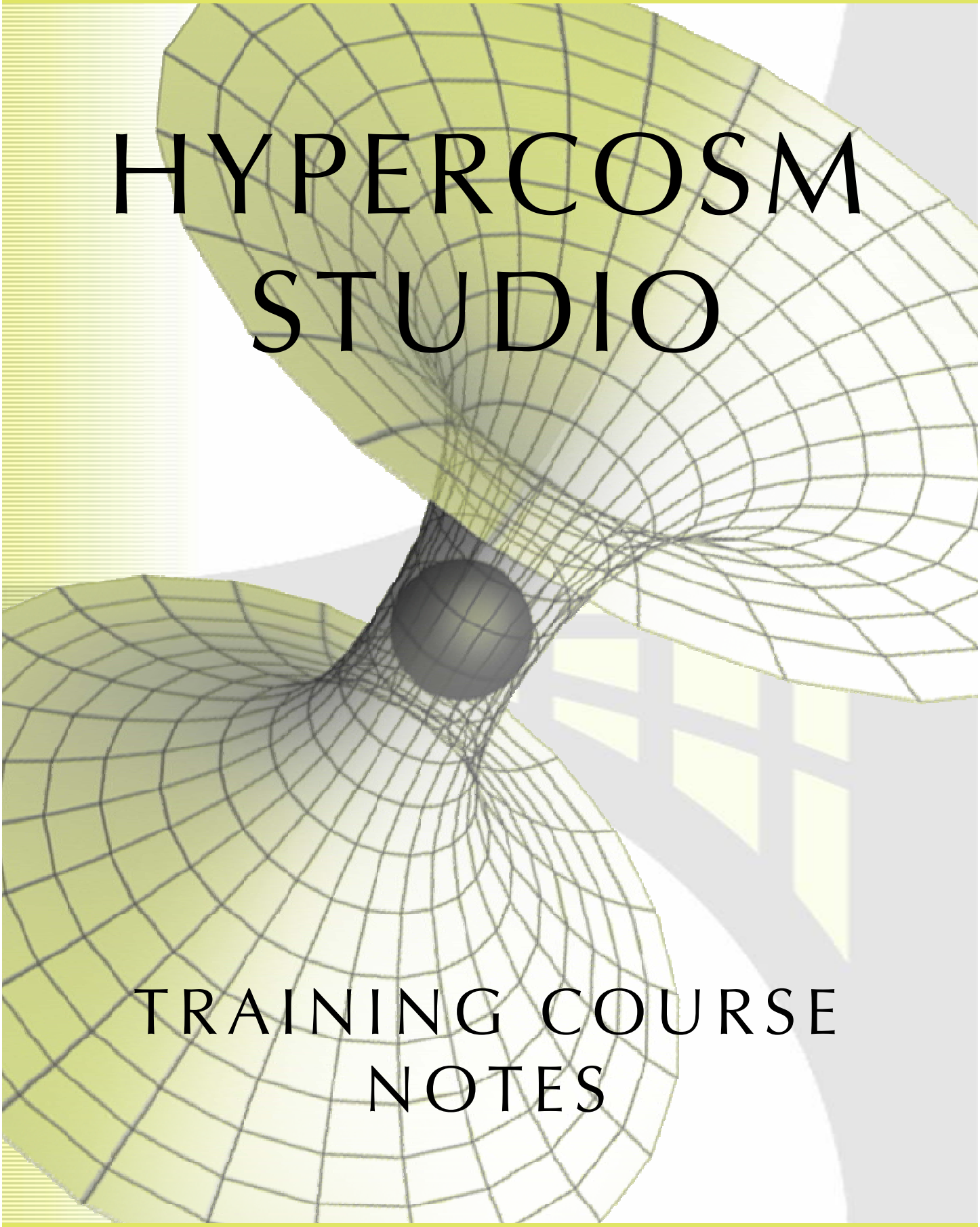
HYPERCOSM LLC



HYPERCOSM STUDIO

TRAINING COURSE NOTES

WWW.HYPERCOSM.COM

HYPERCOSM STUDIO TRAINING COURSE NOTES

HYPERCOSM LLC

1212 FOURIER DRIVE

MADISON, WI

53717

WWW.HYPERCOSM.COM

AUGUST 2008

TABLE OF CONTENTS

LESSON 1:
HYPERCOSM OVERVIEW

LESSON OBJECTIVES:

- Learn the advantages of using the Hypercosm system for creating and displaying 3D content
- Understand the range of 3D tools available to you from Hypercosm.
- Understand how Hypercosm compares with other media types that are currently available (Java, video, Flash, Macromedia, etc.)

LESSON CONTENTS:

- What Is Hypercosm
- Hypercosm System Benefits
- Hypercosm System Components
- Hypercosm Content Creation Workflow
- Hypercosm File Types

COPYRIGHT©2006

WHAT IS HYPERCOSM?

The Hypercosm system is a system for creating and deploying interactive 3D simulations. These simulations can be used for a variety of applications including training, scientific visualization, and education.

THE SPECTRUM OF 3D CREATION TOOLS

The Hypercosm system fills a gap which exists between visual graphical content creation programs (such as 3ds Max™) which are relatively easy to use but don't allow behavior and interactivity to be specified and graphics libraries (such as OpenGL) which can be very powerful but require extensive programming skills to use.

CATEGORY 1: CAD SOFTWARE

There currently are a fairly wide variety of 3D computer aided design (CAD) packages available that allow to you create a wide variety of 3D models. However, these 3D models are static and don't have any animation or behavior associated with them. They are like 3D paperweights.

Examples:

- AutoCAD™, SolidWorks™, ProE™, Catia™

Benefits:

- Graphical 3D modeling
- Tools are relatively mature and standardized

Disadvantages:

- No animation, interactivity, or simulation capabilities
- Useful for design, but not so useful for training, education, or communication

CATEGORY 2: 3D MODELING AND ANIMATION SOFTWARE

3D modeling and animation packages allow you to created 3D models and also to specify complex animation. These packages also typically allow you to define complex rendering and surface attributes that make these models highly realistic. Although these software packages allow you to define complex animation, to share the animated graphics with others it must be rendered out to video. This eliminates the ability to interact with the 3D models, which is one of the strongest benefits of 3D graphics.

Examples:

- Alias Maya™, Autodesk's 3ds Max™, NewTek's Lightwave™

Benefits:

- 3D animation
- Allow definition of realistic surface attributes and materials

Disadvantages:

- No interactivity or simulation capabilities
- Output is video - files are huge, unwieldy to send electronically

CATEGORY 3: 3D GRAPHICS AND SIMULATION APIS

Graphics libraries or higher level "scene graph" libraries allow you to write programs to bring your shapes and objects to life, but the programming must be done in a very tedious and low level language such as C. This makes programming 3d graphics unattainable to all but experienced professionals and impractical for most applications because it would take too much work even for a pro to be worth the effort.

Examples:

- OpenGL™, Direct3D™, Open Inventor™, Multigen-Paradigm™, GL Studio™

Benefits:

- True 3D simulation
- Unlimited interactivity

Disadvantages:

- Content creation is difficult, time consuming, and expensive
- Output is executable programs, which are not well suited for distributing over the web for reasons related to security and file size

CATEGORY 4: HYPERCOSM RAPID 3D APPLICATION DEVELOPMENT SYSTEM

At the core of Hypercosm is a set of sophisticated rendering software that is controlled by the user through a simple interpreted graphics language. This high level scripting language shields the user from having to deal with the complexities associated with the graphics and also makes it possible to safely distribute the simulations over the Internet. This system makes it possible to create simulations that have all of the complex behavior and interactivity of the 3D graphics and simulation APIs, but in a form that is practical to create and deploy. Probably the most similar software to Hypercosm which currently exists is VRML 2.0. VRML 1.0 is a static scene description language with no animation or dynamic abilities. VRML 2.0 has added the ability to create dynamic content but through an external scripting language. Hypercosm has integrated these two functions into one system.

HYPERCOSM SYSTEM BENEFITS

The Hypercosm system allows you to publish 3D content just like you would publish other multi media content such as images, video or sound. Hypercosm content is a powerful web-based medium for interactive online communication of 3D visual information that can be viewed by a wide audience. The Hypercosm system uses a unique technique for building and deploying 3D simulation based content that has the following specific benefits.

BENEFITS FOR CONTENT CREATION

- **Interactivity**
  Rather than viewing static images, the viewer can navigate through a 3D scene interactively and gain a much better understanding of the content.

- **Flexibility**
  The real world is complex. Hypercosm's powerful scripting language makes it possible to create 3D simulations that not just look like the real thing, but also act like the real thing. With Hypercosm, you can add unlimited interactivity to your 3D scenes.

- **Rapid Application Development**
  Using a high-level script language that has been designed for 3D simulation allows complex content to be created much faster than conventional techniques (e.g. using low-level system programming languages such as C or C++).

BENEFITS FOR CONTENT DISTRIBUTION

- **Low File Size**
  Hypercosm's patented approach to encoding 3D content results in low file sizes can be delivered over the web in a practical and effective manner.

- **Safety / Security**
  Because the script code that drives Hypercosm applets is interpreted, it is inherently safe. You can run Hypercosm applets without any fears that they may contain viruses, worms, or other malicious content.

- **Encryption**
  Models and animations are compressed and encrypted into a form that is safe to post on a website and cannot be changed or edited in any way.

HYPERCOSM SYSTEM COMPONENTS

The Hypercosm system is broken down into two essential families of software components: viewing and development. The viewing software is the Hypercosm Player, which is used to display 3D content that is in Hypercosm format. The development software is Hypercosm Teleporter, which translates 3D content from 3ds Max ™ to Hypercosm format and Hypercosm Studio, which is a scripting tool that allows developers to program custom behaviors for Hypercosm applets.

HYPERCOSM PLAYER

The Hypercosm Player (see Figure 1.1) is free software that allows users to interactively view 3D content. By viewing the 3D content in Hypercosm Player the user can navigate the 3D scene by rotating, panning, and zooming. The user is also able to change the rendering modes (smooth shading, flat shading, or wireframe) of the 3D scene. Hypercosm content can also have many other sophisticated interactions and behaviors such as the simulation of physics.

The Hypercosm Player can be used to publish 3ds Max™ models that have been exported with Hypercosm Teleporter. Hypercosm Player can be integrated with a web browser (Internet Explorer, Firefox, etc.) or as a standalone 3D player.

Figure 1.1: Example of Hypercosm Player

HYPERCOSM TELEPORTER

Hypercosm Teleporter (see Figure 1.2) translates 3D model and animation information from an existing 3D authoring tool, such as Autodesk's 3ds Max™, and transforms it into a web deployable Hypercosm 3D applet that can be viewed using the Hypercosm Player. When Hypercosm Teleporter exports a 3D scene, it encrypts and compresses the 3D information so the 3D content can be shared in an efficient and effective manner. By using Hypercosm Teleporter the user is able to share his or her 3D content with anyone that has the Hypercosm Player installed.

Figure 1.2: Export to Hypercosm Teleporter

HYPERCOSM STUDIO

Hypercosm Studio (see Figure 1.3) is a scripting tool used by Hypercosm developers to add complex behavior, interactivity, and even physics to Hypercosm 3D content. Using Hypercosm Studio, you can program directly in Hypercosm's high level, 3D graphics oriented OMAR (Object Oriented Modeling, Animation, and Rendering) language to add high level functionality relatively quickly and easily.

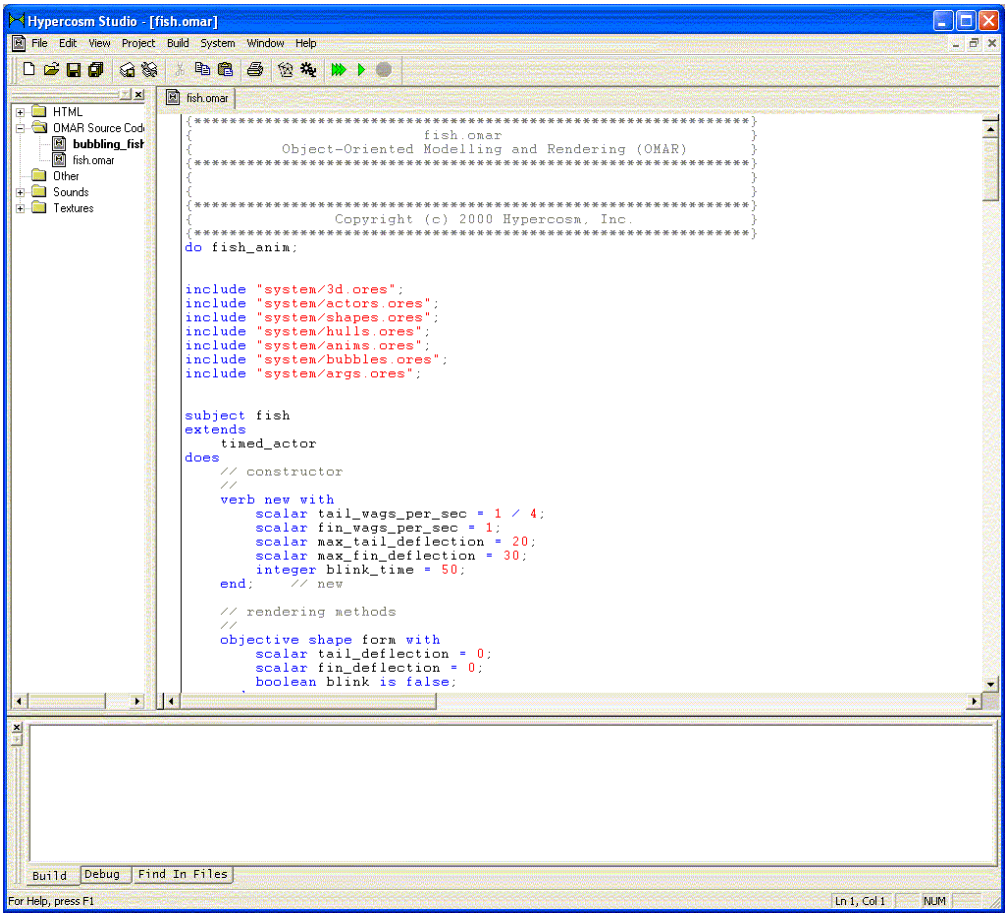


Figure 1.3: Example of Hypercosm Studio

HYPERCOSM CONTENT CREATION WORKFLOW

Each content creation project has its own unique aspects and challenges, but all tend to follow a standard workflow as described below.

STEP 1: BUILD

The first step is to build the 3D models that will be used in the simulation. This is done using COTS 3D modeling software. We encourage content creators to use Autodesk's 3ds Max™ software because Hypercosm's tools (Hypercosm Teleporter, in particular) have been optimized to work very efficiently with models created using this software.

STEP 2: EXPORT

Next, Hypercosm Teleporter is used to export the scene from the COTS modeling software into Hypercosm's OMAR script code. This script code represents the entire underlying representation of the 3D scene. By modifying this script code, you can change any aspect of the scene including the shapes of objects, the materials applied to surfaces, or the animation of objects.

STEP 3: SCRIPT

Once you have access to the script code that represents the scene, you can add your own additional script to add complex elements that can't be created using the COTS modeling tools that are used to create the models and animations. These sorts of elements include user interactions, physics, and sounds.

Hypercosm File Types

In the process of creating Hypercosm content, you will encounter and use four new file types that are unique to the Hypercosm system. These file types include the following:

HYPERCOSM SCRIPT FILES

There are two types of Hypercosm script files: ".OMAR" files and ".ORES" files. Both files contain text written in the OMAR language. ".OMAR" files contain a "main" section that defines a starting point for actually running the program. ".ORES" files are "OMAR Resource" files that contain script code that can be used by other ".OMAR" and ".ORES" files but can't be run by itself. These files contain a wide variety of utility functions and 3D shape definitions.

Figure 1.4: Hypercosm Script (.OMAR) File Icon

Figure 1.5: Omar Resource (.ORES) File Icon

HYPERCOSM STUDIO PROJECT FILES

Hypercosm Studio Project (.HSP) Files are used to keep track of the various files that may be used in a Hypercosm project. A project may include a number of OMAR and ORES script files along with a number of resources. These resources include graphics files for textures, sounds, and text. Also, the web page that contains an applet may also reference Flash or Shockwave files. The Hypercosm Studio Project File contains links to all of the files used in a project and gathers together these files whenever a project is compiled to a web page.

Figure 1.6: Hypercosm Studio Project (.HSP) File Icon

HYPERCOSM APPLET FILES (.HCVM)

Hypercosm applet (.HCVM) files are the files that are actually distributed over the web and run by the Hypercosm Player. These files are designated by the .HCVM file extension which stands for "Hypercosm Virtual Machine", the software engine that is needed to run these files. Hypercosm applet files contain a compressed or "compiled" version of the script code that is contained in the .OMAR and .ORES files that are used to create a .HCVM file. The applet file has been stripped of all symbolic information contained in the script code such as variable and type names and will therefore be many times smaller than the set of .OMAR and .ORES files that go into building the applet file. Since all human readable information has been stripped out of the applet file, applet files are similar to binary executable files since they no longer are human readable but can be executed by a computer.

Figure 1.7: Hypercosm Applet (.HCVM) File Icon

LESSON 2: RUNNING HYPERCOSM APPLETS

LESSON OBJECTIVES:

- Know how to run Hypercosm applets
- Understand how Hypercosm applets are structured
- Understand the standard Hypercosm applet user interface

LESSON CONTENTS:

- Installing the Hypercosm Player
- Running Hypercosm applets
  - Hypercosm applet file structure
  - 3D Scene Navigation
  - Hypercosm Applet Dock Bar
  - Standard Controls
- Hypercosm Control Panel

INSTALLING THE HYPERCOSM PLAYER

The first thing that you need to do before Hypercosm applets can be run is to install the Hypercosm Player. The Hypercosm Player is freely downloadable from the Hypercosm web site at: www.hypercosm.com/download/player . The Hypercosm Player is freely distributable so you can distribute it along with your content if you wish. If you are distributing the content via CD, it's often a good idea to include the Hypercosm Player along with the content since the user may not necessarily be connected to the Internet. If you know that the user of the content will be connected to the Internet, then the content may provide a link to the current version of the player on Hypercosm's web site. This will ensure that the user installs the most up-to-date version of the Hypercosm Player.

CHECKING TO SEE IF THE HYPERCOSM PLAYER IS INSTALLED

In order to check to see if the Hypercosm Player is currently installed on your system, you can consult your system's control panel. To do this, follow these instructions:

1) **Open the System Control Panel**
   To open the control panel click the "Start" button and then select the "Settings" menu item as shown below and select the "Control Panel" menu option to the right.

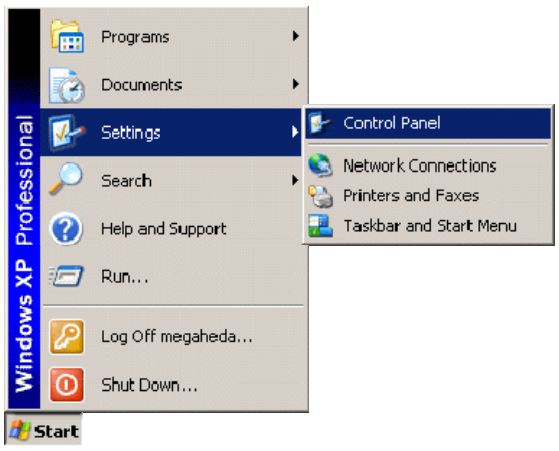


Figure 2.1: The Windows Settings Menu

2) Select "Add or Remove Programs"

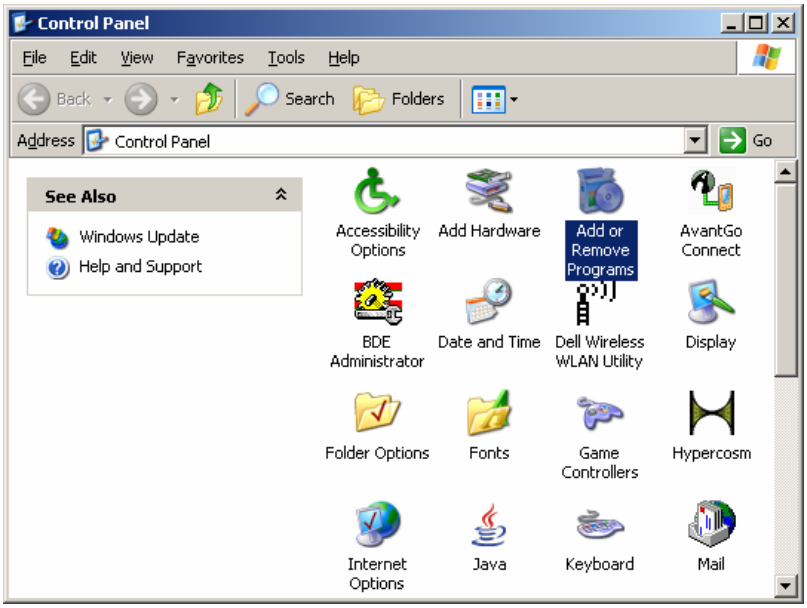


Figure 2.2: The System Control Panel

3) Search for "Hypercosm Player" in the list of installed programs
If Hypercosm Player is installed, it will appear as shown below in the list of installed programs.

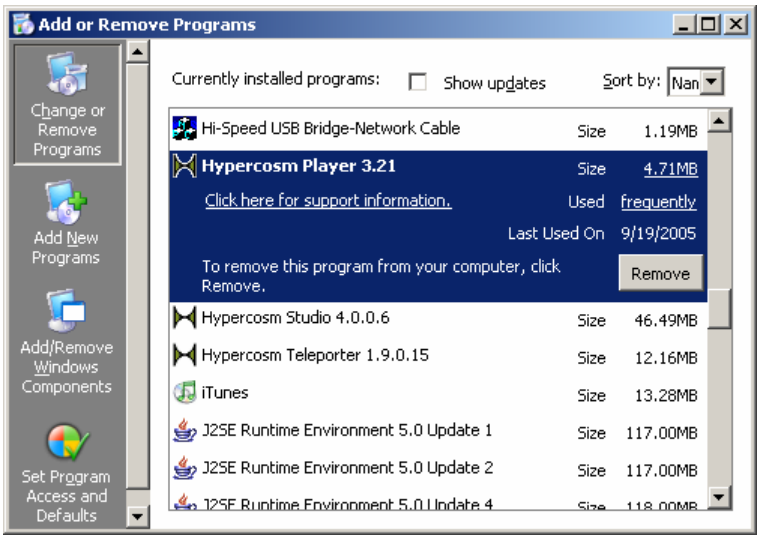


Figure 2.3: The Add or Remove Control Panel

RUNNING HYPERCOSM APPLETS

Once the Hypercosm Player is installed, running Hypercosm applets is easy. There are two ways to run Hypercosm applets.

1. **Directly**
   To run the Hypercosm applet directly, simply double-click on the applet (.HCVM) file. This will start the Hypercosm Player, which will open up a window containing the graphics displayed by the Hypercosm applet. To stop the Hypercosm applet, simply close the window.

2. **Indirectly**
   The second way to run a Hypercosm applet is by displaying a document containing a Hypercosm applet. Most often, this is a web page. When the web page is loaded, the Hypercosm Player will automatically start and display the applet. When you close the web page, the Hypercosm Player will also shut down.

HYPERCOSM APPLET FILE STRUCTURE

Hypercosm content is often composed of a collection of different files that are all needed to describe a 3D scene or simulation. These files include the following:

1) **The Applet (.HCVM) File**
This is the main file that contains the Hypercosm code defining the geometry in the scene and the scene behavior. This file is normally played by opening up the web page accompanying the applet in a standard web browser such as Internet Explorer, Firefox, or Netscape. The applet file may also be double-clicked on and run by itself. This will cause the applet to run and display a simulation in its own window instead of inside of a frame in a web page.

2) **A Web Page (.HTML) File**
This is a standard HTML file that describes the web page containing the Hypercosm applet.

3) **Texture (.JPG, .JPEG, .GIF, or .PNG) Files (Optional)**
These are files used to map details on the surfaces of objects in the 3D scene.

4) **Sound (.WAV, .MP3) Files (Optional)**
These are sounds that may be played by the simulation.

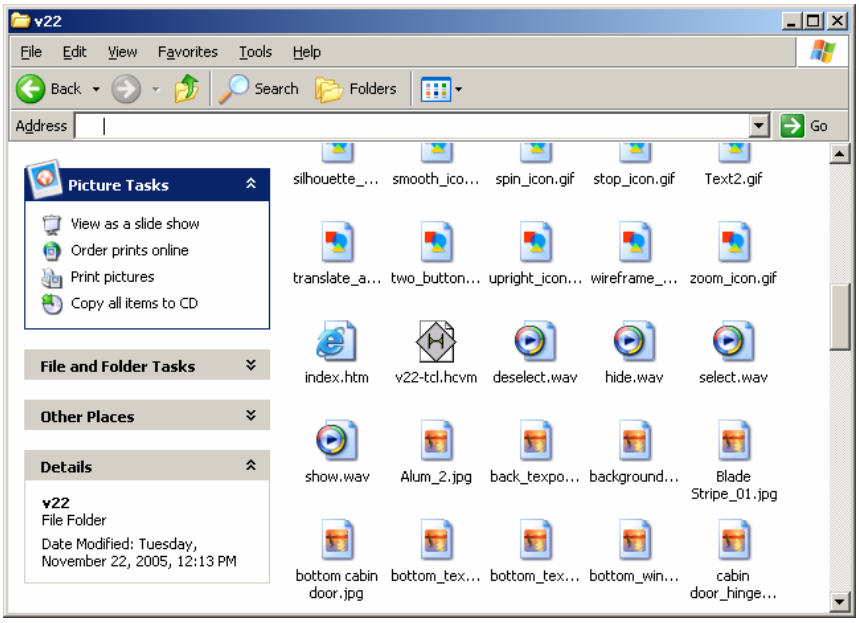


Figure 2.4: An Example of a Typical Folder Containing a Hypercosm Applet

3D SCENE NAVIGATION

When viewing 3D content with the Hypercosm Player the user can use the mouse to move around the 3D space. The standard mouse controls allow the user to rotate, pan, and zoom.

ROTATE

To make the scene rotate, click and hold the left mouse button and drag the pointer horizontally or vertically in the direction you want the scene to rotate.

Figure 2.5: Left button to rotate

PAN

To change your direction of view, click and hold the right mouse button and drag the pointer in the direction that you want to your view to move.

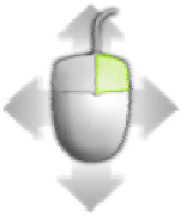


Figure 2.6: Right button to pan

ZOOM

To zoom in or out of the scene, click and hold both mouse buttons (or the middle mouse button), while you drag the pointer towards you to zoom in or away from you to zoom out.

Figure 2.7: Both buttons to zoom

STANDARD HYPERCOSM APPLET DOCK BAR

The applet docking bar is displayed at the bottom of the running applet and is the place where the applet controls are kept.

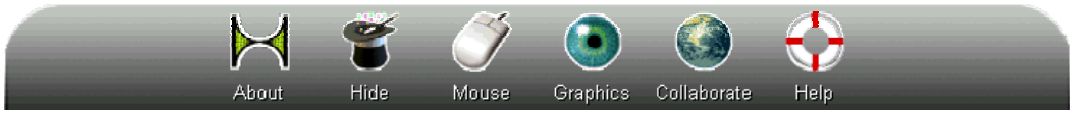


Figure 2.8: Applet Docking Bar

STANDARD CONTROLS

Hypercosm applets using the standard docking bar may contain any of the icons shown below. Applets may be also be configured to omit or add any of these controls, but most standard applets will contain this full set of controls.

ABOUT ICON

This icon can be clicked to cause the web browser to bring up the Hypercosm web page. The Hypercosm web page is a source of information about the various software tools that are available to create and display Hypercosm content and also provides useful support and troubleshooting information.

Figure 2.9: The About Icon

HIDE ICON

This icon toggles the docking bar's auto-hiding feature. When enabled, auto-hiding will cause the dock to disappear when the user moves the cursor away from the dock. When the user moves the cursor back to the location of the dock, the dock will reappear. If the auto-hiding feature is not selected, then the dock will always be displayed.

Figure 2.10: The Hide Icon

MOUSE ICON

This icon can be clicked to display a set of controls for setting the number of mouse buttons to use. Normally, the controls are configured to use a two button mouse. If the user wants a one button interface, then this control allows the applet to be configured to use just a single mouse button.

Figure 2.11: The Mouse Icon

GRAPHICS ICON

This icon can be clicked to bring up a set of controls for configuring the way the 3D graphics are drawn. This is where the user can change the rendering mode of the scene (smooth shaded, flat shaded, wireframe, etc).

Figure 2.12: The Graphics Icon

COLLABORATION ICON

This icon can be clicked to display the collaboration controls. The collaboration controls can be used to configure an applet to run in synchronization with another applet running on another machine on the same network. Hypercosm collaboration allows peer-to-peer collaboration (one person to one other person) across a network without firewalls. The other user may be located across the hall or across the world. If an IP address for the other user is provided in the web page, then the applet will connect with that user. Otherwise, the applet will try to connect with another user on the same subnet. The collaboration controls allow the two users to switch off being the "leader"

and the "follower". When the user is playing the "leader", then they control the view for themselves and for the other user. When the user is playing the "follower", then they see scene from the point of view of the leader and have no control over the view.

Figure 2.13: The Collaborate Icon

HELP ICON

This icon can be clicked to bring up information about how to use the Hypercosm applet and the various applet controls.

Figure 2.14: The Help Icon

THE HYPERCOSM CONTROL PANEL

On occasion, you may find it necessary to check to see what version of the Hypercosm Player you are running or possibly to change the Hypercosm Player's configuration. You can do this by using the Hypercosm Player Control Panel.

OPENING THE HYPERCOSM CONTROL PANEL

In order to open up the Hypercosm Control Panel, follow the steps shown below:

1) **Open the System Control Panel**
   To open the control panel click the "Start" button and then select the "Settings" menu item as shown below and select the "Control Panel" menu option to the right.

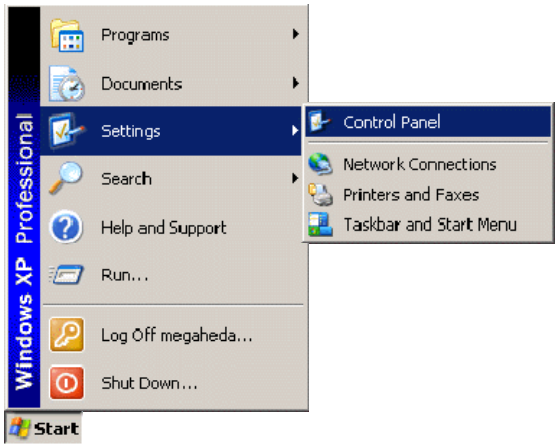


Figure 2.15: The Windows Settings Menu

2) **Open the Hypercosm Control Panel**
Once the control panel window is open you should see a window that is similar to the one pictured below. One of the icons in this window will be a Hypercosm logo the label "Hypercosm". Double click this icon to bring up the Hypercosm control panel.

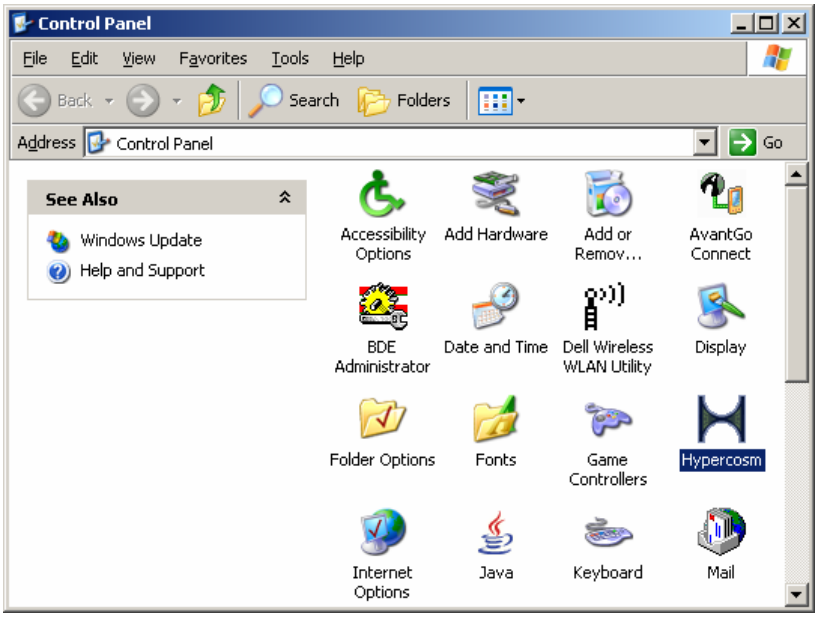


Figure 2.16: The System Control Panel

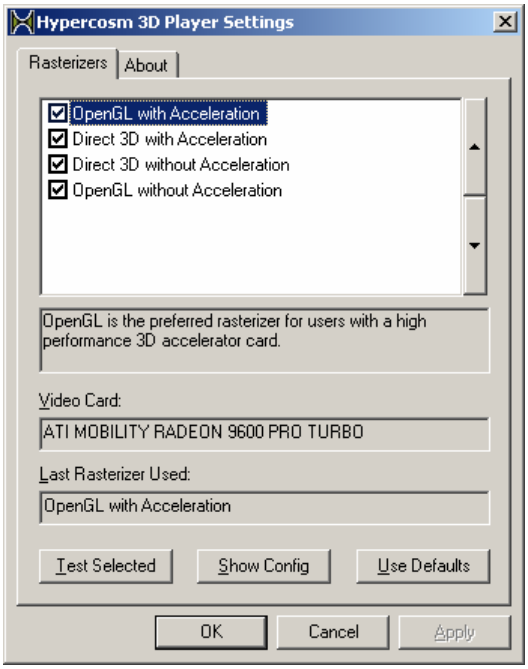


Figure 2.17: The Hypercosm Control Panel

FINDING THE HYPERCOSM PLAYER VERSION NUMBER

To find out what version of the Hypercosm Player you are using, click the "About" tab at the top of the Hypercosm Control Panel. The version number is listed at the top of the text box.

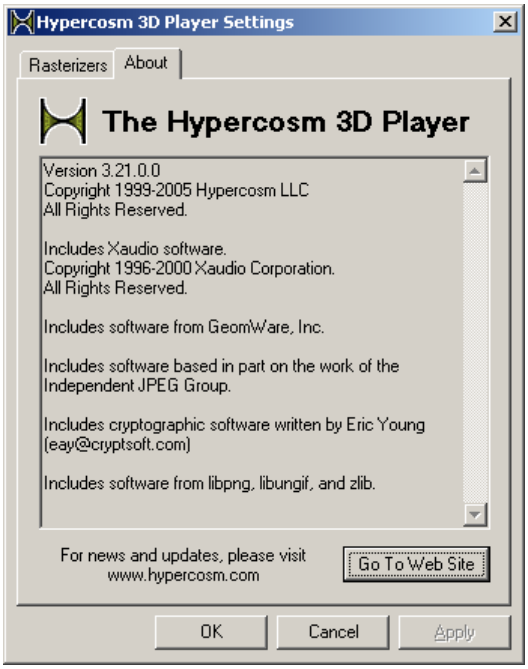


Figure 2.18: The Hypercosm Control Panel About Tab

LESSON 3: HYPERCOSM STUDIO INTRODUCTION

LESSON OBJECTIVES:

- Become familiar the Hypercosm Studio user interface
- Understand how to open, create, edit, and save files
- Know how to change system settings

LESSON CONTENTS:

- User Interface Layout
- Opening Files
- Code Editing
- System Settings

USER INTERFACE LAYOUT

Hypercosm Studio's user interface is similar to other code editing integrated development environments (IDE's) for writing text based code. It is composed of three main panels and a toolbar as shown below:

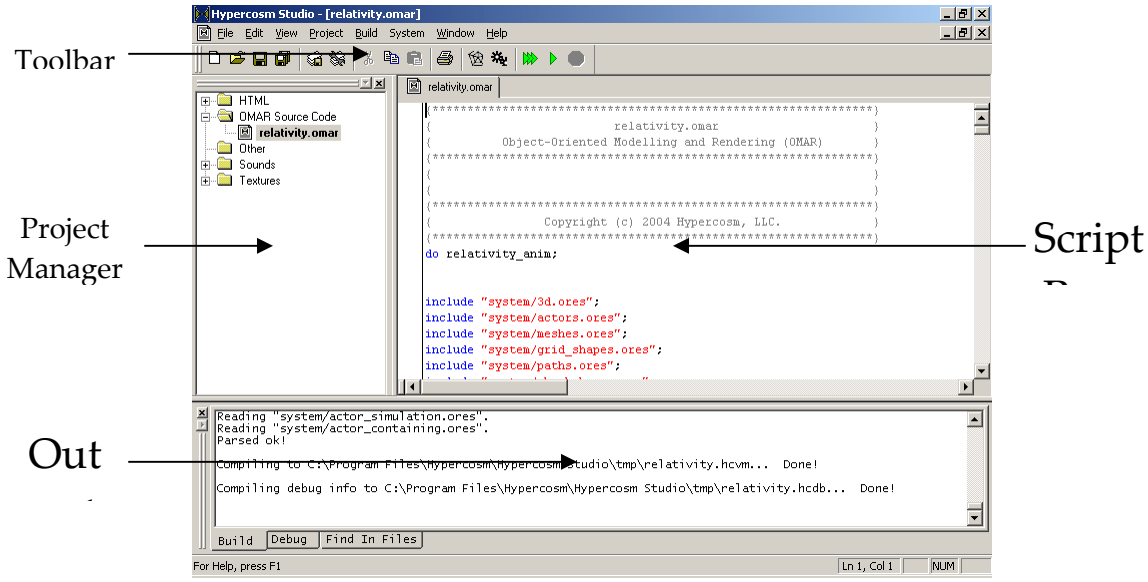


Figure 4.1: The Hypercosm Studio User Interface

THE TOOLBAR

The toolbar contains a set of shortcut keys to activate commonly used tools. These tools are described in more detail in the following sections.

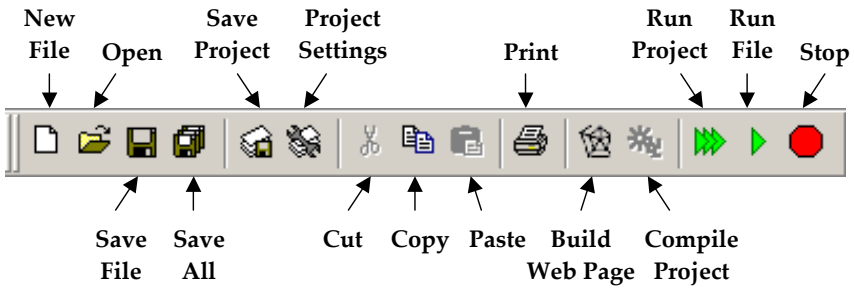


Figure 4.2: The Toolbar

THE PROJECT MANAGER PANE

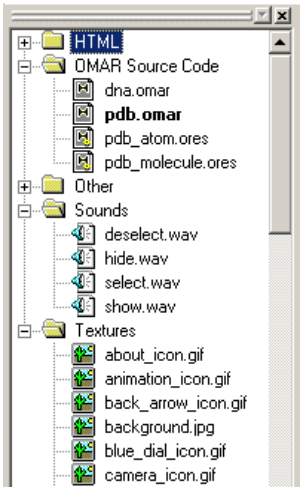


Figure 4.3: The Project Manager Pane

The project manager pane is the tall, thin pane on the upper left side of the window. This panel has a list of the files in the project and allows you to easily navigate and open files belonging to the project. The files that are listed here include script files, texture files, sound files, web pages, text files, and other project resources.

THE SCRIPT PANE

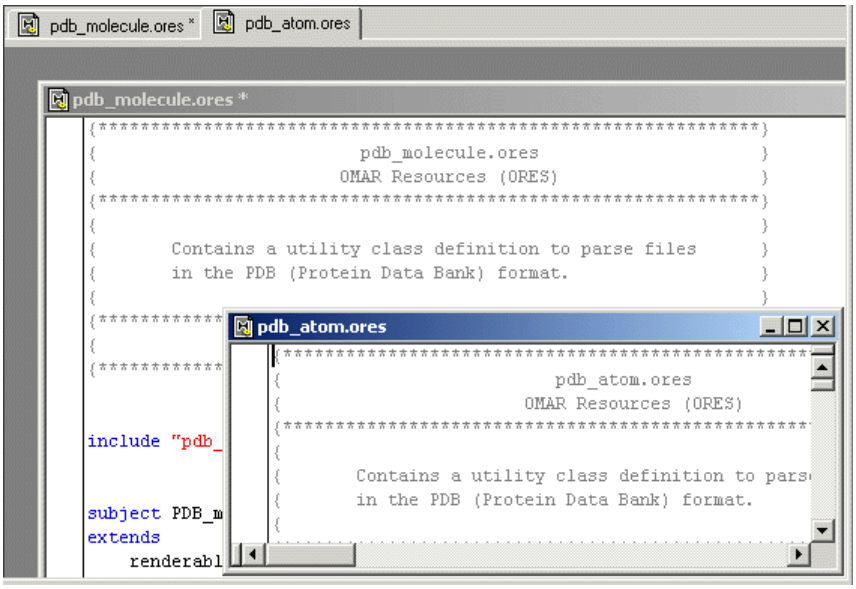


Figure 4.4: The Script Pane

The large pane on the right is used to view and edit script code. This pane is capable of displaying multiple subwindows each of which may contain a separate file. If the script subwindows are maximized, then you can navigate between the files by using the tabs at the top of the script pane.

THE OUTPUT PANE

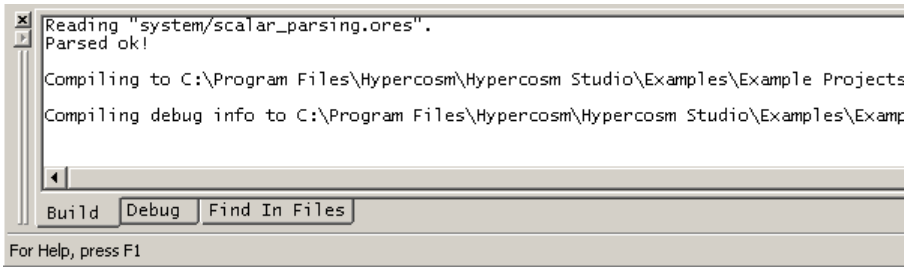


Figure 4.5: The Output Pane

The output pane is the short, wide pane located at the bottom of the Hypercosm Studio window. This pane is used to display compile time and run time information. It has three separate tabs: Build, Debug, and Find in Files.

THE BUILD TAB

When this tab is selected, the output pane shows error messages generated by the compiler.

THE DEBUG TAB

When this tab is selected, the output pane shows messages generated by the Player or generated by "write" statements in your script code. Write statements are often used in the process of debugging complex applets.

FIND IN FILES

This tab is used to display results returned by the Find and "Find and Replace" functions.

OPENING FILES

To open a script file in Hypercosm Studio, simply select "Open…" from the "File" menu or press the open file icon on the toolbar.

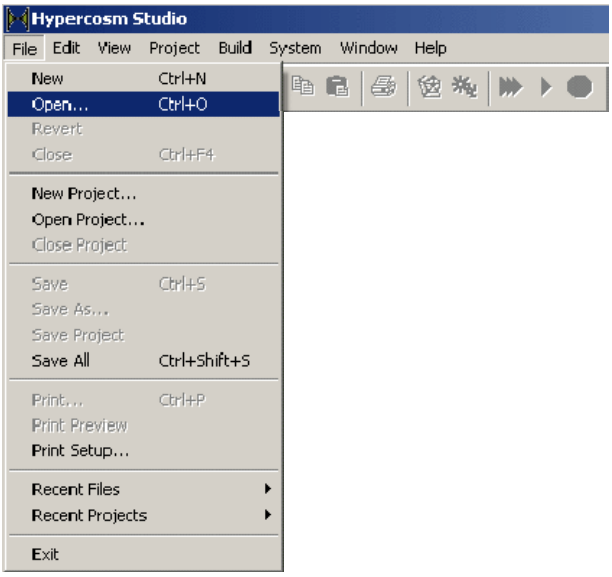


Figure 4.6: The Open File Menu

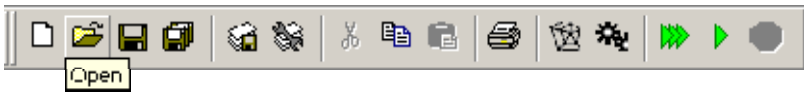


Figure 4.7: The Open File Icon on the Toolbar

This will bring up a dialog box as shown below. You can navigate to a project or using the buttons at the bottom, you can quickly jump to a directory where Hypercosm script files are located.

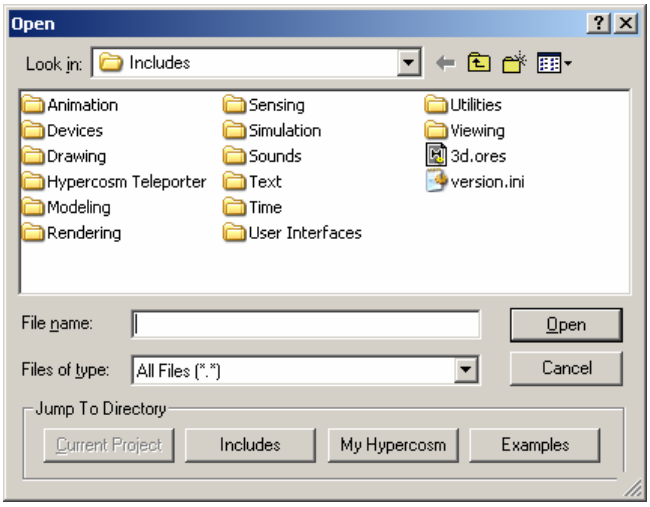


Figure 4.8: The "Open File" Dialog Box

CURRENT PROJECT

This button jumps to the current project if a project is open. If no project is open, then this button will be disabled.

INCLUDES

This button jumps to the directory "Hypercosm Studio/Includes/" where the standard set of ".ORES" include files are located. These include files each contain bits and pieces of Hypercosm functionality that you can use to add additional capabilities to your own projects.

MY HYPERCOSM

This button jumps to the directory "My Documents/My Hypercosm/" where you may put your own Hypercosm projects. You can locate your own projects anywhere, but it's sometimes convenient to store them in a centralized location such as this to keep them together.

EXAMPLES

This button jumps to the directory "Hypercosm Studio/Examples" where a collection of sample scripts and projects are located.

EDITING FILES

Hypercosm Studio also has standard code editing features. Once you've created or opened a file, you can use the text editing features under the Edit menu. They include commands for undoing, using the clipboard, and selecting and finding text.

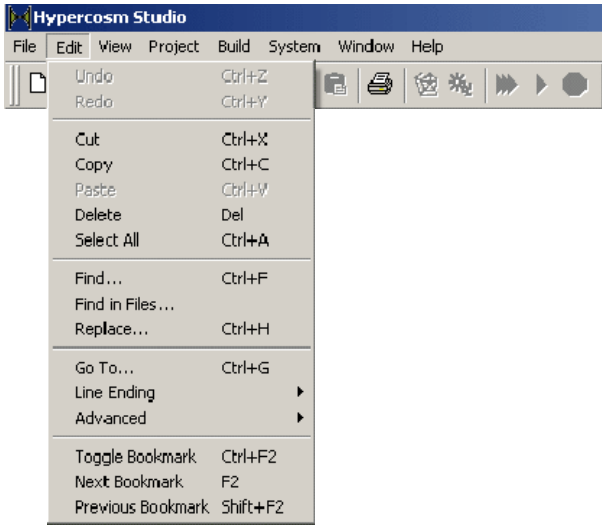


Figure 4.9: The Edit Menu

FIND IN FILES

One often used feature of Hypercosm Studio is the ability to search an entire directory of files for a pattern of text. This is used quite often when searching for definitions of script code utilities that you'd like to use.

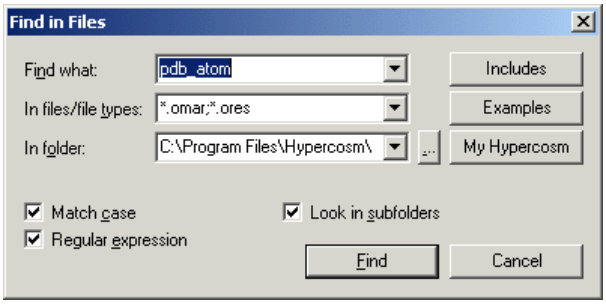


Figure 4.10: The Find In Files Dialog Box

The Find In Files dialog box also has a set of buttons to the right that set the search path to commonly used directories.

INCLUDES

This button is used when you want to search through the set of utility .ORES files that are included with Hypercosm Studio.

EXAMPLES

Use this search path to search through the set of examples provided with Hypercosm Studio.

MY HYPERCOSM

Use this search path to search your own Hypercosm script files that may be stored in the folder "My Documents/My Hypercosm/".

COMPILING FILES

To compile a file, select "Compile File…" from the "Build" menu or press the compile file icon on the toolbar.

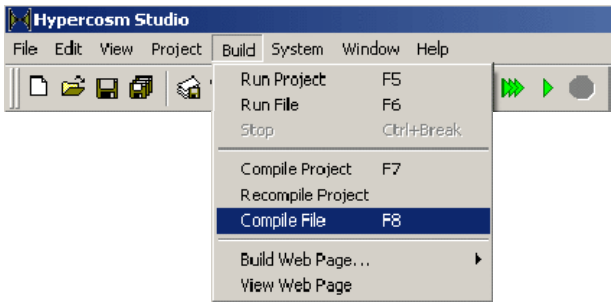


Figure 4.11: The Compile File Menu

When you compile a file, it will be compiled to a ".HCVM" file with the same name as the ".OMAR" file that it was compiled from and it will be placed in the same directory as the script file.

RUNNING FILES

To run a Hypercosm script file, select "Run File…" from the "Build" menu or press the run file icon on the toolbar. This will first compile and then run the file.

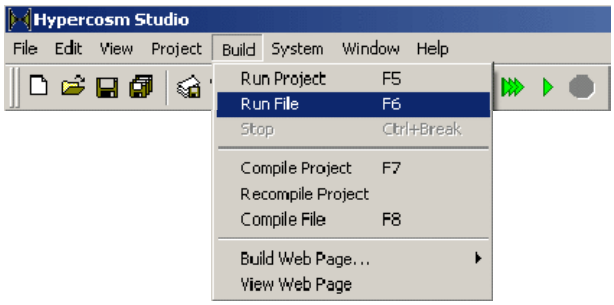


Figure 4.12: The Run File Menu

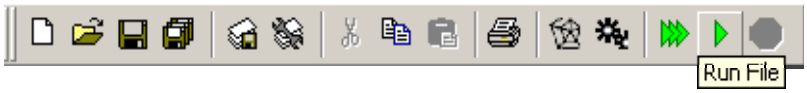


Figure 4.13: The Run File Icon on the Toolbar

When Hypercosm Studio runs the file, it does not create an applet file in the directory of the script file. To create an applet file, you should use the "Compile File" command instead.

CHANGING THE VIEW AND OPTIONS

The view menu is used to change a variety of options that determine how the interface looks and operates. These settings will not change the operation of the compiler, but merely change the aesthetics and operation of the user interface.

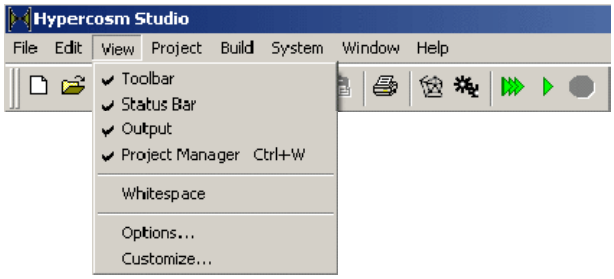


Figure 4.14: The View Menu

The first four items in the view menu are used to hide and show the various panes of the user interface. For example, if you are only working on OMAR files without projects, then you may choose to hide the Project Manager pane.

WHITESPACE

"Whitespace" is a term that refers to the spaces and tabs in a text document. On occasion, it may be useful to graphically show the spaces and tabs. This option is used for that purpose. The default setting for this value is to not graphically show the whitespace.

```
»   shape·instance·is
»   »   stopwatch·with
»   »   »   time·=·clock_timer·get_time;
»   »   »   rotate·by·90·around·<1·0·0>;
»   »   end;
»   end;»   //·instance
»
»
»   verb·check_keys·is
»   »   if·key_down·char_to_key·"·"·then
»   »   »   reset;
»   »   end;
»   end;»   //·check_keys
»
»
»   verb·draw_instructions
»   »   at·scalar·x·=·-.9,·y·=·.8;·
»   is
»   »   verb·draw_line
»   »   »   string·type·line;
```

```
shape instance is
    stopwatch with
        time = clock_timer get_time;
        rotate by 90 around <1 0 0>;
    end;
end;    // instance


verb check_keys is
    if key_down char_to_key " " then
        reset;
    end;
end;    // check_keys


verb draw_instructions
    at scalar x = -.9, y = .8;
is
    verb draw_line
        string type line;
```

Figure 4.15: Whitespace Shown (Left) vs. Hidden (Right)

OPTIONS

The options dialog box can be used to change settings that determine how the code will appear and how it is formatted as well as a variety of other miscellaneous settings that determine Hypercosm Studio operation.

EDITOR COLORS AND FONT

The colors and font that are used to display code are always a matter of personal preference. These parameters can be changed in the editor colors and font options dialog box. You can even change the colors that are used to highlight keywords in the OMAR language, the colors used to display numbers, the colors used for comments and other specific details.

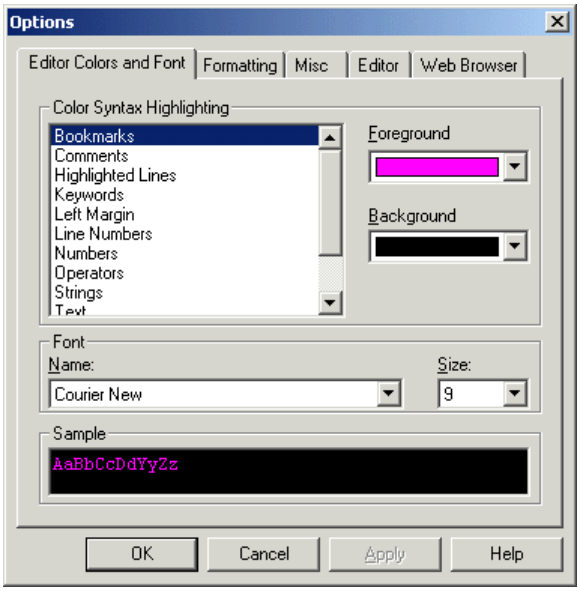


Figure 4.16: The Editor Colors and Font Tab

FORMATTING

The formatting tab is used to change the way code is indented.

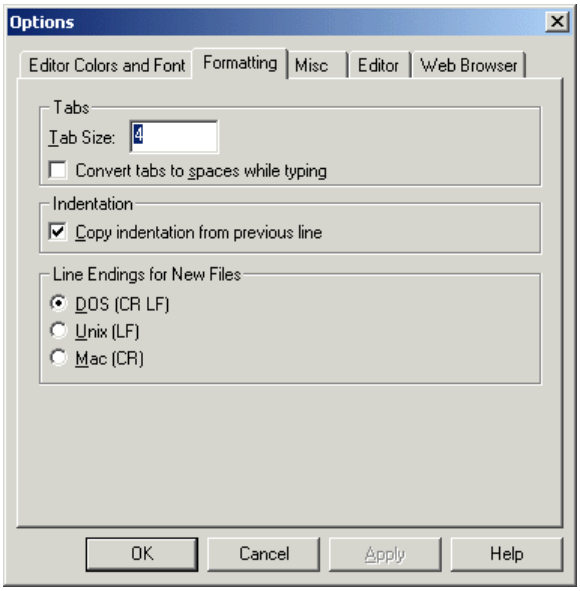


Figure 4.17: The Formatting Tab

MISC

The misc tab is used to set a variety of system settings that determine how Hypercosm Studio operates.

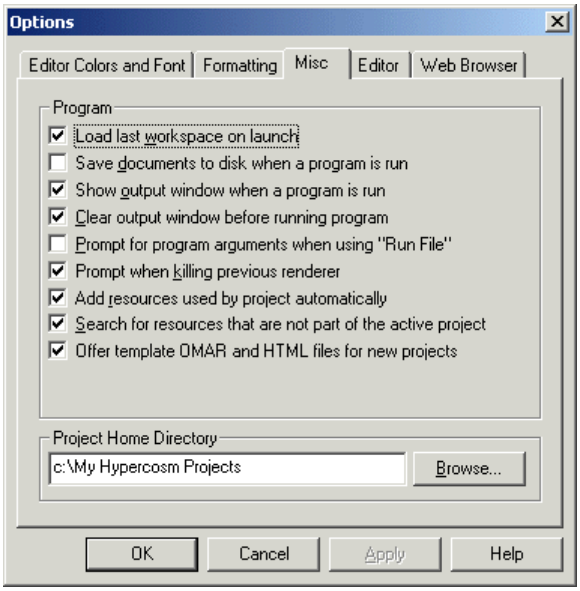


Figure 4.18: The Misc Tab

EDITOR

The editor tab provides a set of additional options for changing the formatting done by the code editor.

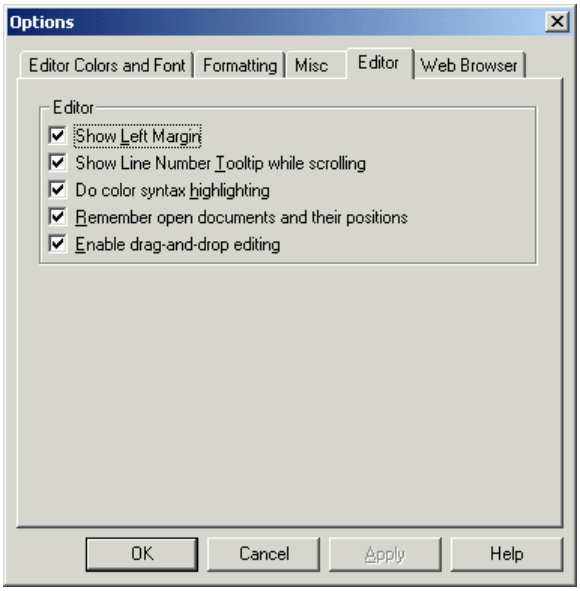


Figure 4.19: The Editor Tab

WEB BROWSER

The web browser tab is used to specify what browser to use to display applets in a web page.

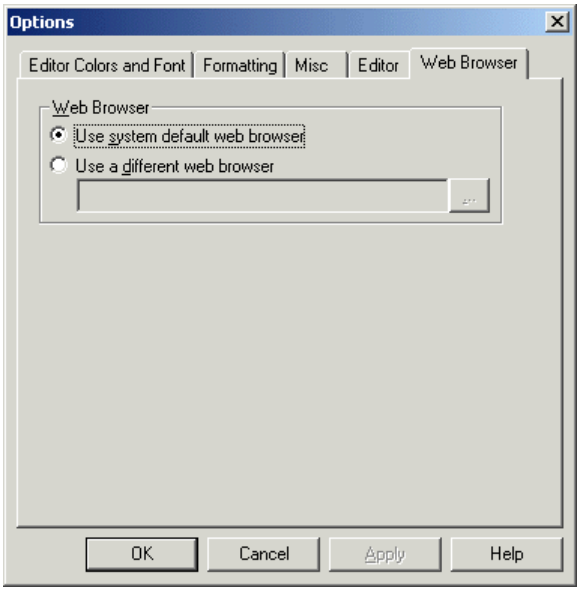


Figure 4.20: The Web Browser Tab

CHANGING SYSTEM SETTINGS

The System menu is used to change system settings. These system settings consist of a series of paths that tell Hypercosm Studio where to look for files that it uses.

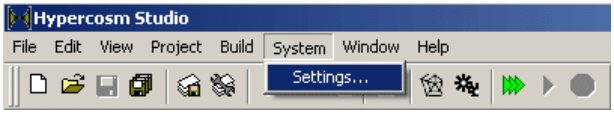


Figure 4.21: The System Settings Menu

SOURCE PATHS

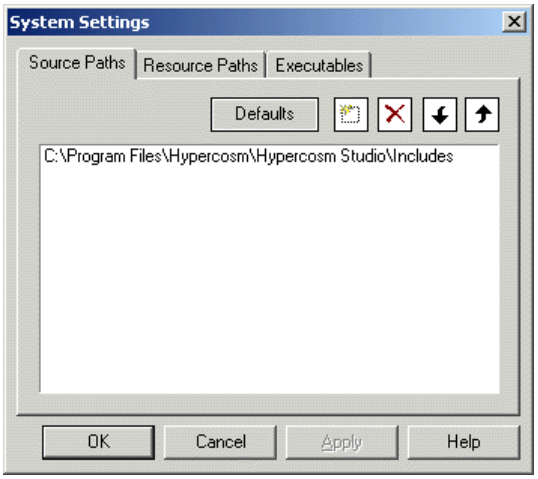


Figure 4.22: Source Paths System Settings

The "source paths" system settings determine where Hypercosm Studio looks for source (script) files. If the source file is not in the current project, then Hypercosm Studio will next look in this set of search paths for the file.

RESOURCE PATHS

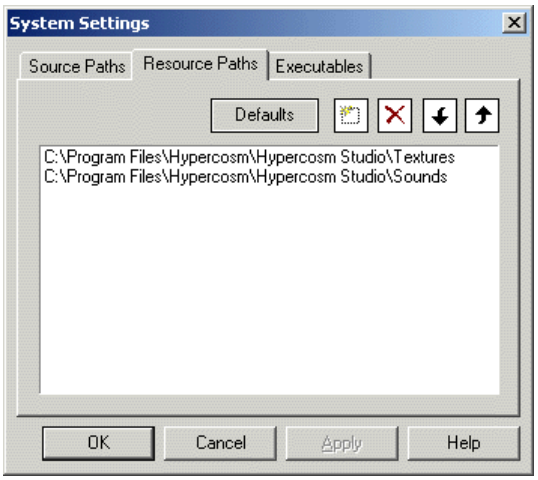


Figure 4.23: Resource Paths System Settings

The resource paths system settings determines where Hypercosm Studio looks for resource (textures, sounds, etc) files. If the resource file is not in the current project, then Hypercosm Studio will next look in this set of search paths for the file.

EXECUTABLE PATHS

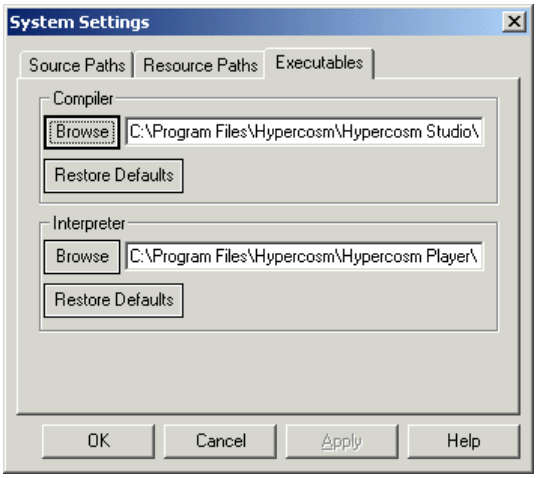


Figure 4.24: Executable Paths System Settings

Hypercosm Studio relies upon a pair of auxiliary executable programs in order to function. The first program is called a "compiler". This is the program that actually reads the source code from your .OMAR and .ORES files and converts it into a form where it can be executed (the .HCVM applet file). The second program is the program that actually executes the applet code to draw the graphics and perform other tasks specified by an applet. This program is called an "interpreter". By default, Hypercosm Studio places these helper programs in the directory "c:/Program Files/Hypercosm Studio/Bin/". If you want to change the location of these programs for whatever reason, you can. However, if you move these programs around, you'll need to change the executable paths settings to point to the compiler and interpreter programs in order for Hypercosm Studio to continue to work properly.

CHANGING THE WINDOW LAYOUT

The Window menu allows you to quickly change how your windows are arranged. This can be useful when you'd like to compare two files side by side.

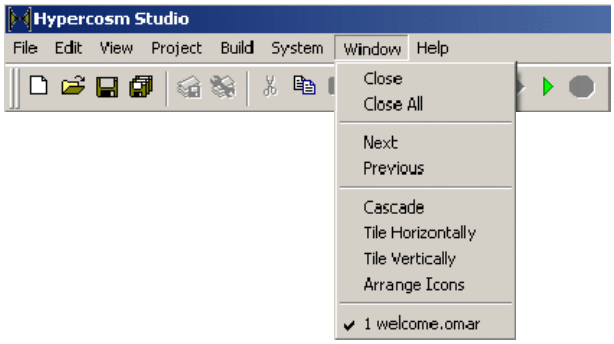


Figure 4.25: The Window Menu

FINDING VERSION INFORMATION

The Help menu is used to bring up a window that displays information about Hypercosm Studio.

Figure 4.26: The Help Menu

The About Box is used to find out the version number of Hypercosm Studio. The version number appears right below the Copyright notice.

LESSON 4: HYPERCOSM STUDIO PROJECTS

LESSON OBJECTIVES:

- Understand the function of projects
- Know how to open and existing project or create a new project
- Know how to add files to a project
- Know how to compile and run a project

LESSON CONTENTS:

- What are projects used for?
- Opening projects and creating new projects
- Adding files to a project and selecting a main file
- Compiling and running projects
- Changing project settings

WHAT ARE PROJECTS USED FOR?

Hypercosm projects are used to keep track of the collection of files that are used in building a 3D simulation. These files include script files, texture files, sound files and other resource files. When you compile a project into a finished web page, Hypercosm Studio collects all of these files together and copies the required resources into a directory along with the applet. This makes it easy to prepare an applet for distribution on the Internet, or on a CD or other recording medium.

OPENING PROJECTS

To open a project in Hypercosm Studio, just click on the "File" menu and select "Open Project".

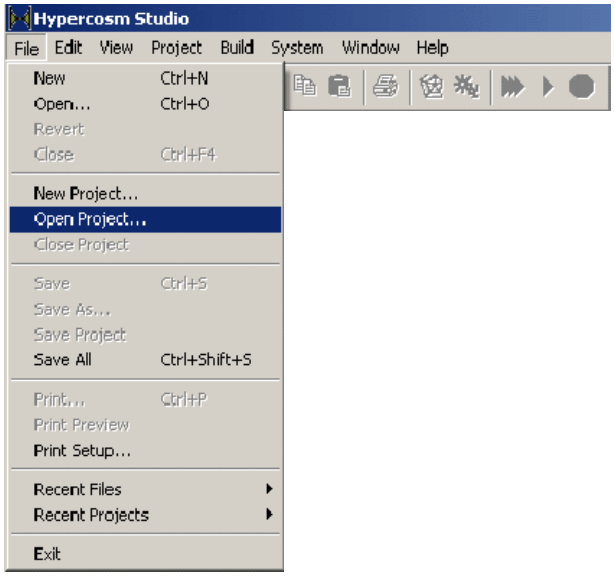


Figure 5.1: The Open Project Menu

This will bring up a dialog box as shown in Figure 5.2. You can navigate to a project or using the buttons at the bottom, you can quickly jump to a directory where projects are located.

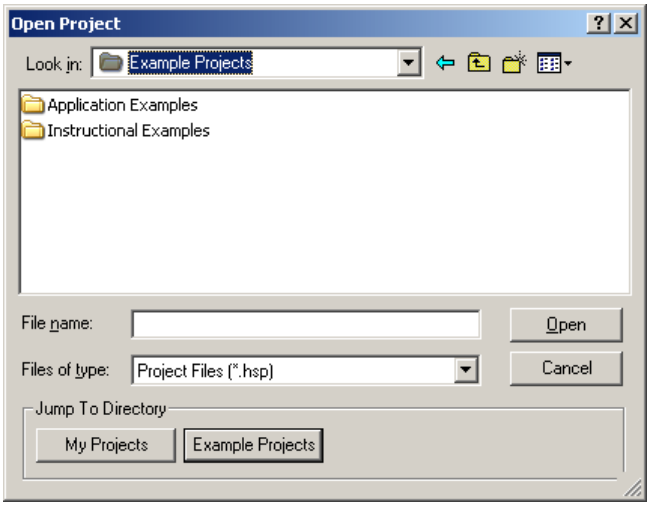


Figure 5.2: The Open Project Dialog Box

MY PROJECTS

The first button will jump to the directory "My Documents/My Hypercosm/My Projects/". When you install Hypercosm Studio, a directory is created for you in the "My Documents" folder called "My Hypercosm/". This directory has two subdirectories called "My Projects/" and "My Scripts/". If you like, you can store your Hypercosm projects and scripts in these directories and they will be easy to get to. You can store your projects anywhere you like on your computer.

EXAMPLE PROJECTS

This button will jump to "c:/Program Files/Hypercosm Studio/Examples/Example Projects/" folder. This folder contains a variety of example projects that are useful to examine for instructional purposes when you are learning Hypercosm scripting. Inside this directory, there are two additional directories, "Application Examples" and "Instructional Examples".

APPLICATION EXAMPLES

The "Application Examples" directory includes a set of finished applications that were created for a number of different purposes including education, entertainment, and scientific visualization.

INSTRUCTIONAL EXAMPLES

The "Instructional Examples" directory includes a set of simple example projects that show how to use the various features and capabilities of the Hypercosm system. These can be very useful for learning Hypercosm scripting.

CREATING NEW PROJECTS

To create a project, simply select "New Project" on the "File" menu.

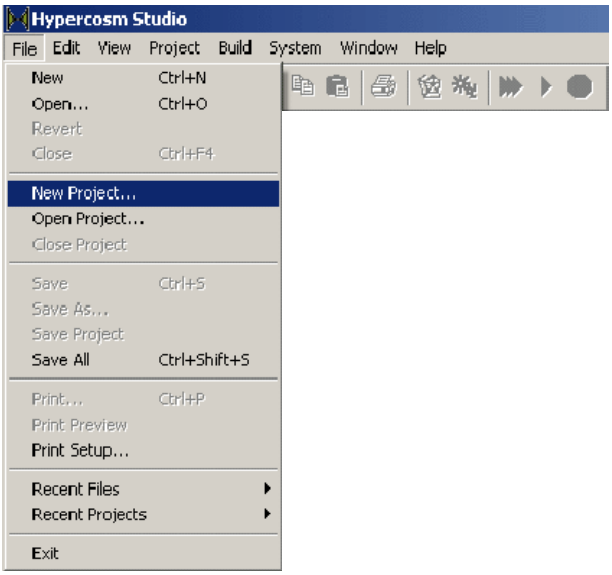


Figure 5.3: The New Project Menu

This will bring up a dialog box as shown below:

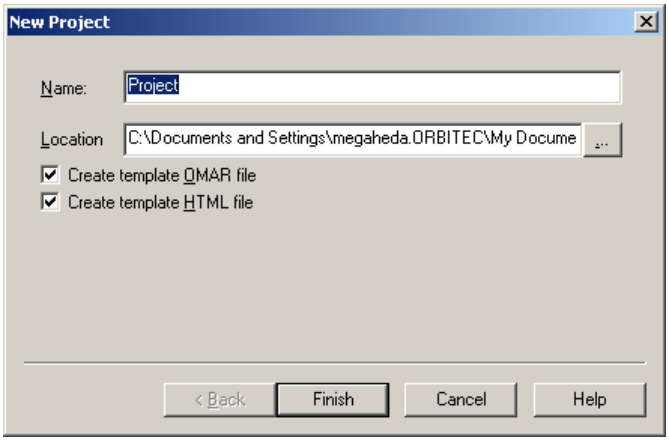


Figure 5.4: The New Project Dialog Box

By default, Hypercosm Studio will create template ".OMAR" and ".HTML" files for you. If you already have your own ".OMAR" or ".HTML" files that you'd like to use, then you can deselect these options and then add your own files to the project at a later time.

ADDING FILES TO A PROJECT

Once a project is created, it needs to be populated by files. To add a new file, just select "Add Files" from the "Project" menu (see Figure 5.5). This will bring up a dialog box as shown in Figure 5.6. Note that multiple files may be simultaneously selected from this dialog box and added to the project in one step.

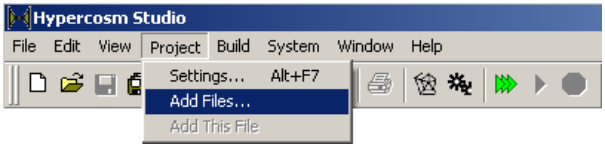


Figure 5.5: The Add Files Menu

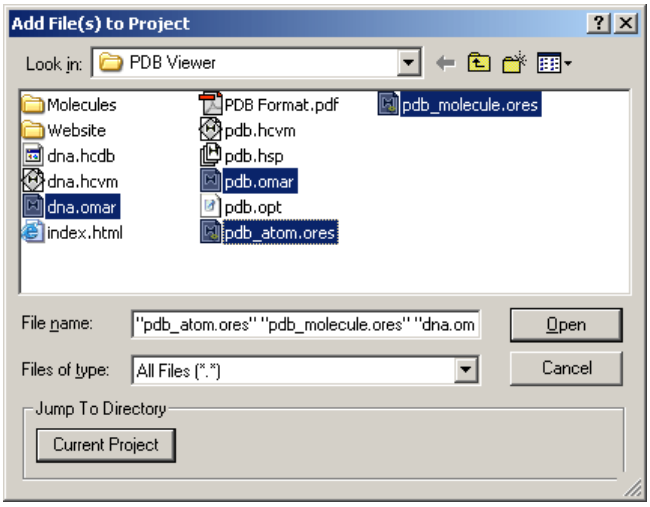


Figure 5.6: The Add File(s) Dialog Box (shown adding multiple files at once)

SELECTING A MAIN FILE

Once a collection of files is added to the project, a main file must be selected. A project may potentially contain multiple ".OMAR" files, each of which has can be run on its own. The project needs to know which of these files is to be used as the main file of the project and contains the main entry point for the simulation. There are two ways to select the main file.

USING THE PROJECT MANAGER

To select the main file using the Project Manager window, move your mouse cursor over the file in the Project Manager window that you want to select as the main file and click the right button on the mouse. This will bring up a popup menu as shown below. On this popup menu, move your cursor down to select the last option "Set As Main".

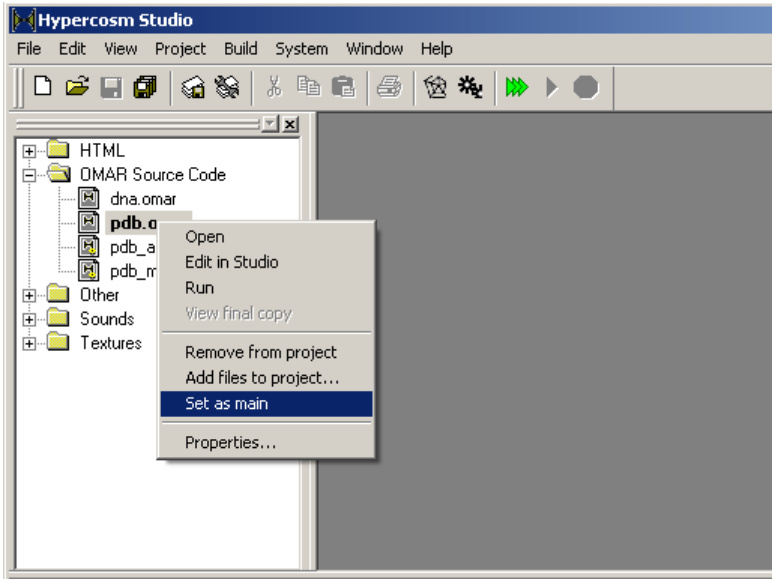


Figure 5.7: Setting the Main File Using the Project Manager

USING THE PROJECT SETTINGS WINDOW

An alternative way to set the main file (if you don't have a two button mouse, for instance) is to use the Project Settings window. To do this, go to the "Project" menu at the top of the window and click on "Settings…" as shown in Figure 5.8. This will bring up the dialog box as shown in Figure 5.9. Using this dialog box, you can highlight the file that you want to select as main and hit the "Set as Main" button.

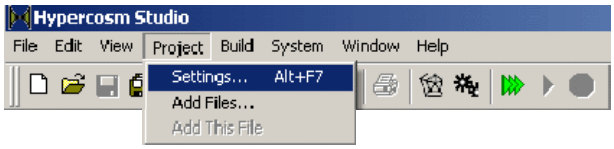


Figure 5.8: The Project Settings Menu

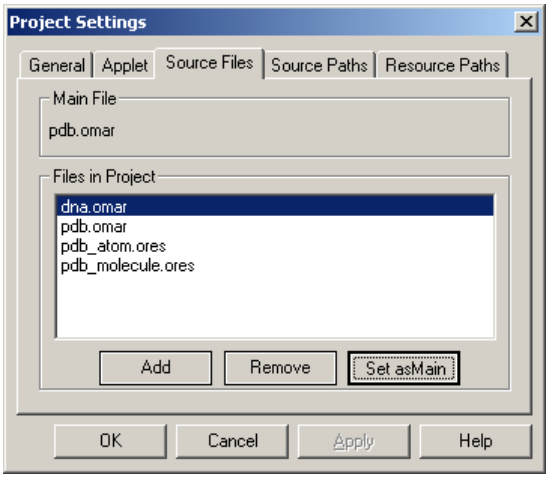


Figure 5.9: Setting the Main File Using the Project Settings

Once a file has been selected as main using either method, it will appear in bold face type in the Project Manager window.

COMPILING PROJECTS

Once files have been added to a project and a main file has been selected, then the project can be compiled. To compile a project, simply select "Compile Project" from the "Project" menu as shown in Figure 5.10 or press the compile project icon on the toolbar as shown in Figure 5.11.

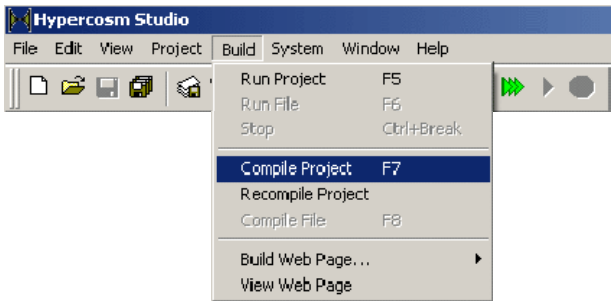


Figure 5.10: Compile Project Menu

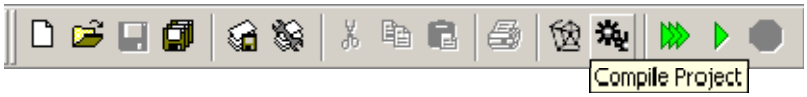


Figure 5.11: The Compile Project Icon on the Toolbar

As a project is being compiled, a list of all the files that are included in the project or referenced by files included in the project will appear in the Build tab of the Output window. Any compile errors will appear in the Output window. If you click on the error message, Hypercosm Studio will conveniently open the file containing the error and place the insertion point near where it believes the error is located. This makes it relatively quick and easy to locate and fix compile time errors.

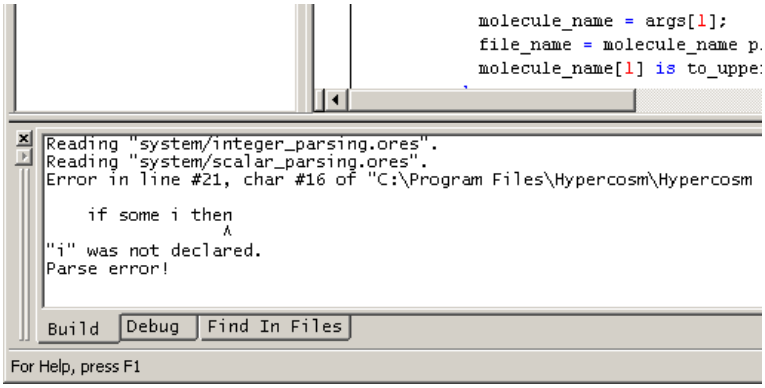


Figure 5.12: A Compile Error Displayed in the Output Window

If the project has already been compiled and the applet is up to date, then Hypercosm Studio will display a message in the Output window that says the applet is up to date. On occasion, you may need to force the compiler to update the applet. This is most often because a file is changed that is referenced by one of the files in your project but not specifically included in the project. In this case, the modification date of the script files in the project will be older than the creation date of the applet so Hypercosm Studio believes that the applet is up to date. In this case, you can force Hypercosm Studio to rebuild the applet just by selecting the "Recompile Project" command under the "Build" menu.

RUNNING PROJECTS

Once a project has been compiled, you can run it by selecting "Run Project" from the "Build" menu as shown in Figure 5.13. You can also select the run project icon from the toolbar as shown in Figure 5.14.

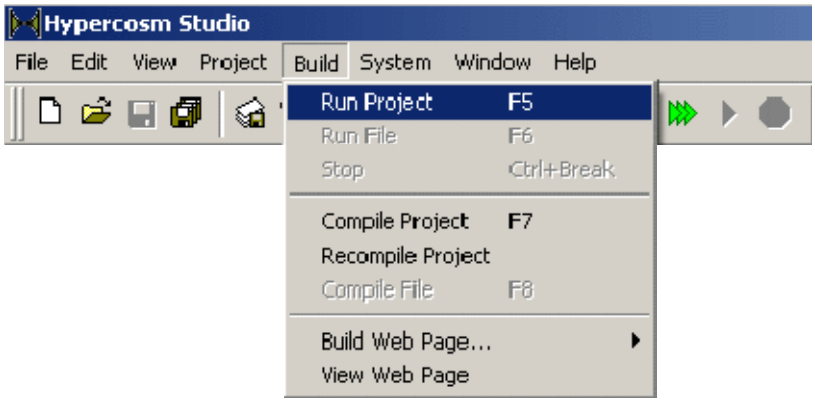


Figure 5.13: The Run Project Menu

Figure 5.14: The Run Project Icon on the Toolbar

If the project has not already been compiled, then Hypercosm Studio will compile the project before it attempts to run it.

ADDING RESOURCES TO THE PROJECT AUTOMATICALLY

A useful convenience offered by Hypercosm Studio is the ability to add resources to the project automatically. Resources are textures files, sound files, or text files used by an applet. Rather than requiring you to add each of these resources individually, Hypercosm Studio has a mechanism for listening for what resources are requested by an

applet when it runs and then searching for and adding those resources to the project automatically. Since it's not uncommon for a project to require a few dozen textures, this feature can save a lot of time. Normally, this feature is activated by default, but it can be deactivated. This feature can be either activated or deactivated by going to the "View" menu and selecting "Options". This will bring up a dialog box with a series of tabs at the top. Select the "Misc" tab and you will see a series of checkboxes for different items. The checkbox for "Add resources used by project automatically" is third from the bottom (see figure 5.15 below).

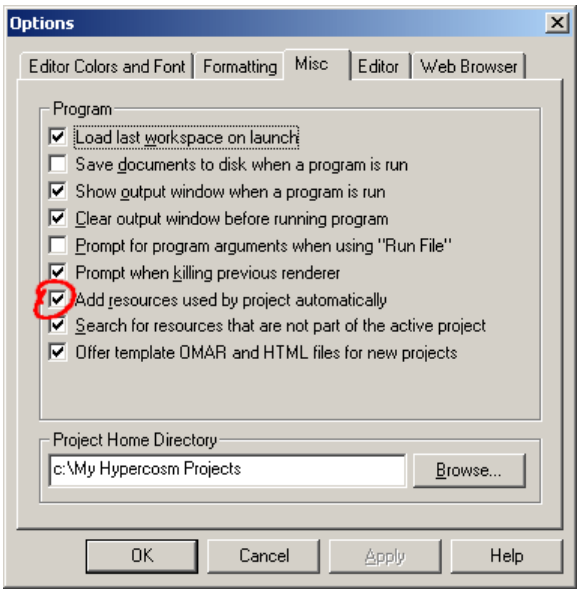


Figure 5.15: Option to Add Resources Used By Project Automatically

CHANGING PROJECT SETTINGS

In addition to the system settings of Hypercosm Studio, each project can also have its own settings. This is useful because sometimes a particular project will require additional search paths to be used for finding a set of common files. In order to add or change project settings, simply go to the Project menu and select "Settings…" as shown in Figure 5.8. This will bring up a dialog box as shown in Figure 5.16. This dialog box will have a series of tabs across the top. Each of these tabs is described in the next section.

GENERAL PROJECT SETTINGS

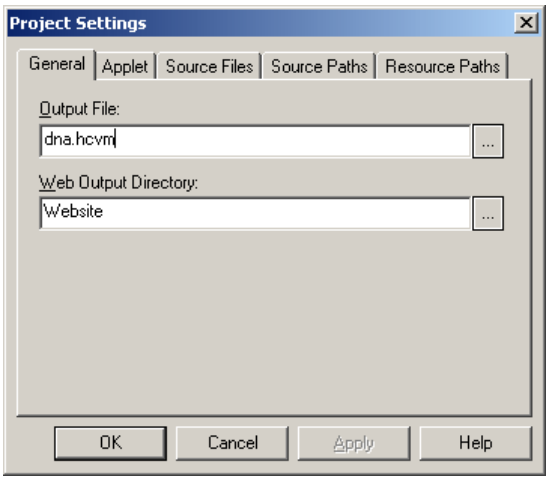


Figure 5.16: General Project Settings

OUTPUT FILE

This is the name that will be given to the applet file when it is compiled. It is also the name that will be used in a web page if you choose the "Build Web Page…" option from the "Build" menu.

WEB OUTPUT DIRECTORY

This is the name of the directory that will be used to accumulate the applet and its set of associated resources when you choose the "Build Web Page…" option from the "Build" menu.

APPLET PROJECT SETTINGS

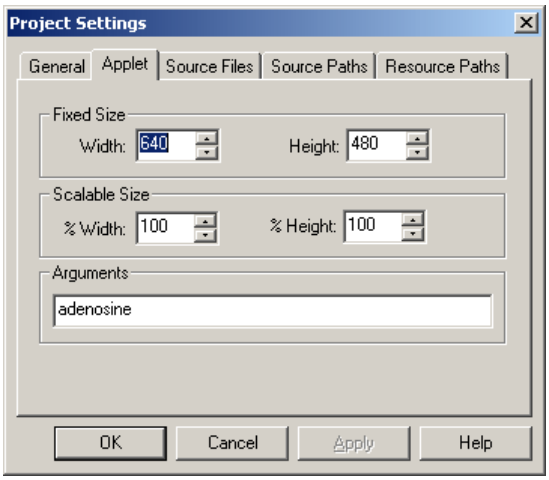


Figure 5.17: Applet Project Settings

FIXED SIZE

The fixed size width and height are used to set the size of the window when the applet runs. In addition, this width and height are used to set the size of the applet in the web page when you select "Build Web Page… Fixed Size" from the "Build" menu.

SCALABLE SIZE

The scalable size width and height are used to set the size of the applet as a percentage of its frame in the web page when you select "Build Web Page… Scaled" from the "Build" menu.

ARGUMENTS

Program arguments are a series of text strings that are passed into the applet when it runs. This is a simple and convenient way to pass a small amount of text-based information from a web page into an applet. This allows applets to be configured by their associated web page. Once the arguments are passed into the applet, it is the applet's responsibility to interpret the arguments. Some applets respond to a variety of program arguments while other applets may not respond at all to program arguments.

SOURCE FILES PROJECT SETTINGS

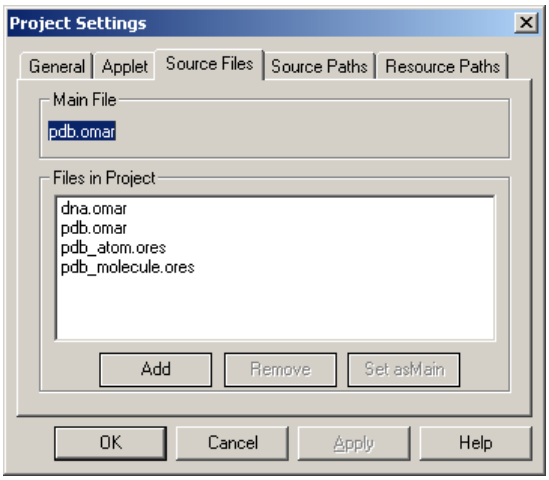


Figure 5.18: Source Files Project Settings

This dialog box can be used to add files to a project, remove files from a project, and to set a file as the main file in the project. The functions performed by this dialog box are most often done using the Project Manager pane in the main Hypercosm Studio window. However, if you have chosen to hide the Project Manager pane (using the View menu), then you can use this dialog box instead to perform these functions.

SOURCE PATHS PROJECT SETTINGS

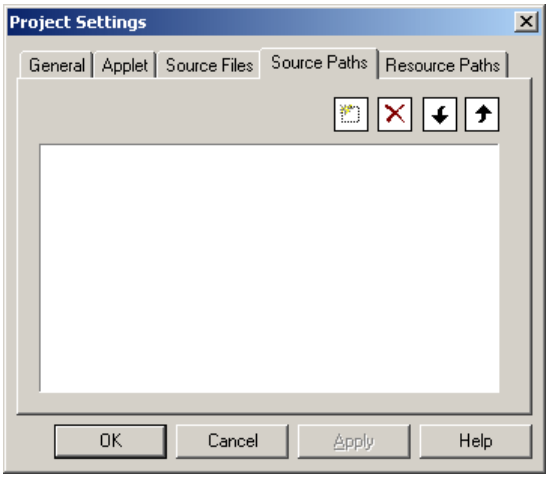


Figure 5.19: Source Paths Project Settings

The Source Paths dialog box is used to add additional paths to the project where the compiler can look for script files. When the compiler searches for a script file, it will look for the file in the locations listed below searching the list in order from top to bottom:

1) Project directory
2) System source paths
3) Project source paths

RESOURCE PATHS PROJECT SETTINGS

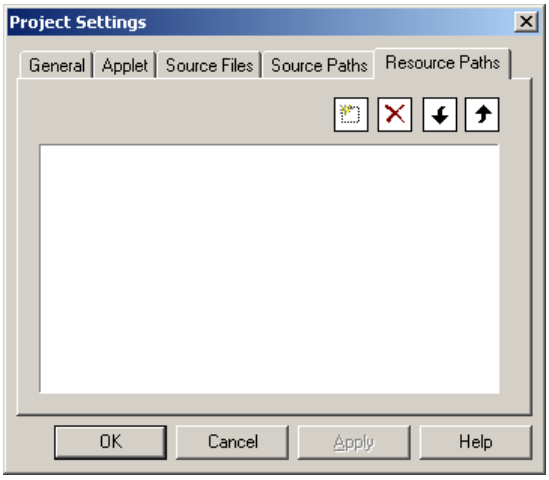


Figure 5.20: Resource Paths Project Settings

The Resource Paths dialog box is used to add additional paths to the project where the compiler can look for resource files (textures, sounds, text, etc.) When the compiler searches for a resource file, it will look for the file in the locations listed below searching the list in order from top to bottom:

1) Project directory
2) System resource paths
3) Project resource paths

LESSON 5: CREATING HYPERCOSM ENHANCED WEB PAGES

LESSON OBJECTIVES:

- Know how to embed Hypercosm applets into web pages
- Know how to organize resources used by Hypercosm enhanced web page

LESSON CONTENTS:

- Building and viewing web pages automatically
- Creating custom web pages

ADDING AN APPLET TO A WEB PAGE

Once you've created an OMAR program and an accompanying Hypercosm Studio project, you can then create an applet and add it to your HTML file. What you should read now depends on how you want to do this:

- If you want to add an applet to the template HTML file that was created when you started a project, you can skip ahead to the "Building the Web Page Automatically" section.

- If you want Hypercosm Studio to insert an applet automatically into an existing web page, then read the "Preparing Your HTML File" section, followed by the "Building the Web Page Automatically" section.

- If you want Studio to generate code that you can then insert by hand into an existing HTML file, then read the "Inserting Hypercosm Code by Hand" section.

Note: to insert two different applets into the same page, you have to insert them by hand because Studio cannot insert two different applets automatically.

PREPARING YOUR HTML FILE

Hypercosm Studio can insert an applet into your HTML page automatically. Remember that Studio first makes a copy of your HTML page, and inserts the applet into the copy, not into the original page.

To do this, you must first do two things:

1. Hypercosm Studio must know where to find the HTML file. If you haven't already added the HTML file to your project, follow the directions given in the "Adding Files to Projects" section.

2. Hypercosm Studio must know where in the HTML file to add the HTML code which links in your Hypercosm applet. The rest of this section explains how to specify the location of that HTML code in an HTML file.

Follow these directions to specify the location of the Hypercosm applet in your HTML file.

1. Open the HTML file with a text editor or with your web design program.

2. Move to the spot in the HTML file where you want to call the Hypercosm applet.

3. Insert the following two lines of text at exactly the spot where you want the applet to appear. These lines are case sensitive, so be sure to type them exactly as they appear. If you place any text between the two lines, Hypercosm Studio will delete it when it inserts the applet.

   <!--HypercosmAppletBegin-->
   <!--HypercosmAppletEnd-->

Note that if you have coded your HTML file by hand, you can just type in the text as it appears here. If you are using a web design program, you must make sure that your program won't modify this text in any way, either because it doesn't understand it, or it thinks it's unnecessary. Most web design programs give you the capability to add comments or code that is used by other applications; check the documentation that came with your web design program if you are unsure how to do this.

Once you've inserted this text into your HTML file and saved the file, you are ready to have Studio build the web page automatically.

BUILDING THE WEB PAGE AUTOMATICALLY

To build the web page automatically, click on the "Build" menu and select "Build Web Page…". This will present a menu with two options as shown in Figure 6.1. Select either "Scaled" or "Fixed Size". You can also click the "Build Web Page" icon on the toolbar to build a scaled web page as shown in Figure 6.2.

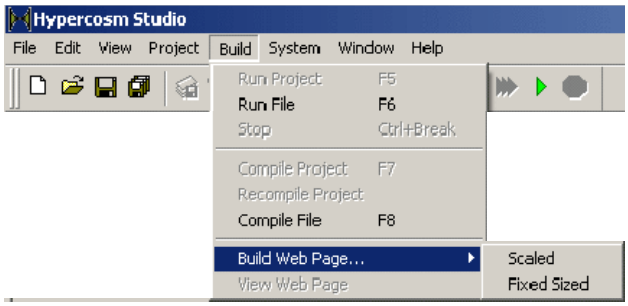


Figure 6.1: The Build Web Page Menu

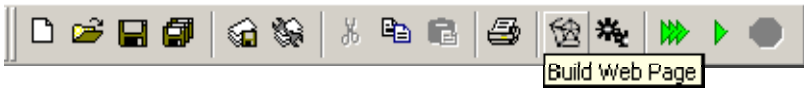


Figure 6.2: The Build Web Page Icon on the Toolbar

When Hypercosm Studio builds a web page using the following steps:

1. **Compile the Applet**
   Hypercosm Studio compiles your program into HCVM format (Hypercosm's byte code format), and places the resulting HCVM file in your project's directory (or wherever you specified the "output file" be placed in your project's settings). Hypercosm Studio then copies that HCVM file and places the copy into the "Web Output Directory" (which is the Website subdirectory by default). Note that this effectively creates two HCVM files: one in your project directory, and one in your web output directory.

2. **Copy the Resource Files**
   Hypercosm Studio makes a copy of the project's HTML file, and places that copy in the subdirectory called /Project/Website/ where Project represents the name of your project. Hypercosm Studio also copies any necessary sound or texture files to the Project/Website subdirectory. If you'd prefer that the applet, HTML, and associated resource files be placed in a different directory, you can change the default "Web Output Directory" in your project's settings.

3. **Make the Web page**
   To make the web page, Hypercosm Studio opens the copy of the ".HTML" file and looks for the string that tells it where to insert the applet. The template ".HTML" file that automatically generated when you created the project already contains this string. If you are working with your own HTML, then you should have already inserted this string yourself, using the directions given in the "Preparing Your HTML" File" section above. Once Hypercosm Studio finds the string, it inserts a reference to the Hypercosm applet file (known as an "applet tag") into the copy of your HTML file. To build a web page, choose Build > Build Web Page > Scaled or Fixed Sized, or click on the Build Web Page button on the toolbar. The settings for Scaled or Fixed Sized can be set by going to Project > Settings > Applet. The "Fixed Size" setting will place the applet in your ".HTML" file that will have a width and height of specified pixel values. The "Scaled" setting will place the applet in the html file according to a percentage of the total html page size. You can follow the progress of the build in the output pane of Hypercosm Studio where messages are displayed indicating the build progress. Once the process is complete, Hypercosm Studio displays a success message in the output pane.

LESSON 6: INTRODUCTION TO THE OMAR LANGUAGE

LESSON OBJECTIVES:

- Understand the unique language features of OMAR
- Understand basic OMAR language structures

LESSON CONTENTS:

- An OMAR overview
- Hypercosm language requirements
- Similarities and differences with other languages
- Basic program structure
- Elements of programming
- Comments
- Language vocabulary
- Language formatting
- The include directive

AN OMAR OVERVIEW

There are perhaps hundreds of different programming languages. It probably seems like the last thing that the world needs is yet another new programming language. If you are a programmer, then this just means another new syntax to get used to and even worse, another new set of semantic rules which may get confused with languages you already know. But there are a number of very good reasons for OMAR. It was developed because there simply are no other languages which have the features of safety, architecture independence, general purposefulness, and expressiveness that are required for today's web based graphics applications. You may say that people have gotten along just fine so far with "C", the standard system programming language these days. Yes, it is theoretically possible to write bug-free, somewhat understandable code in C, but in practice, code written in C is almost always bug-ridden, difficult to read and understand, and almost impossible to share and reuse. A better solution is clearly needed.

HYPERCOSM LANGUAGE REQUIREMENTS

To better understand the reasons for learning a new scripting language, it is instructive to enumerate the requirements that lie behind the language design choices.

1. Security

The first requirement for any language that is going to be used to drive web based content is security. The language must make it impossible to write malicious code that can damage a user's computer and modify their files. Because of this requirement, all web-based languages are interpreted rather than compiled to run on the hardware's CPU. When a language is interpreted, a piece of software called an interpreter checks each instruction before executing it to make sure that it is safe. Any language that is compiled to run directly on the computer's CPU can potentially perform any operation because at the present time, no computer hardware (with the exception of some research oriented LISP machines) has the capability of ensuring run time security.

2. Platform Independence

Another requirement for languages that may be used in web applications is platform independence. The language should have no preconceived notions that pertain to the type of hardware that it will be used on.

3. Ease of Use

Many content developers that will be using a language that is intended for 3D graphics will tend to come from a graphics and web design background rather than from a system programming background. This means that the language that is used should be as friendly and easy to use as it's possible for a programming language to be.

4. Performance

A language that is intended to be used for real-time 3D graphics applications should be as high performance as possible. Often, 3D simulations require computations to simulate real time physics, compute 3D geometry, compute 3D transformations and other computationally intensive tasks. All of these computations must be performed in real-time to keep the simulation running at a reasonable frame rate. This means that the interpreted language will need to be as high-performance as possible.

SIMILARITIES AND DIFFERENCES WITH OTHER PROGRAMMING LANGUAGES

In this section, we take a look at existing scripting and programming languages.

	Security	Platform Independence	Ease of Use	Performance
C / C++	No	No	No	Yes
C#	Yes	No	No	Yes
Java	Yes	Yes	No	Yes
Delphi / Pascal	No	No	Yes	Yes
Perl	Yes	Yes	No	No
Python	Yes	Yes	Yes	No

Javascript	Yes	Yes	Yes	No
Ruby	Yes	Yes	Yes	No
Visual Basic	Yes	No	Yes	Yes
Smalltalk	Yes	Yes	No	No
Lisp	Yes	Yes	No	No
Tcl	Yes	Yes	No	No
OMAR	Yes	Yes	Yes	Yes

Figure 7.1: Programming Language Characteristics

	Java	**C#**	**Python**	**Java-Script**	**OMAR**
Object-Oriented	Yes	Yes	Yes	No	Yes
Static Type Checking	Yes	Yes	No	No	Yes
Dynamic Arrays	Yes	Yes	Yes	Yes	Yes
True Multidimensional Arrays	No	Yes	Yes	Yes	Yes
Optional Parameters	No	No	Yes	Yes	Yes
Keyword Parameters	No	No	Yes	No	Yes
Garbage Collection	Yes	Yes	Limited	Yes	Yes
Reference Types	No	Yes	No	No	Yes
Function Variables	No	Yes	Yes	Yes	Yes

Figure 7.2: Programming Language Features

Of all of the programming languages listed, the closest to OMAR are probably Java and C#. These languages are all strongly typed, safe, and compiled. The main differences between languages such as Java and C# and the OMAR language mostly involve syntax and ease of use. As you can see in the following examples, system programming languages such as Java and C# typically have a steep learning curve and are difficult to learn for non-professional programmers.

```
public class hello {
        static public void main(String[] argv) {
                System.out.println("Hello world!");
        }
}
```

Figure 7.3: Hello World program in Java

EXAMPLE: "HELLO_WORLD.OMAR"

```
do say_hello;

verb say_hello is
        write "Hello World!";
end;
```

Figure 7.4: Hello World program in OMAR

WHERE TO GO FOR MORE INFORMATION ABOUT OMAR

In the following sections, we will provide a short introduction to the OMAR language. This is intended to give the user enough information to get started in writing OMAR scripts. However, this is by no means a comprehensive course on OMAR or programming. To really understand it, you will need to spend some time writing code in OMAR. For a detailed description of the programming language, there is a complete language reference manual available from Hypercosm called ***"The OMAR Programming Language"***. You'll want to have this at your side as you become more experienced in OMAR scripting and begin to tackle more advanced projects.

BASIC PROGRAM STRUCTURE

Each OMAR program contains a single program header followed by one or more program declarations.

THE PROGRAM HEADER

The program header tells where to begin executing the code. The program header consists of the reserved word "do" followed by one or more identifiers separated by commas. These identifiers are names of subprograms to execute. If more than one subprogram is listed in the header then the subprograms will be executed in the order that they are listed. Program headers can only be found in ".OMAR" files. ".ORES files contain OMAR code, but with no program header since they can not be executed directly. Note that there may be many subprograms declared in the body of the code that are not listed in the header. All subprograms that are executed are called either directly or indirectly from one of the subprograms listed in the header.

PROGRAM DECLARATIONS

Following the program header is a list of declarations. The declarations compose the body of the program. Declarations fall into three general categories: data declarations, type declarations, and subprogram declarations.

EXAMPLE: "PROGRAM_STRUCTURE.OMAR"

```
do task1, task2;                          // program header listing subprograms to run

verb task1 is                             // first subprogram declaration
        // Program instructions go here
end;

verb task2 is                             // second subprogram declaration
        // Program instructions go here
end;
```

Figure 7.5: Basic Program Structure

ELEMENTS OF PROGRAMMING

If you have never written a computer program before, then let's take a minute to examine some general characteristics of computer programming languages. If you already know how to program, then you can move ahead to the next section, "Comments"

DATA

The first component of computer languages is data. Everything that the computer can manipulate or calculate is somehow encoded in some form of data. Data is defined as a series of variable declarations.

```
integer counter = 0;
```

Figure 7.8: An Example Variable in OMAR

INSTRUCTIONS

The second component of computer languages is instructions. Instructions are what describe all the actions and changes to data that occur when a program runs. Instructions are defined as a series of "statements".

```
counter = itself + 1;
```

Figure 7.9: An Example Statement in OMAR

DEFINITIONS

The third part of the computer language is definitions. Since we are writing code in a high level language, some of our code is necessary only to clarify our meaning to the compiler. These definitions are only part of the translation process and do not actually get translated into either data which is operated on or instructions which are executed by the computer. For example, a declaration of a user defined data type is simply a template which is used by the compiler and neither holds data nor is executed by the computer. Definitions are defined as a series of "types".

```
enum day_of_the_week is Monday, Tuesday, Wednesday, Thursday, Friday;
```

Figure 7.10: An Example Type Definition in OMAR

COMMENTS

It is also possible to add your own comments into the code. These comments are ignored by the compiler, and therefore do not add any features to the program but simply make the program easier to understand. In addition, comments are often used to temporarily disable a piece of code that is not working properly without permanently erasing it from the file. There are two types of comments in OMAR: line comments and block comments.

LINE COMMENTS

The first type of comment is called a "line comment" because it spans only one line in the program. Line comments are indicated by the forward slashes, "//". When this symbol is encountered, all text until the end of the line is ignored by the compiler, so you can include any kind of text or symbols on the line as a comment.

```
// This is a comment
integer a;                    // This is a comment following a code declaration
```

Figure 7.6: Line Comments in OMAR

BLOCK COMMENTS

The next, more powerful form of comments is the "block comment". Block comments may span multiple lines and may even include other line or block comments. Block comments are formed by enclosing the commented text by a pair of curly braces. Generally, block comments should be used mostly for commenting out blocks of code and not in cases where line comments may be used instead because a misplaced curly brace can result in a large block of you program being commented out.

```
{ This is a block comment}

{
This is also a block comment
}

{
This shows how block comments can enclose other                    {block comments}
and also // line comments
}
```

Figure 7.7: Block Comments in OMAR

LANGUAGE VOCABULARY

The text of a computer program is made up of a variety of different components. The text can be broken down into reserved words, identifiers, and special symbols.

RESERVED WORDS

Normally when you process human language, you rely on certain words having a predefined meaning that is not subject to change. Computer languages also have words that you make up yourself to name things and words that have a predefined, unchangeable meaning. Those unchangeable words are called "reserved words". The complete list of reserved words for the OMAR language is listed below.

and	dot	is	real	vector
anim	double	its	return	verb
boolean	else	itself	scalar	while
byte	elseif	long	shader	with
case	end	mod	shape	
char	false	not	short	
complex	for	or	subject	
const	imag	parallel	switch	
cross	integer	perpendicular	then	
div	if	picture	true	
do	include	question	type	

Figure 7.11: Reserved Words

IDENTIFIERS

The words in the computer program that are not reserved words are words that you define yourself. These are called "identifiers". An identifier is a word that is used to name something. When new variables are created or new data types are defined, they must be given names to identify them in a unique way. The rules for creating new identifiers is that (1) they must not be a reserved word and (2) they must begin with a letter and may be followed by letters, numbers, or the underscore character, "_". The maximum length of identifiers is 256 characters, which should be plenty for most names.

Lowercase	Uppercase	Digits	The Underscore
"a" .. "z"	"A".. "Z"	"0".. "9"	"_"

Figure 7.12: The Allowable Alphabet for Identifiers in OMAR

In the example below, we show examples of valid identifiers in the OMAR language.

```
name
counter
number_of_widgets
a
file_not_found
```

Figure 7.13: Valid Identifiers in OMAR

Identifier	Reason Why Identifier is Invalid
24_bit_mode	identifiers may not begin with a number
integer	identifier is a reserved word
why?	identifiers must be only letters, number, or underscore characters
24-bit-mode	same reason as above

Figure 7.14: Invalid Identifiers in OMAR

CASE SENSITIVITY

Identifiers and reserved words are **not case sensitive** which means that capital letters and small letters are not considered to be different. This means, for example, that the identifier, "goo", is the same as the identifier "GOO", "Goo", or "GoO".

SPECIAL SYMBOLS

In various places in the language, special symbols are required. These symbols may not be used in any other parts of the program where they are not specifically required and may not be used as parts of identifiers.

```
(       )       [       ]       ;       :       <       >       +
*       /       =       ^       {       }       ..      .       -
```

Figure 7.15: Special Symbols in OMAR

LANGUAGE FORMATTING

Like most modern languages, the OMAR language uses a "free-format". This means that the amount of whitespace between the keywords and identifiers makes no difference in the meaning of the program. For example, the following two code examples have different formatting, but have the exact same meaning and will produce the same results. This is different from the scripting language Python, for example, where the formatting actually impacts the program's meaning.

```
while (counter < 10) do                    while (counter < 10) do counter = itself + 1; end;
        counter = itself + 1;
end;
```

Figure 7.16: Free Formatting In OMAR

A SIMPLE EXAMPLE

To illustrate a few specific features and also what the OMAR language looks like in general, we'll look at a simple example.

EXAMPLE: "COUNTDOWN.OMAR"

```
do countdown;

// This example counts down from 10 to 1
//
verb countdown is                          // A declaration of a subprogram
        integer counter;                   // A variable declaration

        counter = 10;                      // An instruction
        while counter > 0 do               // More instructions!
                write counter, ;
                counter = itself - 1;
        end;
        write "blast-off! ", ;
end;    // countdown
```

Figure 7.17: A Simple Example

THE INCLUDE DIRECTIVE

Creating a program that is contained inside of a single .OMAR file is fine for small programs, but when a program becomes more complex than the most basic example, one should break the program into a series of files. The "include" directive is used to include code from another file.

```
do example;

include "other_file.ores";          // The file "other_file.ores" contains declarations that can be
                                    // referenced in this file now that the file has been included.
...
```

Figure 7.18: The Include Directive

NESTED INCLUDES

Often, we include a file that includes other files. This can be thought of as hierarchy (tree) of files. When you include a file that includes other files, it has the effect of including all of the files that are referenced in this tree. This makes it easy to include quite a large number of files just by using a single include directive.

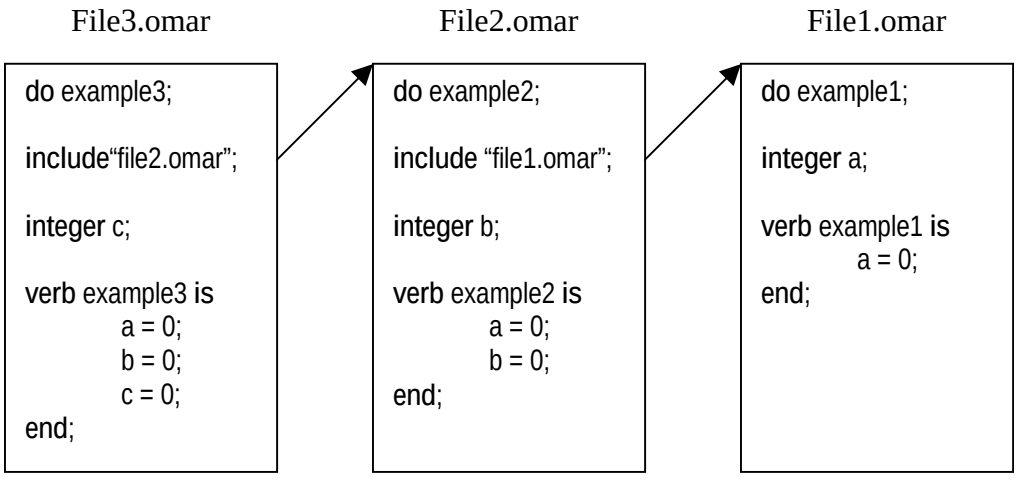


Figure 7.19: Nested Includes

SYSTEM INCLUDES

When files are included, they are usually either part of the current project or they are files that are provided as part of Hypercosm Studio. These Hypercosm Studio files are called "system includes". They are located in the "system includes" directory, "Hypercosm Studio/Includes/". In order to cause the compiler to look in the "system includes" directory for a file, preface the name of the file in the include directive with the word "system/".

EXAMPLE: "USING_TRIGONOMETRY.OMAR"

```
do use_trig;

include "system/trigonometry.ores";          // The file "trigonometry.ores" contains the definition of the
                                             // sine function used below along with other
verb use_trig is                              // trigonometric functions
        write "sin 30 = ", sin 30, ;
end;
```

Figure 7.20: A System Include Directive

NATIVE INCLUDE FILES

When perusing the list of .ORES include files that are provided, you may notice that some of them are prefixed by the word "native". These "native" include files are special because they contain a series of "native" methods, variables, and type declarations.

NATIVE DECLARATIONS

"Native" declarations are special because their implementation is provided not by the script code in the file, but by "native" code that must exist inside the interpreter that executes the applet. Because the implementation of these native entities is part of the Hypercosm Player, these native definitions should not be tampered with.

```
native verb pause_applet
        for integer milliseconds;
end;
```

Figure 7.21: An Example Native Method Declaration

File Name	File Contents
native_bitwise.ores	Contains functions to manipulate data at the bit level
native_chars.ores	Contains native character functions
native_cursor.ores	Contains the native mouse cursor changing method
native_data_container.ores	Contains a container used by native data packing
native_data_packer.ores	Contains utilities for packing primitives to raw bytes
native_data_resources.ores	Contains a definition of downloadable data resources
native_data_unpacker.ores	Contains utilities for unpacking primitives from bytes
native_dates.ores	Contains a function for finding the current date
native_devices.ores	Contains utilities for dealing with native events
native_display.ores	Contains native display properties and query methods
native_exec_script.ores	Contains a method to call Javascript in a web page
native_images.ores	Contains the native image definition and methods
native_ip_address.ores	Contains a native IP address definition
native_keyboard.ores	Contains native keyboard querying methods
native_lights.ores	Contains the native lighting primitive definitions
native_links.ores	Contains native URL based hyper-linking methods
native_materials.ores	Contains the native definitions of materials
native_math.ores	Contains native math functions
native_messages.ores	Contains native message handling functions
native_mouse.ores	Contains native mouse querying functions
native_nurbs_curve.ores	Contains native functions for handling curves
native_overlay_text.ores	Contains the native overlay text drawing method
native_picking.ores	Contains native utilities for doing picking
native_proximity.ores	Contains native utilities for sensing proximity
native_rendering.ores	Contains the native rendering properties
native_sensing.ores	Contains native functions for collision detection
native_shape_colors.ores	Contains native definitions of default shape colors
native_shapes.ores	Contains native definitions of the primitive shapes
native_sounds.ores	Contains native definitions of sounds
native_tcp_client_socket.ores	Contains a native client socket networking definition
native_tcp_server_socket.ores	Contains a native server socket networking definition
native_textures.ores	Contains native texture definitions
native_time.ores	Contains native time functions
native_trans.ores	Contains native definitions of transformations
native_trans_stacks.ores	Contains native transformation stack utilities
native_udp_socket.ores	Contains native UDP networking definitions
native_untextured_materials.ores	Contains native definitions for untextured materials
native_viewing.ores	Contains native definitions of viewing paramaters

Figure 7.22: Native ".ORES" files

LESSON 7:

VARIABLES AND DATA TYPES

LESSON OBJECTIVES:

- Understand how to store data using the primitive data types
- Know what types of operations can be performed on each primitive data type

LESSON CONTENTS:

- Variable declarations
- Primitive data types
- Constant declarations

COPYRIGHT©2006

VARIABLES

Variables are used to store data which is used by a computer program. They are, in effect, the representation of the world inside of the computer. Variables can be used to represent such things as color, temperature, positions of objects - almost anything that can be measured quantitatively or represented symbolically. Once data is stored in variables, it can be retrieved, examined and changed. Each variable that we use must be declared before we can use it. The variable declaration is where the variable is "born" and given a name. The name given to the variable must be unique so that there is no confusion as to which variable we are referring to. Some variables also have initial values, while others remain undefined until they are given a value at some point in the future. To create a new variable, we use the following format:

```
<data type name> <variable name> <optional initializer> ;
```

Figure 8.1: Variable Declaration Syntax

EXAMPLE:

```
integer counter = 0;
scalar temperature = 98.7;
double pi = 3.1415926535897932384;
scalar speed_of_light = 3 E 8;
string name = "Willy";
vector location = <5 0 100>;
```

Figure 8.2: Example Variable Declarations

VARIABLE DECLARATIONS

Variable declarations can only be placed in certain locations in the program. Either they will lie completely outside of any subprograms or they will be found at the very beginning of the subprogram. A variable declaration will never be found right in the middle of a group of statements. They always come at the beginning of a block of code.

DATA TYPES

Each variable is said to be of a certain data "type". The type determines what kind of data the variable can store. Once the variable is created, it can never change its type. A set of basic, or "primitive" data types is defined by the OMAR language. More complex data types can be built up from these primitive types. The primitive data types that are recognized by the OMAR programming language are the following: boolean, char, byte, short, integer, scalar, double, complex, and vector. Generally, OMAR's set of primitive types is very similar to the primitive types available in Java or C#. The main exceptions are (1) OMAR has a special "vector" type for efficiently handling 3D vector data and (2) there is no special primitive string type as there is in Java. In OMAR, strings are implemented as arrays of characters. A more detailed description of the properties of each of these data types is given in the next section.

Type Name	Contents	Size	Min Value	Max Value
Boolean	True Or False	1 Bit	N.A	N.A.
Char	Unicode Char	16 Bits / 2 Bytes	Chr(0)	Chr(32767)
Byte	Signed Integer	8 Bits / 1 Byte	-128	127
Short	Signed Integer	16 Bits / 2 Bytes	-32768	32767
Integer	Signed Integer	32 Bits / 4 Bytes	-2.147 Billion	2.147 Billion
Long	Signed Integer	64 Bits / 8 Bytes	-9.223 Quintillion	9.223 Quintillion
Scalar	Single Precision Floating Point Number	32 Bits / 4 Bytes	+/- 1.402 E -45	+/- 3.40 E 38
Double	Double Precision Floating Point Number	64 Bits / 8 Bytes	+/- 4.94 E -324	+/- 1.79 E 308
Complex	A Pair of Scalars	64 Bits / 8 Bytes	Same As Scalar	Same As Scalar
Vector	A Triplet of Scalars	96 Bits / 12 Bytes	Same As Scalar	Same As Scalar

Figure 8.3: Primitive Data Types

COMMONLY USED DATA TYPES

Although a variety of types are provided for completeness, in practice we find that the vast majority of scripting code uses the following five data types:

```
boolean
char
integer
scalar
vector
```

Figure 8.4: Most Commonly Used Data Types

BOOLEAN

A boolean value is either true or false. These values are useful for describing such things as whether something is on or off or any other situation where there are only two states. The constants, true and false, are predefined and represent the two possible boolean values. Two boolean values may be combined using boolean operators to yield a boolean result. If an expression involving both ands and ors is evaluated, then the and operator takes precedence over the or operator. For example, the expression, (a and b or c) is evaluated as ((a and b) or c).

Operator	Purpose
and	A logical operator that returns a true value only if both operands are true.
or	A logical operator that returns a true value if one or both operators are true.
not	A logical operator that returns a false value if the operand is true and a true value if the operand is false.

Figure 8.5: Boolean Operators

EXAMPLE:

```
boolean done is false;
boolean overflow is (number > limit);
boolean error is (number = 0) and not found;
boolean fractional is (a <> trunc a);
```

Figure 8.6: Example Boolean Variable Declarations

CHAR

A char is a variable that is used to represent a single character symbol. Examples of characters are the letters of the alphabet "a" through "z", the capital letters of the alphabet "A" through "Z", the characters representing the digits "0" through "9", and special symbols such as the period, ".", or the semicolon, ";". To denote a particular printable character, place the character's printable symbol within double quotes. In addition, a character may also be a special non printable symbol which has some special meaning to the computer such as tab, space, or carriage return. Since these characters are non-printable, they are specified by integer values according to the ASCII standard. Non-printable characters can be created by passing an integer ASCII value into the "chr" function. Commonly used characters and character manipulation functions are defined in the file "chars.ores" which is located in the directory "Hypercosm Studio/Includes/Utilities/Strings/".

EXAMPLE:

```
char ch is "A";                          // a printable character
char tab is chr 9;                       // a non-printable character
char name[] is "Fred Freugelbugger";     // an array of chars
```

Figure 8.8: Example Char Variable Declarations

STRINGS OF CHARS

Often, we need a way of creating a variable to hold a list, or "string", of characters. Strings are implemented as arrays of chars. You can define a type "string" to stand for the type "array of chars" so you don't have to type the array indicating brackets each time. To assign a string as a unit, use the "is" operator. To assign a string character by

character, use the "=" operator. When assigning strings using the "=" operator, the strings must be of the same size or else a run time error will result. For this reason, it is usually preferable to use the "is" operator instead.

EXAMPLE:

```
type string is char[];

string type name is "Barney Squidsnorker";
string type names[] is ["Bob" "Mary" "Joe" ];        // An array of strings (equivalent to an array of
                                                     // arrays of chars - char[][])
```

Figure 8.9: Example String Variable Declarations

INTEGER

Integers are the familiar counting numbers (1, 2, 3, etc.) and are useful for representing things that can have only whole number values such as the number of characters in a file or the number of objects in a list. Integers may be combined in expressions utilizing the integer operands to yield an integer result.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
div	Division
mod	Modulo (Remainder)
- (Unary)	Negation

Figure 8.10: Integer Operators

EXAMPLE:

```
integer i = 0;
integer goo = -1000;
integer value = trunc 3.1415;
integer a = -value;
integer b = 10 + 5 * a;
```

Figure 8.11: Example Integer Variable Declarations

SCALAR

Scalars are values that represent a continuous range of numbers, like height, temperature, or radius, where there can be a theoretically infinite number of intermediate values. In the mathematical world, these are known as real numbers. Computer scientists refer to the method of representing real numbers in computers as "floating-point" arithmetic. There are a variety of operators that can take integer and scalar operands and produce a scalar result. Since integers are a subset of scalars, the integers can automatically be converted to scalars where the situation warrants it.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
- (Unary)	Negation

Figure 8.12: Scalar Operators

EXAMPLE:

```
scalar temperature = 32.0;
scalar infinity = 1e10;
scalar pi = 3.1415926;
scalar x = 0;
scalar length = sqrt 2;
scalar a = (-pi / 4) + length * x;
```

Figure 8.11: Example Scalar Variable Declarations

VECTOR

A vector is simply a set of three scalars. We can represent many concepts in the real world by these triplets of three numbers. We say that the universe is three dimensional because any point or direction can be described with respect to a certain frame of reference by three numbers.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication by a scalar or a vector
/	Division by a scalar or a vector
dot	Dot (scalar) product
cross	Cross (vector) product
parallel	Parallel component of a vector
perpendicular	Perpendicular component of a vector
- (Unary)	Negation (reverse direction)

Figure 8.12: Vector Operators

EXTRACTING THE COMPONENTS OF A VECTOR

The three components of the vector are called the x, y, z components of the vector. They can be extracted by using the dot product operator or by using the dot (.) operator.

Field of Vector	Expression
X component	V **dot** <1 0 0> or V.x
Y component	V **dot** <0 1 0> or V.y
Z component	V **dot** <0 0 1> or V.z

Figure 8.13: Extracting the Component of a Vector

EXAMPLE:

```
vector location = <10 –30 20>;
vector direction = location cross -<1 2 5>;
vector z_component = direction parallel <0 0 1>;
```

Figure 8.14: Example Vector Variable Declarations

CONSTANTS

Constants are just like variables except that their values can not be changed. All constants must be given an initial value which is permanent. Constants may be declared any place that variables are declared. The format for declaring constants is similar to variables.

```
const <data type name> <variable name> <optional initializer> ;
```

Figure 8.15: Constant Declaration Syntax

EXAMPLE:

```
const scalar freezing_point = 32;
const scalar pi = 3.14159265;
const scalar e = 2.718281828;
const boolean on is true, off is false;

pi = 3.14159265;                    // Error - a constant may only be assigned by its initializer
```

Figure 8.16: Example Constant Declaration

LESSON 8:

SIMPLE STATEMENTS IN OMAR

LESSON OBJECTIVES:

- Understand what statements are available in OMAR
- Understand how to use these statements
- Understand how to combine statements together

LESSON CONTENTS:

- Assignment statements
- Conditional statements
- Looping statements

COPYRIGHT©2006

WHAT STATEMENTS ARE USED FOR

We've previously seen how to represent data by declaring variables of different types. However, just representing data isn't very interesting. In order to make the program dynamic, we need to have commands or instructions to do something with that data. The simplest units of these instructions are called "statements". Statements are the core building blocks that make up the instructions and algorithms that are the heart of any computer program.

THE ASSIGNMENT STATEMENT

The most basic statement is the assignment statement. It is used to assign a value to a variable. The value of the variable is specified by an expression. The expression may be as simple as the name of another variable of the same type or as complex as a mathematical formula involving many different terms. It is considered an error to assign variables to themselves since this effectively does nothing. The basic form of the assignment statement is as follows:

```
<variable> <assignment operator> <expression> ;
```

Figure 9.1: Assignment Statement Syntax

ASSIGNMENT OPERATORS

Unlike many other popular languages, in OMAR there are different assignment operators used to assign different types. The first assignment operator, "=", is used to assign numerical types and the contents of structures and arrays. Another assignment operator, "is", is used to assign symbolic types which can only assume a limited set of values. These types include boolean, char, enumerated types, and references to structures and arrays.

Data Type	Example	Assignment Operator
Numerical Type	short, byte, integer, long, scalar, double, complex, vector	=
Symbolic Type	boolean, char, or structure or object reference	is

Figure 9.2: Assignment Operators

```
integer a;
boolean done;

a = 10;
done is false;
```

Figure 9.3: Assignment Statement Examples

EXPRESSIONS

Expressions are formed by evaluating a sequence of operators or functions and their operands to yield a value as a result. The types of the operators will be checked to make sure that they match the operators and functions that are used. The functions may be built in functions or user defined functions.

Boolean Expressions	done and not found (a > b) (a <> b) done is true; a = 5
Numerical Expressions	a + b sqrt (a*a + b*b)

Figure 9.4: Example Expressions

OPERATOR PRECEDENCE

The order in which the operators are applied depends upon the rules of precedence. Operators with the highest precedence will be applied first followed by operators of lower precedence. Operators of the same precedence will be applied from left to right. The precedence rules can always be overridden by adding parentheses around the expressions that are to be evaluated first. The operator precedence used in OMAR is similar to other languages. See the OMAR language reference for details.

```
2<3<4 * 5                  = (2 < 3) and (3 < (4*5))     =    true
4 * 5+3 * 3                = (4*5) + (3*3)               =    29
4*2 ^ 2                    = 4 * (2^2)                   =    16
true and false or true     = (true and false) or true    =    true
```

Figure 9.5: Operator Precedence

PRONOUNS: ITS AND ITSELF

In English, we find that we frequently need to refer to an idea repeatedly. To do so, we use a pronoun to substitute for a thing that was previously mentioned. In OMAR, we have introduced two pronouns, "itself" and "its" to accomplish this same purpose. The pronoun "itself" is used to refer to whatever the most previous expression referred to. The pronoun, "its", is used with structures and is used to refer to a field of a structure denoted by an expression.

EXAMPLE:

```
integer i = 1, iterations = 0;
boolean parity is false;
linked type chain is linked_list;

while some chain do
        i = itself * 2;
        iterations = itself + 1;
        parity is not itself;
        chain is its next;
end;
```

Figure 9.6: Pronoun Examples

THE "WRITE" STATEMENT

One of the most common things that we need to be able to do is to write out messages that can tell us about the status of the running applet. These messages are used to show the flow of execution through the program or show the status of variables. The "write" statement is provided for this purpose. When the "write" statement executes, the output will be displayed in the "Output" window at the bottom of the Hypercosm Studio user interface. These output statements will not be visible when the applet runs outside of Hypercosm Studio. The "write" statement can take arguments of all of the primitive types, char, byte, short, integer, long, scalar, double, complex, and vector plus arrays of characters, which are also known as "strings". The form of a "write" statement is as follows:

```
write expr1, expr2, expr3, ... exprN ;      // Where expr1 ... exprN are
                                            // expressions which evaluate to
                                            // a primitive type or an
                                            // array of chars (string)
```

Figure 9.7: "Write" Statement Syntax

WRITING A NEW LINE

If the write procedure is called with no arguments, then it will print a new line character. Usually, when output messages are printed, we want to go to the next line at the end of the message, so usually the last argument in a "write" statement is an empty argument.

EXAMPLE:

```
integer N = 10, a = 64;
scalar b = 64;
complex c = <4 1>;

write "hello world!", ;
write "the value of the variable, N, is ", N, ;
write "the values of a, b, and c are ", a, ", ", b, ", and ", c, ;
```

Figure 9.8: "Write" Statement Examples

CONDITIONAL STATEMENTS

One of the most basic things that a computer can do is test for some condition and take a different course of action depending on the outcome. Frequently, conditionals are nested in a complex sequence. If the execution of the program is represented as a flow chart, then the conditionals occur whenever there is a "fork in the road" or a place where the path branches into multiple paths.

THE "IF-THEN" STATEMENT

The most basic form of conditional is the "if-then" statement. The "if-then" statement relies upon solving a boolean expression that determines whether or not a certain action is taken. If the boolean expression evaluates to true, then the statements are executed. If the boolean expression evaluates to false, then nothing happens and the computer goes on to the next statement. The basic form of the "if-then" statement is as follows:

```
if <boolean expression> then
        <declarations>
        <statements>
end;
```

Figure 9.9: "If-Then" Statement Syntax

EXAMPLE:

```
if (sqr b - 4 * a * c < 0) then
        write "no roots found", ;
end;
```

Figure 9.10: "If-Then" Statement Example

THE "IF-THEN-ELSE" STATEMENT

A more complex form of the "if-then" statement is the "if-then-else" statement. This statement works by deciding upon one of two possible courses of action based on the value of the boolean expression. If the boolean expression evaluates to true, then the first block of statements is executed, else, the second block of statements is executed. The form of the "if-then-else" statement is as follows:

```
if <boolean expression> then
        <declarations1>
        <statements1>
else
        <declarations2>
        <statements2>
end;
```

Figure 9.11: "If-Then-Else" Statement Syntax

THE "IF-THEN-ELSEIF" STATEMENT

Another useful form of the "if" statement is used when we want to perform a series of tests and execute some statements as soon as we find a condition that evaluates to true, otherwise, keep testing. This statement is called the "if-then-elseif" statement. The form of this statement is as follows:

```
if <boolean expression1> then
        <declarations1>
        <statements1>
elseif <boolean expression2> then
        <declarations2>
        <statements2>
        .
        .
elseif <boolean expressionN> then
        <declarationsN>
        <statementsN>
end;
```

Figure 9.12: "If-Then-Elseif" Statement Syntax

THE "IF-THEN-ELSEIF-ELSE" STATEMENT

The last possible form of the "if-then" statement is created by fitting an "else" clause onto the "if-then-elseif" statement. This is useful when we want to perform a series of tests that each execute some statements if true, and if none of the conditions are true, then execute the statements inside the else block. This is the "if-then-elseif-else" statement, which is illustrated below:

```
if <boolean expression1> then
        <declarations1>
        <statements1>
elseif <boolean expression2> then
        <declarations2>
        <statements2>
        .
        .
elseif <boolean expressionN> then
        <declarationsN>
        <statementsN>
else
        <declarations>
        <statements>
end;
```

Figure 9.13: "If-Then-Elseif-Else" Statement Syntax

LOOPING STATEMENTS

Probably the most powerful construct in computer programming is the loop. A loop is a means of repeatedly executing some action. The looping structures provided enable the program to do something a fixed number of times or until a certain condition is satisfied.

THE "WHILE" STATEMENT

The "while" statement is used to repeatedly execute a sequence of statements as long as a particular condition holds true. It is used whenever we don't know how many times the loop will have to be executed. This kind of loop is used for things like: while not finished, do task; while no character has been found, read keyboard, etc. The condition is tested for at the beginning of the loop, before any statements are executed, so it is possible for the statements not to be executed at all.

```
while <boolean expression> do
        <declarations>
        <statements>
end;
```

Figure 9.14: "While" Statement Syntax

EXAMPLE:

```
integer i = 10;

while (i > 0) do
        write "i = ", i, ;
        i = itself – 1;
end;
```

Figure 9.15: "While" Statement Example

THE "FOR" STATEMENT

The "for" statement is used whenever we want to execute the statements of a loop a fixed number of times. The variable that is used to count the number of iterations in the loop must be of an integral type and may be used in expressions inside of the loop but its value may not be changed. The start expression and end expression must also evaluate an integer.

```
for <type> <counter> <assignment_operator> <start expr> .. <end expr> do
        <declarations>
        <statements>
end;
```

Figure 9.16: "For" Statement Syntax

EXAMPLE:

```
for integer counter = 1 .. 10 do
        write "counter = ", counter, ;
end;
```

Figure 9.17: "For" Statement Example

LESSON 9: PROCEDURES AND FUNCTIONS

LESSON OBJECTIVES:

- Understand the different types of subprograms
- Understand how to declare parameters for a subprogram
- Understand the different types of parameter declarations
- Understand how to call a subprogram and how to pass in values for the parameters

LESSON CONTENTS:

- Procedures and functions as “black boxes”
- Scoping
- Parameters
- Mandatory Parameters
- Optional Parameters
- Keyword Parameters
- Reference Parameters
- Scoping Modifiers

THE BLACK BOX PHILOSOPHY

Subprograms are a way of breaking down code into manageable pieces. In order to solve any complex problem, we must find ways of breaking down the task into manageable units. This philosophy is exploited tirelessly in science and computers. The idea is often illustrated by visualizing each component of the solution as a black box, where we know what goes into the box and what comes out but have no knowledge of what goes on inside. Ideally, we should have no need to know what goes on inside the black box.

THE CONCEPT OF SCOPING

Just as an engine has a number of internal parts that are necessary for it to work but are not used by any other parts of the car, a subprogram can also have its own data, data types, and even its own subprograms which are not accessible to the rest of the program. If something is accessible from a particular place in the program, then we say that it is "visible". All the things which are visible to a subprogram comprise the "scope" of the subprogram. The visibility of program declarations is determined by the textual arrangement of the program. Each time we begin a new subprogram, we can declare new things which are not visible to the rest of the program. These things are referred to as "local" because they can only be accessed within that subprogram. In addition to the things in its own local scope, a subprogram can also have access to things which are declared in the scopes which enclose itself. If two variables with the same name but from different scopes are visible, then the one with the closer scope takes precedence and the others are invisible. Usually, subprograms will not be nested, so they will have access only to their own scope and to the global scope.

EXAMPLE: "SCOPING.OMAR"

```
do a, b;

// Global scope
//
integer i = 0;

verb a
is      // Beginning scope of a -scopes visible: global scope, scope of a
        //
        integer j;

        // Statements of a
        //
        j = i;
end;    // Ending scope of a

verb b
is      // Beginning scope of b - scopes visible: global scope, scope of b (but not a)
        //
        integer k;

        verb c
        is      // Beginning scope of c- scopes visible: global scope, scopes of b and c (but not a)
                //
                integer l;

                // Statements of c
                //
                l = i;
                k = l;
        end;    // Ending scope of c

        // Statements of b
        //
        k = i;
end;    // Ending scope of b
```

Figure 10.1: Scoping

PROCEDURES

Procedures are subprograms that are called upon to perform a particular subtask. In the OMAR language, procedures are called "verbs". There are two parts to using procedures. First, the procedure must be declared. The declaration tells what the procedure does, how it is supposed to do it, and how to call the procedure. Once the procedure is declared, we can call it to run the code inside of the procedure. To call a simple procedure, simply state the name of the procedure.

```
verb <procedure name> is
        <declarations>
        <statements>
end;
```

Figure 10.2: Simple Procedure Declaration Syntax

```
<procedure name> ;
```

Figure 10.3: Simple Procedure Call Syntax

EXAMPLE: "PROCEDURES.OMAR"

```
do write_averages;

integer n1, n2;

verb write_average is                          // procedure declaration
        write "average = ", (n1 + n2) / 2, ;
end;    // write_average

verb write_averages is
        n1 = 10;
        n2 = 30;
        write_average;                         // procedure call

        n1 = 15;
        n2 = 7;
        write_average;                         // procedure call
end;    // write_averages
```

Figure 10.4: Simple Procedure Example

FUNCTIONS

Often, subprograms are used to do a small part of a larger calculation. In these instances, it is desirable to have the subprogram return a value. This type of subprogram is known as a function. In the OMAR language, functions are called "questions". Function declarations are similar to procedure declarations except that we must precede the declaration with the name of the type of data to return. Also, inside of the body of a function, we must include an "answer" statement to tell the function to end and return a value to the caller. The "answer" statement in a function must be followed by an expression of the same data type as the type that is returned by the function. The last statement in a function must be either an answer statement or a conditional statement where all conditions end in an answer statement.

```
<return type> question <function name>
is
        <declarations>
        <statements>
end;
```

Figure 10.5: Simple Function Declaration Syntax

EXAMPLE: "FUNCTIONS.OMAR"

```
do find_averages;

integer n1, n2;

scalar question get_average is                  // function declaration
        answer (n1 + n2) / 2;                   // answer statement
end;    // get_average

verb find_averages is
        n1 = 10;
        n2 = 30;
        write "average = ", get_average, ;      // function call

        n1 = 15;
        n2 = 7;
        write "average = ", get_average, ;      // function call
end;    // find_averages
```

Figure 10.6: Simple Function Example

PARAMETERS

We have seen how to break the program down into manageable parts with subprograms. What is needed is a way for all of these individual pieces to be neatly connected so that the pieces are independent but work together. Parameters are the means by which the various parts of the program are interconnected. They allow us to pass data into and out of each subprogram. They are like the wiring connecting a series of black boxes.

THE PROBLEM WITH GLOBAL VARIABLES

The black box analogy of program structure assumes that there is some way to get the data into the black box and out of the black box. In our previous examples, we used global variables to communicate with the subprogram. Although this works fine, it is not considered good programming style because:

1) **It's not easy to tell how to use the subprogram**
   We can't tell how to interface with the subprogram simply by looking at the header of the subprogram. Instead we must actually examine the code to see how it works.
2) **The subprogram relies on the existence of global data.**
   A subprogram that relies upon global data can't stand alone as its own unit. If we change the definition of the global data, then the subprogram will break. This violates the primary objective of black boxes that we are trying to achieve.

THE SUBPROGRAM INTERFACE: PARAMETERS

The solution to our problems is to specify a way to send values to the subprogram and receive values back to the main program. This is done with parameters. Parameters are simply variables that belong to the subprogram and are changed each time the subprogram is entered to run it with different initial conditions. Parameters are declared immediately after the subprogram name and before the local declarations. Parameters are declared and used in the same way with any type of subprogram (procedures or functions).

```
verb <procedure name>
        <parameter declarations>
is
        <declarations>
        <statements>
end;
```

Figure 10.7: Parameter Declaration Syntax

DIFFERENT TYPES OF PARAMETERS

One of the main differences between OMAR and other languages is that it provides a variety of different ways of passing parameter values. In the next few sections, we will examine these different parameter types in detail. The following description provides a brief overview of some of the unique parameter passing styles available in OMAR.

OPTIONAL PARAMETERS

In many cases, there are logical default values for parameters to have. For example, it's natural for a sphere to have a default radius of 1. Another example might be to define a car object with doors that open and close depending upon the value of a parameter. Since it's logical to have the doors default to the closed position, this is a good choice for an optional parameter.

KEYWORD PARAMETERS

In some cases, the procedure call can be stated very naturally by relying on the fact that the values of certain parameters are often proceeded by keywords. For example, if we want to create an arrow, it's natural to say "arrow from <here> to <there>" where the values of the endpoints of the arrow are indicated by the keywords "from" and "to". When we call the subprogram, we expect to find the values of the endpoints in the places that are indicated by the keywords "from" and "to".

RETURN VALUE PARAMETERS

The parameter is like an access road to the procedure. Sometimes, we only need a one way street with data going into the procedure but not returning. In other circumstances, we need a two way street where data can be passed into and return from the procedure.

MANDATORY PARAMETERS

As their name implies, mandatory parameters are parameters that must be supplied to the subprogram. Mandatory parameters are the standard way of passing parameters in most standard programming languages such as C, C++, or Java. An example of mandatory parameters is the parameters that are used in the built in function, sin. To call the sin function, you must supply an argument immediately following the name of the function. According to the declaration of sin, the argument must take a scalar value. If no argument is supplied, an error message will be issued. Mandatory parameters are used whenever there is no logical choice for default values for the parameters.

DECLARATION OF MANDATORY PARAMETERS

Mandatory parameters are specified in the subprogram declaration by listing their declarations immediately following the name of the subprogram. There is no limit to the number of parameters that may be declared.

```
verb <procedure name>
        <type name> <parameter name> ;
        …
is
        <declarations>
        <statements>
end;
```

Figure 10.8: Mandatory Parameter Declaration Syntax

ASSIGNMENT OF MANDATORY PARAMETERS

When the subprogram with mandatory parameters is called, a list of expressions is expected following the name of the subprogram. The expressions must evaluate to the proper type to be assigned to the parameters that they correspond to (for example, integer expressions for integer parameters, string expressions for string parameters etc). The number of expressions for parameter values in the procedure or function call must match the number of parameter declarations in the procedure or function declaration.

```
<procedure name> <parameter values> ;
```

Figure 10.9: Mandatory Parameter Assignment Syntax

EXAMPLE: "MANDATORY_PARAMS.OMAR"

```
do find_averages;

include "system/math.ores";

scalar question get_average
        integer a, b;                           // mandatory parameter declaration
is
        answer (a + b) / 2;
end;    // average

verb find_averages
is
        scalar a;

        a = get_average 30 40;
        a = get_average (round 3.5) 40;
        a = get_average -10 40;

        a = get_average 30 .5;                  // Compile Error! - .5 is not an integer
        a = get_average (sqrt 10) 40;           // Compile Error! - sqrt 10 is not an integer
        a = get_average 30 40 15;               // Compile Error! - too many parameters
end;    // find_averages
```

Figure 10.10: Mandatory Parameter Example

NOTES:

Most mathematical functions use mandatory parameters because there usually aren't any default values that make sense for these types of functions.

EXAMPLE:

```
scalar question cos
        scalar angle;
end;

x = cos 60;
```

OPTIONAL PARAMETERS

Optional parameters are useful when it is not necessary to always specify all of the parameters. For optional parameters, default values must be specified in the parameter declaration. If parameter values are not specified in the procedure call, then the parameter will take on the default values specified in the parameter declaration.

DECLARATION OF OPTIONAL PARAMETERS

To signify the beginning of the optional parameters section, add the keyword, "with" at the end of the mandatory parameters section followed by the declarations of the optional parameters. All optional parameter declarations must be followed by an initializer that assigns a default value.

```
verb <procedure name>
        <mandatory parameter declarations>
with
        <type name> <parameter name> <initializer> ;        // optional parameter declarations
        …
is
        <declarations>
        <statements>
end;
```

Figure 10.11: Optional Parameter Declaration Syntax

ASSIGNMENT OF OPTIONAL PARAMETERS

Optional parameters are assigned by following the name of the procedure by the keyword "with" and then listing a number of parameter assignment statements until the keyword "end" is given.

```
<procedure name> with
        <parameter name> <initializer> ;
        …
end;
```

Figure 10.12: Optional Parameter Declaration Syntax

EXAMPLE: "OPTIONAL_PARAMS.OMAR"

```
do init_arrays;

verb init_array
        integer array[];                    // Mandatory parameter for table
with
        integer value = 0;                  // Optional parameter for table contents
is
        for each integer i in array do
                i = value;
        end;
end;    // init_array

verb init_arrays
is
        integer table[1..10];

        init_array table;                   // Fill table with 0s (default value)
        init_array table with               // Fill table with -1s
                value = -1;
        end;
end;    // init_arrays
```

Figure 10.13: Optional Parameters Example

NOTES:

Many graphical primitives use optional parameters because they often have default parameter values. For example, a sphere has an optional parameter for the radius that has a default value of 1.

MANDATORY KEYWORD PARAMETERS

Sometimes it is desirable to require extra words to be inserted into the procedure call to make the procedure call more readable. These extra words are known as keyword parameters.

DECLARATION OF MANDATORY KEYWORD PARAMETERS

Mandatory keyword parameters are declared just like the mandatory parameters except that before the variable declaration comes one or more special identifiers that are the keywords. Any identifier can be used as a keyword so long as it's not a reserved word. The keywords signify that the value of the parameter will follow.

```
verb <procedure name>
        <keyword> <type name> <parameter name> ;
        …
is
        <declarations>
        <statements>
end;
```

Figure 10.14: Mandatory Keyword Parameter Declaration Syntax

ASSIGNMENT OF MANDATORY KEYWORD PARAMETERS

To assign values to keyword parameters, state the keyword followed by an expression which can be evaluated to provide a value for that keyword. The parameter keyword / value pairs must be given in the order as they are listed in the declaration.

```
<procedure name> <keyword / parameter value pairs> ;
```

Figure 10.15: Mandatory Keyword Parameter Assignment Syntax

EXAMPLE: "MANDATORY_KEYWORD_PARAMS.OMAR"

```
do init_arrays;

verb init_array
        integer array[];                    // Mandatory parameter for table
        to integer value;                   // Mandatory keyword parameter for table contents (no default)
is
        for each integer i in array do
                i = value;
        end;
end;    // init_array

verb init_arrays is
        integer table[1..10];

        init_array table to 0;              // Fill table with 0s
        init_array table to -1;             // Fill table with -1s
end;    // init_arrays
```

Figure 10.16: Mandatory Keyword Parameters Example

NOTES:

Mandatory keyword parameters are used quite frequently in the definition of various graphical utility functions because they make the graphics code very easy to read. For example a number of the transformation methods use keyword parameters effectively:

EXAMPLE:

```
verb rotate
        by  scalar angle;
        around vector axis;
end;

rotate by 60 around <0 0 1>;
```

OPTIONAL KEYWORD PARAMETERS

Optional keyword parameters are like a hybrid between the optional parameters and the keyword parameters previously discussed. Optional keyword parameters are used when it is desirable to use keywords in the procedure call but not to require all the parameters to be given. In this case, we can use the presence of the keyword to signify that we are overriding the default parameter value by providing a new value. Since it is possible that optional keyword parameters are not assigned in the procedure call, they must always have default values.

DECLARATION OF OPTIONAL KEYWORD PARAMETERS

Optional keyword parameters are declared just like the mandatory keyword parameters except that they are given initializers. Optional keyword parameters are declared by giving the keyword followed by the variable declaration followed by the initializer.

```
verb <procedure name>
        <keyword> <type name> <parameter name> <initializer> ;
        …
is
        <declarations>
        <statements>
end;
```

Figure 10.17: Optional Keyword Parameter Declaration Syntax

ASSIGNMENT OF OPTIONAL KEYWORD PARAMETERS

The optional keyword parameters are assigned like the mandatory keyword parameters except that the order that the keywords and parameters values come in is flexible and any or all of the parameter assignments may be omitted.

```
<procedure name> <keyword / parameter value pairs> ;
```

Figure 10.18: Optional Keyword Parameter Assignment Syntax

EXAMPLE: "OPTIONAL_KEYWORD_PARAMS.OMAR"

```
do init_arrays;

verb init_array
        integer array[];                    // Mandatory parameter for table
        to integer value = 0;               // Optional keyword parameter for table contents (with default)
is
        for each integer i in array do
                i = value;
        end;
end;    // init_array

verb init_arrays
is
        integer table[1..10];

        init_array table;                   // Fill table with 0s (the default value)
        init_array table to 0;              // Fill table with 0s
        init_array table to -1;             // Fill table with -1s
end;    // init_arrays
```

Figure 10.19: Optional Keyword Parameters Example

REFERENCE PARAMETERS

It is sometimes necessary to return data from a subprogram. This is most easily done using reference parameters. Reference parameters in OMAR are similar to reference parameters in C++ or "var parameters" in Pascal or Delphi. Note that Java has no analogue to reference parameters since Java has no generalized reference or pointer data type.

DECLARATION OF REFERENCE PARAMETERS

Reference parameters are declared just like mandatory parameters except that the reserved word, "reference" precedes the parameter name. When this is done, the parameter becomes a two-way link to the variable which is passed in so any changes that are made to the parameter will be reflected in the variable when the procedure is finished executing.

```
verb <procedure name>
        <type name> reference <parameter name> ;
        …
is
        <declarations>
        <statements>
end;
```

Figure 10.20: Reference Parameter Declaration Syntax

ASSIGNMENT OF REFERENCE PARAMETERS

Reference parameters are also assigned similarly to mandatory parameters, with the parameter values immediately following the subprogram name. One slight difference between reference parameters and mandatory parameters is that a **variable** must be passed in to the reference parameter instead of an **expression**. For example, let's say we have a procedure named "increment" that takes an integer reference parameter and adds 1 to its value. In this case, the procedure call "increment a;" would be valid assuming that a is an integer variable. The procedure call "increment a + 1;", however, would not be valid because we need the name of a variable to place the returned value and "a + 1" is not a valid name of a variable.

```
<procedure name> <variable names> ;
```

Figure 10.21: Reference Parameter Assignment Syntax

EXAMPLE: "REFERENCE_PARAMS.OMAR";

```
do swap_numbers;

verb swap_integers
        integer reference i;
        integer reference j;
is
        integer k = i;

        i = j;
        j = k;
end;    // swap

verb swap_numbers
is
        integer a = 1, b = 2;

        write "a, b = ", a, ", ", b, ;
        swap_integers  a b;                    // note that after this call, the values of a and b will be changed
        write "a, b = ", a, ", ", b, ;
end;    // swap_numbers
```

Figure 10.22: Reference Parameters Example

SCOPING MODIFIERS

Whenever we have the concept of scoping and local variables, there is always the possibility that we may declare a new variable in the local scope that has the same name as a variable in an enclosing scope and therefore hides this other variable. This phenomenon is sometimes known as "variable shadowing".

Variable shadowing can become a problem particularly when using optional parameters because the parameter values are assigned by name. If we want to assign a parameter the value of a different parameter in a different scope with the same name, then we have a problem.

In order to overcome this difficulty, there are two things that may be done. First, we might choose to rename the local variable to avoid the ambiguity altogether. Second, we may precede the variable name by a scoping modifier in order to bypass the most local scope and refer to the hidden scope. There are two scoping modifiers that may be used for this purpose.

THE "GLOBAL" SCOPING MODIFIER

The first scoping modifier, the keyword "global", is used whenever we want to bypass the local scope and refer to a variable in the "global" or outermost scope.

EXAMPLE: "GLOBAL_SCOPING.OMAR"

```
do global_scoping;

integer i = 5;                          // Global i

verb global_scoping
is
        integer i = 10;                 // Local i "shadows" global i

        write "i = ", i, ;              // Will write the value of local i (10)
        write "i = ", global i, ;       // Will write the value of global i (5)
end;    // global_scoping
```

Figure 10.23: Global Scoping Modifier Example

THE "STATIC" SCOPING MODIFIER

The second scoping modifier is used only in procedure or function calls when we are assigning optional or optional return parameters. In this case, the need is to bypass the scope of the procedure being called (the dynamic scope) and refer instead to the context of the place in the program where the procedure call is made (the static scope). For this, we use the scoping modifier, "static".

EXAMPLE: "STATIC_SCOPING.OMAR"

```
do static_scoping;

verb write_integer with
        integer i = 0;
is
        write "i = ", i, ;
end;    // write_integer

verb static_scoping  is
        integer i = 1;                  // Local i

        write_integer with              // Entering scope of write_integer
                i = static i;           // Bypass scope of write_integer to get at local i (value = 1)
        end;                            // Leaving scope of write_integer
end;    // static_scoping
```

Figure 10.24: Static Scoping Modifier Example

LESSON 10:
SIMPLE GRAPHICS PROGRAMMING IN OMAR

LESSON OBJECTIVES:

- Understand how to define simple 3D graphics in OMAR

LESSON CONTENTS:

- Include files for 3d graphics
- New verb types – shape, picture, and anim
- Changing the view
- Using 3D coordinates
- Changing shape parameters
- Defining and using shapes
- Adding simple interactivity

COPYRIGHT©2006

INCLUDE FILES FOR 3D GRAPHICS

An include statement is used to import OMAR code from other OMAR files. Producing 3D graphics requires the inclusion of a set of graphical resources that provide the foundation for 3D graphics. All of the essential graphics resources can be included simply by including the single file "3d.ores", which is located in the directory "Hypercosm Studio/Includes/". All OMAR script files that perform 3D graphics will begin with the statement "include system/3d.ores" immediately following the header statement. Other include statements may be added to include additional resources, but almost all 3D OMAR programs will begin with this statement.

EXAMPLE:

```
do example;                    // Program header

include "system/3d.ores";      // System includes needed for 3D
...
```

Figure 11.1: Include Directive Required for 3D

File name	Purpose
3d.ores	This is the "root" level include for all 3d graphics and contains additional include statements to include definitions for most commonly used 3D features.
native_shapes.ores	This file contains definitions of the native shapes that are built-in to the Hypercosm system such as the mesh, block, sphere, cone, etc.
native_rendering.ores	This file contains definitions of native rendering attributes such as rendering mode, facets, material, and color.
native_lights.ores	This file contains definitions of the different types of native lights such as distant lights, point lights, and spot lights.
native_viewing.ores	This file contains definitions of viewing parameters such as eye point, lookat point, and field of view.

Figure 11.2: Some Standard Include Files Used for 3D

GRAPHICS EXTENSIONS TO OMAR: SHAPES, PICTURES, AND ANIMS

To make the graphics programming easier to understand, a variety of graphical subprogram types have been added as extensions to the language. They are syntactically identical to procedures but have been given different names to reflect the different jobs which they do. The procedural extensions are shapes, pictures, and anims. These subprograms are like procedures because they describe actions to take place but do not return any values. These actions can be viewed as "create this shape", "draw this picture" and "animate this sequence of pictures".

Procedure Type	Purpose
shape	A shape is a procedure that describes how to create a new geometric shape
picture	A picture is like a shape, but it describes a scene that is rendered when the picture is called.
anim	An anim is a procedure that repeatedly calls a picture that changes in order to give the appearance of motion.

Figure 11.3: New Graphical Procedure Types

A SIMPLE EXAMPLE

At a bare minimum, an OMAR graphics file requires a picture. The picture is what causes a window to open and a scene to be rendered. Running a picture is the only way to display something on the screen using the Hypercosm system. Even if the header statement runs an anim, the anim must in turn run a picture in order to produce an image. Inside a picture are shape statements that indicate what objects are to appear in the image. Hypercosm will only display a shape if it is called inside a picture declaration.

```
picture <name> is
        <shapes>
end;
```

Figure 11.4: A Basic Picture Declaration

EXAMPLE: "SIMPLE.OMAR"

```
do simple;

include "system/3d.ores";
include "system/lighting.ores";

picture simple
is
        block;

        // lighting used in the scene
        //
        default_lights;
end;
```

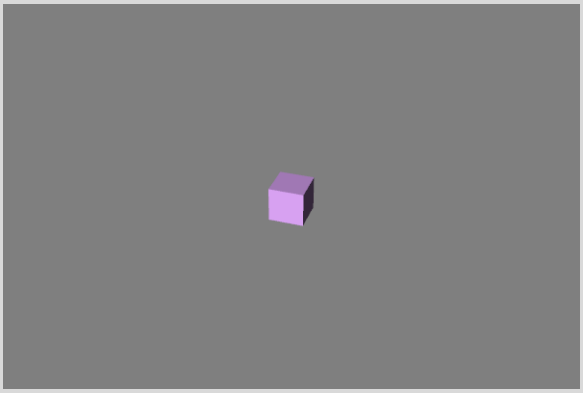


Figure 11.5: A Simple Graphics Program

CHANGING THE VIEW

While Hypercosm graphics do appear on a 2D computer screen, the Hypercosm world is truly three-dimensional. The size of objects on the screen is determined by a combination of their intrinsic size and the properties of the camera that is used to view them. Objects appear large on the screen if the camera is placed relatively close to them, and objects appear to be very small if the camera is relatively far away. Any object that is outside of the camera's field of view won't appear on the screen at all. In the previous example, the view of the scene that was used was defined by the default parameters that are defined by the Hypercosm "include" files. In most cases, we want to set up a view that's more appropriate for our model. This easiest way to do this is simply to set the system variables, eye, lookat, and field_of_view. These variables are defined in the file "native_viewing.ores", which is located in the directory "Hypercosm Studio/Includes/Viewing/".

```
vector eye = <10 –30 20>;
vector lookat = <0 0 0>;
scalar field_of_view = 60;
```

Figure 11.6: Native Viewing Variables

USING 3D COORDINATES

In 3D graphics systems, the locations of objects are defined using 3D coordinates. 3D coordinates work by specifying a location relative to another point. The point that we define objects relative to in our 3D world is called the "origin". The directions that we use to define coordinates are the X axis, Y axis, and Z axis. If we are located at the origin looking towards the Y axis, then the X axis can be thought of as extending from left to right, the Y axis from back to front and the Z axis from down to up.

A NOTE ABOUT UNITS

You might be wondering exactly what size the units are in the 3D coordinate system. If you move an object one unit away, how far will that be on-screen? If you want to place an object ¾ of the way across the window, how many units should you move it? This is frequently a source of confusion or those starting out in 3D graphics. Units in the Hypercosm word are completely relative. How they appear on screen depends upon camera placement and perspective. There is no way to distinguish between a scene depicting a small object viewed from up close and a scene depicting an identical but larger object viewed from correspondingly far away. You can imagine the standard unit of length used to be any type of unit you choose – a millimeter, a yard, or a light year – just so long as you use them consistently.

EXAMPLE: "CHANGING_THE_VIEW.OMAR"

```
do changing_the_view;

include "system/3d.ores";
include "system/lighting.ores";

picture changing_the_view with
        eye = <4 –4 4>;
        lookat = <0 0 0>;
        field_of_view = 60;
is
        block;
        default_lights;
end;
```

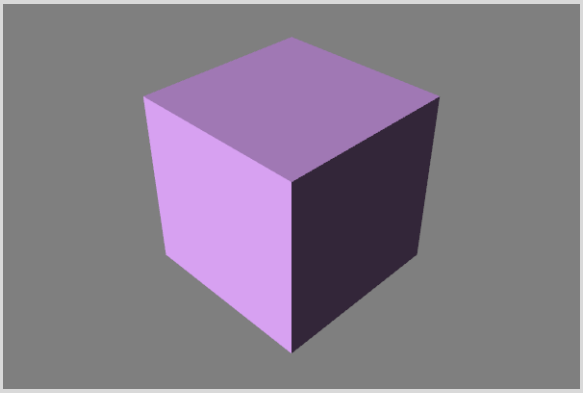
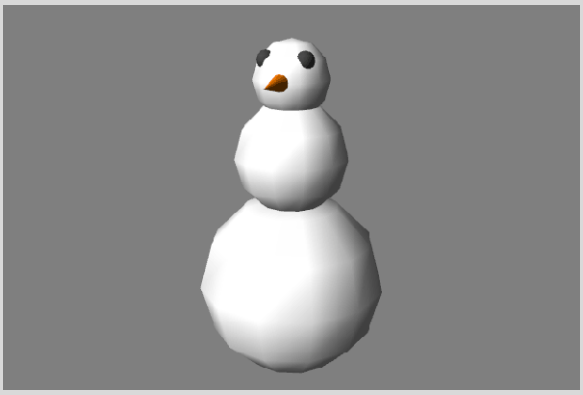


Figure 11.7: Changing the View

CHANGING SHAPE PARAMETERS

In the previous example, we simply listed the name of a shape inside of the picture. In most cases, you actually modify how the shape is created by setting parameters when you create it. You will need to look at the definition of the shape to know if it has any parameters and what their definitions are before you can try changing them.

EXAMPLE: "SNOW_SCENE.OMAR"



```
do snow_scene;

include "system/3d.ores";
include "system/lighting.ores";

picture snow_scene with
        eye = <2 -6 4>;
        lookat = <0 0 2>;
is
        sphere with
                center =<0 0 1>;  color = white;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6; color = white;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;  color = white;
        end;

        // nose
        cone with
                end1 = <0 -.3 3.2>;  end2 = <0 -.8 3.2>;  radius1 = .1; radius2 = 0; color = orange;
        end;

        // eyes
        sphere with
                center = <-.2 -.2 3.4>; radius = .1;  color = charcoal;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1;  color = charcoal;
        end;

        default_lights;
end;    // snow_scene
```

Figure 11.8: A Simple Graphics Using Shape Parameters for Modeling

DEFINING AND USING SHAPES

Although we could technically define our 3D scenes just by declaring a bunch of shapes inside of a picture, for even moderately complex scenes this would get unwieldy.

HIERARCHICAL MODELING

A better way of organizing 3D scenes is to group objects together into more complex shapes. This is done in Hypercosm using "shape" methods, which are similar to picture methods except that they don't draw their component shapes until the shape is instantiated in a picture. The process of declaring new shape definitions and then reusing them later is the basis of "hierarchical modeling".

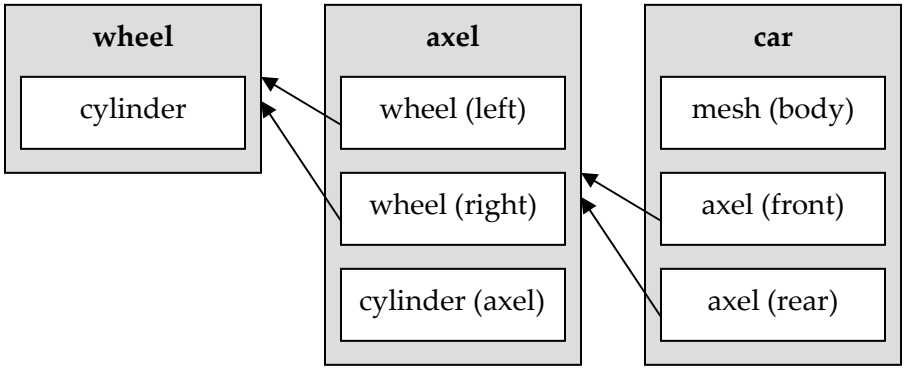
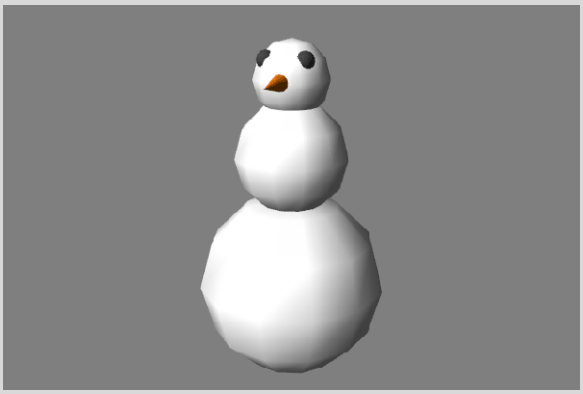


Figure 11.9: Principles of Hierarchical Modeling

```
// declaration of new shape
//
shape <shape name>
        // optional shape parameters would go here – declared as in a procedure
is
        // definition of shape in terms of simpler shapes
end;

<shape name>;           // instantiation of new shape
```

Figure 11.10: How to Declare and then Instantiate a Shape

EXAMPLE: "SNOWMAN.OMAR"



```
do snowman_scene;

include "system/3d.ores";
include "system/lighting.ores";

shape snowman is
        sphere with
                center =<0 0 1>;
                color = white;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6;
                color = white;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;
                color = white;
        end;

        // nose
        cone with
                end1 = <0 -.3 3.2>;  end2 = <0 -.8 3.2>;  radius1 = .1; radius2 = 0;
                color = orange;
        end;

        // eyes
        sphere with
                center = <-.2 -.2 3.4>; radius = .1;
                color = charcoal;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1;
                color = charcoal;
        end;
end;    // snowman

picture snowman_scene with
        eye = <2 -6 4>;
        lookat = <0 0 2>;
is
        snowman;
        default_lights;
end;    // snowman_scene
```

Figure 11.11: Example of Declaration and Instantiation of a Shape

ADDING SIMPLE INTERACTIVITY

In the previous examples, we were able to create and view 3D models, but we didn't have a way of moving them around and examining them interactively. This is easily accomplished using a utility animation called "mouse_controlled_shape". The definition of the mouse controlled shape and related animations is stored in a file called "anims.ores" that you'll need to include if you want to use the mouse_controlled_shape. The file "anims.ores" is located in the directory "Hypercosm Studio/Includes/Animation/". When you use a "mouse_controlled" anim, the lights are automatically added for you, so you don't need to specify them.

Anim name:	**Description:**
mouse_controlled_shape	Allows shapes to be controlled interactively
mouse_controlled_animated_shape	Allows animated shapes to be controlled interactively
mouse_controlled_actor	Allows actor objects to be controlled interactively
mouse_controlled_picture	Allows pictures to be controlled interactively

Figure 11.12: Mouse Controlled Anims

EXAMPLE: "INTERACTIVE_SNOWMAN.OMAR"

```
do interactive_snowman;

include "system/3d.ores";
include "system/anims.ores";
include "snowman.omar";

anim interactive_snowman with
        eye = <2 -6 4>;
        lookat = <0 0 2>;
is
        mouse_controlled_shape snowman with
                auto_camera is on;
        end;
end;
```

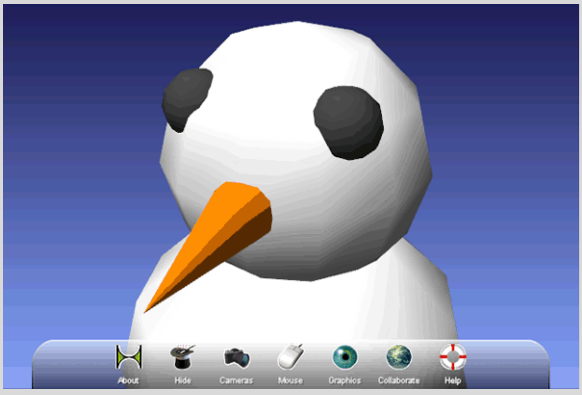


Figure 11.13: Using a Mouse Controlled Shape

LESSON 11:
3D MODELING IN OMAR

LESSON OBJECTIVES:

- Understand how to define simple 3D graphics in OMAR

LESSON CONTENTS:

- What is modeling?
- Hypercosm shape primitives
- Hypercosm emulated 3ds Max™ shapes
- Transformations
  - Relative transformations
  - Absolute transformations
- Colors
- Materials
- Textures

COPYRIGHT©2006

WHAT IS MODELING?

Modeling is the process that we use to represent 3D shapes, surfaces, and materials using computer software. Modeling can be a complex task simply because the real world is complex and contains so many different types of shapes.

HYPERCOSM SHAPE PRIMITIVES

In Hypercosm as in other graphics systems, every shape that is displayed must be described in terms of simpler entities that the software already knows how to deal with. These simple 3D shapes are called "primitives". The simple 3D shapes that Hypercosm understands are all declared in the file "native_shapes.ores", which is located in the directory "Hypercosm Studio/Includes/Modeling/Shapes/".

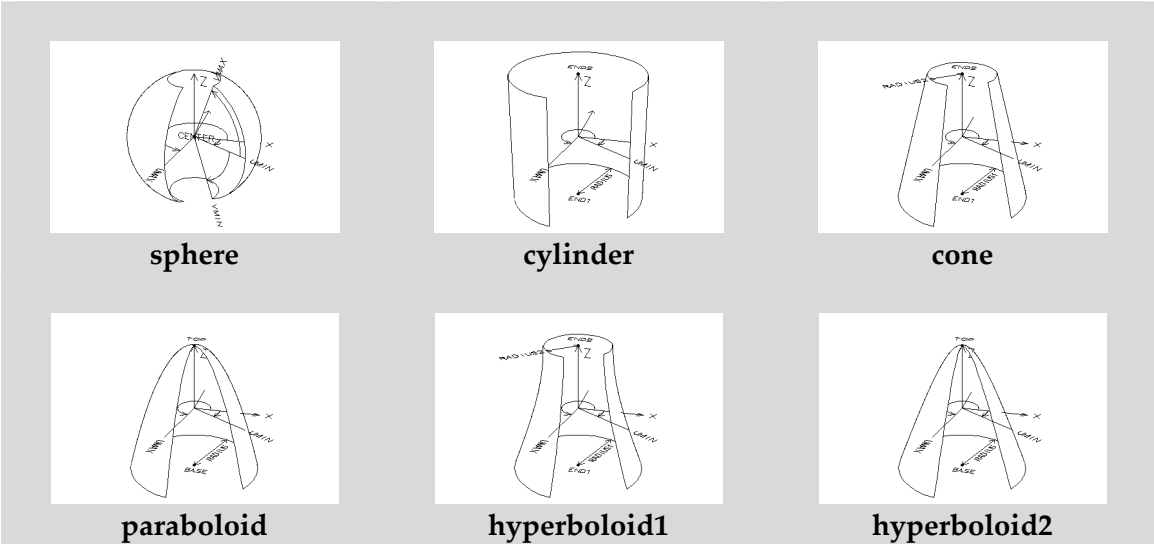


Figure 12.1: Hypercosm "Quadric" Primitives

NOTES: EXAMPLES OF PRIMITIVES

In the Hypercosm "Example Projects/Instructional Examples" directory, there are sample projects that show how to use each of the Hypercosm primitives.

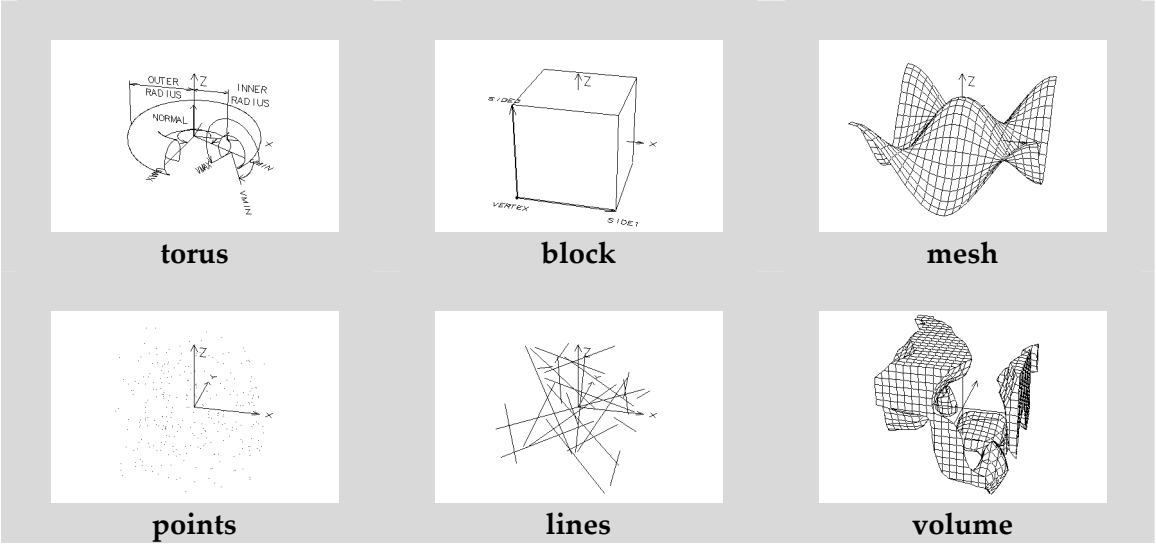


Figure 12.2: Hypercosm "Non-Planar" Primitives

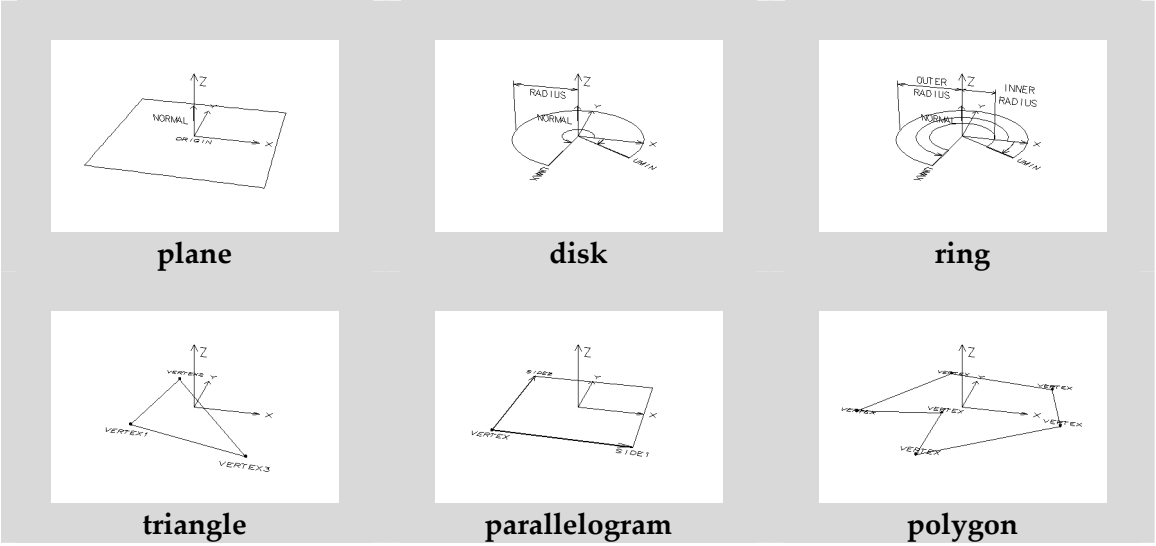


Figure 12.3: Hypercosm "Planar" Primitives

HYPERCOSM 3DS MAX™ SHAPES

When a 3D file has been translated from another 3D modeling package using Hypercosm Teleporter, it is converted to script code. To achieve the highest level of fidelity in the translation process, Hypercosm Teleporter preserves the set of primitives that are part of the original 3D modeling package. To do this, it creates a new "emulated" set of primitives using the script code that preserve the properties of the original 3D modeling package. These emulated primitives are written entirely in the OMAR script language.

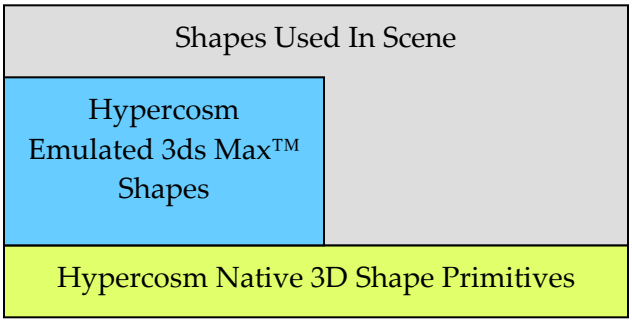


Figure 12.4: Primitives Used In "Teleported" Script Files

NOTES: MORE ON 3DS MAX™ PRIMITIVES

To view examples that demonstrate each of the 3ds Max™ primitives translated into Hypercosm script, open up the sample project contained in "Example Projects/Instructional Examples/Hypercosm Teleporter/3ds/3ds Shapes".

You can see the definitions and implementations of Hypercosm's emulated 3ds Max™ primitives in the ".ORES" files that are contained in the folder "Hypercosm Studio/Includes/Hypercosm Teleporter/3ds/".

TRANSFORMATIONS

When we create an instance of an object, we often wish it to have a different position or size than when the object was defined. This is accomplished by what are known as transformations. Transformations include actions such as moving, rotating, magnifying, stretching, and skewing.

RELATIVE TRANSFORMATIONS

Relative transformations specify the size or location of an object relative to the way that it was originally defined in its declaration. These are the most commonly used type of transformations. For instance, if the object in the declaration is 2 units high, we can make it 4 units high by magnifying by 2. To explicitly make the object 4 units high regardless of how it was defined, we would need to use an absolute transformation, which is covered later. The declarations of the relative transformations are found in the file "transformations.ores", which is located in the directory "Hypercosm Studio/Includes/Modeling/Transformations/". The more commonly used ones are listed below:

Relative Transformation Name	Example
direct	direct from <0 0 1> to axis;
magnify	magnify by 5;
move	move to <0 0 10>;
orient	orient from <0 0 1> to axis;
rotate	rotate by 30 around <0 0 1>;
scale	scale by 5 along <0 0 1>;

Figure 12.5: Examples of Commonly Used Relative Transformations

APPLYING RELATIVE TRANSFORMATIONS

Relative transformations are applied by placing transformation calls inside of a "with" block that is part of the shape instance. This is the same block where you would specify optional parameters if the shape had any.

```
<shape name> with
        <relative transformations>
end;
```

Figure 12.6: How to Apply Relative Transformations

EXAMPLE: "SNOW_FAMILY.OMAR"


```
do snow_family_scene;

include "system/3d.ores";
include "system/anims.ores";
include "snowman.omar";

shape snow_family  is
        // Dad
        snowman with
                move to <-.7 0 0>;
        end;

        // Mom
        snowman with
                magnify by .8; move to <.7 0 0>;
        end;

        // Kids
        snowman with
                magnify by .5; move to <-.5 -1 0>;
        end;
        snowman with
                magnify by .5; rotate by 30 around <0 0 1>; move to <.5 -1 0>;
        end;
end;    // snow_family

anim snow_family_scene is
        mouse_controlled_shape snow_family with
                auto_camera is on;
        end;
end;    // snow_family_scene
```

Figure 12.7: Example of Relative Transformations

ORDER OF TRANSFORMATIONS

Once we define an object, we can create several instances of the object, all with different locations, orientations, and dimensions. More than one transformation may be applied

to a single instance, for instance, it may be rotated, then stretched, then moved. Note that the order of the transformations is important. An object that has been stretched, then rotated looks different from an object that has been rotated, then stretched.

ABSOLUTE TRANSFORMATIONS

The main difficulty with the relative transformations is that sometimes we may want to manipulate an object without having to look at the dimensions that it was defined with. For example, say that we include a model of a car into our picture and we want to set it onto a road. If our road is 1 unit wide and we want the car to fit nicely onto the road, then if we were to use the relative transformations, we would need to examine the model of the car to see how big it is and then magnify it by the (road width / car width). If we use the absolute transformations, however, we can just specify to make the car a certain width no matter how big or small the car was originally defined to be. The declarations of the absolute transformations are found in the file "abs_trans.ores", which is located in the directory "Hypercosm Studio/Includes/Transformations/". The more commonly used ones are listed below:

Absolute Transformation Name	**Example**
dimensions	dimensions of <10 20 30>;
size	size of 10 along z_axis;
limit	limit z_min to 0;

Figure 12.8: Examples of Commonly Used Absolute Transformations

APPLYING ABSOLUTE TRANSFORMATIONS

Absolute transformations are specified a little differently than relative transformations. Since the absolute transformations need to know the dimensions of the object in order to work, they must take place after the object has been created. The relative transformations are listed in a block of statements beginning at the keyword "with" which is executed before the object is actually created by the computer. The absolute transformations must occur after the object has been built, so they are listed in a later block of statements beginning with the keywords "return with".

```
<shape name> return with
        <absolute transformations>
end;
```

Figure 12.9: How to Apply Absolute Transformations

EXAMPLE: "SNOW_FAMILY2.OMAR"


```
do snow_family_scene;

include "system/3d.ores";
include "system/anims.ores";
include "system/abs_trans.ores";
include "snowman.omar";

shape snow_family with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        // Dad – 6 feet tall
        snowman return with
                size of 6 along z_axis; move to <-1 0 0>;
        end;

        // Mom – 5 feet tall
        snowman return with
                size of 5 along z_axis; move to <1 0 0>;
        end;

        // Kids – 3 feet tall
        snowman return with
                size of 3 along z_axis; move to <-.5 -2 0>;
        end;
        snowman return with
                size of 3 along z_axis; rotate by 30 around <0 0 1>; move to <.5 -2 0>;
        end;
end;    // snow_family

anim snow_family_scene is
        mouse_controlled_shape snow_family with
                auto_camera is on;
        end;
end;    // snow_family_scene
```

Figure 12.10: Example of Absolute Transformations

COLOR

Because of certain characteristics of the human eye, the perception of almost all imaginable colors can be created by mixing red, green, and blue in different proportions. While computer monitors may appear to display a full spectrum of colors, they actually use only three: red, green, and blue.

REPRESENTING COLORS

In the Hypercosm system, all colors are specified using red-green-blue (RGB) values, which indicate exact amounts of red, green, and blue to mix in ranges from 0 to 1.

ASSIGNING COLORS TO SHAPES

To assign a color to a shape, you must set the global color variable, "color", when you create an instance of that shape. The global color variable is declared in the file "native_rendering.ores", which is located in the directory "Hypercosm Studio/Includes/Rendering/". This variable is of the type, "color". The color type works the same way as the vector type because colors, like vectors, are specified by three scalar numbers.

```
color type color;
```

Figure 12.11: The Global Color Variable

```
sphere with
        color = <1 0 0>;            // sets the color to red (R = 1, G = 0, B = 0)
end;
```

Figure 12.12: Assigning a Color to a Shape

HIERARCHY OF COLORS

The system for assigning colors to shapes works in a hierarchical manner. When you assign a color to an instance of a shape, that color only applies to the "unpainted" parts of the shape that haven't already had a color assigned. For parts of the shape that have already been assigned a color, the previously assigned color takes precedence. The color that gets assigned to a primitive is the color that is closest to the shape in the modeling hierarchy.

PREDEFINED COLORS

For your convenience, Hypercosm provides a number of predefined colors so you can refer to colors by name instead of by RGB values. The following chart shows the names of available colors and the RGB values that each color represents. These colors are defined in the file "common_colors.ores", which is located in the directory "Hypercosm Studio/Includes/Drawing/Colors/".

Name	RGB Value
black	0 0 0
white	1 1 1
grey	.5 .5 .5
red	1 0 0
green	0 1 0
blue	0 0 1
cyan	0 1 1
magenta	1 0 1
yellow	1 1 0
orange	1 .5 0
brown	.35 .2 0
gold	.9 .8 .3
maize	.8 .7 0
brick	.5 .15 0
rust	.7 .3 .1
charcoal	.2 .2 .2
raspberry	1 0 .5
pink	1 .6 .7
flesh	.9 .7 .6
beige	1 .9 .85
lime_green	.5 .8 0
olive	.4 .5 0
evergreen	0 .4 .25
teal	0 .75 .6
aqua	0 .75 .75
turquoise	0 .7 .9
sky_blue	.6 .75 1
azure	.35 .3 .75
lavender	.8 .6 .9
purple	.6 .15 .75
violet	.5 0 .9
eggplant	.3 0 .2

Figure 12.13: Predefined Colors (from "common_colors.ores)

COLOR MODULATION FUNCTIONS

In addition to the predefined colors, for convenience Hypercosm provides two utility functions, "light" and "dark" to make colors lighter or darker by mixing them with white or black.

```
color type question light
        color type color;
end;

color type question dark
        color type color;
end;
```

Figure 12.14: Color Modulation Functions

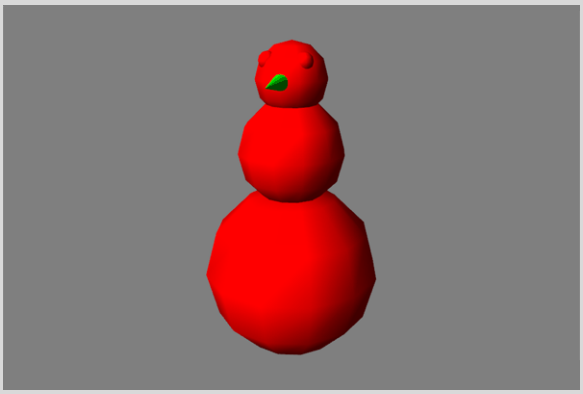
```
color = light yellow;
color = dark green;
color = light light yellow;
color = dark dark green;
```

Figure 12.15: Use of Color Modulation Functions

DEFAULT COLORS

If you don't assign any colors to your objects, you may be surprised to find that they still are displayed in various colors. This is because each primitive shape has a default color assigned to it. If you don't assign a shape any colors at all, then each primitive will take on the default color associated with that primitive. The default shape colors are listed in the file "native_shape_colors.ores", which is located in the directory "Hypercosm Studio/Includes/Modeling/Shapes/".

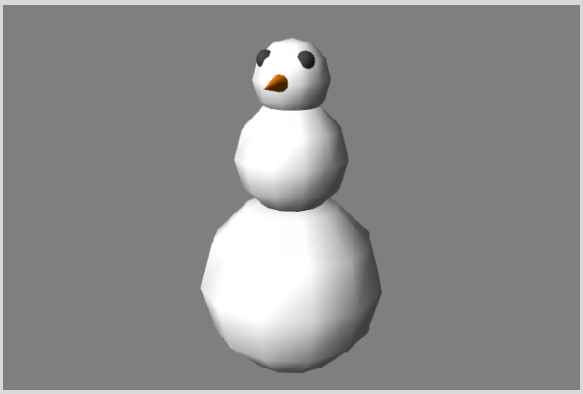
EXAMPLE: "DEFAULT_COLORS.OMAR"



```
do example;

include "system/3d.ores";
include "system/lighting.ores";

shape snowman is
        sphere with
                center =<0 0 1>;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;
        end;

        // nose
        cone with
                end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>;
                radius1 = .1; radius2 = 0;
        end;

        // eyes
        sphere with
                center = <-.2 -.2 3.4>; radius = .1;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1;
        end;
end;    // snowman

picture example with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        snowman;
        default_lights;
end;    // example
```

Figure 12.16: Example of Default Colors

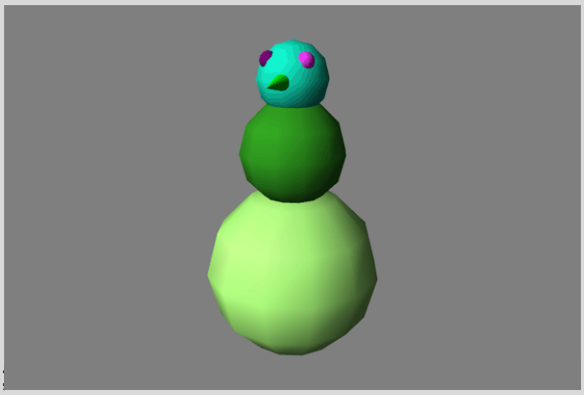
EXAMPLE: "COLORED_SNOWMAN.OMAR"



```
do example;

include "system/3d.ores";
include "system/lighting.ores";

shape snowman is
        color = white;

        sphere with
                center =<0 0 1>;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;
        end;

        // nose
        cone with
                end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>;  radius1 = .1; radius2 = 0;
                color = orange;
        end;

        // eyes
        sphere with
                center = <-.2 -.2 3.4>; radius = .1;
                color = charcoal;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1;
                color = charcoal;
        end;
end;    // snowman

picture example with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        snowman;
        default_lights;
end;    // example
```

Figure 12.17: Example of Application of Colors

EXAMPLE: "RANDOM_COLORS.ORES"



```
do example;

include "system/3d.ores";
include "system/lighting.ores";
include "system/random.ores";

shape snowman
is
        sphere with
                center =<0 0 1>;
                color = vrandom from black to white;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6;
                color = vrandom from black to white;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;
                color = vrandom from black to white;
        end;

        // nose
        cone with
                end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>;  radius1 = .1; radius2 = 0;
                color = vrandom from black to white;
        end;

        // eyes
        sphere with
                center = <-.2 -.2 3.4>; radius = .1;
                color = vrandom from black to white;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1;
                color = vrandom from black to white;
        end;
end;    // snowman

picture example with
        eye = <3 -8 4>; lookat = <0 0 2>;
is
        snowman; default_lights;
end;    // example
```

Figure 12.18: Example of the Use of Random Colors

MATERIALS

How color varies across a surface in response to light depends upon a variety of factors other than color. To control the appearance of objects in a more precise matter, we use materials. Materials allow you to more closely approximate the appearance of objects in the real world. Materials are defined in the file "native_materials.ores", which is located in the directory "Hypercosm Studio/Includes/Rendering/Materials/";

ASSIGNING MATERIALS TO SHAPES

To assign a material to a shape, you must set the global material variable, "material", when you create an instance of that shape. The global material variable is declared in the file "native_rendering.ores", which is located in the directory "Hypercosm Studio/Includes/Rendering/". This variable is of the type, "material". The material type defines a number of attributes that define particular aspects of the material that contribute to the material's apparent color.

```
material type material;
```

Figure 12.19: The Global Material Variable

```
sphere with
        material is wood;          // sets the material of the sphere
end;
```

Figure 12.20: Assigning a Material to a Shape

COMMON MATERIALS

For convenience, a set of common materials is predefined for you in the file "common_materials.ores" which is located in the directory "Hypercosm Studio/Includes/Rendering/Materials/". This set of materials includes a variety of metals as well as materials like wood, rubber, or stone.

PREDEFINED MATERIAL FUNCTIONS

For convenience, a small set of material generation functions is included in order to make it easier to create and assign materials. The three main predefined materials are: chalk, plastic, and metal.

CHALK

A chalk material is a material that has no highlights and reflects light in a diffuse way. The main parameter to the "chalk" function is a color, which becomes the main "diffuse" color of the material.

```
sphere with
        material is chalk colored red;              // sets the material of the sphere
end;
```

Figure 12.21: Assigning a Chalk Material to a Shape

PLASTIC

A plastic material is distinguished by having a white shiny highlight. Plastic materials take a color as a parameter that becomes the main "diffuse" color to which the white highlight is added.

```
sphere with
        material is plastic colored red;            // sets the material of the sphere
end;
```

Figure 12.22: Assigning a Plastic Material to a Shape

METAL

A metal material is distinguished by a colored, semi shiny highlight. For example, a copper sphere will have an orange colored highlight whereas a gold sphere will have a yellowish highlight.

```
sphere with
        material is metal colored red;              // sets the material of the sphere
end;
```

Figure 12.23: Assigning a Metal Material to a Shape

CONSTANT COLOR

A constant_color material is distinguished by a lack of shading. The entire surface of an object with a constant color material will appear the exact same color regardless of the direction of the light or other lighting conditions.

```
sphere with
        material is constant_color red;                 // sets the material of the sphere
end;
```

Figure 12.24: Assigning a Constant Color Material to a Shape

THE MATERIAL HIERARCHY

Materials are applied in a hierarchy in a similar way to colors. One difference is that materials always override colors. If you have a model that has colors assigned to various parts and you apply a material to it, the material will cover all of the colors.

"OVERRIDE" MATERIALS

The other difference is that you can override the hierarchical way that materials are applied. Each material has a parameter called "override". If you set the override parameter to true, then that material will cover all other materials in the model hierarchy that don't have the "override" set. This is particularly useful when you have a model that has had a series of materials assigned and you want to paint over the entire model with a single material. For example, this feature is particularly useful when you'd like to "highlight" a model or assembly.

```
assembly with
        material is (chalk colored red with override is true);    // sets the material of the assembly
end;
```

Figure 12.25: Assigning an Override Material to a Shape

TEXTURES

The easiest way to add interesting, complex features to the surfaces of your shapes is to use textures. When a shape is textured, an image from an image file is mapped onto the

surface of the 3D object. Using textures, you can map an image of the earth onto a sphere, or create realistic materials such as wood, brushed metal or stone. You can even "fake" a large amount of geometry when the user won't be close enough to see the difference between the texture and the actual geometry. For example, you could map an image of a crowd across the seats of a stadium or map an image of bumps or ridges onto a knob or dial.

ASSIGNING TEXTURED MATERIALS TO SHAPES

The easiest way to assign a texture is to create a material type that incorporates an image file as a texture. Each of the predefined material generation functions (chalk, plastic, metal, constant_color) have a keyword parameter that lets to easily add a texture. The easiest way to do this is with the "poster" keyword parameter. When you create a material using this keyword, you must provide a string parameter for the name of the image file to use. The material will do the rest.

```
sphere with
        material is plastic poster "earth.jpg";          // set the shape's material and the material's texture
end;
```

Figure 12.26: Assigning a Textured Material to a Shape

TEXTURE IMAGE FILE TYPES

The Hypercosm system can create textures from GIF, JPEG, or PNG files. The appropriate file format to use depends upon the type of image that you want to use. JPEG files are best suited for continuous tone images such as digital photographs. GIF files are best for images with just a few colors such as decals or diagrams. PNG files have the benefit that they can have a transparency channel that allows varying degrees of transparency.

EXAMPLE: "TEXTURED_SNOWMAN.OMAR"


```
do textured_scene;

include "system/3d.ores";
include "system/lighting.ores";
include "system/common_materials.ores";
include "system/anims.ores";

shape textured_snowman is
        sphere with
                center = <0 0 1>;
                material is golden;
        end;
        sphere with
                center = <0 0 2.4>; radius = .6;
                material is wood;
        end;
        sphere with
                center = <0 0 3.2>; radius = .4;
                material is plastic poster "flag.gif";
        end;

        // nose
        //
        cone with
                end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>; radius1 = .1; radius2 = 0;
                material is plastic colored orange;
        end;

        // eyes
        //
        sphere with
                center = <-.2 -.2 3.4>; radius = .1; material is metal;
        end;
        sphere with
                center = <.2 -.2 3.4>; radius = .1; material is metal;
        end;
end;    // textured_snowman

anim textured_scene with
        eye = <2 -6 4>; lookat = <0 0 2>;
is
        mouse_controlled_shape textured_snowman;
end;    // textured_scene
```

Figure 12.27: Assigning a Textured Materials to Shapes

LESSON 12:

SIMPLE ANIMATION

LESSON OBJECTIVES:

- Understand the principles of animation
- Be able to create simple animations in OMAR

LESSON CONTENTS:

- The principle of animation
- Acceptable frame rates
- Anims
- Controlling animation speed
- Animated shapes

COPYRIGHT©2006

THE PRINCIPLE OF ANIMATION

Animation is, literally, the process of bringing something to life. To bring imagery to life, you must make it dynamic, that is, to give it the characteristic property of life, the ability to change over time. You can create the illusion of an animated image by presenting a sequence of images in quick succession. If each image is only slightly different from the previous one and the time between each image is short, then the brain fills in the gaps between the images and we perceive a continuous, fluid animation. This is the underlying principle of all forms of animation, including motion pictures, television, video, and Saturday morning cartoons. It is also how Hypercosm animation works.

ACCEPTABLE FRAME RATE

Fluid animation requires a frame rate of around 30 frames per second. Video and television are shown at 30 frames per second. Film projectors traditionally present a slightly lower frame rate of 24 frames per second. Some special projector systems, such as IMAX, present 60 frames per second. For many applications, a much lower frame rate is acceptable, however below about 10 frames per second, the eye can detect the changes between frames and the resulting animation may appear jerky instead of fluid and smooth.

Animation Type	Frame Rate (frames per second)
Video / Television	30
Film	24
Interactive Computer Animation	10 - 30

Figure 13.1: Frame Rates of Different Forms of Animation

ANIMATION IN HYPERCOSM

Hypercosm animations make use of a special type of OMAR procedure known as an anim. Inside an anim is a looping statement that tells the computer to create pictures repeatedly. Each picture differs slightly from the one before it. An anim is just like a verb or procedure except that it is allowed to call pictures. The graphical procedures (anims, pictures, and shapes) may call the non graphical procedures (verbs and questions) but not vice versa. Note that an anim cannot directly call a shape. In order to use a shape, an anim must call a picture that calls a shape.

Subprogram Type	Types of Subprograms that May be Called
verb, question	verb, question
shape	shape, verb, queustion
picture	shape, verb, question
anim	picture, verb, question

Figure 13.2: Subprogram Types

EXAMPLE: "ROTATING_SNOWMAN.ORES"

```
do rotating_snowman;

include "system/3d.ores";
include "snowman.omar";

picture scene with
        scalar rotation = 0;
is
        snowman with
                rotate by rotation around <0 0 1>;
        end;
        distant_light;
end;    // scene

anim rotating_snowman with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        scalar angle = 0;

        while true do
                scene with
                        rotation = angle;
                end;
                angle = itself + 1;
        end;
end;    // rotating_snowman
```

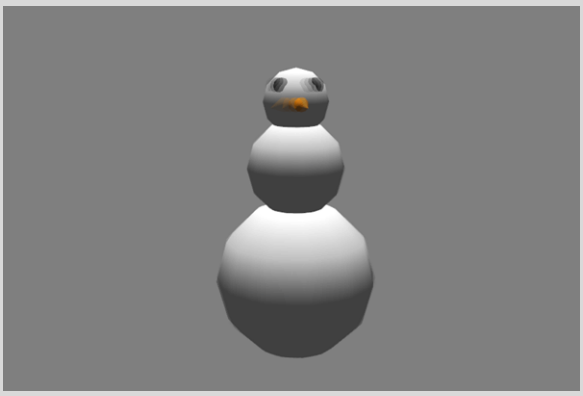


Figure 13.3: A Simple Animation

CONTROLLING ANIMATION SPEED

In the previous example, there is a problem with the way that the animation has been specified. Note that each time a picture is drawn, the rotation increases by 1 degree. This means that the speed of the animation is dependent upon the speed of your computer. Normally, we'd like the animations that we create to be able to run on a range of different types of computers. Also, computers tend to get a little bit faster every year and so animations that are coded in this way will speed up as the computers that they run on get faster. A better way is clearly needed. The answer is to control the animation speed using the system's clock.

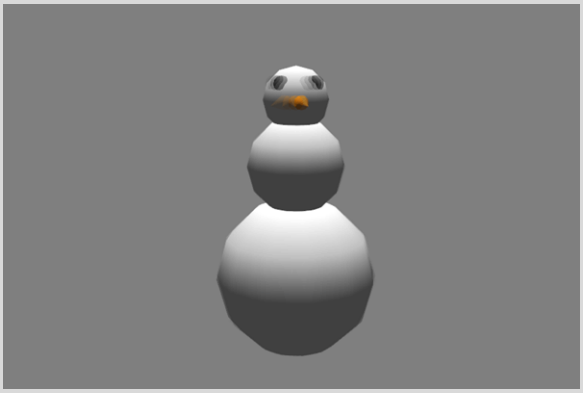
THE TIME FUNCTIONS

Hypercosm provides a set of time functions in order to make it easy to get the current time or the elapsed time from the previous frame. We use these functions to control animation speed. These functions are declared in the file "native_time.ores", which is located in the directory "Hypercosm Studio/Includes/Time/".

Function	Description
scalar question get_seconds;	This method returns the number of seconds since the applet was started.
time type question get_time;	This function returns the number of hours, minutes, and seconds since midnight.
scalar question get_frame_duration;	This function returns the amount of time that has elapsed since the last picture was drawn.

Figure 13.4: The Time Functions

EXAMPLE: "CONSTANT_ROTATION.OMAR"



```
do constant_rotation;

include "system/3d.ores";
include "rotating_snowman.omar";

anim constant_rotation with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        scalar rpm = 10, angle = 0;

        while true do
                scene with
                        rotation = angle;
                end;

                // use get_frame_duration to control the advancing of the rotation angle
                //
                angle = itself + rpm / 60 * get_frame_duration * 360;
        end;
end;    // constant_rotation
```

Figure 13.5: A Time Controlled Animation

ANIMATED SHAPES

Animation is such a commonly performed function that it helps to have a standardized way of doing it. In the previous example, the code for computing the rotation is separate from the place where the rotation is applied. A better approach might be to have all of the rotation code inside of the shape and then the only thing the shape needs to "know" about is the passing of time. To do this, we create an "animated shape". An animated shape is simply a shape with a scalar time parameter. The time parameter is what allows the system to know when the shape is changing and when the drawing needs to be updated.

EXAMPLE: "ANIMATED_SNOWMAN.OMAR"

```
do animated_snowman;

include "system/3d.ores";
include "snowman.omar";

shape rotating_snowman with
        scalar time = 0;
is
        scalar rpm = 10;
        scalar angle = rpm / 60 * time * 360;

        snowman with
                rotate by angle around <0 0 1>;
        end;
end;    // rotating_snowman

picture scene is
        rotating_snowman with
                time = get_seconds;
        end;
        distant_light;
end;    // scene

anim animated_snowman with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        while true do
                scene;
        end;
end;    // animated_snowman
```

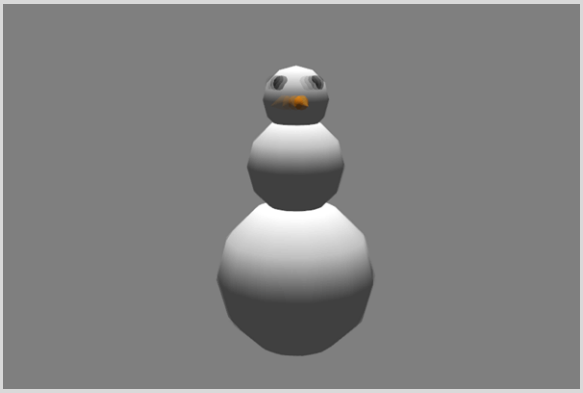


Figure 13.6: An Animated Shape

MOUSE CONTROLLED ANIMATED SHAPES

One other convenient aspect of using animated shapes is the ability to use them with the "mouse_controlled_animated_shape" anim. This is a handy animation utility that makes it easy to view and interact with animated shapes. All you need to do is to give your shape a time parameter and then pass it to this anim. The animation, user interaction, and lighting will all be taken care of for you.

EXAMPLE: "INTERACTIVE_ANIMATED_SNOWMAN.OMAR"

```
do interactive_animated_snowman;

include "system/anims.ores";
include "system/3d.ores";
include "snowman.omar";

shape rotating_snowman with
        scalar time = 0;
is
        scalar rpm = 10;
        scalar angle = rpm / 60 * time * 360;

        snowman with
                rotate by angle around <0 0 1>;
        end;
end;    // rotating_snowman

anim interactive_animated_snowman with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        mouse_controlled_animated_shape rotating_snowman;
end;    // interactive_animated_snowman
```

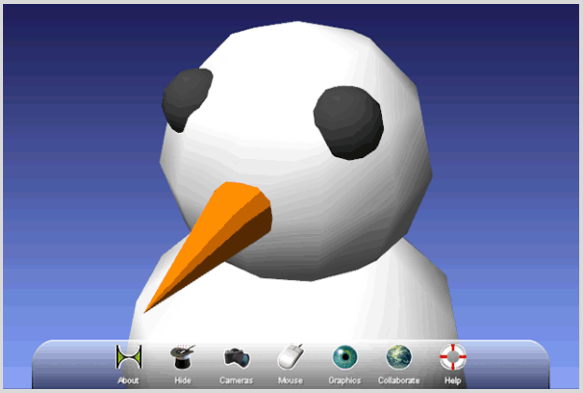


Figure 13.7: A Mouse Controlled Animated Shape

LESSON 13:

INPUT – THE KEYBOARD AND MOUSE

LESSON OBJECTIVES:

- Understand how to use the keyboard input
- Understand how to use the mouse input

LESSON CONTENTS:

- Keyboard functions
- Keyboard click functions
- Keycodes and characters
- Mouse location and button functions
- Mouse click functions

COPYRIGHT©2006

THE KEYBOARD

The keyboard is often a useful way to control animations and simulations. For this purpose, Hypercosm provides a set of keyboard functions that allow you to determine the state of any key on the keyboard or to report key press events.

KEYBOARD QUERYING

The first function, "key_down" is used to query the current status of the keyboard. The "key_down" function is located in the file "native_keyboard.ores", which is located in the directory "Hypercosm Studio/Includes/Devices/Keyboard/". This function takes a keycode as a parameter and then checks whether the corresponding key is pressed. If that key is pressed, "key_down" answers true, otherwise it answers false.

```
boolean question key_down
        integer key;
end;
```

Figure 14.1: The Keyboard Querying Function

KEYCODES

Keycodes are a way of identifying each key on the keyboard. Since all computers have different keyboards, a certain key may have different keycodes on different computers, or may not even exist on certain computers. On a Macintosh, for example, the keycode for the "A" key is 0, whereas on a UNIX machine, it is 97. If you were to use only your operating system's keycodes to refer to keys, then your programs would not be portable between different makes of computers. As a way around this problem, Hypercosm provides its own keycodes to attempt to make programs more portable. The Hypercosm system handles the conversion between the Hypercosm keycode and the system-dependent keycode internally. The Hypercosm keycodes are all defined in the file "keycode.ores", which is locataed in the directory "Hypercosm Studio/Includes/Devices/Keyboard/".

CONVERTING BETWEEN KEYCODES AND CHARACTERS

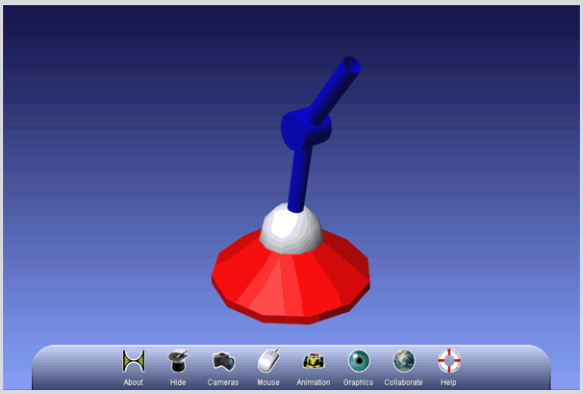
For the common letter and symbol keys on the keyboard, it's easiest to refer to the keys by character instead of by keycode. You can use the functions "key_to_char" and "char_to_key" to convert between characters and the Hypercosm keycodes. These functions are located in the file "keycode.ores", which is located in the directory "Hypercosm Studio/Includes/Devices/Keyboard/".

```
char question key_to_char
        integer key;
with
        boolean shift is false;
end;

integer question char_to_key
        char c;
end;
```

Figure 14.2: Keycode to Character Conversion Functions

Note that there are many special keys on a keyboard, such as F1 or Page Up, that do not correspond to any printable character and cannot therefore be converted to chars. For this reason, the conversion functions only work for the letter, number, and symbol keys, and for certain special keys such as the Space, Tab, and Enter keys. If you need to know the keycode of a special key such as F1, then you can look in the resource file, "keycode.ores", where you can find the conversion function definitions together with documented conversion tables.

EXAMPLE: "KEY_CONTROL_ROBOT.OMAR"




```
do key_controlled_robot;

include "system/3d.ores";
include "system/anims.ores";
include "system/shapes.ores";

shape robot with
        scalar base_rotation = 0;
        scalar arm1_angle = 0, arm2_angle = 0;
is
        shape base is
                cylinder with
                        end1 = <0 0 0>; end2 = <0 0 .1>; radius = 1;
                end;
                cone with
                        end1 = <0 0 .1>; radius1 = 1; end2 = <0 0 .5>; radius2 = .4;
                end;
                sphere with
                        center = <0 0 .5>; radius = .4; material is plastic colored white;
                end;
        end;    // base
        shape arm with
                scalar angle = 0;
        is
                cylinder with
                        end1 = <0 0 0>; end2 = <0 0 1>; radius = .1;
                end;
                rod with
                        end1 = <-.2 0 1>; end2 = <.2 0 1>; radius = .2;
                end;
                cylinder with
                        end1 = <0 0 0>; end2 = <0 0 1>; radius = .1;
                        rotate by angle around <1 0 0>; move to <0 0 1>;
                end;
        end;    // arm

        base with
                material is plastic colored red;
        end;
        arm with
                angle = arm2_angle; move to <0 0 .4>;
                rotate by arm1_angle around <1 0 0>;rotate by base_rotation around <0 0 1>;
                move to <0 0 .5>; material is plastic colored blue;
        end;
end;    // robot
```

```
anim key_controlled_robot with
        eye = <2 -6 4>;
        lookat = <0 0 1>;
is
        scalar angle1 = 60, angle2 = 10, angle3 = 30;

        verb check_keys is
                const integer angle1_key = char_to_key of "j";
                const integer reverse_angle1_key = char_to_key of "u";
                const integer angle2_key = char_to_key of "k";
                const integer reverse_angle2_key = char_to_key of "i";
                const integer angle3_key = char_to_key of "l";
                const integer reverse_angle3_key = char_to_key of "o";

                if key_down angle1_key then
                        angle1 = itself + 5;
                elseif key_down reverse_angle1_key then
                        angle1 = itself - 5;
                elseif key_down angle2_key then
                        angle2 = itself + 5;
                elseif key_down reverse_angle2_key then
                        angle2 = itself - 5;
                elseif key_down angle3_key then
                        angle3 = itself + 5;
                elseif key_down reverse_angle3_key then
                        angle3 = itself - 5;
                end;
        end;    // check_keys

        shape scene with
                scalar time = 0;
        is
                robot with
                        base_rotation = angle1; arm1_angle = angle2; arm2_angle = angle3;
                end;
        end;   // scene

        mouse_controlled_animated_shape scene doing check_keys;
end;   // key_controlled_robot
```

Figure 14.3: A Keyboard Controlled Robot

KEYBOARD EVENTS

One potential problem with using the keyboard query function is that it looks at the instantaneous state of the keyboard. If you press the key fast enough, it's possible that the applet may miss the key press, especially if the frame rate is not very high. A better way to handle this type of situation is to keep a queue of key press events. Each time you strike a key on your keyboard, your computer takes note by entering an event in the keyboard event queue. Hypercosm provides a means of checking this event queue for keyboard events by calling the function "get_key". This function is located in the file "native_keyboard.ores", which is located in the directory "Hypercosm Studio/Includes/Devices/Keyboard/". When "get_key" is called, it removes the oldest event remaining in the queue and returns information about it. The integer that get_key returns is the keycode of the key that was struck.

```
integer question get_key
return with
        boolean shift;
        boolean alt;
        boolean control;
end;
```

Figure 14.4: The Keyboard Event Querying Function

In addition to returning a keycode, get_key also has optional return parameters to indicate whether a modifier key – Shift, Alt/Option, Control, or Caps Lock – was pressed down when the key was struck. There is no caps-lock parameter. Instead, the shift parameter is adjusted accordingly. If Caps Lock is down and a letter key is struck, then shift will be true; otherwise, the Caps Lock key has no effect.

The program shown in Figure 14.5 listens for key events, and writes out the keycode of the last key event encountered. This little program can be used as a handy utility to figure out what the keycodes are for various keys on the computer keyboard.

EXAMPLE: "SHOW_KEYCODES.OMAR"


```
do show_keycodes;

include "system/3d.ores";
include "system/string_conversions.ores";
include "system/native_overlay_text.ores";

integer keycode = 0;

picture show_keycode is
        string type message;

        message is "last key = ";
        message add integer_to_string of keycode;
        overlay_text message;
end;    // show_keycode

verb update_keycode is
        integer new_keycode = get_key;          // get the most recent key event from the queue

        if new_keycode <> 0 then
                keycode = new_keycode;
        end;
end;    // update_keycode

anim show_keycodes is
        while true do
                show_keycode;
                update_keycode;
        end;
end;    // show_keycodes
```

Figure 14.5: Keycode Display

THE MOUSE

The most common means for interacting with 3D models and simulations is by using the mouse. For this purpose, Hypercosm provides a set of mouse functions that allow you to determine the state of the mouse or to report mouse events.

THE MOUSE LOCATION

For querying the current location of the mouse, Hypercosm provides the function "mouse_down", in the file "native_mouse.ores", which is located in the directory "Hypercosm Studio/Includes/Devices/Mouse/". The function "get_mouse" returns a vector that contains the location of the mouse cursor. The x component of the vector indicates the horizontal position of the cursor and the y component indicates the vertical position. The z component is not used. The components of the vector may be extracted by using the dot operator.

```
vector question get_mouse
        enum coords is raster, screen;
        in coords type coords is screen;
end;   // get_mouse
```

Figure 14.6: The Mouse Location Querying Function

SCREEN COORDINATES

The default coordinate system that is used to record the mouse cursor location is known as "screen coordinates". In screen coordinates, the center of the window is at <0 0 0>. The upper left corner of the viewing window is at <-1 1 0>, the upper right corner is at <1 1 0>, the lower right corner is at <1 –1 0> and the lower left corner is at <-1 –1 0>.

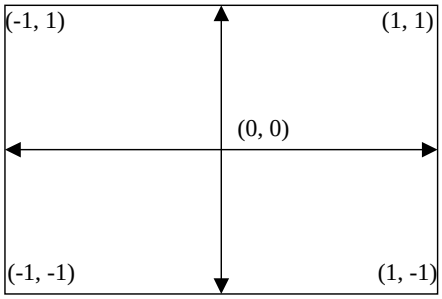


Figure 14.6: Screen Coordinates

RASTER COORDINATES

The other option for mouse coordinate systems is raster coordinates. This coordinate system returns coordinates in "pixels" with the origin at the upper left corner of the window. This coordinate system is not usually used because we usually don't want appets to behave in a way that is dependent upon applet size.

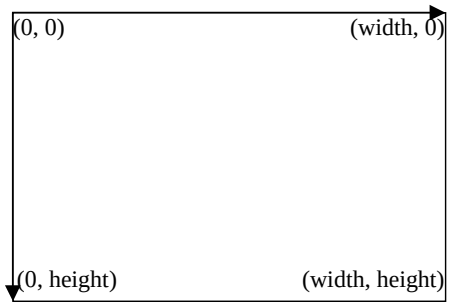


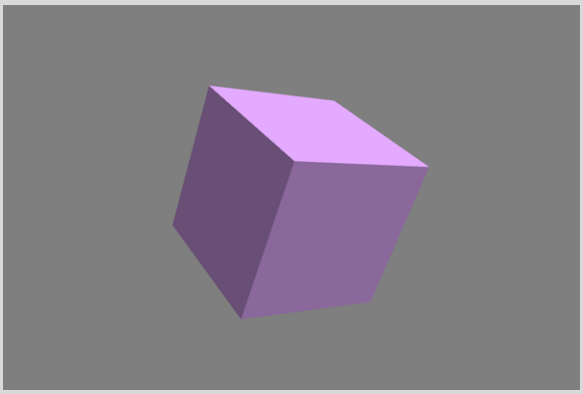
Figure 14.7: Raster Coordinates

EXAMPLE: "SHOW_MOUSE_COORDS.OMAR"


```
do show_mouse_coords;

include "system/3d.ores";
include "system/native_overlay_text.ores";
include "system/string_conversions.ores";

picture scene is
        vector screen_coords, raster_coords;
        string type message;

        // query mouse location
        //
        screen_coords = get_mouse in screen;
        raster_coords = get_mouse in raster;

        // write mouse location in screen coords
        //
        message is "mouse in screen coordinates = ";
        message add vector_to_string of screen_coords;
        overlay_text message at <0 .1 0>;

        // write mouse location in raster coords
        //
        message is "mouse in raster coordinates = ";
        message add vector_to_string of raster_coords;
        overlay_text message at <0 -.1 0>;
end;    // scene

anim show_mouse_coords is
        while true do
                scene;
        end;
end;    // show_mouse_coords
```

Figure 148: Querying the Mouse Location

EXAMPLE: "MOUSE_ROTATION.OMAR



```
do mouse_rotation;

include "system/3d.ores";
include "system/mouse.ores";

picture scene with
        eye = <0 -8 0>;
is
        // query the mouse location
        //
        vector mouse = get_mouse;

        distant_light from <1 -3 2>;
        block with
                rotate by mouse.y * -180 around <1 0 0>;      // rotate by mouse's y coordinate
                rotate by mouse.x * 180 around <0 0 1>;       // rotate by mouse's x coordinate
        end;
end;    // scene

anim mouse_rotation is
        while true do scene; end;
end;    // mouse_rotation
```

Figure 14.7: Example of Mouse Location Querying

THE MOUSE BUTTON

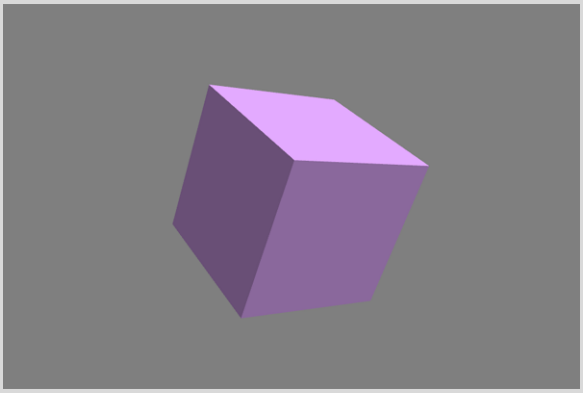
Another useful function is the "mouse_down" function. The "mouse_down" function is defined in the file "native_mouse.ores", which is located in the directory "Hypercosm Studio/Includes/Devices/Mouse/". This function takes a mouse button number as a parameter and then checks whether the corresponding mouse button is pressed. If the button is pressed, the function answers true, otherwise it answers false.

```
boolean question mouse_down
        button integer number = 1;
end;    // mouse_down
```

Figure 14.8: The Mouse Button Querying Function

On a two or three button mouse, the left button is button 1 and the right button is button 3. On a three button mouse, the middle button is button 2. On a one button mouse, the only mouse button is number 1.

EXAMPLE: "SIMPLE_MOUSE_CONTROL.OMAR"



```
do example;

include "system/3d.ores";

vector location = <0 0 0>, orientation = <0 0 0>;

picture scene with
        eye = <0 -8 0>;
is
        distant_light from <1 -3 2>;
        block with
                rotation of orientation;
                move to location;
        end;
end;    // scene

anim example is
        vector old_mouse = <0 0 0>;

        while true do
                vector new_mouse = get_mouse;                   // query mouse location
                vector delta = new_mouse - old_mouse;

                scene;
                if mouse_down of left then                      // query mouse button status
                        location = itself + <delta.x 0 delta.y> * 4;
                else
                        orientation = itself + <delta.y 0 delta.x> * 360;
                end;
                old_mouse = new_mouse;
        end;
end;    // example
```

Figure 14.9: Using the Mouse Location and Button Querying Functions

MOUSE CLICKS

The mouse_down function is useful for interactions that require the user to press and hold down a mouse button as in the example shown above. However, mouse_down is not very useful for interactions that require the user to click or double_click a mouse_button. This is because mouse_down ony checks the current state of the mouse. If the animation frame rate is slow, then mouse_down can easily miss a mouse click. Instead of using mouse_down to detect mouse clicks, Hypercosm provides the function "get_click". This function is defined in the file "native_mouse", which is located in the directory "Hypercosm Studio/Includes/Devices/Mouse". When using this function, mouse clicks are all captured and stored on a queue inside the Hypercosm Player. When get_click is called, a mouse event is removed from the queue and the information about the precise location and conditions of this mouse click is returned.

```
enum click is down, double_click, up;

click type question get_click
        enum coords is raster, screen;
        in coords type coords is screen;
        of button integer requested_button = 0;
return with
        integer button;
        vector location;
        boolean shift;
        boolean alt;
        boolean control;
        boolean caps_lock;
end;   // get_click
```

Figure 14.10: The Mouse Click Querying Function

The get_click function has a number of useful optional return parameters. The button parameter indicates which mouse button was clicked. The location parameter indicates where the cursor was when the click occurred. The boolean parameters – shift, alt, control, and caps_lock – indicate which of the corresponding modifier keys were pressed when the click occurred.

LESSON 14: PICKING

LESSON OBJECTIVES:

- Understand the picking process
- Know how to tag objects for selection
- Know how to test for object touching

LESSON CONTENTS:

- What is picking?
- The picking process
- Tagging shapes using selecton sets
- Testing for mouse touching

COPYRIGHT©2006

WHAT IS PICKING?

Picking is the process of determining what objects in a 3D scene appear to have been touched by the mouse cursor on the screen. This process is somewhat more complicated than it would be for a 2D graphics application because the mouse cursor is located in the 2D coordinate system of the screen and the scene is in a 3D space. To do picking, we need to test objects that lie along a ray that is projected from the eye, through this point on the screen and into the 3D scene. Although this process is non-trivial, luckily the difficult part is taken care of for you by the Hypercosm system and the 3D graphics hardware on your computer system.

THE PICKING PROCESS

To do picking, you must first include the file "native_picking.ores" in your OMAR file. This file is located in the directory "Hypercosm Studio/Includes/Sensing/" and contains the utilities described in the next few sections. Once you've done this, the picking process occurs in two parts:

1. Tagging shapes in the scene with picking identifiers
2. Testing to see if a particular picking identifier has been touched by the mouse cursor

TAGGING SHAPES USING SELECTION SETS

The first tagging part of the process is performed by setting a global variable called "selection_set".

```
native integer selection_set;
```

Figure 15.1: The Selection Set Variable

To tag a particular shape, set this variable inside the "with" block when you instantiate the shape. The "selection set" is an integer identifier that identifies that shape as being selectable. Note that you can have multiple objects that are disconnected share the same selection set and then from a selection point of view, they will behave like they are part of the same objects.

```
<shape name> with
        move to <0 0 1>;                    // transformation
        material is plastic colored red;    // setting material attributes
        selection_set = 1;                  // setting selection attributes
end;
```

Figure 15.2: Tagging a Shape

TESTING FOR MOUSE TOUCHING

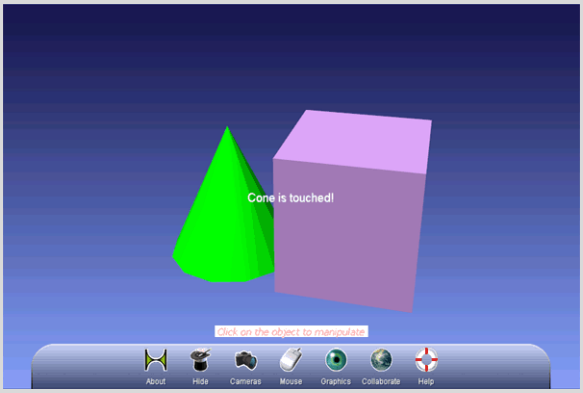
Once you have tagged your shapes in the scene with selection set identifiers, you can then test the mouse cursor location to see if any of these objects are being touched. This is done by using the function "is_shape_touched".

```
native boolean question is_shape_touched
        using integer selection_set;
return
        at scalar depth;
end;   // is_shape_touched
```

Figure 15.3: The Mouse Touching Function

Note that "is_shape_touched" also has an optional parameter for returning the depth or distance to the point where the touching has occurred. This can be used to find out the location of the actual touch point in the 3D scene by projecting a ray this distance from the eye in the direction projected by the cursor.

EXAMPLE: "SIMPLE_PICKING.OMAR"




```
do example;

include "system/3d.ores";
include "system/anims.ores";
include "system/native_picking.ores";

shape scene is
        block with
                move to <1 0 0>;
                selection_set = 1;                      // set selection set of block
        end;
        cone with
                move to <-1 0 0>;
                selection_set = 2;                      // set selection set of cone
        end;
end;

verb picking is
        if is_shape_touched using 1 then                // test selection set of block
                overlay_text "Block is touched!";
                write "block touched", ;
        elseif is_shape_touched using 2 then            // test selection set of cone
                overlay_text "Cone is touched!";
                write "cone touched!", ;
        end;
end;    // picking

anim example with
        eye = <2 -8 4>;
is
        mouse_controlled_shape scene doing picking;
end;    // example
```

Figure 15.4: An Example of Picking

LESSON 15: SOUND

LESSON OBJECTIVES:

- Know how to create sounds and play sounds
- Know how to include sounds in your projects

LESSON CONTENTS:

- Why Use Sound?
- Creating new sounds
- Playing sounds

WHY USE SOUND?

One of the most compelling reasons to use interactive 3D graphics is the fact that it is a very compelling and engaging medium. One reason that interactive 3D is so engaging is because it engages both our eyes and our hands at the same time. Interactive 3D is inherently a multisensory experience. The more sensory channels that we can simultaneously engage, the more compelling the experience will be. A relatively easy way to make 3D applications even more compelling is to use sound to engage our sense of hearing. There is a reason that sound has been an integral component of video game design since its inception. For training applications, one of the most compelling ways to engage the user is to (1) provide a set of instructions to read while we (2) show instructions graphically animated while we (3) allow the user to manipulate the scene with his or her hands while at the same time we (3) play audio instructions for the user to hear.

CREATING SOUNDS

Sounds are stored in external files. Sound file format that can be used are .WAV files and .MP3 files. To use a sound in Hypercosm, you must first create a sound object that references an external sound file. The definition of the sound objects is stored in the file "native_sounds.ores", which is located in the directory "Hypercosm Studio/Includes/Sounds/".

```
sound type <sound name> named string type name;
```

Figure 16.1: Creating a new sound

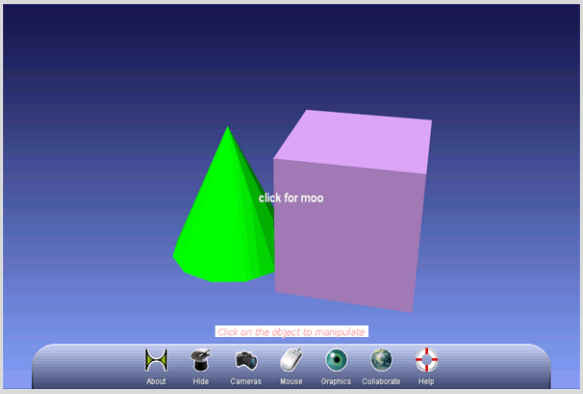
PLAYING SOUNDS

Once you have created a sound, you can call "methods" of that sound object in order to use it. Methods are just like procedures or "verbs" that you've previously encountered except that they are preceded by the name of the sound object that you want to perform the method.

```
<sound name> play;
```

Figure 16.2: Playing a sound

EXAMPLE: "SONIC_SHAPES.OMAR"




```
do example;

include "system/3d.ores";
include "system/anims.ores";
include "system/native_picking.ores";
include "system/native_overlay_text.ores";

// create new sounds
//
sound type sound1 named "quack.wav";
sound type sound2 named "moo.wav";

shape scene is
        block with
                move to <1 0 0>; selection_set = 1;
        end;
        cone with
                move to <-1 0 0>; selection_set = 2;
        end;
end;    // scene

verb picking is
        if is_shape_touched using 1 then
                overlay_text "click for quack";
                reset_frame_events;
                if some get_click then
                        sound1 play;                    // play sound1
                end;
        elseif is_shape_touched using 2 then
                overlay_text "click for moo";
                reset_frame_events;
                if some get_click then
                        sound2 play;                    // play sound2
                end;
        end;
end;    // picking

anim example with
        eye = <2 -8 4>;
is
        mouse_controlled_shape scene doing picking;
end;  // example
```

Figure 16.3: Using sounds

LESSON 16:

TEXT

LESSON OBJECTIVES:

- Know the different types of text available
- Know how to create 2D overlay and 3D renderable text displays

LESSON CONTENTS:

- Types of text display
- Overlay text
- Renderable text

TYPES OF TEXT DISPLAY

In previous examples, we've made use of "write" statements to print statements to the output window in Hypercosm Studio. While this works fine for developing applets, without Hypercosm Studio, there's no way to view the text output. To display text inside of the applet with the 3D graphics, Hypercosm provides two different methods: 2D overlay text and 3D renderable text.

2D OVERLAY TEXT

As you might expect, 2D overlay text is simply regular two dimensional text that is drawn on top of the 3D graphics. When you view and applet that displays the standard Hypercosm dock bar interface with the icons and other controls at the bottom of the window, the text labels that you see are drawn using overlay text. To draw overlay text, include the file "native_overlay_text.ores", which is located in the directory "Hypercosm Studio/Includes/Text/Overlay Text". This file contains the interface to the procedure, "overlay_text". The overlay_text procedure uses parameters for the string of characters to display, a location and various parameters that control the appearance and font of the text.

```
native verb overlay_text
        string type string;
        at screen_coords type position = <0 0 0>;
with
        string type font_family is default_font_family;
        boolean bold is false;
        boolean italic is false;
        boolean underlined is false;
        integer size = get_default_title_text_size;
        horizontal_alignment type horizontal_alignment is center;
        vertical_alignment type vertical_alignment is middle;
        color type color = white;
        boolean transparent_background is true;
        color type background_color = black;
end;    // overlay_text
```

Figure 17.1: Definition of Overlay Text

TEXT POSITION

The position for the text is specified in screen coordinates where <0 0 0> is at the center of the applet window, <-1 –1 0> is at the lower left and <1 1 0> is at the upper right. The z coordinate of the screen coordinates is not used. Note that "overlay_text" is a verb, not a shape. The location where it draws is not affected by "move to" or other transformations.

FONT

Overlay text draws text using a font chosen from the set of fonts available from the operating system. The "font_family" string allows you to specify a case-insensitive comma-separated list of font names for the Player to look for. If the "font_family" is not specified, the Player will look for "Arial" as a default font. If "Arial" can't be found, the Hypercosm Player will use the system's default font.

SIZE

The size parameter is in pixels and is the height of the text (the font ascent plus the font descent). The text will be drawn using the best available match to the requested size. It's good practice to set the size to be a fraction of your window height, so that the text is resized proportionally if your applet is run at a different window size.

EXAMPLE: "OVERLAY_TEXT_EXAMPLE.OMAR"

```
do overlay_text_example;

include "system/3d.ores";
include "system/native_overlay_text.ores";

picture overlay_text_example is
        overlay_text "Overlay Text!" at <0 0 0> with
                bold is true;
                underlined is true;
                horizontal_alignment is center;
                color = white;
                background_color = red;
                transparent_background is false;
        end;
end;    // overlay_text_example
```

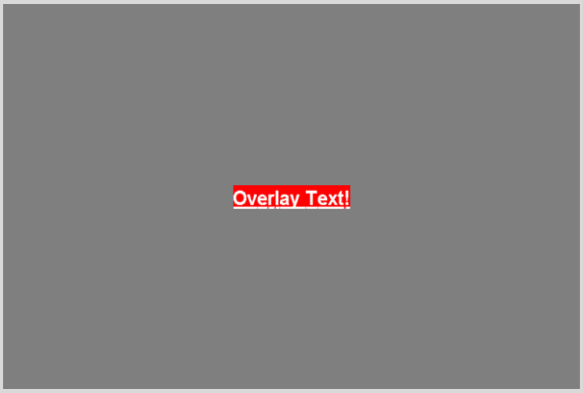


Figure 17.2: Using Overlay Text

3D RENDERABLE TEXT

While overlay text is quite useful, there are occasions where you want the text to exist in your 3D world and to be affected by the camera, lighting and shading. This is where renderable text is used. To use renderable text, include the file "renderable_text.ores", which is located in the directory "Hypercosm Studio/Includes/Text/Renderable Text/". Inside this file, you'll find the definition of a shape called "text". Note that this definition of text is a true shape and therefore may be transformed, included in other shapes, and have materials applied just like other shapes.

```
shape text
        string type string;
        using text_style type text_style is simple_text;
with
        renderable_font_family type font_family is text_style's font_family;
        boolean bold is text_style's bold;
        boolean italic is text_style's italic;
        boolean underline is text_style's underline;

        scalar size = text_style's size;
        scalar depth = text_style's depth;
        scalar length = text_style's length;

        horizontal_alignment type h_align is text_style's h_align;
        vertical_alignment type v_align is text_style's v_align;

        color type color = text_style's color;
        material type material is text_style's material;
end;
```

Figure 17.3: Definition of Renderable Text

EXAMPLE: "RENDERABLE_TEXT_EXAMPLE.OMAR"


```
do renderable_text_example;

include "system/3d.ores";
include "system/anims.ores";
include "system/renderable_text.ores";

shape words is
        text_style type style1 with
                color = orange;
                size = .75;
        end;

        text "plain & simple & centered";

        text "Underlined Text" with
                underline is on;
                color = light brown;
                move to <0 -1.5 3>;
        end;

        text "left-aligned, bottom-aligned" using style1 with
                move to <-3 0 1>;
                h_align is left;
                v_align is bottom;
        end;

        text "right-aligned, top-aligned" using style1 with
                move to <3 0 -1>;
                h_align is right;
                v_align is top;
        end;
end;    // words

anim renderable_text_example with
        eye = <-5 -24 0>;
is
        mouse_controlled_shape words;
end;    // renderable_text_example
```

Figure 17.4: Using Renderable Text

LESSON 17:

2D OVERLAY GRAPHICS

LESSON OBJECTIVES:

- Be able to create 2D overlay graphics and controls

LESSON CONTENTS:

- What overlays are used for
- The overlay transformation
- Scaling overlay graphics

USES FOR 2D OVERLAY GRAPHICS

In simulation and training applications, we often find a need for "overlay" graphics. Overlay graphics are 2D graphics that lie on top of the 3D graphics and are not affected by the transformations of the underlying 3D scene. These kinds of overlay graphics are typically used for displays of various kinds. The most familiar examples of this type of display are cockpit instrument displays that are used in most flight simulators, driving simulators, and other vehicle simulators.

THE OVERLAY TRANSFORMATION

In Hypercosm, 2D scenes are treated as a subset of 3D scenes. They are specified in the same way that you would specify the shapes of a 3D scene except that the geometry lies in the X-Y plane. When the 2D shapes are to be displayed, we rely upon a utility transformation procedure to place the 2D shapes in the 3D scene in such a way that they are aligned and scaled properly with respect to the camera. The transformation that is used to perform this task is the "overlay" transformation that is contained in the file "view_alignments.ores", which is located in the directory "Hypercosm Studio/Includes/Viewing/".

```
verb overlay
        at scalar distance;
end;
```

Figure 18.1: The Overlay Transformation

This overlay transformation will transform your shape in such a way that a unit square on the X-Y plane will be transformed to fill the screen. Since the overlay actually exists in the same 3D space as the rest of the 3D scene, you must provide a distance to the overlay which describes how far from the camera the overlay will be. This distance must be greater than 0.

SCALING OVERLAY GRAPHICS

Scaling overlay graphics presents a few minor challenges. As described above, the standard overlay transformation stretches an object to fit the display window. If we want an overlay that encompasses the entire window, then this is fine. For many applications, however, we want the overlay to take up just a portion of the window. In this case, we must think about how we want to specify the size of the graphic.

DEALING WITH VARIABLE WINDOW ASPECT RATIO

Since the overlay transformation automatically scales the graphic to the size of the window, it's easy to specify the size of the graphic just by scaling it by a fraction of the window width or height. The one remaining thing to think about is that if we do this, the aspect ratio of the graphic may not be correct. The overlay method stretches an object to fit the aspect ratio of the window. In cases where you don't want the graphic to stretch, you must scale the graphic by the inverse of this aspect ratio to counter this stretching. The aspect ratio of the window can be found by the expression: (height / width).

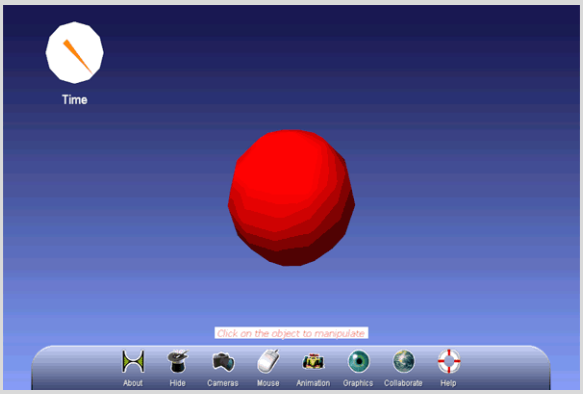
SPECIFYING THE OVERLAY GRAPHIC SIZE IN PIXELS

One remaining way to specify the overlay graphic size is in pixels. Normally, we don't ever want to specify the width and height of a graphic in terms of pixels because we'd like to be able to resize the applet and have all of the graphics in the Hypercosm applet scale along with the window. However, if we do want the overlay graphic to always remain a constant size, then we can use the window size to scale the graphic to achieve this effect. The size of the window can be found by looking at the window size variables, "width" and "height", which are contained in the file "native_display.ores".

```
native integer width;
native integer height;
```

Figure 18.2: Window Size Variables

EXAMPLE: "OVERLAY_GAUGE.OMAR"




```
do overlay_gauge;

include "system/3d.ores";
include "system/anims.ores";
include "system/view_alignments.ores";

shape gauge with
        scalar value = 0;
is
        disk with color = white; end;
        triangle <0 1 0> <-.1 -.5 0> <.1 -.5 0> with
                rotate by value * 360 around <0 0 -1>;
                move to <0 0 .1>;                          // move dial shape above disk
                color = orange;
        end;
end;    // gauge

shape scene with
        scalar time = 0;
is
        scalar gauge_size = .1, aspect_ratio = height / width;
        vector gauge_location = <-.75 .75 0>, label_location = gauge_location - <0 (gauge_size * 2) 0>;

        sphere;
        gauge with
                value = get_seconds;
                scale by gauge_size along <1 0 0>;
                scale by gauge_size / aspect_ratio along <0 1 0>;
                move to gauge_location;
                overlay at 1;                              // apply overlay transformation
        end;
        overlay_text "Time" at label_location with         // draw label
                vertical_alignment is top;
        end;
end;    // scene

anim overlay_gauge with
        eye = <2 -8 4>;
is
        mouse_controlled_animated_shape scene;
end;    // overlay_gauge
```

Figure 18.3: An Example Overlay Display

LESSON 18: BEGINNING OBJECT ORIENTED PROGRAMMING

LESSON OBJECTIVES:

- Understand the concepts of classes and objects
- Be able to create simple classes

LESSON CONTENTS:

- What is object oriented programming?
- Class components
- Class declarations
- Class instances or "objects"
- Calling an object's methods
- Constructors

COPYRIGHT©2006

WHAT IS OBJECT ORIENTED PROGRAMMING

Object-oriented programming (OOP) is mainly a way to organize and package code so that it can be more easily managed, maintained, and reused. Although languages such as C++ make object oriented programming quite complex and the terminology used to describe the features of these languages is often confusing, the basic concepts of object oriented programming are quite simple.

CLASSES

The basic idea behind object oriented programming is that instead of having the data and instructions for each conceptual entity spread throughout the program or grouped together simply by convention, there ought to be a way that you can explicitly group together the data and instructions which belong to a particular idea and package them as one complete description of that idea. This conceptual unit is called a "class".

CLASSES AND OBJECTS

A class is a way of describing a conceptual entity. When we actually write a program, however, we deal with things not as conceptual entities, but as concrete instances of those entities. For example, a class might describe the concept of a "list". However, when we actually run a program, we are interested not in lists as a concept, but in actually creating and using specific lists. When we create a specific instance of a class, it is called an "object". Objects are instances of classes.

ANALOGIES FOR CLASSES

Perhaps a useful way to look at classes and objects is by analogy with Plato's concept of Ideals. Plato thought that reality could be described as a series of platonic ideals that captured the essence of things that are actually present in reality. For example, he thought that there is an idealized concept of a tree that is independent of any specific tree (this also ties into DNA). This is similar to the idea of classes and objects. The class is sort of like the Platonic ideal, whereas the object is sort of like the manifestation of the Platonic ideal in reality. The class is a template for an object – the DNA, if you will.

CLASS COMPONENTS

The two main components of the class are (1) the data that are used to represent an object and (2) the instructions that describe what that object can do.

DATA – MEMBER VARIABLES

The data consists of a number of variables that are called "member" variables. These are declared just like regular variables except that they are part of the class. In most cases, a new set of these variables is created each time we create an instance of that class (object).

INSTRUCTIONS - METHODS

The instructions that belong to the class are packaged as a series of procedures or functions. These procedures and functions are known as "methods". Sometimes they are also called "member functions". These methods are declared just like regular procedures and functions and work similarly except that they can access the member variables of that class.

CLASS DECLARATIONS

Class declarations can be fairly complex constructions since they may be composed of a number of fields and methods. To make class declarations easier to read, in OMAR they are broken up into two parts: the interface and the implmentation. Note that this is different from languages such as Java and C# where the class declaration has no specific interface section.

```
subject <name>
does
        <method declarations>           // The interface portion:
has
        <member declarations>
is                                      // The impementation portion:
        <declarations>
end;
```

Figure 19.1: The Basic Class Declaration

In the OMAR language, classes are called "subjects" in keeping with the English language paradigm where a subject is an object that is the origin of a predicate consisting of a verb and possibly more arguments (parameters). This will become clearer as we begin to write some object oriented code.

THE INTERFACE

When using a particular class, we ought to be able to view only the information that tells how the class is to be used and to skip the implementation details of the class. This idea should already be familiar from the procedure and function declarations that we have been using. You can use a procedure just by knowing the name and parameters that the procedure accepts without knowing the details of the implementation. In a similar way, the class declaration lists the names and parameters of all available methods at the top without including their implementations.

THE INTERFACE

The complete declarations of all methods complete with implementations are given in the last section of the class declaration where all of the inner workings and details of the class must be defined.

CLASS INSTANCES OR "OBJECTS"

As described before, classes are like templates for objects – they describe how those objects are represented and what they can do, but classes don't actually do anything by themselves. In order to actually do something with a class, you must make an "instance" of the class or "object".

```
<class name> type <variable_name>;
```

Figure 19.2: Creating an Instance of a Class

```
subject circle
does
        // methods
        //
        verb set_radius
                to scalar radius;
        end;
        scalar question get_circumference;
        scalar question get_area;
        verb print;
has
        // members
        //
        scalar radius = 1;
is
        // implementation
        //
        verb set_radius
                to scalar radius;
        is
                circle's radius = radius;
        end;    // set_radius

        scalar question get_circumference is
                answer 2 * pi * radius;
        end;    // get_circumference

        scalar question get_area is
                answer pi * radius * radius;
        end;    // get_area

        verb print is
                write "circle with radius = , radius, ", circumference = ", get_circumference,
                 ", area = ", get_area, ;
        end;    // print
end;    // circle
```

Figure 19.3: An Example Class

```
circle type circle;
circle type circle1, circle2;
```

Figure 19.4: Example Class Instances (Objects)

CALLING AN OBJECT'S METHODS

In OMAR, to invoke an object's method, you simply state the name of the object followed by the method name and the method's parameters, if any. Note that the dot operator is not required as it is in Java or C++. The name of the method simply follows the name of the object. In OMAR, when one these "objects" is used in conjunction with one of its methods, then it is known as a "subject" because from the standpoint of English grammar, this is a more accurate description of its role in the statement.

```
circle type circle;                          // create an object that we can call methods on

circe set_radius to 10;                      // call a procedure method (verb)
write "circle's area  = ", circle get_area, ;  // call a function method (question)
```

Figure 19.5: Calling an Object's Methods

A COMPARISON WITH ENGLISH LANGUAGE GRAMMAR

When calling methods on objects, the similarities between the OMAR language syntax and the English language sentence structure become apparent. This is no accident and helps contribute to much more readable code.

Sentence Structure	English Language	OMAR Language
Verb	Stop!	quit;
Verb – Object(s)	Find lawyers, guns, and money.	rotate by 100 around x_axis;
Subject - Verb	Fred writes.	circle print;
Subject – Verb – Object(s)	Fred wrote my term paper.	circle set_radius to 5;

Figure 19.6: Comparison with English Language Grammar

CONSTRUCTORS

In the example above, creating a new instance of the circle class was a simple matter of stating the type name followed by the instance name of the new object. For more complex objects, however, it's often desirable to perform some kind initialization procedure every time that a new instance is created. This initialization procedure is called a constructor. Constructors are differentiated from the other methods in a class by

reserving the name "new" for constructors. Any method that you declare which is named "new" will be invoked every time an instance of this class is created.

```
subject circle
does
        // methods
        //
        verb new with
                scalar radius = 1;
        end;
has
        // members
        //
        scalar radius = 1;
is
        // implementation
        //
        verb new with
                scalar radius = 1;
        is
                circle's radius = radius;
        end;    // set_radius
end;    // circle
```

Figure 19.7: An Example Constructor

If a class has a constructor defined, then whenever you make an instance of that class, the constructor is called. If the constructor method has parameters defined, then your object declaration must supply values to those parameters when the object is created.

```
circle type circle;                     // constructor is called, but its optional parameters is not specified

circle type small_circle with           // constructor is called with a value of .5 for the radius parameter
        radius = .5;
end;

circle type large_circle with           // constructor is called with a value of 100 for the radius parameter
        radius = 100;
end;
```

Figure 19.8: Calling A Constructor

LESSON 19: INTERMEDIATE OBJECT ORIENTED PROGRAMMING

LESSON OBJECTIVES:

- Know how to create a subclass by extending an existing class
- Understand how dynamic binding works
- Know how to declare and implement class interfaces ("adjectives")

LESSON CONTENTS:

- Code Reuse
- Inheritance
- Extending a class
- Constructor chaining
- Overriding methods
- Interfaces

COPYRIGHT©2006

CODE REUSE

There exists, out there somewhere, code for doing just about every imaginable thing. Programs, however, are very complex, intricately connected things, like the spaghetti wiring inside of an old stereo system. If you want a particular set of features in your program, you often find that it is easier (and less error prone and dangerous) to rewrite everything from scratch than to modify an existing program. Obviously, this is one reason why software creation is so time consuming and expensive. Object oriented programming is an effort to make code more reusable and more robust.

INHERITANCE

The basic idea behind inheritance is that you can create classes that are derived from existing classes instead of writing each one from scratch. These derived classes inherit the functionality of the class that they are derived from (the parent class), but they may also extend or override the functionality of their parent class. If a class extends another class, then it may have additional data and / or methods that the parent class does not have. Experienced object oriented programmers know that one of the most difficult things about designing well structured code is to carefully build these class hierarchies, so that each class neither does too much nor does too little. If the class does too much, then it is overly difficult and cumbersome to use. It the class does too little, then you end up with lots of little classes, each only slightly different and hard to distinguish, and the program once again becomes difficult to understand.

SINGLE INHERITANCE

In OMAR, each class can have just one parent class. This is how inheritance is handled in many other languages such as Java and C#. A few other modern languages allow a class to have multiple parent classes. This is known as “multiple inheritance”. This is a powerful feature, but causes a number of complications and problems, which is why OMAR uses the single inheritance model.

EXTENDING A CLASS

To add new features to an existing class, you create a new class that extends the features of the old class. This new class is known as a "subclass" because it is a more specific version of the more general class that it was derived from. This terminology may be a little confusing because a "subclass" actually has a superset of the functionality of its parent class. A subclass is less general but has more methods and members than its parent class. To create a new subclass, use the following format:

```
subject <class name>
extends
        <parent class name>             // Class that this class is derived from
does
        <method declarations>           // The interface
has
        <member declarations>
is                                      // The impementation
        <declarations>
end;
```

Figure 20.1: Extending a Class

All classes are considered to be implicitly derived from a base class, the "object" class. If you don't have an "extends" clause in your class declaration, there is an invisible "extends object" clause which is implicitly added for you. Note that when you extend a class, you inherit not only the functionality of the immediate superclass, but you also inherit the functionality of every superclass all the way back to the "object" class. The way to think about it is that if class B extends class A, then an instance of B IS an instance of A. For example, if you have an "employee" class that extends the class, "person", then each employee IS a person. Likewise, if you extend the employee class by an "executive" class, then each executive IS also an employee and IS a person.

CONSTRUCTOR CHAINING

When we create a new class by extending an existing class, we inherit all of the fields of that existing class and therefore, we must make sure that the integrity of those fields is maintained. This leads to the requirement that constructors must be called not only to initialize the fields of objects of their own class, but also to initialize the fields of objects belonging to any classes that are derived from this class. This is known as "constructor chaining". The way that this is enforced is to require the first line of any constructor which belongs to a class with a superclass constructor to call that superclass constructor before doing anything else.

```
subject subclass
extends
        superclass                      // This is a class with a constructor defined
does
        verb new;                       // This subclass must have its own constructor defined since
has                                     // its superclass has a constructor defined
        integer stuff;
is
        verb new is                     // The first line of the subclass's constructor must be a call
                superclass new;         // to its superclass's constructor to initialize its inherited fields

                stuff = 0;              // After, we can initialize the subclass's noninherited fields
        end;
end;    // subclass
```

Figure 20.2: Constructor Chaining

```
subject person
does
        verb new
                named string type name;
        end;    // new
        string type question get_name;
has
        string type name;
is
        verb new
                named string type name[];
        is
                person's name = name;
        end;    // new

        integer question get_name is
                answer name;
        end;    // get_name
end;    // person

subject employee
extends
        person
does
        verb new
                named string type name;
                paid integer salary = 0;
        end;    // new
        integer question get_wages;
has
        integer salary;
is
        verb new
                named string type name;
                paid integer salary = 0;
        is
                person new named name;                  // Call superclass constructor
                employee's salary = salary;
        end;    // new

        integer question get_wages is
                answer salary;
        end;    // get_wages
end;    // employee
```

Figure 20.3: Example of Extending a Class

OVERRIDING METHODS

Sometimes, when we create a new subclass, we find that an inherited method is not quite what we want and we can provide a new definition for the inherited method that is more specific to the new subclass. This process of substituting a new method for one that has been inherited is known as "overriding". To override an inherited method, all we have to do is to define a new method for the subclass that has the same name and parameters as an inherited method. When an instance of this derived class calls the method of this name, the new method will be called instead of the inherited one.

```
subject executive
extends
        employee
does
        verb new
                named string type name;
                paid integer salary = 0;
        with
                integer stock_options = 0;
        end;    // new

        integer question get_wages;                     // Overridden method
        integer question get_perks;
has
        integer stock_options;
is
        verb new
                named string type name;
                paid integer salary = 0;
        with
                integer stock_options = 0;
        is
                employee new named name paid salary;
                executive's stock_options = stock_options;
        end;    // new

        integer question get_wages is
                answer salary + stock_options;
        end;    // get_wages

        integer question get_perks is
                answer stock_options;
        end;    // get_perks
end;    // executive
```

Figure 20.4: Extending a Class and Overriding Methods

DYNAMIC BINDING

When you create a new class using inheritance, you expect that if you have provided this class with a replacement method to override an existing method from its parent class, then the new method will be called instead of the old one. Dynamic binding is a fancy term for the technique that is used internally to make this happen. Using this technique of overriding methods and using "dynamic binding" to select the correct method to use, you can write very generic code and then let the objects, themselves, decide the appropriate method implementation to use. For example, suppose that you had to write a program to compute the payroll from a list of employees. You would need to have different routines to compute the wages of each type of employee. To implement this in an object-oriented manner, you could build a class hierarchy where there was a basic employee class plus more specific subclasses for different types of employees (such as executives, contractors, etc.). Each different type of employee could have a different method for computing his or her wages depending upon the type of employee. By structuring the code in this way, the payroll application would not have to be concerned with the details of how each employee computed his or her wages, it would simply call a method to compute wages and then defer to the employee class for the particular implementation. Using object-oriented programming, this is automatically handled in a way that is intuitive and elegant.

```
// Employees of different types
//
employee type grunt named "bill", secretary named "suzie", salesperson named "maggie",
        manager named "joe";
executive type president named "leo", ceo named "barb";

// Table of employees
//
employee type staff[] = [grunt manager president secretary ceo salesperson];

// Code to sum wages of staff
//
integer sum = 0;
for each employee type employee in staff do

        // Here, get_wages will either execute the employee's implementation or
        // the executive's implementation depending upon the type of the employee
        //
        sum = itself + employee get_wages;
end;
```

Figure 20.5: Example of Using Dynamic Binding

INTERFACES

We have already seen how we can extend classes through inheritance. There exists another way that we can extend our classes: through a mechanism known as "interfaces". In the OMAR programming language, interfaces are called "adjectives". An interface (or "adjective") consists of a collection of method interfaces. This is similar to the list of method interfaces that are typically found in the "does" section of a class. Any class that "implements" an interface must provide implementations of each of the methods described in that interface. When a class implements a particular interface, it's basically like a promise that the class will have the set of functionality described in the interface. That is why interfaces are called "adjectives" in OMAR. They basically describe a certain set of functionality that a class may have without describing anything about how that class is supposed to implement that functionality.

```
adjective <adjective name>
does
        <method declarations>
end;
```

Figure 20.6: Declaring an Interface ("Adjective")

In order to have a class implement a particular interface, we must first list the interface in the "extends" clause, before the name of the parent class.

```
subject <class name>
extends
        <adjective names> <parent class name>
does
        <method declarations>           // The interface
has
        <member declarations>
is                                      // The impementation
        <declarations>
end;
```

Figure 20.7: Implementing an Interface ("Adjective")

```
adjective printable
does
        verb print;
end;

subject person
extends
        printable object
does
        verb new
                named string type name;
        end;    // new
        string type question get_name;
        verb print;                             // This method must be here because the person
has                                             // class implements the printable interface which
        string type name;                       // contains the print method
is
        verb new
                named string type name[];
        is
                person's name = name;
        end;    // new

        integer question get_name is
                answer name;
        end;    // get_name

        verb print is
                write "person named ", name, ;
        end;    // print
end;    // person
```

Figure 20.8: Example Declaring and Implementing an Interface

LESSON 20: OBJECT ORIENTED ANIMATION

LESSON OBJECTIVES:

- Understand how to create actors

LESSON CONTENTS:

- Principles of object oriented animation
- Actors
- Mouse controlled actors
- Actor animation
- Timers
- Timed actors

COPYRIGHT©2006

PRINCIPLES OF OBJECT ORIENTED ANIMATION

We have seen how object oriented programming is concerned with the description and creation of "objects" that each have certain sets of behaviors that are described through the methods that they implement. This notion of objects is well suited to programming animation because we can create a scene composed of objects that have methods for displaying themselves as well as methods for animating themselves and triggering other behaviors.

ACTORS

The most basic type of object in an object oriented animation is known as an "actor". An actor is basically any object that "knows" how to render itself and therefore display itself in a 3D scene. In order to make objects renderable, we require that they implement the renderable interface, which is declared in the file "rendering.ores". The renderable interface has only a single method, "instance", which causes the object to create an instance of itself in a scene. This instance method is a shape and therefore can be transformed and have materials applied just like any other shape.

```
adjective renderable
does
        shape instance;
end;
```

Figure 21.1: The Renderable Interface

```
subject actor
extends
        renderable object
does
        shape instance;
end;
```

Figure 21.2: An Actor Declaration

MOUSE CONTROLLED ACTORS

Once you have created an actor, you can add mouse interactivity to enable you to manipulate the actor using the mouse controls in a similar way to the "mouse_controlled_shape" utility that was used previously. This is done by using the mouse_controlled_actor utility. To use this utility, simply include the file "anims.ores" just as described earlier when using the mouse_controlled_shape.

```
do example;

include "system/3d.ores";                       // for core 3D rendering functionality
include "system/anims.ores";                    // for mouse_controlled utilities

subject actor
extends
        renderable object
does
        shape instance;
is
        // actor implementation goes here
end;    // actor

anim example
is
        actor type actor;                       // create an instance of the actor

        mouse_controlled_actor actor with       // pass actor to mouse_controllec_actor
                auto_camera is on;              // utility to add interactive mouse controls
        end;
end;    // example
```

Figure 21.3: Using a Mouse Controlled Actor

ACTOR ANIMATION

Actors often require the ability to execute some code each frame in order animate themselves. This is easily accomplished by having your actors implement the

"updateable" interface. The updateable interface has only a single method, update, which performs the animation code and is called once per animation frame.

```
adjective updatable
does
        verb update;
end;
```

Figure 21.4: The Updatable Interface

```
subject actor
extends
        renderable updatable object
does
        shape instance;
        verb update
end;
```

Figure 21.5: An Updateable Actor Declaration

TIMERS

The animated actors that we previously described have the ability to animate themselves by executing a piece of code each frame. In previous sections, we learned that for animation to be consistent from machine to machine, it needs to be "time based" rather than "frame based". To more easily create time based object oriented animations, Hypercosm provides a series of timers that allow the passage of time depicted by an actor to be easily controlled. These timers are located in the directory "Hypercosm Studio/Includes/Time/Timers". The standard timers provided are as follows:

Timer Name	Description
clock_timer	Updates itself using the system clock.
stopwatch_timer	A clock_timer that may be paused
timer	A stopwatch_timer that may run at a variable rate
interval_timer	A timer that has a particular duration
cycle_timer	An interval_timer that repeats through a cycle

Figure 21.5: Standard Timers

TIMED ACTORS

The basic structure of a timed actor is shown below. Note that in the actor's update method, we must be sure to remember to update the timer.

```
subject actor
extends
        renderable updatable object
does
        shape instance;
        verb update;
has
        timer type timer;
is
        verb update is
                timer update;
        end;
end;
```

Figure 21.5: A Timed Actor Declaration

LESSON 21: INTEGRATING WITH JAVASCRIPT

LESSON OBJECTIVES:

- Understand how to communicate from a Hypercosm applet to a web page
- Understand how to communicate from a web page to a Hypercosm applet

LESSON CONTENTS:

- JavaScript for integrating web application components
- Communicating from a web page to a Hypercosm applet
  - Sending messages from a web page to a Hypercosm applet
  - Listening for messages in a Hypercosm applet
- Communicating from a Hypercosm applet to a web page

JAVASCRIPT FOR INTEGRATING WEB APPLICATION COMPONENTS

As web applications get more and more complex, developers find that it's not feasible to create the entire web application using a single tool. The best solution is often to create a web application using a variety of tools. This allows each component to be created using the tools that are most appropriate. It may make sense to have some user interface components implemented in Macromedia Flash connected to some graphical 3D components that are implemented using Hypercosm and some networking components implemented using Java. JavaScript serves as a sort of universal "glue" for connecting these various web based components together and for sending information between them.

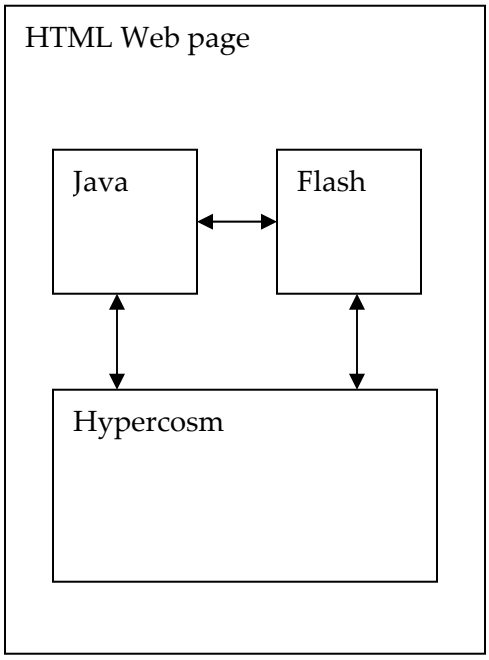


Figure 22.1: JavaScript for Connecting Web Components

COMMUNICATING FROM A WEB PAGE TO A HYPERCOSM APPLET

Hypercosm makes it possible to communicate information from a web page into a Hypercosm applet that is a component on that web page. This is done using a message passing mechanism. The messages that are passed into the Hypercosm applet consist of strings of characters only. There are two parts to this mechanism:

1) There must be JavaScript code in the web page that sends messages from the web page to a Hypercosm applet.
2) There must be code in the Hypercosm applet to listen for messages and then to act upon messages that are received and recognized.

SENDING MESSAGES FROM THE WEB PAGE TO THE HYPERCOSM APPLET

The task of sending messages from the web page to a Hypercosm applet is illustrated by the code below:

```
<SCRIPT LANGUAGE="JavaScript">

var InternetExplorer = navigator.appName.indexOf("Microsoft") != -1;

function GetPlugin()
{
        return InternetExplorer ? window.HyperX1 : window.document.HyperX1;
}

function SendMessage(message)
{
        GetPlugin().SendMessage(message);
}

</SCRIPT>
```

Figure 22.2: Sending Messages from the Web Page To A Hypercosm Applet

LISTENING FOR MESSAGES IN A HYPERCOSM APPLET

The task of listening for messages in a Hypercosm applet is made possible by a pair of native methods that is included in the file "native_messages.ores", which is located in the directory "Hypercosm Studio/Includes/Utilities/Messaging/". The first of these methods, "get_message_number", is used to return the number of messages that are waiting on the message queue. The second method "get_next_message", returns the next message from the queue.

```
// This method returns the number of messages
// that are waiting on the message queue.
//
native integer question get_message_number;

// This method returns a message from the message
// queue and also removes it from the queue.
//
native string type question get_next_message;
```

Figure 22.3: Message Listening Methods

In order to use these methods in your Hypercosm applet, you will usually want to create a message handling loop that checks to see if there are any messages and if so, retrieves and interprets those messages. An example message handling loop is shown below:

```
verb check_messages is
        while get_message_number > 0 do
                string type message is get_next_message;

                if message matches command then
                        // do something
                end;
        end;
end;   // check_messages
```

Figure 22.4: Message Checking Hypercosm Script Code

COMMUNICATING FROM A HYPERCOSM APPLET TO A WEB PAGE

The second part of the problem is communicating information from the Hypercosm applet back out to the web page. Once this information is received by the web page, it may be routed to other components on the web page including Flash, Java, or even other Hypercosm components. Communicating information out to the web page is done using a simple command "exec_script" that is found in the file "native_exec_script.ores", which is located in the directory "Hypercosm Studio/Includes/Utilities/JavaScript/".

```
// This will execute script when the player is in
// plugin mode.
// example: exec_script "changeCol('#cc0000')" "JavaScript";
// example: exec_script "alert('Hello')" "JavaScript";
//
native verb exec_script
        string type code;
        string type language;
end;
```

Figure 22.3: JavaScript Calling Method

The code string that is the first parameter of the exec_script method is the text of a JavaScript method call and its parameters. If the Hypercosm applet is not run from inside of a web page, then this command will have no effect.

LESSON 22:
EXTERNAL DATA FILES

LESSON OBJECTIVES:

- Understand the issues involved in working with external files
- Be able to download and process external text data files

LESSON CONTENTS:

- Why read data from external files?
- Text file resources
- Downloading text file resources
- Getting text data from a text file resource

COPYRIGHT © 2006

WHY READ DATA FROM EXTERNAL FILES

Any data that we can read in from an external file, we could also just compile into an applet. However, reading the data from an external text file allows the data to be changed at a later date without requiring the applet to be recompiled. This allows content to be created in a way that is much more flexible and maintainable.

TEXT FILE RESOURCES

We have seen how applets can have a variety of external data resources associated with them, such as graphics and sound files. Text files are handled in a similar way. To use an external data file in Hypercosm, you must first create a text file object that references an external text file. The definition of the text file objects is stored in the file "text_file_resources.ores", which is located in the directory "Hypercosm Studio/Includes/Utilities/Networking/Data Resources/".

```
text_resource type <resource name> named <file name>;
```

Figure 23.1: Creating a new text file resource

DOWNLOADING TEXT FILE RESOURCES

One of the issues in dealing with external data file resources is that they are not guaranteed to be ready to use when the applet starts. These resources are all downloaded over the internet independently of the applet file and if they are large, then they can take a considerable time to download before they can be used. When you write your applet code to use these resources, you must be mindful of this fact. There are two ways that you can deal with the situation. These two ways are refered to as "blocking" and "non-blocking".

BLOCKING DOWNLOADING

The easiest way to deal with the resource downloading issue is simply to have the applet stop and wait until the text file resource is downloaded. This works fine for small

resources because the time that it takes to download them is small and the user will most likely not notice the tiny delay in the applet startup. To block the applet from running, simply call the text resource's method "finish_loading" to wait until the resource has finished downloading. Until this call returns, the applet will remain stopped.

```
subject data_resource
extends
        downloadable object
does
        …
        native verb finish_loading;
end;
```

Figure 23.2: Blocking Resource Download Method

```
text_file_resource type text_file_resource named "data.txt";

text_file_resource finish_loading;
```

Figure 23.3: Using Blocking Resource Downloading

NON-BLOCKING DOWNLOADING

The second way of dealing with the downloading issue is to query the resource to see if it is ready and if not, temporarily skip the code that relies upon it. This technique is nice because it enables the applet to continue to run while the file is downloading. To query the applet, use the download_status function.

```
enum download_status is downloading, ready, ready_bad_fingerprint, failed;

subject data_resource
extends
        downloadable object
does
        …
        native download_status type question download_status;
end;
```

Figure 23.4: Non-Blocking Resource Download Method

```
text_file_resource type text_file_resource named "data.txt";

verb update is
        if text_file_resource download_status is ready then
                // process the data
        else
                // do nothing
        end;
end;
```

Figure 23.5: Using Non-Blocking Resource Downloading

GETTING TEXT DATA FROM A TEXT FILE RESOURCE

Once the text file resource has been downloaded, we can then get the text data from it. To do this, use the method "get_lines" to retrieve an array of strings that represent the lines of text in the text file.

```
subject text_file_resource
extends
        data_resource
does
        …
        strings type question get_lines;
end;
```

Figure 23.6: Blocking Resource Download Method

```
text_file_resource type text_file_resource named "data.txt";
strings type text;

text_file_resource finish_loading;                  // load text file resource
text is text_file_resource get_lines;               // retrieve contents of text file resource
for each string type string in text do              // write out contents of text file resource
        write string;
end;
```

Figure 23.7: Processing Text Data From a Text File Resource

NOTES:

There are a set of "parsing" utilities to help you with the task of parsing text from data files and other sources. These parsing utilities are located in the directory "Hypercosm Studio/Includes/Utilities/Parsing/".