Design and Implementation of a Graphical Computer Adventure Game

An Honors Thesis (HONRS 499)

by

Karen M. Woznick
Timothy G. Zoch

Thesis Advisor
Dr. Michael McGrew

Ball State University
Muncie, Indiana

May 1998
Purpose of Thesis

Computers are increasingly becoming a large part of nearly every American's life. Applications such as WordPerfect, Microsoft Office, and others have helped shape this growing interest; however computer games draw the largest audiences to the computer. Games have been fun and entertaining to people of all ages in the past years. In this tradition, we have designed a computer game of our own by emulating a real-world software company. As we worked through the planning, development, and testing phases of production, we learned to program efficiently in a graphical setting, work with deadlines, be creative, and cooperate with other members of our group.

We successfully designed a graphical adventure game and demonstrated it to other students. In addition, we have written documentation to explain each aspect of the game including each form, class, and user control. This documentation is included in the following text.
Chapter One

Introduction
Introduction

Sword of Destiny was born of the ideas of several people. It found its origins on a cool October evening when someone came up with the idea of “let’s get together and write a computer game”. This game, like life itself, grew in an evolutionary process. It started out as a vague idea and grew and developed into the finished product that you have in your hands today. In between it has grown, matured, and gone from something small into something that we hope you find to be a very enjoyable game to play.

In addition to the game itself growing and maturing, we as well grew and matured. We learned a great deal about planning out a fantasy world as well as the computer program that would implement it. We also extensively improved our programming skills in Visual Basic 5 and our graphical design skills in Adobe Photoshop 4. We learned to work as a group, use each other to shape ideas, and accept compromise, for the greater good of the project.

We have completed this project with a revised scope and goals, but to our satisfaction and to that of those who have played our game. We never lost the original ideas, but learned to tone them down and convert them into specific computer code. We leave this project with greater programming and creative competency and a sense of pride in our accomplishment.


**Game Background**

During a long meeting, many ideas for the story were discussed and narrowed down into the world and story we finally used in the game. Not all the details were decided, and some of the details were later changed, but the main ideas were agreed upon before the first lines of code were written.

We wanted a diverse but fairly small world in which the player could roam and explore at will. We originally considered have two warring countries, but later we discarded all political boundaries and influences and settled on a single enemy. We also considered having a time constraint in the game, but then decided to just have the enemy wait until whenever the player reached the point where he was ready to fight him.

Finally, we decided what the items that the player was searching for were, and what they would be used for. Then, we decided where each of the items would be located and how the player would obtain them. This is the result:

1.) **Dagger**
   
   **Monastery**
   
   The player obtains the dagger from the Abbot before the game begins.

2.) **Gem**

   **Golem**
   
   The player must kill the golem to obtain the gem from the golem’s forehead.

3.) **Crosspiece**

   **Pub in desert city**
   
   The player must buy the crosspiece from the bartender.

4.) **Gem**

   **Waterfall**
   
   The player must step onto the tile where the waterfall is located to retrieve the gem.

5.) **Pommel**

   **Barbarian Leader**
   
   The player must kill the Barbarian Leader to take the pommel from around his neck.

6.) **Hilt**

   **Hermit’s Isle**
   
   The player receives the hilt from the hermit by talking to him.

7.) **Scabbard**

   **Abandoned Castle**
   
   The player must step onto the tile where the castle is located to retrieve the scabbard from the ruins.
Chapter Two

The Basic Ideas
Graphical User Interface

The first specific programming part to design was the GUI – the graphical user interface. This is what the user - the person playing the game- sees on the computer screen and what objects the user can manipulate, such as a list or combo box. A graphical user interface differs from a textual user interface in two ways. First, in a GUI, it is the user that directs the program; the program responds to the user’s input. In a textual interface, the program prompts the user for input; the user reacts to the program. Second, in a textual user interface, there is only one source of input - at the prompt - and input only comes from the keyboard. Think of the DOS prompt. However, in a GUI, there may be many sources of user input - command buttons, text boxes, etc., and the user can manipulate them using the keyboard or the mouse.

After several revisions, we finalized the structure of our main viewing screen that contains the following elements:

1. MapGrid
2. Character Picture
3. Health Bar
4. Fatigue Bar
5. Equipment List
6. Skills List
7. Eye Button
8. Talk Button
9. Get Button
10. Attack Button
11. File Button
12. Time Counter

1. MapGrid --- The mapgrid will contain a seven by seven grid of terrain pictures so the player can constantly see where he or she is. On this map will also be other characters such as a traveling minstrel and towns which can be entered.

2. Character Picture --- This will be a picture that the player chose to represent his character at the beginning of the game. When it is clicked on, the picture will disappear and a small will appear showing the character’s statistics, such as age, height, weight, armor rating, etc.

3. Health Bar --- This will be a progress bar from zero to the player’s maximum hit points showing the amount of life (hit points) the player currently has.

4. Fatigue Bar --- This will be a progress bar from zero to the player’s maximum fatigue points showing the player’s current fatigue level. The higher the better; as fatigue decreases, the player becomes more tired.

5. Equipment List --- This will be a list box showing what equipment is currently being held by the character, such as clothing, weapons, or food.

6. Skills List --- This will be a list box showing what skills the player currently knows. These may be selected and used at any time.

7. Eye Button --- After clicking on this button, when an item in the inventory is selected, a message box will appear stating what the item is.
8. Talk Button --- After clicking on this button and clicking on an appropriate character on the map, the character will give the player some information via message boxes.

9. Get Button --- After clicking on this button and clicking on an item in the equipment list, the item will be removed from the player's inventory.

10. Attack Button --- After clicking on this button and clicking on an appropriate character on the map, a new screen will appear in which a fight will ensue between the two characters.

11. File Button --- When this button is clicked, a small screen will appear with the options to save the current game, load a new game, cancel and return to the game, or exit the game.

12. Time Counter --- This will be activated by various actions during the game, such as: use of skills, movement, combat, and other time dependent actions. These activities will have a time value associated with them that will relate to the time passing in the game. How much time passes will then be shown by the time counter.
**Combat System**

The combat system is a very complex element of the game. It involves dealing with many variables, often all at once. It has to take into account what the monster is doing, what the player is doing and how to determine who gets to do what to whom when. A lot of time and thought went into planning the system. Even more went into testing the system once it was developed.

The combat system depends heavily on modified values to help determine an action at any given time. The following is a table of positive and negative bonuses applied towards different aspects of the combat system based on four statistics in the game.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Bonus Modifier</th>
<th>Statistic</th>
<th>Bonus Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-5</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-4</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-3</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>-3</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>-2</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>-2</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>-2</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>-1</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>-1</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>27</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>28</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>30</td>
<td>6</td>
</tr>
</tbody>
</table>

Whenever a bonus to a statistic is applied it uses the following table. 12-14 are the low ends for average and as such, apply no modifier. Anything above or below that applies an appropriate modifier whether it be to damage (strength) or hit roll (agility). Health also affects hit points and fitness affects fatigue according to this chart.

**Evade**

Evade is a unique idea for the combat system. It adds a sense of realism to the game by determining whether or not you are hit by a number, differing from armor. Evade is determined by the player’s agility score added to a base of 20 evade points. Evade tends to decrease as a person gets more fatigued.

**Initiative**

A random number generated between one and twenty and the addition of any bonus modifiers from statistics such as Agility and possible equipment modifiers determines initiative. Initiative selects who gets the first attack in any given round of combat.
The Beginning

The Attack loop begins by determining whether or not a hit has occurred, this is done by the following formula: (hit roll + hit modifiers) compared to (evade * (fatigue points / 100) <this number will be a float) if the hit is better than or equal to the evade, then the person doing the hitting connects for damage. The other case is that the swinger misses. If this occurs, the combat turn is over.

Damage

If a hit occurs, the damage subroutine is called. A digit is passed into the subroutine to determine the level of attack that will occur. A one denotes a light attack, a two, a medium level attack, a three is a heavy attack.

All of the attack levels are based of the medium attack. The medium attack does base damage of the damage roll plus any damage bonuses. The light attack does half the damage of the medium attack, but requires fewer fatigue points to initiate. The heavy attack spends the most fatigue points (about one quarter) but also does the most damage (double the amount of the medium attack). All attacks are subtracted from the defender's armor.

Fatigue points are expended on each attack. The light level attack expends 3 points. The medium attack expends 15 points. The heavy attack is the most devastating to your ability to move, draining 25 fatigue points or a full quarter of your possible fatigue points.

A sample: Joe vs the goblin
The goblin has an evade of 24. Joe has an evade of 17. Joe swings at the goblin, rolling a 23 for his hit. Joe misses and his turn ends. The goblin rolls a 20. This hits Joe. The goblin "selected" a medium level attack. So the goblin then rolls for his damage (he does 45 points). Joe's armor of 35 absorbs all but 10 points of damage, which Joe has to take. Battle would then continue until someone flees or someone is dead.

Modification

After some testing and further work, there were changes made to the attack loop. One of the first modifications that was made was in making a change in the way that evade is calculated. The original formula for assessing evade as affected by fatigue was rather harsh. Alterations were made to the algorithm to correct that error. The algorithm was changed to:

$$\text{EVADE} = \text{EVADE} - ((100 + \text{FATIGUE}/200) \times \text{EVADE})$$

This made the loss of fatigue less drastic to the player's or monster's ability to avoid attacks than it had been before.

Another less drastic modification was made to the damage dealing system. The middle level attack was not changed. However, slight adjustments were made to the heavy level attacks. The heavy attack now follows an (2 * damage dice) + bonus damage algorithm. The reasons for these alterations are to control damage to make sure that id
did not get out of hand, which it would have done in the case of the heavy attack were the damage modifier to be added before multiplying by two.

**Armor**

Armor is directly related to damage. Armor allows a character to absorb damage that is dealt from a monster, and vice versa. The original idea was that armor would absorb a number of points of damage equal to the armor rating the player had. On further reflection and discussion, it was decided that armor would absorb a percentage of the damage dealt up to a maximum of 95 percent for a perfect 100 armor rating.

**ARMOR RATINGS CHART**

<table>
<thead>
<tr>
<th>Armor Name</th>
<th>Rating</th>
<th>Affect Evade</th>
<th>% Absorbed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buckler</td>
<td>6</td>
<td>-1</td>
<td>6%</td>
</tr>
<tr>
<td>Small Shield</td>
<td>12</td>
<td>-2</td>
<td>12%</td>
</tr>
<tr>
<td>Large Shield</td>
<td>20</td>
<td>-4</td>
<td>20%</td>
</tr>
<tr>
<td>Plate Boots</td>
<td>15</td>
<td>-3</td>
<td>15%</td>
</tr>
<tr>
<td>Helmet</td>
<td>15</td>
<td>-3</td>
<td>15%</td>
</tr>
<tr>
<td>Leather Armor</td>
<td>10</td>
<td>-4</td>
<td>10%</td>
</tr>
<tr>
<td>Chain Mail Armor</td>
<td>20</td>
<td>-5</td>
<td>20%</td>
</tr>
<tr>
<td>Plate Mail Armor</td>
<td>30</td>
<td>-6</td>
<td>30%</td>
</tr>
<tr>
<td>Full Plate Armor</td>
<td>45</td>
<td>-8</td>
<td>45%</td>
</tr>
</tbody>
</table>

Note: The heavier the armor, the greater the negative affect on evade. The reason for this is that the heavier armors would slow down a real person, but they do offer better protection. A heavily armored knight may get hit more often, but does not take nearly the damage from the attack that a person in cloth clothing would. Armor lore helps to negate this. As the player learns to use armor better, the loss to evade is negated.

**Skills In Combat**

Many skills are utilized in combat. Most work without a need to be called. These improve the chances to hit, improve evade, reduce the fatigue expended, or add damage to the attack. There are, however, three skills that must be called during combat. The three are special attack and defense forms that must be called. These three are Punch, Kick, and Parry.

Punch and kick are very similar. Both are physical attacks that take a combat turn to execute. They both deal damage slightly more a medium level attack would do and expend a little less fatigue than the medium level attack does. This gives the player a chance to inflict slightly better damage than a regular attack would without spending many valuable fatigue points to deal the damage.

Parry is a lot like defend. While in parry mode, the player assumes a defensive stance. This provides a better chance to avoid an attack. Unlike defend however, the player does not benefit from a recovery of fatigue points while in battle. The reason for this is that while parrying, there is a slight chance of mounting a counter attack that cannot be avoided by the enemy.
Defend

Defend allows the player to go to a defensive style of combat. While in a defensive stance, the player has less of a chance of being hit. Being defensive also allows the player to "rest" during combat. This "rest" allows the player to regain a few lost fatigue points that would usually not be regained until after the battle is over.

Escape From Battle And Pursuit

Every once in a while, a battle is going to go badly, and the player needs a way to escape. Flee allows the player a 50% chance of getting out of combat. Monsters may also attempt to flee as well and also have the same 50% chance to get away. Both the monster and the player have the option to pursue the fleeing counterpart. By choosing to pursue, there is a 50% chance that the opponent will not escape.

Monsters

A "monster" in game terms is any creature with which the player may engage in combat. Monsters can come in many shapes and sizes from goblins to dragons. These monsters are very different from one another, but they are all based off of the same template. The template and brief description of the items in the template follows. A more detailed description of those items that need it follows the template.

<table>
<thead>
<tr>
<th>Name</th>
<th>Monster’s Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Hit Points</td>
<td>Total number of hit points the monster has</td>
</tr>
<tr>
<td>Current Hit Points</td>
<td>Amount of hit points the monster has left</td>
</tr>
<tr>
<td>Fatigue Points</td>
<td>Fatigue points the monster has to use</td>
</tr>
<tr>
<td>To Hit</td>
<td>Bonus to the hit roll</td>
</tr>
<tr>
<td>To Damage</td>
<td>Bonus to the damage roll</td>
</tr>
<tr>
<td>Light %</td>
<td>Percent chance that the monster will use a light attack</td>
</tr>
<tr>
<td>Medium %</td>
<td>Percent chance the monster will use a heavy attack</td>
</tr>
<tr>
<td>Drift</td>
<td>Factor of how the monsters attacks drift when it takes damage</td>
</tr>
<tr>
<td>Has Special</td>
<td>Does the monster have a special attack?</td>
</tr>
<tr>
<td>Special</td>
<td>The special that the monster has</td>
</tr>
<tr>
<td>Special %</td>
<td>Percent chance the monster will use the special</td>
</tr>
<tr>
<td>No Flee</td>
<td>How often the monster will try and prevent the player from fleeing</td>
</tr>
<tr>
<td>Flee %</td>
<td>Percent at which the mob’s first priority is escape</td>
</tr>
<tr>
<td>Armor</td>
<td>Monster’s ability to absorb damage</td>
</tr>
<tr>
<td>Evade</td>
<td>Monster’s ability to avoid attack</td>
</tr>
<tr>
<td>Damage</td>
<td>Number of “sides” on the dice rolled to determine the amount of damage done</td>
</tr>
<tr>
<td>Damage Dice</td>
<td>Number of dice rolled in determining damage</td>
</tr>
</tbody>
</table>
Hit points are the total life force that the monster possesses. The current hit points of the monster indicate the amount of life the monster has left. At the beginning, current hit points and total hit points are equal, as the monster takes damage, the current hit points will drop, indicating that the monster has taken damage. The system works similarly for the player, whose hit points are indications of the total amount of life the player has remaining.

Fatigue points illustrate how tired the monster gets during the battle. The more fatigue points expended, the more tired the monster is. As the monster grows tired, it will not be able to defend itself as well as when it was fresh, also, it will not be able to hit as hard as when the battle started.

Light and heavy attack percentages are used to determine the likelihood that the monster will use that level of attack in combat. By taking these two percentages, adding them together, and subtracting that total from one hundred, the percentage of the medium level attack is obtained. This allows the same latitude of level of attack to both the player and the monster. Drift is the way that a monster’s attacks will drift as it loses hit points. Drift is in a range from positive to negative 10. If a monster has a 5 drift, then its attacks, when under 50 percent of its hit points, would drift 5 percent better towards the heavy and medium attack. If the five had been negative, then the drift would have been towards the light and medium attacks.

Next, there are the 3 template slots used for specials. Specials are special attack forms that the monster may have, such as a dragon's fire breath, a really good fighter's hay maker punch, and the like. When used in attack, the special takes one attack turn to use.(in effect serving as the monster’s attack for that round). The Has Special slot in the template is a simple binary digit. One indicates that the monster has a special attack form, a zero indicates that the monster has no special attack form and the other special slots are ignored. The Special slot indicates the special that the monster possesses and the special percentage indicates the likelihood of the monster using the special at any given time during combat.

The final area of interest is the attack and how monsters deal with it. Monsters have evade and armor ratings just like the players do. These statistics are used just as they are by the player. Evade helps to avoid attacks and armor mitigates the damage done by an attack. The way the monster determines the damage it does is slightly different from that of the player. The “To Hit” field provides the monster’s bonus as it attempts to hit the player. The monster has a field called damage. This is the factor that determines how much damage the monster might do. For example, a monster with a damage of 4 would do 1 through 4 points of damage, whereas a monster with a 10 for damage would do one through 10 points of damage. The damage dice number is the number of times that damage is rolled. So, if a monster had a 3 dice number and a 2 damage number, the monster would “roll” 3 times and get a result of between one and two on each roll. These rolls would then be added up and the entire number done is the damage inflicted. It is to this number that the to damage modifier is added (either before or after modification for a light or heavy attack depending) to provide the total amount of damage done to the player by the monster.
**MonsterType** reflects the type of monster encountered. There are four types of monsters in the game. The type of monster helps to determine what type of treasure the monster may give the player when the monster is killed. The value may also influence various skills, such as animal lore.
Chapter Three

Description of the Code
FirstScreen

This is a simple screen showing a welcome message and a menu with options to start a new player, load a player, or exit. It is the first screen a player will see, although it is not the first that is loaded. It also reads in all of the item information from the item database.

ExitCommand_Click
   Unloads all currently loaded forms and ends the program.

Form_Load
   Calls PaintBack and centers labels and buttons on the screen.

Initialize_Items
   Reads in all the items from a database into a global item array.

LoadPlayerCommand_Click
   First calls Initialize_Items, then loads a player from a file and unloads the current form.

NewPlayerCommand_Click
   First calls Initialize_Items. Then calls StoryForm, NewPlayerForm, and SkillsForm. Each of these forms waits for the previous one to be unloaded before it is loaded. Unloads FirstScreen form.

PaintBack
   Uses an embedded for loop to paint the background picture to all of the form.
**StoryForm**

This is a simple form used to display the introductory story. Since the story is too long to fit onto one screen, it is divided into 3 parts, with a button at the bottom of each page for the player to continue to the next page.

**Form_Load**

This simply sets up the location of the text box and buttons, and paints the form picture to the entire screen. At this time, only the page2 button is visible.

**OkayButton_Click**

When the okay button is clicked on, the form is unloaded, allowing the main game form to be visible.

**Page2_Click**

This sets the page2 button to not visible, loads the second page of the story, and sets the page3 button to visible.

**Page3_Click**

This sets the page3 button to not visible, loads the third page of the story, and sets the okay button to visible.
NewPlayerForm

This is the form that is loaded first if the player chooses the new player option from the FirstScreen. Here he will choose a name and picture for his character, and set his personal statistics.

A picture of the NewPlayerForm:

![NewPlayerForm Image]

Global Variables:

- Byte Stats_Number
  - The following six variables are used to determine whether the player is increasing or decreasing each statistic.
  - Byte LastStrengthValue
  - Byte LastAgilityValue
  - Byte LastFitnessValue
  - Byte LastHealthValue
  - Byte LastIntelligenceValue
  - Byte LastPersonalityValue

FemaleImage1_Click

Sets the frame to surround the selected image. Sets the player’s picture to "female1.gif". Sets the player’s sex to female.

FemaleImage2_Click

Sets the frame to surround the selected image. Sets the player’s picture to "female2.gif". Sets the player’s sex to female.
FemaleImage3_Click
Sets the frame to surround the selected image. Sets the player’s picture to “female3.gif”. Sets the player’s sex to female.

Form_Load
Sets the player’s image to “temp” so that at the end we can tell whether or not he has chosen a picture. Sets the GameForm.cur_x and GameForm.cur_y to the appropriate load position for a new player. Centers the labels and images and paints the background with the form’s picture. Initializes StatPointsLabel to 40 and sets the LastStrengthValue, LastAgilityValue, LastFitnessValue, LastHealthValue, LastIntelligenceValue, and LastPersonalityValue to the current values on the labels.

MaleImage1_Click
Sets the frame to surround the selected image. Sets the player’s picture to “male1.gif”. Sets the player’s sex to male.

MaleImage2_Click
Sets the frame to surround the selected image. Sets the player’s picture to “male2.gif”. Sets the player’s sex to male.

MaleImage3_Click
Sets the frame to surround the selected image. Sets the player’s picture to “male3.gif”. Sets the player’s sex to male.

OkayCommand_Click
When the new player clicks the okay button, several checks are made before the new character is set up. First a check is made to see whether there is any text in the name text box. If not, a message is displayed asking the new player to enter a name for his character. Then, a check is made to see if the image name is still “temp”. If so, then the player has not chosen a picture for his character, so a message is displayed asking him to do so. Then a check is made to see if the StatPointsLabel.Caption is zero. If it is not zero, then the player has not finished distributing his statistics points, so a message is displayed asking him to do so. Once all of these conditions are satisfied, the new player’s character is set up. His name and statistics are set to those displayed on the screen, and then several functions are called: Calculate_HitBonus, Calculate_DamageBonus, Calculate_FatigueBonus, and Calculate_HitPointsBonus. Then the player’s evade is set to twenty plus his agility rating, and his armor is set to one, since he is not wearing any armor. A zero can not be used because it would cause division by zero errors in the attack functions. Then, the following items are added to the player’s inventory: a monk’s robe, a pair of sandals, the special dagger, a loaf of bread, a hunk of cheese, and some wine. Then, the monk’s robe is set to the player’s body location, the sandals are set to the player’s feet location, and the special dagger is set to the player’s wielded position. This is used to show which items are currently being used by the player. A special item called nothing is set to the player’s head, shield, and about body, to avoid future errors that might be caused if the player is not wearing anything in these positions. The new player’s money is then set to 5 silver pieces, his bandaged, poisoned, performed, and
begged flags are all set to zero. Then his height and weight are set according to whether he has chosen to be male or female. Finally, the form is then unloaded.

The rest of the subroutines are similar in implementation, one for each of the six statistics.

**StrengthPoints** Change

First compares LastStrengthValue to the current StrengthPoints.Caption. If LastStrengthValue is less that the StrengthPoints.Caption, then a check is made to determine what range the value of the StrengthPoints.Caption is in. If it is between eight and nineteen, then a one is subtracted from the StatPointsLabel.Caption. Then a check is made to see if the StatPointsLabel is now less than zero. If so, the player has run out of points to distribute, so the one is added back on and the StrengthPoints.Caption is set back to the LastStrengthValue. If the StrengthPoints.Caption value is between eighteen and twenty-three, the process is repeated with a two. If the StrengthPoints.Caption value is between twenty-two and twenty-seven, the process is repeated with a three. If the StrengthPoints.Caption is greater than twenty-six, the process is repeated with a four. This way, the player can easily raise his statistics in the beginning, but to raise a statistic to a very high value would require most of the available points.

If the LastStrengthValue is greater than the StrengthPoints.Caption then a check is first made to determine whether the StatPointsLabel equals forty. If so, nothing happens. Otherwise, a check is made to determine which range the StrengthPoints.Caption is in. Depending on which range, a different number of points is added back to the StatPointsLabel.Caption. If the StrengthPoints.Caption is between seven and eighteen, one point is added on, between seventeen and twenty-two, two points are added, between twenty-one and twenty-six, three points are added, and between twenty-five and thirty, four points are added. Finally, the LastStrengthValue is set to the new value of the StrengthPoints.Caption.

A picture of the NewPlayerScreen:
SkillsForm

This simple form is used to allow the new player to distribute a number of points to various skills. These basic skills are used to get the player started in the game, allow him to make money, and enable him to learn more advanced skills.

Global Variables:
These variables are used to keep track of whether the player is increasing or decreasing the particular skill.
- Byte LastAnimalLoreValue
- Byte LastArcheryValue
- Byte LastBrawlingValue
- Byte LastCampingValue
- Byte LastChoppingValue
- Byte LastCommerceValue
- Byte LastFireBuildingValue
- Byte LastFishingValue
- Byte LastForagingValue
- Byte LastHikingValue
- Byte LastHuntingValue
- Byte LastItemLoreValue
- Byte LastSocialValue
- Byte LastStealthValue
- Byte LastWrestlingValue

Form_Load
This sets up the positions of the labels and initializes all the global variables. It also paints the background with the form's picture.

OkayButton_Click
First a check is made to see whether the SkillsPointsLabel is zero. If it is not zero, then the player has not distributed all of his points, so an appropriate message is displayed. If it is zero, then the player's skill ratings are set to those on the labels, and then the form is unloaded.

Each of the rest of the subroutines on this form follow the same layout. There is one subroutine for each of the skills on the form.

AnimalLorePoints_Change
First a check is made to see if LastAnimalLoreValue is greater or less than the AnimalLorePoints.Caption. If it is less (meaning that the player has increased the AnimalLorePoints.Caption), then a one is subtracted from the SkillsPointsLabel.Caption. If the SkillsPointsLabel.Caption is less than 0, then a one is added back and the AnimalLorePoints.Caption is set back to the LastAnimalLoreValue. Otherwise, if the LastAnimalLoreValue is greater than the AnimalLorePoints.Caption (meaning that the player has decreased the
AnimalLorePoints.Caption), then a one is added to the SkillsPointsLabel.Caption, unless it is already 250. At the end of the subroutine, the LastAnimalLoreValue is set to the new AnimalLorePoints.Caption.
GameForm

This is the main part of the game. This form is loaded first and every other form and function is called from it. It shows the game map, item and inventory arrays, the player’s picture etc. See the section on the GUI for more details.

A picture of the GameForm:

![GameForm Image]

Global Variables:
- **String State** - Keeps track of which command button was pushed last. This enables the Map_Click subroutine to determine which event should now occur.
- **Integer Map** which is a 100 * 100 matrix to keep track of the game map.
- **Integers Cur_X and Cur_Y** which keep track of the current location on the map.
- **Boolean Fire** to determine whether the player has built a fire.

**Add**

This function adds a new item to both the player’s inventory and the equipment combo box on the form. This way, both lists are exactly the same and an item in the inventory can be accessed by using the selected index from the equipment box. This function also checks to see if the player’s inventory is full (100 items).

**Appraising**

First checks to see if the player is holding something, then calculates a value for the item by examining the item’s value and the player’s skill at appraising. Also, there is a skill check to determine whether the player’s skill at appraising improves, which is done by calling the Improve_Skill function.

**ArrowButton_Click**
Sets the carving menu to invisible, checks to see if the player has enough fatigue points, and makes a skill check to determine whether the player was successful at carving an arrow. If the player was successful, the wood is removed from his inventory and an arrow placed there. If the player is unsuccessful, the wood is also removed from his inventory and it is determined whether the player's skill at carving was improved, which is done by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**Attack_Click**
Changes the State to "attack" if this same button was not previously pushed. Else it cancels the last click and returns State to normal. Also changes the mousepointer to the appropriate icon.

**Bandaging**
First it is determined if the player is holding a first aid kit, whether he has enough fatigue points, and if he has previously bandaged that day. If he satisfies these conditions, a number is calculated based on the player's skill at bandaging and a random number between 1 and 6. This number is then added on to the player's current hit points, with a maximum of 100. Also, there is a skill check to determine whether the player's skill at bandaging improves, which is done by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**BenchCommand_Click**
Sets the carpentry frame to invisible. A check is made to see if the player has enough fatigue points and a skill check is used to determine if the player was successful at making a bench. If he was successful, the wood is removed from his inventory and a bench is placed there. If he was unsuccessful, the wood is removed from his inventory and he is given a chance to improve his skill at carpentry, which is done by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**BerriesButton_Click**
Erases forage frame from screen. Checks to see if the player has enough fatigue points and makes a skill check to see if player was successful at foraging for berries. If player was unsuccessful, a messagebox will appear telling him so and he will be given an chance to improve his skill, which is done by calling the Improve_Skill function. If player was successful, the appropriate amount of berries are added to his inventory. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

The message displayed when the player has successfully foraged for berries:
You have successfully foraged some berries.

BowCommand_Click

Sets the carpentry menu to invisible, checks to see if the player has enough fatigue points, and makes a skill check to determine whether the player was successful at making a bow. If the player was successful, the wood is removed from his inventory and a bow placed there. If the player is unsuccessful, the wood is also removed from his inventory and it is determined whether the player's skill at carpentry was improved, which is done by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

Butchery

Checks to see that the player is holding a carcass and if he has enough fatigue points. If so, a skill check is performed to determine if the player was able to butcher the carcass. If he is unsuccessful, he is given a chance to improve his skill at butchery, but the carcass is removed from his inventory. If he is successful, meat is added to his inventory, and he is given a chance to skin the carcass. If he chooses to skin the carcass, a fatigue check and a skill check are performed. If he is successful, a fur pelt is added to his inventory. Otherwise, he is given a chance to improve his skill at skinning, which is done by calling the Improve_Skill function. The carcass is then removed from his inventory. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

Camping

First checks to see if player can camp at his current location. If he can camp there and he successfully passes a skill check, the player's picture is removed from the map and replaced by a tent. Then a menu appears with options for resting, building a fire, and cooking. If the player fails the skill check, he is given a chance to improve his skill, which is done by calling the Improve_Skill function. Although in reality a person expends fatigue when setting up a camp, we discovered that if we require fatigue points when camping, a player may end up with no fatigue points left and no way to recover them, since he would be unable to camp and then rest. Therefore, we made camping a skill that requires no fatigue points.

The camping menu:
CancelButton_Click
Returns the player to the game forms and sets the disk options menu to not visible.

CancelCarpentryButton_Click
Sets the carpentry menu to not visible.

CancelCarvingButton_Click
Sets the carving menu to not visible.

CancelForageButton_Click
Sets the foraging menu to not visible.

Carpentry
Checks to see if the player is holding some wood, then displays a menu with options for crafting several different items.

Carving
Checks to see if the player is holding some wood, then displays a menu with options for carving several different items.

ChairCommand_Click
Sets the carpentry menu to not visible, checks to see if the player has enough fatigue, then makes a skill check to see if the player successfully crafted a chair. If so, the wood is removed from the player’s inventory and replaced with a chair. If the player is unsuccessful, the wood is removed and the player is given a chance to improve his skill at carpentry, which is done by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

Character_Click
Character picture is removed and the statistics label is shown. All player statistics are updated to reflect current status.
Here is the Statistics Label:
See Statistics_GetFocus for reversal.

**Chopping**

First checks to see if the player's location has trees, then checks to see if the player has enough fatigue points, then checks to see if the player is wielding a bladed weapon. That satisfied, the tree will always be successfully chopped down, but the time that passes will depend on the player's skill. Then, if a player does not succeed in passing a skill check, he is given the chance to improve his skill at chopping, which is done by calling the `Improve_Skill` function.

Here is the message that is displayed when the player is finished chopping down the tree:

![Message: You have finished chopping down the tree.](image)

**CookCommand_Click**

Calls Cooking when the player clicks on the cook button.

**Cooking**

Checks to see if the player knows both firebuilding and camping, then checks to see if the player is holding meat and if he has enough fatigue points. After successfully passing a skill check, the meat is removed and replaced by some dinner. If the skill check is not passed, the meat is removed from the player's inventory and he is given a chance to improve his skill at cooking, which is done by calling the `Improve_Skill`
function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**Disk_Click**

Makes the disk options menu visible.

**Equipment_Click**

Determines the type of item that was selected and acts appropriately. If the item is food, it is removed from the player’s inventory and adds 2 points each to the player’s fatigue and hit points. If the item is clothing or armor, the correct body location is set to that item. **Check_Armor** is then called to recalculate the player’s armor rating and evade. If the item is a weapon, then the player’s wielded flag is set to that item.

**Eyeball_Click**

Changes the State to “eye” if this same button was not previously pushed. Else it cancels the last click and returns State to normal. Also changes the mousepointer to the appropriate icon.

**Fire_Building**

First checks to see if the player is at a location suitable for building a fire and if he has enough fatigue points, then does a skill check with an appropriate modifier for the terrain. If the player is on any terrain with snow, there is a negative fifteen modifier to his fire building rating, meaning that fifteen is subtracted from his rating when comparing it to a random number for a skill check. If the player is on swamp terrain, there is a negative ten modifier. If the player is on any other terrain, there is no modifier. If the player fails a skill check, he is given a chance to improve his skill, which is done by calling the **Improve_Skill** function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**Fishing**

First checks to make sure that the player is standing next to some kind of water. Then checks to see if the player is holding a fishing pole and has enough fatigue points.
If all of these cases are true, then a skill check is performed to see if the player successfully catches a fish. If so, the fish is added to the player’s inventory and a messagebox is displayed telling him what has occurred. Otherwise, the player is given a chance to improve his skill, by calling the `Improve_Skill` function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**FishingPoleCommand_Click**

Determines if the player has enough fatigue points to carve a fishing pole and if he is holding some wood, then performs a skill check to determine whether the player successfully carved a fishing pole. If he is successful, the wood is removed from his inventory and a fishing pole placed there. If the player was unsuccessful, the wood is removed from his inventory and he is given a chance to improve his skill by calling the `Improve_Skill` function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**Foraging**

First determines if the player is in an area that allows foraging. If the player can forage in the area, then the forage menu is displayed. Otherwise, a messagebox appears telling him that he can not forage in this area.

The forage menu:

```
-Forage
What would you like to forage for?
```

**Form_Load**

This is the main function for this form. It is also the very first function that is called when the game is run. It initializes the State variable to normal and calls Initial_Setup to set up the form. Then the game map is set up. Calls FirstScreen. Loads the current player to the cur_x and cur_y coordinates. Loads the player’s picture to the Character image box. Sets the value of the health progress bar to the player’s
FruitButton_Click
Sets the forage menu to not visible. Checks to see if the player has enough
fatigue points. A skill check is performed with a plus thirty modifier to the player’s
foraging skill rating. If he is successful, some fruit is added to the player’s inventory. If
he is unsuccessful, he is given a chance to improve his skill by calling the Improve_Skill
function. Whether or not the player was successful, fatigue points are subtracted and time
passes, although to a different degree.

Hand_Click
Changes the State to “hand” if this same button was not previously pushed. Else
it cancels the last click and returns State to normal. Also changes the mousepointer to the
appropriate icon.

HerbsButton_Click
First, the forage menu is set to not visible. Then a check is made to see if the
player has enough fatigue points. Then a skill check is performed with a plus ten
modifier to the player’s foraging skill to see if the player was successful. If the player is
successful, some herbs are added to his inventory. Otherwise, he is given a chance to
improve his skill by calling the Improve_Skill function. Whether or not the player was
successful, fatigue points are subtracted and time passes, although to a different degree.

Hunting
Several checks are made within this function to determine what type of terrain the
player is currently on, which results in different actions. First, if the player is standing on
a bridge, a messagebox appears telling the player that he can not hunt while on a bridge.
If the player is on grasslands, then a skill check is performed with a negative five
modifier to the player’s hunting skill rating. If the player is on hills, a skill check is
performed with a negative twenty modifier to the player’s skill rating. If the player if on
swamps, a skill check is performed with a negative twenty-five modifier to the player’s
skill rating. If the player is on trees, a skill check is performed with a plus five modifier.
If the player is on desert, a skill check is performed with a negative forty modifier.
If the player is on snowy grasslands, a skill check is performed with a negative ten modifier. If
the player is on snowy hills, a skill check is performed with a negative twenty-five
modifier. If the player is on snowy trees, then a skill check is performed with no
modifier to the player’s hunting skill rating. If the player is on any other terrain, a
messagebox appears telling him he cannot hunt on this terrain. Also, there is a check to
see if the player has enough fatigue points. If the player was successful in the skill check,
acarcass is added to his inventory. Otherwise, the player is given a chance to improve
his skill, which is done by calling the Improve_Skill function. Whether or not the player
was successful, fatigue points are subtracted and time passes, although to a different
degree.

Identify_Item
An item is passed to this function, and an appropriate message is outputted,
depending on the type of item and the player’s skill at item lore. This means that if the
player’s item lore skill rating is high, a more descriptive message is given. If his rating is low, a vague message will be given.

**Improve_Skill**

This function obtains a random number, then compares the number to twenty. If the number is below twenty, then the player has improved at the skill and a one is returned, where it will then be added to the player’s skill. Otherwise, a zero is returned. This is used so that a player will only improve in his skill twenty percent of the time he fails.

**Initial_Setup**

Sets top, left, height, and width variables for all visible items on the game form, including: MapGrid (picture box), Character (image), Health (progress bar), Fatigue (progress bar), Equipment (combo box), Skills (combo box), Eyeball (command button), Hand (command button), Mouth (command button), Attack (command button), Disk (command button), TimeClock (picture box), and MoveButtons(command buttons).

**Initialize_Characters**

This function reads in all of the character information into an array, so it can be quickly accessed at any time using an array index.

**Learn_Skill**

This function differs from the **Improve_Skill** function because it is used when a player is improving their skill by learning from a character, rather than improving once he has failed. As a result, he improves more rapidly when learning from a character. Also, this function incorporates the player’s intelligence rating to give a bonus.

**MapGrid_Click**

Checks the **State** variable and performs task based on the value of **State**. If **State** is “normal” nothing is done.

If **State** is “eye”, it will determine what is being pointed to, then display a message box stating what the object is by calling the **Identify_Item** subroutine.

If **State** is “mouth”, it will determine what is being pointed to, then determine whether it is a character that can be spoken to. If the player can talk to the character, a speechbox will appear containing information that may be useful to the player’s quest. Otherwise an error message will appear telling the player that this is not an object that can be spoken to.

If **State** is “hand”, it will determine what is being pointed to, then determine if the player can get the item. If the player can get the item, it is placed in the player’s inventory. Otherwise, an error is displayed telling the player that they cannot get this item. If the player has selected an item in his inventory, that item is removed from the inventory.

If **State** is “attack”, it will determine what is being pointed to, then determine if the character can attack the monster. If the player can attack, the **AttackForm** will appear where the attack will commence. If the player cannot attack, then an error message will be displayed telling the player that they cannot attack this item.
If State is "begging", it will determine whether the player is pointing to someone who can be begged from. If so, the Begging subroutine on the TownForm is called. Otherwise, an error message is displayed telling the player that there is no one there to pickpocket.

If State is "pickpocketing", it will determine whether the player is pointing to someone who can be pickpocketed. If so, the PickPocketing subroutine on the TownForm is called. Otherwise, an error message is displayed telling the player that there is no one there to pickpocket.

After every click on the mapgrid, the mousepointer is returned to normal.

**Mouth_Click**
Changes the State to "mouth" if this same button was not previously pushed. Else it cancels the last click and returns State to normal. Also changes the mousepointer to the appropriate icon.

**PassTime**
Receives an number specifying how many minutes are to pass and increments the TimeClock minutes by that number.

**Quit_Button**
When the quit button is clicked, this subroutine is called and each of the existing forms is unloaded.

**RandomNumber**
This is a short function that returns a number between one and one hundred, using the function built into Visual Basic that returns a number between zero and one.

**Remove**
This subroutine removes an item from both the equipment combo box on the form and the player’s inventory. It cycles through the player’s inventory to remove the empty slot so that the equipment box list and the inventory remain exactly the same.

**Rest**
This subroutine receives a number specifying the number of hours that the player is resting. It calls PassTime for that number times sixty to pass the appropriate number of minutes. Then it adds an amount onto the player’s fatigue and hitpoints and updates the progress bars on the form.

**RestCommand_Click**
Removes the camping menu from the screen and displays the rest menu.
The rest menu:
Rest

How many hours would you like to rest?

RestOkayCommand_Click

Removes the rest menu from the screen and calls Rest with the number of hours that the player specified. Displays the camping menu.

RootsButton_Click

First, the forage menu is set to not visible. Then a check is made to see if the player has enough fatigue points. Then a skill check is performed to see if the player was successful. If the player is successful, some roots are added to his inventory. Otherwise, he is given a chance to improve his skill by calling the Improve_Skill function. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

Show_MessageBox

This subroutine receives an image and a string as parameters. It sets the messagebox's image to the image received and the messagebox's label to the string it received. Then it displays the messagebox.

Show_SpeechBox

This subroutine receives an image and a string as parameters. It sets the speechbox's image to the image received and the speechbox's label to the string it received. Then it displays the speechbox.

Here is a sample of a visible speechbox:

I didn't see any men. Why don't you follow all the young men to the south. Maybe they can tell you something.
**SaveButton_Click**

When this button is clicked, the player’s information is saved.

**Skills_Click**

When an item in the skills combo box is selected, this subroutine is called. It determines which skill was selected and then calls the appropriate subroutine.

**Skinning**

Checks to see that the player is holding a carcass and if he has enough fatigue points. If so, a skill check is performed to determine if the player was able to skin the carcass. If he is unsuccessful, he is given a chance to improve his skill at skinning, but the carcass is removed from his inventory. If he is successful, a fur pelt is added to his inventory, and he is given a chance to butcher the carcass. If he chooses to butcher the carcass, a fatigue check and a skill check are performed. If he is successful, some meat is added to his inventory. Otherwise, he is given a chance to improve his skill at butchery, which is done by calling the `Improve_Skill` function. The carcass is then removed from his inventory. Whether or not the player was successful, fatigue points are subtracted and time passes, although to a different degree.

**StartFireCommand_Click**

Calls Fire_Building.

**Statistics_GotFocus**

Returns the statistics display to invisible and sets the character image to visible.

**StopCampingCommand_Click**

Removes the camping menu from the screen. Replaces the tent icon with the character’s icon. Sets the fire variable to false.

**Update_Skills**

Checks the player’s skill rating for a number of skills. The skills with a rating greater that zero are displayed in the skills combo box.

**Timer1_Timer**

Turns the timer off and calls `Initialize_Characters`.
WalkingStickCommand_Click

Removes the carving frame from the screen. Checks to see if the player has enough fatigue points to carve a walking stick. Then performs a skill check to see if the player was successful at carving a walking stick. If he was successful, the wood is removed from his inventory and a walking stick is added. Otherwise, the wood is removed and he is given a chance to improve his skill by calling the Improve_Skill function.

The message displayed when the player has successfully carved a walking stick:

You have successfully carved a walking stick.
**AttackForm**

The following is a function by function analysis of the attack form. All functions involved in combat, from fleeing to actually hitting for damage, are contained within this form.

A picture of the **AttackForm**:

![AttackForm Image]

**Archery**

This function is used to determine the proficiency with the bow, the only ranged weapon in the game. The better the player is with the bow, the better the evade and hit bonuses the player receives for using that weapon are.

**ArmorLore**

**ArmorLore** is called at the beginning of the combat sequence. This checks the player’s proficiency at wearing armor. Depending at the level of proficiency, the player receives a bonus to evade. If the check is failed, then the evade bonus is not applied and the skill is checked to see if it improves.

**Attack**

The attack function runs the combat for the player. It first checks to see if the player hits the creature. If the hit is failed, then the player’s turn is over and the combat progresses onward. If the player does manage to beat the evade number of his opponent, then the player succeeds in hitting the monster and damage calculations begin. Damage is divided into three levels: light, medium, and heavy. The amount of damage done depends on bonuses to damage, the weapon wielded and the level of attack chosen. A light attack does half the damage of a medium attack while a heavy attack does approximately one and a half times as much damage as the medium attack. The
monster's armor is then used to soak a percentage of the damage equal to the armor rating of the monster. If the monster is killed control is passed to the \textit{Killed} function.

\textbf{AttackButton\_Click}

\textit{AttackButton\_Click} is the decision by the player to attack the monster. Chosen during the player's turn, it then activates the attack level submenu so the player can choose the level of attack he wishes to execute on the monster.

\textbf{BlindFighting}

At night, it is a little tougher to fight. \textit{BlindFighting} negates the penalties the player suffers while fighting in the dark, providing a distinct advantage over the monsters that do not get the same advantages.

\textbf{BluntWeaponsFighting}

\textit{BluntWeaponsFighting} determines the mastery of the use of weapons like maces. If the player uses such a weapon, then this skill check is performed. A successful check allows the player benefits to the number rolled while trying to hit the monster and the amount of damage done.

\textbf{CheckMon\_Flee}

\textit{CheckMon\_Flee} is the heart of the monster's "artificial intelligence". It is from this function that monster makes its decisions as to what it will do during a round of play.
The **LightAttack_Click** button enables the player to select the light level attack. It is passed control from the player options menu by clicking on the attack button there. It then passes control to **CheckFatigue** to ensure the player has enough fatigue to expend to perform the attack.

The **MediumAttack_Click** is one of the choices from the from the attack submenu. It selects the middle level of attack that spends average fatigue and does average damage. Control is then passed on to **CheckFatigue** to determine if the player has the Fatigue points left to execute this level of attack.

**MonFlee**

**MonFlee** executes the monsters attempt to escape. The function is called from **CheckMonFlee** when the monster's hit points have fallen below the **FleePercent** statistic contained in the monster's data file. The function allows a 50 percent chance of escape. If the monster successfully meets the escape check, the player can choose to pursue the monster or let it escape. If the chooses to pursue the monster the Pursue function is called, otherwise the monster makes its getaway and combat ends.

**MonPursue**

**MonPursue** governs the monster’s pursuit when the player tries to flee from combat. The function first checks the monster’s NoFlee number, A number set to determine how often the monster will attempt to prevent the player from fleeing. If the check is successful, the monster attempts to block the player’s attempts to escape. There is a 50 percent chance that the monster will block the escape. If the monster fails to meet the first check, or does not succeed in blocking the escape of the player, the player escapes and the combat cycle ends. There is a special case: if the monster’s no flee equals eleven, there is no possibility of escape from combat.

**MonsterAttack**

This function governs the monster’s non-special attacks. Like the player attack function, the first thing the monster function determines is if the monster hits in combat. If the monster connects successfully, it selects the level of attack based on the monster’s attack percentages. The function then determines the amount of damage done to the player and calculates whether or not the player has died (when a player reaches zero hit points, they are dead).

**MonsterSpecial**

The **MonsterSpecial** function governs the special attack process. This function is called from **CheckMonFlee** if the monster has a special. The percentage likelihood of the monster using its special is then compared to a randomly generated number to determine if the monster does use its special. If this check is passed, the correct special is called upon and then executed. If the check fails, then the monster attack function is called and the monster takes its regular physical attack.
**Parry**

Parry is a defensive form of combat that allows for the player to mount an occasional counteroffensive if the monster happens to miss its attack. This counter has no chance of missing if successful. The damage done by the strike is moderate.

**ParryCommand_Click**

This function activates the parry skill. It is called from the skills submenu and calls the Parry function described above.

**PlayerOption**

PlayerOption allows the player to choose actions during combat. The function enables the player’s option menu when the player has a turn in combat. The menu allows from the selection of four options: attack, defend, skills, and flee. Attack and skills provide ways of doing damage. Defend makes you harder to hit. Fleeing is the way the player can escape from battle. The player can choose any of these buttons and use them at any time during combat. After the selection is made, the appropriate functions are called.

**Punch**

This function executes the punch command when it is called from the skills submenu. The function first checks the amount of fatigue to ensure that there is enough to execute the attack. If there is not enough, the function returns to PlayerOption to allow for another choice of actions. If the player has enough fatigue, the punching action continues by performing a check to determine if the player connects with the attack. If the player hits the monster with the punch, damage is then calculated and the appropriate amount of fatigue is subtracted. In either a hit or miss case, the player’s turn is increased by one and CheckTurn is called.

**PunchCommand_Click**

This command calls the Punch function. It is activated when the skills button is chosen. If the skill is unlearned, then the skill button cannot be used.

**Pursue**

The Pursue function is called from MonFlee if the player has chosen to block the monster attempting to flee. The player has approximately a 50 percent chance of blocking the monster, which allows the player to continue the fight. If this check fails, then the monster escapes and the combat ends.

**RandomNumber**

This function is the heart and soul of the combat system. It is from here that the number for every check and damage determination is created. This is done by seeding the random number generator with the system clock and then random numbers are generated whenever needed.

**SkillButton_Click**
This function is used when the skills option from the player’s options is selected. When selected, it brings up the skills submenu. This allows the player to use any of the selected combat skills in battle. If the skills are unlearned, they are grayed out. This makes them unusable. For this reason, a cancel button was added. This returns you to the main options window.

**ColdBreath**

*ColdBreath*, like its twin *FireBreath*, is one of several special procedures that certain monsters use this form of direct attack. This attack, when it hits, does from 3-15 points of direct damage to the player. Armor soaks no damage from a breath-based attack. Also, the player’s chances to avoid the attack are lessened due to the wide area that the breath attack covers.

**FireBreath**

*FireBreath* is the identical twin of *ColdBreath*. It also is a wide area attack, which reduces the player’s chance of evading the attack. Also, like *ColdBreath*, the damage is done directly to the player and no chance of the armor soaking the damage is included.

**Paralyze**

Paralyze is another special procedure used by certain monsters. This attack deals no direct damage to the player. However, a successful Paralyze attack puts the player in stasis for one to four rounds of combat action. This allows the monster a flurry of free attacks while the player sits helpless.

**Hamstring**

Hamstring is the devious attack of the brigand. This special not only deals damage to the player, but also, reduces efficiency in a fight. The hamstring special deals a little damage but also robs the player of evade points.

**Haymaker**

A Haymaker is a massive physical attack used by some monsters to deal out massive amounts of damage. It is a powerful punch, which can knock the player off balance for a turn when it connects. This is a very effective weapon for the monsters that possess this attack.

**Poison**

Most commonly used by some wild animals, Poison inflicts damage continually. If a monster poisons a player, the player’s Poisoned flag is set to true. From that point until the end of the day, during any attack, the player takes damage on even numbered rounds due to the poison burning in the player's blood.
TownForm

This form is used to represent the several towns in the game that the player can enter. Here the player can talk to characters, buy and sell items, and rest.

A picture of the TownForm:

![TownForm Image]

Global Variables:
String TownState to represent what state the player is in, for example, talking, begging, identify, etc.
Item array BuyItemArray(30) to represent the items available to buy at this particular town.
Integer Index to keep track of the next open position in the BuyItemArray

Add
Receipts as a parameter an item, then adds it to the current player's inventory and the equipment combo box on both the TownForm and the GameForm.

AddBuyItem
Adds an item to the array of available items in the town and displays the item in the combo box in the BuyFrame.

Appraising
First checks to see if the player has selected anything, then calculates a value for the item and displays it in a message. Also, gives the player a chance to improve his skill by calling the GameForm.Improve_Skill function.

Begging
First checks to see if the player can still beg this day (can only beg three times a
day), then checks to see if the player has enough fatigue to beg. Calculates a modifier
based on the player’s social skill rating which is added to the player’s skill rating when
performing a skill check. If the player successfully passes the skill check, then the player
receives either bread or money for his efforts, depending on his skill rating. If the player
is unsuccessful, he is given a chance to improve his skill by calling the

GameForm.Improve_Skill function. Whether or not he was successful, time passes and
points are subtracted from his fatigue.

**BuyCommand_Click**

First checks to see if the player has selected an item. If so, the cost of the item is
determined, and a discount is calculated based on the player’s commerce and barter skill
ratings. Then a check is made to see if the player has enough money and if so, the item is
placed in his inventory. Otherwise an appropriate message is displayed.

**CancelItemCommand_Click**

Sets the BuyFrame to not visible and returns the TownState to “normal”.

**CancelSellCommand_Click**

Sets the SellFrame to not visible and returns the TownState to “normal”.

**Character_Click**

This is a very long function that handles many different options. It first
determines what the TownState is, i.e. “talk”, “identify”, “begging”, etc. Then it
determines what the CurrentTown is, and based on this can determine which character
was clicked on. Then an appropriate function is called, i.e. Identify or Begging. At the
end of the function, the TownState is returned to “normal” and the mousepointer is set
back to the default.

**ExitCommand_Click**

Resets the CurrentTown variable to -1 and unloads the TownForm.

**Form_Load**

This subroutine is more complicated than any of the other Form_Load functions
because this form is used for several different towns, each of which has a different setup.
This is where the pictures, labels, commands, and combo boxes are set up. Each one has
a left, top, height, and width variable, which can be set in this subroutine. Inside the large
picture box, 20 tiles are loaded which will contain a background display and possibly a
picture in the foreground. The player’s picture is also loaded here. Then a case statement
is used to determine which town is currently being loaded. Depending on which town is
being loaded, a different set of characters, inns, shops, and wells are loaded. At the end
of the subroutine, the TownState variable is set to “normal”, the skills combo box is
updated with the player’s available skills, and the items list is updated to reflect what the
player currently has in his inventory.

**Identify**
This subroutine receives as a parameter a Character. A check is made to determine the player’s ItemLore skill rating, and based on that, an appropriate message is displayed. If the player’s ItemLore rating is less than thirty, a vague message will be given. If the player’s ItemLore rating is greater than twenty-nine but less than sixty-five, a more descriptive message is given. If the player’s ItemLore rating is greater than sixty-four, the most descriptive message is given. Lastly, the mousepointer is set back to the default, and the TownState is set back to “normal”.

**IdentifyCommand_Click**

First checks to see if the TownState is already “identify”. If so, it returns the TownState to “normal” and sets the mousepointer back to the default. Otherwise, it changes the TownState to “identify” and sets the mousepointer to an appropriate icon.

**Items_Click**

First makes sure that there is something selected in the items combo box. Then it checks to determine what the TownState is. If it is “eye” then the TownState is returned to “normal” and GameForm.Identify is called for the selected item. If it is “sell” then the SelectedItemLabel in the SellFrame is set to that item’s name. Otherwise, the item’s type is determined and an appropriate action taken. If the item is food, it is eaten. If the item is clothing or armor, it is worn, and if it is a weapon, it is wielded.

**Performance**

First a check is made to see if the player has already performed 3 times that day. If not, then a modifier is calculated based on the player’s social skill. After making sure that the player has enough fatigue, a skill check is made using the player’s performance skill rating plus the modifier. If the skill check is passed, a check is made to determine what range the player’s performance skill is in. If the player’s performance rating is less than 30, then the player will receive some bread, which is placed in his inventory after a message is displayed. If the player’s performance rating is greater than thirty, then the player will receive some coins which are added to the player’s money after a message is displayed. However, the player has a chance of receiving more coins if his rating is above sixty. A half-hour of game time is passed, fatigue is subtracted from the player’s fatigue rating, and the player’s performed value is incremented.

If the player fails the skill check, twenty minutes of game time is passed, fatigue is subtracted from the player’s fatigue rating, and a message is displayed telling him that he was unsuccessful at performing. Then the player is given a chance to improve his performance and social skill ratings by calling the GameForm.Improve_Skill function.

**Remove**

Received as a parameter an item, which is removed from the currentplayer’s inventory and the equipment combo box on both the TownForm and the GameForm. Then the inventory is cycled through to remove the empty slot, keeping the inventory an exact copy of the equipment combo boxes.

**PickPocketing**
This subroutine first checks to see if the player has enough fatigue. Then a modifier is calculated if the character is one of several special characters that would react differently to pickpocketing. Then a skill check is performed using the player’s pickpocketing skill rating plus the modifier. If the player passes the skill check, then a check is made to see what range the player’s pickpocketing skill rating falls into. The higher the player’s skill rating, the more money he will be able to steal. If the player fails the skill check, a check is made to see if his social skill rating is above ninety. If so, he escapes. Otherwise, the town guard will attack him. Then, he is given a chance to improve his skill by calling the \textbf{GameForm.Improve\_Skill} function.

\textbf{RestCommand\_Click} \\
First passes twelve hours of game time. Then, increases the player’s fatigue and hit points and updates the progress bars accordingly.

\textbf{SellCommand\_Click} \\
Displays the \texttt{SellFrame} and sets the \texttt{TownState} to “sell”. \\
A picture of the \texttt{SellFrame}:

\begin{center}
\includegraphics[width=0.3\textwidth]{sell.png}
\end{center}

\textbf{SellItemCommand\_Click} \\
First checks to make sure that there is an item selected, then checks to make sure the selected item is not one of the special items. Then, an extra bonus is calculated using the player’s commerce and barter skill ratings. Next, a selling price is determined by considering the actual value of the item and the extra bonus previously calculated. Then the item is removed from the player’s inventory and the \texttt{Update\_Silver} subroutine is called.

\textbf{Show\_MessageBox} \\
Displays a message box with the image and text passed to it.

\textbf{Show\_SpeechBox} \\
Displays a speech box with the image and text passed to it.

\textbf{Skills\_Click} \\
Makes a check to see which skill was selected, then calls an appropriate function or sets the \texttt{TownState}.

\textbf{Update\_Silver}
Corrects the **SilverLabel** to reflect the current player’s money amount.

**Update Skills**
Checks to see what skills the player currently knows, then displays them in the skills combo box.

**TalkCommand_Click**
If the **TownState** is already set to “talk”, then the **TownState** is set back to normal and the mousepointer set to default. Otherwise, the **TownState** is set to “talk” and the mousepointer is set to an appropriate icon.
Module

This is a section of the code that is accessible to all forms in the game. Since variables can not easily be passed from one form to another, global variables can be kept here and the information accessed or changed from any of the forms.

**CurrentPlayer** --- holds the information for the currently active player

**ItemArray** --- list of every item available in the game

**CharacterArray** --- list of every character in the game

**CurrentMonsterNumber** --- the number of the monster currently being used in combat

**CurrentTown** --- an integer representing the town that the player is currently in

**PlayerType** --- a type used in saving and loading player information

**Pause** --- a subroutine that allows the game to pause for a given number of seconds
**Character Class**

This is a class used to hold information about the characters in the game. There are fields for the character's name, picture, description, and speech.

**Name** --- holds a short name for the character

**IDNumber** --- an index for the array contained in the game; allows the character information to be accessed

**Picture** --- contains the filename for the character's picture

**Description1** --- contains a short description of the character

**Description2** --- contains a longer description of the character

**Description3** --- contains a more detailed description of the character

**Speech** --- contains the message that the character will give to the player - a hint to where an item or further information is located

Also, in the class are get and let functions for each data field. These allow the data fields to be retrieved or set to a new value.
**Item Class**

Deciding how to handle item information was another big step. Since Visual Basic, our chosen programming language, does not handle classes and inheritance in the same manner and to the extent that C++ does, I had to revise the way I had originally intended to handle this information. I had wanted to create classes for armor, weapons, food, etc. which would all inherit from class item. Then, by creating the inventory and equipment lists as arrays of items, they could contain armor, equipment, or any combination of items.

However, being unfamiliar with classes in Visual Basic 5, and after reading that Visual Basic does not include inheritance that works in this manner, I decided to use a single class item with the following members:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td>IDNumber</td>
<td>Byte</td>
</tr>
<tr>
<td>Picture</td>
<td>String</td>
</tr>
<tr>
<td>Description1</td>
<td>String</td>
</tr>
<tr>
<td>Description2</td>
<td>String</td>
</tr>
<tr>
<td>Description3</td>
<td>String</td>
</tr>
<tr>
<td>Weight</td>
<td>Byte</td>
</tr>
<tr>
<td>Height</td>
<td>Byte</td>
</tr>
<tr>
<td>Wear-Position</td>
<td>Byte</td>
</tr>
<tr>
<td>Type</td>
<td>Byte</td>
</tr>
<tr>
<td>Special1</td>
<td>Byte</td>
</tr>
<tr>
<td>Special2</td>
<td>Integer</td>
</tr>
<tr>
<td>Value</td>
<td>Integer</td>
</tr>
</tbody>
</table>

**Name of Item**
Short string to describe what item is.

**IDNumber**
A number used for locating the item in the ItemArray index.

**Picture**
The file name of the image for this item.

**Description1**
A short description of the item for use by players with a very low item lore rating.

**Description2**
A slightly longer description of the item for use by player’s with a mid-range item lore rating.

**Description3**
A long description for player’s with a high item lore rating.
Weight
The weight of the item in stones.

Height
The height of the item in hands.

Wear-Position
The place on the player’s body that the item is worn. The following locations are available:

<table>
<thead>
<tr>
<th>Value</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Head</td>
</tr>
<tr>
<td>2</td>
<td>Body</td>
</tr>
<tr>
<td>3</td>
<td>Feet</td>
</tr>
<tr>
<td>4</td>
<td>About Body</td>
</tr>
<tr>
<td>5</td>
<td>Wielded</td>
</tr>
<tr>
<td>6</td>
<td>Shield</td>
</tr>
</tbody>
</table>

Type
The type of the item. The following types are available:

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sword</td>
</tr>
<tr>
<td>2</td>
<td>Piercing Weapon</td>
</tr>
<tr>
<td>3</td>
<td>Blunt Weapon</td>
</tr>
<tr>
<td>4</td>
<td>Axe</td>
</tr>
<tr>
<td>5</td>
<td>Ranged Weapon</td>
</tr>
<tr>
<td>6</td>
<td>Armor</td>
</tr>
<tr>
<td>7</td>
<td>Clothing</td>
</tr>
<tr>
<td>8</td>
<td>Food</td>
</tr>
<tr>
<td>9</td>
<td>Light Source</td>
</tr>
<tr>
<td>10</td>
<td>Other</td>
</tr>
</tbody>
</table>

Value
The value of the item in silver pieces.
Monster Class

While working on the monster class, it was important to consider all of the possible things that a monster might do during combat. We had to think when the monster might attack, how it might attack, how strong it would be and when it would flee. The monster class is our attempt to take all these variables into consideration and use them to serve as a template for all of the monsters in the game.

<table>
<thead>
<tr>
<th>Name</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_HitPoints</td>
<td>Integer</td>
</tr>
<tr>
<td>Current_HitPoints</td>
<td>Integer</td>
</tr>
<tr>
<td>Fatigue</td>
<td>Byte</td>
</tr>
<tr>
<td>Picture</td>
<td>String</td>
</tr>
<tr>
<td>Armor</td>
<td>Byte</td>
</tr>
<tr>
<td>Evade</td>
<td>Byte</td>
</tr>
<tr>
<td>Damage</td>
<td>Byte</td>
</tr>
<tr>
<td>DamageDice</td>
<td>Byte</td>
</tr>
<tr>
<td>ToHit</td>
<td>Byte</td>
</tr>
<tr>
<td>ToDam</td>
<td>Byte</td>
</tr>
<tr>
<td>LightAttackPercent</td>
<td>Byte</td>
</tr>
<tr>
<td>MediumAttackPercent</td>
<td>Byte</td>
</tr>
<tr>
<td>FleePercent</td>
<td>Byte</td>
</tr>
<tr>
<td>NoFlee</td>
<td>Byte</td>
</tr>
<tr>
<td>HasSpecial</td>
<td>Boolean</td>
</tr>
<tr>
<td>SpecialAttackPercent</td>
<td>Byte</td>
</tr>
<tr>
<td>Special</td>
<td>Byte</td>
</tr>
<tr>
<td>Drift</td>
<td>Integer</td>
</tr>
<tr>
<td>Level</td>
<td>Byte</td>
</tr>
<tr>
<td>MonsterType</td>
<td>Byte</td>
</tr>
</tbody>
</table>

Name

This category contains the monster’s name. It is saved as a string of characters.

Total_HitPoints

This is the total amount of life force that the monster possesses. This number helps in performing the calculations for determining if a monster will attempt to flee.

Current_HitPoints

Used to determine the amount of damage the monster has taken. Current_HitPoints starts out equal to the Total_Hitpoints statistic. As the monster takes damage, the damage is subtracted from the Current_HitPoints. The Current_Hitpoints, when compared to the Total_HitPoints, determines the if the monster will attempt to flee if it is less than the FleePercent.
Fatigue
The original intent was for the monster to use fatigue in the same way that it is by
the players. However, due to time constraints, this statistic was not used.

Picture
This contains the file name of the picture of the monster. This is the picture that
is used on the combat screen as the player fights the monster.

Armor
The monster’s armor rating works in the same way as the player’s armor rating
works. It absorbs a percentage of the damage done to the monster equal to the armor
rating.

Evade
The evade value determines how hard the monster is to hit. If the player’s hit
number is lower than the monster’s evade, then the player misses the monster. So, the
higher this statistic, the tougher the monster is.

Damage
The damage is the number that a random number is divided by as to provide a
remainder between zero and that number. By adding one, the amount of damage rolled is
halfway determined.

DamageDice
DamageDice is the second part used in determining the amount of damage dealt
by the monster. The number received after Damage is calculated multiplies
DamageDice. The ToDam bonus is then added. This number is then reduced by the
appropriate amount, determined by the amount of armor the player is wearing. The
remaining number is subtracted from the player’s hitpoints.

ToHit
ToHit is added to the number determining if the monster hits the player. It the
combination of the number and the ToHit is greater than the player’s modified evade
score, the monster hits the player. Otherwise the monster misses.

ToDam
ToDam is a damage bonus added on to the damage total after all of the other
calculations have been done. This bonus damage provides the monster with a constant
level of damage for the monster’s otherwise very random attacks.

LightAttackPercent
The LightAttackPercent determines how often, on average, the monster will use a
light attack. This number, with the MediumAttackPercent, determines how often the
monster will use a heavy attack.

MediumAttackPercent
This number determines the likelihood of the monster mounting a medium level attack. Added to the LightAttackPercent, it provides the percentage of the time a heavy attack is used.

**FleePercent**

This number is an integral part of determining at which point the monster will flee. This number is compared to the formula which involves Total_HitPoints and Current_HitPoints. If the number is less than the FleePercent, then the monster tries to escape.

**NoFlee**

This value reflects the amount of the time the monster will allow the player to try and escape. The values range from 0-11 a 0 means that the monster will never attempt to block your fleeing while a 10 means the monster will try very hard to stop you. An 11 is a special value that means the monster will not let you escape under any circumstances.

**HasSpecial**

This is a Boolean value. If the value is true, the monster has a special attack it can use. If it is false, the monster has no special attack.

**SpecialAttackPercent**

This number represents the percent of the time that the monster will use its special if it has one. If the monster has no special, this number equals zero.

**Special**

This number indicates the special that the player has. A 0 would be FireBreath. 1 equals Paralyze. A 2 is The Haymaker special. Hamstring is 3. Poison equals 4. Finally, 5 is ColdBreath.

**Drift**

Drift fell victim to a lack of time. Had there been more time, Drift would have caused the monster’s attacks to become heavier or lighter after the monster had suffered more than a 50 percent hit point loss.

**Level**

The level of the monster is determined by the toughness of the monster. If the monster’s level is high, its likelihood of being encountered is less.

**MonsterType**

There are four types of monsters in the game: animals (0), humanoids (1), giants (2), and monsters (3). Every monster created falls into one of these classes and is assigned the appropriate value.
**Player Class**

When designing the player class, we had to add a large number of elements to keep track of the many different variables. The following is a list of most of the variables we included.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Name</td>
<td>player’s name</td>
</tr>
<tr>
<td>Byte</td>
<td>Age</td>
<td>player’s age starting at 18</td>
</tr>
<tr>
<td>Boolean</td>
<td>Sex</td>
<td>male or female</td>
</tr>
<tr>
<td>Byte</td>
<td>Height</td>
<td>defaults set for male and female</td>
</tr>
<tr>
<td>Byte</td>
<td>Weight</td>
<td>defaults set for male and female</td>
</tr>
<tr>
<td>Integer</td>
<td>Money</td>
<td>amount of silver pieces</td>
</tr>
<tr>
<td>Byte</td>
<td>HitPoints</td>
<td>amount of hit points player currently has</td>
</tr>
<tr>
<td>Byte</td>
<td>Max_Hp</td>
<td>maximum amount of hit points the player can have</td>
</tr>
<tr>
<td>Byte</td>
<td>Fatigue</td>
<td>amount of fatigue points player currently has</td>
</tr>
<tr>
<td>Byte</td>
<td>Max_Fp</td>
<td>maximum amount of fatigue points the player can have</td>
</tr>
<tr>
<td>Byte</td>
<td>Armor</td>
<td>percent of damage player can soak from a monster’s attack</td>
</tr>
<tr>
<td>Byte</td>
<td>Evade</td>
<td>ability to avoid monster’s attacks</td>
</tr>
<tr>
<td>Byte</td>
<td>Agility</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Byte</td>
<td>Strength</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Byte</td>
<td>Fitness</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Byte</td>
<td>Health</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Byte</td>
<td>Intelligence</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Byte</td>
<td>Personality</td>
<td>8 to 30</td>
</tr>
<tr>
<td>Item[100]</td>
<td>Inventory</td>
<td>list of items the player currently has</td>
</tr>
<tr>
<td>Byte</td>
<td>NextIndex</td>
<td>next available index in the item array</td>
</tr>
<tr>
<td>String</td>
<td>Picture</td>
<td>filename of the player’s chosen picture</td>
</tr>
<tr>
<td>Boolean</td>
<td>Parry</td>
<td>true when player chooses parry from the skills list, false after monster’s turn</td>
</tr>
<tr>
<td>Item</td>
<td>Body</td>
<td>item currently worn on the player’s body</td>
</tr>
<tr>
<td>Item</td>
<td>AboutBody</td>
<td>item currently worn about the player’s body</td>
</tr>
<tr>
<td>Item</td>
<td>Feet</td>
<td>item currently worn on the player’s feet</td>
</tr>
<tr>
<td>Item</td>
<td>Wielded</td>
<td>item currently wielded by the player</td>
</tr>
<tr>
<td>Item</td>
<td>Shield</td>
<td>item currently used as a shield</td>
</tr>
<tr>
<td>Item</td>
<td>Head</td>
<td>item currently worn on the player’s head</td>
</tr>
<tr>
<td>Integer</td>
<td>DamageBonus</td>
<td>bonus to the amount of damage done by the player to the monster</td>
</tr>
<tr>
<td>Integer</td>
<td>HitBonus</td>
<td>bonus to the check to see if the player hits the monster</td>
</tr>
<tr>
<td>Boolean</td>
<td>Poisoned</td>
<td>whether or not the player is currently poisoned</td>
</tr>
<tr>
<td>Boolean</td>
<td>Bandaged</td>
<td>whether or not the player has bandaged today</td>
</tr>
</tbody>
</table>
Byte Performed how many times the player has performed today
Byte Begged how many times the player has begged today

Also included were byte values for each of the player’s skill ratings, which were too numerous and self-explanatory to mention here. See the player class code to view the specific skills.

A set and let function is included for each of the data fields, which allow the data fields to be accessed and altered.

**Player Attributes**

There are six attributes in the game: strength, agility, intelligence, personality, health, and fitness. These attributes affect different aspects of the player’s abilities.

**Strength**, the physical strength of the player, improves the player’s ability to cause damage to monsters.

**Agility** is how nimble the player is. It affects the player’s ability to hit monsters. It also decreases the likelihood that a monster during combat will hit the player.

**Health** determines how healthy the player is. This is reflected in the total number of hit points that the player has.

**Fitness** indicates how rugged the player is and how physically fit the player is. It is a representation of the amount of physical exertion the player can endure. For this reason, the player’s fatigue point total is determined by fitness.

**Intelligence** is how smart the player is. It influences how well a player is able to learn a new skill.

**Personality** indicates how well liked or hated the player is. It provides modifiers for actions involved in dealing with other people.
Message

This is a user control designed to display an image and a message. It is used when a player acquires an item or performs a skill.

Property Let LabelText
This function sets the label on the user control to the string that is passed to it.

Property Let MessagePicture
This function sets the image on the user control to the picture that is passed to it.
SpeechBox

This is a user control designed to display a picture and a message. It is used when the player is talking to a character in the game. A separate user control was created for this purpose so that the font size could be smaller, due to the length of the character's messages.

Property Let LabelText
This function sets the label on the user control to the string that is passed to it.

Property Let MessagePicture
This function sets the picture on the user control to the picture that is passed to it.
**StatisticsWindow**

This is a user control designed to display player information, such as the player’s name, age, and money.

**Property Let PlayerName**
Sets the name label on the user control to the string sent to it.

**Property Let PlayerAge**
Sets the age label on the user control to the number sent to it.

**Property Let PlayerHeight**
Sets the height label on the user control to the number sent to it.

**Property Let PlayerWeight**
Sets the weight label on the user control to the number sent to it.

**Property Let PlayerMoney**
Sets the money label on the user control to the number sent to it.

**Property Let PlayerEvade**
Sets the evade label on the user control to the number sent to it.

**Property Let PlayerArmor**
Sets the armor label on the user control to the number sent to it.

**Property Let PlayerAgility**
Sets the agility label on the user control to the number sent to it.

**Property Let PlayerStrength**
Sets the strength label on the user control to the number sent to it.

**Property Let PlayerFitness**
Sets the fitness label on the user control to the number sent to it.

**Property Let PlayerHealth**
Sets the health label on the user control to the number sent to it.

**Property Let PlayerIntelligence**
Sets the intelligence label on the user control to the number sent to it.

**Property Let PlayerPersonality**
Sets the personality label on the user control to the number sent to it.

**Property Let BackGroundColor**
Sets the background color on the user control to the color constant sent to it.
Appendix I

Introductory Story

By

Christina Jacobs

This story was written after the game was designed as an introduction for new players. The version contained in the actual game introduction was edited by Jeff Hanks and Karen Woznick.
Vale had been sleeping rather soundly before he heard the crash. After he heard the crash, he was wide awake, no question about that. He was attempting to get comfortable and go back to sleep when he heard another crash. He was beginning to think that maybe he should get up and check things out. If he was living over a tavern he might expect sounds like that in the middle of the night, but he was living in a monastery and since all the other students had gone home for the harvest festivals, he was the only one there apart from the monks. Needless to say, it was not a place that produced crashes in the middle of the night.

Vale got up and put on his tunic and boots. He started heading for the door when he heard the clank of armor in the hallway outside. Now that was really an odd thing. Monks do not wear armor. Vale thought perhaps Brother Villi had been sneaking ale again and that was the cause of all the racket. The clanking stopped outside his door and he heard some voices. Well, if it was Brother Villi, he had company. Vale thought better of opening the door and ducked behind the nearby worktable. A few moments later, the door to his room opened and three armor clad warriors were visible. Vale was glad he hadn't walked out into the hallway earlier. One of the warriors held his torch out so that he could see the room. All three of them grunted. It sounded as if they were frustrated or maybe angry. They headed down the hallway and Vale breathed a sigh of relief. They hadn't seen him. He wondered who they were. He decided to follow them and find out.

Vale went to his trunk and opened the lid. He pulled out his breeches, his belt pouch and silver and his dagger. He dressed carefully. He planned on following the warriors for as long as it took to discover who they were, so he needed to be prepared. Vale wished he had a sword or at least a knife bigger than his small dagger. He hoped the warriors had not taken everything from the armory downstairs nor everything from the kitchen. The monastery only had an armory because it gave the monks a way to employ the villagers in hard times. The village around the monastery was very small though, so they were usually able to survive on whatever farming they did. Last time Vale had heard, the village was down to about 40 families.

Vale crept out into the hallway and wondered how far ahead the warriors had gotten. He didn't want to pass them up and get caught later on. He had to make sure to stay behind them. He heard some clanking far ahead and knew then that they had gotten about halfway down the hall. He wondered if there was time to go and check on the Abbot before he followed the warriors. He decided he would attempt it because the Abbot might know who these invaders were and why they had come to the monastery.

Vale headed down to the opposite end of the hall and then up the steps that lead to the tower. When he entered the tower he saw some of the Abbot's belongings spilled out onto the steps. The warriors must have been here then! He rushed up the steps to try and find the Abbot. Vale finally found him behind a couch. It looked as if he had been thrown there. Vale got some coverlets and pillows to try and make him more comfortable. Once the Abbot seemed to be more comfortable, and Vale knew he was in no great danger of dying (at least not in the next few hours) he began to question him. Vale wanted to know who the warriors were and what they wanted and why they would attack a helpless monastery. The answer the Abbot gave him was very surprising.

"This monastery has always hidden a great secret. The secret is so great that I had thought I was the only one that knew of it. I am the only one that knows of it here, but the warriors that have come also know of it, and they know of its great power. It is a power
that they want for themselves. But we cannot let them have it! You must find what they are looking for before they do! You will have an advantage. I can tell you where it is hidden. But first listen. After you have this first piece, you must go and find the others. They are scattered all over the land. Very few people will know where to find them. There may be some pieces that no one will remember where they are hidden. You must find them anyway. There are seven different items and when they are all put together, they can triumph over all evil. The invaders that have come here already have one item and they are looking for the second one here. You need to find the items before they do and then get the final item from them when the proper time comes. Now, I will tell you where to look for the one that is hidden here. Go into the library and find the largest book that you can see. Behind this book there is a small box. Inside the box there is a key. Take the key to the kitchen. Find the box in the pantry that holds the bacon and move the box to the side. There is a very small door in the wall which you can unlock. When you open the small door you will find a map. It may be in pieces by now and some parts may be faded. But it tells where the other objects might be hidden. No one knows if it is accurate. Also, there will be a key. Take this key to the basement and go to the farthest, darkest corner. There you will find a door that will lead you into a tiny room. Inside the room you will find a sword. It looks like any other sword, it might even look like a very poor, almost ruined sword. But that is the first object you will have. Good luck, and remember you must stay ahead of the warriors!
Appendix II

The Code
All function on this form were written by Tim Zoch, unless otherwise noted.

Dim MonsterInit As Byte
Dim PlayerInit As Byte
Dim Evade As Byte
Dim AttackLevel As Byte
Dim Playerturn As Byte
Dim Monsterturn As Byte
Dim CurrentMonster As New Monster

Private Sub Archery()
'called by form load. If the player is wielding a bow, then
If CurrentPlayer.Wielded.ItemType = 5 Then
  'increase player's evade by 15
  Evade = Evade + 15
  'check the skill, if successful,
  If RandomNumber() <= CurrentPlayer.Archery Then
    If CurrentPlayer.Archery < 21 Then
      Evade = Evade + 5
    ElseIf CurrentPlayer.Archery > 20 And CurrentPlayer.Archery < 41 Then
      Evade = Evade + 7
    ElseIf CurrentPlayer.Archery > 40 And CurrentPlayer.Archery < 61 Then
      Evade = Evade + 10
    ElseIf CurrentPlayer.Archery > 60 And CurrentPlayer.Archery < 81 Then
      Evade = Evade + 13
    Else
      Evade = Evade + 15
    End If
  'if the check fails, try to improve the skill.
  Else
    CurrentPlayer.Archery = CurrentPlayer.Archery + GameForm.Improve_Skill
  End If
End If
End Sub

Private Sub Armorlore()
'benefits the player for knowing his armor.
'it is called from form load and performs a skill check
'if the check is successful, the player gets the bonuses
If RandomNumber <= CurrentPlayer.Armorlore Then
  If CurrentPlayer.Armorlore < 31 And CurrentPlayer.Armorlore > 15 Then
    Evade = Evade + 5
  ElseIf CurrentPlayer.Armorlore < 46 And CurrentPlayer.Armorlore > 30 Then
    Evade = Evade + 10
  ElseIf CurrentPlayer.Armorlore < 61 And CurrentPlayer.Armorlore > 45 Then
    Evade = Evade + 15
  ElseIf CurrentPlayer.Armorlore < 76 And CurrentPlayer.Armorlore > 60 Then
    Evade = Evade + 20
  ElseIf CurrentPlayer.Armorlore > 75 Then
    Evade = Evade + 25
  End If
  'if not, try to improve the skill
  Else
    CurrentPlayer.Armorlore = CurrentPlayer.Armorlore + GameForm.Improve_Skill
End If
End Sub

Private Sub Attack()
' check to see if player is wielding a bow, if he is try to subtract
'an arrow. if the player has no arrows, then call player option
If CurrentPlayer.Wielded.ItemType = 5 Then
    Dim Index As Byte
    For i = 0 To CurrentPlayer.Nextindex - 1
        If CurrentPlayer.Inventory(i).Name = "arrow" Then
            Index = i
        End If
    Next i
    If i = CurrentPlayer.Nextindex Then
        AttackMessages.Caption = " You have run out of arrows! "
        PlayerOption
        Exit Sub
    Else
        GameForm.Remove (Index)
    End If
End If

' Player's attack on the monster. It first determines if the player hits the monster
' and if not, closes out the player's turn. Otherwise, takes selected attack and
'determines the amount of damage done to the monster (and if the monster is dead)
If (RandomNumber() Mod 50 + 1 + CurrentPlayer.Hitbonus) < (CurrentMonster.Evade - (100 * Current
Monster.Fatigue / 200) * CurrentMonster.Evade) Then
    AttackMessages.Caption = " You miss the monster. "
    Pause (2)
    Playerturn = Playerturn + 1 'increment player turn
    CheckTurn 'check for who has next turn
Else
    Dim tempphp As Integer
    If AttackLevel = 1 Then
umber() Mod CurrentPlayer.Wielded.Special2) + 1) + CurrentPlayer.Damagebonus) / 2) * (100 - Cu
rrentMonster.Armor) / 100)
        If tempphp <= 0 Then 'if the monster has 0 or fewer hitpoints,
            MsgBox ("You have successfully killed the " + CurrentMonster.Name + ".")
            Killed 'then, he is dead, end combat.
        Else
            'otherwise, subtract from monster's hitpoints, and increment turns by one
            CurrentMonster.Current_HitPoints = tempphp
            MonsterHitPoints.Value = CurrentMonster.Current_HitPoints
            If CurrentPlayer.Wrestling > 0 Then
                If RandomNumber() <= CurrentPlayer.Wrestling Then
                    Monsterturn = Monsterturn + 2
                Else
                    CurrentPlayer.Wrestling = CurrentPlayer.Wrestling + GameForm.Improve_Skill
                End If
            End If
            PlayerFatiguePoints.Value = CurrentPlayer.Fatigue
            Playerturn = Playerturn + 1
    CheckTurn
    End If
ElseIf AttackLevel = 2 Then
Private Sub AttackButton_Click()
    'selects the attack option for the player's turn, sets values on all other
    'options to false, activates the attack sub-menu.
    AttackButton.Enabled = False
    SkillButton.Enabled = False
    FleeButton.Enabled = False
    ItemButton.Enabled = False
    DefendButton.Enabled = False
    LightAttack.Visible = True
    MediumAttack.Visible = True
    HeavyAttack.Visible = True
End Sub

Private Sub Blindfighting()
'first, check to see if it is night time
    Dim temp As Integer
    If GameForm.timeclock1.Hour > 20 And GameForm.timeclock1.Hour <= 5 Then
        'it is, then subtract the following from player and monster
        temp = Armor - 4
        If temp < 0 Then
            Armor = 0
        Else
            Armor = temp
        End If
        CurrentMonster.Armor = CurrentMonster.Armor - 4
        CurrentMonster.ToHit = CurrentMonster.ToHit - 4
        If CurrentPlayer.Blindfighting < 0 Then
            'if the player has blindfighting, then perform a skill check
            'if it succeeds, the losses for being in darkness are negated
            If RandomNumber() <= CurrentPlayer.Blindfighting Then
                Armor = Armor + 4
            Else
                'if it fails, the player tries to improve blindfighting
            End If
        End If
        End If
    End If
End Sub

Private Sub Bluntweaponfighting()
    'checks to see if the player is wielding a blunt weapon
    'if they are and the skill check is passed, apply the appropriate bonuses.
    If CurrentPlayer.Wielded.ItemType = 3 And RandomNumber() <= CurrentPlayer.Bluntweaponfighting Then
        If CurrentPlayer.Bluntweaponfighting < 21 Then
            CurrentPlayer.Hitbonus = CurrentPlayer.Hitbonus + 1
            CurrentPlayer.Damagebonus = CurrentPlayer.Damagebonus + 1
        ElseIf CurrentPlayer.Bluntweaponfighting > 20 And CurrentPlayer.Bluntweaponfighting < 41 Then
        ElseIf CurrentPlayer.Bluntweaponfighting > 40 And CurrentPlayer.Bluntweaponfighting < 61 Then
        ElseIf CurrentPlayer.Bluntweaponfighting > 60 And CurrentPlayer.Bluntweaponfighting < 81 Then
            CurrentPlayer.Damagebonus = CurrentPlayer.Damagebonus + 4
        ElseIf CurrentPlayer.Bluntweaponfighting > 80 Then
        End If
    Else
        'otherwise, try to improve blunt weapons fighting.
    End If
End Sub

Private Sub Brawling()
' check for a proficiency
      If RandomNumber() <= CurrentPlayer.Brawling Then
        If CurrentPlayer.Brawling <= 50 Then
          Evade = Evade + 1
          CurrentPlayer.Hitbonus = CurrentPlayer.Hitbonus + 1
        Else
          Evade = Evade + 2
        End If
      Else
      End If
End Sub

Private Sub CheckMonFlee()
' The big function for the monster, this determines the monster's actions for the 'round of combat.
      If Monsterturn = 0 Then ' if this is the first turn, then,
        If CurrentMonster.HasSpecial = True Then ' does the monster have a special
          Call MonsterSpecial ' if so, call monster special
        Else ' if not,
          MonsterAttack ' call the monster's attack function
        End If
      Else ' if this is any OTHER round ,
        If ((CurrentMonster.Current_HitPoints / CurrentMonster.Total_HitPoints) * 100) <= CurrentMonster.FleePercent Then
          ' if the hitpoints have dropped below the flee level for that monster, call
          Call MonFlee
        ElseIf CurrentMonster.HasSpecial = True Then ' otherwise if he has a special,
          Call MonsterSpecial
        Else
          MonsterAttack ' if all else fails, attack
        End If
      End If
End Sub

Private Sub CheckFatigue()
' checks level of attack and if the the player has the fatigue left for that
' if not, recalls player option
      If AttackLevel = 1 And CurrentPlayer.Fatigue < 5 Then
        AttackMessages.Caption = " You are too tired to do that. "
        PlayerOption
      ElseIf AttackLevel = 2 And CurrentPlayer.Fatigue < 10 Then
        AttackMessages.Caption = " You are too tired to do that. "
        PlayerOption
      ElseIf AttackLevel = 3 And CurrentPlayer.Fatigue < 15 Then
        AttackMessages.Caption = " You are too tired to do that. "
        PlayerOption
      Else
        Attack
      End If
End Sub

Private Sub CheckTurn()
' if the player is poisoned, then this executes to see
' if the player takes damage from the poison.
      Dim tempor As Integer
      If CurrentPlayer.Poisoned = True And Playerturn > 0 And Playerturn Mod 2 = 0 Then