlicht version 1.5. Since Irrlicht 1.6, IrrEdit has been merged into the commercial product CopperCube. Copp.

Windows version

CopperCube/IrrEdit is available at `http://www.ambiera.com/irredit/`. While IrrEdit has been only available for Windows platforms, CopperCube can also be downloaded for Mac OS X:

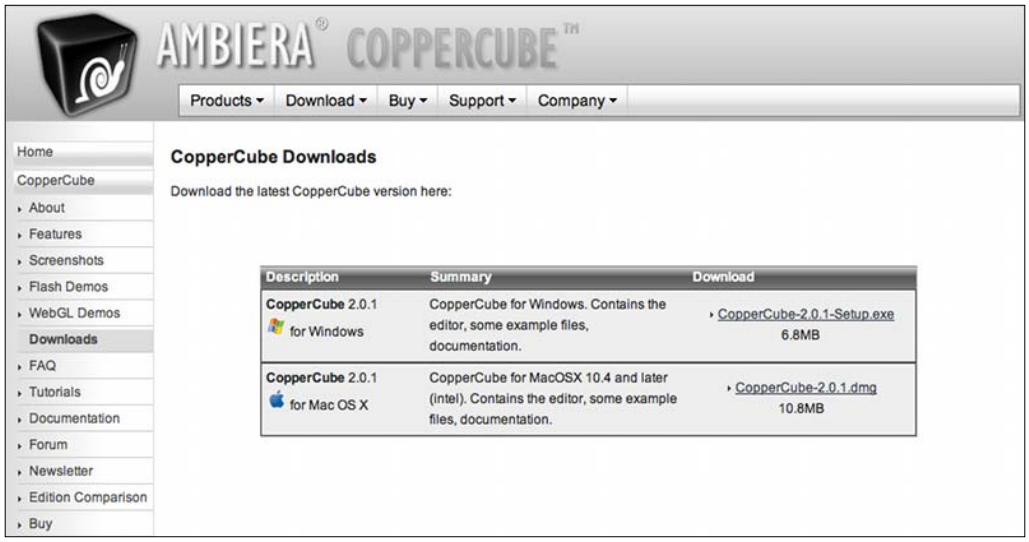


Click on **Downloads** in the left sidebar and you will be redirected to the downloads page. Then click on the file listed under **Latest irrEdit versions** to start the download.

After the download has finished, double-click on `CopperCube-irrEdit-2.0.1-Setup.exe` (or a similar filename if a newer version exists) to start the installation. If you are looking for the Mac OS X version of CopperCube take a look at the next section *Mac OS X version*.

Mac OS X version

The Mac OS X version can only be downloaded from the CopperCube page at `http://www.ambiera.com/coppercube`:



Select the file for Mac OS X by clicking on **CopperCube-*.*.*.dmg** (where the stars represent the current version of CopperCube).

Open the package file and copy `CopperCube.app` to your `application` folder. You probably won't see the `.app` extension. Now you're all set to use CopperCube on a Mac.

What is CopperCube?

CopperCube is a tool for creating applications containing one or more 3D scenes for Windows, Mac OS X, Flash, and WebGL. It allows users to implement events using the scripting language Squirrel or ActionScript, and JavaScript if you plan to export to Flash or WebGL/HTML5 respectively.

While the commercial license of CopperCube costs 99 € in its light version and 299 € in its professional version, there is a trial that works for 14 days, although exporting scenes for Irrlicht works even if the demo version has expired:

The trial and commercial version are exactly the same versions, the trial version can be unlocked by entering a valid serial key at the startup screen.

Why are we using CopperCube?

CopperCube allows us to create scenes in an editor that we can export to the so called `.irr` files and load those into our Irrlicht application.

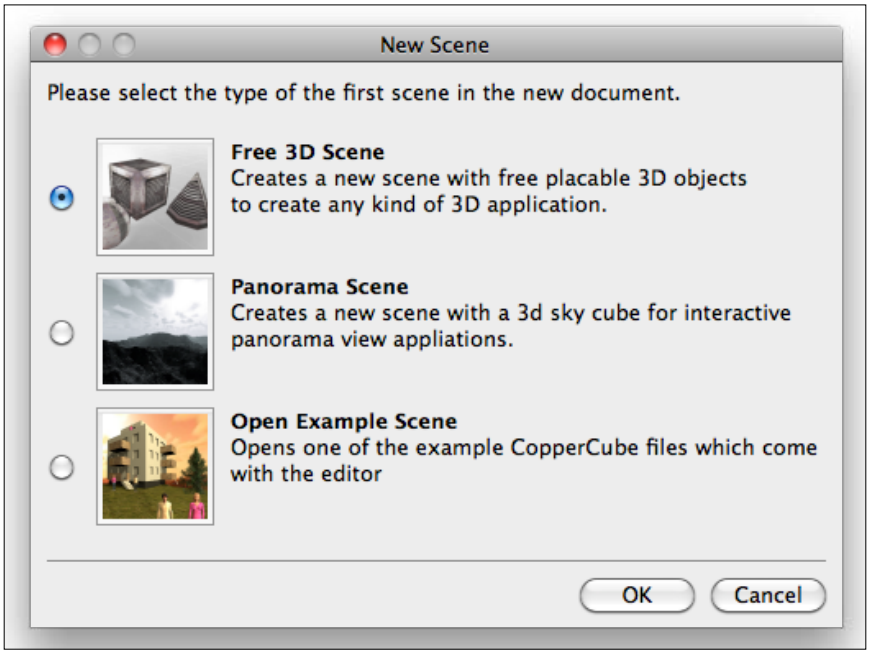
The alternative would be to manually add all the objects we want in our scene, in our code file. If we would have a complex scene, this would bloat up our source file and might make it difficult to manage our source code file. Also, you would need a pretty good imagination for three-dimensionality, because you do not visually see the object that you are going to place in the space.

While on the other hand, every object you place, rotate, and scale, in CopperCube and irrEdit, will be on the exact same spot and with the exact same properties in the application.

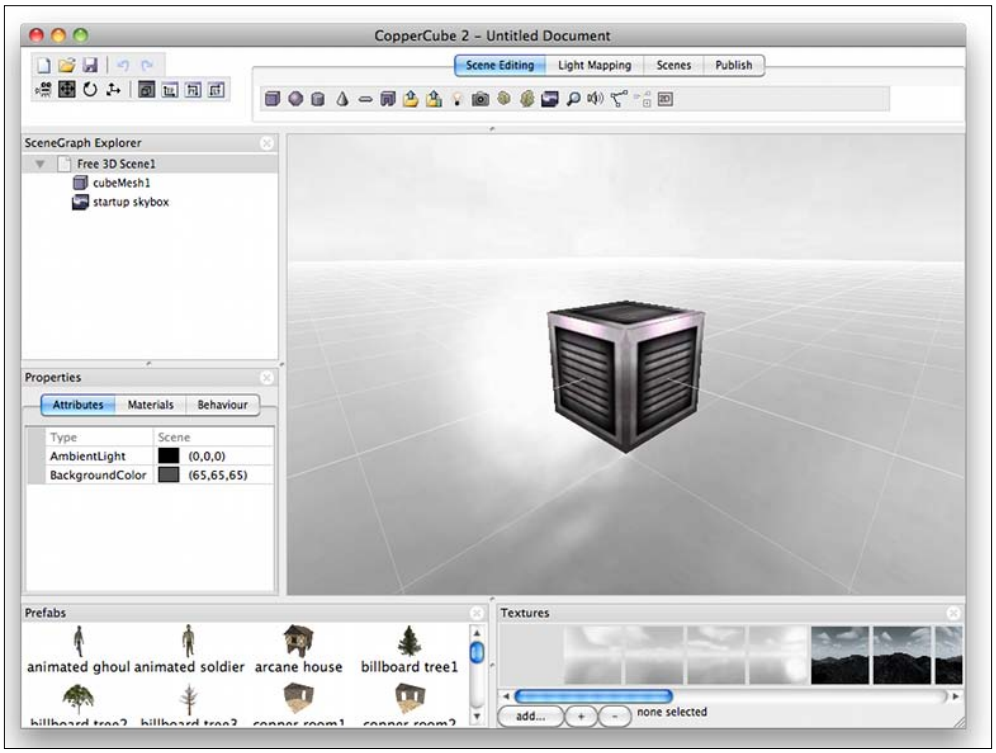
Time for action – creating a new scene

Now that you have installed CopperCube, we will take small steps to create our scene. Let's start with something very simple: creating our first empty scene. Well, almost empty, because even an empty default scene has two objects in it.

1. Start up CopperCube.
2. Make sure **Free 3D Scene** is selected and confirm by clicking on **OK**:



3. Your scene already has a predefined skybox and a textured cube, by default, and should now look as follows:



What just happened?

The CopperCube 3D editor is, in its design, very similar to any 3D graphics application such as Blender, Cinema 4D, LightWave 3D, or 3DS Max.

Navigating in the 3D space is done through the mouse: use the left mouse button to rotate the current camera view and right mouse button, or the mouse wheel, to zoom in or out. Pressing the arrow keys or holding the middle mouse button pans the camera.

Let's explain the buttons that are displayed in the top menu: the toolbar is easy to understand as you may already have seen other applications using the same button style:

These buttons from left to right are for creating, opening, saving files, and undo or redo the last action.

Immediately under those buttons is the toolbar, responsible for navigating the camera as well as objects in 3D space:

Those buttons from left to right have the following effect:

- Move camera
- Move object
- Rotate object
- Scale object

These buttons—if clicked—do the following:

- Switch to perspective mode (where the camera can be moved, panned, and rotated)
- View camera from the top
- View camera from the front
- View camera from the left

The toolbar, on the right side of the previously mentioned buttons has the listed options:

- **Scene Editing**: It provides, for example, geometrical shapes, static and animated meshes, lights, and cameras.
- **Light Mapping**: It creates a light map, out of specified options and the lights in the scene.
- **Scenes**: It lets you manage different scenes. You can only export one scene at a time.
- **Publish**: It allows you to publish one or more scenes as a Windows application, a Mac OS X application, a flash file, or HTML5 and WebGL. We will not be needing this function.

In the bottom bar, prefabricated objects (also known as prefabs) and textures are displayed. Double-clicking on a prefab will place the selected object in the scene, a double-click on a texture will force the selected texture to be drawn on the selected object.

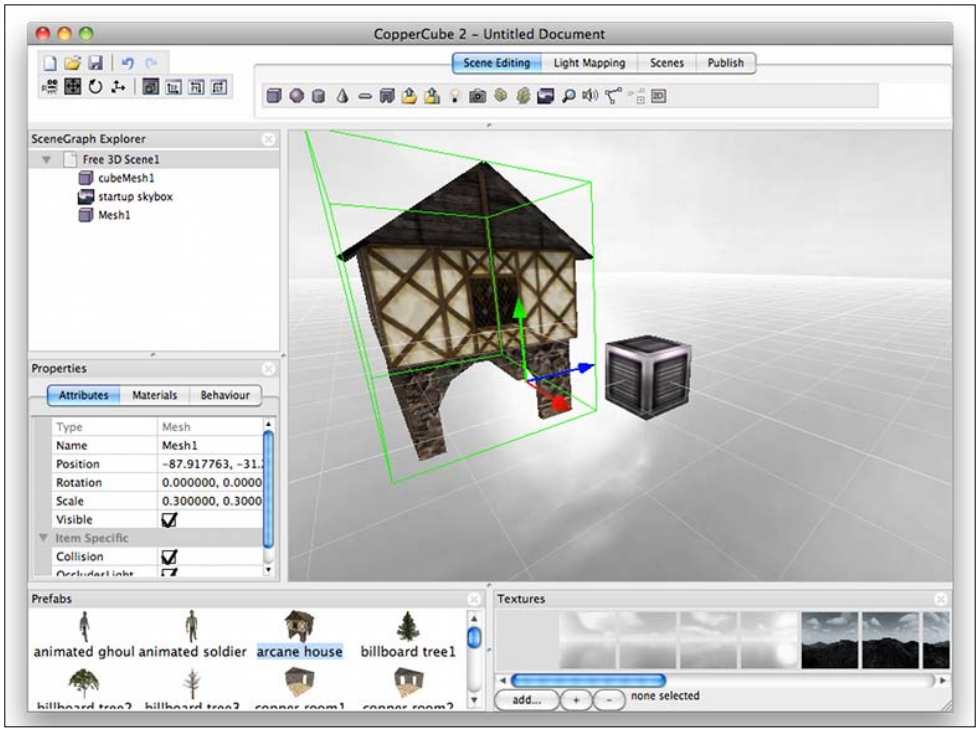
The left sidebar shows a **SceneGraph Explorer**, which displays all the objects currently in the scene in a tree view. You may also have noticed the control element called **Properties** right below it. If an object is selected, attributes of this object will show up and those attributes can be modified. For example, if a mesh is selected, you can also edit the position, rotation, and scale of the mesh or set collision or visible flags.

Time for action – adding meshes to our scene

Now that we have our almost empty scene set up, let's add objects to our scene. CopperCube allows different kinds of objects to be added to the scenes. Let's start with the kind of objects we already know and worked with: meshes.

1. Select **arcane house** in the **Prefab** window and double-click on it to add this mesh to the scene. Move it to the vicinity of the default cube.

2. Keep the house selected and scale down the house to 30% of its original size by changing each value of **Scale** in the **Properties** window from `1.0` to `0.3`:



3. Select the default cube with a left mouse click and delete this object by either pressing *Delete* or selecting **Edit | Delete**.

4. Now add the mesh **animated soldier** to the scene.

5. Scale down the soldier as well so that the proportions fit:

What just happened?

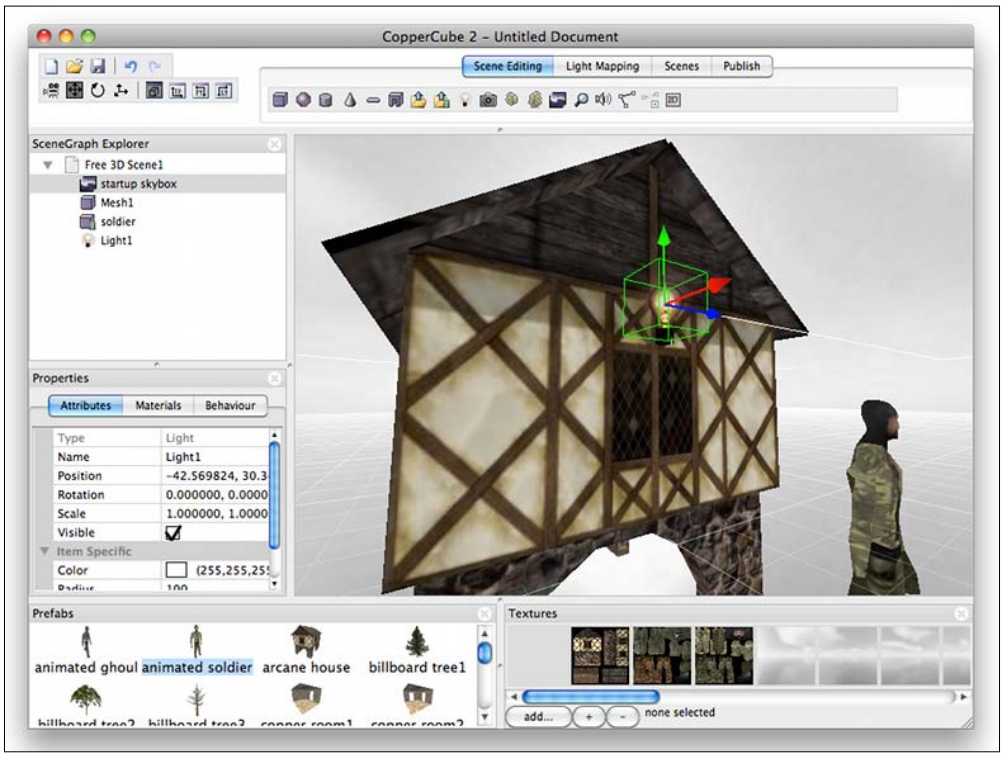
We added two meshes to our scene, one static mesh and one animated mesh. The animated mesh is showing its animation in the editor. If you select the animated mesh and that mesh has a skeletal animation, it will be displayed with blue lines.

Time for action – adding lights

Lights and shadows are an important factor in scenes. They can add realism and set the general mood of a scene. Let's add a light for our scene:

1. Click on the icon that looks like a light bulb to add a light. (The ninth icon from the left.)

2. Move the light to the front of the house:



What just happened?

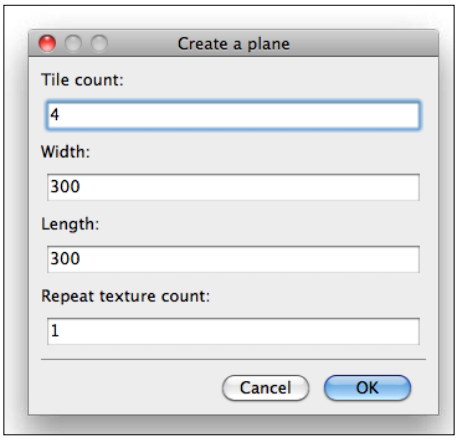
CopperCube differs between placing lights and actually rendering those lights. Note that the lighting in your scene has not changed and shadows are not yet rendered. To do so you will need to enable dynamic light in the object's **Materials** properties window.

In the preceding exercise, we just placed a light, we will pick up rendering lights later in this chapter.

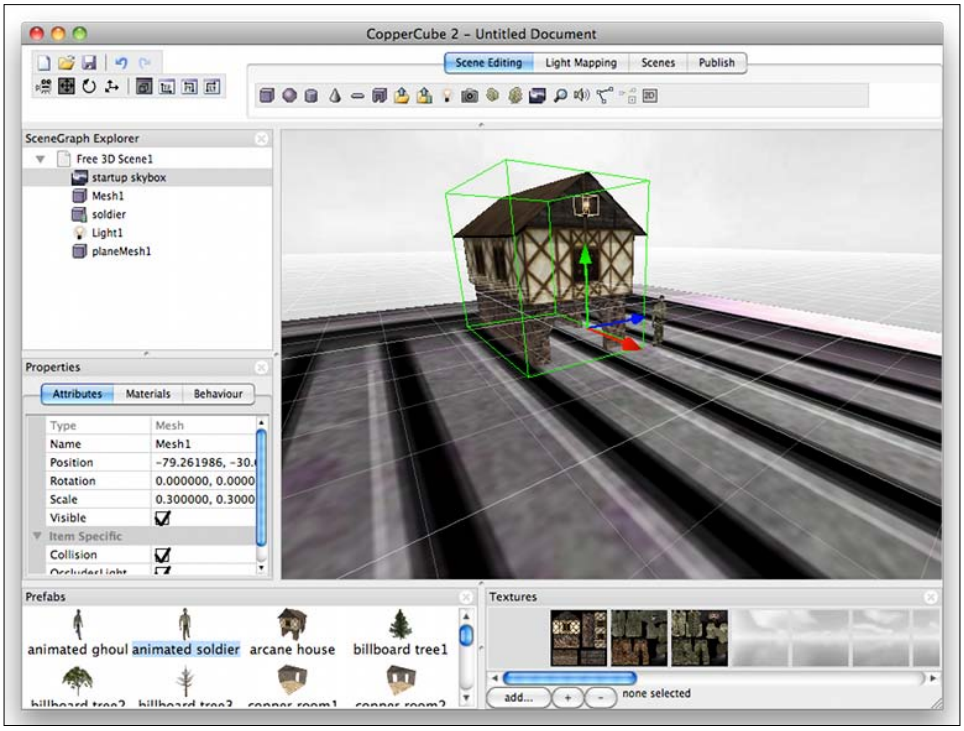
Time for action – adding geometrical objects

CopperCube has a variety of geometrical objects to insert into a scene: cubes, spheres, cylinders, cones, and planes. Our meshes are hanging in the air. Let's change that by adding a plane to create the illusion that there is a ground our meshes are standing on.

1. Click on the icon that looks like a plane. (The fifth icon from the left.)
2. Use the following settings to create our plane:



3. Move the Y-position of the plane until it looks like the meshes are standing on the plane:



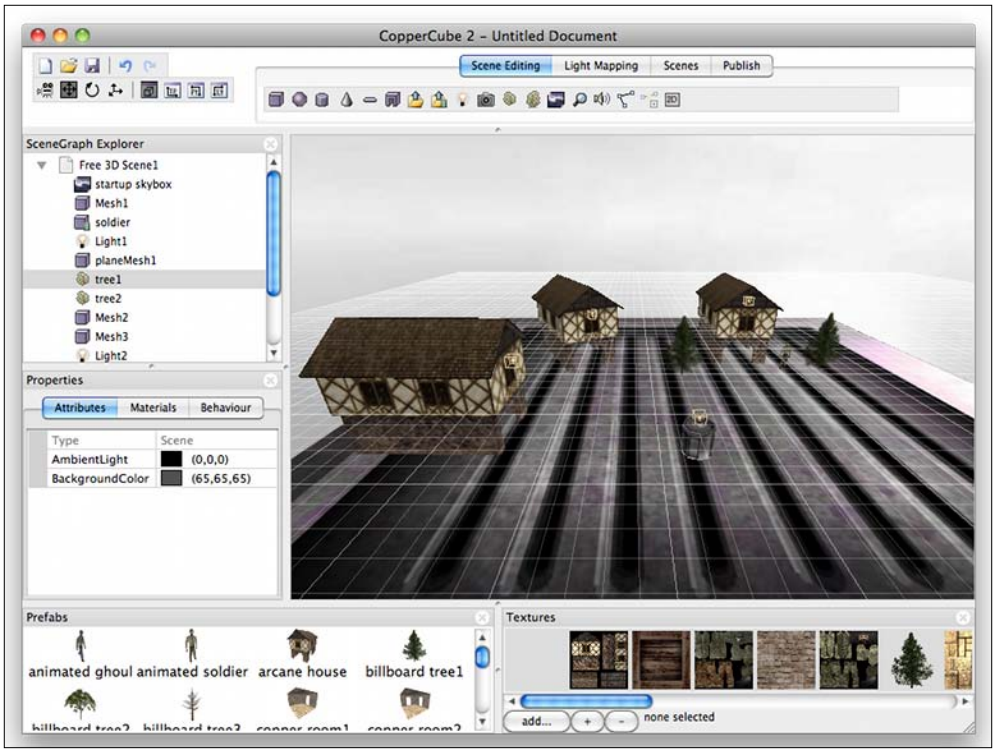
What just happened?

We just added a plane to act as a ground for our objects. This is necessary to have a more realistic scene when we render the lights later in this chapter.

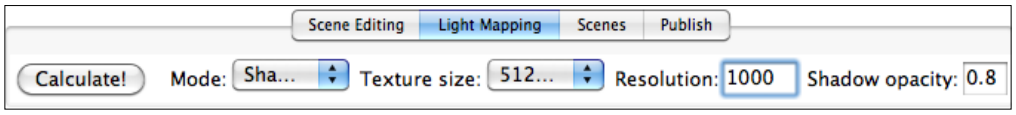
Time for action – finishing up our scene

We already made a nice scene, let's add some complexity and pre-rendered lights and shadows. Our final scene should contain three houses, two trees, and more lights.
Let's begin:

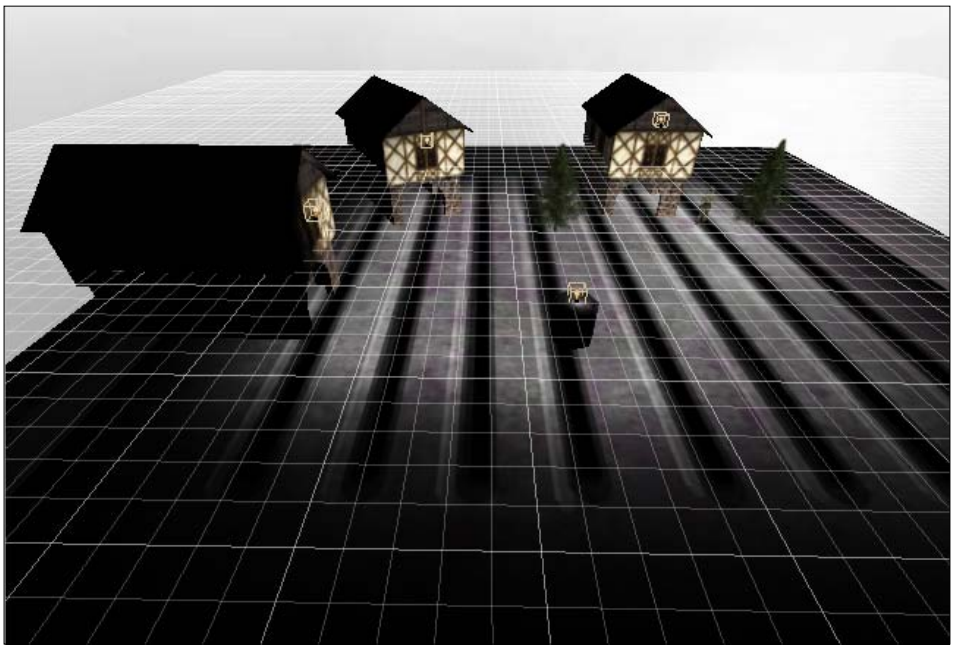
1. Select the house we already have. Right-click on the house and select **Clone**.
2. An identical house has just been spawned on the exact same position as the original house. Move the new house a bit to the center of the plane. Repeat this process and move the third house to the other side of the plane.
3. Change the Y-rotation of the second house by selecting the house and changing the second value of **Rotation** in the **Properties** window to `-30.0`. Then change the Y-rotation of the third house to `-90.0`.
4. Add **billboard tree1** to the scene and clone this tree. Place the first to the left of the first house and the other tree to the right of the first house. You need to scale down the tree to fit the proportions.
5. Create two clones of the light and place each at the front of each building.
6. Create a cylinder (third icon from the left) and move it to the center of the plane.
7. Add an additional light on top of the cylinder.
8. Your scene should now look as follows:



9. Switch to **Light Mapping**, use the default value and click on **Calculate** to render lights and shadows:



10. Your final scene should look as follows:



What just happened?

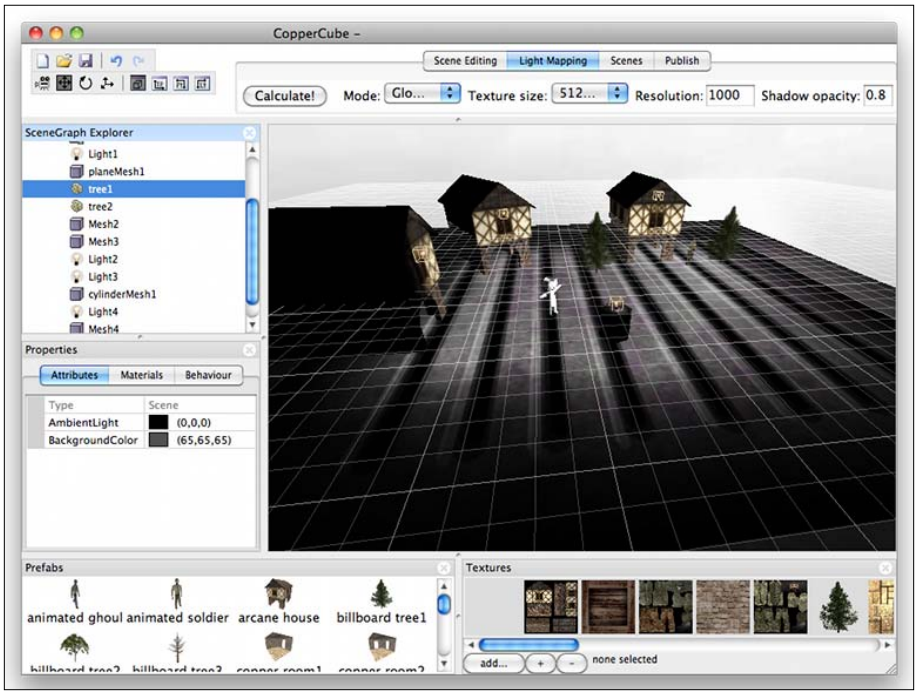
In steps 1 to 8, we begin constructing our more complex scene, using cloning as a tool to have more objects on the screen within a short period of time.

There are four options when rendering lights in the scene:

- **White**: It bakes the scene completely without lights.
- **Diffuse**: It bakes the scene with low contrast and low shadows.
- **Shadows**: Lights and shadows will be baked. Shadows will only appear if the light is able to cast shadows and the objects are able to receive shadows.
- **Global Illumination**: Light is reflected by any surfaces and add more realism to scenes.

Have a go hero – finishing up our scene

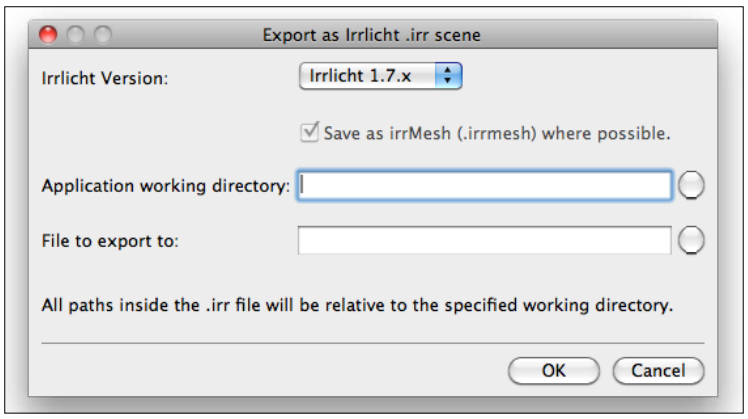
Remember our little human-like figure, we created with Blender. Try to insert this mesh into our scene. Use the **Import**, a static mesh button provided by CopperCube. The scene with the model should then look as follows:



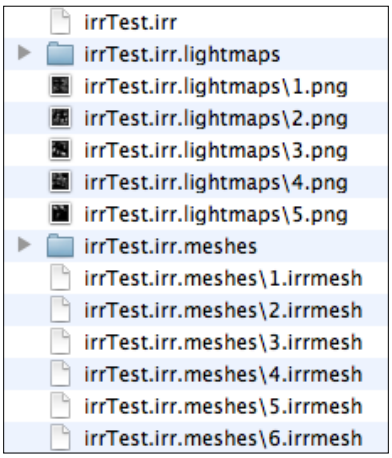
Time for action – exporting our scene

Our scene is finished, let's export this scene to be able to load into an Irrlicht application. Exporting a scene with CopperCube takes only a few steps.

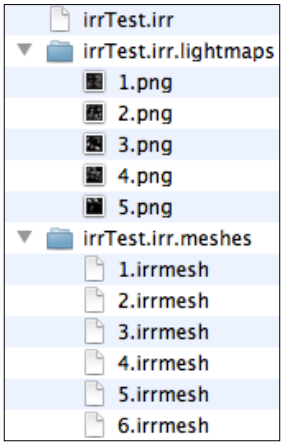
1. Select **File | Export | Export current scene as Irrlicht scene (.irr)...**.
2. Select a directory and a filename where the Irrlicht scene should be exported to:



3. Your exported directory should look as follows:



There is a bug in the Mac OS X version of CopperCube that leads to the Irrlicht scene not to be exported correctly. You need to arrange the folder structure as described in the following screenshot. This bug does not occur in Windows:



Time for action – loading an Irrlicht scene

We are now ready to load the Irrlicht scene into our Irrlicht application. Our starting point is, once more, the template application from *Chapter 2*, *Creating a Basic Template Application*.

1. Add `ISceneManager* smgr = device->getSceneManager();` after the line `IVideoDriver* driver = device->getVideoDriver();`.
2. Add `smgr->loadScene("myScene.irr");` immediately after that. Make sure to replace `myScene.irr` with the filename, you actually called your exported file.
3. Add a camera: `smgr->addCameraSceneNode(0, vector3df(0, 30, -40), vector3df(0, 5, 0));`.
4. Add `smgr->drawAll();` in the game loop between `beginScene()` and `endScene()`:

Make sure all Irrlicht scene content files are in the same directory as the executable. If any model would not be loaded correctly, the scene itself will still be loaded, but just without this model.

The complete code for this example is as follows:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                             dimension2d<u32>(640, 480), 16, false,
                             false, false, 0);

    if (!device)
        return 1;

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    smgr->loadScene("myScene.irr");

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40), vector3df(0,
                             5, 0));

    while (device->run())
    {
        driver->beginScene(true, true, SColor(255, 255, 255, 255));

        smgr->drawAll();

        driver->endScene();
    }

    device->drop();

    return 0;
}
```

Summary

We learned about Irrlicht's scenes and Irrlicht's preferable 3D editor, CopperCube.

Specifically, we covered:

- What the scenes are
- How to use CopperCube/IrrEdit to create scenes for Irrlicht
- How to load scenes made with CopperCube into our Irrlicht application

Now that we've learned about scenes and easy ways to construct and load complex scenes without a whole lot of code, let's go ahead and examine the basic graphical type of the Irrlicht engine in the next chapter: *Using Nodes—The Basic Objects of the Irrlicht 3D Engine*.

7

Using Nodes—The Basic Objects of the Irrlicht 3D Engine

In the previous chapters we learned how to load different 3D objects into our Irrlicht application, how to manipulate those 3D objects, and how to manage scenes.

In this chapter, we will learn about:

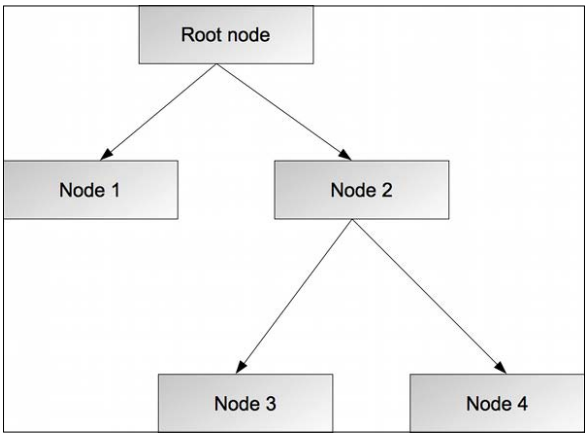
- Scene nodes
- Adding children to node objects
- Animating nodes
- Setting up custom scene nodes

So let's get started.

What is a node?

Think of nodes as the most basic objects visible on the screen, to be more exact each object that is in the scene graph can be traced back to the `ISceneNode` interface. This scene node contains attributes about the current position, scale, and rotation as well as read-only information about the scene graph, parent, and child objects. Irrlicht also provides a method to clone scene nodes.

A node can have any number of child objects, and those child objects, which have to be node objects themselves, can have as many node objects as defined and so on. Nodes on the same level are considered siblings, a node a level above relative to the current node is called parent and a node a level beneath to the current node is called, as already mentioned, a child:



Overview of different nodes

If you take a closer look at the previous examples in this book, you might already have noticed that we used nodes before as well, as there are different kinds of nodes depending on the task they provide. Consequently, this section will show you the most important node types and what they are being used for.

Scene node

Our basic scene node interface is `ISceneNode`. As already mentioned a scene node contains information about the current position. The method `getPosition()` retrieves the position as a three-dimensional float vector, while the setter requires a vector to set the new position.

Any node that is being added has a relative position to the parent node and an absolute position in the world. If the parent node is being moved or manipulated those changes will be assigned to the child nodes. The mentioned methods get or set the relative position to the parent node; only if the parent node is the root node the absolute position will be set. You can still get the absolute position through `getAbsolutePosition()` which returns a `vector3df` as well. You can't set the absolute position if the parent node is not the root node.

The constructor of `ISceneNode` needs the following parameters:

- Parent node (inherited from `ISceneNode`)

- Instance to a scene manager from the type `ISceneManager`
- id from the type `s32` which is a number to identify this instance of `ISceneNode` in the current scene
- Position of this node as `vector3df` (optional)
- Rotation of this node as `vector3df` (optional)
- Scale of this node as `vector3df` (optional)

Camera scene node

A camera scene node represented by `ICameraSceneNode` renders anything that is marked visible in the scene graph from the current point of the view of the camera.

Billboard scene node

A billboard is an object which always faces the camera. It is primarily used for particles such as fire or explosions as well as outdoor vegetation such as trees, grasses, and so on. The interface is called `IBillboardSceneNode`.

Mesh scene node

The mesh can be either `IMeshSceneNode` for a static mesh or `IAnimatedMeshSceneNode` for an animated mesh. More information on meshes for example, what data formats can store animations, can be found in *Chapter 3*, *Loading Meshes*.

Additionally, the interface `IBoneSceneNode` which also inherits from `ISceneNode` provides access to the joints in an animated skeletal mesh.

Light scene node

A light scene node as `ILightSceneNode` is a dynamic light which can be placed in a scene either as a directional or point light.

Normally a light scene node makes the objects to be lit and seen. However, there is also a more specialized light scene node called `IVolumeLightSceneNode` which allows the light itself in the scene to be seen for example, light beams through a window, dust, fog, or steam.

Dummy transformation scene node

This is a special case. The `IDummyTransformationSceneNode` interface does not react to any manipulation by `getPosition`, `setPosition`, `getRotation`, `setRotation`, `getScale`, `setScale` nor does it render itself.

Its purpose is to add transformations anywhere in the scene graph.

Particle system scene node

The particle system (`IParticleSystemSceneNode`) is used to create all kinds of particle effects and rendered in the scene such as explosions, fire, or smoke to name a few.

Terrain scene node

As the name implies the interface `ITerrainSceneNode` allows you to create landscapes out of height maps through its `loadHeightmap` or `loadHeightmapRAW` method.

A height map is usually a gray scale image where a white pixel will transform into the highest point of this terrain and a black pixel to the lowest point. Once we've defined the height map for our terrain we can apply any texture to it for our desired appearance.

Text scene node

The interface `ITextSceneNode` is used to draw two-dimensional text at a specified location in the application. It also allows you to modify the color and, of course, change the text. This scene node is mostly used for debugging purposes.

Working with nodes

Now that you theoretically know how to add child nodes to nodes, let's see how it works in the real application.

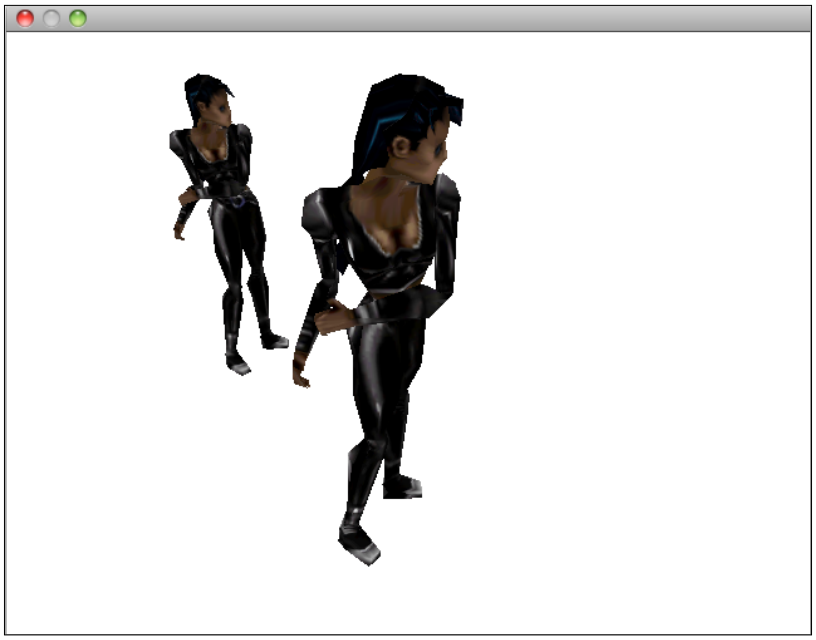
Time for action – working with nodes

We start once again with the template application we created earlier. Also copy everything needed to load the Sydney mesh file (`Sydney.md2` and `Sydney.bmp`) included with the Irrlicht package to where your application can find those, usually the same directory your application executable is in.

If you are unsure what to do, at any point of this example take a look at the first examples from *Chapter 3*, *Loading Meshes* where this is explained in more detail.

1. Add a variable called `smgr` from the type `ISceneManager` which stores an instance of the scene manager via `device->getSceneManager()`.
2. Call `smgr->drawAll();` in the application's loop.
3. Now switch back to the line you added before. Add an animated mesh which loads the Sydney mesh through `smgr->getMesh()`.
4. Directly after that, add `IAnimatedMeshSceneNode* node = smgr->addAnimatedMeshSceneNode(mesh);`.

5. Repeat the last two steps but this time call the assigned variables `mesh2` and `node2` respectively.
6. Add the condition `if (node && node2)` and a curly brace.
7. Disable material lighting for `node`.
8. Add the Sydney material to `node`.
9. Repeat step seven and step eight for `node2`.
10. Link `node2` as a child node to `node` with `node->addChild(node2);`.
11. Set the relative position of node2 to node with `node2->setPosition(vector3df(-30, 0, 40));`.
12. Your code from the `if` condition is now done. Add a closing curly bracket.
13. Add a camera scene node by calling `smgr->addCameraSceneNode(0, vector3df(0, 30, -40), vector3df(0, 5, 0));`.
14. Compile and see the result of what you've done:



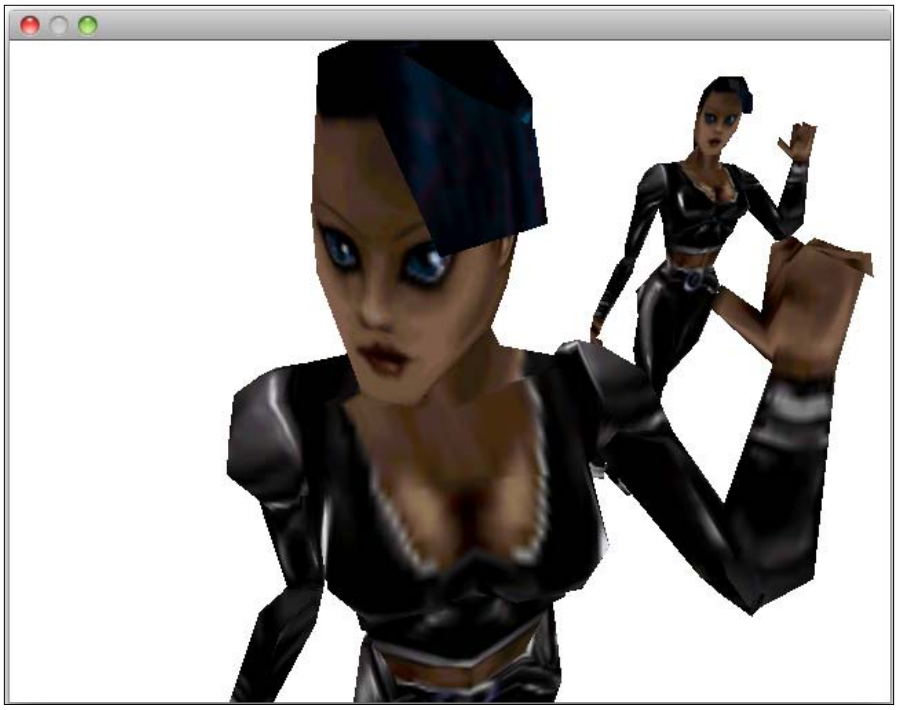
15. Translate the position of `node` by -30 units on the Z axis:

16. Now rotate the node about seventy degrees on the Y axis.

Here is the complete source code of this example:

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace video;

#if defined(_MSC_VER)
  #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
  IrrlichtDevice* device = createDevice(EDT_OPENGL,
    dimension2d<u32>(640, 480), 16, false, false, false, 0);

  if (!device)
    return 1;
```

```
  IVideoDriver* driver = device->getVideoDriver();
  ISceneManager* smgr = device->getSceneManager();

  IAnimatedMesh* mesh = smgr->getMesh("sydney.md2");
  IAnimatedMeshSceneNode* node =
    smgr->addAnimatedMeshSceneNode(mesh);

  IAnimatedMeshSceneNode* node2 =
    smgr->addAnimatedMeshSceneNode(mesh);

  if (node && node2)
  {
    node->setMaterialFlag(EMF_LIGHTING, false);
    node->setMaterialTexture
      (0, driver->getTexture("../../media/sydney.bmp"));

    node2->setMaterialFlag(EMF_LIGHTING, false);
    node2->setMaterialTexture
      (0, driver->getTexture("../../media/sydney.bmp"));

    node->addChild(node2);
    node2->setPosition(vector3df(-30, 0, 40));

    node->setPosition(vector3df(0, 0, -30));
    node->setRotation(vector3df(0, 70, 0));
  }

  smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
    vector3df(0, 5, 0));

  while (device->run())
  {
    driver->beginScene(true, true, SColor(255,255,255,255));

    smgr->drawAll();

    driver->endScene();
  }

  device->drop();

  return 0;
}
```



What just happened?

Even though most of the first nine steps were already covered in Chapter 3, we added the second node to the first node. We need to change the position of the second node or the second node would show up at exactly the same position as the first node and you wouldn't be able to see any difference.

If you change the position or rotation of the parent node each child node will be changed as well, as you can clearly see in step 15 and 16.

You could also add an additional condition to check if the meshes have been correctly loaded and if not close the application with the following lines of code:

```
if ((!mesh) || (!mesh2))
{
  device->drop();
  return 1;
}
```

Animating nodes

Remember how we manipulated our mesh in *Chapter 3*, *Loading Meshes* by manually hard coding. Sure, you could animate your nodes by having a variable whose value changes each frame, or whenever you set it to change, and assigning this value to the setter as a parameter.

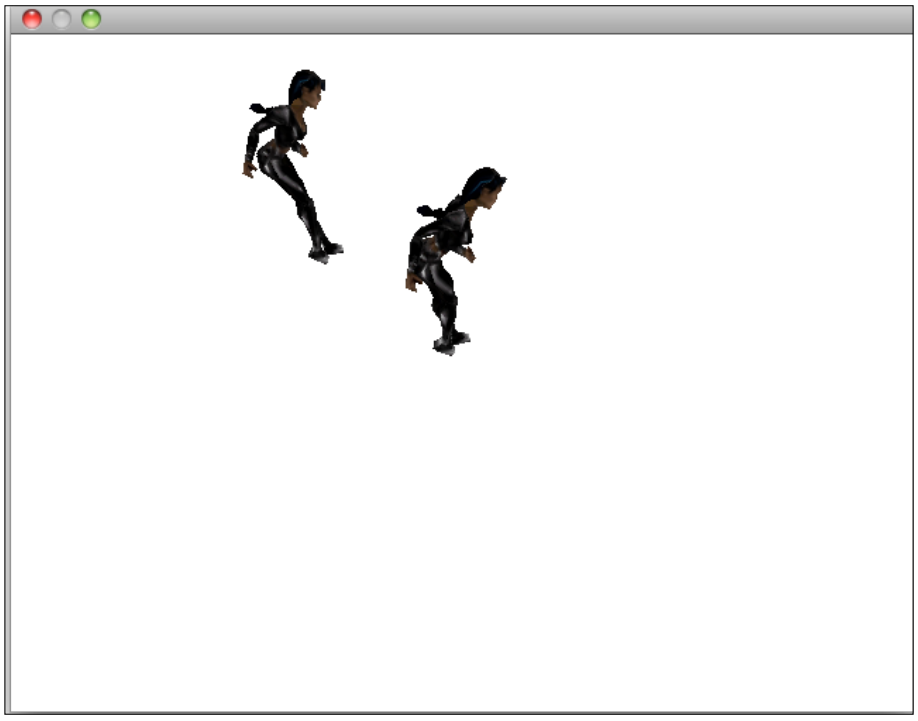
Irrlicht offers different node animators which simplify the process of animating scene objects.

Time for action – animating nodes

Let's dive right in with the source code from the previous example:

1. Delete the lines `node->setPosition(vector3df(0, 0, -30));` and `node->setRotation(vector3df(0, 70, 0));`.
2. Change the line where you add the camera to the scene manager to `smgr->addCameraSceneNode(0, vector3df(0, 80, -70), vector3df(0, -40, 0));`.
3. Before the line where you added the camera insert the definition `ISceneNodeAnimator* anim` which is being assigned with `smgr->createRotationAnimator(vector3df(0.0f, 0.8f, 0.0f));`.
4. Check if `anim` has been assigned correctly, then add the animator to node with `node->addAnimator(anim);`, then drop the animator and set `anim` to `0`.

5. Compile and run the application. You can now see two meshes rotating with one mesh rotating the Y axis and the other mesh rotating around the first as its center:



What just happened?

There are several kinds of animators. This is a list of the most important and most used animators and their respective parameters:

Fly circle

The first parameter is the center of the circle around which the node is going to be animated. The second parameter is the radius of the circle as an `f32` type, the third is the speed of this animation as an `f32` type. The next parameter is the direction in which the node is aligned. The fifth parameter is the start position at the circle which can take values from `0.0f` to `1.0f` where `0.5f` would set the position at the half of the circle. The last parameter specifies if the radius is an ellipse by taking the same range of values as the parameter before. All parameters in this method are optional.

Fly straight

This animator lets a node fly from one position to another within a specified amount of time in milliseconds. The first parameter is the start point, the second parameter is the end point, and the third parameter is the time that passed between moving from the first to the second position. The fourth parameter determines whether the animation should be looped or not. This parameter is optional and set to `false` by default. The fifth and last parameter sets if the animation should reverse after it's finished and create a ping pong–like effect. This parameter is optional as well and set to `false` by default also.

Follow spline

This animator lets the node follow a specified path. The first parameter is the start time of the animation. The second parameter is a collection of points as an array of `vector3df` types. The third parameter defines the speed of the animation. The fourth parameter sets the tightness of the splines.

The fifth parameter is whether to loop the animator by following the spline again from the start. The sixth parameter "pingpong" basically tells the animator to follow the spline backwards after the end has been reached and thus, following the path back and forth.

Rotation

This animator is the easiest to use. It requires a `vector3df` with each value being the speed of how fast the node rotates around the specified axis.

Delete

When using this animator the scene node will be deleted automatically after the specified time in milliseconds.

Texture

Switches through a specified array of textures and applies those to the node. The second parameter is the time that passes between switching each texture of the list. The third parameter determines if the animation should be looped and it is set to be a single animation.

Collision response

This animator detects collision and response in the scene automatically.

Each animator will be created by first creating a variable from the type `ISceneNodeAnimator` and then call the method `create[Type of Animator] Animator` of an instance of `ISceneManager`. For example if we wanted to create a fly straight animator, all we had to do is:

```
ISceneNodeAnimator* anim = smgr->createFlyStraightAnimator(...);
```

Even though `ISceneNodeAnimator` has `ISceneNode` in it, it does not actually inherit from `ISceneNode`.

Once we've created our animator, we can add to our scene node by calling the method `addAnimator`. After that we don't need this animator so we can delete this object by calling the `drop()` method. You should always call `drop()` if you don't need the animator any more.

The method `hasFinished()` returns `true` if the animation has finished. This only works if the animation does not run in a loop.

Have a go hero – exploring other animators

We applied a rotation animator in the above example. Try to apply a fly circle animator to a node yourself. At the same time use the delete animator to remove the node from the scene after 10 seconds. You'll need to use `createFlyCircleAnimator` and `createDeleteAnimator` respectively. Refer to the above explanation for the required parameters and notice that you can add multiple animators to a scene node.

Adding a custom scene node

After you know what a node is, which types of scene nodes exist, and how to animate any node it is now time for us to create our very own custom scene node. This may seem a bit abstract and theoretical and you may be wondering why would we do such a thing?

Let's take this example. We would want to create a game, a simple space shooter to be more exact. So we control a spaceship and fly through the infinity of a universe and have to destroy enemies until we get to the end of the current level. You probably already have games like this. R-Type, Katakis, or the Xbox Aracade title Gravity Wars are popular games of this sub-genre.

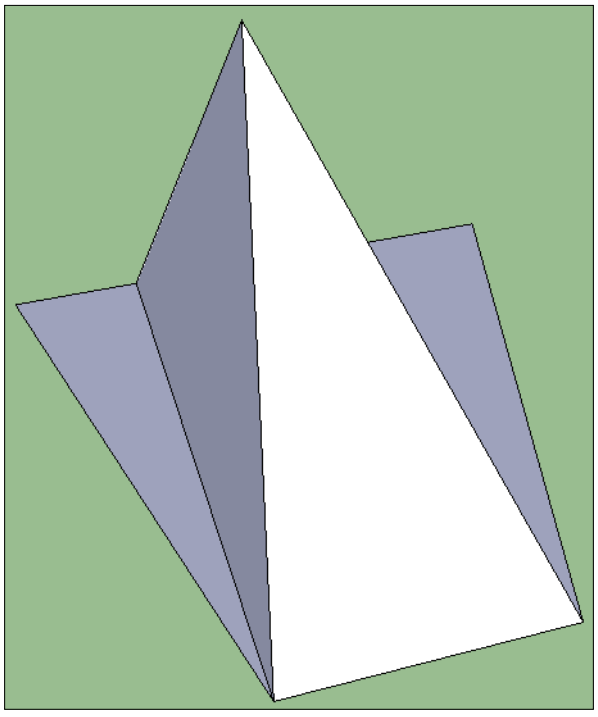
Let's go into detail with this scenario. We would need a few meshes, for our character (the spaceship in our case) and some enemies which will be represented by `IMeshSceneNode` or `IAnimatedMeshSceneNode` depending on if have animated objects or not. Our camera is going to be an `ICameraSceneNode`, while particles, like if shoot or if an enemy explodes, should be generated by a particle system. Alright, but where would custom scene nodes fit in?

Now if we didn't want to use meshes, but instead create the whole object by hand without needing to load any external files, a custom scene node is the way to go here. Once we have this custom scene node which will render a spaceship, we can use it as an attribute inside our game object class (for example, `CSpaceShip` class), together with other data such as remaining hit points, a flag indicating if this object can shoot which could subsequently lead to a particle system being attached to this node, how much energy is left, if this object collides with other objects, and so on.

The best application for custom scene nodes is to render objects that are not covered in the Irrlicht engine by default.

Time for action – adding a custom scene node

Creating a whole game or even a large part of the described scenario would miss the point of this example. We will just create a custom node which should in the end look a spaceship. Well, spaceship might be over exaggerated; a simple spaceship-like object is what we are aiming for. Here is a rough sketch of what our object should look like when it's finished. It basically consists of seven triangles:



We will use the template application from *Chapter 2*, *Creating a Basic Template Application* as our starting point.

1. Add a class called `CSpapeShip` which inherits from `ISceneNode`.
2. Add the following private fields, `vertices` as an array of `S3DVertex` with the length of seven, `box` as a representation of a bounding from the type `aabbox3d<f32>`, and `material` as `SMaterial`.
3. Let's move on to the public section of this class. Add a constructor with the parameters `ISceneNode* parent`, `ISceneManager* mgr` and `s32 id` which inherits from the `ISceneNode` constructor `ISceneNode(parent, mgr, id)`.
4. Add `material.Wireframe = false;` and `material.Lighting = false;` in the constructor.
5. Now add a couple of vertices which will represent the points of the model spaceship:

```
vertices[0] = S3DVertex(0, 0, -20, 1, 1, 0, SColor(255, 0, 0,
255), 0, 1);
vertices[1] = S3DVertex(0, 0, -10, 1, 0, 0, SColor(255, 0, 0,
128), 1, 1);
vertices[2] = S3DVertex(20, 0, -10, 0, 1, 1, SColor(255, 0, 0,
64), 1, 0);
vertices[3] = S3DVertex(0, 20, 0, 0, 0, 1, SColor(255, 255, 0, 0),
0, 0);
vertices[4] = S3DVertex(0, 0, 10, 1, 1, 0, SColor(255, 0, 0, 255),
0, 1);
vertices[5] = S3DVertex(0, 0, 20, 1, 0, 0, SColor(255, 0, 0, 128),
1, 1);
vertices[6] = S3DVertex(20, 0, 10, 0, 1, 1, SColor(255, 0, 0, 64),
1, 0);
```

6. Move on to creating the bounding box by adding this chunk of code:

```
box.reset(vertices[0].Pos);
for (s32 i=1; i<7; ++i) box.addInternalPoint(vertices[i].Pos);
```

7. Add the method `virtual void OnRegisterSceneNode()` which has to be the same in every custom scene with this content:

```
if (IsVisible) SceneManager->registerNodeForRendering(this);
ISceneNode::OnRegisterSceneNode();
```

8. The next method to add is called virtual void render() and this content:

```
u16 indices[] = { 0,1,2,  1,4,3, 2,3,6,  1,3,2,  4,6,3,  4,5,6 };
video::IVideoDriver* driver = SceneManager->getVideoDriver();
driver->setMaterial(material);
```

```
driver->setTransform(video::ETS_WORLD, AbsoluteTransformation);
driver->drawVertexPrimitiveList(&vertices[0], 7, &indices[0], 7,
video::EVT_STANDARD, scene::EPT_TRIANGLES, video::EIT_16BIT);
```

9. Add the method `virtual const core::aabbox3d<f32>& getBoundingBox() const` which returns the private field `box`.

10. The next method `virtual u32 getMaterialCount() const` returns `1`.

11. The last method of this class `virtual video::SMaterial& getMaterial(u32 i)` returns the private field `material`.

12. Add a variable called `smgr` from the type `ISceneManager` which stores an instance of the scene manager via `device->getSceneManager()`.

13. Create an instance of `CSpaceShip` called `mySpaceShip` with the following contructor parameters, `smgr->getRootSceneNode()`, `smgr` and the id of `7`.

14. Create a rotation animator which rotates the node with the speed of `0.33f` around the Y axis.

15. Create a fly straight animator to let the spaceship fly from `0, 0, -30` to `10, 20, 50` in an interval of five seconds with the animation to be looped.

16. Add those two animations to the spaceship. Then drop those animations and set it to `0`.

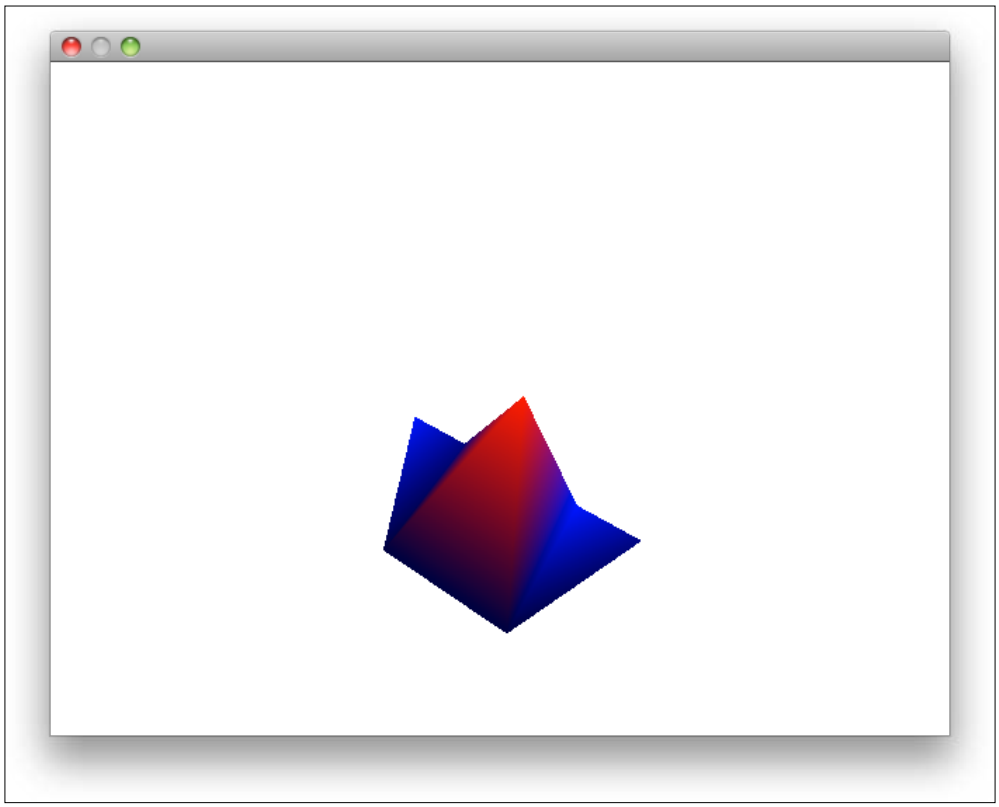
17. Then set the camera with this line of code, `smgr->addCameraSceneNode(0, vector3df(0, 70, -40), vector3df(0, 15, 0));`.

Your `CSpaceShip` class should look like this:

```
class CSpaceShip : public ISceneNode
{
  private:
    S3DVertex vertices[7];
    aaabbox3d<f32> box;
    SMaterial material;

  public:
    CSpaceShip(ISceneNode* parent, ISceneManager* mgr, s32 id) :
      ISceneNode(parent, mgr, id)
    {
      material.Wireframe = false;
      material.Lighting = false;

      vertices[0] = S3DVertex(0, 0, -20, 1, 1, 0,
        SColor(255, 0, 0, 255), 0, 1);
```

```
      vertices[1] = S3DVertex(0, 0, -10, 1, 0, 0,
        SColor(255, 0, 0, 128), 1, 1);
      vertices[2] = S3DVertex(20, 0, -10, 0, 1, 1,
        SColor(255, 0, 0, 64), 1, 0);
      vertices[3] = S3DVertex(0, 20, 0, 0, 0, 1,
        SColor(255, 255, 0, 0), 0, 0);
      vertices[4] = S3DVertex(0, 0, 10, 1, 1, 0,
        SColor(255, 0, 0, 255), 0, 1);
      vertices[5] = S3DVertex(0, 0, 20, 1, 0, 0,
        SColor(255, 0, 0, 128), 1, 1);
      vertices[6] = S3DVertex(20, 0, 10, 0, 1, 1,
        SColor(255, 0, 0, 64), 1, 0);

      box.reset(vertices[0].Pos);
      for (s32 i=1; i<7; ++i)
        box.addInternalPoint(vertices[i].Pos);
    }

  virtual void OnRegisterSceneNode()
  {
    if (IsVisible)
      SceneManager->registerNodeForRendering(this);

      ISceneNode::OnRegisterSceneNode();
  }

  virtual void render()
  {
    u16 indices[] = { 0,1,2,  1,4,3, 2,3,6,  1,3,2,  4,6,3,  4,5,6 };
    video::IVideoDriver* driver = SceneManager->getVideoDriver();

    driver->setMaterial(material);
    driver->setTransform(video::ETS_WORLD, AbsoluteTransformation);
    driver->drawVertexPrimitiveList(&vertices[0], 7, &indices[0], 7,
      video::EVT_STANDARD, scene::EPT_TRIANGLES, video::EIT_16BIT);
  }

  virtual const core::aabbox3d<f32>& getBoundingBox() const
  {
    return box;
  }

  virtual u32 getMaterialCount() const
  {
    return 1;
```

```
    }

    virtual video::SMaterial& getMaterial(u32 i)
    {
      return material;
    }
};
```



What just happened?

This example is more advanced than the examples before. Don't worry if you didn't understand everything that was covered. Let's take one step at a time.

Although it is not recommended, or a common practice, to keep everything in just one main source file, we will do that for the sake of this example. If you would want a more cleaner example you should copy the `CSpaceShip` class into a separate `.h` and `.cpp` file where the header file holds the class, its fields and method prototypes, and the `.cpp` file the implementation of those prototypes.

Even if your custom node wouldn't need a material or a bounding box you still need to implement the methods `OnRegisterSceneNode()`, `render()`, `getBoundingBox()`, `getMaterialCount()`, and `getMaterial()` or else they would link to pure virtual methods which would result in a compiler error.

The constructor sets two attributes to the material. First it sets the model to be rendered in a solid state and sets off the lighting. The next step is to set the vertices manually where the first three parameters set the position, the next three the normals, the parameter after that sets the color of the vertex and the last two parameters are the U and V texture coordinates for the material. Then we set the bounding box of the model by adding each vertex to the bounding box. It actually doesn't matter which `id` you assign to the node.

The next important thing is the `render()` method: We set a bunch of indices which determines the triangles we need to render our model. Then we set the material of our model and render the vertices.

Pop quiz

1. All the scene nodes added to a scene are rendered and visible on screen.
    a. True
    b. False
2. Irrlicht's animators can be used to play the animation of animated mesh scene node (IAnimatedMeshSceneNode) only.
    a. True
    b. False

Summary

We learned about Irrlicht's nodes and how to use node animators.

Specifically, we covered:

- Definition of nodes
- Working with nodes
- Using animators to animate nodes
- Creating a custom scene node

Now that we've learned about nodes and even how to animate those, let's move on to more eye-candy and interactive matters like how to use cameras in the next chapter.

8
Managing Cameras

In the previous chapter, we learnt about nodes and especially how to animate those nodes. Let's push a bit further and take a closer look at a more specialized node: the camera scene node.

In this chapter, we shall:

- Extend our basic template
- Add camera scene nodes
- Manipulate cameras
- Add prefabricated cameras to the scene

So let's get on with it...

What is a camera?

When one thinks of a camera, a sophisticated, technological camera used in expensive blockbuster movies probably comes to your mind. Although it is less of a high-end tool in the Irrlicht 3D engine, the concept remains the same. You need a camera in your scene if you want to see any objects of your scene graph. Remember that 2D objects, except for a few special cases (such as particle system), don't necessarily need a camera object. Aside from this, a camera has a certain position and looks at a certain point of the scene.

As already mentioned in the previous chapter, cameras are also nodes, so logically anything that can be applied to a node can be applied to a camera as well. For example, changing the position or rotation.

Extending our template application

Our template from *Chapter 2*, *Creating a Basic Template Application*, has been used for almost all of the examples of this book so far. In the previous chapters, we learned how to add a scene manager instance, which has been important since then as it contains our scene graph with all our 3D objects currently present in the scene.

Also, it is time to learn a few tricks on how to optimize our template application and some useful tips to improve the user experience of your application.

Time for action – extending our template application

As usual the base for this example is our template application from *Chapter 2*, *Creating a Basic Template Application*.

1. Add `scene` to your namespace, then add an instance of `ISceneManager` with the line `ISceneManager* smgr = device->getSceneManager();`, just after `IVideoDriver* driver = device->getVideoDriver();`.
2. Add `smgr->drawAll();` in our loop.
3. Directly in the "game loop" add the condition `if (device->isWindowActive())`, which results in the current code block we have in our loop and else calls `device->yield()`.
4. Add `gui` to your namespace, an instance of `IGUIEnvironment` named `guienv`, assign it with `device->getGUIEnvironment()` and add call `drawAll()` of this instance in your application loop. If you are not sure, do take a look at the examples from *Chapter 4*, *Overlays and User Interface*, where we created an event receiver for our GUI environment.
5. Now create your own event receiver class that inherits from `IEventReceiver` called `CAppReceiver`. This event receiver will receive keyboard events from our application.
6. Create an array of the bool type called `KeyDown` with the size of `KEY_KEY_CODES_COUNT`.
7. Jump to the constructor of `CAppReceiver` and reset the `KeyDown` array by filling its values with `false`.
8. In the `OnEvent` of `CAppReceiver()` method, add the following lines of code:

```
switch (event.EventType)
{
    case EET_KEY_INPUT_EVENT:
```

```
        {
            KeyDown[event.KeyInput.Key] = event.KeyInput.PressedDown;
            break;
        }
        default:
            break;
    }
```

9. Create a method corresponding to the `KeyDown` that is called `isKeyDown()` and has the parameter `keyCode` of the type `EKEY_CODE`. This method returns the value of the current key that has been pressed down with `return KeyDown[keyCode]`.

10. Now add the same method corresponding to `KeyUp`.

11. Add an instance of `CAppReceiver` called `appReceiver` right before you assign a device.

12. Change the `createDevice` call to receive events from `appReceiver`. **Tip**: Take a closer look at the last parameter of the `createDevice` method.

13. Within your application loop, add a condition to check whether the *ESCAPE* key has been pressed:

```
if (appReceiver.isKeyDown(KEY_ESCAPE))
{
    device->closeDevice();
    return 0;
}
```

14. Compile and run your new template application:

What just happened?

Granted the result of the new template that looks exactly like the old one. This new template will be used throughout the rest of this book. Here are the most important code changes in one place. Our event receiver class should look like this:

```
class CAppReceiver : public IEventReceiver
{
private:
    bool KeyDown[KEY_KEY_CODES_COUNT];

public:
    CAppReceiver()
    {
       for (int i = 0; i < KEY_KEY_CODES_COUNT; i++)
       {
           KeyDown[i] = false;
       }
    }

    virtual bool OnEvent(const SEvent& event)
    {
       switch (event.EventType)
       {
          case EET_KEY_INPUT_EVENT:
          {
              KeyDown[event.KeyInput.Key] =
                     event.KeyInput.PressedDown;
              break;
          }
          default:
             break;
       }
       return false;
    }

    virtual bool isKeyDown(EKEY_CODE keyCode) const
    {
        return KeyDown[keyCode];
    }

    virtual bool isKeyUp(EKEY_CODE keyCode) const
    {
        return !KeyDown[keyCode];
    }

};
```

We have two private arrays, one of which stores the keys that have been pressed down and the other stores the keys that are currently released. The constant `KEY_KEY_CODES_COUNT` stores the maximum number of keys that can be used.

In the `CAppReceiver()` constructor, we reset all values of `KeyDown`. In our implementation of the `OnEvent()` method, we check for key-input events and if a key has been pressed this key state will be stored in the `KeyDown` array. While this query didn't need to be handled with switch-case, but could have been simple with if-condition as well, the switch-case is used here so if we would need to check for mouse or GUI events, we could simply add another case-statement. Still, using switch-case or an if-condition is more of a programmer's choice than a dogma. If you like if-conditions better than switch-case, feel free to change these lines of code.

So, for example, if the *space* key has been pressed the following would happen inside the `OnEvent()` method:

```
KeyDown[KEY_SPACE] = true;
```

If the *space* key would have been released, the arrays would look like this:

```
KeyDown[KEY_SPACE] = false;
```

The point of the `isKeyDown()` and `isKeyUp()` methods is simply to return the current key states of a key code specified by the parameter.

The main entry point would look like this:

```
int main()
{
    CAppReceiver appReceiver;
    IrrlichtDevice* device = createDevice(EDT_OPENGL,
                            dimension2d<u32>(640, 480), 16, false,
                            false, false, &appReceiver);
    if (!device)
        return 1;
    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();
    IGUIEnvironment *guienv = device->getGUIEnvironment();
    while (device->run())
    {
       if (device->isWindowActive())
       {
          if (appReceiver.isKeyDown(KEY_ESCAPE))
          {
             device->closeDevice();
```

```
                return 0;
            }
            driver->beginScene(true, true, SColor(255, 255, 255, 255));
            smgr->drawAll();
            guienv->drawAll();
            driver->endScene();
         }
         else device->yield();
      }
      device->drop();
      return 0;
   }
```

You might have noticed that we created our `CAppReceiver` instance on the stack instead of using a pointer. Don't worry about that at the moment, in our simple application it would not make such a big difference if we would create a class on the stack or the heap. Our class on stack is not being deallocated and if we would care more about memory management, creating a class on the heap would have been our first choice.

The last parameter of the `createDevice()` method can hold a standard event receiver as it has done in this example. Alternatively, you could also set `device->setEventReceiver(NameOfTheEventReceiver);` where `NameOfTheEventReceiver` is a class implementing the interface `IEventReceiver`.

We have everything we would need to set up, the video driver, the scene manager, and even the GUI environment. We don't need the GUI environment in this chapter so, if you feel uncomfortable having a reference to this GUI environment, you can delete it without any regrets.

We extended our application loop quite a bit: if the application window is active, that means we are currently using this window and the application is not hidden in the background or minimized and the application loop is working as usual. If not, the application is working at its absolute minimum and is giving other processes currently running the processing power they need.

The next thing we added is a key-input check where we check if the *Escape* key has been pressed down and if that happened, this would close the application. We need this as a platform-independent way to close the application. If you add a prefab camera controller specifically the FPS camera to the scene, we could only be able to move the mouse within the window and we won't be able to close the application by clicking the **X** in the window bar. While we could still use *ALT* + *F4* under Windows, it is less straight-forward on unixoid systems.

Creating terrains

Before we even get a scene with a camera setup, we would need something in the scene. An empty scene is not very exciting and we wouldn't see any of the playing around with the camera. We already displayed meshes and scenes we created with CopperCube, so how about something we haven't tried before? How about terrains?

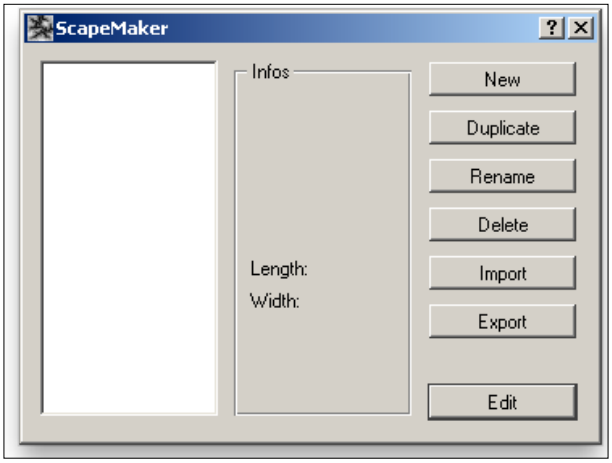
As mentioned in the previous chapter, to create a terrain we need a height map and a texture. The height map represents the geometrical data of the terrain where black pixels are the lowest points of the terrain, white pixels are the highest points of the terrain, and every shade of gray represents a point in space between the highest and lowest point.

There is a free application called ScapeMaker made by Dirk Plate and Matthias Buchetics that allows you to create a height map and textures with just a few steps. Unfortunately, ScapeMaker is only available for Windows platforms and has not been actively-developed for a few years now. Its official homepage is not in service any more, but there are several download portals featuring this application. For example the download is available at `http://www.brothersoft.com/scapemaker-24183.html`.

Time for action – creating terrains

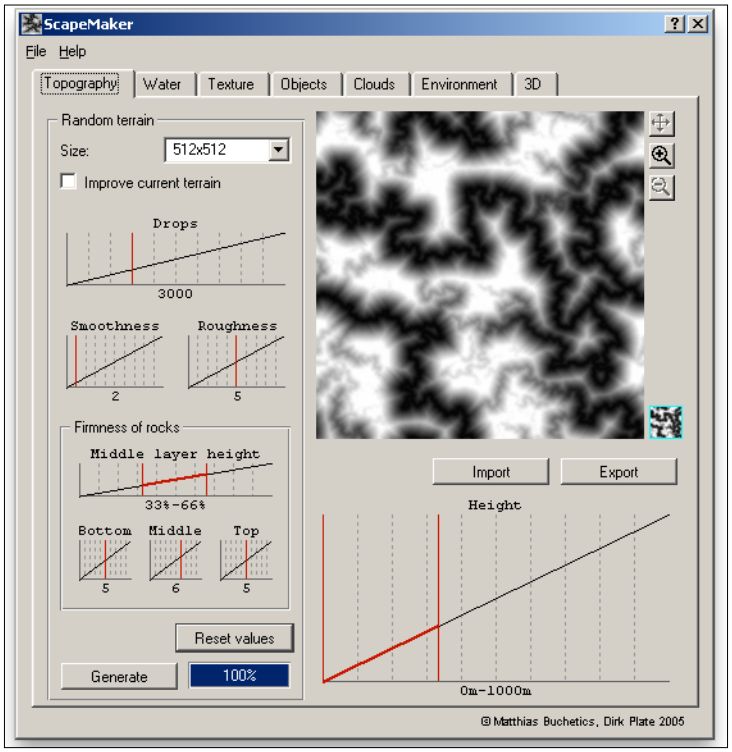
After ScapeMaker has been downloaded, double-click on the downloaded file to install the application.

1. Open ScapeMaker and click on **New**. Enter a name of your choice for your terrain project:



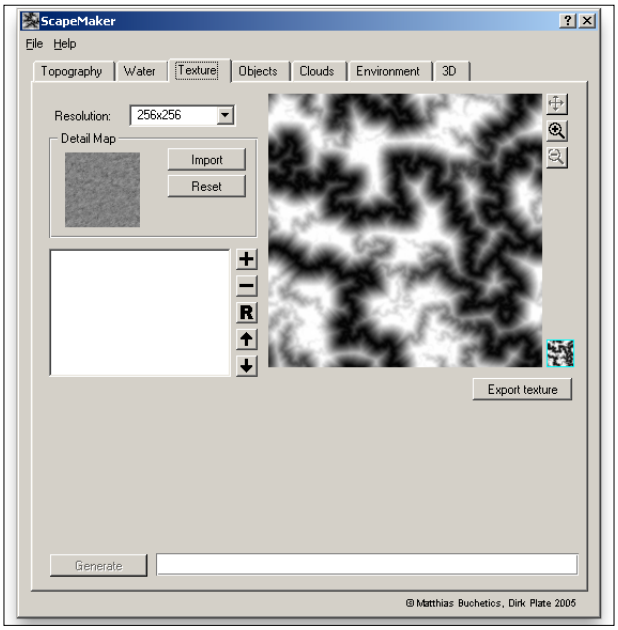
2. Select this terrain and click on **Edit**.

3. A new window will open. By clicking on **Generate** a new height map will be randomly-generated. You may want to adjust the values on the left side by clicking and dragging the sliders. This is not necessary though:



4. Select **Export** to save the previously-generated height map as a bitmap file.

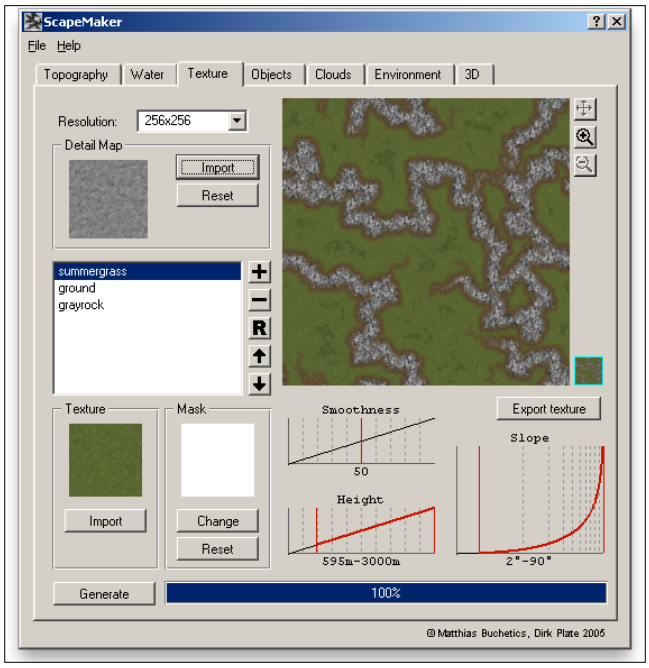
5. Switch to the **Texture** tab:



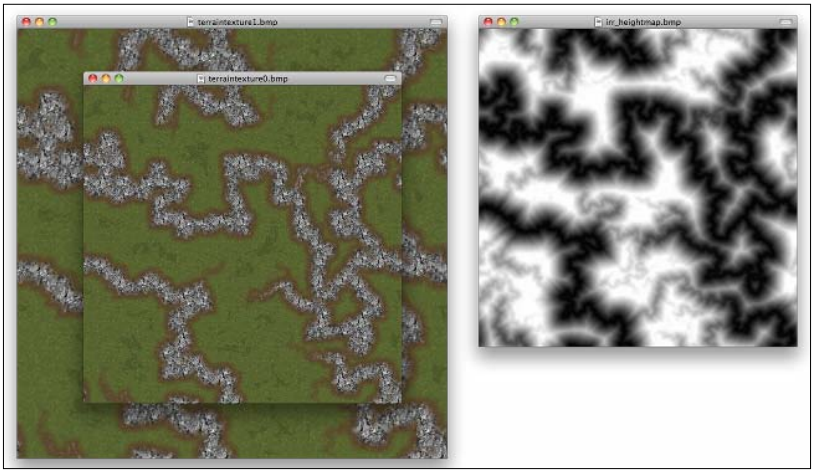
6. Click on the big plus sign to select a texture. ScapeMaker contains a bunch of textures to get you started such as grass, snow, and rock.

7. Select a few textures, for example, **grayrock**, **ground**, and **summergrass**. After you added those three textures, you have to adjust the attributes of those textures. Change the height and slope of the textures to create a smooth transition between the imported textures. The **grayrock** texture should be at the bottom, the **ground** texture in the middle, and the **summergrass** texture at the top.

Click on **Generate** to see if the corresponding texture to the terrain fits and there are no black spots left on the texture:



8. Click on the **Export texture** to save the newly-created texture. Create two textures, one with the resolution of 512 x 512 pixels, the other one with the resolution of 2048 x 2048 pixels. Don't worry if your images won't resemble the following image, because the terrain height map is always randomly-generated:



What just happened?

We just learnt the basics of how to create height maps and textures with ScapeMaker. You might wonder why we created two terrain textures. One reason is that on older computers with less processing or graphic power, we could simply load the small and less memory-consuming texture. Another reason is that Irrlicht allows us to apply more than one texture to the terrain. For example, if the camera would be far away from the terrain, the smaller, less detailed texture will be displayed while the more detailed texture will be drawn on the height map if the camera is closer to the terrain. Even if the textures with different resolutions are not available, Irrlicht will automatically try to reduce the texture details through the mipmap system. But still some systems will run better with the manually-provided smaller texture.

Adding a camera scene node

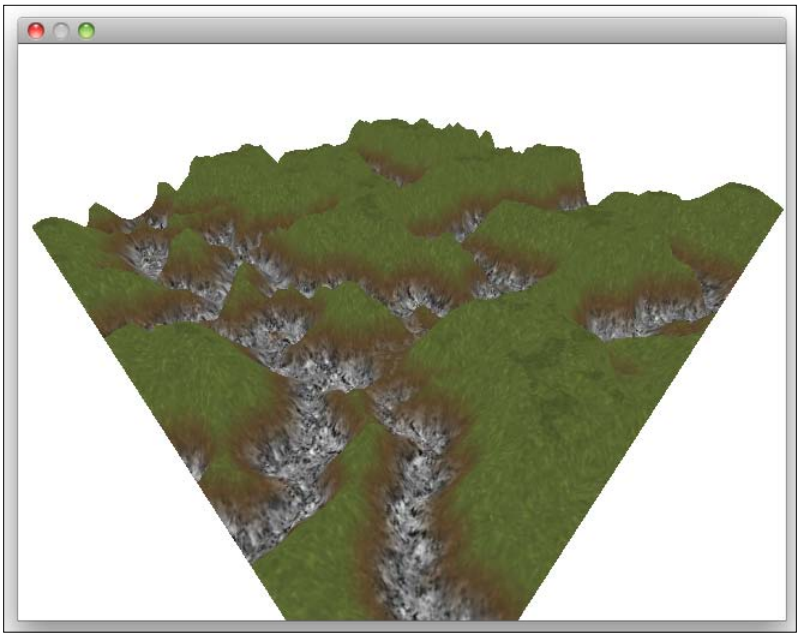
We already did this in every exercise where we displayed any 3D object. Let's now see what a camera scene node exactly did or how to configure this scene node.

Time for action – adding a camera scene node

Copy the terrain height map and the terrain texture files, to somewhere your application could find preferable, from the folder the executable will be run. We use our new template as the base for this exercise.

1. Define `terrain` of the type `ITerrainSceneNode` and initialize this with `smgr->addTerrainSceneNode("irr_heightmap.bmp");`. `irr_heightmap.bmp` is the name of the height map, we exported through ScapeMaker.
2. Call `terrain->setScale(vector3df(1.0f, 0.25f, 1.0f));`.
3. Set the material flag `EMF_LIGHTING` from `terrain` to `false`.
4. Apply the terrain texture to the height map with the line `terrain->setMaterialTexture(0, driver->getTexture("terraintexture0.bmp"));`. Feel free to apply the texture that you see fit.

5. Add a camera with `ICameraSceneNode* camera = smgr->addCameraSceneNode(0, vector3df(0, 200, 0), vector3df(100, 80, 100));`:



What just happened?

In step 1, we added a terrain scene node to the scene. The only parameter `addTerrainSceneNode()` always needs is the filename to be used as the heightmap. The method has ten additional optional parameters that are as follows:

- Parent node (root node if nothing is specified).
- Node ID of this node as `s32`.
- The position of this node.
- The rotation of this node.
- The scale of this terrain node. For example, if you apply a zero factor to terrain's Y scale, your terrain will look flat.
- The color of the terrain as `SColor`.
- Maximum level of detail and patch size of the node terrain. These two parameters are depending on each other and how to work them out is a little bit advanced. You can refer to original comments of `addTerrainSceneNode()` method from `ISceneManager.h` inside the Irrlicht source folder.

- Smoothness of the terrain; the higher the smoothness value, the more the terrain levels are blended together.

In step 2, we adjust the height of the terrain by scaling the Y value down. This is necessary as the mountains of the terrains would look too stretched and not realistic at all.

Step 3 and 4 shouldn't be too much of a challenge as we did something very similar when we applied textures to meshes.

In step 5, we define ourselves an instance of a camera scene node with those parameters:

- Parent scene node.
- Camera position.
- Camera target; the point where the camera looks at.
- Node ID.
- Flag that determines if the camera is active or not. This is set to `true` by default.

All of these parameters are optional.

Have a go hero – rotate and move the camera

Having the camera just standing there might be a little dull. Let's do something a bit more exciting. Why not try to rotate and move the camera around? Your goal is to implement the following:

- Move the camera forward by pressing the *W* key
- Move backwards with the *S* key
- Turn the camera left with the *A* key
- Turn the camera right by pressing the *D* key

You figured it out or are not sure what to do? There are a lot of ways to achieve what we want here. These code snippets are only one possible solution:

```
vector3df cam_pos = vector3df(0, 200, 0);
vector3df cam_target = vector3df(100, 80, 100);

ICameraSceneNode* camera = smgr->addCameraSceneNode(0, cam_pos,
                                                    cam_target);
```

Now add the key events inside the loop and call `camera->setPosition` and `camera->setTarget` with the modified values:

```
if (appReceiver.isKeyDown(KEY_KEY_W))
{
    cam_pos.X += 0.5f;
    cam_target.X += 0.5f;
    cam_pos.Z += 0.5f;
    cam_target.Z += 0.5f;
}
if (appReceiver.isKeyDown(KEY_KEY_S))
{
    cam_pos.X -= 0.5f;
    cam_target.X -= 0.5f;
    cam_pos.Z -= 0.5f;
    cam_target.Z -= 0.5f;
}
if (appReceiver.isKeyDown(KEY_KEY_A))
{
    cam_target.X -= 0.5f;
}
if (appReceiver.isKeyDown(KEY_KEY_D))
{
    cam_target.X += 0.5f;
}
camera->setPosition(cam_pos);
camera->setTarget(cam_target);
```

This solution just changes the camera target of the camera, so the camera does not really rotate, but has a smooth shifting animation instead. Also, the *W* key does not always move the camera forward, if the camera is rotated to the side, pressing *W* moves the camera to the side as well. Note that for the sake of simplicity, we just added up constant speed to the camera position, which is `0.5` in the preceding example that can give different results on different PCs. You can refer to *Chapter 5*, *Understanding Data Types* to learn more about the frame rate for independent movement of objects:

If you would want a true rotation, you would have to call `camera->bindTargetAndRotation(true)` that causes the camera rotation to be updated to the camera target and vice versa. If the target and camera are bound this way, rotating the camera with `camera->setRotation()` method will also update the target's position so that it's in line with the camera's +Z axis.

Adding prefabricated cameras

We know how to add `ISceneNode` instances to our scene and how we can manipulate those. Irrlicht also offers two prefabricated cameras that we are going to use now.

Time for action – adding prefabricated cameras

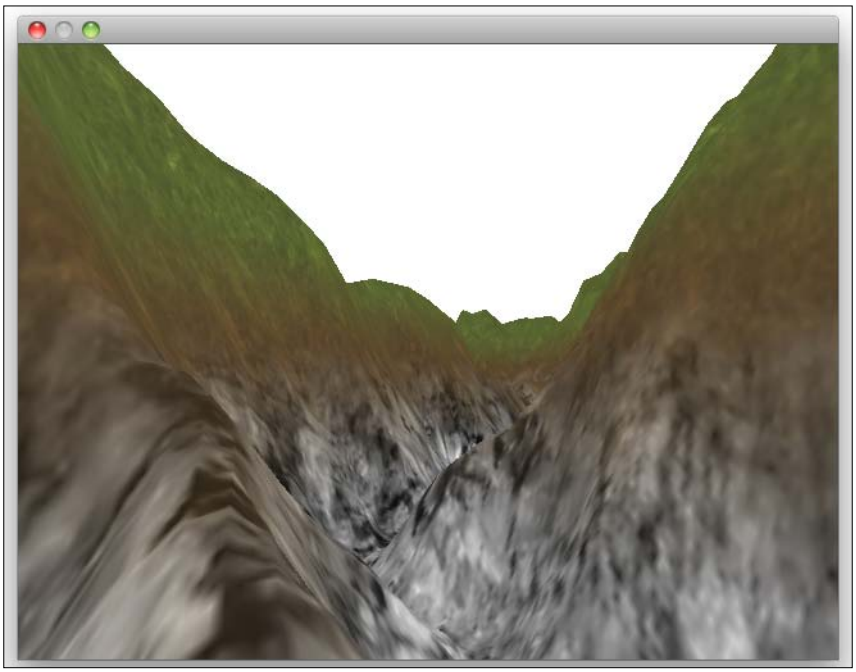
Let's continue right where we left off in the last example. Our goal in this exercise is to have two prefab cameras in the scene that we switch through with the *Space* key. Just delete the camera instance and let's get started.

1. Add a bool variable called `cam_switch` and set it to `false`.

2. Add two scene nodes with one being initialized by `smgr->addCameraSceneNodeFPS()` and the other set by `smgr->addCameraSceneNodeMay().a`. The first variable should be named `fps_camera` and the second `maya_camera`.

3. Check if the *Space* key has been pressed and if that's the case set `cam_switch` to `true`.

4. Now, check if `cam_switch` is `true` to see whether the *Space* key is pressed to switch camera. If it's `true`, check that the active camera is using `device->getActiveCamera()` method. And finally, switch to the camera that is currently not active with `device->setActiveCamera()`. If the FPS camera is active, hide the mouse cursor through the method `device->getCursorControl()->setVisible(false)` or else show the mouse cursor again:



What just happened?

If you play video games, you probably already know what the FPS (First Person Shooter) camera is. The camera looks from the angle of a person walking around in the scene, very similar to how it is done in games like Quake, Doom, or Bioshock.

In Irrlicht, an FPS camera is controlled with the mouse to look around in the scene and the arrow keys to move and it has the following parameters:

- Parent node.
- Rotation speed of the camera.
- Movement speed of the camera.
- Camera node ID.
- Key map array and key map size that allows you to modify which keys are used to move around in the scene.
- Flag to on/off vertical movement.
- Jump speed, which is set to `zero` by default.
- Invert mouse parameter, which if set to `true` makes the camera look up when the mouse is moved down and down when the mouse is moved up. By default, it's set to `false`.
- The last parameter sets if the camera is active.

The Maya camera is named after the graphical modeling application Maya by Autodesk and is controlled in Irrlicht in the same way as in Maya: click and drag with the left mouse button to look around, zoom with the middle mouse button, and dragging the right mouse button lets the camera move.

The method `addCameraSceneNodeMaya()` can be called with the following parameters:

- Parent node
- Rotate speed of the camera
- Zoom speed of the camera
- Translational speed of the camera
- Node ID
- The last parameter sets if the camera is active

Like the `addCameraSceneNode()` method, all these parameters are optional. The methods can be called blank and will then be filled with default values.

When you first start the application, the Maya camera will be active and pressing *Space* switches between the Maya and FPS camera. If you are not sure whether you followed the steps correctly take a look at this code snippet that shows the changes since the last exercise.

```
int main()
{
    [...]

    bool cam_switch = false;
```

```
    ICameraSceneNode* fps_camera = smgr->addCameraSceneNodeFPS();
    ICameraSceneNode* maya_camera = smgr->addCameraSceneNodeMaya();

    while (device->run())
    {
        if (device->isWindowActive())
        {
           if (appReceiver.isKeyDown(KEY_ESCAPE))
           {
              device->closeDevice();
              return 0;
           }

          if (appReceiver.isKeyDown(KEY_SPACE))
          {
             cam_switch = true;
          }

          if (cam_switch)
          {
              if (smgr->getActiveCamera() == maya_camera)
              {
                 smgr->setActiveCamera(fps_camera);
                 device->getCursorControl()->setVisible(false);
              }
              else
              {
                 smgr->setActiveCamera(maya_camera);
                 device->getCursorControl()->setVisible(true);
              }

              cam_switch = false;
          }

          [...]
       }
       else device->yield();
    }

    device->drop();

    return 0;
}
```

Summary

We learned about how to manage cameras with Irrlicht and how to extend the template to register keyboard input events.

Specifically, we covered:

- Definition of cameras
- How to create terrains with ScapeMaker
- Display the terrain with an applied texture
- Manipulate a camera scene node
- Adding prefabricated cameras to the scene

Now that we've learned about cameras, let's go one step further and add lights to our scenes to create a nicer atmosphere, which is also the title of the next chapter: Brightening up scenes with lights.

9
Lightening the Scene

Let there be light.

Genesis 1:3

Lighting plays an essential role in creating a visually appealing game. So far we've discussed about many aspects of Irrlicht engine. But even if we put all those elements together in a game, it still won't look right. If you've been playing 3D games like Prototype, X-Men: Wolverine, you'll notice something is obviously missing. Three things that mainly contribute to the look and feel of a 3D application are lightings, special effects, and shader programs. And these are exactly the things we are going to examine throughout the course of this chapter and the next three. We'll start with lighting.

In this chapter, you'll learn:

- Lighting concepts
- Adding new lights
- Manipulating lights
- Shadows
- Materials
- irrEdit 3D scene editor

Using irrEdit to set up lights

Before we jump into coding, let's start with a visual 3D editor to see how different light settings give different results. We'll use an editor called **irrEdit**. irrEdit is a real-time 3D world editor that can be used to create 3D levels easily. We can save the scenes created with irrEdit in `.irr` format, which is the native format of Irrlicht engine for scenes. `.irr` format is simply an XML file that can be opened with any text editor. It stores all the elements from the whole scene including animators, materials, and particle systems. The rendering engine of this irrEdit editor itself is written in Irrilcht and so it can load all the file formats that the Irrlicht engine supports. The scene manager (`ISceneManager`) interface in Irrlicht engine provides a method called `loadScene()` to load `.irr` files. And irrEdit is free to use as well. To learn more about this editor and other tools provided and supported by Irrlicht, go to **Tool set** page of Irrlicht engine on SourceForge, `http://irrlicht.sourceforge.net/toolset.html`.

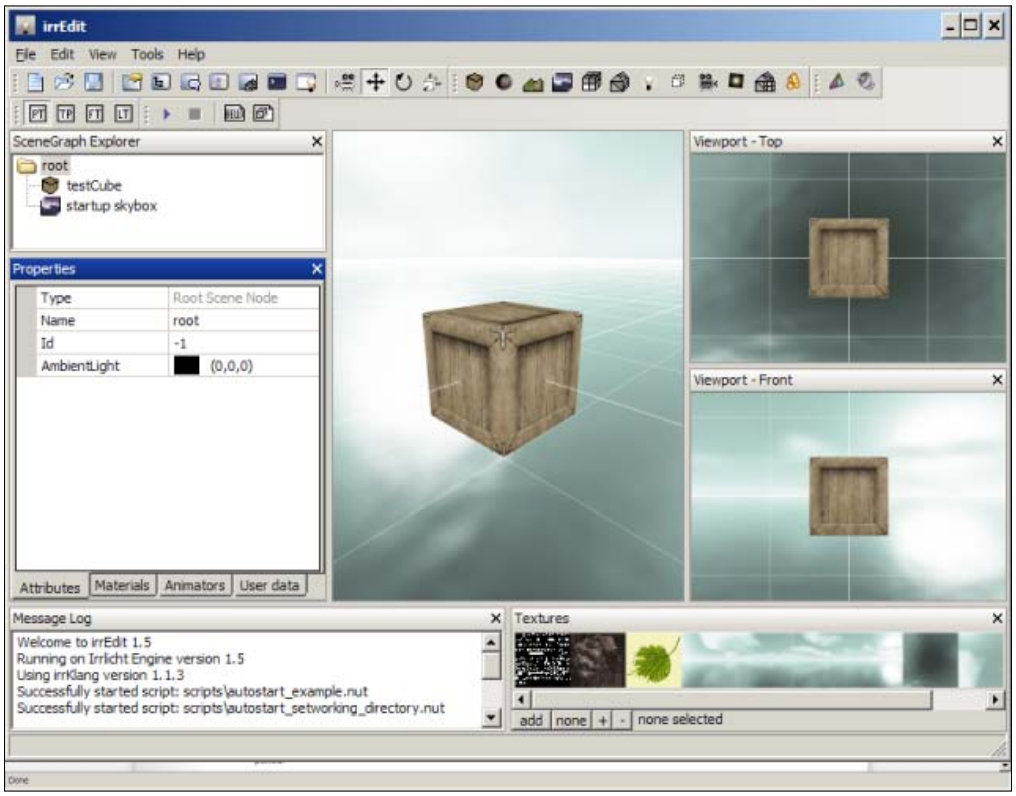
We are not going to use irrEdit to create complex scenes such as complete game levels in this section. But instead, we'll use this editor to create and manipulate different types of light node supported by Irrlicht. That way we'll be able to visually see what the different properties of a light node means and which property to change to achieve the result we want. Then we'll go back to coding and write Irrlicht programs to get the same kind of result we created in irrEdit.

irrEdit is hosted and maintained by Ambiera e.U., an independent game development company run by Nikolaus Gebhardt (niko), who started this Irrlicht engine project. So, go to `http://www.ambiera.com/irredit/index.html`.

When you're at irrEdit page on Ambiera website, you'll see a list of products in the **Downloads** section including **Coppercube-irrEdit** and **irrEdit** itself, making the choice of which one to use a bit confusing. This is because Ambiera also has a technology called CopperCube, which is a 3D rendering engine that can publish 3D scenes for web platforms such as Flash, HTML5 WebGL, and as a standalone Windows application as well. CopperCube is a commercial product and selling at 99€ now with a free trial period of 14 days. After irrEdit version 1.5, the new releases also support CopperCube features to publish 3D scenes for the web. But there is no expiry period if you want to use it only to make scenes for Irrlicht and export as `.irr` files. So it is still safe to use the latest version if you want to. But we will go with irrEdit version 1.5, so download irrEdit and extract the archive to your computer and run `irrEdit.exe`.

The reason we have chosen this, is because we'll finally go back to coding and implement ourselves again what the editor is doing behind the scene. Most of the properties of Irrlicht engine are exposed like the actual variable names from the engine in the irrEdit version:

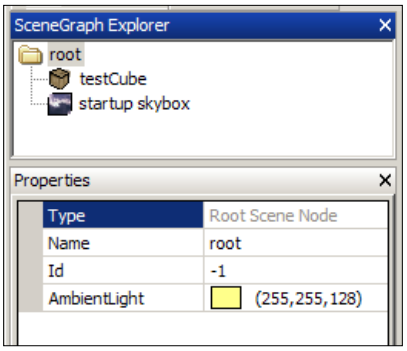
A new scene has a skybox and a cube mesh by default. On the left side, you'll see the **SceneGraph Explorer** and **Properties** panels. **SceneGraph Explorer** displays all the nodes (objects) that exist in the current scene and **Properties** panel displays the attributes, materials, and so on of a selected node. This is what we'll see when we first run the editor:



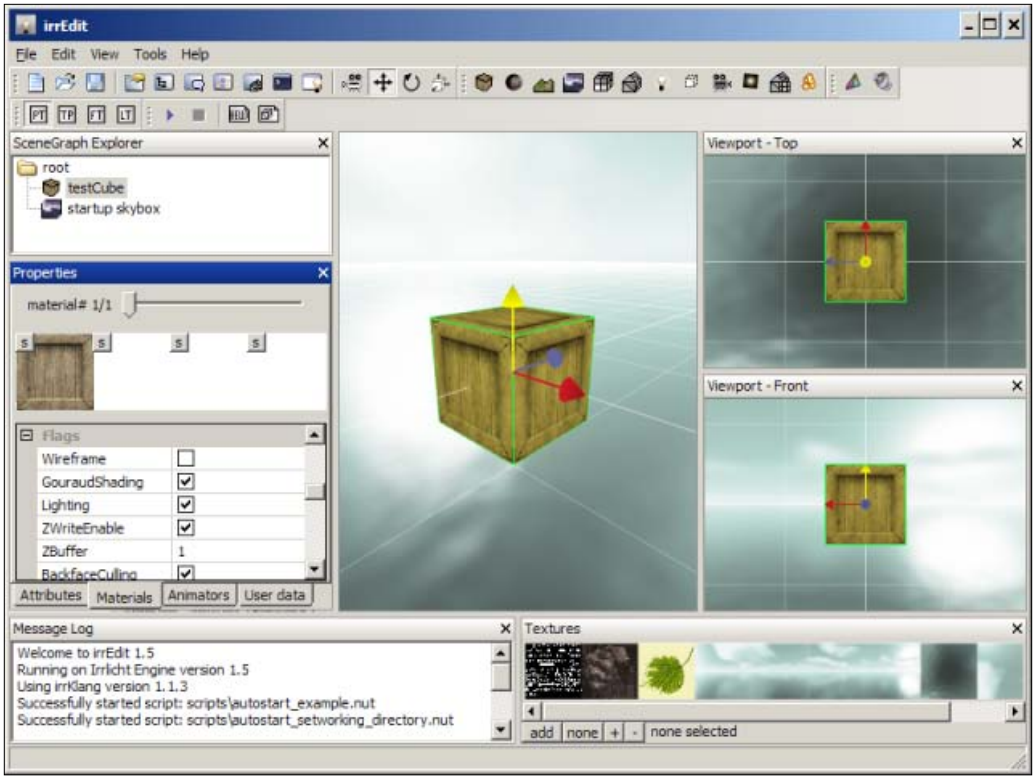
Time for action – creating a global ambient light

We are now going to set the light color for a global ambient light that will affect all the objects in the scene.

1. Choose **File | New** to create a new scene in irrEdit.
2. Select the topmost level node, which is a scene node, called **root** in our case.
3. You'll see an attribute called **AmbientLight** in the **Properties** panel. Change this color to something else other than default black, let's say yellow:



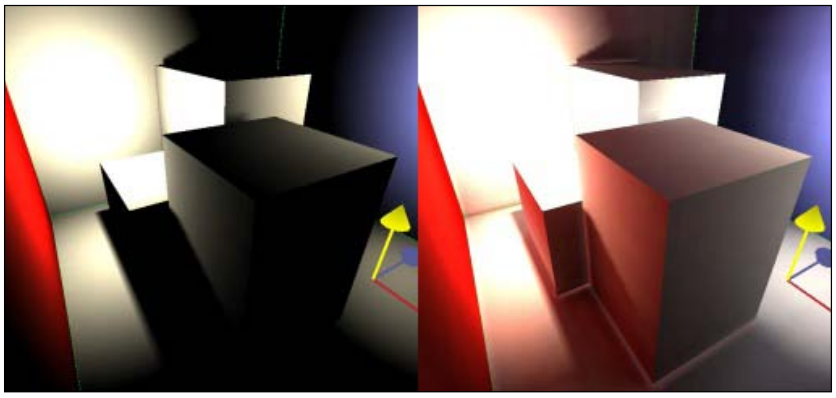
4. Select the **testCube** node from the explorer and the choose **Materials** tab in the **Properties** panel. By default, **Lighting** property is unchecked. Check this checkbox and you'll see the result immediately as shown in the following figure:



We just set the global ambient light for our `root` scene node. Let's explore a bit on ambient light and materials.

Ambient light

Take a look at the objects in your room right now. You can see them even though there's no direct light falling upon them. Because, those objects getting the direct lighting produce reflections that hit the other objects and they reflect again and so on and so forth. This is something known as **global illumination** in computer graphics. Simulating global illumination requires a fair amount of complex calculations and thus, it's an expensive process (at least for the older generation of computers) to use in such applications as games where the fast real-time rendering is highly preferred. The following image is from `http://www.irrlicht3d.org/pivot/entry.php?id=396` that demonstrates the global illumination feature using irrEdit. This is a scene rendered using direct lighting (left) and global illumination (right). As you can see, the image rendered with the global illumination is more realistic. Notice the color of the wall reflected back to the boxes and made them visible with some reddish color, which was not visible in the left image. This is where ambient color kicks in. If we have set up an ambient color for this scene these sides would be visible cheaply, though they won't be that realistic:



So people came up with the idea of ambient light, which is basically an estimated color value to use for the parts of an object with no direct light, so that they won't be completely dark.

One thing to note in Irrlicht is scene level ambient light. The ambient light we specified for our root scene node is the overall light color for the whole scene. It'll affect all the objects regardless of whether they receive direct lighting or not.

But we still can't see any changes with the color of our little cube as soon as we changed the ambient light of `root` scene node. The reason why our cube didn't respond to the ambient light setting is that its material is not set up to do so.

Materials

3D objects' materials play an important role in setting up lights to get a nice looking scene. To enable dynamic lights for our cube, we need to edit the light setting of the cube's material. So when we are writing code, we've to make sure to enable lighting for the material of the object to receive the light.

We can also set the ambient, diffuse, and specular values to our materials. These values mean how much of the incoming light should be reflected. By default, materials in Irrlicht engine are set to fully reflect the ambient light by setting opaque white, RGB (255, 255, 255). That means it'll fully reflect to all 3 color channels (red, green, blue) of incoming light. For example, if we have a global ambient light color red, RGB (255, 0, 0) and the material is set to fully reflect the ambient lighting with the value RGB (255, 255, 255), we'll see the object using that material as red (since we're talking about ambient, it is only for those parts of the object receiving no direct lighting). We can specify how much to reflect for each individual channel of incoming light. For example, if we want to fully reflect only the blue channel of the incoming light, we'll need to set our material ambient to RGB (0, 0, 255). However, since we set the red channel to 0 of our material, we won't see any reflection if the incoming light is any color other than blue. The same idea applies to diffuse and specular components as well.

Emissive color

There is another color property for materials called emissive color. Emissive color is used when we want an object to emit its own light. If there are other lights affecting this object, emissive light color will contribute to the final color, together with all the other lighting effects. If there is no light in the environment affecting it, emissive color works as a glowing color of the object itself. An example use of an emissive color could be for a lamp post to give a glowing light color, even in a scene without any light. But do take note that having an emissive color of an object doesn't work as a light source. To give real lighting that can affect other nodes, we'll need to add and position a real light node at the lamp post as well.

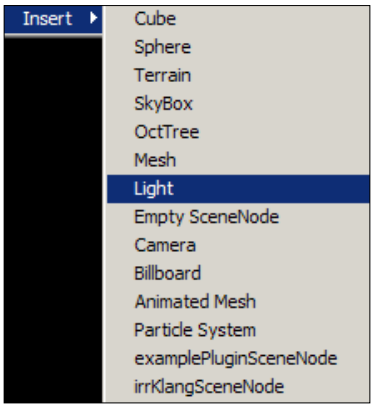
Time for action – adding a light scene node

Now, we've a scene with a cool global ambient light. Next, we'll add a light node to our scene.

1. Create a new scene.
2. Select the **Cube** node from **SceneGraph Explorer**. Select the **Materials** tab and enable lighting in the cube's material properties.

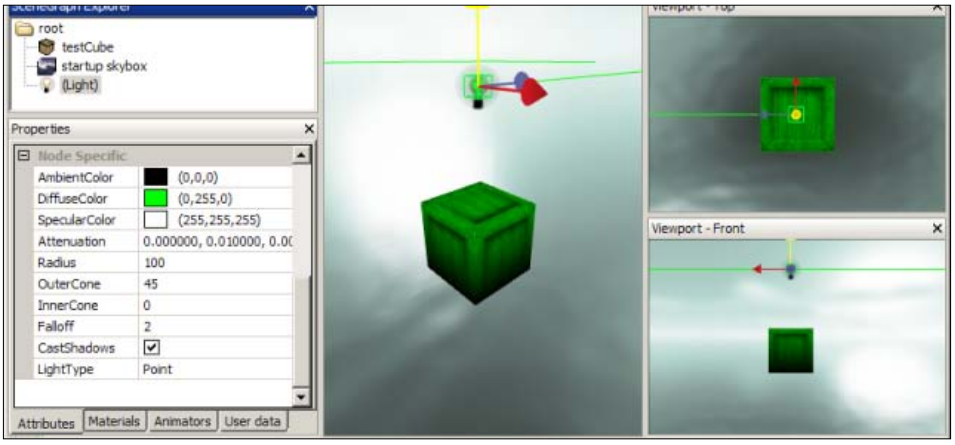
3. And now, let's add a new light node into our scene either by clicking the light bulb icon in the tool bar: or by choosing **Insert | Light** from the right-click on the context menu:

Or you can also choose **Insert | Light** from the right-click on the context menu:



4. Use the transformation gizmos to adjust the position of our light node. You can see all the properties of a light node in the **Properties** panel.

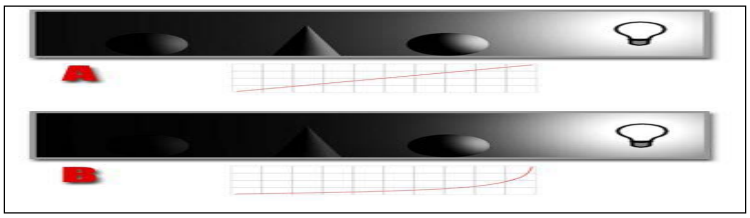
5. Let's set the **DiffuseColor** value to green. The following figure shows the lighting effect to our cube object:



Diffuse color is the main color value that will affect the matte surface of the object. There's one more interesting property of lights called attenuation.

Attenuation

Naturally lights tend to attenuate over the distance travelled. Attenuation determines how much of the light intensity should decrease with the distance from the objects. Irrlicht supports three factors of attenuation: constant, linear, and quadratic. In Irrlicht, attenuation of a light is set, by default, to these values: constant and quadratic member values to `0.0` and linear member to `1.0/light's radius (range)`, meaning that the light will be brightest at the source and will linearly decrease to `1.0/light's radius (range)` at the edge of the light's range:



Time for action – setting up Irrlicht

Let's go back to coding and learn how we can create and manipulate lights using Irrlicht engine. First, we'll create a simple application that we'll later expand to add lights, effects, shadows, and so on.

```
#include <irrlicht.h>

using namespace irr;
using namespace core;
using namespace scene;
using namespace video;
using namespace io;
using namespace gui;

#ifdef _IRR_WINDOWS_
#pragma comment(lib, "Irrlicht.lib")
#pragma comment(linker, "/subsystem:windows /ENTRY:mainCRTStartup")
#endif

int main()
{
```

```
IrrlichtDevice *device = createDevice(video::EDT_DIRECT3D9,
                         dimension2d<u32>(640, 480), 32, false, true,
                         false, 0);

    if (!device)
        return 1;

    device->setWindowCaption(L"Chap9 - Lighting Demo - 1");

    IVideoDriver* driver = device->getVideoDriver();
    ISceneManager* smgr = device->getSceneManager();

    //add a cube scene node
    IMeshSceneNode* myCube = smgr->addCubeSceneNode();
    if (!myCube)
        return 1;

    if (myCube)
    {
       myCube->setMaterialFlag(EMF_LIGHTING, false);
       myCube->setMaterialTexture( 0, driver-
                                >getTexture("../../media/wall.bmp"));
    }

    smgr->addCameraSceneNode(0, vector3df(0, 30, -40),
                             vector3df(0,5,0));

    while(device->run())
    {
       driver->beginScene(true, true, SColor(255, 100, 101, 140));
       smgr->drawAll();
        driver->endScene();
    }

    device->drop();
    return 0;
}
```

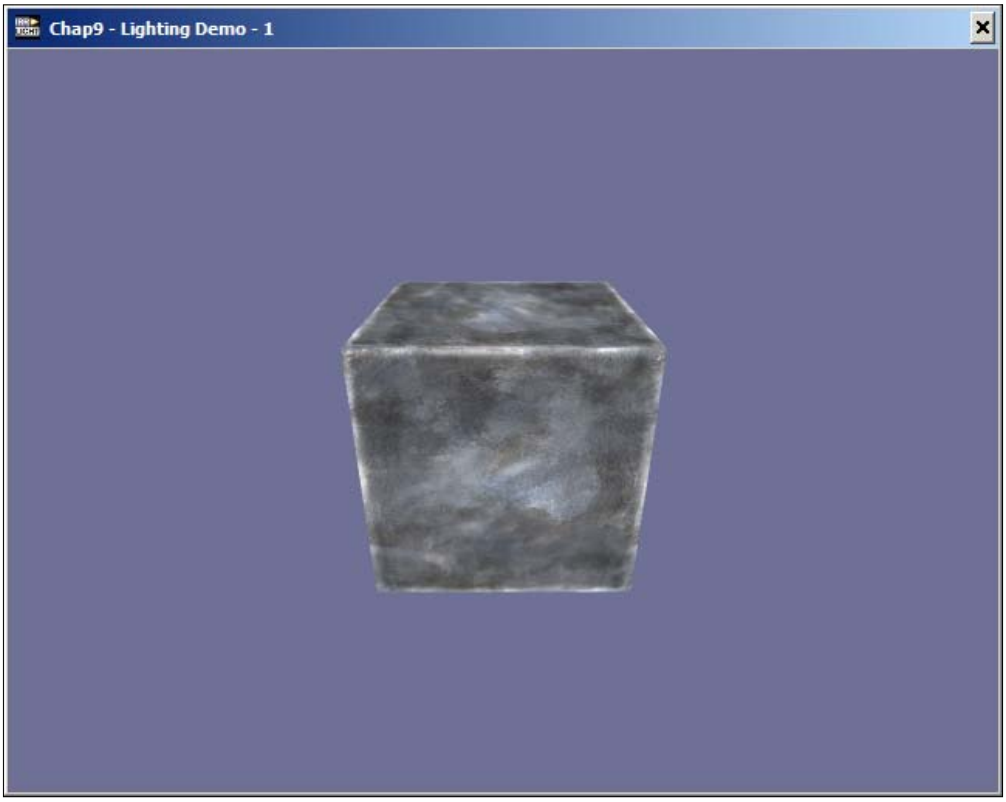
What just happened?

The preceding code listing doesn't do anything fancy yet. We create an Irrlicht device, add a cube scene node and set up texture, add a camera node and that's it. You should be able to run this code successfully (if your IDE is set up properly) and this is what you will see once it's run. (Note: It is assumed that the project folder should be in `irrlicht-x.x\examples\`. Otherwise you may not find the path for this resource `media/wall.bmp`).

Take a look at following line in the preceding code snippet:

```
myCube->setMaterialFlag(EMF_LIGHTING, false);
```

We are setting `EMF_LIGHTING` property to `false`, which means this object doesn't care about any light in the scene. So even though there's no light source in the scene, we can still see our cube:

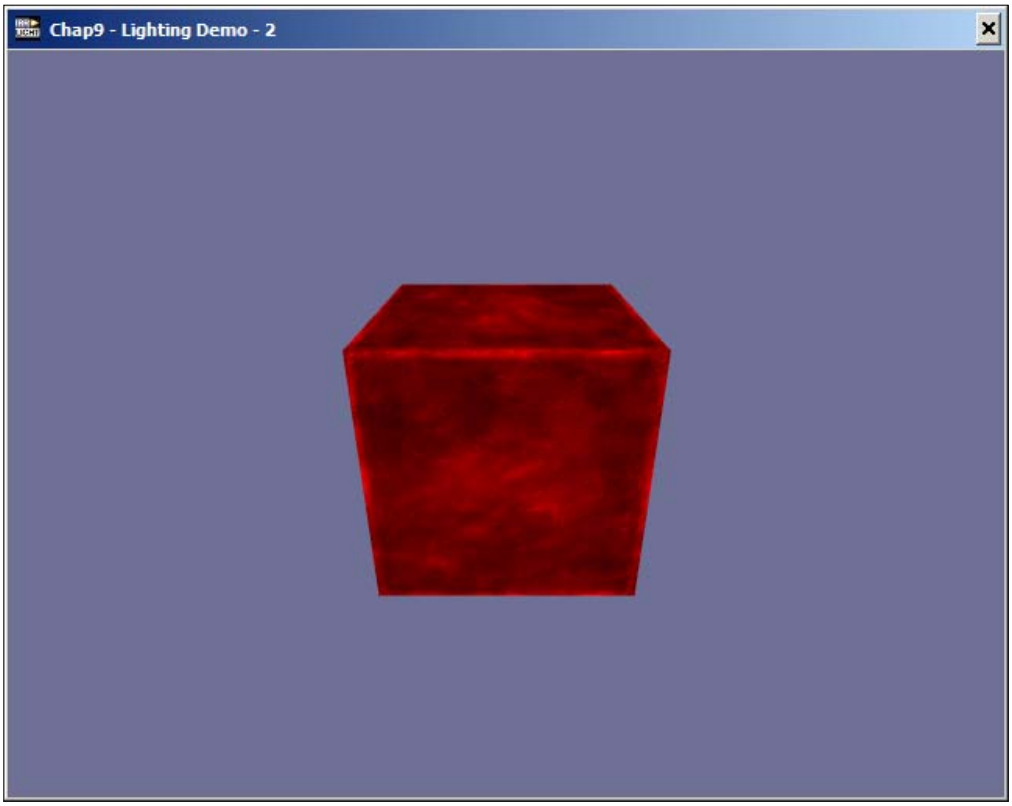


Time for action – setting up global ambient light

Now let's change the preceding code so that our cube node reflects to the ambient light.

1. Set that value to `true`. `myCube | setMaterialFlag(EMF_LIGHTING, true);`.
2. Set the global ambient light color to, let's say, red. `smgr->setAmbientLight(SColor(255, 255, 0, 0));`.

You should see the cube affected by the red ambient light as shown in the following figure:



Have a go hero – a dark scene?

Remove the ambient light from the scene while the `EMF_LIGHTING` property of the cube is still set to `true`. You'll notice that the cube will now be completely dark, because there's no light source to reflect in the scene. Forgetting to put some lights in the scene will cause your scene to be completely dark if the objects' materials are set to reflect incoming lights.

Time for action – creating a custom light node

We'll now add a light node with our own diffuse color. The new updated code is shown in the following code snippet:

```
smgr->addCameraSceneNode(0, vector3df(0, 10, -20), vector3df(0, 0,
                                                             0));
smgr->setAmbientLight(SColor(255, 80, 80, 80));
```

```
ILightSceneNode* myLight = smgr->addLightSceneNode();
myLight->setPosition(vector3df(0.0f, 20.0f, 0.0f));

SLight lightData;
lightData.DiffuseColor = SColor(255, 0, 255, 0);

myLight->setLightData(lightData);

vector3df cubeRotation = vector3df(0.0f, 0.0f, 0.0f);

while(device->run())
{
    myCube->setRotation(cubeRotation);
    cubeRotation += 0.02f;
    if (cubeRotation.X >= 360.0f)
    {
        cubeRotation.X = 0.0f;
        cubeRotation.Y = 0.0f;
        cubeRotation.Z = 0.0f;
    }
}
```

What just happened?

We added a new light node to our scene by calling `addLightSceneNode()` method of scene manager object. Data for the light can be stored in a structure called `SLight`. So, we create a new instance for `SLight` structure and set our desired diffuse color. And finally, we assign this `SLight` instance to our light node by calling `setLightData()` method. The rest of the code changes are just to make our cube rotate around and so a bit more interesting.

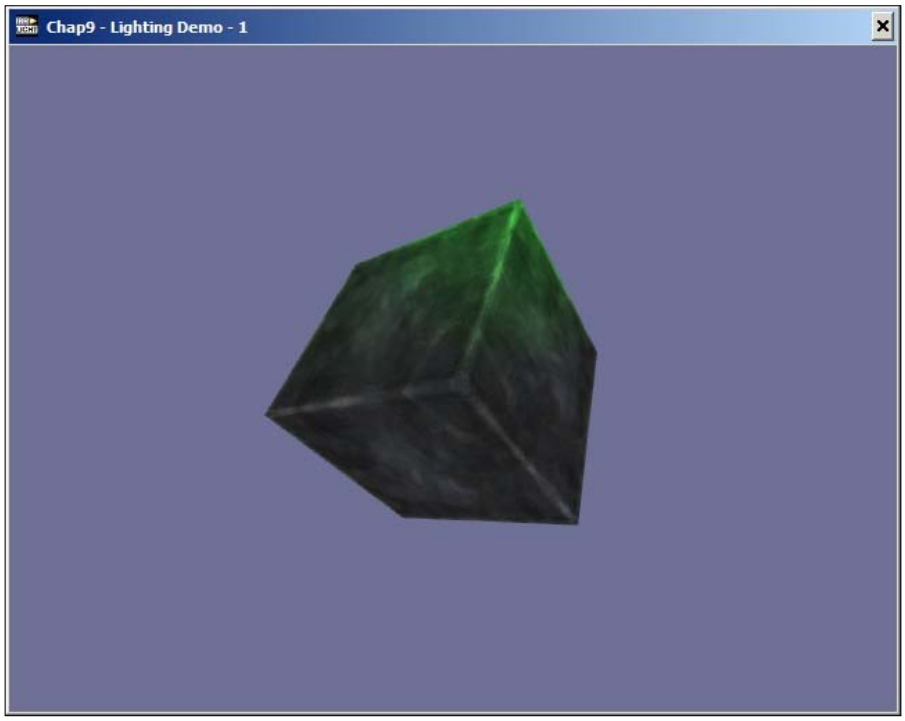
There are basically three types of light in computer graphics: directional light, spot light, and point light. Irrlicht supports all of them. Let's see a little bit more about point light and directional light.

Directional light

Directional light produces parallel light rays in a particular direction. The position is not valid for directional lights, but the rotation is. We can think of the sun as an example of directional light. Though the sun has a position, it's quite far away from us, such that we can assume our directional light at an infinity position. Like we've only one sun in reality, we don't usually need many directional lights in a game level. We only need a few (most of the time, one or two) directional lights for a scene, and directional lights are assumed to be not affected by attenuation. We'll assume the intensity of directional lights will not decrease over the distance travelled. Taking the sun as an example again, directional light can be used to create in-game day and night cycle. We can achieve this by rotating our directional light accordingly with our in-game sun's position and using different sky box images for day and night.

Point light

A point light produces light in all directions like a light bulb. Since it produces light in all directions, the rotation and direction values are not important for a point light source. Unlike directional lights, point lights are affected by attenuation and range values. This is the default type of light in Irrlicht when we create a new light node as well:



Time for action – adding a spot light and an animator

We'll now add a rotating spot light and also a fly-circle animation to our point light. Take a look at the following code and make modifications in your project accordingly:

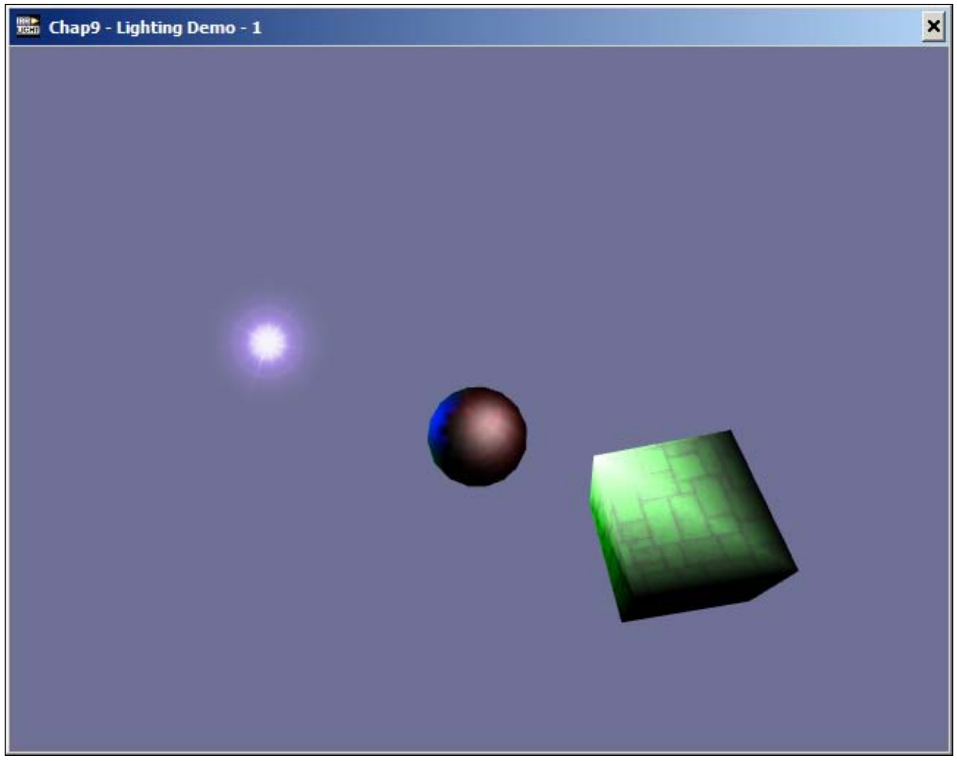
```
ICameraSceneNode* fpsCamera = smgr->addCameraSceneNodeFPS(0, 80.0f,
                                                          0.3f);
fpsCamera->setPosition(vector3df(0, 15, -25));
fpsCamera->setTarget(vector3df(0, 0, 0));

ILightSceneNode* myPointLight = smgr->addLightSceneNode();
myPointLight->setPosition(vector3df(0.0f, 20.0f, 0.0f));

scene::ISceneNodeAnimator* anim = smgr-
                                  >createFlyCircleAnimator(vector3df(0,
                                  0, 0), 50.0f, 0.0005f);
myPointLight->addAnimator(anim);
anim->drop();

ISceneNode* pointLightBill = smgr-
                             >addBillboardSceneNode(myPointLight,
                             dimension2d<f32>(20, 20));
pointLightBill->setMaterialFlag(video::EMF_LIGHTING, false);
pointLightBill->setMaterialType(video::EMT_TRANSPARENT_ADD_COLOR);
pointLightBill->setMaterialTexture(0, driver-
                             >getTexture("../../media/portal1.bmp"));

SLight pointLightData;
pointLightData.DiffuseColor = SColor(255, 0, 255, 0);
pointLightData.SpecularColor = SColor(255, 0, 0, 255);

myPointLight->setLightData(pointLightData);

ILightSceneNode* mySpotLight = smgr->addLightSceneNode();
mySpotLight->setPosition(vector3df(0.0f, 10.0f, 0.0f));

SLight spotLightData;
spotLightData.Type = ELT_SPOT;
spotLightData.DiffuseColor = SColor(255, 255, 0, 0);
spotLightData.OuterCone = 100;
spotLightData.InnerCone = 10;

mySpotLight->setLightData(spotLightData);

while(device->run())
{
    mySpotLight->setRotation(rotation);
}
```

What just happened?

Ok, a few things are going on here. We've applied some of the techniques that are not directly related to lighting. First we change the camera type to FPS camera. Instead of calling `addCameraSceneNode()`, by just calling `addCameraSceneNodeFPS()` and passing rotation and movement speed, we can now use mouse and arrow keys to move around in our scene.

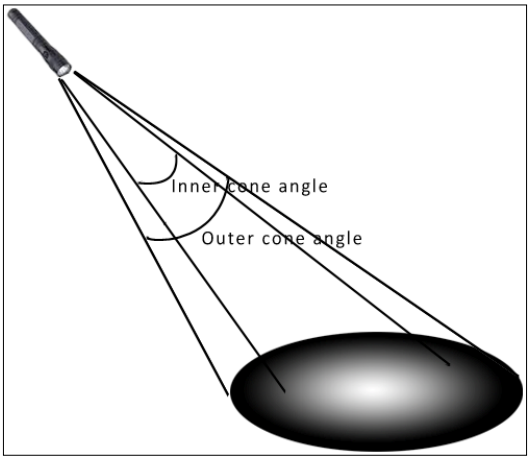
Our point light has a fly-circle animator, but it's not visible yet. So it'd be nice to have an image attached to our light node. For that, we add a billboard scene node to our scene by calling `addBillboardSceneNode()` and pass our point light node as the parent node so that it'll follow wherever our point light goes. Then, we set our material texture to a particle image called `portal1.bmp`, which can be found in the `media` folder. And finally, we create our light data object and assign it to our point light.

Next, we create a new light scene node. This time when we create our light data object, we set the light type as `ELT_SPOT`, and set two special properties for spot lights: outer cone angle and inner cone angle. Run the code and notice the red light effect when the rotating spot light hits our objects:



Spot light

Spot lights are just like torch lights. They have the direction and the color. The direction of a spot light is calculated from its rotation, and they attenuate over the distance travelled. There are two additional properties for the spot lights: outer cone angle and inner cone angle. Spot lights have a brighter inner cone and a larger outer cone. The brightness of a spot light decreases gradually from the inner cone to the outer cone. Take a look at the following illustration to better understand what they mean:



Time for action – manipulating shininess and specular light color

We'll now set the specular color of our simple point light and set the shininess value to our materials.

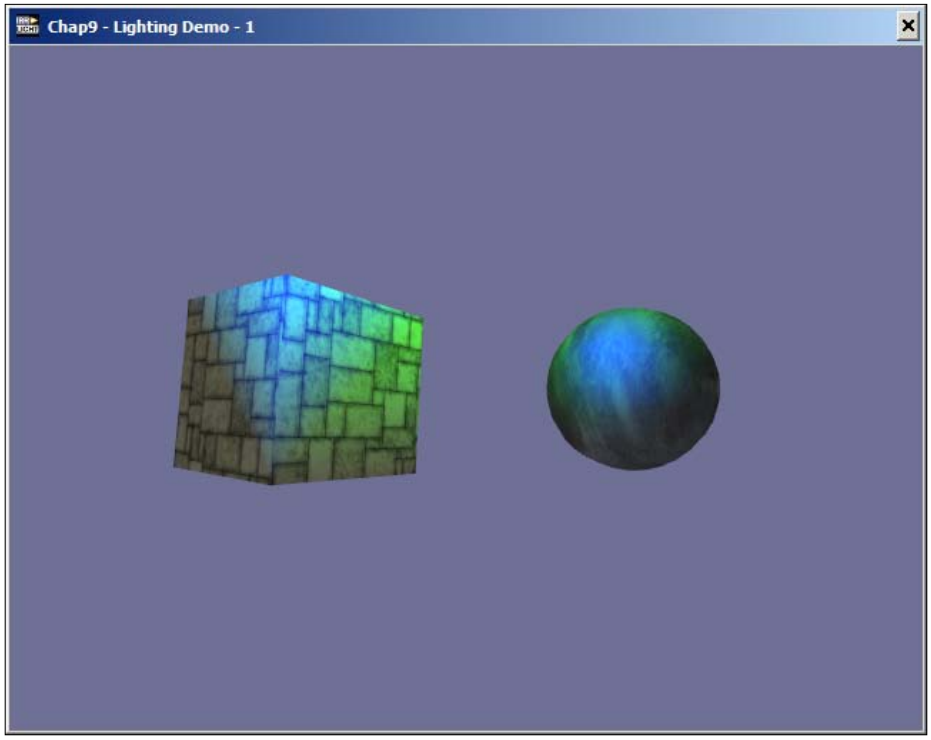
```
IMeshSceneNode* myCube = smgr->addCubeSceneNode();
IMeshSceneNode* mySphere = smgr->addSphereSceneNode();
if (!myCube || !mySphere)
    return 1;

myCube->setPosition(vector3df(-10.0f, 0.0f, 0.0f));
mySphere->setPosition(vector3df(10.0f, 0.0f, 0.0f));

if (myCube && mySphere)
{
    myCube->setMaterialFlag(EMF_LIGHTING, true);
    myCube->getMaterial(0).Shininess = 20;
    myCube->setMaterialTexture( 0,
    driver->getTexture("../../media/wall.jpg") );
```

```
    mySphere->setMaterialFlag(EMF_LIGHTING, true);
    mySphere->getMaterial(0).Shininess = 5;
    mySphere->setMaterialTexture( 0, driver-
                                >getTexture("../../media/wall.bmp"));
}
smgr->addCameraSceneNode(0, vector3df(0, 15, -25), vector3df(0, 0,
                                                              0));
smgr->setAmbientLight(SColor(255, 80, 80, 80));
ILightSceneNode* myLight = smgr->addLightSceneNode();
myLight->setPosition(vector3df(0.0f, 20.0f, 0.0f));
SLight lightData;
lightData.DiffuseColor = SColor(255, 0, 255, 0);
lightData.SpecularColor = SColor(255, 0, 0, 255);
myLight->setLightData(lightData);
vector3df rotation = vector3df(0.0f, 0.0f, 0.0f);
while (device->run())
{
    myCube->setRotation(rotation);
    mySphere->setRotation(rotation);
    rotation +=0.02f;
    if (rotation.X >= 360.0f)
    {
        rotation.X = 0.0f;
        rotation.Y = 0.0f;
        rotation.Z = 0.0f;
    }
}
```

What just happened?

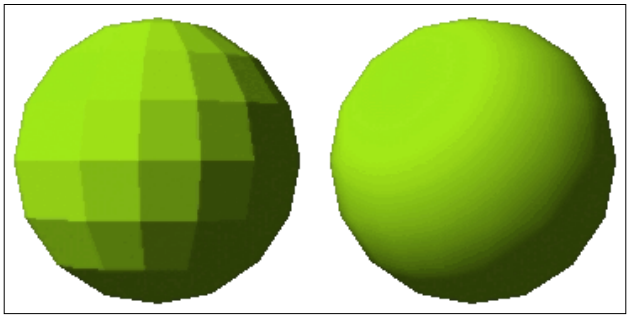
Here we set the shininess of the cube and the sphere `20` and `5` respectively. We can retrieve the material of a mesh scene node by calling `getMaterial()` method and passing the material index, which is `0` in our case, since we only have one material for each object. Once we have material instance, we can access and assign the different properties of a material including shininess.

Specular reflection from an object produces high lights like a shiny gem under a spot light. The shininess value is multiplied with the specular color component of the incoming light affecting the size and brightness of high lights being produced. The amount of reflection depends on the location of the camera viewing the object and it's the brightest along the direct angle of the reflection. Irrlicht recommends that a value of `20` is commonly used. If set to `0`, no specular high lights are being used. This shininess value of a material can be anything in the range between `0.5` and `128`, the higher the value, the smaller and brighter the highlight:

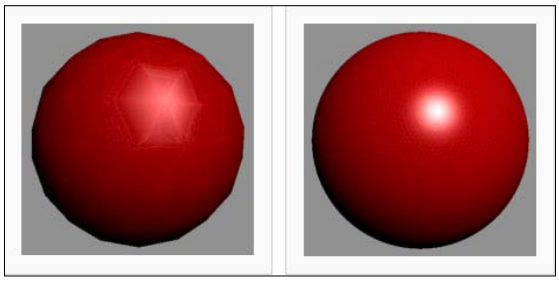


Gouraud shading

Observe how the different shininess values give different specular reflections. You might notice that the specular reflection is harder on the cube model than the sphere model. This is because Irrlicht uses a shading method called **Gouraud** shading to render the materials. Gouraud shading is a simple way to smoothly shade the surface of each polygon by interpolating the color values of nearby vertices. It works pretty well over cheap flat shading methods since a vertex is connected to three or more nearby vertices and blending across the colors of each vertex makes the looks smoother:



But it has a weakness with specular lights (high lights reflection). Since it depends on the number of vertices to interpolate the vertex color between them, it doesn't look good on objects with a low polygon count. And this is exactly what is happening with our cube model. Our cube model only has eight vertices and six faces. Each face is made up of two triangles, so blended shading on such a low poly object results in sharp edges around specular reflections as shown the following image (the image is from `http://en.wikipedia.org/wiki/Gouraud_shading`):



Time for action – adding a shadow

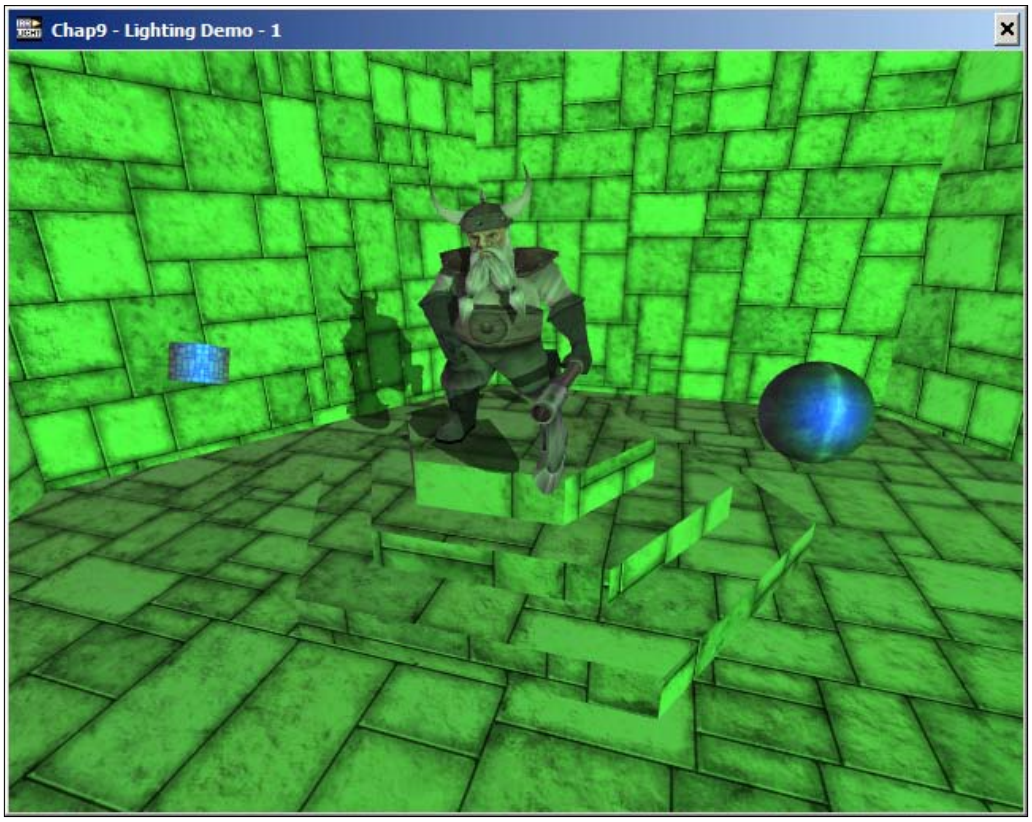
Let's create a more interesting scene with a rotating point light and an animated object with a real-time shadow:

```
IrrlichtDevice *device = createDevice(video::EDT_DIRECT3D9,
                        dimension2d<u32>(640, 480), 32, false, true,
                                        false, 0);

IanimatedMesh* mesh = smgr->getMesh("../../media/room.3ds");

smgr->getMeshManipulator()->makePlanarTextureMapping(    mesh-
                                      >getMesh(0), 0.004f);

IsceneNode* node = 0;
node = smgr->addAnimatedMeshSceneNode(mesh);
node->setMaterialTexture(0, driver-
                        >getTexture("../../media/wall.jpg"));
node->getMaterial(0).SpecularColor.set(0, 0, 0, 0);

//add shadow
IAnimatedMeshSceneNode* dwarf = smgr->addAnimatedMeshSceneNode(smgr-
                                >getMesh("../../media/dwarf.x"));

dwarf->addShadowVolumeSceneNode();
dwarf->setPosition(core::vector3df(-90, 70, 30));
dwarf->setAnimationSpeed(15);
dwarf->setScale(core::vector3df(2, 2, 2));
dwarf->setMaterialFlag(video::EMF_NORMALIZE_NORMALS, true);

smgr->setShadowColor(video::Scolor(150, 0, 0, 0));

IcameraSceneNode* fpsCamera = smgr->addCameraSceneNodeFPS(0,80.0f,
                                                          0.3f);
fpsCamera->setPosition(vector3df(0, 150, -250));
fpsCamera->setTarget(vector3df(0, 0, 0));
```

What just happened?

We need to pay a little bit of attention to `createDevice()` function call. The fifth parameter of this function is set to `true`. Casting a real-time dynamic shadow is an expensive process and Irrlicht uses something called stencil buffer to render real-time shadows. This fifth parameter of device creation is a flag to enable or disable this buffer. If this is set to `true` and you don't see any shadow, or if your program is running too slow, it's probably because your graphics card doesn't support this stencil buffer.

We loaded `room.3ds` and `dwarf.x` model files to our scene. Our dwarf model also has animations. Adding shadows in Irrlicht can be accomplished just by calling `addShadowVolumeSceneNode()` method. We then scale it a bit to match with our room size. We then set the shadow color of our scene by calling `setShadowColor()` function and pass a little transparent black color:



Pretty cool, huh?

Summary

In this chapter, we've covered:

- Basic knowledge of lighting and materials in computer graphics
- Using irrEdit to set up lighting for our scenes
- Adding and manipulating lights in Irrlicht
- Adding shadows

Moreover, we've also learnt how to use built-in animators to do simple animations using translation, rotation, and scaling.

10
Creating Eye Candy Effects with Particle Systems

Now, we've learnt the basics of implementing dynamic lights in Irrlicht. Next, we'll take a look at implementing particle effects. We'll also learn how to use irrEdit as a particle editor to add particle systems to our application.

In this chapter, you'll learn the following:

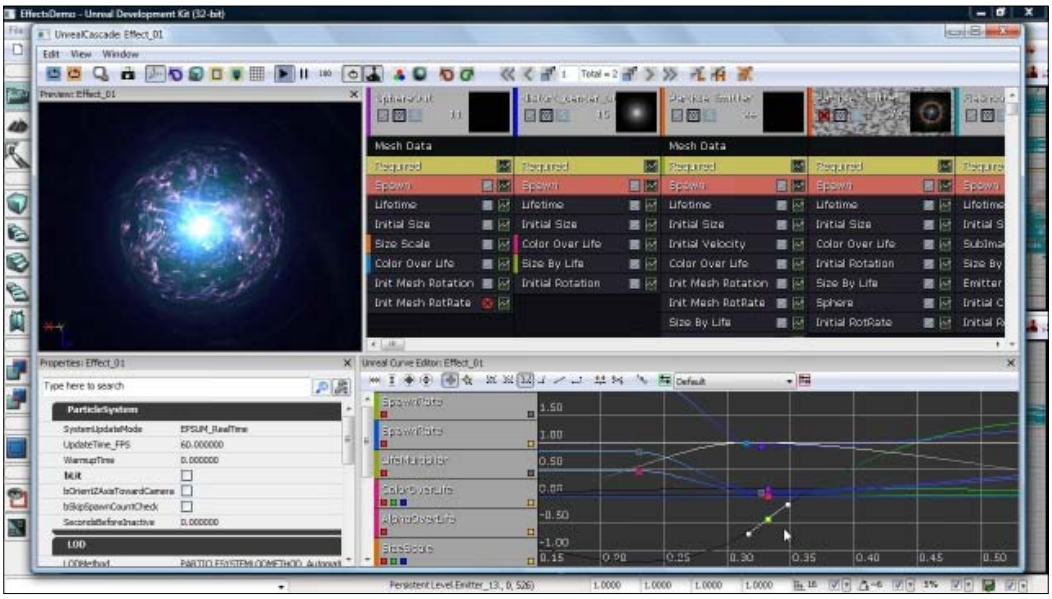
- Understanding particle systems
- Emitter and Affector objects
- Adding and removing particles
- Creating simple effects

Particle systems

Particle systems are widely used to generate natural phenomena effects such as fire, smoke, explosions, sparks, weather (rain, snow, and so on), and imaginary visual effects such as spells, power ups, and the list goes on. Those visual effects are very hard to create manually and even if it's doable, it won't look realistic. For that reason, a person called William T. Reeves came up with an idea to programmatically simulate them in a more realistic way using physics concepts, and demonstrated it in his work of creating a **Genesis** effect at the end of the movie Star Trek II. Reeves is regarded as a person who coined the term *particle systems* in his paper, *Particle Systems— a Technique for Modeling a Class of Fuzzy Objects*, published in 1983.

Basically, the idea behind it was that any particle system consists of particles that have properties such as weight, life-time, and so on, that are affected by the physical properties of the virtual world such as gravity, wind direction, and so on, and emitters that have properties of how these particles should be emitted for example, emission rate, force, and so on.

As you can imagine, creating visual effects with particle systems is really a creative process that requires a lot of time and effort to tweak and fine-tune the various properties of particles and emitters. If we've to recompile the code for every little property value we change, it's going to be a really frustrating and tedious process which will also require more time. So we use tools to tweak those properties of particles and emitters, visually to aid productivity. These tools to edit particle systems are called, guess what, particle editors. There are a lot of particle editors out there that are available as standalone products, as well as integrated tools of a particular game and graphic engines. Among them, Unreal Engine's Cascade is one of the most advanced and powerful particle editors, see the following screenshot:



So what about Irrlicht? Well, the irrEdit editor that we have seen and used in previous chapters can also be used as a simple particle editor. Though it's not as advanced and feature-rich as Cascade, it should do the job that we need at the moment—which is to design and place particle systems inside game levels.

As of today, there's an interesting channel on YouTube called VintageCG that hosts various old computer-generated special effects including the creation of the previously-mentioned Genesis effect for the movie Star Trek II.

You can check that out here: `http://www.youtube.com/watch?v=Qe9qSLYK5q4&NR=1`.

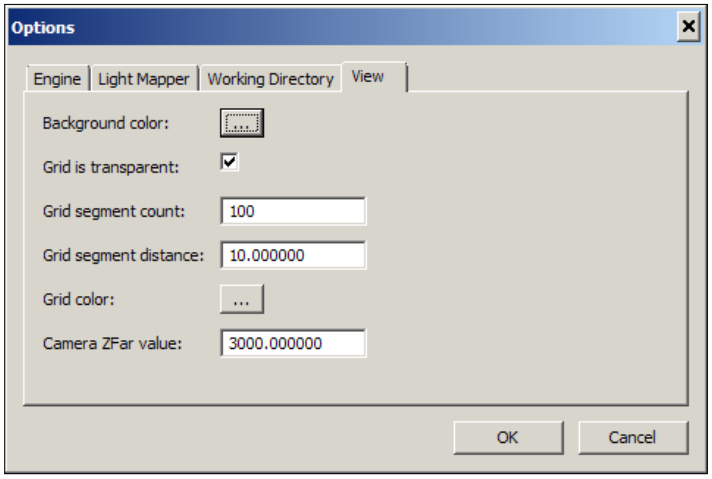
You should also take a look at the particle effects created by Karl Sims, in 1988:

`http://www.karlsims.com/particle-dreams.html`

Time for action – creating a simple particle effect in irrEdit

We'll now create a simple particle effect in irrEdit editor before going into coding particle systems:

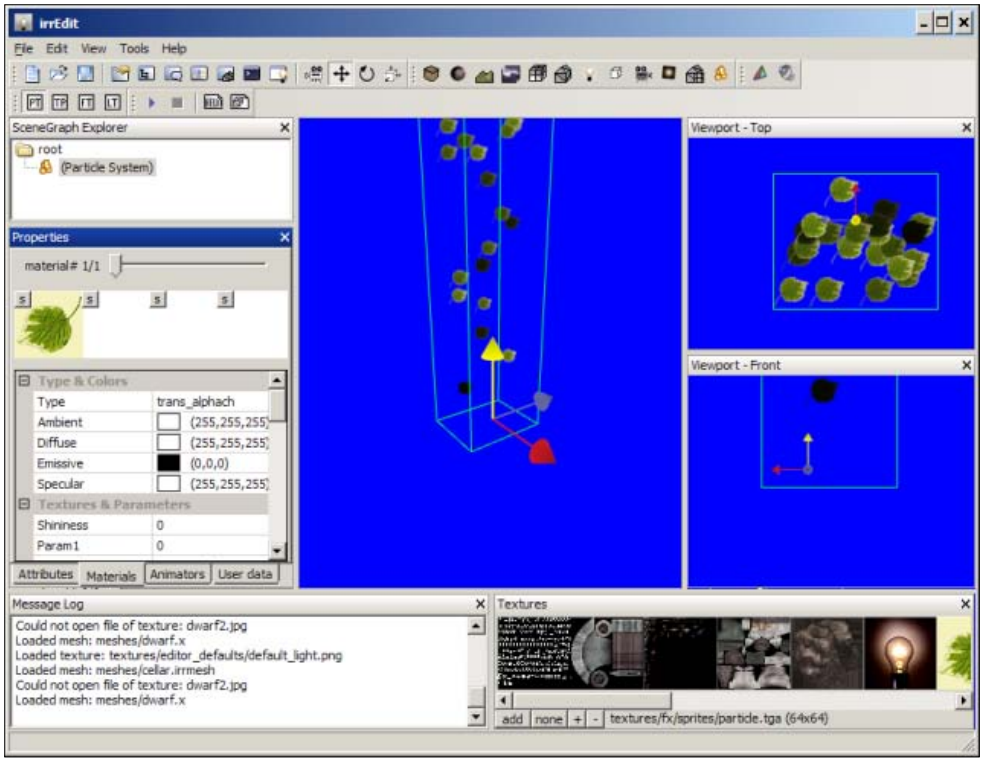
1. Create a new scene.
2. Delete everything in the scene graph.
3. Go to **Tools | Options**. Choose the **View** tab and set the default background color of view to blue:



4. Add a new particle system by either choosing **Particle System** from the toolbar or by choosing **Insert | Particle System** from the context menu inside the view:

What just happened?

We created a new scene and added a default particle system to the level. In step 2 and 3, we deleted every object inside the level and set the view background color to blue. This is just to make the particle system more obvious, just for demo purposes. You'll see leaves emitted from a point inside the level as shown in the following screenshot:



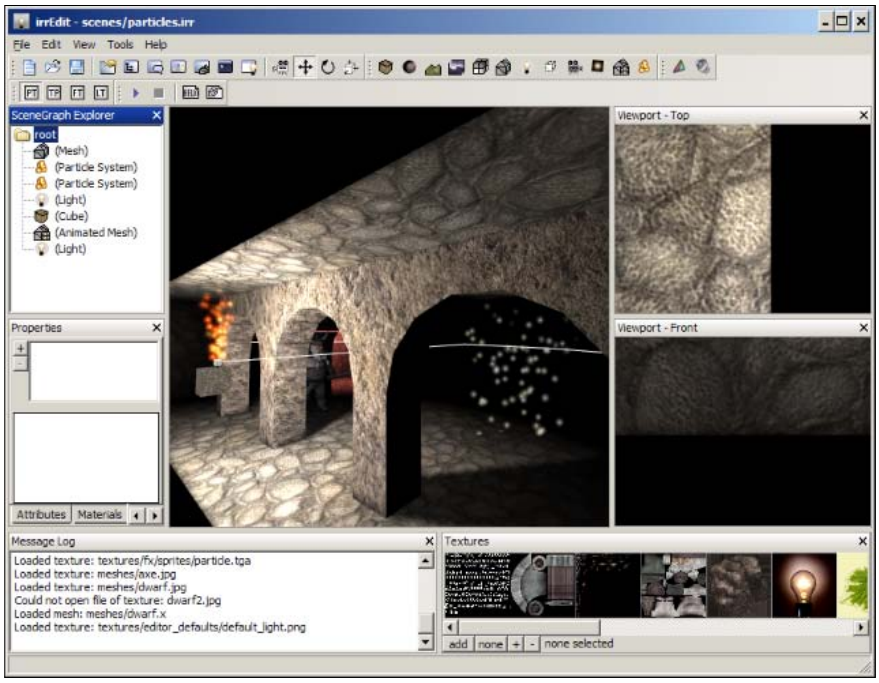
Have a go hero – exploring particle systems in irrEdit

irrEdit comes with an example scene with particle systems. Open `particles.irr` under the `scenes` folder and you'll see something as shown in the next image. Select the particle systems added and study the attributes set to them from the **Properties** panel. Next, we will modify the particle's texture and material.

Time for action – modifying the particle material

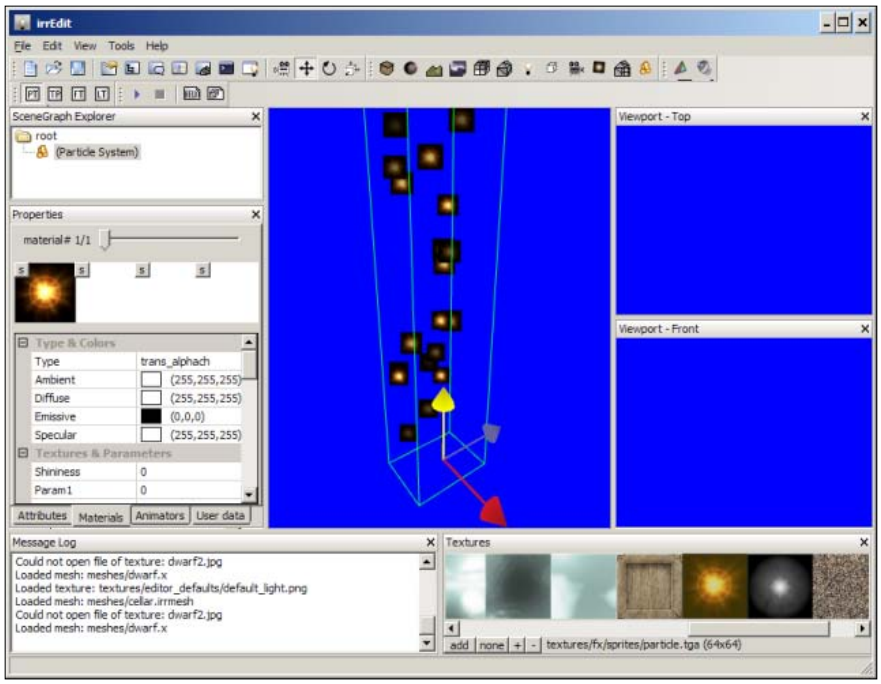
Obviously, emitting leaves from somewhere in a game level is not something that we want to do here. So, let's change those leaves into something that look nicer:

1. Create a new scene, add a particle system and select that particle system.
2. Go to the **Materials** tab in the **Properties** panel.
3. Click on the **none** button from the **Textures** panel:



4. Click on the **s** texture image of the **Materials** tab.
5. Select **particle.tga** sprite and click on the **s** button again. (If you can't find that sprite, click on **add** and add some appropriate sprites to be used as particles):

6. Change material type to `trans_add`:

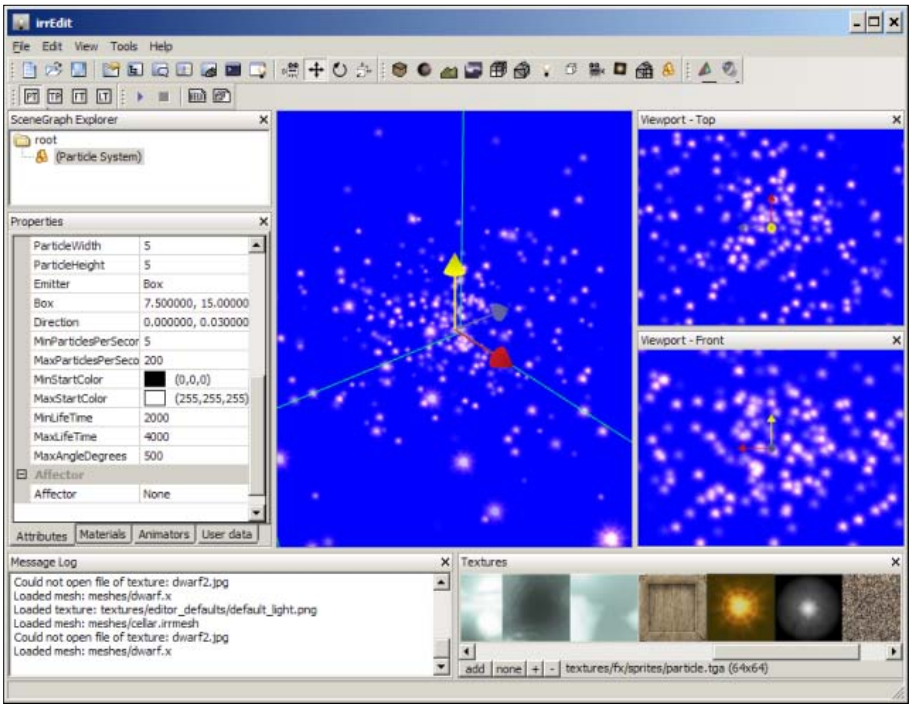


What just happened?

We added a default particle system, same as we did previously. Next, we replaced the default material with the one we like. Once we've changed the texture sprite, the particle system will use it immediately. But it doesn't look right. This is because of the type of material we chose for our particles. Irrlicht has different modes of material to use with different types of texture, lighting, effects we want to create, and so on. By default, `trans_alphach` is used to create new particles. It means to use the texture's alpha channel and make it transparent and blend in with the background color, a technique known as **alpha blending**. It was working fine with the previous leaf texture, because that leaf texture is in `.png` format with an alpha channel, so it was not rendered with square boxes. Now with our new `particle.tga` texture, the alpha is not stored in a separate alpha channel but with black color. So, we changed the material type to `trans_add`, which means to make the texture transparent based on grey scale (total black means 100% transparent) and to calculate the final color of the material by adding the source and destination colors. This technique is called additive blending. If you don't get all these things right away, never mind. Just keep in mind that additive blending, `trans_add` in irrEdit, is mainly used to render particle systems.

Have a go hero – playing around with the emitter properties

Select the particle system we just created. You'll see different pairs of minimum and maximum properties. These are the ranges that the emitter can randomly pick up any value in between. This way, we can achieve various particles emitted randomly while keeping them in our preferred range. The property names are quite self-explanatory. Try to play around with these properties to achieve an effect like this:



Time for action – adding a particle system in Irrlicht

Now, let's get started with coding. To create a particle system in Irrlicht, we can use the following procedure:

1. First we'll add a particle system to our scene.

```
IParticleSystemSceneNode* ps = smgr-
                               >addParticleSystemSceneNode(false);
```

2. Next, we'll create an emitter to emit that particle.

```
IParticleEmitter* em = ps->createBoxEmitter(
                          aabbox3d<f32>(-5, 0, -5, 5, 1, 5),
```

```
                                    vector3df(0.0f, 0.1f, 0.0f),
                                    50, 200,
                                    SColor(0, 0, 0, 255),
                                    SColor(0, 255, 255, 255),
                                    800, 1000, 0,
                                    dimension2df(10.f, 10.f),
                                    dimension2df(20.f, 20.f));
```

3. Set the emitter to the particle system we just created and release the emitter since we've already set it to our particle.

```
ps->setEmitter(em);
em->drop();
```

4. Set the particle system properties.

```
ps->setPosition(vector3df(-70, 60, 40));
ps->setScale(vector3df(2, 2, 2));
ps->setMaterialFlag(EMF_LIGHTING, false);
ps->setMaterialFlag(EMF_ZWRITE_ENABLE, false);
ps->setMaterialTexture(0, driver-
                       >getTexture("../../media/fireball.bmp"));
ps->setMaterialType(EMT_TRANSPARENT_ADD_COLOR);
```

What just happened?

We added a particle system to our scene using the `addParticleSystemSceneNode()` method of scene manager. We can pass a `true` to this method if we want to create a simple particle system with Irrlicht's default emitter. Since we are going to create our own emitter, we passed `false`. Next, we created a box emitter by calling `createBoxEmitter()` and passed the properties of our emitter. Here is the explanation of this method's parameters available from the header file:

- **box**: The box for the emitter. The particles will be emitted from any point within the range of this box dimension.
- **direction**: Direction and speed of particle emission.
- **minParticlesPerSecond**: Minimal amount of particles emitted per second.
- **maxParticlesPerSecond**: Maximal amount of particles emitted per second.
- **minStartColor**: Minimal initial start color of a particle. The real color of every particle is calculated as random interpolation between `minStartColor` and `maxStartColor`.

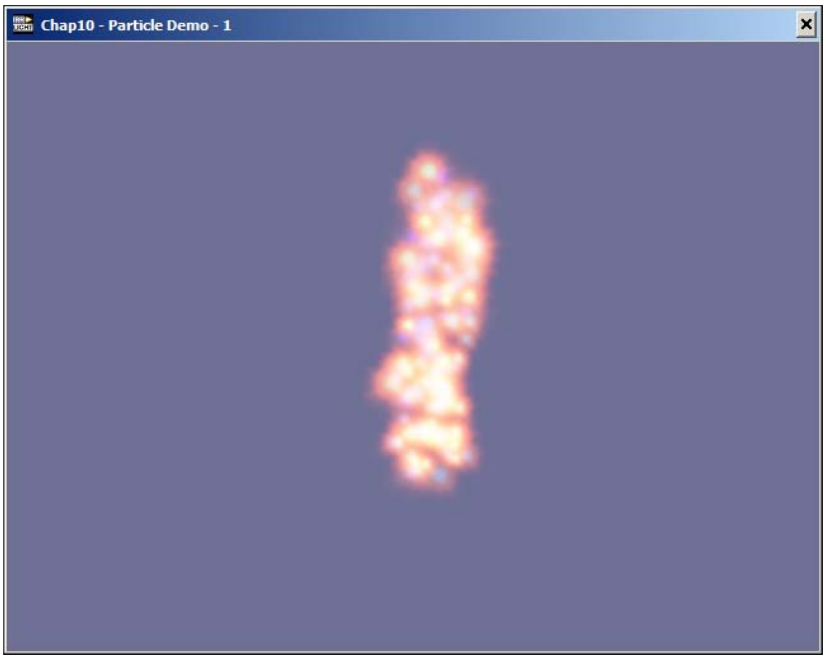
- **maxStartColor**: Maximal initial start color of a particle. The real color of every particle is calculated as random interpolation between `minStartColor` and `maxStartColor`.
- **lifeTimeMin**: Minimal lifetime of a particle, in milliseconds.
- **lifeTimeMax**: Maximal lifetime of a particle, in milliseconds.
- **maxAngleDegrees**: Maximal angle in degrees, the emitting direction of the particle will differ from the original direction.
- **minStartSize**: Minimal initial start size of a particle. The real size of every particle is calculated as random interpolation between `minStartSize` and `maxStartSize`.
- **maxStartSize**: Maximal initial start size of a particle. The real size of every particle is calculated as random interpolation between `minStartSize` and `maxStartSize`.

This function will return a pointer to the created particle emitter.

To become an advanced Irrlicht programmer, it is important to explore the Irrlicht's source code. In Visual Studio, you can go to the definition of a particular method by choosing **Go to Definition** from the right-click context menu. Books will teach you better if you use them in conjunction with reading the actual source code accompanied by the original authors' valuable comments. Irrlicht comes with a lot of useful comments. Learning to know the actual engine source code is essentially important because, from time to time, you'll find bugs within the engine. And when you do some research on Irrlicht forums, you'll find some discussions and possible solutions. If there is a solution that is not integrated into a release version, you'll need to modify the engine's source to fix that bug yourself. So making yourself familiar with the engine's source will be an advantage for you and you can also learn a lot from there as well. This is one of the benefits of open source software.

We then call the `setEmitter()` method to use the emitter we've created to emit the particle system. The last part is setting up the properties of the particle system. `EMT_TRANSPARENT_ADD_COLOR` is the same as `trans_add`, when we are using irrEdit. We also set the `EMF_ZWRITE_ENABLE` property of our material to `false`. This flag tells Irrlicht, whether the depth value should be written, or discarded. If `zwrite` is enabled, elements in front can cover things behind, which would not be rendered at all. By turning this flag off, it is ensured that without any assumption for the render order, all elements are rendered. This is often a good idea for transparent elements, which shall not remove things from behind, but instead let them shine through.

Since our particle material is transparent and the colors are additively blended together, we don't really want to use this z-buffer feature. So we disable this flag:



Go to the definition of `EMT_TRANSPARENT_ADD_COLOR` and see what other material types are available. Try to learn the differences between various materials from the comments provided.

Time for action – adding a simple water surface

Adding an effect like a water surface with reflections and refractions is quite easy in Irrlicht. Let's take a look at how to do this:

1. Create a plane mesh from the code using `addHillPlaneMesh()` method.

```
mesh = smgr->addHillPlaneMesh("waterMesh",
                              dimension2d<f32>(30, 30),
                              dimension2d<u32>(30,30), 0, 0,
                              dimension2d<f32>(0, 0),
                              dimension2d<f32>(10, 10));
```

2. Pass that mesh to the `addWaterSurfaceSceneNode()` method with other properties.

```
node = smgr->addWaterSurfaceSceneNode(mesh->getMesh(0), 3.0f,
                                      300.0f, 30.0f);
```

3. Set the position and other properties.

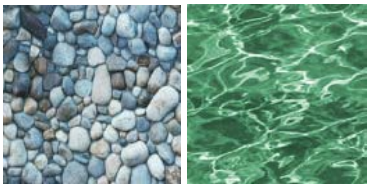
```
node->setPosition(vector3df(0, 7, 0));
node->setMaterialTexture(0, driver-
                         >getTexture("../../media/stones.jpg"));
node->setMaterialTexture(1, driver-
                         >getTexture("../../media/water.jpg"));
node->setMaterialType(EMT_REFLECTION_2_LAYER);
node->setMaterialFlag(EMF_FOG_ENABLE, true);
```

What just happened?

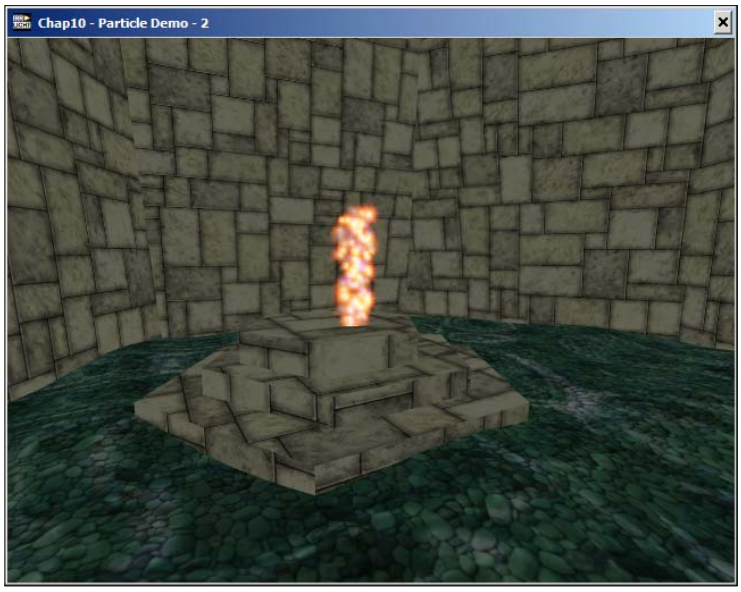
We made a mesh procedurally using the `addHillPlaneMesh()` method. This is a method to generate hills like meshes on the fly. Here we are using it to generate a plane mesh with no hills on it. And thus, we passed zeroes to hill height and hill count parameters. The first and second parameters are the tile size and number of tiles respectively for both x and y dimensions.

Then we add the water surface scene node and pass the plane mesh we've created together with some properties to generate waves on our water surface. These three main properties are wave height, wave speed, and wave length.

Next, we added two textures and set the material type to EMT_REFLECTION_2_LAYER. This material type takes two textures. The first one is the base texture for the water and the second one is used as the reflection of the environment:



This will finally create a water surface with interesting reflection and refraction effects as shown in the following image:



Have a go hero – adding some fog to the scene

Again, try to go to the definition of the `addHillPlaneMesh()` method and study the source code. Next, use a function called `setFog()` to add some fog. Research that function in Irrlicht source and study how to use it yourself. The final look of your effect should look something as shown in the following image from some distance. (Hint: `setFog()` is a method of `IVideoDriver` class):

Time for action – adding a particle affector

Imagine that you throw an object into the air. In this case, you can be assumed as an emitter and the object as a particle. What happens once you've released it from your hand? You now have no control over this object, but the force of gravity. So the gravitational force attracts the object back down. Let's take a look at another example, such as fire particles. Once you've started it, a number of environmental effects will affect the fire such as the direction of the wind. In Irrlicht, those kind of external influences are called **affectors**. Emitters are only responsible for how the particles should be emitted initially. Once a particle has been emitted, its properties such as size, transparency, direction, and so on, are modified by affectors. There are six affectors implemented in Irrlicht under `IParticleAffector.h` and enumerated under `E_PARTICLE_AFFECTOR_TYPE`:

```
enum E_PARTICLE_AFFECTOR_TYPE
{
    EPAT_NONE = 0,
    EPAT_ATTRACT,
    EPAT_FADE_OUT,
    EPAT_GRAVITY,
    EPAT_ROTATE,
    EPAT_SCALE,
    EPAT_COUNT
};
```

Let's try to add fade out the gravity affectors to our previously created simple particle:

1. Currently our particles are deleted once they reach some time between minimum and maximum life time limits. It'd be nice to make them fade out gradually instead of just deleting them immediately. We can do that by adding a fade out affector:

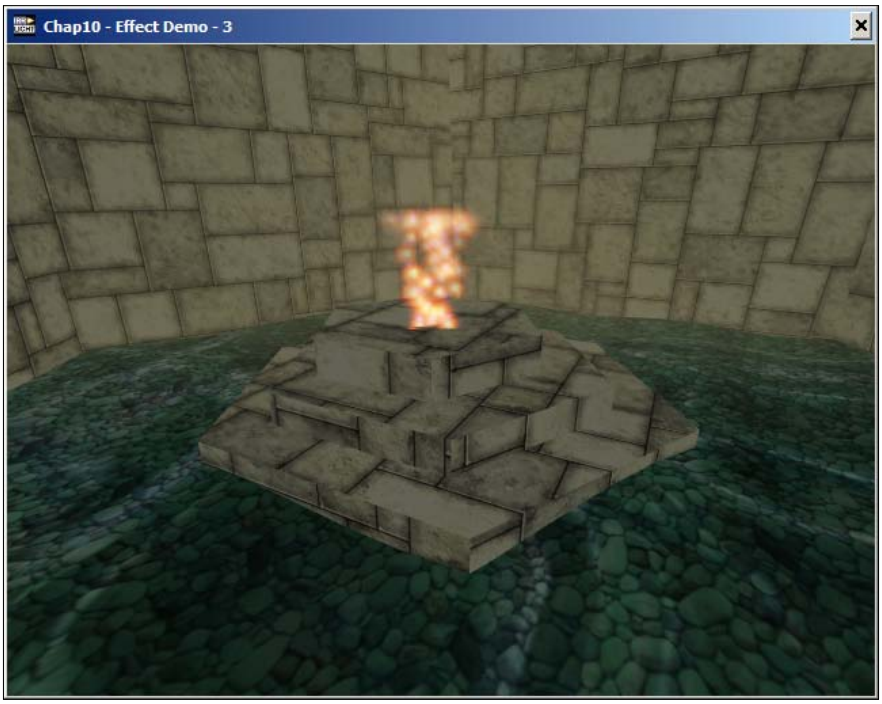
```
IParticleAffector* pa = ps->createFadeOutParticleAffector();
```

2. `ps` is an existing particle system we've created. Then add the affector to our particle system, and then drop it:

```
ps->addAffector(pa);
pa->drop();
```

3. Next, we'll create a gravity affector and set the gravitational force, in this case it is `0.1` downward in Y direction:

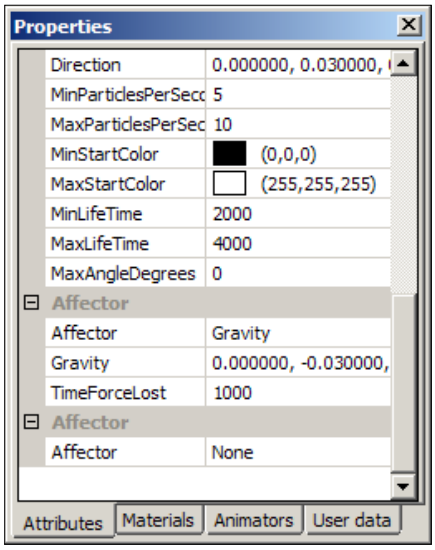
```
pa = ps->createGravityAffector(vector3df(0.0f, -0.1f, 0.0f));
ps->addAffector(pa);
pa->drop();
```



What just happened?

We just created two affectors and added to our particle system. If you run this example, you should see the particles fading out and falling back to the ground as shown in the preceding image. You might be thinking what if you want to do some effects that need new affectors other than the six provided? Well, you can extend the existing affectors to your needs, or you can even write your own affectors. Affectors in Irrlicht are derived from `IParticleAffector`, which describes an interface to other developers that sets the requirements of what a particle affector class should implement.

For example, the gravity we previously added is implemented as `IParticleGravityAffector`, which extends from the `IParticleAffector` class. So you can create your own affectors if you need one by extending the `IParticleAffector` class. As a quick note, you can also add affectors to your particle systems in irrEdit as well, by selecting your particle system and choosing the affector from the **Properties** panel as shown in the following screenshot:



Time for action – activating a particle effect on a mouse event

In the last part of this chapter, we'll extend the particle system we've created in previous tutorials to emit the particles only when you are pressing down the left mouse button. We'll cover how to receive mouse events, how to check the state of the particle system, and decide whether to start or stop the emission:

1. First, we need to create an event receiver class which extends from the `IEventReceiver` class. Following is a simple modification from Irrlicht's mouse and joystick example tutorial. We just removed the joystick part:

```
class MyEventReceiver : public IEventReceiver
{
public:
    //We'll create a struct to record info on the mouse state
    struct SMouseState
    {
        bool LeftButtonDown;
        SMouseState() : LeftButtonDown(false) { }
    } MouseState;

    virtual bool OnEvent(const SEvent& event)
    {
       if (event.EventType == irr::EET_MOUSE_INPUT_EVENT)
       {
          switch(event.MouseInput.Event)
          {
             case EMIE_LMOUSE_PRESSED_DOWN:
                  MouseState.LeftButtonDown = true;
                  break;

             case EMIE_LMOUSE_LEFT_UP:
                  MouseState.LeftButtonDown = false;
                  break;

             default:
                  break;
          }
       }
       return false;
    }

    const SMouseState & GetMouseState(void) const
    {
        return MouseState;
    }

    MyEventReceiver()
    {
    }
};
```

2. In the main function, create `MyEventReceiver()` instance and pass that instance while creating the device:

```
MyEventReceiver receiver;
device = createDevice(EDT_DIRECT3D9, dimension2d<u32>(640, 480),
                      32, false, true, false, &receiver);
```

3. Just before entering the main loop, create a flag to check the particle system state:

```
bool hasEmitter = true;
```

4. Move the `em->drop()` line to the end of the main function just before `device->drop()`.

5. In the main loop, we'll check the mouse state and set the emitter accordingly:

```
if(receiver.GetMouseState().LeftButtonDown)
{
    if (!hasEmitter)
    {
       ps->setEmitter(em);
       hasEmitter = true;
    }
}
else
{
    if (hasEmitter)
    {
       ps->setEmitter(0);
       hasEmitter = false;
    }
}
```

6. Run the program and it should only emit the particles while pressing down the left mouse button:

What just happened?

We implemented the event receiver class and in the `OnEvent()` method, we listened for mouse states. We moved the emitter's drop method to the end of the function, as we still want to keep the reference of the emitter object. And in the main loop, we checked the mouse states and our flag, `hasEmitter` to decide the particle emission. We can pass `0` to particle system's `setEmitter()` method to deactivate the emitter and reassign the actual emitter to activate it again. The idea of this tutorial can be used to trigger particular effects based on the input, for example, explosion when a bullet hits the barrel or a gun fire effect when the player click the mouse button.

Summary

We took an approach to try out irrEdit with particle scene nodes and see the result visually. Then we learnt how we can achieve the same result by programming in Irrlicht.

In particular, we've covered:

- How the particle systems work
- Using irrEdit to add and manipulate particle systems
- Adding and manipulating particle systems in Irrlicht
- What are affectors and how we can use them

We've also learnt how to implement a basic input event receiver and start and stop the emission based on the mouse button states. Next, we'll study how to read and write data to external files in Irrlicht including parsing and saving data in XML format.

11
Handling Data and Files

Besides being just a 3D graphics engine, Irrlicht offers some useful utility classes for input/output handling with various kinds of file formats. In the previous chapters, we learnt how to load and use the files such as meshes, textures, and so on. So we can imagine that Irrlicht has implemented some ways to handle and process files and data beneath the scene. The good news is that Irrlicht implements these file processing methods in a structured and object-oriented way that other developers can easily understand, use, and extend. These file-related classes and methods are implemented under the `irr::io` *namespace.*

In this chapter, we'll explore more on the following topics:

- Accessing the file system
- Reading and parsing data from an external file
- Writing data to a file
- Loading data from XML files
- Writing to XML files
- Loading data from archives

Loading data from an external file

Accessing the file system in Irrlicht usually goes like this. First we'll create an instance of the `IFileSystem` interface by calling the `IrrlichtDevice::getFileSystem()` method. We can check if the file exists using the `IFileSystem::existFile()` method. If the file exists, we'll create the file handle instance and open the file by calling `IFileSystem::createAndOpenFile()`. This will return the `IReadFile` handle to the file you are accessing. Use the `IReadFile::read()` method to read the contents from the file and put it into the buffer.

Time for action – loading data from an external file

Let's try to write a simple program where we will go through the previously described process to read a file and print out the contents of that file. Since this is a very short code listing, I'll just paste it here first and we'll examine that in the next session. The following code listing opens a file, reads the contents, and prints it out in the console window:

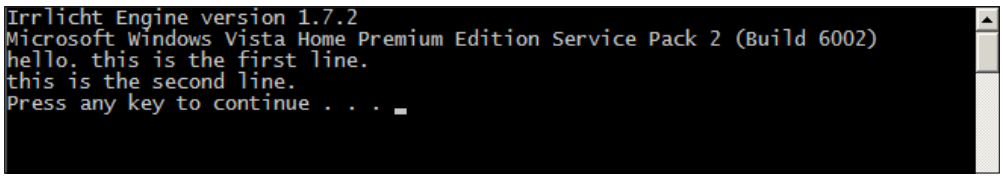
```
#include <irrlicht.h>
#include <iostream>

using namespace irr;
using namespace core;
using namespace scene;
using namespace video;
using namespace io;
using namespace gui;

#ifdef _IRR_WINDOWS_
    #pragma comment(lib, "Irrlicht.lib")
#endif

int main()
{
    IrrlichtDevice* irrDevice = createDevice(EDT_NULL,
                                dimension2d<u32>(0, 0), 16, false);

    if(!irrDevice)
        return 1;

    char* filename = "data.txt";

    IFileSystem* fs = irrDevice->getFileSystem();
    if (!fs->existFile(filename))
        return 1;

    IReadFile* f = fs->createAndOpenFile(filename);
    if (!f)
        return 1;

    char ch;
```

```
    while (0 != f->read(&ch, 1))
        std::cout << ch;

    f->drop();
    irrDevice->drop();
        device->sleep();
    return 0;
}
```

To run the preceding program successfully, you need to have a file named `data.txt` in your working directory. Add a few lines in that file, for example:

```
hello. this is the first line.
this is the second line.
```

And here's the result of running the preceding code:



What just happened?

First, we included `iostream` as we want to use the `cout` function to output the contents of the file to the console window. Then we only put one `pragma` comment that is to link with Irrlicht static library so that you don't need to set up IDE specific compiler configurations to link with that library. Next, we created an instance of the `IrrlichtDevice`. Since we're not doing any graphic stuff here, we don't need anything special and just pass the null device for the video driver. Next, we get the `IFileSystem` instance and check whether the file we want to read exists or not. If the file exists, we proceed to open the file and get the `IReadFile` instance. Actually, we can just directly proceed to `createAndOpenFile()` without checking the file's existence, because that function will fail anyway if there's no file existing. But, at least we learn how to check whether a file exists or not.

Once we've got the `IReadFile` instance, we can use it's `read()` method to start reading the file contents. The `read()` method needs two parameters: the first one is the pointer to the buffer where read bytes are saved and the second is the amount of bytes to read from the file.

So this code, `f->read(&ch, 1)`, basically means that we'll read `1` byte at a time from file `f` and put that byte in the buffer `ch`. We read the whole file and printed out each character in the console window until no byte is left.

Time for action – reading data in tokens, line-by-line

We'll now try to read the data in both token-by-token and line-by-line methods. For that we implemented two additional functions. These two methods are based on the sample snippets posted on the Irrlicht forum by *monchito*:

```
bool readLine(IReadFile* f, stringc& str)
{
    char  ch;
    str = "";
    while (0 != f->read(&ch, 1))
    {
        if (ch == '\n')
           return true;
        else
           str += ch;
    }
    return false;
}
bool readToken(IReadFile* f, stringc& str, char tokenToSplit)
{
    char  ch;
    str = "";
    while (0 != f->read(&ch, 1))
    {
       if (ch == tokenToSplit || ch == '\n')
           return true;
       else
           str += ch;
    }
    return false;
}
```

To test the preceding two functions, we can modify our `main()` function as follows:

```
stringc s;

std::cout << "Testing readToken" << std::endl;

while (readToken(f, &s, ' '))
    std::cout << s.c_str() << std::endl;

std::cout << "Testing readLine" << std::endl;

f->seek(0);
```

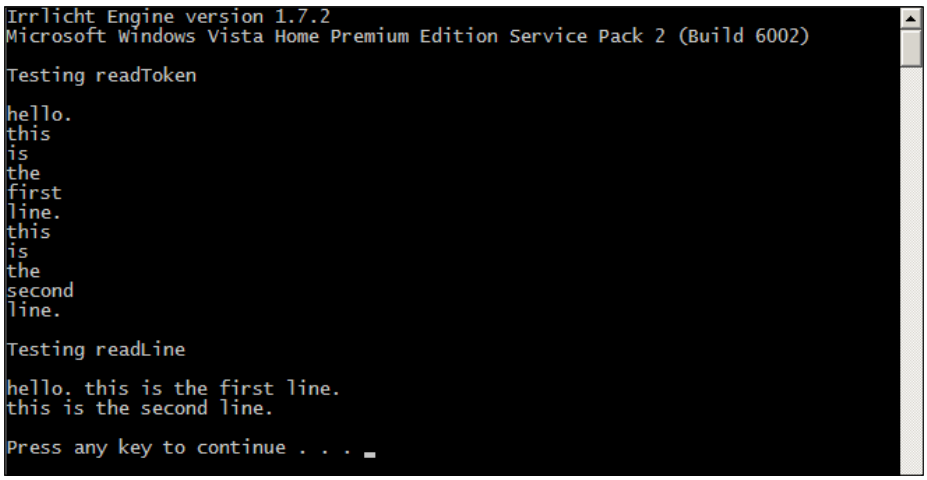
```
while (readLine(f, &s))
    std::cout << s.c_str() << std::endl;

f->drop();
```

What just happened?

The `readLine()` function accepts the `IReadFile` instance and the string variable to put the read bytes. Basically, it reads byte-by-byte as in our first sample code, but it checks for the line feed character `\n`. If the current character is not the line feed character, it continues reading the next byte and concatenates this newly read character to the current string. Once the `\n` character is found, it exits from the function leaving the string with a complete line.

The next function `readToken()` takes in the same parameters as `readLine()` with an extra one to determine which character it should use to tokenize while reading the contents. It'll read the file until it finds such a character or the new line feed.

Next, we test the same `data.txt` file with these two new functions. You should see something like this. For the `readToken()` function, we pass a space character to be used as the token to split. Take note of the `seek()` method. This method sets the current position for the next reading. Since we've done reading the whole file, we need to go back to the start of the file by calling the `seek()` method with `0` parameter:



Time for action – saving data to a file

Writing data to a file is also pretty simple in Irrlicht. After you've created the `IrrlichtDevice`, replace the previous file reading code with the following:

```
char* filename = "saved_data.txt";
IFileSystem* fs = irrDevice->getFileSystem();
IWriteFile* f = fs->createAndWriteFile(filename, fs-
                                       >existFile(filename));
if (!f)
    return 1;
stringc data = "test data saved";
f->write(data.c_str(), data.size());
```

We create `IWriteFile` by using `createAndWriteFile()` instead of `IReadFile` and `createAndOpenFile()`. The `createAndWriteFile()` requires one more additional parameter that is whether to append the contents to that file or overwrite it. So what we did here with that second parameter is that we used the return value from the `existFile()` method. If the file exists, it'll return `true` and we'll pass that `true` to `createAndWriteFile()`, which means to append the data to the existing file. Otherwise we'll just pass `false`, which is the default for `createAndWriteFile()` as well.

Now if you check your file explorer, you should see a file called `saved_data.txt` and inside this file you should see the text we saved: `test data saved`.

Time for action – loading data from an XML file

Next, we'll see how to load the data from an XML file. Irrlicht has an XML parser library called `irrXML`. `irrXML`, which can also be downloaded as a separate product from `http://www.ambiera.com/irrxml/`. Here's an excerpt from Ambiera describing what `irrXML` is about:

> `irrXML` *is a simple and fast open source XML parser for C++. Why another XML parser? The strengths of* `irrXML` *are its speed and its simplicity. It ideally fits into real-time projects that need to read XML data without overhead, like games.* `irrXML` *was originally written as part of the Irrlicht Engine, but after it has become quite mature it now has become a separate project.*

We'll stick with the `irrXML` that Irrlicht already has, since it contains all the necessary files for `irrXML`. So after you've learnt how to use this cool XML parser in this chapter, you can use it for whatever projects that need to parse XML files. To test the code, let's create a simple XML file first with the following data:

```
<?xml version="1.0"?>
<player name="Aung Sithu">
    <level>9</level>
    <hp>100</hp>
    <xp>500</xp>
</player>
```

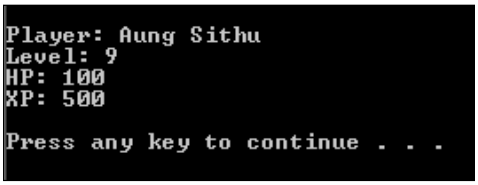
And add the following code in the `main()` function after creating `IrrlichtDevice`:

```
IFileSystem* fs = irrDevice->getFileSystem();
IXMLReader* xml = fs->createXMLReader("data.xml");

stringc playerName = "Player: ";
stringc level = "Level: ";
stringc hp = "HP: ";
stringc xp = "XP: ";

while(xml && xml->read())
{
    switch(xml->getNodeType())
    {
       case EXN_ELEMENT:
       {
          stringc nodeName = xml->getNodeName();

          if (nodeName.equals_ignore_case("player"))
          {
              playerName.append(xml->getAttributeValue(L"name"));
              cout << playerName.c_str() << "\n";
          }
          else if (nodeName.equals_ignore_case("level"))
          {
              xml->read();
              level.append(xml->getNodeData());
              cout << level.c_str() << "\n";
          }
          else if (nodeName.equals_ignore_case("hp"))
          {
             xml->read();
             hp.append(xml->getNodeData());
             cout << hp.c_str() << "\n";
          }
          else if (nodeName.equals_ignore_case("xp"))
          {
             xml->read();
             xp.append(xml->getNodeData());
             cout << xp.c_str() << "\n";
```

```
            }
        }
        break;
    }
}
cout << endl;
delete xml;
```

What just happened?

We use the `equals_ignore_case()` method from the Irrlicht `string` class implemented in `irrString.h` to compare two strings and if it's the element node, we're trying to read the next segment of the XML and used `getNodeData()` to get the value for that node:



Time for action – writing data to an XML file

Next, we'll write a small program that can save the player information in XML format:

1. First we need to get the file system object and create an XML writer object:

```
IFileSystem* fs = irrDevice->getFileSystem();
IXMLWriter* xml = fs->createXMLWriter("saved_data.xml");
```

2. Next, we'll create our variables to hold the data from XML. For the sake of simplicity, we'll just use string type for all the variables:

```
stringw playerName = "Aung Sithu";
stringw level = "20";
stringw hp = "50";
stringw xp = "1000";
```

3. We then write our XML header. The following code will write:

```
<?xml version="1.0"?>
xml->writeXMLHeader();
```

4. Next, we write our `player` element with an attribute `name` and the value with my name. This will write out `<player name="Aung Sithu">` in our XML file and then enter a new line:

```
xml->writeElement(L"player", false, L"name", playerName.c_str());
xml->writeLineBreak();
```

5. Then we write another element `level` inside our `player` element. Put a text value for it and close the tag. This will write out `<level>9</level>` in our XML. And we do the same for `hp` and `xp` elements as well:

```
xml->writeElement(L"level");
xml->writeText(level.c_str());
xml->writeClosingTag(L"level");
xml->writeLineBreak();

xml->writeElement(L"hp");
xml->writeText(hp.c_str());
xml->writeClosingTag(L"hp");
xml->writeLineBreak();

xml->writeElement(L"xp");
xml->writeText(xp.c_str());
xml->writeClosingTag(L"xp");
xml->writeLineBreak();
```

6. Finally, write the closing tag `</player>` for our `player` element and delete the XML handle:

```
xml->writeClosingTag(L"player");

delete xml;
```

As you can see, writing XML files in Irrlicht is really simple and the process just flows naturally. Check in your drive and you should have a file called `saved_data.xml` with the following contents:

```
<?xml version="1.0"?>
<player name="Aung Sithu">
    <level>9</level>
    <hp>100</hp>
    <xp>500</xp>
</player>
```

Time for action – reading specific data types from an XML file

`irrXML` has some helper methods to read the attribute values of specific data types from the XML. These methods basically do fast conversion between string and other numeric types such as integers and floats. Let's see how we can take advantage of those methods:

1. First, let's prepare an XML file something as follows:

```
<?xml version="1.0"?>
<player name="Aung Sithu" level="9" xp="900" speed="0.05" />
```

2. Then add the following code to your `main()` function after creating the Irrlicht device:

```
IFileSystem* fs = irrDevice->getFileSystem();
IXMLReader* xml = fs->createXMLReader("player_data.xml");

if(!xml)
    return 1;

stringc playerName = "Player: ";
int level = 0;
int xp = 0;
float speed = 0;

cout << endl;

while(xml && xml->read())
{
    switch(xml->getNodeType())
    {
       case EXN_ELEMENT:
       {
          stringc nodeName= xml->getNodeName();
          if (nodeName.equals_ignore_case("player"))
          {
             playerName.append(xml->getAttributeValue(L"name"));
             cout << playerName.c_str() << "\n";
             level = xml->getAttributeValueAsInt(L"level");
             cout << "Level: " << level << "\n" ;
             xp = xml->getAttributeValueAsInt(L"xp");
             cout << "XP: " << xp << "\n" ;
             speed = xml->getAttributeValueAsFloat(L"speed");
             cout << "Speed: " << speed << "\n" ;
          }
```

```
            }
            break;
        }
    }
    cout << endl;
    delete xml;
```

The preceding code listing is pretty much self-explained. We created some int and float variables and used `getAttributeValueAsInt()` and `getAttributeValueAsFloat()` to read the data into int and float type variables.

Time for action – reading data from an archive

Irrlicht can also load the files from an archived file. Putting the files in an archive will hide the game-related data and resources from the users and make the size smaller depending on the compression method used. To make the archive safe, we can also provide a password to the archive and use it in the code to unarchive.

To test the following code, we need to prepare a small archive. We'll use the previous `player_data.xml` and put it in an archive. On Windows systems, you can easily do that by right-clicking on it and choose **Send To | Compressed (zipped) Folder**. And name that ZIP file as `archived_data.zip`. By default, Irrlicht can read and load from ZIP, TAR, PAK, and PNK archives. If you want to use other compression methods such as TAR, you can use free and open-source file archiving software such as 7zip:

```
IFileSystem* fs = irrDevice->getFileSystem();
if (!fs->addFileArchive("archived_data.zip"))
   return 1;
IXMLReader* xml = fs-
                  >createXMLReader("archived_data/player_data.xml");
```

Once we've added our archive file to the Irrlicht file system, we can just use it as we usually do.

What just happened?

Let's take a look at the `addFileArchive()` method:

```
addFileArchive(const path& filename, bool ignoreCase=true,
               bool ignorePaths=true,
               E_FILE_ARCHIVE_TYPE archiveType=EFAT_UNKNOWN,
               const core::stringc& password="") =0;
```

The first parameter is obviously the name of the archive to add to the file system. The second parameter is a flag that decides whether or not to ignore case when accessing files inside that archive. If set to `true`, files in the archive can be accessed without writing all letters in the correct case. The third parameter is a flag to ignore paths and if it's set to `true`, files in the added archive can be accessed without its complete path. For example, in our preceding sample code, instead of writing `archived_data/player_data.xml` we can just access the `player_data.xml` file without having to specify the `archived_data` folder structure.

The fourth parameter is the archive type. By default, the type of archive will depend on the extension of the file name. If we want to use a different extension then we can use this parameter to force a specific type of archive. For example, we can name our ZIP files with `.dat` or whatever extension, and trick the users about the actual type of archive.

The last one is the password that is used in case of encrypted archives.

Summary

With this we've come to summarize what we've learnt so far in this chapter. We've covered:

- Irrlicht file system
- Reading and writing data to XML files
- Using archives to compress and hide the contents of our application

There is still some more advanced stuff you can do with Irrlicht, such as implementing your own archive loader if Irrlicht doesn't support a proprietary or encrypted file storage method you want to use. And there are also a lot of small and useful methods such as getting your project's working directory, getting file name, absolute paths, and so on. If something is not working as expected and you can't access the file or can't find the file you just created, it could be the working directory or file path issues. You can use the previously-mentioned functions to check the path where your program is currently reading and writing files. It is recommended to refer to the Irrlicht source solution file and `IFileSystem.h` for more details. They are all well-documented.

The next chapter is about the shader programs. Shaders are an important topic in computer graphics and we'll learn how to use them in our Irrlicht programs.

12
Using Shaders in Irrlicht

It's time for a chapter about those infamous shaders. This chapter is going to be an advanced one with some of the theoretical concepts as well. Though these concepts are introduced here in the later part of this book they are pretty much an important foundation for anyone to become a graphics programmer. If this book is not targeted for Irrlicht beginners but instead for graphics programmers, you will find these concepts in the very first chapters of any graphics programming books.

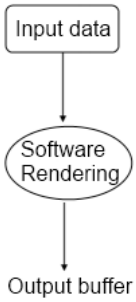
In this chapter, we'll explore more on the following topics:

- What is a shader
- Fragment/Pixel shader
- Vertex shader
- Geometry shader
- Non-photorealistic (NPR) rendering
- Practical use of shaders
- Adding shaders to an Irrlicht program

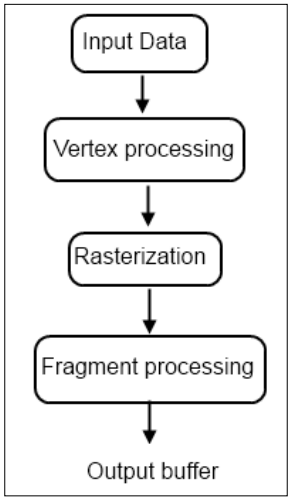
Rendering pipeline

In order to understand shaders we need to go back a little bit into history and study how the graphics rendering have been evolving over time. Back in the days of the early 90s rendering was done by hardcore programmers inside their proprietary software rendering engines. Even though a lot of breakthroughs were happening in the films and entertainment industry before that, interactive real time rendering was not developing that fast because of the available limited hardware power. There was no hardware dedicated to process graphics calculations. In 1992, iS Software released a game called Wolfenstein 3D which is generally regarded as a pioneer in the 3D FPS genre. The main software rendering engine used by this game is called the Wolfenstein 3D engine and later released as an open source to the public. The engine was primarily developed by John Carmack who is considered to be a legendary games and graphics programmer in the industry. (P.S: Personally I'd vote for Niko for initiating and releasing Irrlicht engine which is a lot easier to use.)

The rendering pipeline these days would simply looks like this:



By the late 90s, companies such as 3dfx, nVidia, and ATI started producing graphics processing units (GPUs) which are also known as graphics cards. Together with graphics cards, APIs to interface with those GPUs were also introduced. 3dfx's voodoo chipsets were quite famous among the gamers during the mid 90s but weren't strong enough to compete with others such as nVidia, which later went on to acquire 3dfx. And among the APIs, OpenGL, and Direct3D became dominant as the hardware graphics cards only support those APIs to interface with them. During that era of hardware accelerated graphics rendering the rendering pipeline looked something like this:

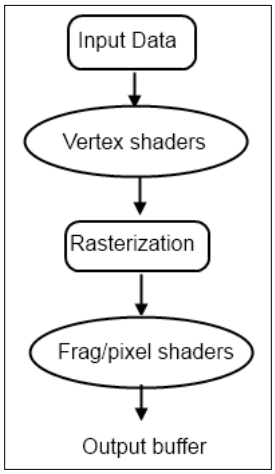


In such a pipeline the application is only responsible for feeding data into the hardware. Since we are talking about 3D scenes, such data can include vertices, textures, camera viewport, and so on. The graphics card then calculates all the necessary processing and output a frame buffer which is ready to be displayed on the monitor. This fashion of rendering a pipeline is known as fixed function pipeline as the functions of the whole rendering process cannot be intervened by outside programmers and instead, are fixed and hardcoded inside the hardware. There's not much difference from software rendering, instead now the calculations are happening inside dedicated hardware and a lot faster than before.

Shaders

Due to this fixed function pipeline, programmers couldn't control the process of rendering states and thus, it was difficult to create games with unique visual styles. That's why all those 3D games before 2000 looked quite the same. But in 2001 a new type of graphics chipset entered the market, GeForce 3 developed by nVidia. This is the first hardware that supports programmable graphics pipeline in compliance with Direct3D 8 API. The graphics processing unit can now run small custom programs called shaders. There were only two shaders available, vertex shaders and fragment/pixel shaders. The programmers can write their own shaders and determine how a vertex should be processed and displayed during the rendering process. Initially, it was pretty hard to become a shader programmer as Assembly language was used to write shaders, but now there are a number of high level 'C' style languages that compile your code to assembly.

Microsoft developed HLSL (High Level Shading Language) to be used with Direct3D API. OpenGL has GLSL (OpenGL Shading Language). Some graphics cards developers also provide their own High Level Shading Languages such as Cg from nVidia. Cg shaders are basically independent of graphics API. In this chapter we'll use GLSL and Irrlicht's OpenGL driver to demonstrate how to use shaders with Irrlicht engine. Now the rendering pipeline becomes something like this:



Vertex shaders

Vertex shaders are the programs that can manipulate vertex data such as position, normal, color, and so on. We can pass either all the vertex data into the graphics pipeline or only the data we want to manipulate. They are run every time for each vertex given to the pipeline.

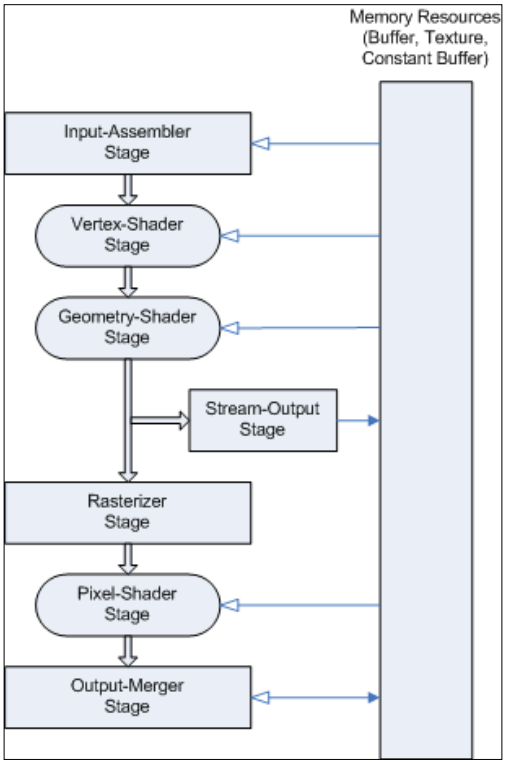
Fragment /pixel shaders

Fragment and pixel shaders are basically the same and used interchangeably. A fragment is just a block of pixels. Like vertex shaders, they are executed inside GPU, once for every pixel in a specified 3D mesh. Fragment shaders operate after the geometry assembly and rasterization but before final output to the buffer. The output of a fragment shader is a final pixel color and z-value for the final rasterization step.

Geometry shaders

A geometry shader is a new type of shader that can generate new types of primitives based on the primitives sent into the pipeline as input. Vertex shaders can only alter the input primitives and the amount of output is always exactly the same as the input primitives. Since geometry shaders can generate new primitives such points, triangles, they are really useful to modify the input mesh complexity, for example, depending on the level of detail (LOD) required. Geometry shaders are supported starting from DirectX 10 and OpenGL 3.2 (though there are some extensions to be used with an older version of OpenGL).

So a more complete figure of a latest rendering pipeline will finally look like this:



Anyway, we're not going to talk about how to write all these shaders in this chapter. This is really out of scope here and in fact, it can be a series of lengthy books. If you want to learn more about GLSL and OpenGL shader programming, lighthouse3d.com has some great tutorials. In this section, we'll focus on how to use Irrlicht to apply the shaders created by others. However, even to use the existing shaders, we still need to understand some basic concepts and that's what we're going to study in the next few sections. The shaders we'll use in this chapter are based on a famous OpenGL book called OpenGL Shading Language (also known as the Orange Book).

Pop quiz – rendering the pipeline and shaders

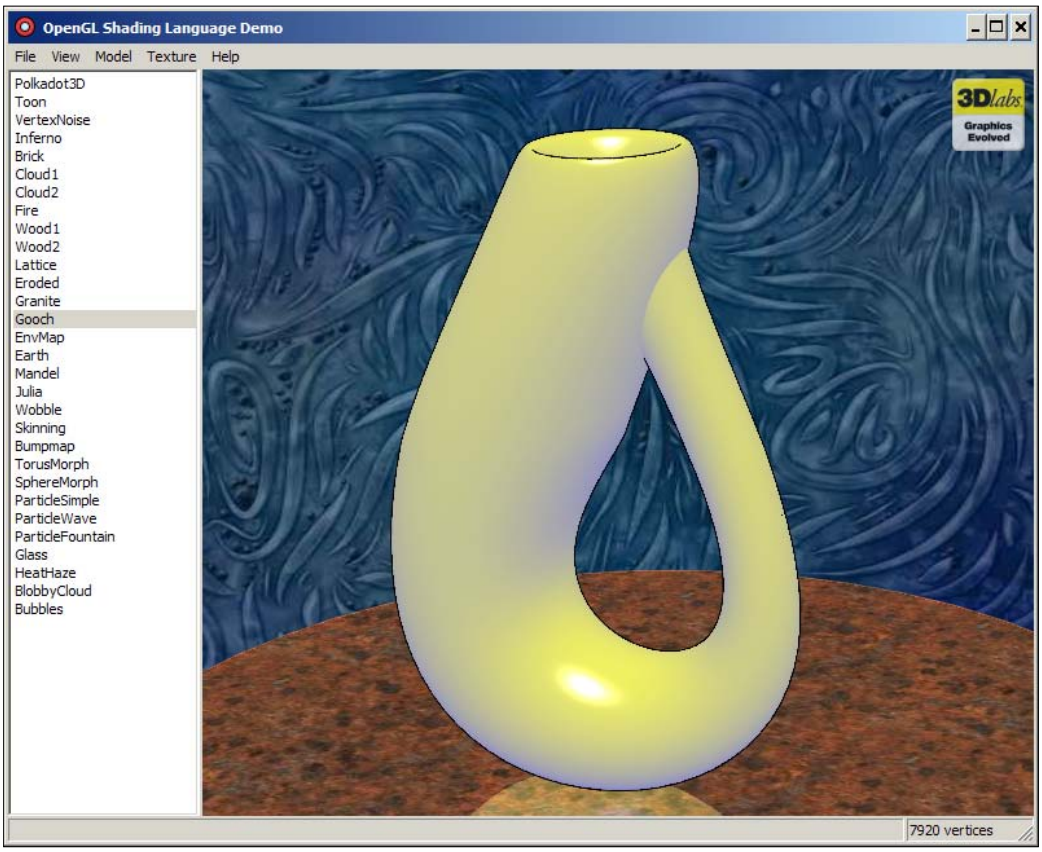
1. Which of the following is a graphics API independent shader language?
    a. HLSL
    b. GLSL
    c. Cg
2. The final colouring of the output pixel can be determined in:
    a. Vertex shader
    b. Fragment shader
    c. Geometry shader

Time for action – setting up the GLSL demo

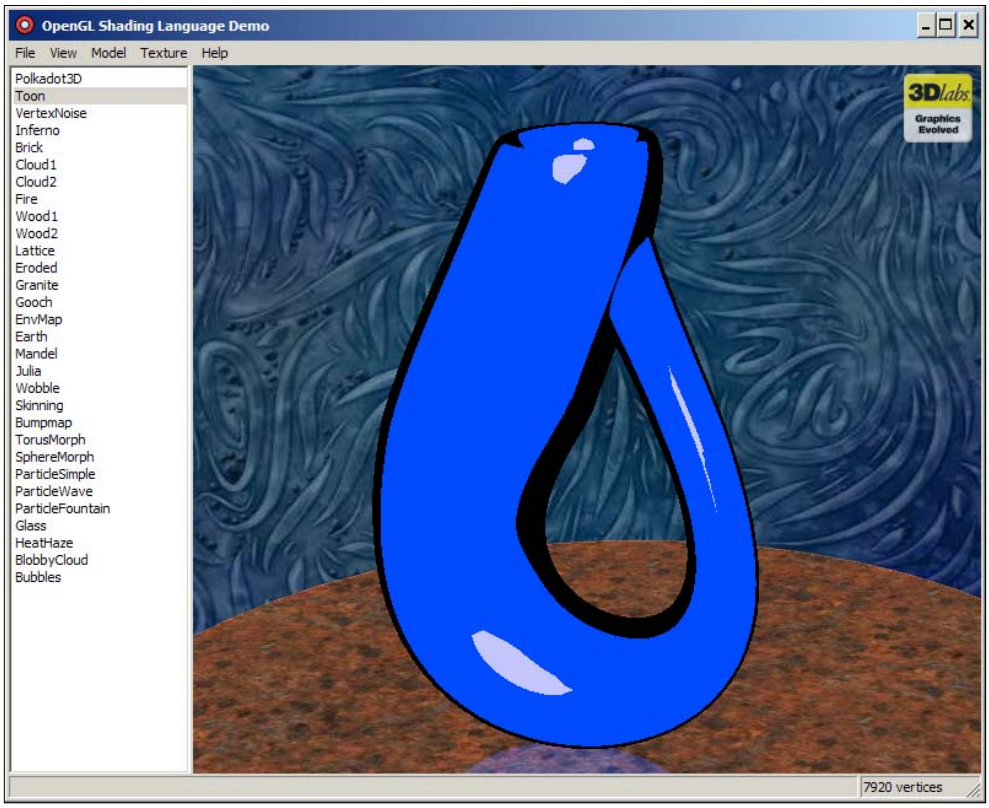
We'll now download and install the GLSL demo from the Orange Book and in the next section, we'll learn how to use some of those shaders from Irrlicht:

1. Go to 3dshaders.com.
2. Choose shader examples from the left menu.
3. Download and install the GLSLdemo Executable from the example list.

4. Run the GLSL Demo program once you've installed it:



These images show two GLSL examples, gooch shader and toon shader. We'll learn how to use these two shaders in our Irrlicht program. The reason we chose them is because they are popular and simple shaders for non-photo realistic rendering (NPR) style. Gooch shading is widely used for technical illustrations and toon shading (also known as cell shading) is used to create cartoon style rendering of 3D scenes and characters:



Such shading effects can create an attractive aesthetic experience if combined with appropriate character and texture designs. There are a lot of games using such toon style shading of which the game XIII from Ubisoft is one of them. See the following screenshot.

Data communication

Since shaders are separate programs from the host Irrlicht program, we need some ways to communicate data between them. In GLSL there are three types of variables for data communications between host and shaders, and between shaders themselves. They are uniform, varying, and attribute types:

- `Uniform` type variables are read-only and available in both vertex and fragment shaders. They do not change during a rendering pass, for example the position or color of a light.
- `Attributes` are the read-only input values for vertex shaders only, for example, the vertex position or normals.
- `Varyings` are used for passing data from a vertex shader to a fragment shader. They can be read and write in vertex shader but can only be read in fragment shader. To use varyings you have to declare the same varying in both vertex and fragment shader. All those three types must be declared in global scope:

Time for action – using a toon shader

Let's copy the `Toon.frag` and `Toon.vert` files from the GLSL demo to your Irrlicht project folder and here's our vertex shader:

```
varying vec3 Normal;
void main(void)
{
  Normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

And this is our little fragment shader:

```
uniform vec3 DiffuseColor;
uniform vec3 PhongColor;
uniform float Edge;
uniform float Phong;
varying vec3 Normal;

void main (void)
{
  vec3 color = DiffuseColor;
  float f = dot(vec3(0,0,1),Normal);
  if (abs(f) < Edge)
    color = vec3(0);
  if (f > Phong)
    color = PhongColor;
  gl_FragColor = vec4(color, 1);
}
```

To feed the constant uniform values to our shader from the Irrlicht program we need to implement the `IShaderConstantSetCallBack` interface, the `OnSetConstants` method in particular. This is how our `ToonShaderCallBack` class should look like:

```
class ToonShaderCallBack : public IShaderConstantSetCallBack
{
public:
  virtual void OnSetConstants(IMaterialRendererServices* services,
    s32 userData)
  {
    services->setPixelShaderConstant("DiffuseColor",
      reinterpret_cast<f32*>(&vector3df(0.0f,0.25f,1.0f)), 3);
    services->setPixelShaderConstant("PhongColor",
      reinterpret_cast<f32*>(&vector3df(0.75f,0.75f,1.0f)), 3);
    float edge = 0.5f;
    services->setPixelShaderConstant("Edge",
      reinterpret_cast<f32*>(&edge), 1);
    float phong = 0.98f;
    services->setPixelShaderConstant("Phong",
    reinterpret_cast<f32*>(&phong), 1);
  }
};
```

And finally following are the steps that we need to do in our main program:

1. Set the shader file paths:

```
path toonPsFileName = "Toon.frag";
path toonVsFileName = "Toon.vert";
```

2. Get an instance of `IGPUProgrammingServices` to run shaders on the GPU:

```
IGPUProgrammingServices* gpu = driver-
>getGPUProgrammingServices();
```

3. Create a new instance of our shader callback.:

```
ToonShaderCallBack* ts = new ToonShaderCallBack();
```

4. Use the `addHighLevelShaderMaterialFromFiles` method to add our shaders to the GPU and store the return value in a variable. The return value will be `-1` if the shader does not compile or some other problem arises. This value should be checked so that some action could be taken if the shader doesn't work:

```
s32 toonSolidShader = 0;

toonSolidShader = gpu->addHighLevelShaderMaterialFromFiles(
  toonVsFileName, "vertexMain", EVST_VS_1_1,
    toonPsFileName, "pixelMain", EPST_PS_1_1,
    ts, EMT_SOLID);
```

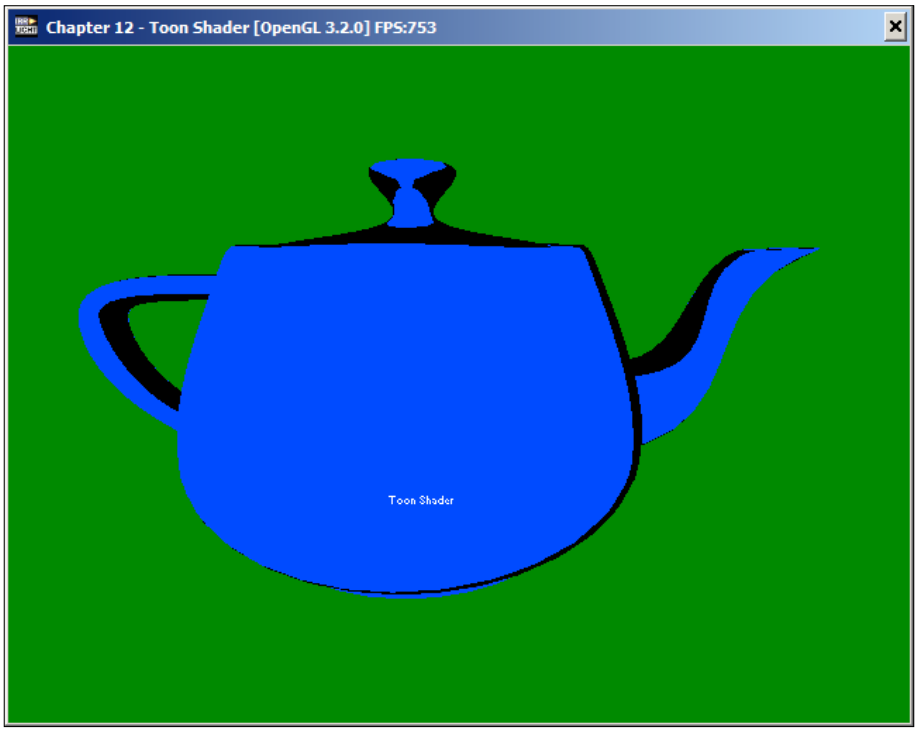
5. Load a mesh on which we want to apply our shader and add into the scene:

```
IAnimatedMesh* mesh = smgr->getMesh("teapot.obj");
IAnimatedMeshSceneNode* meshNode = smgr->addAnimatedMeshSceneNode(
mesh);
```

6. Use our shader material as the desired material type:

```
meshNode->setMaterialType((E_MATERIAL_TYPE)toonSolidShader);
```

And the result of our toon shader on a standard Utah teapot model can be seen in the following image:



What just happened?

There are three parts in the previous tutorial. First we prepared our vertex and fragment shaders which are `Toon.vert` and `Toon.frag` so that they are under the path accessible from our Irrlicht program.

In our vertex shader, we can see a varying type at the start. So we can understand that this variable is going to be shared with the fragment shader. Inside the main method we transformed the vertex position and normal to model-view space. Actually, `gl_Position` is also a varying type but just implicitly declared by the OpenGL. There are several other variables similar to `gl_Position` like for example, `gl_PointSize` and `gl_ClipDistance` which will not be covered here. One thing to note is that such pre-declared variables are different for different shading languages. For example, there's no such variables in Direct3D and only a few automatically passed inputs. So, in such language, more work is required writing an output to a variable. You can always check the latest specification of GLSL from this link `http://www.opengl.org/registry/`.

In the fragment shader, we can find four uniform variables. These are the values we'll be passing from our Irrlicht host program. Then there's the varying value which will be read from our vertex shader. The basic idea of a toon shader is clamping colors vigorously to make it look like cartoon shading. So we pass in some angle thresholds and color values.

The second part is implementing the `ToonShaderCallBack` class, overriding the `OnSetConstants` method of the `IShaderConstantSetCallBack` interface. And we passed the necessary values to fragment the shader program through `setPixelShaderConstant` with their respective variable names.

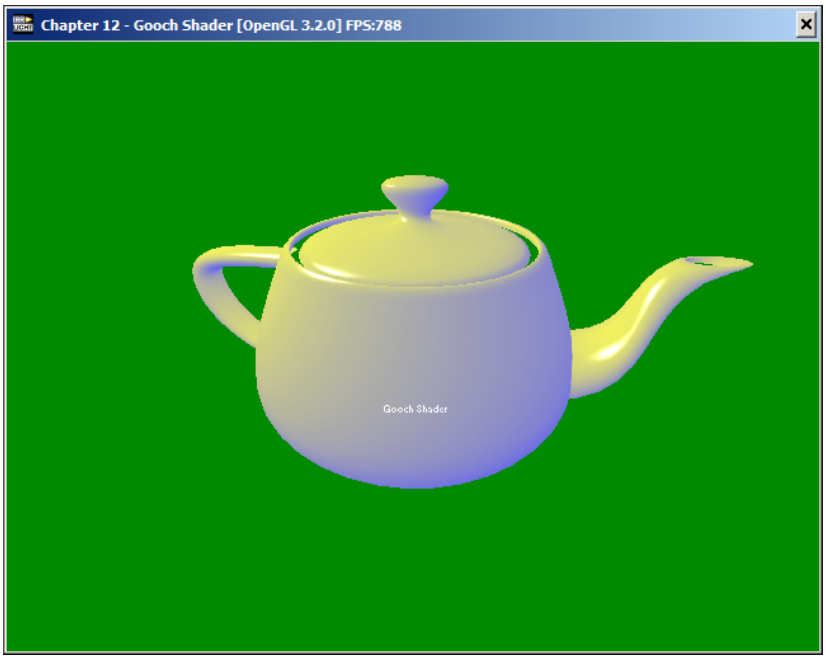
And the third part is finally using the shaders and callback class from the main function which is pretty much explained previously.

Time for action – applying a gooch shader

You can find `Gooch.vert` and `Gooch.frag` inside the project folder of this chapter. In the gooch vertex shader it takes the position of the light from the host program. So we'll need to send that from our callback as well. So implement a class called `GoochShaderCallBack` derived from `IShaderConstantSetCallBack`. Get the current camera position from our scene and pass to the vertex shader's LightPosition variable. And use the following values for respective properties from the fragment shader:

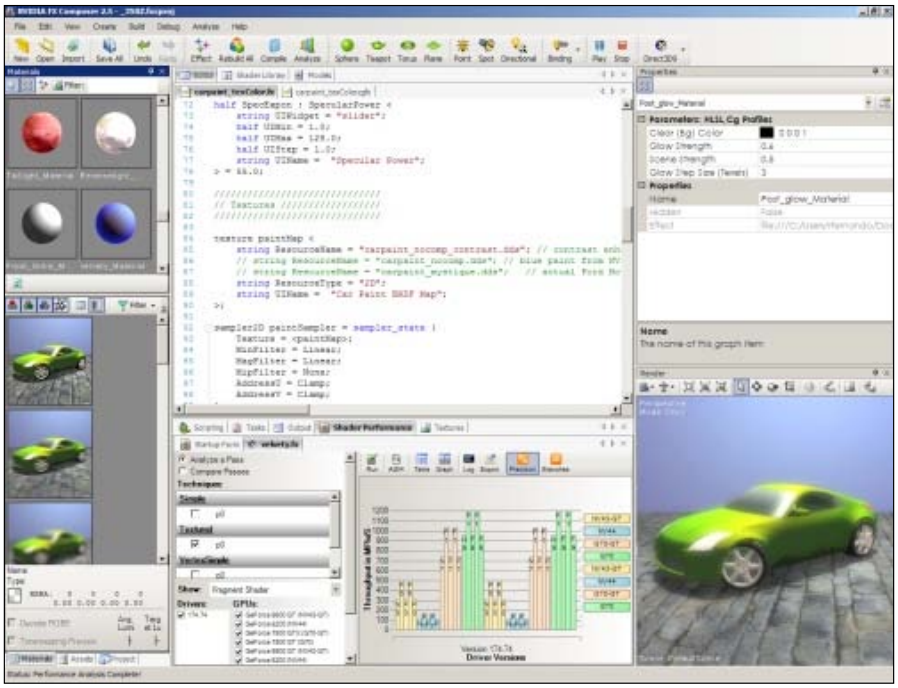
```
SurfaceColor = 0.75f,0.75f,0.75f
WarmColor = 0.6f,0.6f,0.0f
CoolColor = 0.0f,0.0f,0.6f
DiffuseWarm = 0.45f
DiffuseCool = 0.45f
```

Refer to the previous toon shader sample for the other steps. If you run this sample, you should see something like this:

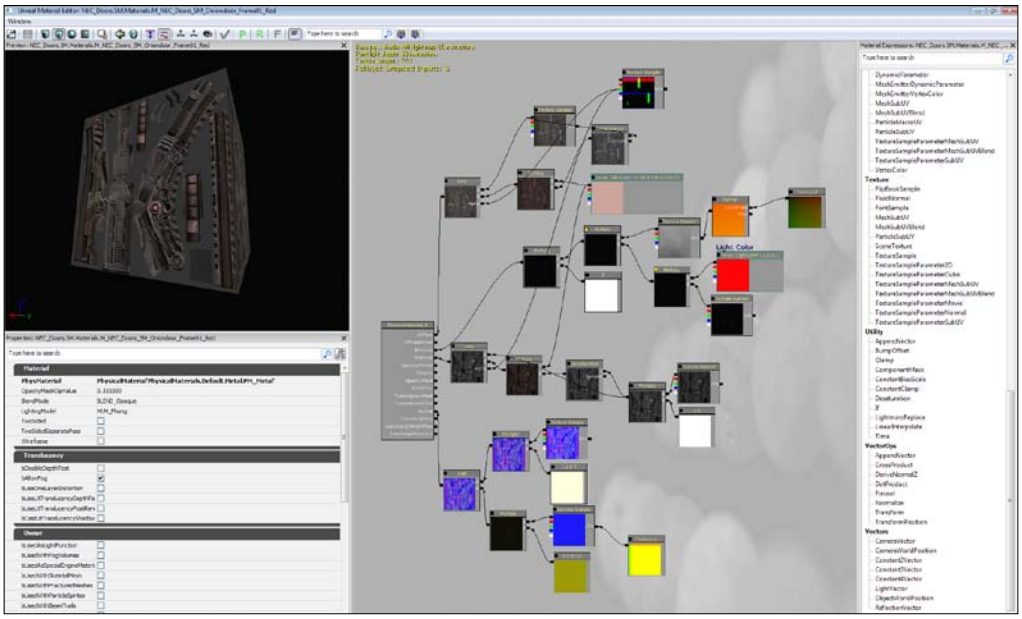


Going further with shaders

Nowadays it is pretty rare to hand code shaders like we did in this chapter. Instead there are GUI editors available to edit and test shaders in real time. For DirectX HLSL shaders, there is FX Composer which can be downloaded from `http://developer.nvidia.com/fx-composer`. OpenGL has OpenGL Shader Designer developed by TyphoonLabs and can be downloaded from `http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml`. For a quick and easy preview and edit of shader files you can use the Shader Maker from Clausthal University of Technology, `http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml`:



Among the game engines, Unreal Engine comes with one of the most powerful GUI-based material editors as shown in the following screenshot:



Summary

To summarize, we've explored the following areas in this chapter:

1. Rendering a pipeline and different types of shader
2. Introduction to OpenGL Shading Language (GLSL)
3. Using shaders in Irrlicht
4. Several tools available to design shaders

Of course, this chapter can only scratch the surface of the shaders world. You should definitely check out the Orange Book and the tutorials from `lighthouse3d.com` if you want to learn more about GLSL and try to use some other shaders together with Irrlicht. You should also have basic knowledge on Cg shaders since they are also widely used. Shaders are like the particle effects we learnt in previous chapters. It's not all about technical stuff, it's a combination of art and technology and takes time to master and combine these two different domains. Not all types of programmers will be fit to implement special effects and shaders. So don't be too distressed if you don't get all of this right away.

And finally, this is the last chapter of the book. So why don't you go ahead and create your own little game with Irrlicht? For example, a jump'n'run style game in which the player has to finish a simple course within a specified time limit and the best scores will be saved in an XML data file when the application closes and the same XML file will be loaded into the application as soon as it starts with the highest scores written on the screen before the actual game starts.

Or you could try a simple space shooter and use the space ship scene node we created earlier. Enemy ships would be quite similar to this scene node. You could use shaders to create a glow effect for ships and projectiles to give this a Geometry Wars look. You shouldn't skimp on particles: Use them for the ship's trail and when a space ship explodes.

Whatever your future endeavors with the Irrlicht 3D graphics engine might be, if your goal is to become a game programmer, I'd strongly suggest to create a complete game with menu screen, scoring, game over screen, and so on. And I wish you the best of luck and hope you enjoyed this book.

A

Deploying an Irrlicht Application

In the previous chapters, we learnt the basic aspects of Irrlicht, how to use scene nodes, meshes and shaders. In this appendix, we will take a look at how to deploy your own Irrlicht application, in case you want to distribute your application to family, friends or even customers.

In this appendix, we will:

- Learn the dos and don'ts for cross-platform deployment
- Learn how to deploy the source code

So let's get on with it...

What is meant by "deployment"?

Software deployment is a general process that can consist of different things depending on the application we would want to deploy. For example, deploying a web application is completely different from deploying a mobile application to Android, Apple mobile devices, or Irrlicht applications. Basically, what we mean by software deployment is that making our application available to be used by the actual end users. We build, compile the project in release mode, removing all the debug symbols, go through several test cases, and make the program more efficient. Deployment could be a better term for web applications where the server side scripts are physically moved from a development/production environment to a live server environment where the users would go and access the applications. But in terms of desktop applications, Irrlicht applications in particular, what we are more interested in is creating a package of our application with all the necessary files and resources so that it can be easily distributed and extracted, installed or deployed and used on another PC that may not have the same configurations we use. So let's take a look how to create such as package for different target platforms.

Deploying Irrlicht applications on Windows platforms

With Irrlicht being a dynamic library by default, the most important step is to bundle `Irrlicht.dll` with your application. `Irrlicht.dll` and your executable have to be in the same folder. In any other case, when the user starts the application, he or she will get an error message.

If you used Visual Studio to compile your application, the user will need the Visual C++ Redistributable depending on the version number you used.

Here are the links for those redistributables:

Visual C++ 2005 Redistributable: `http://www.microsoft.com/downloads/en/details.aspx?familyid=32bc1bee-a3f9-4c13-9c99-220b62a191ee&displaylang=en`

Visual C++ 2008 Redistributable: `http://www.microsoft.com/downloads/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf&displayLang=de`

Visual C++ 2010: Redistributable: `http://www.microsoft.com/downloads/en/details.aspx?familyid=A7B7A05E-6DE6-4D3A-A423-37BF0912DB84&displaylang=en`

If this redistributable is not installed and the application is launched nevertheless the application just grays out and the user receives an error message. You don't need any redistributables if you used the combination CodeBlocks with MingW as your IDE and compiler.

Although installers have received some flak in the PC world, it is still recommended to create installers for bigger projects. Installers also have the advantage to bundle additional dependencies such as the Visual C++ redistributable package.

InnoSetup (`http://www.jrsoftware.org/isinfo.php`) lets you create an all-in-one installer executable file using a descriptive language that is very similar to INI files. InnoSetup also supports a scripting language that has similarities to Pascal and Delphi.

WixEdit (`http://wixedit.sourceforge.net/`) is a setup creator as well and its handling is not unlike InnoSetup. The difference is that WixEdit creates MSI files and needs XML instead of INI files for its input.

Deploying the Irrlicht applications on Linux platforms

As there are all sorts of different Linux distributions it is really hard to determine whether the application would work as desired on the target platform. With the first chapter in mind, you might remember we needed a whole lot of development packages. Those development packages are of course not needed for end users.

Different Linux distributions use different package managers, apt-get for Ubuntu, yum for Fedora, zypper for OpenSUSE to name a few. Creating a package for each of these distributions can be a lot of work that would go beyond the scope of this chapter. It is sufficient if you create a package out of the executable and all the assets that are needed.

The most important thing to remember is that end users need to have the latest and hardware-accelerated OpenGL drivers installed or your application will either not start or have terrible performance.

Start helper script for Linux

It is usually more common to ship 32-bit and 64-bit executables of your application in the Linux world than it is on the Windows platform. To ease the user experience, we will create a simple bash start script that automatically starts the correct executable, instead of the user having to pick the correct executable for himself or herself. In this case, our compiled 32-bit executable is called `myExecutable-32` and our 64-bit executable consequently `myExecutable-64`:

```
#!/bin/sh

KERNEL=`uname -m`

if [ ${KERNEL} = "x86_64" ]
then
  # 64 bit
  ./myExecutable-64
else
  # Assume 32 bit
  ./myExecutable-32
fi
```

The first line is needed for this script to be executed as a bash script. After you saved this script file on your hard drive you may need to execute the command `chmod a+x ./myFile` where `myFile` stands for the filename you have chosen. Try to choose a simple name so the user immediately understands the intent of this script like `start.sh` or `launch.sh`.

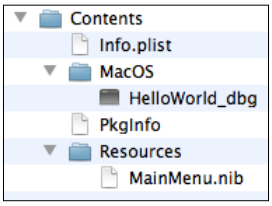
We store the result of the command `uname -m` in the variable `KERNEL`, which is the architecture. Even with an i386-compatible processor, there are still different architecture types for this platform. Netbooks, for example, have the architecture type **ia32**.

The command `arch` is equivalent to `uname -m`. Checking for the `/lib64` folder is not recommended because some applications may use this folder even when you are running a 32-bit kernel.

If you are running the script on an unsupported architecture by the script such as ARM or PowerPC Linux systems, those two applications won't run of course. If you are planning on targeting those platforms, you need to change this script accordingly.

Deploying for Mac OS X platforms

If you have some experience with Mac OS X you probably already know that Mac OS X application with its `*.app` file extension are basically just folders. If you right-click on these application bundles and select **Open Contents** you can view what exactly is in one application bundle:

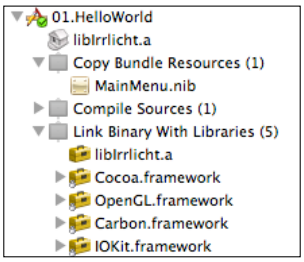


Any application is more or less structured like this example. `Info.plist` is basically an XML file that contains information about the application itself such as its name, its version, its icon, the company name, and language.

`PkgInfo` is a simple text file where the first four characters are APPL and the last four characters the abbreviation of your application. If no application abbreviation has been specified, the second half of the file are just question marks. When using the Xcode IDE, this file will be automatically updated if the key `CFBundleSignature` in `Info.plist` has been changed.

If you are familiar with Objective-C and Cocoa development, you might already know what `*.nib` files are: These files are usually created by the Xcode form designer and contain the visual components of Cocoa applications such as an application form with buttons, input boxes, or similar user input fields. In the case of an Irrlicht application, and as the name already suggests, this nib-file consists of just the main menu.

If you compiled your Irrlicht application with Xcode, the IDE takes care of keeping the application bundle up to date. Just make sure your assets such as models, icons or graphics are listed **Copy Bundle Resources**. If this is not the case, your application won't display your assets:



Creating universal applications and compatibility

By default, applications are compiled for the current version of the operating system. Getting this executable to work on other versions might be a hassle: While running a Leopard-compiled application on Snow Leopard is usually not an issue, it can be the other way round. To ensure that the compiled application runs on earlier versions of Mac OS X you need to do two things. First of all an earlier version of the Mac OS X SDK needs to be installed on the hard drive which you have as an option when installing Xcode. Then, open the project settings, go to the **Build** tab and set **Base SDK** to the SDK you want, for example, Mac OS X 10.4 SDK.

Since 2005, Apple computers ship with an Intel processor, before that every Mac since 1994 had a Motorola or IBM PowerPC processor. Even though Intel Macs dominate nowadays, there are still a number of PowerPC Mac users out there. To make sure PowerPC Macs can run your applications you need to create a PowerPC executable. Apple has introduced the concept of universal applications, which is basically one executable that contains executables for different architectures. To set a universal application up, go to the project settings and look for the item **Architectures** under the **Build** tab. This field takes one or more of the following values:

- ppc (For 32-bit PowerPC Macs such as PowerMac G4, iBook, etc.)
- ppc64 (For 64-bit PowerPC Mac such as PowerMac G5)
- i386 (For 32-bit Intel Macs with a Core Duo, Core 2 Duo, i3, i5, i7 or Xeon processor)
- x86_64 (For 64-bit Intel Macs with a Core 2 Duo, i3, i5, i7 or Xeon processor)

Depending on which values you picked, the resulting executable contains the binary code for those architectures. If you are working on a PowerPC Mac, you cannot compile Intel binaries.

If you plan to release your application for the Mac App Store, you have to keep in mind that universal binaries containing a PowerPC executable are not allowed, only Intel Mac-compatible applications (32-bit, 64-bit or both) will get approved.

Deploying the source code

Just to be complete here is a quick how-to on deploying the source code of our Irrlicht application.

Why deploy the source code?

If you want to release your application as open-source software, you would want to create an additional package for users who would want to look into how your application has been made or recompile the source code for themselves.

The other case might be that you are working in a team and have to exchange source files. In this case, it is strongly recommended to use version control software though.

Deploying the source

Make sure you just have plain text files, such as C/C++ headers (`*.h`), C++ source files (`*.cpp`), project files such as `*.sln` for Visual Studio or `*.cbproj` for CodeBlocks. Just use a packager of your choice and create a compressed package of all files necessary. In the Linux world, source code packages usually have a `*.tar.gz` file extension.

If you release your application as open-source software, you should also include a full text version of the license of your liking in your package file. If Irrlicht code is in the package, it also requires the zlib license to be included, and probably also all other license files that are contained in the Irrlicht SDK.

Summary

We learned some techniques about software deployment and what we need to know if we deploy our own Irrlicht application.

Specifically, we covered:

- Definition of deployment
- What is important to keep in mind when deploying your application for different platforms

B
Pop Quiz Answers

Chapter 2, Creating a Basic Template Application

Question	Answer
1	a
2	b

Chapter 3, Loading Meshes

Question	Answer
1	a
2	b

Chapter 4, Overlays and User Interface

Question	Answer
1	a Sprite sheet width should be completely divisible by number of columns and height by rows.
2	b OnEvent method of the class which implements IEventReceiver interface will be executed.

Chapter 5, Understanding Data Types

Question	Answer
1	b
2	a
3	b
4	c
5	c
6	b

Chapter 7, Using Nodes—The Basic Objects of the Irrlicht 3D Engine

Question	Answer
1	b For example, nodes like camera are not rendered.
2	b Animators are for basic manipulations of scene nodes and not just for animated mesh scene node.

Chapter 12, Using Shaders in Irrlicht

Question	Answer
1	c
2	b

Index

Symbols

A

B

C

D

E

F

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Thank you for buying

Irrlicht 1.7 Realtime 3D Engine Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

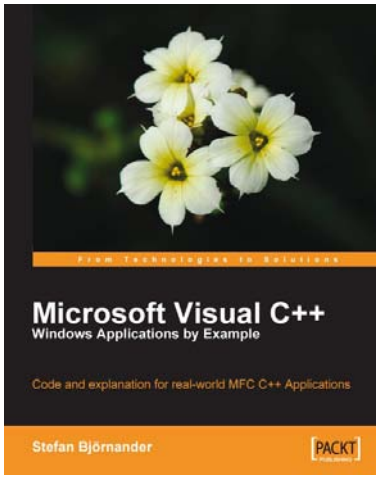
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



Microsoft Visual C++ Windows Applications by Example

ISBN: 978-1-847195-56-2 Paperback: 440 pages

Code and explanation for real-world MFC C++ Applications

1. Learn C++ Windows programming by studying realistic, interesting examples
2. A quick primer in Visual C++ for programmers of other languages, followed by deep, thorough examples
3. Example applications include a Tetris-style game, a spreadsheet application, a drawing application, and a word processor

---

Blender 2.5 HOTSHOT

ISBN: 978-1-84951-310-4 Paperback: 332 pages

Challenging and fun projects that will push your Blender skills to the limit

1. Exciting projects covering many areas: modeling, shading, lighting, compositing, animation, and the game engine
2. Strong emphasis on techniques and methodology for the best approach to each project
3. Utilization of many of the tools available in Blender 3D for developing moderately complex projects
4. Clear and concise explanations of working in 3D, along with insights into some important technical features of Blender 3D

Please check **www.PacktPub.com** for information on our titles