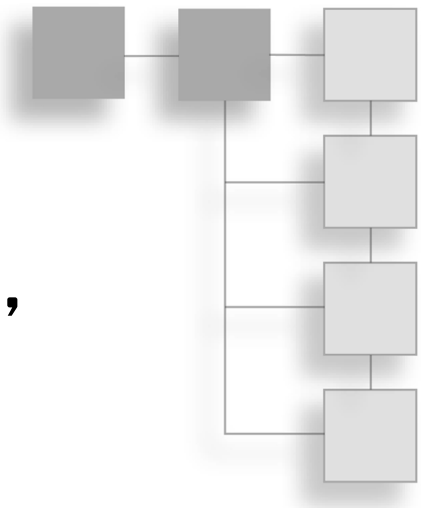
Chapter 14



Practice, Practice, Practice

This chapter discusses specific ways you can revise software. Practices dealt with in previous chapters are emphasized in this context. The approach used involves examining code from Stripe 14, which you can find on the CD. Before making changes to the code, the *Ankh* team first assessed the risks involved. An important part of this activity centered on documenting and diagramming the areas of the system under consideration and the reasons actions might be taken. Generally, why you might perform such work depends on the context of your effort. For example, you might be responding to user complaints, market pressures, or a technical imperative to optimize your code. Revising the code for a game or any other software product presents an enormous field of activity, and it's impossible to cover very many of them in one short chapter. Still, a few topics provide a good starting place:

- Examining the stripes to find candidates for revision
- Determining the more feasible options
- Planning and justifying revisions
- Estimating the effort required
- Developing ways to test the revisions
- Implementing the code
- Revising supporting materials
- Implementing the extension

Software Revision

In many environments, where a given game engine or game code framework (many developers consider an engine to apply only to graphical components) has met with technical

and marketing success, those in executive positions wisely elect to revise rather than re-create. Revision possesses superiority over re-creation after a successful release because revision can almost always improve on an existing framework, whereas starting work on a new product poses all the risks that usually accompany new efforts.

At the same time, revision poses numerous risks. Risks arise because executives, managers, designers, and developers sometimes cannot resist the enthusiasm they feel when they have initially released a successful game. The first impulse, rightfully, should be to follow up with either an improved version of the first game or another game of the same type. Following this impulse can lead to enormous profits because customers are willing to invest in new versions of the games they like or games that are similar to those they like. On the other hand, if developers act on unbridled impulses, risks result. One major risk involves trying to cram a multitude of features into the new release. If you do such a thing, you can create a game that is far more complex and far less perfect than the first. If defects plague the game that follows on your success, you risk losing the victory you worked so hard to earn.

Modifications

This chapter addresses revision in general, but some mention should be afforded at the outset to one specific type of revision: modification. When you develop a modified (or mod) game, you use the framework of an existing game and apply new themes and characters to it. In essence, you present to the world a game you want to be viewed largely as original. You do not present such a game to the world as a new release of an already established game. It is simply a new game that resembles others. This can work well in many instances. Some players want to play games that are modifications of older games. They enjoy this type of product.

There is a subtle art to developing a mod successfully. Success rests in part on recognizing, from the first, the limitations and potentials that apply to mod development. One of the first things to consider is that when you develop a mod, you do so to save development expenses. If you embrace a given game framework as a starting point and then proceed to rework it so extensively that you take as much or more time than you would have required had you begun from scratch, you have clearly defeated your purpose.

Designing a modification involves giving attention to what you can do with the framework as it exists rather than what you can do with the framework if you rework it. If you do not take the time to evaluate the framework and use as much as possible those features that it already possesses, you can easily end up involving yourself in a development effort that easily equals or exceeds in complexity that of an original development effort. This happens because when you rework existing code, you have to perform a great deal of analysis that development of original code does not require.

Still, if you have a framework you want to revise, the economics of revision are well established. Getting your money's worth involves clearly identifying how you can get the most from the revisions or extensions you make to the code you start with. Starting with a body of well-tested, proven code is better than starting from scratch. You are assured success if you take time to evaluate the effort involved in each action you intend to take. Such work involves attending to the scope, complexity, maintainability, and extensibility of the framework you are working with.

Scope and Complexity

Many case histories indicate that success with the development of a game can be hazardous. A common scenario arises when an enthusiastic group of developers enjoys success with a first release and has ample money with which to develop a second release. This is where the trouble starts.

Such a group of developers can decide to concentrate their energies on modifying the first release so that it incorporates an enormous number of untried, risky features. What results is a product that customers do not like and developers regret ever having released. This phenomenon offers an interesting area for psychological study.

The psychological issues that surface involve both software engineering and game design. From the software engineering perspective, the issue is that the developers release a feature-rich game that is plagued with defects. From the game design perspective, the issue is that the customers—most of whom probably bought the second release because they were satisfied with the first—express deep discontentment with the defects in the game and report that they find the game dissatisfying to play.

From the software engineering perspective, the problems result because cramming a multitude of new features into a limited framework is likely to overwhelm the design of the framework. At a certain point, any container can become too small if you try to pack too much into it. Games are like any other containers. Their design determines how much they can be modified and extended.

From the perspective of the customer, the situation resembles that of someone who is forced to take an advanced math course immediately after completing an elementary one. The curve proves too steep. The feature richness is overwhelming. The comfort that the player experienced playing the old game has been lost.

In both cases, development and design, a *cybernetic* or *ecological* quality can be said to govern the extent to which extensions or modifications can be made to the framework of the game. To drive home this notion, many design experts strongly emphasize the notion of KISS (Keep It Simple, Stupid). As harsh as this piece of advice might sound, it lies at the root of a multitude of successfully extended or revised games. Enhancing an existing game

(either as a new release or as a modification) provides an opportunity to increase player satisfaction in an incremental way. The goal of design is to keep the primary experience that the user has of the game consistent with what the user expects while improving the game in ways that satisfy him. Each new feature should represent a gradual step up from the old.

The same general rule applies to technical aspects of a game that are hidden from the player. From the development perspective, controlling complexity involves restricting the modifications you make. You must balance the scope of such an effort so that it does not lead you to completely redesign the existing framework. Effective modifications involve things like increasing performance, streamlining the user interface, and equipping the game with better visual qualities. They do not involve starting from scratch and creating a "super game." If you are going to create a super game, it is best to start from scratch.

Figure 14.1 provides an abstraction of the notion of scope and complexity. One pattern accommodates a possibly endless extension. The other shows disjuncture and fragmentation, implying that the original design does not anticipate the newer extensions.

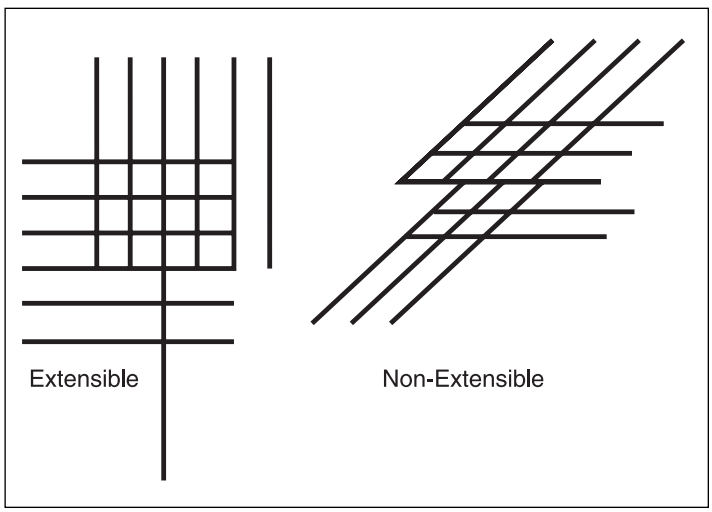


Figure 14.1
Consider at what point modifications can overburden the existing framework.

Technical Symmetry

Enhancing the hidden, technical properties of a game poses risks that are similar to those that arise when you incorporate a vast array of new features visible to the user. The crux of the problem lies in incorporating too much too soon. When too much too soon is incorporated into a game framework, testing and other quality assurance measures can receive short shrift. In addition, feature and functional proliferation can result in a product that lacks technical symmetry. When a product possesses technical symmetry, its technical features consistently represent the same general level of engineering sophistication. Nothing stands out in a glaring, ungraceful way.

With respect to requirements that specify revisions to a software product, the metrics that contribute to defining symmetry can include the number of logical decisions that a given module includes, the anticipated effort that the implementation of the changes or additions involves, and the number of classes that must be revised. Many other criteria can also be included.

Figure 14.2 provides graphical representations of symmetrical and asymmetrical patterns of complexity. In the trends that Figure 14.2 depicts, the development team might ask why one requirement possesses so much more complexity than the others. Is the feature focus of the game to center on this one requirement? If this is the case, the requirement might receive approval. But if this is not the case, the team should perhaps reconsider whether it has adequately refined the requirement.

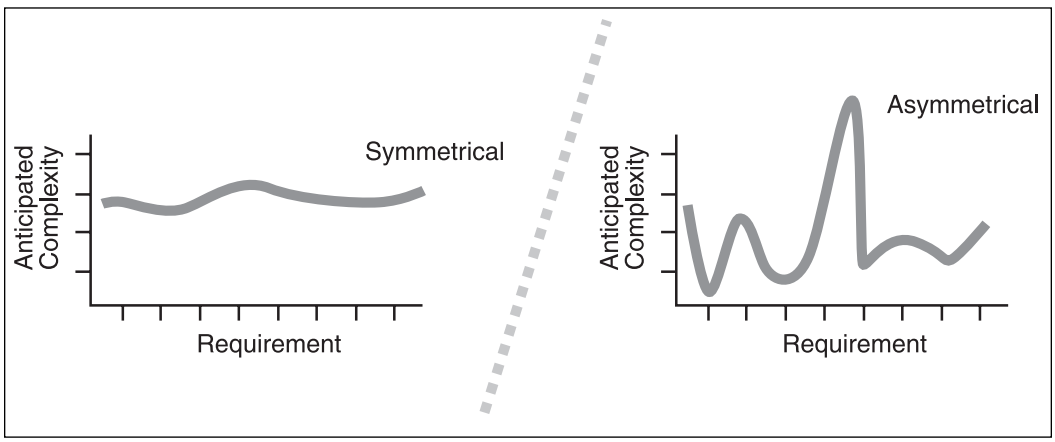


Figure 14.2
An asymmetrical pattern of complexity begs the question of whether revisions have been refined adequately.

Gauging the Impact of Requirements

Complexity of implementation detail encompasses the number of changes or additions that you make to the existing software to achieve new functionally. Consider the requirements that are named in Table 14.1. Suppose that these requirements represent first and second releases of the same game.

Table 14.1 Requirements Comparisons for Revision

No.	Current Requirement	Revision Requirement
1	The game shall feature wavering images that resemble fire whenever a targeted object explodes.	The game shall feature effects for any object designated as a target such that when the target explodes, the flames produced shall precisely simulate those of the materials composing the target.
2	The game shall feature sounds that mark the death or injury of characters.	The game shall precisely imitate the sounds of characters as they are injured, killed, or otherwise affected by actions in the game.
3	Networked players shall be able to communicate with each other.	Networked players shall be able to communicate with each other during gameplay via written, voice, and video communications on a real-time basis.

The two columns of requirements listed in Table 14.1 represent rough formulations, but they still clearly convey requirements that differ substantially as to the extent to which they require new functionality to be implemented. Consider, for example, revision requirement 1, which concerns creating realistic effects for explosions. Taken literally, the requirement makes it necessary for someone on the development team to possess a strong knowledge of chemistry, because the requirement cannot be satisfied unless someone establishes the combustion properties of the substances from which the target objects are made. The way that a burning object appears to burn depends on the combustion properties of the elements that compose it. If the developers for this game were held to the requirement, they would face an enormous task, one that is widely separated from the current requirement. The question then arises as to whether the scope of the current requirement anticipates that of the revision.

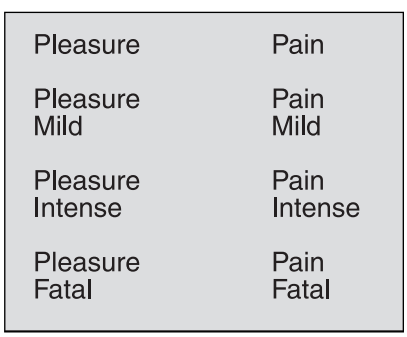


Figure 14.3
The simple utility of pleasure and pain.

Consider the revision of current requirement 2, which concerns the sounds that characters make as they are affected by the actions in the game. Suppose that the way the gameplay affects players can be categorized in four basic ways, as Figure 14.3 illustrates.

What tasks are associated with implementing sounds that convey these categories of pleasure and pain? First, the developers have to identify the significant interactions that define the lives of the characters. Second, they must categorize these interactions to accord with

the states of pleasure and pain. Beyond this, the software developers must provide the functionality that looks up the sound that is assigned to each interaction. The developers might ask how much the revision requirement exceeds the current requirement in terms of complexity.

The final set of requirements, current and revision requirements 3, concerns player interactions through an Internet-distributed game. The revision requirement contains some involved new functionality. It is not possible to tell from the information given how the developers implemented current requirement 3, but they might have simply linked the game to a browser. A dialog box display of online players might have provided these links.

Consider again the possibility of making use of browser capabilities to implement revision requirement 3. Through instant messaging and video streaming, everything might be covered. The difficulty then lies once again in developing local features that allow players to connect to each other using browser capabilities. But then the implications of the requirement are that players connect in the context of play. Suppose that the context of play is an RPG. In this context, players appear or disappear according to the visibility that the game gives them. Game sessions conform to either elective or assigned player interactions, and a database tracks sessions over periods extending from minutes to hours.

Suddenly, the complexity of the game could increase in an enormous way, with the demand that if an extended server system does not already support the game, one will have to be developed to do so. Add to this that some type of architecture must be developed to accommodate the way that the game clients open and close communication sessions.

As all three sets of requirements in Table 14.1 reveal, the complexity that is introduced into a game can increase exponentially if the team does not make an effort to refine requirements for revisions according to the impact they might have on both the system and the development effort.

Linear Growth in Complexity

One simple approach to controlling the complexity that requirements for revisions might impose on a game involves maintaining a linear gradient in the growth of the complexity that you allow requirements for revisions to create. Such a gradient can apply to a system, a module, or a component. Figure 14.4 illustrates a linear complexity growth model.

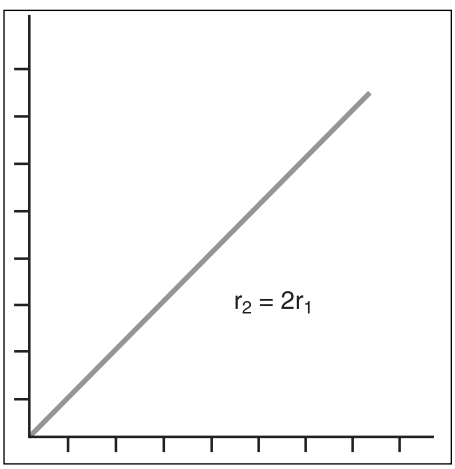


Figure 14.4
Complexity that grows on a linear basis guards against overwhelming development tasks and products that lose their symmetry.

In Figure 14.4, r_2 designates the revised or resultant complexity, whereas r_1 designates the existing complexity of the system. Determining complexity is difficult. (Discussion of this topic appears in Chapter 12, "Numbers for Nabobs.") It suffices in this context to note that the number of decision points or calls within an operation serves as a simple measure of complexity.

The prevailing risk that revision presents centers on quality. Consider, for example, what happens if you extend a thoroughly tested, successful game framework by quickly implementing a set of highly visible features that possess glaring technical and aesthetic flaws. The results almost inevitably ruin the player's experience and damage the game's reputation. Figure 14.5 shows what happens as the complexity of requirements increases. If your team has achieved a high level of testing coverage for the functionality of the game over its successive releases, quality is compromised if your team creates requirements that call for implemented functionality that you do not have time or resources to test.

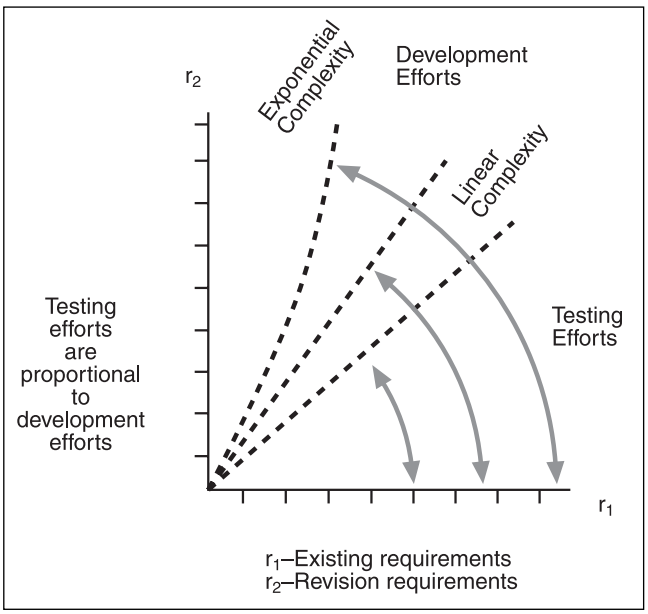


Figure 14.5
If you sustain the same level of test coverage across all aspects of your revision effort and the same level of test coverage from revision to revision, it is likely that your product is evolving in a consistent, linear way.

Determining the Scope of Revisions

As a software developer, you might not enjoy the prerogative of deciding when the product you work with is to be revised. That prerogative might reside with the marketing or customer support group. On the other hand, it's almost always the prerogative of the engineering group to challenge or question the scope of revisions. In fact, not doing so constitutes, in some respects, a failure to perform. If your company has successfully released a product, you endanger the success each time you revise it. A product enjoys what might be viewed as a natural product life. (Its utility or appeal is almost always destined to diminish after months or years.) Nonetheless, if a revised version of the product possesses substantial flaws, the engineering effort kills the product long before its market viability might come to an end.

In addition to life in the consumer marketplace, a software product can face problems when one company sells it to another. Cultures of development treat products in different ways. If a poorly designed but feature-rich product moves from a company that is deficient in design capabilities to a company that excels in design capabilities, the feature status of the game can suffer. This occurs because the engineers who are employed for the acquiring company might decide to put aside feature changes while they rework the architecture.

You can reduce the risks of neglecting product strengths if you create a scope document that balances the general ways in which products can be revised. A scope document defines the general intention of the release. Consider the following points of departure:

- **New features.** The revision aims to incorporate new features. The existing design adequately supports these features.
- **Design extension.** The revision aims to incorporate substantial new features that require extension of the design.
- **Component merger.** The revision aims to merge existing components. The focus of the effort is on the implantation of interfaces that make this merge possible.
- **Fixes and optimizations.** The revision aims at fixes and optimizations. No changes to the architecture or functionality of the game are anticipated.

If you take time to consider, generally, the primary objectives of your revision effort, you immediately put in place a thematic focal point for deliberations about what revisions are acceptable and what the scope of the revision effort should be.

Changes to Ankh

When the development team proposed revision of *Ankh*, its revisions were limited to optimizations. Arriving at the decisions about what to revise required several sessions of debate. The process involved accumulating suggestions and evaluating risks.

To conduct a review of proposed revisions, you can treat the whole process like a requirements gathering session. Participants in the sessions should come prepared to propose requirements for the release. Suggested revisions can be recorded formally and subjected to preliminary debate.

Optimization Candidates

When the team examined *Ankh* for potential revisions, its members proposed several options. Among the options considered were the following:

- **High resolution.** Increasing the game resolution would make the game more attractive visually. After a first pass through the development of the code, refitting the assets and some of the classes with enhanced capabilities would be a relatively easy way to improve the game.

- **Real-time.** *Ankh* is a turn-based game. Making it into a real-time game would give it a different flavor. Developing an AI that uses real-time capabilities would offer opportunities to make the game framework open to different teaching and learning scenarios. For example, both turn-based and real-time modules might be made available to those who want to rework the code for their own purposes.
- **Component optimization.** Numerous technical features of the game were not designed as thoroughly the first time through as they might have been. Among those was the way compression capabilities of DirectX were used. The capabilities were enhanced during a recent release of DirectX. Another item of concern was the performance hit that resulted when vectors were used instead of hashes.
- **Multiplayer.** The game could be made multiplayer as a way to enhance its appeal and demonstrate different development options.

These and other ideas emerged at different times as the team intermittently discussed revising the game. All suggestions were general at first but represented real possibilities for all members of the team.

General Risk Assessment

Assessing the general risks that proposed revisions pose sometimes requires a great deal of effort. That's because different team members favor different revisions. It is hard to relinquish an idea that you have for revision of a system you have worked on for weeks or months. Some of the discussion was fairly heated for the *Ankh* team, even going so far as to involve position papers. The risk assessment sessions rendered the following decisions:

- **High resolution.** The team rejected this revision because it offered little that genuinely enhanced the game and did not provide enough opportunities to change the code in ways that might be suitable for this book. (Needless to say, such a reason would not usually arise in most commercial game development efforts.)
- **Multiplayer.** Modifying the game to be multiplayer arose as an interesting option, and no one voiced strong objections to it. This revision seemed to be an acceptable—if somewhat unexciting—prospect. Reasons for rejecting this revision arose when the team considered that changing the game to multiplayer would bring only minor overall benefit to the game but require a fairly extensive set of changes.
- **Component optimization.** The team realized that component optimization was the best option when it became evident that the game could be enhanced in a variety of ways and that the enhancements could be distributed over a set of seven areas of the system that ranged from minor to involved in complexity. Distribution of the effort over seven tasks minimized the overall impact that the inability to implement any one task might present. On the other hand, overall system quality would be enhanced with the completion of any or all of the revisions.

- **Real-time.** The debate over real-time optimization of the game was prolonged and extensive. Some on the team favored the idea of real-time because it would provide the game with a framework that many people would find interesting. The downside of real-time optimization was the amount of work that such an optimization would involve in duplicating the AI and other modules.

Optimization Selections

The *Ankh* team decided to put aside all proposed revisions except those involving optimization of performance. Such performance optimizations affected immediately visible features of the game, such as the opening dialogue. (See Figure 14.6.) The central task became one of isolating the optimizations that would achieve this end. Among the candidate optimizations were the following:

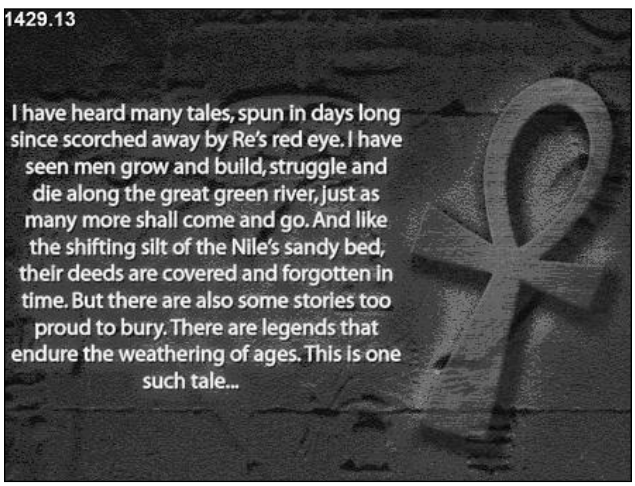


Figure 14.6
Performance impacted the smoothness with which even the images showing the background story displayed.

- **Revise `CResourceMgr` to replace vectors with hashes.** With respect to `sort` and `find` operations, hashes provide greater efficiency than vectors. They are also relatively easy to implement.
- **Revise `CImage` so that it uses DirectX-compressed textures in addition to normal textures.** The class would have to load textures with a *.dxt extension, but it would also be able to use *.png and *.jpg files. Consideration of this change led to discussion of ways that users might be able to create suitable files for the game. Photoshop and a utility that is packaged with the DirectX Software Development Kit (SDK) enable system users to convert files to these formats. This change would result in the use of less memory, generally, so increased performance would likely result.

- **Revise `CTileMap`, `CGraphics`, and `CWorld` so that the game does not draw the tile map to nonvisible areas.** This optimization would involve increasing the sensitivity of the software to detect when tiles need to be refreshed. If there's no change to the area that encompasses the tile, there's no reason to refresh the tile.
- **Revise `CTileMap` so that nothing is drawn below the surface created by the tiles.** This would reduce the amount of work that the graphics card has to do. (See Figure 14.7.)

Figure 14.7
Tuning operations involving the Z buffer increase performance.

- **Revise `CWorld` so that drawing orders are changed.** For example, buildings should be drawn first. Anything that is behind the Z buffer would not have to be drawn. This optimization would also eliminate all instances that allow the same part of the screen to be drawn twice.
- **Revise `CMesh` so that state changes are minimized.** One measure would be to detect whether characters or buildings are off the screen. If so, they should not be redrawn. (See Figure 14.8.)

- **Revise `CEmitter` so that it is updated while drawing.** The original approach to developing the emitter did not follow the code model given in the SDK. Following the model would increase performance.

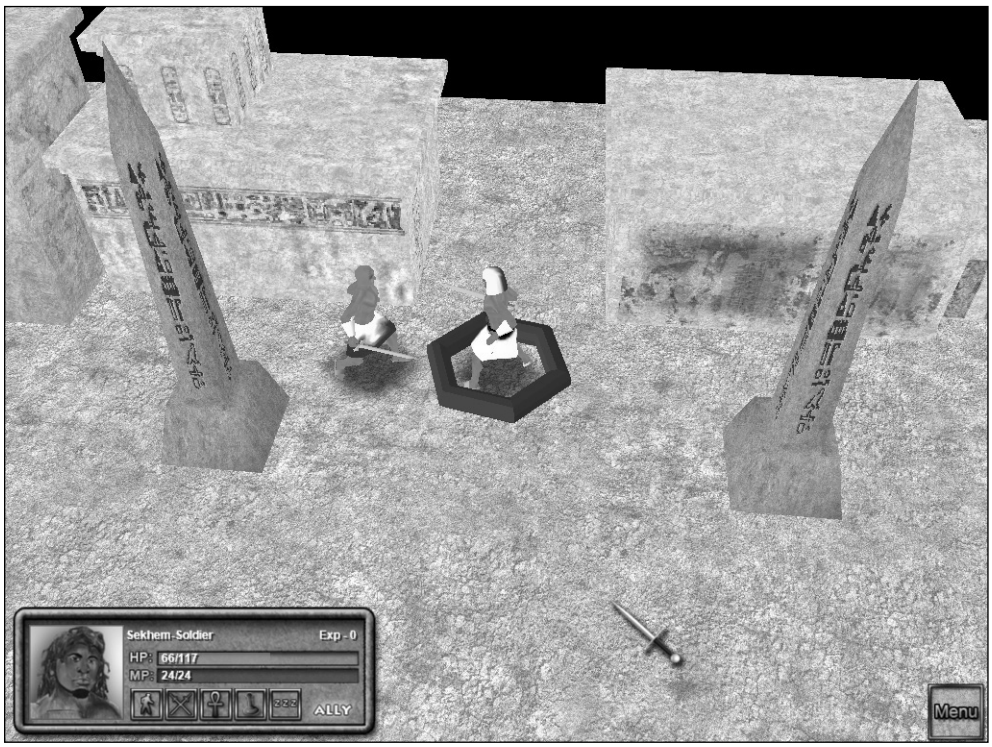


Figure 14.8
Achieving efficiency in the order in which the system paints tiles, buildings, and other meshes was one objective of the proposed revisions.

Ranks of Difficulty and Priority

Ranking the risks posed by difficulty involved considering the scope of the changes to be made, the effort required, and the technical complexity of the changes. The team could assess the risk that the implementation of a given change posed against the benefit that the change promised. Given limited time and resources, how much would benefits justify risks? Changes that pose few risks but also bring few increases in performance or aesthetic richness should probably receive low priority. Extensive change resulting in few benefits poses a high-risk situation and likewise merits low priority. Changes that pose high risks but bring large increases in performance or aesthetic richness should receive a high priority.

Such reasoning stands up in regular test situations. For example, if testers or users find a defect that crashes the game, the result clearly affects both performance and aesthetic

richness in an extensive way. Removing a fatal error always amounts to an extensive change. Thus, you are justified if you assign the change high priority.

Table 14.2 shows a risk-rank view of the classes considered for change. Notice that `CMesh` receives first priority. Even though it poses the greatest risk, it results in the greatest benefit. Changing `CResourceMgr` involves making changes to almost every member operation it contains but results in a net gain to system performance and aesthetic richness that is marginal.

Table 14.2 Ranking of Revision Risks

Rank	Risk	Priority
1	`CMesh`—Most Risk	`CMesh`—First Priority
2	`CEmitter`	`CEmitter`
3	`CWorld`	`CImage`
4	`CTileMap`	`CWorld`
5	`CImage`	`CTileMap`
6	`CResourceMgr`	`CResourceMgr`

Evaluating Classes and Operations

To arrive at estimations of how changes or additions can impact a system, you must inspect the code that the changes impact. For example, the *Ankh* team could not establish the risks and priorities documented in Table 14.2 until it had opened the files containing the classes and carefully examined the operations and attributes that the classes contained to determine how much the changes would be likely to result in altered code.

Revisions to CResourceMgr

The scope of changes involved in revising `CResourceMgr` was extensive. Changing vectors to hashes affected almost every operation. Making this set of changes received low priority even though, in terms of complexity, it posed low risk. In such situations, you can look at the work from different perspectives. One thing to consider is that making extensive changes to a class that communicates with many other classes, even if the changes appear trivial, can result in a testing nightmare. This was one factor that had bearing on the rating of the change. Again, however, the changes were not considered all that difficult, so the testing effort was not deemed a major issue (probably much to the team's hazard). In this case, the tradeoff between the work needed to make the changes and the net gain in performance pushed the set of changes to the tail end of the priority list.

Changes to CMesh

Changing `CMesh` to reduce the number of state changes and prevent buildings and characters that were not onscreen from being drawn involved altering `CMesh:Draw()` and

`CWorld:Draw()`. The complexity of this one operation was a good indication of the overall complexity of `CMesh`. The change involved work with approximately 130 lines of code in the implementation file. It presented a relatively complex testing situation.

Scope of CMesh Changes

Figure 14.9 shows partial details of a UML class diagram representing the attributes and operations of `CMesh`. Two salient features are the `Draw()` operation and the `CImage` objects. The `Draw()` operation has a `DWORD` parameter. The UML diagram adequately details most of the complexity, but one point of concern arises with an association in the `Draw()` operation with a pointer to a `CGraphics` object. The `CGraphics` object is global, declared in the util.cpp file. An *Ankh* class, `CImage`, forms several aggregations with `CMesh`. These, however, do not directly impact the `Draw()` operation. Generally, changes to the operation involve attending to messages among `CMesh`, DirectX objects, and `CGraphics`.

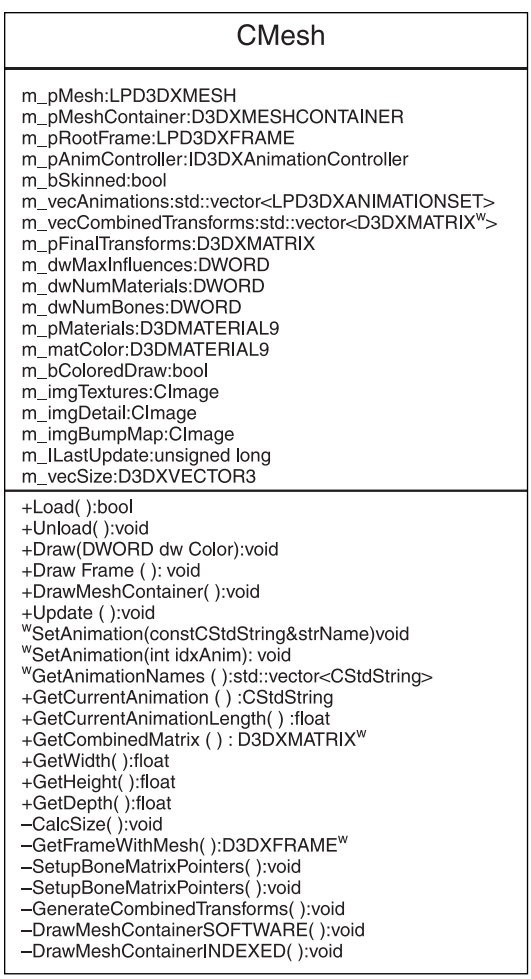


Figure 14.9
A UML class diagram reveals most of the complexity of `CMesh`.

The Code

A final step in assessing risk and planning changes involves going to the code. In some instances, you must add operations. In other instances, additions aren't necessary.

The operation that was central to limiting state changes and changing the visibility of meshes centered on the `Draw()` operation. Because the implementation of the operation consumed approximately 130 lines of code, it was a somewhat large operation. `CMesh:Draw()` contained calls to `CMesh:Update()` and `CMesh:DrawFrame()`, adding to the complexity:

```
//////////////////////////////////////////////////////////
// CMesh::Draw()                                        //
// Draws the mesh.                                      //
//////////////////////////////////////////////////////////
void CMesh::Draw(DWORD dwColor)
{
      if(dwColor != D3DCOLOR_XRGB(255,255,255))
      {
            D3DXCOLOR color(dwColor);
            m_matColor.Diffuse = color;
            m_matColor.Ambient = color;
            m_bColoredDraw = true;
      }
      else
       {
            m_bColoredDraw = false;
       }

       if(m_bSkinned)
      {
            Update();
            DrawFrame(m_pRootFrame);
            return;
      }
      // For each material, render the polygons that use it
      for( DWORD i=0; i<m_dwNumMaterials; i++ )
       {
             // Set the material and texture for this subset
            if(m_bColoredDraw)
              {
                  Graphics->GetDevice()->SetMaterial(&m_matColor);
              }
             else
              {
                  Graphics->GetDevice()->SetMaterial( &m_pMaterials[i] );
              }

            if(m_imgBumpMap && Graphics->GetBumpMapping())
            {
```

```
// Set the direction of the light for the bump mapping
D3DLIGHT9 light;
Graphics->GetDevice()->GetLight(0,&light);
D3DXVECTOR3 vec = light.Direction;
DWORD dwTFactor = VectorToRGB(&vec);
Graphics->GetDevice()->
   SetRenderState(D3DRS_TEXTUREFACTOR,dwTFactor);
if(Graphics->GetDetailMaps()>0)
{
       // Scale the detailmap
       D3DXMATRIX matTexture;
       D3DXMatrixScaling(&matTexture,4,4,4);
       Graphics->GetDevice()->
          SetTransform(D3DTS_TEXTURE3,&matTexture);
        Graphics->GetDevice()->
          SetTextureStageState( 3,
     D3DTSS_TEXTURETRANSFORMFLAGS,D3DTTFF_COUNT2 );
}
else
{
       Graphics->GetDevice()->
            SetTextureStageState( 3,
            D3DTSS_TEXTURETRANSFORMFLAGS,D3DTTFF_DISABLE );
}
Graphics->GetDevice()->SetTextureStageState( 1,
                       D3DTSS_TEXTURETRANSFORMFLAGS,
                                  D3DTTFF_DISABLE );
Graphics->GetDevice()->
           SetTextureStageState(
           2, D3DTSS_TEXTURETRANSFORMFLAGS,
           D3DTTFF_DISABLE );
// Bump map pass
Graphics->GetDevice()->SetTextureStageState(
0,D3DTSS_COLORARG1,D3DTA_TEXTURE);
//normal
Graphics->GetDevice()->SetTextureStageState(
0,D3DTSS_COLORARG2,D3DTA_TFACTOR);
//light vector
Graphics->GetDevice()->SetTextureStageState(
0,D3DTSS_COLOROP,D3DTOP_DOTPRODUCT3);
// Render the texture
Graphics->GetDevice()->SetTextureStageState(
      1, D3DTSS_COLORARG1, D3DTA_TEXTURE);
Graphics->GetDevice()->SetTextureStageState(
      1, D3DTSS_COLORARG2, D3DTA_CURRENT);
Graphics->GetDevice()->SetTextureStageState(
      1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
Graphics->GetDevice()->SetTextureStageState(
      1, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);
Graphics->GetDevice()->SetTextureStageState(
```

```
                        1, D3DTSS_COLOROP, D3DTOP_MODULATE2X);
                Graphics->GetDevice()->SetTextureStageState(
                        1, D3DTSS_ALPHAOP, D3DTOP_MODULATE2X);
        // Add the lighting
                Graphics->GetDevice()->SetTextureStageState(
                2,D3DTSS_COLORARG1,D3DTA_CURRENT);                      //normal
                Graphics->GetDevice()->SetTextureStageState(
                2,D3DTSS_COLORARG2,D3DTA_DIFFUSE);                      //light vector
                Graphics->GetDevice()->SetTextureStageState(
                2,D3DTSS_COLOROP,D3DTOP_MODULATE);
                // Detail map it
                if(Graphics->GetDetailMaps()>0)
                {
                        Graphics->GetDevice()->SetTextureStageState(
                        3, D3DTSS_COLORARG1, D3DTA_TEXTURE);
                        Graphics->GetDevice()->SetTextureStageState(
                        3, D3DTSS_COLORARG2, D3DTA_CURRENT);
                        Graphics->GetDevice()->SetTextureStageState(
                        3, D3DTSS_COLOROP, D3DTOP_MODULATE2X);
                        Graphics->GetDevice()->SetTextureStageState(
                        3, D3DTSS_TEXCOORDINDEX, 1);
                        Graphics->GetDevice()->SetTexture(
                        3, m_imgDetail->GetTexture());
                }
                else
                {
                        Graphics->GetDevice()->SetTexture(3,NULL);
                        Graphics->GetDevice()->SetTextureStageState(
                        3,D3DTSS_COLOROP,D3DTOP_DISABLE);
                }

                // Set up the textures; stage 0 is the
                // bump map, and stage 1 is the texture
                Graphics->GetDevice()->SetTexture(
                0, m_imgBumpMap->GetTexture());
                Graphics->GetDevice()->SetTexture(
                1, m_imgTextures[i]->GetTexture());
        }
        // No bump map, just a regular texture
        else if(m_imgTextures && m_imgTextures[i])
        {
                Graphics->GetDevice()->SetTexture(
                0, m_imgTextures[i]->GetTexture() );
                Graphics->GetDevice()->SetTexture( 1, NULL);
                Graphics->GetDevice()->SetTexture( 2, NULL);
        }
        // No texture!
        else
         {
```

```
                    Graphics->GetDevice()->SetTexture( 0, NULL);
                }
            // Draw the mesh subset
                 m_pMesh->DrawSubset( i );
                Graphics->GetDevice()->SetRenderState(
                            D3DRS_ALPHABLENDENABLE,TRUE);
    }

                // Reset to default state
                Graphics->GetDevice()->SetTextureStageState(
                                0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
                Graphics->GetDevice()->SetTextureStageState(
                                0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
                Graphics->GetDevice()->SetTextureStageState(
                                0, D3DTSS_COLOROP, D3DTOP_MODULATE);
                Graphics->GetDevice()->SetTextureStageState(
                                1, D3DTSS_COLOROP, D3DTOP_DISABLE);
                Graphics->GetDevice()->SetTextureStageState(
                                2, D3DTSS_COLOROP, D3DTOP_DISABLE);
                Graphics->GetDevice()->SetTextureStageState(
                                3, D3DTSS_COLOROP, D3DTOP_DISABLE);
}//end Draw()
```

Other Changes

Unfortunately, page constraints prohibit a detailed discussion of all changes to the *Ankh* system. However, by using such tools as UML diagrams, code inspections, and generalized risk assessment and priority ratings, the team was able to discern how to proceed with revisions. The revisions that this chapter shows represent only minor changes. Such changes do not necessarily characterize industry practices, but it is common for a product revision to consist of numerous minor changes.

The revisions that were designated to become Stripe 15 of *Ankh* were symmetrical. In other words, all revisions required modifications of 1–3 member operations. Likewise, all required modifications took no more than three classes. The following list summarizes the revisions:

- **Changes to `CWorld`.** Changes to `CWorld` supplemented those to `CMesh`. The change that received first priority involved reducing the number of state changes. This class was involved in two revisions. Because of this fact, the team had to assess whether the two revisions would conflict with each other. The revisions did conflict. Both the work to eliminate redundant drawing and to detect objects drawn below the tile surface involved making changes to `CWorld:Draw()`. The team dealt with the situation by coordinating activities so that those involved in making the changes were aware of what the other was doing and could assess how to order their activities to prevent rework.

- **Changes to `CEmitter`.** Changing `CEmitter` so that it would draw particles and chunks with greater efficiency involved altering `CEmitter:Draw()`. Like the changes to `CMesh`, the changes to `CEmitter` required work up front to determine the scope of the changes and the risks involved.
- **Revisions to `CImage`.** The second lowest risk was assigned to `CImage`. Because it allowed the system to reduce memory use and speed performance in fairly substantial ways, it was ranked third in priority. This revision involved changes to `CImage::Load()`.
- **Revisions to `CTileMap`, `CGraphics`, and `CWorld`.** This set of revisions centered on reducing drawing to nonvisible areas. After some analysis, the team determined that only `CTileMap:Draw()` and `CWorld:Draw()` required revision.
- **Changes to `CTileMap`.** Revisions to eliminate drawing below the tile map involved changes to `CTileMap:Draw()`.

Specifying Revisions

You can specify revisions in the same way that you specify primary functionality. You can create a software requirements specification for the revision. The requirements specification for revisions must refer to the primary requirements, if possible, so that you can examine the requirements for revision to determine whether they conflict with the primary requirements. Following are the basic scenarios for merging requirements:

- **Extension.** If you consider the changes made to *Ankh*, the use of the compressed format for asset files constituted an extension of existing functionality. The team retained the old functionality, because if the system encountered *.jpg files, for instance, they would still be read. The revision made possible the automatic compression of files.
- **Substitution.** The use of hashes to replace vectors in the `CResourceMgr` class represented substitution. In this instance, the team decided that the hash container was superior to the vector container. However, the operational interface of the hash container differed little from the vector container, so much of the work involved replacing the container instances. Of course, even though the operational interfaces of the two classes displayed extensive similarities, testing was necessary to establish the success of the substitution.
- **Supersession.** The changes to `CEmitter` constituted an instance of supersession. The change fell under this heading because the implemented code provided a superior solution to the problem, one that used components and algorithms that spoke of an evolved understanding of the problem and its solution. Supersession implied that the resulting code would provide the same functionality as the old code. The moment of supersession arrived with the superior way that the code provided the functionality.

- **Conflict.** As an example of a conflict, consider that if the *Ankh* team had enhanced the game so that it could be displayed only on high-resolution monitors, it might have excluded the vast majority of its prospective customer group. A nonfunctional requirement for *Ankh* might state that the game shall be executable using the widest possible variety of graphics cards.

Use Case Confirmation

Generally, when you engineer requirements for revisions, you can follow the same procedures that you follow when you develop requirements for a system you construct from scratch. Accordingly, regarding the prospective changes to the functionality supported by `CMesh`, a requirement might read

> The system shall restrict painting of meshes to the plane that is defined by the floor of the level.

Given this beginning, you could then create a use case to test the general conceivability of the requirement. Figure 14.10 illustrates a possible use case.

Use Case Name: Manipulate character
Requirement(s) Explored: rev 1
Player (Actor) Context (Role): Player
Precondition(s): Character editor is open.
Trigger(s): Player places character on tile surface.
Main Course of Action: 1. Player moves slider to position character. 2. System positions character according to slider position. 3. System preserves character above tile surface. 4. Player again moves slider to reposition character. 5. System positions character according to slider position. 6. System preserves character above tile surface.
Alternate Course(s) of Action: 1a. Player moves multiple sliders. 4a. Player again moves multiple sliders.
Exceptional Course(s) of Action: 2a. System fails to move character. 3a. System merges character into tile surface. 5a. System does not reposition character. 6a. System merges character into tile surface.

Figure 14.10
A use case serves as a proof of concept for the requirement.

Configuring Revisions

When you determine the scope of a revision, you establish on the most basic level which classes and other components your revisions will affect. To plan the configuration of a revision, you can create a configuration management plan (or update the existing plan). You should name the impacted files and show how to configure them to most effectively facilitate the development effort.

Designing Revisions

The extent to which revisions impact the existing system design depends largely on whether you need to add or factor components. The impact of breaking existing components into new components is greater in most cases than adding new components. That's because if one class (a client) depends on another (a server), the server is likely to impact the client if its operations are moved to classes with different names.

The need to factor did not arise with the revisions that the team proposed for *Ankh*. For example, most of the interactions of the `CMesh` operations involved DirectX, Boost, and `CGraphics` objects. In Figure 14.11, the package symbols represent the DirectX and Boost libraries, and the composition associations indicate that objects from these libraries support operations in `CMesh`. The `CImage` object relates to the `CMesh` object on the basis of aggregation. The object from the `CGraphics` class is declared globally. Calls using its operations occur within the `CMesh:Draw()` operation. Extending the functionality of the `CMesh` class involves no changes to the existing design.

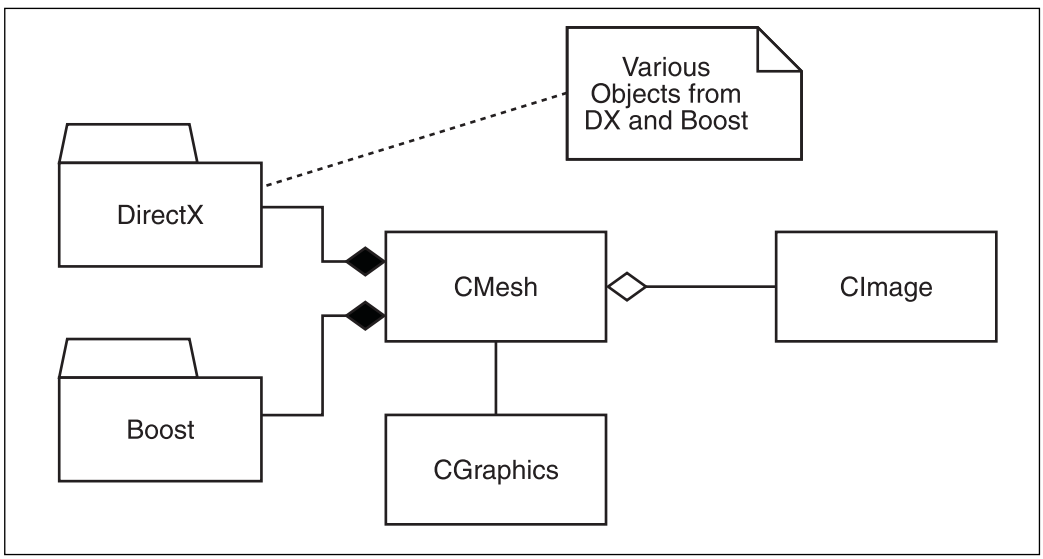


Figure 14.11
The scope of the changes extends over the DirectX, Boost, and a few *Ankh* classes.

Implementing Revisions

One contributing factor to successful implementations of releases involves precisely identifying the components that the release requires you to revise. Identification of components enables you to operate surgically, first changing key operations within existing classes and then testing these through integration with the largely unchanged whole. In the instance of the changes to `CMesh`, the team identified `CMesh:Draw()` as the primary target for changes. The following body of code resulted from the revision of `CMesh:Draw()`:

```
void CMesh::Draw(DWORD dwColor)
{
   // Set the material/color. The color is not WHITE.
   if(dwColor != D3DCOLOR_XRGB(255,255,255))
   {
       D3DXCOLOR color(dwColor);
       m_matColor.Diffuse = color;
       m_matColor.Ambient = color;
       m_bColoredDraw = true;
   }
   else
       m_bColoredDraw = false;
     // If you're skinned, call DrawFrame instead
   if(m_bSkinned)
   {
       Update();
       DrawFrame(m_pRootFrame);
       return;
   }

   // For each material, render the polygons that use it
   for( DWORD i=0; i<m_dwNumMaterials; i++ )
   {
        // Set the material and texture for this subset
       if(m_bColoredDraw)
           Graphics->GetDevice()->SetMaterial(&m_matColor);
       else
           Graphics->GetDevice()->SetMaterial( &m_pMaterials[i] );

       if(m_imgBumpMap && Graphics->GetBumpMapping())
       {
           // Set the direction of the light for the bump mapping
           D3DLIGHT9 light;
           Graphics->GetDevice()->GetLight(0,&light);
           D3DXVECTOR3 vec = light.Direction;
           DWORD dwTFactor = VectorToRGB(&vec);
           Graphics->GetDevice()->SetRenderState(
                        D3DRS_TEXTUREFACTOR,dwTFactor);
                    if(Graphics->GetDetailMaps()>0)
           {
```

```
                Graphics->ApplyStateBlock("MeshDetailAndBumpMap");
                Graphics->GetDevice()->SetTexture(
                                3, m_imgDetail->GetTexture());
            }
            else
            {
                Graphics->GetDevice()->SetTexture(3,NULL);
                Graphics->GetDevice()->SetTextureStageState(
                               3,D3DTSS_COLOROP,D3DTOP_DISABLE);
                Graphics->ApplyStateBlock("MeshBumpMap");
            }
                      // Set up the textures; stage 0
                      // is the bump map, and stage 1 is the texture
            Graphics->GetDevice()->SetTexture(
                               0, m_imgBumpMap->GetTexture());
            Graphics->GetDevice()->SetTexture(
                               1, m_imgTextures[i]->GetTexture());

        }
        // No bump map, just a regular texture
        else if(m_imgTextures && m_imgTextures[i])
        {
            Graphics->GetDevice()->SetTexture(
                           0, m_imgTextures[i]->GetTexture() );
            Graphics->ApplyStateBlock("MeshNoBumpMap");
        }
        // No texture!
        else
            Graphics->GetDevice()->SetTexture( 0, NULL);
              // Draw the mesh subset
         m_pMesh->DrawSubset( i );
        Graphics->GetDevice()->SetRenderState(
                           D3DRS_ALPHABLENDENABLE,TRUE);
    }
        // Reset to default state
        Graphics->ApplyStateBlock("ResetTextures");
}
```

Testing Revisions

Testing revision work differs little from testing the work of primary implementation. A difference does distinguish the two types of work, however. When you revise a product, your testing effort must concentrate on integration from the first. Testing components in isolation is almost secondary. The reason for this should be clear. Even if a development effort renders an excellent component, you cannot subordinate the operational integrity of the entire system to the one component. Testing of the component should assume an integration or system bias. In other words, if the component does not communicate with

the system, then the sanity of the system, rather than that of the component, should be assumed first.

Developing test cases is covered in Chapter 11, "Evident Evil—The Art of Testing." Here, it is useful to show that you can develop a black-box test procedure from the use case illustrated in Figure 14.10. The test procedure allows you to access existing functionality of the system and to operate the system so that it tests the new functionality. Figure 14.12 illustrates a test procedure for the revision.

Test Identifier: S15_ITC_01
Requirement(s) addressed: 1
Prerequisite conditions: Character designer open and character selected.
Test input: Player manipulates character position.
Expected test results: No slider manipulations position character beneath tile surface.
Criteria for evaluating results: Manipulate all sliders and see that character does not merge with tile surface.
Instructions for conducting procedure: 1. Move the slider to position the character. 2. Verify that the character moves with slider position. 3. Verify that regardless of the movement, the character stays above the tile surface. 4. Move the cursor off the slider and click to deactivate the slider. 5. Move the cursor back onto a slider and move the slider to reposition the character. 6. Verify that the system repositions the character. 7. Verify that the character does not merge with the tile surface.
Features to be tested: Slider, painting of character in character designer.
Requirements traceability: See use cases for rev requirement 1.

Figure 14.12
A test black-box test procedure provides a convenient way to verify revised functionality.

As Figure 14.13 shows, you can use the character editor for *Ankh* to test the revised functionality, which the test procedure stipulates.

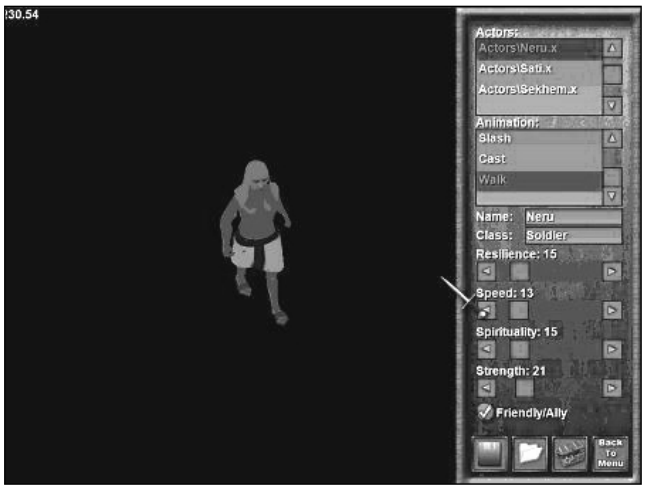


Figure 14.13
The character editor provides a context for testing the new requirement.

Conclusion

This chapter dealt with the dangers and advantages that you encounter when you revise a game framework. Reference was made to a "game framework" in preference to "game engine" because the emphasis has been on using an existing body of code, complete with meshes, textures, and other assets, to create either a modified game or introduce a new release of the existing game. No core set of functionality is necessarily implied, as would be the case with a game engine.

The dangers that modifications or revisions pose usually originate with a failure to thoroughly investigate the scope of the proposed revisions. If you extensively alter an existing framework, you can end up expending more effort than if you had started from scratch. Given that the intention of revision encompasses reuse and improvement of an existing body of code, revision proves worthwhile only if you selectively refine a limited portion of the existing framework.

If you pursue several revisions, you can benefit from assessing the scope of the revisions comparatively. A symmetrical relationship among revisions occurs when all of the revisions possess roughly similar scopes. For example, most of the revisions proposed for *Ankh* involved modifying or extending 1–3 classes. Had any one revision strayed far from this range, it might have been appropriate to question it.

Planning revisions involves assessing risk and setting priorities. You can assess risk by comparing the benefit to be derived from the proposed revision to the amount of work required to bring about the revision. If the amount of work is disproportionate to the improvement that the work will bring to the system, you should consider putting the revision aside. Along similar lines, if you are dealing with a set of revisions, you can reduce risk by assessing which revision has the highest priority. Determining the priority of a revision depends on such factors as how much work it requires, whether it poses high risk, and what benefit it brings to the system. Other factors include whether the revision has dependencies and whether failure to implement the revision poses a risk to the overall revision effort.

Following are books that extend the discussion in this chapter:

Aßmann, Uwe. *Invasive Software Composition.* New York: Springer-Verlag, 2003.

Blunden, Bill. *Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code.* Berkeley, California: Apress, 2003.

Jacobson, Ivar, Martin Griss, and Patrick Jonsson. *Software Reuse: Architecture, Process, and Organization for Business Success.* Reading, Massachusetts: Addison Wesley Longman, 1997.