TEACHER TRAINING MANUAL

MULTIMEDIA APPLICATIONS FOR EDUCATION

Index

Part 3E: MULTIMEDIA

PART THREE/E

MULTIMEDIA

Chapter Five
How to Conceive and Design an Interface

University of Patras (GR)

1. How to conceive an Interface

1.1 Introduction

A video game has many different forms; from simple flash games you play online to the newest hit for one of the big consoles. Some video games are simple while others require a great deal of practice to get the hang of.
A lot of people, however, don't really know what goes on behind the screen. You can see the moving symbols and colors, the sounds, and know that they react when you press buttons, but what actually is happening back there?
A video game is software that runs on a computer or video game console that uses a display and has a method with which a player can control the game.
While that definition is nice and simple, in order to efficiently understand video game design, you really need to know the mechanics behind it all (programming, graphic design, sound design, music composition).

1.1.1 History

The first video games were designed in the 1960s and 1970s by programmers for whom creating games was a hobby, since there was no way to sell them or earn money from creating games (the games required large mainframe computers to play). Some were designed by electrical engineers as exhibits for visitors to computer labs (OXO, Tennis for Two), others by college students who wrote games for their friends to play (Spacewar!, Star Trek, Dungeon).
Some of the games designed during this era, such as Zork, Baseball, Air Warrior and Adventure later made the transition with their game designers into the early video game industry.
Early in the history of video games, game designers were often the lead programmer or the only programmer for a game, and this remained true as the video game industry dawned in the 1970s. This person also sometimes comprised the entire art team. This is the case of such noted designers as Sid Meier, Chris Sawyer and Will Wright. A notable exception to this policy was Coleco, which from its very start separated the function of design and programming.
Today, it is rare to find a video or computer game where the principal programmer is also the principal designer, except in the case of casual games, such as Tetris or Bejeweled. With very complex games, such as MMORPGs, or a big budget action or sports title, designers may number in the dozens. In these cases, there are generally one or two principal designers and many junior designers who specify subsets or subsystems of the game. In larger companies like Electronic Arts, each aspect of the game (control, level design or vehicles) may have a separate producer, lead designer and several general designers.

1.2 The art of creating the videogame environment

Game design is primarily an artistic process, but it is also a technical process. The game designer pursues grand artistic goals even as she grinds through mountains of code. During the process of developing the game, he inhabits two very different worlds, the artistic world and the technical world. How does one manage the integration of such dissimilar worlds? In short, how does one go about the process of designing a computer game?
In the first place, game design is far too complex an activity to be reducible to a formal procedure. Furthermore, the game designer's personality should dictate the working habits he uses. Even more important, the whole concept of formal reliance on procedures is inimical to the creative imperative of game design.

1.2.1 Choose a goal and a topic

This vitally important step seems obvious, yet is ignored time and time again by game designers who set out with no clear intent. Game designers will admit under close examination that they sought to produce a "fun" game, or an "exciting" game, but that is more often than not the extent of their thinking on goals.
A game must have a clearly defined goal. This goal must be expressed in terms of the effect that it will have on the player. It is not enough to declare that a game will be enjoyable, fun, exciting, or good; the goal must establish the fantasies that the game will support and the types of emotions it will engender in its audience. Since many games are in some way educational, the goal should in such cases establish what the player will learn. It is entirely appropriate for the game designer to ask how the game will edify its audience.
The importance of a goal does not become obvious until later in the game design cycle. The crucial problems in game development with microcomputers are always problems of trade-offs. Everything that the game designer wants to do with her game costs memory, and memory is always in short supply with microcomputers. Thus, the designer must make trade-offs. Some game features can be included, and some must be rejected.
How do you select a proper goal? There is no objective answer to this question; the selection of a goal is the most undeniably subjective process in the art of computer game design. This is your opportunity to express yourself; choose a goal in which you believe, a goal that expresses your sense of aesthetic, your world view. Honesty is an essential in this enterprise; if you select a goal to satisfy your audience but not your own taste, you will surely produce an anemic game. It matters not what your goal is, so long as it is congruent with your own interests, beliefs, and passions. If you are true to yourself in selecting your goal, your game can be executed with an intensity that others will find compelling, whatever the nature of the game. If you are false to yourself, your game will necessarily be second-hand, me-too.
There are situations in which it is not quite possible to attain the purity of this artistic ideal. The realities of the marketplace demand that such games be written, and it is better that they be written by mature professionals than by simpering fools. Such emotionally indirect games, however, will never have the psychological impact, the artistic power, of games coming straight from the heart.
Once you have settled on your goal, you must select a topic. The topic is the means of expressing the goal, the environment in which the game will be played. It is the concrete collection of conditions and events through which the abstract goal will be communicated. For example, the goal of STAR RAIDERS apparently concerns the violent resolution of anger through skillful planning and dexterity. The topic is combat in space. The goal of EASTERN FRONT 1941 concerns the nature of modern war, and especially the difference between firepower and effectiveness. The topic is the war between Russia and Germany.
Selecting a good topic can be time-consuming, for each potential topic must be carefully examined for its ability to successfully realize the goals of the game. Many topics carry with them some excess emotional baggage that may interfere with the goals of the game.

1.2.2 Videogame Environment

You now have a clear idea of the game's ideals but you know nothing of its form. You are now ready to begin the concrete design phase. Your primary goal in the design phase is to create the outlines of three interdependent structures: the I/O structure, the game structure, and the program structure. The I/O structure is the system that communicates information between the computer and the player. The game structure is the internal architecture of causal relationships that define the obstacles the player must overcome in the course of the game. The program structure is the organization of mainline code, subroutines, interrupts, and data that make up the entire program. All three structures must be created simultaneously, for they must work in concert. Decisions primarily relating to one structure must be checked for their impacts on the other structures.

1.2.3 Evaluation of the Videogame Environment

After you create the three structures: the I/O structure, the game structure, and the program structure, you have to evaluate them. You are satisfied that all three structures will work and that they are compatible with each other. The next stop in the design phase is to evaluate the overall design for the most common design flaws that plague games. The first and most important question is: does this design satisfy my design goals? Does it do what I want it to do? Will the player really experience what I want him to experience? If you are satisfied that the design does pass this crucial test, proceed to the next test.

Examine the stability of the game structure. Remember that a game is a dynamic process. Are there any circumstances in which the game could get out of control? For example, if the game has money in it, could a situation arise in which the player finds himself the owner of ridiculously large amounts of money? In short, does the game structure guarantee reasonable upper and lower bounds on all values? If not, re-examine the game structure carefully with an eye to structural changes that will right the situation. If you have no other options, you may be obliged to put them in by brute force (e.g., "IF MONEY > 10000 THEN MONEY 10000")

Now probe the design for unanticipated shortcuts to victory. A player who can find a way to guarantee victory with little effort on his part will not derive the full benefit of your game. Insure that all unintended shortcuts are blocked so that the player must experience those processes that you want him to experience. Any blocks you place must be unobtrusive and reasonable. The player must never notice that he is being shepherded down the primrose path. An example of obtrusive blocking comes from the game WAR IN THE EAST (trademark of Simulations Publications, Inc). This wargame deals with the Eastern Front in World War 11. The Germans blitzed deep into Russia but their advance ground to a halt before Moscow. To simulate this the designers gave the Germans an overwhelming superiority but also gave them a supply noose whose length was carefully calculated to insure that the Germans would be jerked to a dead halt just outside Moscow. The effect was correct, but the means of achieving it were too obvious, too obtrusive.

The last and most crucial decision is the decision to abort the game or proceed. It should be made now, before you commit to programming the game. Do not hesitate to abort the game now; even if you abort now you will still have I earned a great deal and can say that the effort was worthwhile. A decision to give up at a later stage will entail a real loss, so give this option careful consideration now while you can still do it without major loss. Abort if the game no longer excites you. Abort if you have doubts about its likelihood of success. Abort if you are unsure that you can successfully implement it. I have in my files nearly a hundred game ideas; of these, I have explored at length some 30 to 40. Of these, all but eight were aborted in the design stage.

2. The interface of 2D videogames

2D interfaces are flat designs. This type of design is not intended to offer exploration. 2D interfaces are useful for presenting static information.
The most common use for a 2D interface is printed material such as books, magazines, maps or advertisements, interface of 2D videogames.

2.1 Define a Look

Defining the look and feel of an interface is the fun part of the design process [1]. Artist and designers with a passion for creativity look forward to this stage of development. The early concept stage is the fun part of the process.
When working on the look and feel of a game, have fun and take the opportunity to be creative. This is a great place to experiment and to come up with something totally unique. The look of the design is what the end users will remember. If the functionality of an interface is good, the user won't even notice it. If you don't enjoy designing the look of a game, you may not be cut out for interface design.

2.1.1 Create a Mock-Up

The best way to define a distinctive look for your game is to create sample art, or a mock-up of the interface. The goal of creating this sample art is not to have a final product but to define and visualize the look and feel of the entire interface. Don't worry about having the right options listed. It is more important to show what your buttons will look like than it is to get the right button. By creating art that looks like a real interface, you make it easy for anyone who needs to review and approve your design. It does not require a lot of imagination or guesswork on the part of the producer or art director to get the idea if they can simply see it.
A mock-up can guide your design throughout the process. Once your mock-up has been created, reviewed, and approved, a standard has been set. The rest of the interface can be designed to fit in with the look and feel of the sample art. The entire interface should look and feel just like this sample art. It will be much faster to design the rest of the interface once you have set the look and feel. Much less experimentation is needed once you've found the style for your interface. Figure 3.1 shows an example of a mock-up.

Figure 3.1: The mock-up of the title screen defines the colors and style of the entire interface.

A mock-up of a single screen of the interface and just a few more pieces of art, such as some important buttons from other screens, is all you need to define a look. Figure 3.2 shows a couple of these extra elements that you might want to include in the mock-up

[1] Game Interface Design, by Brent Fox.

phase. You don't need every detail to establish your interface style. Often, the best screen to mock-up is the title screen. Legal screens, company logo screens, and even the opening cinematic sequence may appear before this title screen in the final game, but typically the title screen is the first in which options appear for the user. There are some games that have a separate title screen from the main menu, but it is usually the first screen with active buttons, and it will often contain the game logo, as well. Because it contains so many important elements, the title screen is ideal to use as a mock-up screen.

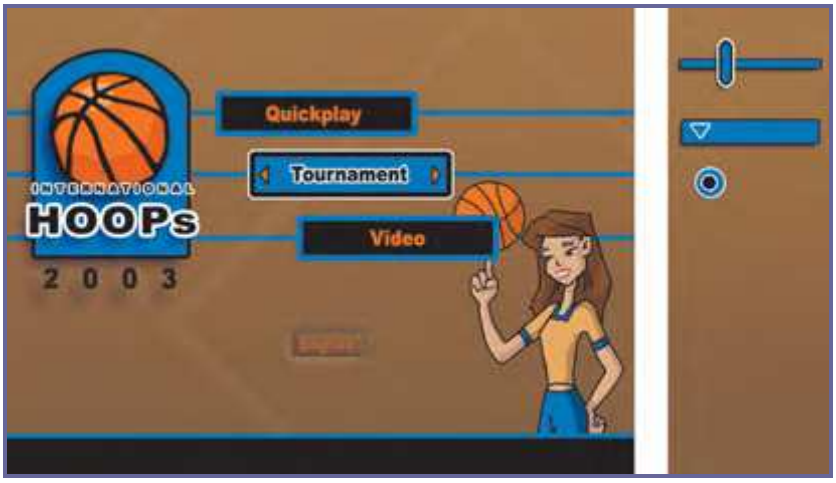

Figure 3.2: Creating a few more interface elements helps to better establish the look of the interface.

2.1.2 Working with Logos

Working with publishers and their game logos can be tricky business. The game publisher often provides the logo, and it is important to get this logo as early as you can. Too often, the publisher waits until near the end of the project to even decide on the name of the game, much less the design logo.

If the publisher is dragging its feet on coming up with a logo, create a temporary logo that captures the feel that will be used in the final logo. This may not be easy to guess, but it is better to have some reference, however flawed, than to have no reference.

Try to establish the look of the logo early, even if the name must change later. Communicate with the publisher and make sure they agree with the direction you're taking. It is always a pain when, say, the game has a blackand - orange interface and the publisher brings you a green and purple logo at the end of the project and asks you to change the entire interface to match the cool, new logo. Even if the new logo is cool looking, if it doesn't mach the rest of the interface, it can mean you'll have to spend weeks reworking the art. Figure 3.3 shows how a logo can be out of place if the interface was not designed around the look of the logo.

Figure 3.3: The colors in this logo don't go well with the rest of the interface.

2.1.3 Define a Color Scheme

Color is a very important part of an interface. What color is your game? This is a good question to answer early. Anyone who looks at your interface should be able to see at a glance the color scheme of the entire game. Keeping the colors consistent throughout the game creates a unified look. Everything from the box cover to the in-game interface should reflect this color scheme and help define your game. Too many dramatic changes in color from screen to screen will make the game feel inconsistent. When creating a color chart, make sure it feels like you want your game to feel. If you are working on a game for young children, for example, then bright, saturated colors may be appropriate. The colors would probably be very different for a game based on a horror story. In such a game, the colors should look like they belong in a horror movie—a lot of black with orange or green accents may be a good choice for such a scary game. Take a look at Figure 3.4 and compare the two color charts to see how a feel can be created using only color.

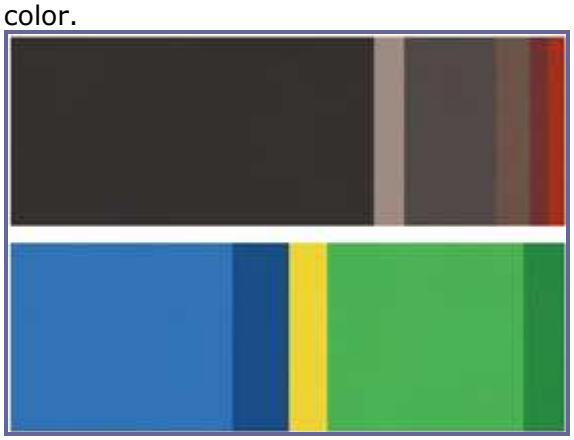

Figure 3.4: Just by looking at these two different color schemes, you can tell whatkind of games they might be used for.

The subject matter of the game can often direct the color choice. If you are working on a game that takes place in a jungle, green is probably the wise color choice. If you are working on a game with demons and gargoyles, then red and black may be a logical choice. Your colors should feel like they fit with the subject matter.

Images sometimes actually get in the way of making a color choice. If you have an illustration or photo of a cool-looking red car, you may be influenced to choose red as one of the colors for your interface, even if red isn't the best color choice. It is better to make the color choice first and then adjust the image to match your color scheme. This way, you have made the color choice independent of the colors in an image.

A good way to separate the color choice from all of the other decisions is to make a color chart. Create a file that is made up of the colors you will use in the interface. This color chart should not only contain the colors you will use in the design, but it should also have the correct proportions of each of the colors. It should roughly represent the amount of each color you will use in the actual interface. If an accent color is used in the design, it should only take up a small amount of space on the color chart. This way, the colors in your chart will feel like the final interface. Make sure to refer to this chart when working on the interface, so that you don't lose the color balance you've established. Take a look at the color chart in Figure 3.5 and see how the yellow color is much smaller than the green tones. Now look at the final interface screen in Figure 3.6 and see how the color is balanced similarly to the color chart.

Figure 3.5 This color chart establishes the colors of an interface.

Figure 3.6 Compare the color chart with this final interface.

2.2 Express Yourself in the Design

Go for it! Make your design unique. This is your chance to really express your creativity. The best interface designs push the feel of the game. If you decide to design an interface with a retro, 1950s-America feel, then make sure that all of the elements fit together. Don't just design a standard interface that has a few elements that fit the 1950s theme—make it really feel like the 1950s. Look at clothing styles, colors, cars, and kitchen appliances that were used in the 1950s. Choose some of the elements that capture the feeling you are looking for. A button on a radio or the grill of a car may inspire the shape, color, and design of your interface. If the game takes place in ancient Japan and you decide to design the interface with a classic Japanese feel, then go all the way. Look at ancient Japanese art and calligraphy. Choose a font that looks Asian or like calligraphy. Use colors, plants, cloth patterns, or anything that imparts the feel you're striving for.

2.3 Research and Inspiration

Coming up with ideas for your design is not always easy. It's not uncommon to hit a creative roadblock when designing an interface.When you feel like you can't come up with any good ideas, there are a couple of techniques that you can use to help inspire yourself. Don't let a slump hold you back for long!

2.3.1 Make Lists

A common and very effective brainstorming method is to create lists. Sit down and just start writing. Write down good ideas and even ideas that may not seem so good at the moment—just let them flow. Create several lists. List objects associated with the game. List emotions associated with the game. List actions associated with the game. Create as many categories as you can. Combine different words and phrases from different lists and see what you can come up with. You may come up with some unexpected solutions using this technique. Figure 3.7 shows an example of the beginnings of a brainstorming list.

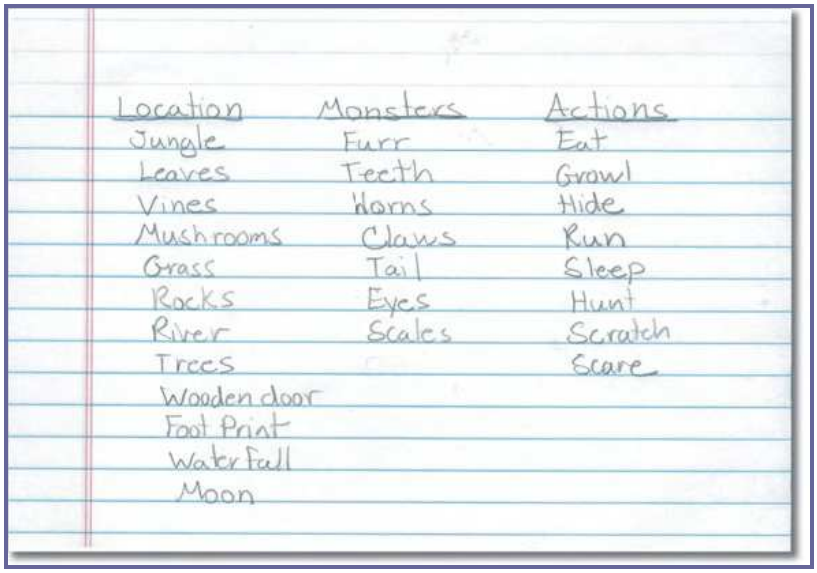

Figure 3.7: These are lists that were created for a children's game about a family of monsters that live in the jungle.

2.3.2 Search for Images

Another great creativity-inspiring technique is searching the Internet for cool, interesting, or thoughtprovoking images. You can go on a virtual field trip anywhere in world and see what things—buildings, clothing, flora, fauna, and so on— look like. If you are unfamiliar with the subject, you can quickly find visuals to help you approximate the look you're going for in your design. When creating an interface for a Formula One racing game, for example, you could search the Internet for photographs of the cars, the crowds, the tracks, and the drivers. You may find images of elements you wouldn't have thought of without looking at photos. Skid marks on the track, dented railing along the track, and helmets worn by the drivers may all provide inspiration and direction—and you may not have thought of them if you hadn't searched for images online.

An amazingly large amount of information and photos can be found on the Internet. Be very careful not to violate any copyright laws, though. Use the photos and images you find for inspiration, but don't actually use any photos if you don't have the copyright on them. If you really need a specific photo, you may be able to purchase the rights to use it. Stock photography vendors will be happy to help you out.

You can find inspiration in other places, as well. Art galleries, libraries, and the theater can all be places where you can find inspiration. Constantly keep your eyes open. On my drive into the office, I pass several old factories with rusted metal walls, steel, and rivets.

I think of how many great images and textures that can be found in these old, beat-up buildings. I often carry a digital camera so that I can stop and take a picture of anything visually arresting I come across during the day.
Another place I find a lot of inspiration is at the movies. There are so many visually stunning movies. For example, if I am working on a game that takes place in ancient Egypt, I will rent (or go see, if there is anything out) a great movie that shows architecture and art from Egypt. Animated movies are also great for inspiration. I have watched many movies with a sketchbook in my hand, ready to capture any inspiring image I see.
Check out your competition. Find out what they came up with when confronted with a similar design challenge. See what you're competing against and learn if any unique and interesting design solutions have appeared in other games. Understand what users have come to expect of games in the genre you're working in. Avoid any urge to copy the design of other games, though. It's easy to make a game that looks only slightly different to a competing game. Playing a game like this won't be very enjoyable or impressive for users. Make your design original. Don't sell your own abilities short—even if a competitor has a great interface design, their design doesn't represent the only great solution available.

2.3.3 Thumbnails

Thumbnails are very small sketches. They are often only an inch or two wide. They are used to quickly run through a bunch of concepts. These little sketches can be very useful when designing an interface. When I skip thumbnails and go straight to working on a full size image, most of the time I end up getting stuck and having to go back and create the thumbnails after all.
It is easy to get too excited about an interface and either skip or spend too little time on the thumbnail sketch stage. Be patient, and make a lot of thumbnails. Thumbnails are easy and fast to make, and they can allow you to try out literally hundreds of ideas quickly. If you dig right in and start creating full-size color layouts before you make thumbnails, you'll only be able to try a limited number of approaches. Take advantage of the ease of creating thumbnails and create a lot of them.

2.3.4 Work Quickly

Keep your thumbnail designs small and simple. As the name implies, thumbnails are typically small, and they do not contain much detail; they are simply meant to help you arrange the basic layout. If they are drawn too large, you may be tempted to add too much unnecessary detail. Such unnecessary detail will slow you down and can distract you from finishing the basic layout.
Make your thumbnails quickly and keep them rough. They should be used for internal direction and should not be shown to a publisher until they have been cleaned up. Spending extra time creating tight thumbnails can be a waste. Even without the details, it is amazing how much information you can convey in a small thumbnail. The best way to make thumbnails is the old fashioned way—with a pencil and paper. It is hard to match the speed of using a pencil for thumbnail sketches. Speed is the key with thumbnail sketches. You want to try a lot of ideas quickly.
Some artists struggle to maintain a small scale and to get the screen proportions correct. The best way to solve this problem is to print out a page of small, blank boxes that are the correct size and proportions. These boxes can then be used as borders for hand-drawn thumbnails.

2.3.5 Push for Variation

It is a good idea to push yourself to create more thumbnails than you're initially inclined to make. You will often be more creative the further you go into the thumbnail process. At first, you may tend to create thumbnails that look similar to other interfaces you have created. Once you have run through your standard set of ideas, you will be forced to come up with more creative ideas. Don't stop when it starts to become difficult to come up with another idea. This is often the point when new ideas appear. When you get

stuck, you can create variations on each design. It is also a good idea to create many completely new layouts.
Although it's good practice to create a lot of thumbnails, it's often a mistake to present hundreds of ideas to the publisher for approval. The publisher may legally own all of the art created in association with the game, but they seldom require that you show them every scribble and sketch. Not only does it take longer for the publisher to sort through a large number of thumbnails, but inevitably the publisher will choose the one you like the least. Not everyone has the ability to envision a finished product from a thumbnail. Don't run the risk that a publisher can't see past your pencil scribbling to the magic beneath. As I said before, most thumbnails should only be used internally; if a publisher requests to see thumbnails, it is a good idea to clean up and present one or two sketches. Choose the thumbnails that you have already determined to be the best solutions.

2.4 Creativity versus Standards

Creativity is essential, but make sure that you use it in the right place. You must balance new and original ideas with standard approaches. Gamers have come to expect certain standards, and in many cases, it is better if they don't have to think too much about a new approach. Just because you think it will be cool to, say, have the "highlighted" button grow dark instead of light up does not mean it is necessarily a good idea; it may confuse the user and take him longer to understand which button is selected. This does not mean that darkening the selected button will never work. You just need to consider what the user is expecting to see and understand that if your menu does something different, then it may make it harder for the user to navigate.

2.5 Using Photographs

Photographs can be very useful and cool-looking in your interface, but they must be used correctly or they will hurt your design. In some cases, using photos may be the very best solution or even a requirement. If you are making a game that uses the name of the latest sports star, you may need to include a photo of the athlete on the box and in the interface. A game with a movie license may also require a photo of the star. In many other instances, photos can be a crutch and can make a very bad or uninteresting interface. Photos should only be used when they are the logical solution, and not just because it's easier to use a photo than an illustration or to create your own background. It is very obvious when an interface designer uses a bunch of stock photographs that have not been properly touched up just to save time. It just looks bad. I have a personal preference for using illustrations over photos. I usually avoid photos in an interface. Other than when you need to show a likeness of a famous person, I suggest always using an illustration. It will take more time and require more skill, but I think that a quality illustration has the potential to look much better than a photo. You will find that only a small percentage of the big, triple-A games use photos in the interface. Photos rarely fit well with the art in the game. If your budget is tight, of course, you may need to use photos instead of illustrations. You need to be aware that this may be a weak point in your interface if you don't take the extra time to use the photos well. Making your own collage of several photos, using filters and effects, and making other adjustments to these photos can really help. Do your best to choose your photos wisely. I have seen some instances wherein an interface designer used photos and it resulted in a quality interface, but these instances are rare.
If you use photographs in your design, make sure they are of good quality. A digital camera can be incredibly useful when making games. The problem that comes along with using digital cameras, however, is that they are so easy to use that everyone thinks that he or she is a great photographer and that there is no need to spend money on a professional photographer or purchase stock photography. Many designers think, "I can take a picture of my own football and get it just the way I want it." But in reality, the shot they end up with is not nearly as good as a professional photographer could do.

Photos you have taken yourself are great for reference, but they must be high quality if you want to use them in your interface. Figure 3.8 is a photo that I took that is a little washed out. It is not a high-quality photo.

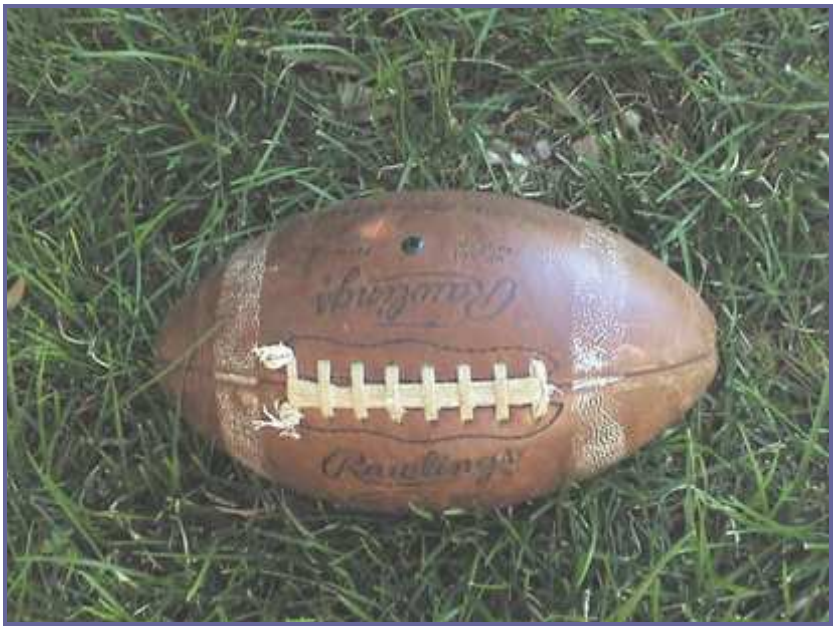

Figure 3.8: Digital photos that are not taken by a professional can hurt an interface design.

Don't be afraid to take digital photos, just understand your limits. There are many, many uses for photos. For example, they can serve as great references, as they capture details that you might not be able to remember without them. They can also provide a great start for textures. They can be used the same way the Internet can be used for research. Photos that are to be used in your game can be touched up and edited. It's hard to fix a bad photo, but it's easy to improve a good photo. Whether you took the photos or they were taken by a professional photographer, there are many techniques that can be used to make the photo more interesting. Simple adjustments include changing the image levels and saturation. You can also try techniques like colorizing the photos or adding other filters. If you plan on using photos often, learn as many techniques as you can to get the most out of your photos. Figure 3.9 shows a photo that has been touched up using several different methods.

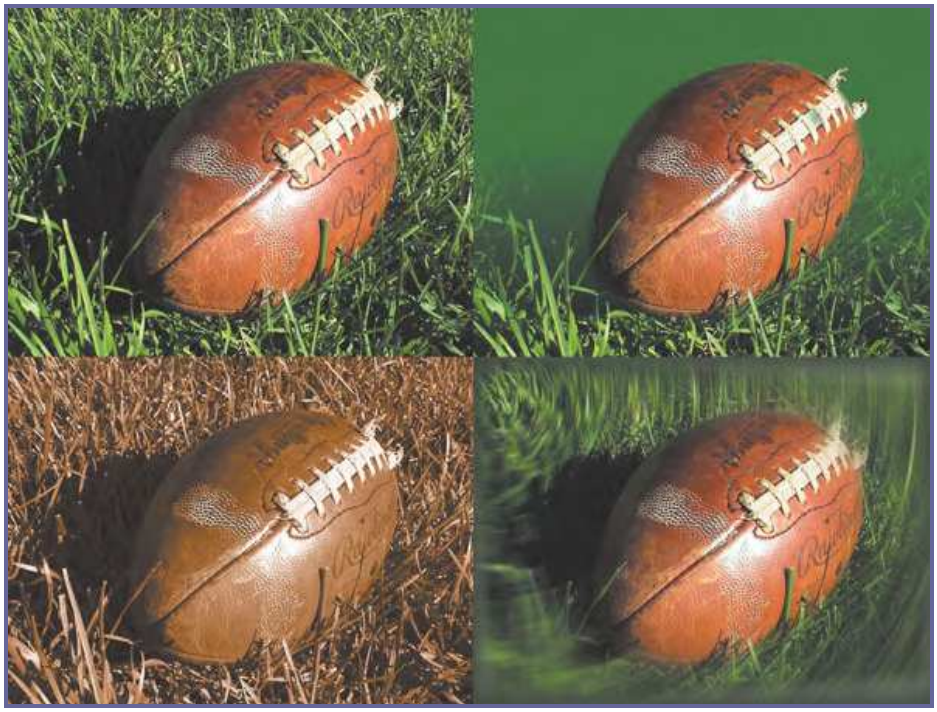

Figure 3.9: An average photo has been adjusted in several different way to help

enhance the photo and make it more suitable for use in an interface.

2.6 Illustrations

In place of a photograph, you might want to consider using an illustration. This approach can really improve the look of an interface. The subject matter of your game can determine which to use. For example, while a sports game may be a great place to use a photograph of all the players, an illustration may be much better solution for a fantasy game. The style of illustrations used in an interface can help define the look and feel of a game. Are the illustrations stylized or realistic, detailed or simple, colorful or desaturated? As with photographs, poor illustrations will hurt a design, but great illustrations can improve an interface significantly.
If you are confident in your skills as an illustrator, then you should do your own illustrations. If you can't produce top-notch illustrations in the style that would best fit your interface, get an illustrator with the style you need for your game. Just because your illustration style does not fit the game does not make you a bad illustrator. Don't force your illustration or illustration style into a design if it doesn't work just because you want the design to be "all yours." Figure 3.10 shows a sample of an illustration that would be hard to beat with a photograph.

Figure 3.10 Using an illustration instead of a photo here allowed for brighter colors.
This image would have been difficult to photograph.

3. The interface of 3D videogames

In the real world around us, we perceive objects to have measurements in three directions, or dimensions. Typically we say they have height, width, and depth.When we want to represent an object on a computer screen, we need to account for the fact that the person viewing the object is limited to perceiving only two actual dimensions: height, from the top toward the bottom of the screen, and width, across the screen from left to right.
Therefore, it's necessary to simulate the third dimension, depth "into" the screen. We call this on-screen three-dimensional (3D) simulation of a real (or imagined) object a 3D model [2]. In order to make the model more visually realistic, we add visual characteristics, such as shading, shadows, and textures. The entire process of calculating the appearance of the 3D model—converting it to an entity that can be drawn on a two-dimensional (2D) screen and then actually displaying the resulting image—is called *rendering*.

[2] 3D Game Programming All in One, by Kenneth C. Finney.

3.1 Introduction

3.1.1 Coordinate Systems

When we refer to the dimensional measurement of an object, we use number groups called *coordinates* to mark each *vertex* (corner) of the object. We commonly use the variable names X, Y, and Z to represent each of the three dimensions in each coordinate group, or triplet. There are different ways to organize the meaning of the coordinates, known as coordinate systems. We have to decide which of our variables will represent which dimension—height, width, or depth—and in what order we intend to reference them. Then we need to decide where the zero point is for these dimensions and what it means in relation to our object. Once we have done all that, we will have defined our *coordinate system*.

When we think about 3D objects, each of the directions is represented by an axis, the infinitely long line of a dimension that passes through the zero point. Width or left-right is usually the X-axis, height or up-down is usually the Y-axis, and depth or near-far is usually the Z-axis. Using these constructs, we have ourselves a nice tidy little XYZ-*axis system*, as shown in Figure 3.11.

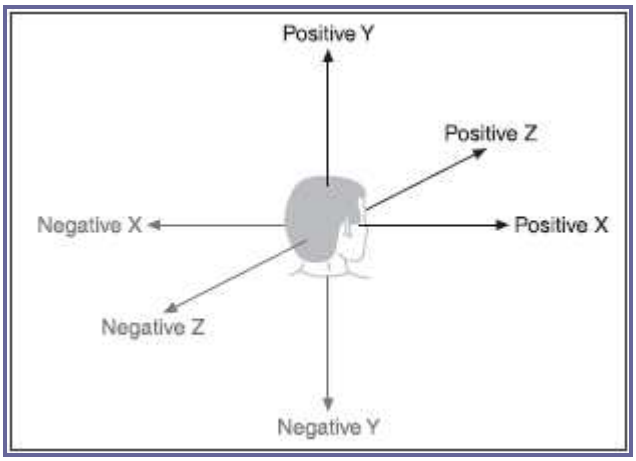

Figure 3.11: XYZ – Axis System

Now, when we consider a single object in isolation, the 3D space it occupies is called object space. The point in *object space* where X, Y, and Z are all 0 is normally the geometric center of an object. The *geometric center* of an object is usually inside the object. If positive X values are to the right, positive Y values are up, and positive Z values are away from you, then as you can see in Figure 3.12, the coordinate system is called *left-handed*. The Torque Game Engine uses a slightly different coordinate system, a *right-handed* one. In this system, with Y and Z oriented the same as we saw in the left-handed system, X is positive in the opposite direction. In what some people call *Computer Graphics Aerobics*, we can use the thumb, index finger, and middle finger of our hands to easily figure out the handedness of the system we are using (see Figure 3.13). Just remember that using this technique, the thumb is always the Y-axis, the index finger is the Z-axis, and the middle finger is the X-axis.

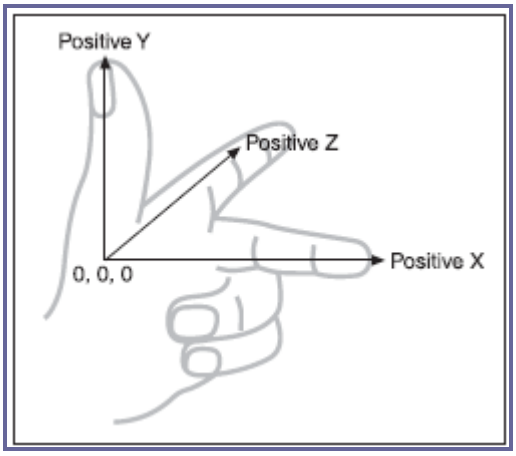

Figure 3.12: Left-handed coordinate system with vertical Y-axis.

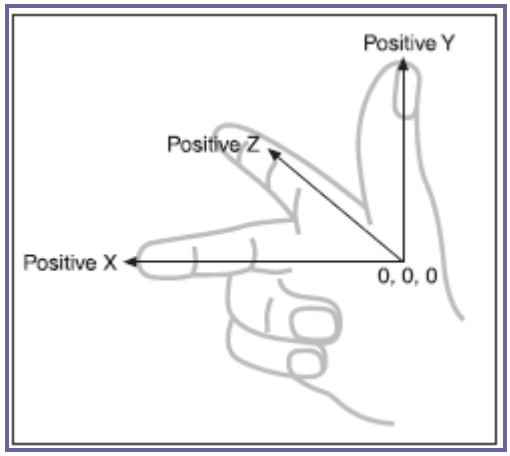

Figure 3.13: Right-handed coordinate system with vertical Y-axis.

With Torque, we also orient the system in a slightly different way: The Z-axis is updown, the X-axis is somewhat left-right, and the Y-axis is somewhat near-far (see Figure 3.14). Actually, *somewhat* means that we specify left and right in terms of looking down on a map from above, with north at the top of the map. Right and left (positive and negative X) are east and west, respectively, and it follows that positive Y refers to north and negative Y to south. Don't forget that positive Z would be up, and negative Z would be down. This is a right-handed system that orients the axes to align with the way we would look at the world using a map from above. By specifying that the zero point for all three axes is a specific location on the map, and by using the coordinate system with the orientation just described, we have defined our *world space*.

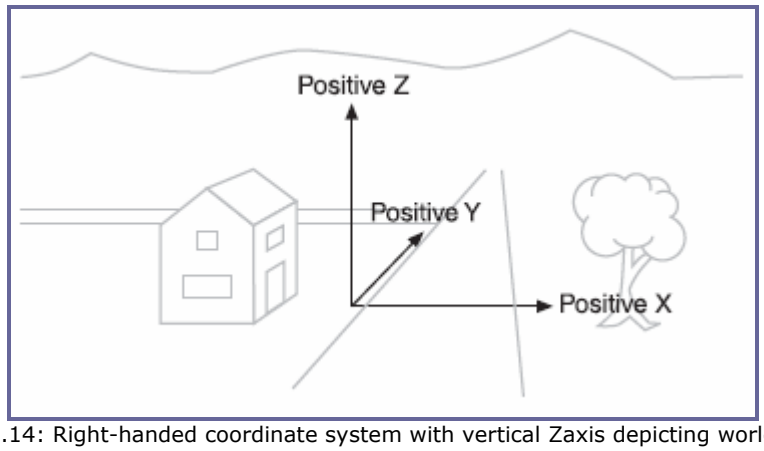

Figure 3.14: Right-handed coordinate system with vertical Zaxis depicting world space.

Now that we have a coordinate system, we can specify any location on an object or in a world using a coordinate triplet, such as (5,_3,_2) (see Figure 3.15). By convention, this would be interpreted as X=5, Y=_3, Z=_2. A 3D triplet is always specified in XYZ format. Take another peek at Figure 3.15. Notice anything? That's right—the Y-axis is vertical with the positive values above the 0, and the Z-axis positive side is toward us. It is still a right-handed coordinate system. The right-handed system with Y-up orientation is often used for modeling objects in isolation, and of course we call it *object space*, as described earlier.We are going to be working with this orientation and coordinate system for the next little while.

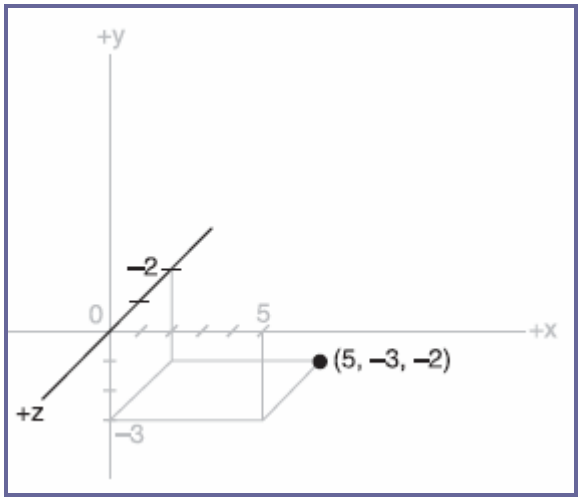

Figure 3.15 A point specified using an XYZ coordinate triplet.

3.1.2 3D Models

Let's take a closer look, by starting with a simple 3D shape, or *primitive*—the cube—as depicted in Figure 3.16.
The cube's dimensions are two units wide by two units deep by two units high, or 2_2_2. In this drawing, shown in object space, the geometric center is offset to a position outside the cube. I've done this in order to make it clearer what is happening in the drawing, despite my statement earlier that geometric centers are usually located inside an object. There are times when exceptions are not only possible, but necessary—as in this case.
Examining the drawing, we can see the object's shape and its dimensions quite clearly. The lower-left-front corner of the cube is located at the position where X=0, Y=1, and Z=_2. As an exercise, take some time to locate all of the other vertices (corners) of the cube, and note their coordinates.

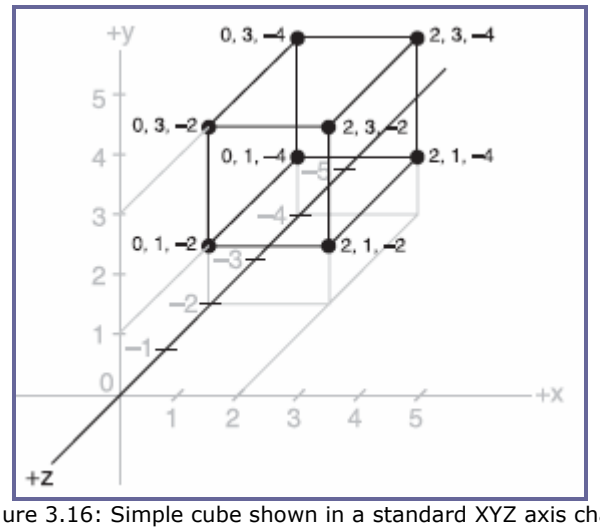

Figure 3.16: Simple cube shown in a standard XYZ axis chart

If you haven't already noticed on your own, there is more information in the drawing than actually needed. Can you see how we can plot the coordinates by using the guidelines to find the positions on the axes of the vertices? But we can also see the actual coordinates of the vertices drawn right in the chart. We don't need to do both. The axis lines with their index tick marks and values really clutter up the drawing, so it has become somewhat accepted in computer graphics to not bother with these indices. Instead we try to use the minimum amount of information necessary to completely depict the object.

We only really need to state whether the object is in object space or world space and indicate the raw coordinates of each vertex.We should also connect the vertices with lines that indicate the edges.
If you take a look at Figure 3.17 you will see how easy it is to extract the sense of the shape, compared to the drawing in Figure 3.16.We specify which space definition we are using by the small XYZ-axis notation. The color code indicates the axis name, and the axis lines are drawn only for the positive directions. Different modeling tools use different color codes, but in this book dark yellow (shown as light gray) is the X-axis, dark cyan (medium gray) is the Y-axis, and dark magenta (dark gray) is the Z-axis. It is also common practice to place the XYZ-axis key at the geometric center of the model.

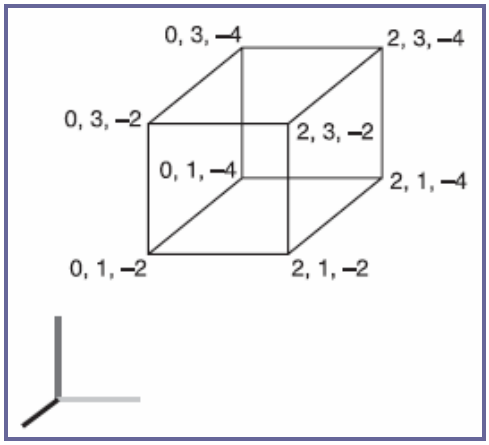

Figure 3.17: Simple cube with reduced XYZaxis key

Figure 3.18 shows our cube with the geometric center placed where it reasonably belongs when dealing with an object in object space.

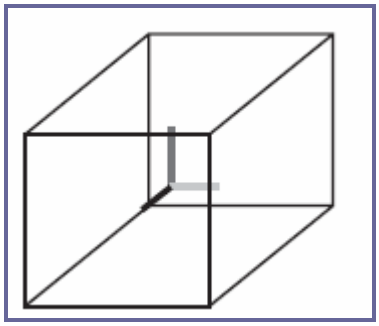
Figure 3.18: Simple cube with axis key at geometric center

Now take a look at Figure 3.19. It is obviously somewhat more complex than our simple cube, but you are now armed with everything you need to know in order to understand it. It is a screen shot of a four-view drawing from the popular shareware modeling tool MilkShape 3D, in which a 3D model of a soccer ball was created.
In the figure, the vertices are marked with red dots (which show as black in the picture), and the edges are marked with light gray lines. The axis keys are visible, although barely so in some views because they are obscured by the edge lines. Notice the grid lines that are used to help with aligning parts of the model. The three views with the gray background and grid lines are 2D construction views, while the fourth view, in the lowerright corner, is a 3D projection of the object. The upper-left view looks down from above, with the Y-axis in the vertical direction and the X-axis in the horizontal direction. The Zaxis in that view is not visible. The upper-right view is looking at the object from the front, with the Y-axis vertical and the Z-axis horizontal; there is no X-axis. The lower-left view shows the Z-axis vertically and the X-axis horizontally with no Y-axis. In the lowerright view, the axis key is quite evident, as its lines protrude from the model.

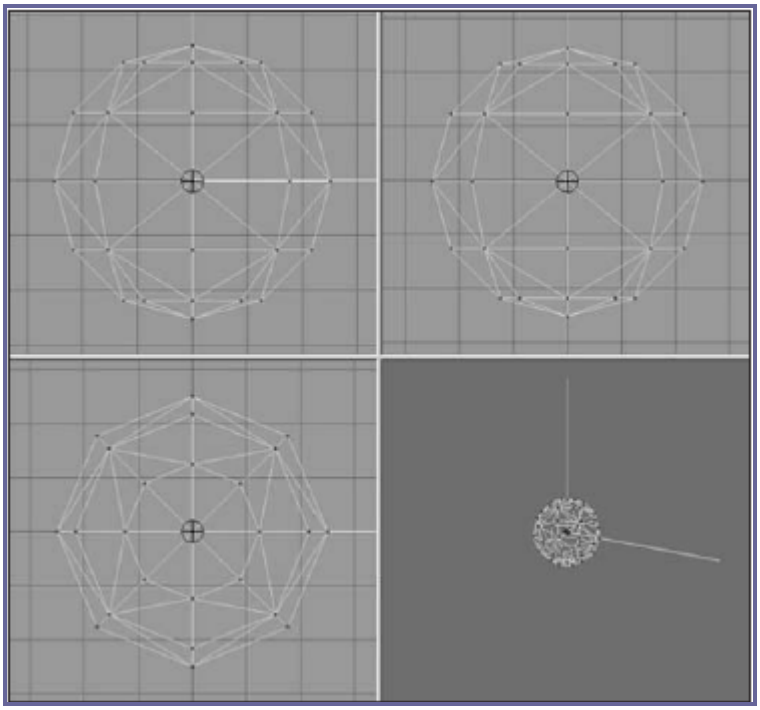

Figure 3.19: Screen shot of sphere model

3.1.3 3D Shapes

We've already encountered some of things that make up 3D models. Now it's time to round out that knowledge.
As we've seen, vertices define the shape of a 3D model. We connect the vertices with lines known as *edges*. If we connect three or more vertices with edges to create a closed figure, we've created a *polygon*. The simplest polygon is a triangle. In modern 3D accelerated graphics adapters, the hardware is designed to manipulate and display millions and millions of triangles in a second. Because of this capability in the adapters, we normally construct our models out of the simple triangle polygons instead of the more complex polygons, such as rectangles or pentagons (see Figure 3.20).

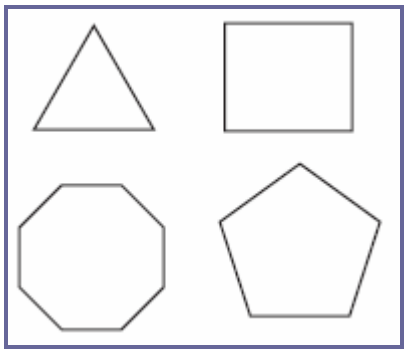

Figure 3.20: Polygons of varying complexity

By happy coincidence, triangles are more than up to the task of modeling complex 3D shapes. Any complex polygon can be decomposed into a collection of triangles, commonly called a *mesh* (see Figure 3.21). The area of the model is known as the surface. The polygonal surfaces are called facets—or at least that is the traditional name. These days, they are more commonly called faces. Sometimes a surface can only be viewed from one side, so when you are looking at it from its "invisible" side, it's called a hidden *surface*, or hidden face. A double-sided face can be viewed from either side. The edges of *hidden surfaces* are called *hidden lines*. With most models, there are faces on the back side of the model, facing away from us; called *backfaces* (see Figure 3.22). As mentioned, most of the time when we talk about faces in game development, we are talking about triangles, sometimes shortened to *tris*.

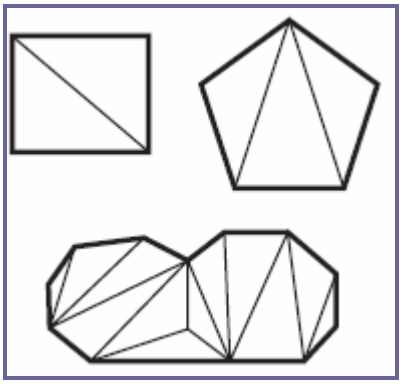

Figure 3.21: Polygons decomposed into triangle meshes

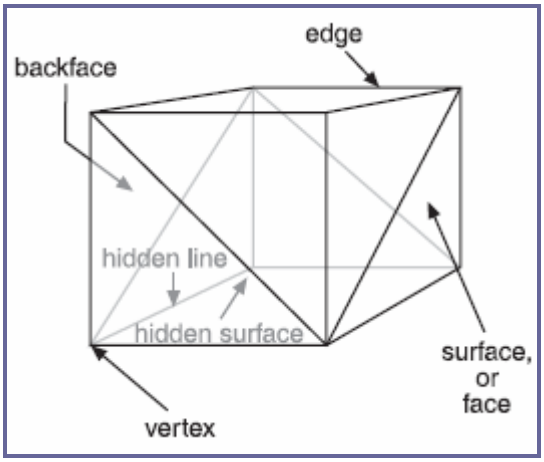

Figure 3.22: The parts of a 3D shape

3.2 Displaying 3D Models

After we have defined a model of a 3D object of interest, we may want to display a view of it. The models are created in object space, but to display them in the 3D world, we need to convert them to world space coordinates. This requires three conversion steps beyond the actual creation of the model in object space.

1. Convert to world space coordinates.
2. Convert to view coordinates.
3. Convert to screen coordinates.

Each of these conversions involves mathematical operations performed on the object's vertices.
The first step is accomplished by the process called *transformation*. Step 2 is what we call *3D rendering*. Step 3 describes what is known as *2D rendering*. First we will examine what the steps do for us, before getting into the gritty details.

3.2.1 Transformation

This first conversion, to world space coordinates, is necessary because we have to place our object somewhere! We call this conversion *transformation*. We will indicate where by applying transformations to the object: a scale operation (which controls the object's size), a *rotation* (which sets orientation), and a *translation* (which sets location).
World space transformations assume that the object starts with a transformation of (1.0,1.0,1.0) for scaling, (0,0,0) for rotation, and (0,0,0) for translation.
Every object in a 3D world can have its own 3D transformation values, often simply called *transforms* that will be applied when the world is being prepared for rendering.

3.2.2 Scaling

We scale objects based upon a triplet of scale factors where 1.0 indicates a scale of 1:1. The scale operation is written similarly to the XYZ coordinates that are used to denote the transformation, except that the scale operation shows how the size of the object has changed. Values greater than 1.0 indicate that the object will be made larger, and values less than 1.0 (but greater than 0) indicate that the object will shrink.
For example, 2.0 will double a given dimension, 0.5 will halve it, and a value of 1.0 means no change. Figure 3.23 shows a scale operation performed on a cube in object space. The original scale values are (1.0, 1.0, 1.0). After scaling, the cube is 1.6 times larger in all three dimensions, and the values are (1.6, 1.6, 1.6).

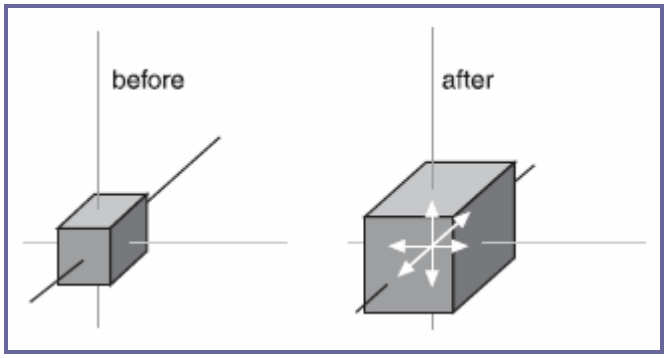

Figure 3.23 Scaling

3.2.3 Rotation

The rotation is written in the same way that XYZ coordinates are used to denote the transformation, except that the rotation shows how much the object is rotated around each of its three axes. In this book, rotations will be specified using a triplet of degrees as the unit of measure. In other contexts, radians might be the unit of measure used. There are also other methods of representing rotations that are used in more complex situations, but this is the way we'll do it in this book. Figure 3.24 depicts a cube being rotated by 30 degrees around the Y-axis in its object space.
It is important to realize that the order of the rotations applied to the object matters a great deal. The convention we will use is *the roll-pitch-yaw* method, adopted from the aviation community. When we rotate the object, we roll it around its longitudinal (Z) axis. Then we pitch it around the lateral (X) axis. Finally, we yaw it around the vertical (Y) axis. Rotations on the object are applied in object space.
If we apply the rotation in a different order, we can end up with a very different orientation, despite
having done the rotations using the same values.

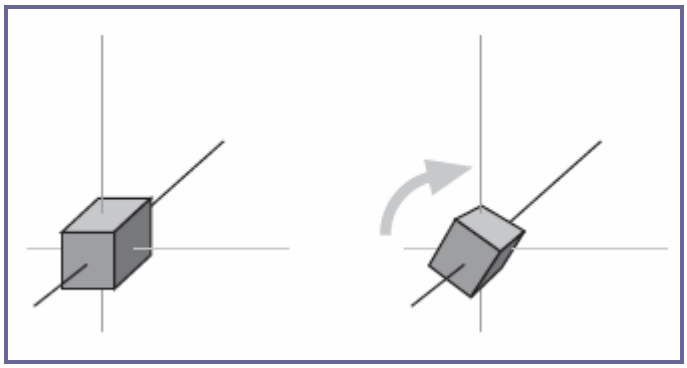
Figure 3.24 Rotation

3.2.4 Translation

Translation is the simplest of the transformations and the first that is applied to the object when transforming from object space to world space. Figure 3.25 shows a translation operation performed on an object. Note that the vertical axis is dark gray. As I said earlier, in this book, dark gray represents the Z-axis. Try to figure out what coordinate system we are using here. I'll tell you later in the chapter. To translate an object, we apply a vector to its position coordinates. Vectors can be specified in different ways, but the notation we will use is the same as the XYZ triplet, called a vector triplet. For Figure 3.25, the vector triplet is (3,9,7). This indicates that the object will be moved three units in the positive X direction, nine units in the positive Y direction, and seven units in the positive Z direction. Remember that this translation is applied in world space, so the X direction in this case would be eastward, and the Z direction would be down (toward the ground, so to speak). Neither the orientation nor the size of the object is changed.

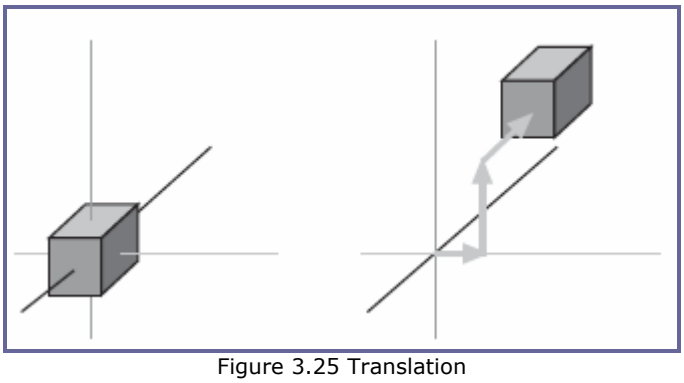

Figure 3.25 Translation

3.2.5 Full Transformation

So now we roll all the operations together. We want to orient the cube a certain way, with a certain size, at a certain location. The transformations applied are scale (s)=1.6,1.6,1.6, followed by rotation (r)=0,30,0, and then finally translation (t)=3,9,7. Figure 3.26 shows the process.

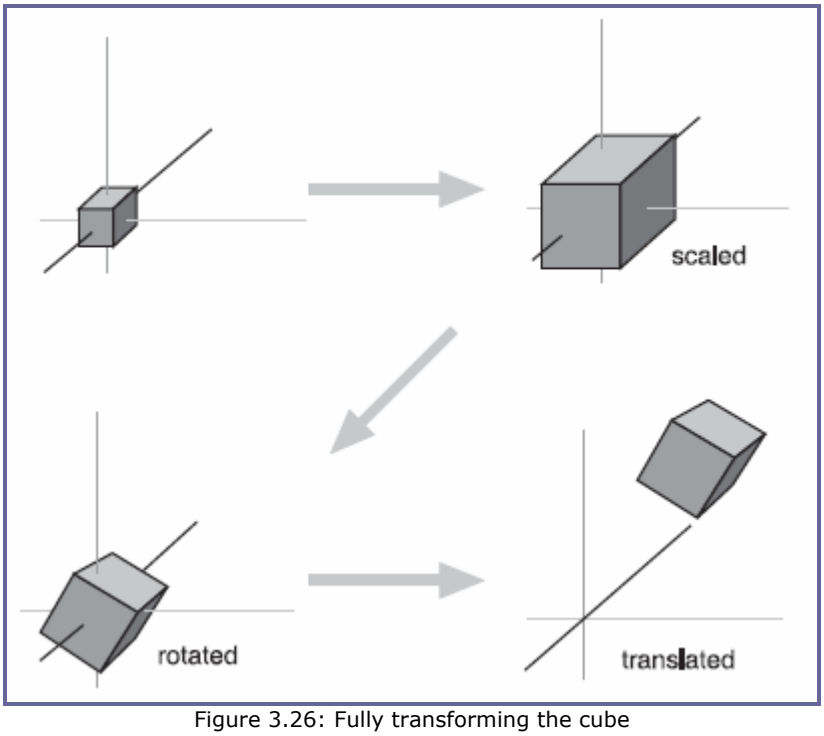

Figure 3.26: Fully transforming the cube

The order that we use to apply the transformations is important. In the great majority of cases, the correct order is scaling, rotation, and then translation. The reason is that different things happen depending on the order. You will recall that objects are created in object space, then moved into world space. The object's origin is placed at the world origin. When we rotate the object, we rotate it around the appropriate axes with the origin at (0,0,0), then translate it to its new position. If you translate the object first, then rotate it (which is still going to take place around (0,0,0), the object will end up in an entirely different position as you can see in Figure 3.27.

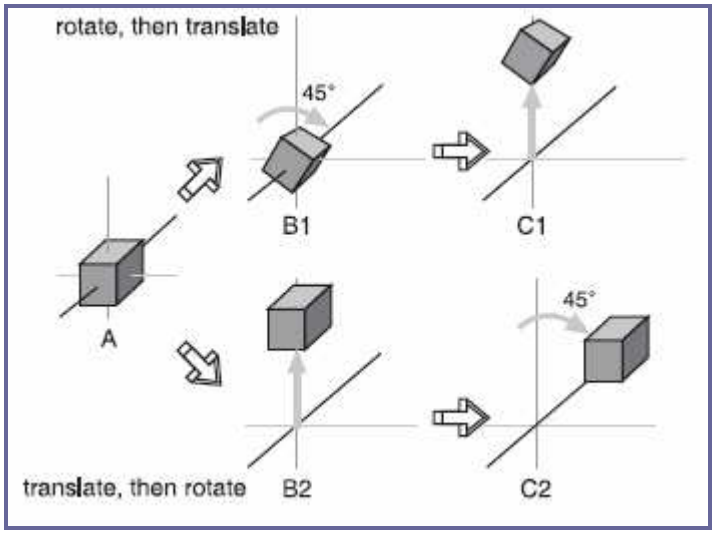

Figure 3.27: Faces on an irregularly shaped object

3.2.6 Rendering

Rendering is the process of converting the 3D mathematical model of an object into an on-screen 2D image. When we render an object, our primary task is to calculate the appearance of the different faces of the object, convert those faces into a 2D form, and send the result to the video card, which will then take all the steps needed to display the object on your monitor.

We will take a look at several different techniques for rendering, those that are often used in video game engines or 3D video cards. There are other techniques, such as ray-casting, that aren't in wide use in computer games—with the odd exception, of course—that we won't be covering here.

In the previous sections our simple cube model had colored faces. In case you haven't noticed (but I'm sure you did notice), we haven't covered the issue of the faces, except briefly in passing.

A face is essentially a set of one or more contiguous co-planar adjacent triangles; that is, when taken as a whole, the triangles form a single flat surface. If you refer back to Figure 3.22, you will see that each face of the cube is made with two triangles. Of course, the faces are transparent in order to present the other parts of the cube.

3.2.7 Flat Shading

Figure 3.28 provides an example of various face configurations on an irregularly shaped object. Each face is presented with a different color (which is visible as different shades). All triangles with the label A are part of the same face; the same applies to the D triangles. The triangles labeled B and C are each single-triangle faces.

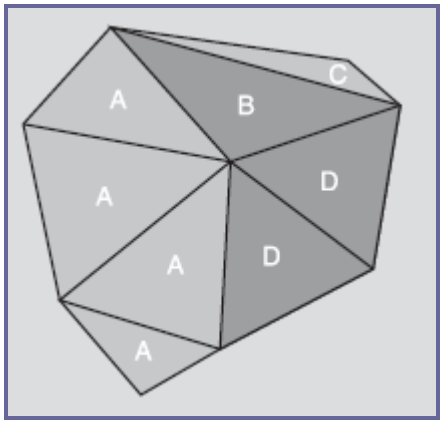

Figure 3.18 Faces on an irregularly shaped object

When we want to display 3D objects, we usually use some technique to apply color to the faces. The simplest method *is flat shading*, as used in Figure 3.27. A color or shade is applied to a face, and a different color or shade is applied to adjacent faces so that the user can tell them apart. In this case, the shades were selected with the sole criterion being the need to distinguish one face from the other.
One particular variation of flat shading is called Z-*flat shading*. The basic idea is that the farther a face is from the viewer, the darker or lighters the face.

3.2.8 Lambert Shading

Usually color and shading are applied in a manner that implies some sense of depth and lighted space. One face or collection of faces will be lighter in shade, implying that the direction they face has a light source. On the opposite side of the object, faces are shaded to imply that no light, or at least less light, reaches those faces. In between the light and dark faces, the faces are shaded with intermediate values. The result is a shaded object where the face shading provides information that imparts a sense of the object in a 3D world, enhancing the illusion. This is a form of flat shading known as lambert shading (see Figure 3.29).

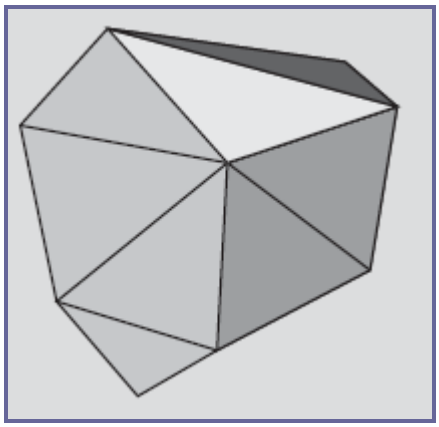
Figure 3.29 Lambert-shaded object.

3.2.9 Gouraud Shading

A more useful way to color or shade an object is called gouraud shading. Take a look at Figure 3.30. The sphere on the left (A) is flat shaded, while the sphere on the right (B) is gouraud shaded. Gouraud shading smoothes the colors by averaging the normals (the vectors that indicate which way surfaces are facing) of the vertices of a surface. The normals are used to modify the color value of all the pixels in a face. Each pixel's color value is then modified to account for the pixel's position within the face. Gouraud shading creates a much more natural appearance for the object, doesn't it? Gouraud shading is commonly used in both software and hardware rendering systems.

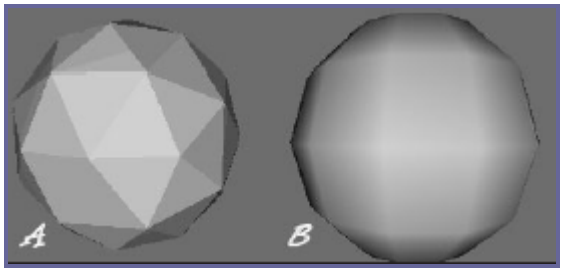

Figure 3.30: Flat-shaded (A) and gouraudshaded (B) spheres

3.2.10 Phong Shading

Phong shading is a much more sophisticated—and computation-intensive—technique for rendering a 3D object. Like gouraud shading, it calculates color or shade values for each pixel. Unlike gouraud shading (which uses only the vertices' normals to calculate average pixel values), phong shading computes additional normals for each pixel between vertices and then calculates the new color values. Phong shading does a remarkably better job (see Figure 3.31), but at a substantial cost.
Phong shading requires a great deal of processing for even a simple scene, which is why you don't see phong shading used much in real-time 3D games where frame rate performance is important. However, there are games made where frame rate is not as big an issue, in which case you will often find phong shading used.

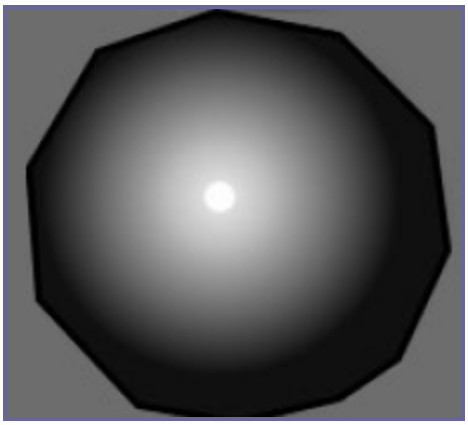
Figure 3.31: Phong-shaded sphere

3.2.11 Fake Phong Shading

There is a rendering technique that looks almost as good as phong shading but can allow fast frame
rates. It's called *fake phong shading*, or sometimes *fast phong shading*, or sometimes even *phong approximation rendering*.Whatever name it goes by, it is not phong rendering. It is useful, however, and does indeed give good performance.
Fake phong shading basically employs a bitmap, which is variously known as a *phong map*, a *highlight map*, a *shade map*, or *a light map*. I'm sure there are other names for it as well. In any event, the bitmap is nothing more than a generic template of how the faces should be illuminated (as shown in Figure 3.32).
As you can tell by the nomenclature, there is no real consensus about fake phong shading. There are also several different algorithms used by different people. This diversity is no doubt the result of several people independently arriving at the same general concept at roughly the same time—all in search of better performance with high-quality shading.

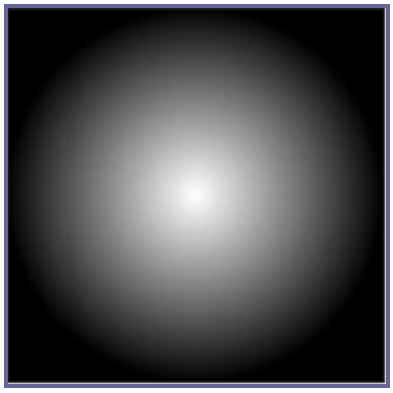

Figure 3.32 Example of a fake phong highlight map

3.2.12 Texture Mapping

Texture mapping is covered in more detail in Chapters 8 and 9. For the sake of completeness, I'll just say here that texture mapping an object is something like wallpapering a room. A 2D bitmap is "draped" over the object, to impart detail and texture upon the object, as shown in Figure 3.33.
Texture mapping is usually combined with one of the shading techniques covered in this chapter.

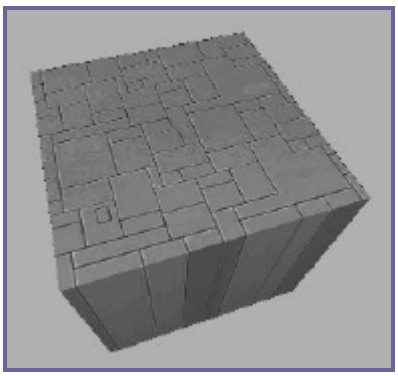

Figure 3.23 Texture-mapped and gouraud-shaded cube

3.2.13 Shaders

When the word is used alone, shaders refer to shader programs that are sent to the video hardware by the software graphics engine. These programs tell the video card in great detail and procedure how to manipulate vertices or pixels, depending on the kind of shader used.
Traditionally, programmers have had limited control over what happens to vertices and pixels in hardware, but the introduction of shaders allowed them to take complete control.
Vertex shaders, being easier to implement, were first out of the starting blocks. The shader program on the video card manipulates vertex data values on a 3D plane via mathematical operations on an object's vertices. The operations affect color, texture coordinates, elevation-based fog density, point size, and spatial orientation.
Pixel shaders are the conceptual siblings of vertex shaders, but they operate on each discrete viewable pixel. Pixel shaders are small programs that tell the video card how to manipulate pixel values. They rely on data from vertex shaders (either the engine-specific custom shader or the default video card shader function) to provide at least triangle, light, and view normals.
Shaders are used in addition to other rendering operations, such as texture mapping.

3.2.14 Bump Mapping

Bump mapping is similar to texture mapping.Where texture maps add detail to a shape; bump maps enhance the shape detail. Each pixel of the bump map contains information that describes aspects of the physical shape of the object at the corresponding point, and we use a more expansive word to describe this—the texel. The name texel derives from texture pixel.
Bump mapping gives the illusion of the presence of bumps, holes, carving, scales, and other small surface irregularities. If you think of a brick wall, a texture map will provide the shape, color, and approximate roughness of the bricks. The bump map will supply a detailed sense of the roughness of the brick, the mortar, and other details. Thus bump mapping enhances the close-in sense of the object, while texture mapping enhances the sense of the object from farther away.
Bump mapping is used in conjunction with most of the other rendering techniques.

3.2.15 Environment Mapping

Environment mapping is similar to texture mapping, except that it is used to represent effects where environmental features are reflected in the surfaces of an object. Things like chrome bumpers on cars, windows, and other shiny object surfaces are prime candidates for environment mapping.

3.2.16 Mipmapping

Mipmapping is a way of reducing the amount of computation needed to accurately texture-map an image onto a polygon. It's a rendering technique that tweaks the visual appearance of an object. It does this by using several different textures for the texture-mapping operations on an object. At least two, but usually four, textures of progressively lower resolution are assigned to any given surface, as shown in Figure 3.34. The video card or graphics engine extracts pixels from each texture, depending on the distance and orientation of the surface compared to the view screen.

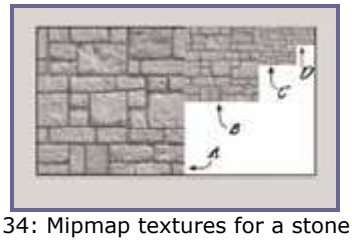

Figure 3.34: Mipmap textures for a stone surface.

In the case of a flat surface that recedes away from the viewer into the distance, for pixels on the nearer parts of the surface, pixels from the high-resolution texture are used (see Figure 3.35). For the pixels in the middle distances, pixels from the medium-resolution textures are used. Finally, for the faraway parts of the surface, pixels from the low-resolution texture are used.

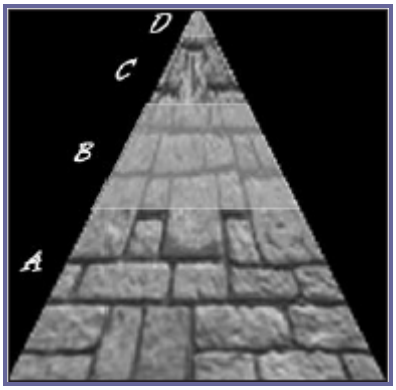

Figure 3.35: Receding mipmap textures on a stone surface

3.2.17 Scene Graphs

In addition to knowing how to construct and render 3D objects, 3D engines need to know how the objects are laid out in the virtual world and how to keep track of changes in status of the models, their orientation, and other dynamic information. This is done using a mechanism called a scene graph, a specialized form of a directed graph. The scene graph maintains information about all entities in the virtual world in structures called nodes.

The 3D engine traverses this graph, examining each node one at a time to determine how to render each entity in the world. Figure 3.36 shows a simple seaside scene with its scene graph. The nodes marked by ovals are group nodes, which contain information about themselves and point to other nodes. The nodes that use rectangles are leaf nodes. These nodes contain only information about themselves.

Note that in the seaside scene graph, not all of the nodes contain all of the information that the other nodes have about themselves.

Many of the entities in a scene don't even need to be rendered. In a scene graph, a node can be anything. The most common entity types are 3D shapes, sounds, lights (or lighting information), fog and other environmental effects, viewpoints, and event triggers.

When it comes time to render the scene, the Torque Engine will "walk" through the nodes in the tree of the scene graph, applying whatever functions to the node that are specified. It then uses the node pointers to move on to the next node to be rendered.

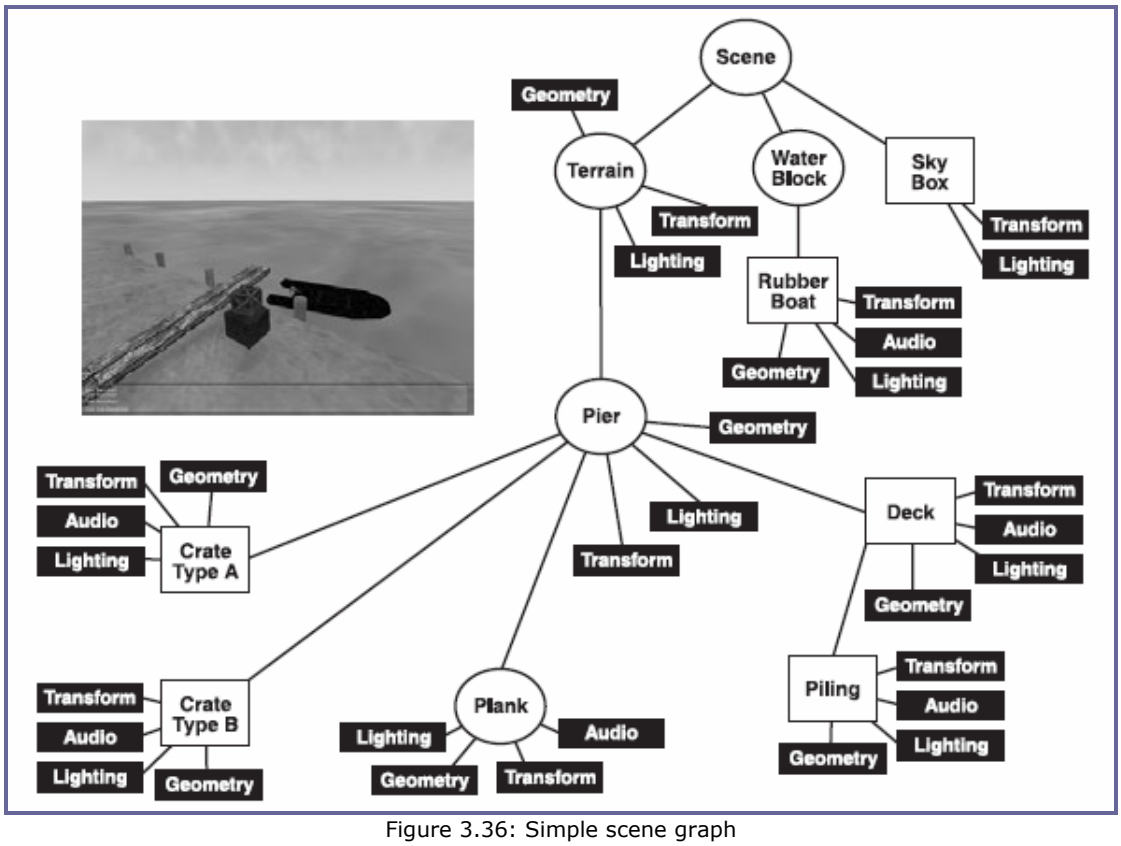

Figure 3.36: Simple scene graph