Subhunt: A Submarine Action Game

Ian Parberry[1]
Department of Computer Sciences
University of North Texas

William R. Pensyl[2]
School of Visual Arts
University of North Texas

October 1997

Abstract

Subhunt is a 2.5D, sprite based, real-time, first-person, single-player submarine action game produced by faculty and students at the University of North Texas. Players find themselves piloting a submarine with the task of locating and destroying enemy shipping while managing critical resources and trying not to get killed. Visual and audio cues create the illusion of action that allows the player to become completely immersed in the *Subhunt* virtual world.

1 Introduction

Subhunt is a 2.5D, sprite based, real-time, first person, single-player submarine action game produced by a multidisciplinary team of faculty and students at the University of North Texas. Players find themselves piloting a submarine with the task of locating and destroying enemy shipping while managing the resources of air, torpedoes, battery charge, and diesel fuel. Visual and audio cues create the illusion of action that allows the player to become completely immersed in the *Subhunt* virtual world, which can be seen in the sample screenshots shown in Appendix A.

The main body of this manuscript is divided into six sections, each of which is divided into subsections. Section 2 explains some basic production issues. Section 3 describes the sound, music, and artwork. Section 4 covers some of the more important code design issues. Section 5 is devoted to a discussion of the AI used by the enemy ships. Section 6 describes some of the graphics tricks used to increase speed. Section 7 describes the properties of the virtual world in which the game is played. We conclude with pointers to more information on *Subhunt*, and two Appendices. Appendix A contains some screen shots, and Appendix B contains diagrams of the control panel and lists of keyboard commands.

2 Production

Subhunt was produced by faculty and students in the Department of Computer Sciences, the School of Visual Arts, and the College of Music at the University of North Texas. This section discusses some basic production issues. Section 2.1 discusses our motivation in creating a computer game. Section 2.2 describes our target architecture, and the reasons for choosing it. Section 2.3 describes our target audience. Section 2.4 outlines our marketing strategy. Finally, Section 2.5 details student participation in the project.

2.1 Motivation

What business does a university have in producing a commercial computer game? Our primary goal is to establish UNT as a place where game programmers and artists can learn and advance the tools of their trade. The production of *Subhunt* is a strategy that contributes to this goal in the following ways:

- It provided the authors of this paper with hands-on experience in the production of a computer game, from initial design through to shipping the final product.
- It provided the students engaged in the project with hands-on experience that will help them to find jobs in the computer game industry.

[1] Author's address: Dept. of Computer Sciences, University of North Texas, P.O. Box 311366, Denton, TX, 76203-1366. Email: ian@cs.unt.edu. URL: http://hercule.csci.unt.edu/ian.
[2] Author's address: School of Visual Arts, University of North Texas, P. O. Box 305100, Denton, TX, 76203. Email: pensyl@art.unt.edu.

- It is an ongoing experiment in an unconventional funding source. Government funding for basic and applied research is on the decline, and industry sources are less willing to contribute money without concrete results. While it is unlikely that *Subhunt* will bring in significant funds, it will demonstrate our ability to see a project through to completion on a very limited budget[3].

2.2 Target Architecture

We chose for our target architecture the most popular platform at the start of the project in 1995: the Intel processor under DOS (or in a DOS box under Windows 95). The Soundblaster sound card from Creative Laboratories is supported. Game input can come from several sources:

1 From the keyboard. The default keyboard assignment (see Table 3 and Table 4 in Appendix B) can be changed by the user and saved for future use.
2 From the joystick. The joystick controls the rudder and ailerons for horizontal and vertical movement respectively. Up to four buttons are supported.
3 From the mouse, which can be used in one of four modes:
3.1 Finger mode, in which the mouse is used to press buttons on the control panel.
3.2 Controlling the rudder (left to right motion) and throttle (up and down motion).
3.3 Controlling the rudder (left to right motion) and ailerons (up and down motion).
3.4 Controlling the rudder only (left and right motion).

Custom device drivers were written for each device.

The code consists of 28000 lines of C++ and 2000 lines of assembly code. The executable was created with the Watcom C/C++ compiler, Version 10.6, and uses the DOS4GW 32-bit DOS extender. The graphics are VGA, Mode 13, 320x200 pixels with 256 colors. All of the code was custom built, using no libraries or other commercial code.

2.3 Target Audience and Game Play

Subhunt targets the easiest audience in the game industry: teenage males. This audience is attracted to fast-paced real-time action games. The aim of *Subhunt* is to sink enemy vessels without incurring too much damage to your submarine, while simultaneously managing the resources of air, battery charge, fuel, and torpedoes. The reward for sinking ships includes:

- The immediate feedback of seeing and hearing them explode and sink.
- The accumulation of powerups from the wreckage. These objects recharge the player's resources, enabling longer game play. (Powerups are described more fully in Section 7.7.)
- The reduction of future damage to the player's submarine, also enabling longer game play.
- The awarding of medals and progress to the next level (with new challenges) when all enemies have been sunk.

2.4 Marketing Strategy

The distribution rights to *Subhunt* were released under contract to Spectrum Pacific Publishing, a shareware publishing company. A fully functional public-domain version of the game can be downloaded from one of several sites on the World-Wide Web (see Section 8). This shareware version of the game consists of one mission set containing 3 missions. This allows several hours of game play, during which the player will (it is hoped) become immersed in the game. To receive the full game, which has three additional mission sets with a total of 12 missions, the player must order the commercial version on CD-ROM from the shareware publisher.

2.5 Student Participation

A total of 18 students participated in the creation of *Subhunt* on a voluntary basis. Students wrote 4000 of the 30000 lines of code, composed the 15 pieces of music, and drew the more than 1000 files of artwork in the game.

3 Multimedia

Subhunt is a multimedia game, meaning that graphics, sound effects, and music contribute to player's willing suspension of disbelief. This section describes the multimedia elements of *Subhunt*. Section 3.1 gives details of sound card support. Section 3.2 describes the sampled sound effects. Section 3.3 describes the MIDI music. Finally, Section 3.4 discusses the artwork.

3.1 Sound Card Support

Subhunt supports the Soundblaster 16 (FM synthesis) and Waveblaster (MIDI wave table) sound cards from Creative Laboratories, which are among the most popular sound cards for PC gamers. The player selects the sound card driver to be used in *Subhunt* by first running a custom sound setup program called

[3] The total amount of cash expended, exclusive of faculty salaries, was less than $7000.

`sndsetup.exe.` This program presents the user with a menu from which the sound card type is selected. It then plays sound and music while allowing the user to set initial values for the (software controlled) master volume, sound effects volume, and music volume.

Since some players will not necessarily have an appropriate sound card, audio feedback is supplemented with a text analog in the form of a heads-up display (or HUD) that overlays the periscope window. Text scrolls through this window in real time as the game progresses. For example, when the submarine is hit by a shell the player sees a large explosion in the periscope window, hears an explosion (if a sound card is present), and sees the text "Shell hit" scroll by on the HUD.

3.2 Sound Effects

The sound driver was written by Steve Wilson, a senior undergraduate student in the Computer Science program. Sounds are preloaded into memory from a packed VOC format file. During game play, individual sounds are played asynchronously using DMA. Up to four sounds can be mixed and played together dynamically. There are 60 sound effects sampled at 12 KHz in mono, using 1MB of RAM. Sound effects are a combination of sounds from royalty-free CD-ROMs, sounds sampled from everyday life with a microphone, and faked sounds that have been modified using standard public domain audio editing software.

3.3 Music

The music for *Subhunt,* composed by Steve Wilson and Jeremiah Isaacs, consists of 15 original compositions[4], most of which are 3-5 minutes in duration. The music sets the mood for the game, with 3 compositions on a central theme for each of the registered mission sets. In contrast to the active music during gameplay, a mellow hornpipe is used while the menu system is being displayed.

The music is in MIDI type 0 format. MIDI format was chosen for compactness, because it consists essentially of a sequence of key-up and key-down commands. The music driver, which was written by Steve Wilson, supports both FM synthesis and MIDI wavetable using the MPU 401 chipset.

[4] One for each of the 12 missions, one for the menu system, one for use in `sndsetup.exe`, and a logo chord.

3.4 Artwork

The art for *Subhunt* was primarily created by students enrolled in "Art and Design of the Computer Game" taught by the second author during Spring semester 1997 (see Section 9 for a full list of names). This is a new course offering in the School of Visual Arts taught in conjunction with "Game Design and Programming" in the Department of Computer Sciences. All artwork was designed following Design Process, allowing the student artists to understand the complete process of preparing art for the computer game. The students conceptualized the artwork by interviewing the producer of the game and developing initial artwork for presentation. Following approval of these sketches by the producer finished drawings were produced on paper prior to final creation on the computer.

To create the final artwork off-the-shelf art tools were used, primarily Kinetix *3DStudio Max* for 3D modeling and rendering. Additional modeling was completed in Caligairi *TrueSpace*. Adobe *Illustrator* and *Photoshop* were used extensively for design and for image processing prior to final integration into the code.

Each ship and active playing asset was created using a texture mapped polygon mesh model, which was then lighted and rotated through 360 degrees. All of the models were rendered at 10-degree increments, and these images were used as ship sprites (for example, Figure 1 shows 8 of the 36 images of the patrol boat). The camera was placed at the water level in order to simulate the perspective of the periscope, ensuring that the bottoms of the ships are flat.

Figure 1 *Some of the patrol boat artwork.*

Following rendering each image was processed individually in a procedure developed by second author and the students to create nonanti-aliased edges and indexed into the 256-color game palette. Since *Photoshop* version 3.01 did not have batch processing, each image was processed using a sequence of function key commands set up in the

program. This procedure allowed the students to automate the process to some degree. The finished images were transferred to the programmers from the School of Visual Art's file server.

4 Code Design

This section describes some of the major design decisions that were made during the creation of the code for *Subhunt*. Section 4.1 introduces the main frame loop, consisting of frame composition and frame blitting, which are discussed in Sections 4.2 and 4.3, respectively. Section 4.4 discusses the use of timers, and Section 4.5 discusses the 2D versus 3D parts of the game. Code and data organization are outlined in Sections 4.6 and 4.7, respectively. Section 4.8 sketches some security issues with respect to piracy, and finally, Section 4.9 discusses designing with foreign language translation in mind.

4.1 The Main Frame Loop

The main loop for *Subhunt* consists of two tasks. Firstly, compose a frame of graphics in an array in memory called the *offscreen buffer*. Secondly, blit (block load) the offscreen buffer quickly to video memory.

The offscreen buffer resides in general-purpose memory for following reason. The sprite engine uses the so-called *painter's algorithm*; first draw the background, then draw the objects on top of it, from furthest to nearest. Overwriting pixels in this manner is cheaper than computing which objects occult others in the view screen. Thus, each pixel may be overwritten several times, which is faster in general-purpose memory than video memory[5].

The most important facts to remember about the main frame loop are:

- The frame rate will vary from one computer to another depending on processor speed and graphics capability.
- The frame rate will vary within a game depending on the amount of action.

It is imperative that the graphics and game action remain smooth despite these challenges. The frame rate should:

- Average at least 24 fps (frames per second).
- Be at least 12 fps in the worst case.

[5] Video memory hardware is often intrinsically faster than general-purpose memory, but most of the access cycles are allotted to the *video serializer* (the piece of hardware that draws from the video memory to the screen), rather than to the CPU.

4.2 Frame Composition

The steps in composing a frame of *Subhunt* are as follows:

- Process the player input from the device drivers.
- Update the objects (includes physical motion and state change).
- Draw the background, a texture-mapped sea and sky (see Section 6.1).
- Draw the foam (see Section 6.3).
- Sort the visible objects (including ships, static objects, and land sprites) on distance from the sub using Quicksort (see Hoare [3]).
- Draw sprites (see Section 6.2) in decreasing order of distance.

4.3 Blitters

To speed up the frame loop, different areas of the screen are treated in different ways according to the level of activity that they experience:

1. Active areas of the screen that usually change in every frame are blitted to the video memory at the end of each iteration of the frame loop.
2. Areas of video memory that do not change (such as the background of the instrument panel) are loaded once to the video memory and left there.
3. Areas of video memory that change slowly (such as flashing lights) are written directly to video memory when the need arises.

We arranged for the artist to deliver the viewport artwork with active areas painted in a reserved palette position (specifically, palette position 0) as in Figure 2. (Compare this with the screenshot in Figure 5 in Appendix A). Note that some of the active areas are irregularly shaped.

Figure 2 The control panel artwork. The four active areas to be blitted are depicted in black.

Since *Subhunt* has three main viewports (see Appendix A), the coding of fast blitters is a tedious, time-consuming, and error-prone task that cries out to

be automated. We constructed a tool called Qblit (for Quick blitter) that inputs a pcx file *f* and outputs a piece of assembly code that loads from the offscreen buffer to video memory (in the fastest manner possible) those pixels drawn in palette position 0 in *f*. Qblit was then run on the three viewports to create custom assembly code blitters that were then compiled into the project (see Section 4.6).

4.4 Timer Based Code

In order to ensure that the game runs at the same speed on a variety of processors (which at the time of shipping ranged from a 66MHz 80486 to a 233MHz 80586), the code is heavily timer-based. A fast millisecond game timer replaced the default clock interrupt service routine. All actions in the game depend upon that timer. For example:

- Ships moves a distance that depends on the ship's speed and time since last move.
- Each ship has a vector and a desired vector, the latter being set periodically by the AI (see Section 5). If the desired vector is different from the current vector, then the former is changed by an amount dependent on the turn distance and time since last turn.

4.5 2D Versus 3D

Subhunt is technically what is known in the computer game industry as a 2.5D game. While the player has what is apparently a 3D viewpoint, the action actually takes place on what is essentially a 2D board that is "popped up" into the third dimension using sprites, which are the computer graphic equivalent of cardboard cutouts.

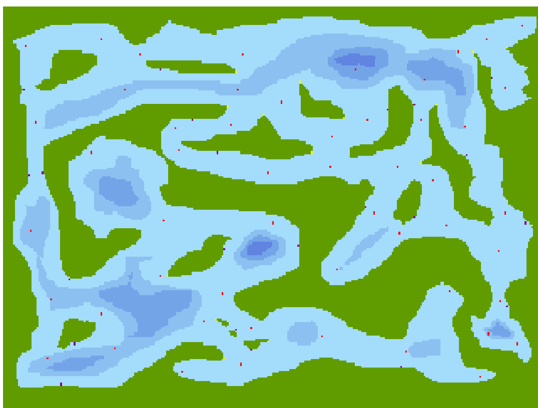

Figure 3 *A long range scan map.*

Subhunt is heavily based on a 2D map that is input as a pcx file (for example, see Figure 3), each pixel of which represents a rectangular *tile* of sea or land. The palette position of each pixel determines what is located in the corresponding tile. The map has many purposes:

- User display: The map is displayed on the sub control panel (if the sub is at periscope depth) as a long range scan map, with ship and sub positions overlayed[6].
- Depth calculation: The pixel value for each tile indicates the depth of water.
- Land detection: The ships and sub project a line forward from their current position on the map to detect imminent land collision.
- Collision detection: Collision detection is optimized by only having objects compute relative distances when they occupy the same tile (see Section 7.5).
- Coastline skeletonization: The coastline is reduced to a line, which is displayed in the active sonar (see Section 6.4) and is used to display the coastline in the periscope window as a series of sprites, one per pixel.
- Static objects: The location of immobile objects (see Section 7.7) is encoded as colored pixels.
- Motion planning: The map also contains the locations of vertices in the motion planning graph (see Section 5.3).

4.6 Code Organization

The code for *Subhunt* is divided into 56 header files, 47 C++ files, and 8 assembly code files. The assembly code files are as follows:

1. DMA_CODE.ASM: code for DMA
2. MBLIT.ASM: blitter for camera view
3. PKEYS.ASM: keyboard driver
4. SBFMASM.ASM: music driver
5. SB_DSP.ASM: sound driver
6. SCREEN.ASM: video driver
7. VBLIT.ASM: blitter for periscope view
8. ZBLIT.ASM: blitter for control panel

The C++ code files are as follows:

1. ASMSOUND.CPP: sound driver
2. BOARD.CPP: long range scan map
3. BRESLINE.CPP: Bresenham's algorithm
4. BUTTONS.CPP: pushbutton animation
5. COMPASS.CPP: compass
6. CRSHRS.CPP: compiled sprite for crosshairs
7. DAMAGE.CPP: damage control
8. DEMO.CPP: demo mode
9. FOAM.CPP: foam animation manager

---

[6] However, to enhance gameplay, some enemy vessels do not appear on the long range scan, and false land information may be included in certain missions.

10. GEOMETRY.CPP: geometry
11. GODMODE.CPP: God mode manager
12. GRAPH.CPP: motion planning
13. HEADSUP.CPP: heads-up text manager
14. HELP.CPP: help manager
15. HUNT.CPP: `main()`
16. INDEX.CPP: file manager
17. INPUT.CPP: input manager
18. JOY.CPP: joystick driver
19. KBD.CPP: keyboard settings manager
20. LEVELS.CPP: level description manager
21. LOADBAR.CPP: load bar animation
22. MEDALS.CPP: medal manager
23. MENU.CPP: menu manager
24. MIDIMOD.CPP: music driver
25. MOUSE.CPP: mouse driver
26. OXYGEN.CPP: control panel dial manager
27. PALFX.CPP: palette effects
28. PCX.CPP: pcx file input
29. PCXSNAP.CPP: screen snapshot saver
30. PREDICT.CPP: sub prediction manager
31. RANDOM.CPP: pseudorandom number generator
32. SBFM.CPP: music (FM synthesis)
33. SBFM_ISR.CPP: music (FM synthesis)
34. SBSPRTS.CPP: general sprite manager
35. SCNRIO.CPP: scenario manager
36. SETTINGS.CPP: custom game settings manager
37. SHIPS.CPP: ships and fleet control
38. SAVER.CPP: save game manager
39. SONAR.CPP: sonar
40. SOUND.CPP: sound settings manager
41. SSMGR.CPP: ship sprite manager
42. SPRITE.CPP: sprites
43. STRTABLE.CPP: string table manager
44. SVGA.CPP: SVGA for intro screen
45. TORPS.CPP: torpedo manager
46. TEXT.CPP: text manager
47. TYPEWRTR.CPP: typewriter text animation

4.7 Data Organization

Subhunt uses the following files. Note that packed files have been used for the sounds, music, art, and text. Index information is included in separate index files for security reasons (see Section 4.8).

- DEFAULT.REG: sound register data
- DEMO.DAT: demo keystroke file
- DOS4GW.EXE: 32-bit DOS extender
- IMAGES.PPX: packed art file
- KBDMAP.DAT: player keyboard map
- LVLINDX.DAT: text index information
- MIDI.IBK: MIDI instrument bank file
- PMDINDX.DAT: music index information
- PPXINDX.DAT: image index information
- SETTINGS.DAT: player preferred settings
- SNDSETUP.EXE: sound setup executable
- SUBHUNT.EXE: *Subhunt* executable
- SUBHUNT.PFL: packed text file
- SUBHUNT.PMD: packed music
- SUBSOUND.VPF: packed sound effects
- *.SVG: saved games

4.8 Security Issues

The threat of piracy is an important issue that affects code design from the earliest stages. While it is true that everything a game designer can do a pirate can undo, the role of security code is to prevent casual hacking and to make the pirate's job as painful as possible. Security assumptions must include:

- Pirates will use state-of-the art symbolic disassemblers and debuggers to examine the code structure and data files.
- Pirates will view cracking a game as a challenge, and hence will devote large amounts of time to the task without necessarily having any regard for potential profit.
- The game will eventually be cracked.

Some of the security measures undertaken in *Subhunt* include[7]:

- Input Files: The music file, the art file, and the text data file are all encrypted. Some files use a slightly different encryption method. The method chosen makes some data appear only marginally distinguishable from random strings. Index files may be encrypted using a different password from the corresponding packed data file.
- Saved Game: The saved game files are also encrypted, and contain information that prevents the player from receiving medals by saving a game that is close to completion, then replaying the saved game repeatedly or sharing it with a friend.
- Medal Files: The medal files are encrypted and use multiple orthogonal encrypted checksums to prevent alteration by players.
- God Mode: The game supports God mode, in which players are invulnerable and use no resources. God mode is a favorite target of hackers and pirates. There are two God mode passwords. The first, which will be made public, does not allow the player to win medals. The second, which will not be released, allows the player to win medals. To get into God mode, the player hits the F12 key, then types in a password. What happens next is unorthodox. The player is

[7] For obvious reasons, not all of the security measures will be described.

momentarily put into God mode no matter what password is entered. A number of asynchronous password verification agents scattered throughout the executable code become active at various randomly chosen intervals ranging from several milliseconds to several seconds after the password is entered. Each has the responsibility of checking a single byte of the password, and will cancel God mode if an error is detected. The lack of proximity of these agents to the password acquisition code in the executable file and in time is intended to make the task of locating and disabling them tedious.

4.9 Foreign Language Translation

Subhunt may be played in English or German, the selection being made when `sndsetup.exe` is run. The task of incorporating foreign language translations is made easier if it is incorporated into the design process at an early stage. For example:

- Avoid using text in the artwork. Use icons instead wherever possible.
- Load all of your text from external text files.
- Do not use quoted text in your code. Instead, load all strings from a text file into a string table.
- Include accents in your custom fonts.

5 Artificial Intelligence

It is easy to make AI that is unbeatable, since the AI has access to complete information about the player's sub. The AI can be in the right place at the right time, and every missile, shell, and depth charge can be a hit. The problem is to devise imperfect AI that is somewhat unpredictable, believable, challenging, and yet defeatable. The ship AI has three components:

1. A rule based system (Section 5.1)
2. Agents (Section 5.2)
3. Motion planning (Section 5.3).

5.1 State and Rules

The actions taken by the ships are governed by rules that are dependent on:

- Sub visibility
- Distance from sub
- Mood
- State
- Armament

The moods are: scared, concerned, balanced, feisty, and aggressive. The current mood depends on the amount of damage taken.

The ship states are:

- Wandering (heading in a random direction with long period)
- Avoiding (heading away from sub's last known position)
- Attacking (attacking the sub)
- Exploding (stopped, with animated explosion)
- Sinking (stopped, disappearing under water)
- Dead (nonexistent, used for garbage collection)
- Waiting (stopped for a specific period)
- Touring (under control of motion planning)
- Aground (too close to land, shortly to begin sinking)
- Panicking (unpredictable behavior)
- Patrolling (moving back and forth between two points using motion planning)
- Dithering (heading in a random direction with very short period)
- Rotating (turning in place)
- Harassing (actively seeking sub)
- Parked (stopped in place)
- Circling (travelling in a circle)
- Ramming (heading for sub if visible).

The attacking state has four phases when the sub is at periscope depth:

- Charge (head directly for sub, firing torpedoes and forward deck guns if available)
- Broadside (circle sub, firing all deck guns if available)
- Retreat (run away from the sub, firing rear deck guns and laying mines if available)
- Stand down (go back to default state, which is usually either Touring or Patrolling).

The ship changes attack phase dependent on mood, distance from sub, and a random factor.

5.2 Agents

Each ship AI has three competing and cooperating agents (after Minsky [4]), which are implemented as follows. The *default agent* does nothing. The other agents spend some of the time asleep, awakening after a specific time period to sample the current conditions and take action based on that sample.

The *land avoidance agent* is in charge of making sure that the ships do not run into the land. It is a high priority agent that is frequently active. The action taken is dependent on the current ship state and a random factor. Possible land avoidance actions include breaking to the right, breaking to the left, and reversing course. Nonetheless, it is quite feasible to drive a ship onto land, as we will see in the next paragraph.

The *sub detection agent* is responsible for detecting the player's submarine and taking appropriate action. The action taken depends on the current ship state and a random factor. It becomes active every 2-4 seconds depending on the ship's mood. Its decisions will override those of the land avoidance agent, and hence may accidentally lead to the ship being driven aground. The sleep time means that ships will only take action 2-4 seconds after the sub becomes visible, and that information about the sub's location will be imperfect. Sub visibility is dependent on the following factors:

- Distance to ship
- Type of engine in use (diesel or electric)
- Intervening land, if any
- Current speed
- Time since a torpedo was last fired
- Depth
- Whether active sonar is in use
- A random factor

5.3 Motion Planning

The maps in *Subhunt* can contain more land than is often seen in submarine games (see, for example, Figure 3). The problem with having too much land is that it constrains motion. As described in the previous section, the AI takes care of land avoidance, but simple avoidance of land is not sufficient for realistic game play since it typically results in ships acting like rubber balls that ricochet from landfall to landfall. Instead, ships should appear to implement sensible plans for reaching destinations.

Up to 64 tiles in each map are labeled as special locations called *vertices*. Each vertex represents a valid destination for ships in the Touring state. When a ship enters the Touring state, it chooses a random vertex *v* as destination and travels a shortest path from its current location through intermediate vertices to *v*. It then waits for a random amount of time in the Waiting state before picking a new random destination. A ship that is in the Harassing state will head for the vertex that is nearest the sub. On arrival, if the sub is visible from that vertex, the ship will go into the Attack state. Otherwise (the sub has moved on while the ship was en route), the ship heads for a random vertex, and from there resumes the chase. Ships in the Patrolling state will use motion planning to move back and forth between two fixed vertices.

The shortest path between vertices is computed using Floyd's algorithm (see Floyd [2], or for more contemporary coverage, see any standard algorithms text such as Cormen, Leiserson, and Rivest [1]). Floyd's algorithm uses a precomputed table that allows the shortest path between any pair of vertices to be computed quickly. The table is constructed from the vertex information in the map in less than a second on entry to each level. The initial graph for Floyd's algorithm consists of the vertices indicated in the map, joined by edges between those pairs of vertices for which a line can be drawn from one to the other without hitting land. The amount of storage required for the table is two 64 by 64 arrays of 8-byte integers, for a total of 64K bytes.

An interesting and desirable effect of this approach is that ships tend to move in reasonably wide sea-lanes, as does shipping in everyday life.

6 Graphics

Subhunt's graphics provide the majority of the feedback to the player. We will discuss briefly four of the techniques used. Section 6.1 describes the quick-and-dirty texture mapping algorithm used for the sea and sky. Section 6.2 discusses some of aspects of sprite animation. Section 6.3 discusses the animation of sea foam. Section 6.4 describes the animation technique used for the sonar screen.

6.1 Texture Mapping

Some elementary texture mapping techniques are used to draw the sea and the sky to the periscope window. The horizon is kept horizontal to simplify matters. Rather than perform an accurate texture map to the tiles of the sea and sky, we use a sloppy algorithm that is fast, yet provides enough visual cues to the player.

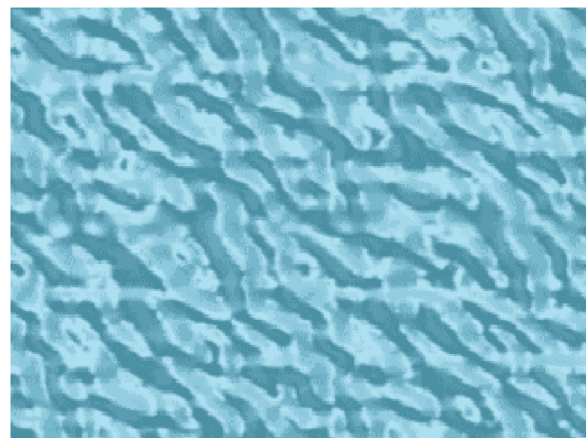

Figure 4 *The ocean texture.*

Consider the example of texture-mapping the ocean (the sky is similar). The process starts with a toroidal

ocean tile, that is, an ocean texture drawn from above that wraps from top to bottom and left to right (see Figure 4). The texture mapping algorithm maintains the x and y coordinates of a source point p within this tile. A rectangle is drawn from the tile using p as the top left corner (wrapping around at the right and/or bottom as necessary) to the offscreen buffer in a position that covers the periscope window in the current view (control panel, fullscreen periscope, or missile camera). The drawing is done in a manner that allows for motion and perspective correction, as follows.

Suppose the sub is moving forwards (the other case is left as an exercise for the reader). Viewpoint motion is handled by having the source point p move with the sub. More importantly, the direction in which p moves depends on which direction the periscope is facing. For example, if the periscope is facing forwards, then p moves up. If the periscope is facing to the left, then p moves to the right. In general, if the periscope is at angle ϑ from straight ahead, then p moves at angle of $-\vartheta$ from up in the tile.

Perspective correction is faked by dividing the periscope window into horizontal bands of equal height, and copying every second row of the ocean texture into the lowest band, every third row into the next band, every fourth row into the next band, and so on. Since there is no horizontal perspective correction, each band can be drawn very quickly using block moves. No attempt is made to correct the fact that a *different* set of alternating rows is used as p moves vertically in the tile, resulting in a shimmering effect that is suggestive of ocean waves.

6.2 Sprite Animation

A *sprite* is game industry terminology for a small graphic image to be loaded to video memory in dirty rectangle animation. There are two basic kinds of sprites: computed sprites and compiled sprites. *Computed sprites* are loaded from art files, and include transparent pixels indicated using a reserved palette position (we used palette position 0). They are drawn by a single general-purpose piece of code that successively examines each pixel in the image, drawing only the nontransparent ones.

Almost all of the sprites in *Subhunt* are computer sprites. For example (as mentioned in Section 3.4), each ship sprite is stored at a fixed size, viewed from one of 36 different angles. The correct angle is selected at runtime, and the appropriate sprite image is drawn to the offscreen buffer, performing clipping and scaling on-the-fly.

Sprite scaling is an interesting task that can unintentionally be made harder than necessary. In *Subhunt*, sprites can be drawn larger or smaller than actual size. For higher image quality, ship sprites are 200-300 pixels wide, and are therefore scaled down more often than up. Downscaling can be achieved by skipping certain rows and columns of pixels, for example, half size can be achieved by drawing every second row and column. Suppose we call this a *skip distance* of 2. Drawing at 2/3 scale requires alternating skip distances of 1,2,1,2,1,2,… etc. Other scaling factors may require increasingly complex skip distances, for example, 3/11 scale requires skip distances of 3,4,4,3,4,4,… etc.

Some game programming texts recommend that a table of skip distances be maintained for a range of useful scale factors. A better approach is to use a floating point skip distance of $1/s$ when the scale factor is s. The current position within the sprite image is kept as a floating point value and type-cast to an integer before retrieving the pixel. This approach can be further accelerated by using fixed point values with a 16-bit integer part and a 16-bit fractional part instead of floating point numbers.

The ability to scale sprites to an arbitrary fraction can be a two-edged sword. Changing the scale factor slightly may change the skip distance but not the scaled sprite's width or height, causing the pixels of the scaled image to shift slightly in a disturbing roiling fashion known as *sprite creep*. This effect can be avoided by recording for each sprite the last used scale factor. If the new scale factor does not change the sprite's height or width, then the old scale factor is used instead.

In contrast, a *compiled sprite* is drawn with a custom piece of code that draws only the nontransparent pixels without even examining the others. Compiled sprites should be used when:

- The sprite has lots of transparent pixels.
- Speed is an issue (for example, when the sprite is very large).
- The sprite will not be clipped, scaled, rotated, or otherwise processed in any way.

This was the case for the crosshairs in the fullscreen periscope view (see Figure 6 in Appendix A). A new tool similar in nature to Qblit (see Section 4.3) was constructed. This program inputs a sprite image and outputs a C++ function that draws the image directly to the offscreen buffer. This tool was used to create a compiled sprite for the crosshairs, which was then compiled into the project (see Section 4.6).

6.3 Sea Foam

The objects in a computer game should not only move through the landscape, they should also act on the landscape. In *Subhunt*, moving objects leave a foam trail in the water. Foam trails appear:

- In a line behind torpedoes (which are otherwise invisible).
- In a swath behind ships and the player's sub. In the former case the foam trail provides valuable visual feedback about the direction in which the ship is moving. In the latter case, it provides extra visual feedback that the periscope is pointing backwards.
- In a large expanding ring for the wavefront of an EMF burst (see Section 7.6).
- In a small expanding ring around splashes, and as the final part of the animation of a sinking ship.

Foam animation is achieved using white pixels to represent foam flecks. A list of flecks is maintained, each with an expiration time. Foam trails are created by having flecks created near the ship's center, each with a slight random variation in its expiration time in order to make the end of the trail appear ragged. A foam ring with center c and diameter d is created by placing a number of flecks at distance d from c in random directions. The value of d is started at zero and increased slowly over the lifetime of the ring. A slight variation in fleck expiration time completes the effect of an expanding ring of foam.

6.4 Sonar

There are two kinds of sonar in *Subhunt*. Firstly, there is passive sonar, which merely listens, and shows only ships. Secondly there is active sonar, which sends out pings and listens to the return echoes, showing ships, objects, and the shoreline. Since real sonar screens require substantial training to interpret, *Subhunt*'s sonar display actually looks like a radar display with a sweep arm and simulated plasma screen style fade-out.

The sonar animation uses 16 reserved palette positions containing various shades of yellow-green. Blips are initially drawn in a bright yellow. After a small fixed time interval, the palette position of each blip is darkened by one shade until it matches the dark green of the background. The sweep arm is initially drawn in an intermediate yellow, and is faded out along with the blips to leave a long sweep trail. It is drawn as a one-pixel wide Bresenham line from the center of the sonar screen to a point near the outside. We were initially faced with the problem that an arbitrary Bresenham line will not necessarily cover all pixels when swept in a circle, leaving embarrassing gaps in the plasma screen. This problem was easily rectified by choosing the radius of the sweep arm to be a few pixels greater than the width of the sonar window (which is clipped by the blitter).

7 The Virtual World

This section will describe briefly the virtual world that *Subhunt* provides the player. Section 7.1 describes the level design process, which delivers world data to the program. Section 7.2 describes the scenarios for the four mission sets. Section 7.3 describes the ships that are encountered. Section 7.4 describes the deck guns and the method of simulating shells. Section 7.5 describes the handling of torpedoes and missiles. Section 7.6 briefly describes other weapons, including depth charges and EMF bursts. Section 7.7 describes the static objects encountered in the game. Section 7.8 describes the possible outcomes of playing *Subhunt*: winning or losing.

7.1 Level Design

```
map="map0.pcx";

submarine{
  location=(10,5);
  vector=southeast;
  fuel=50;
  battery=50;
  air=50;
}

powerup{
  class=mystery;
  location=(240,113);
  vector=0;
}

ship{
  class=speedboat;
  name="scuzzbucket";
  location=(297,113);
  vector=north;
  maxspeed=3;
  state=touring;
}

minestring{
  location=(280,90) to (310,90);
}
```

Table 1 *Sample declarative code for level design.*

Level design is the task of designing the worlds to be encountered in the various levels of the game. Game companies will typically invest a person-year or more of effort in creating a *level design editor,* which is used for a period of a few months by specialized level designers and then discarded[8]. Instead, *Subhunt* inputs a pcx map containing information about static objects (see Section 4.5), and inputs information about mobile objects from a text file using a custom declarative language with a C-like syntax (for example, see Table 1).

7.2 Mission Scenarios

Subhunt has four mission sets, each consisting of three missions with a common scenario.

- Covert Operation: A war scenario in which the player is to destroy all shipping.
- Nemesis: A terrorist scenario in which the player is to destroy terrorist craft that are harassing tourist cruise liners.
- Eco Offense: An eco-friendly scenario in which the player is to sink longline trawlers and whalers.
- Leviathan: A mystery scenario in which the player finds him or herself facing various sea monsters.

Covert operation is shipped with the shareware version of *Subhunt*. The remaining mission sets are shipped only with the registered version.

7.3 Ships

Subhunt has a total of eighteen different vessels.

- The speedboat, patrol boat, destroyer, and battleship appear in Covert Operation.
- The jetski, cruise liner, airplane, and oil derrick appear in Nemesis.
- The Trawler, whaler, dolphin, and whale appear in Eco Offense.
- The barge, oil tanker, crab, moray, octopus, and waste dump appear in Leviathan.

Each of the ships differ in various properties, some of which are shown in Table 2. Armor comes in various strengths which determine how many torpedo hits are necessary to sink the ship. Armament includes deck guns, torpedos, depth charges, EMF burst, and other projectile weapons.

Not all ships will appear on the long range scan. Of those that do not, some are static and some are mobile. Those that are static must be discovered by exploration. Those that are mobile can be tracked by noticing their tendency to harass civilian vessels (which fortunately *do* appear on the long range scan).

The initial position of enemy ships is part of the level design, however, in some missions at some difficulty levels enemy vessels may created dynamically. Learning how to circumvent this is part of the challenge facing the player.

Vessel	Guns	Torps	Dpth Chrg	EMF Burst	Other	Enemy	Visible	Armor
Speedboat						•	•	
Patrol boat	•	•				•	•	L
Destroyer	•	•	•			•	•	H
Battleship	•					•	•	VH
Jetski		•				•		
Cruise liner							•	
Airplane		•	•			•	•	
Oil Derrick	•					•	•	VH
Trawler				•		•	•	
Whaler				•		•	•	
Dolphin								
Whale								
Barge							•	
Oil tanker							•	
Crab	•		•			•		H
Moray					•	•		
Octopus					•	•		L
Waste dump				•		•		M

Table 2 *Ship types in Subhunt. The dots in columns 2-6 indicate that the vessel has, respectively, guns, torpedoes, depth charges, EMF burst, and other weapons. A circle in column 7 indicates that the vessel is an enemy ship that must be sunk. A circle in column 8 indicates that the vessel is visible on the long range scan map. The last column indicates armor: blank=none, L=light, M=medium, H=heavy, VH=very heavy.*

7.4 Deck Guns and Shells

The deck guns may be front, rear, or center mounted. The caliber of shell and number of shells fired per round differ from gun to gun. Each gun takes a certain amount of time to reload, which varies by a small random amount each time. Shell launch is animated as a puff of gray smoke.

The obvious way of implementing shells is to use complicated AI to predict the target coordinates at

[8] Some companies will ship a level editor with their game, but not necessarily the one they used to create the game!

launch time, and then compute (and perhaps even animate) the parabolic path of each shell. However, in real life moving shells are almost impossible to see. We instead modeled each shell as an abstract Markov process:

- Each shell is inserted into a queue on launch, tagged with the time of impact, which is a function of the distance to the sub at time of launch.
- At each frame, those shells whose time of impact has arrived are removed from the queue, and their impact is animated.
- For each shell impact, the probability p of hitting the sub is computed. This is a function of the distance that the shell has traveled, the number of course and speed changes that the sub has made recently, and a random term. A biased coin with probability p of success is flipped. If the coin flip is successful, a shell hit on the sub is animated. Otherwise, a shell miss (a splash in the water) is animated, the location of which is a random function that depends upon the sub's current location and most recent change in speed and direction.

The end result is that the player sees a shell launch, waits an amount of time that appears proportional to distance, and then sees either a shell hit or a shell miss. Evasive action increases the probability of a shell miss, but the random factor ensures that it is not a sure thing.

The guns are programmed to fire at sub-launched torpedoes and missiles in preference to the sub. A shell splash that is sufficiently close to a torpedo will detonate it prematurely. This will curb the hit-rate of both the player and the enemy vessels.

7.5 Torpedoes and Missiles

The sub has three types of torpedoes and two types of missiles. These include:

- Dumb torpedo: Travels in a straight line from launch.
- Homing torpedo: If locked onto a target at launch (indicated by an icon on the HUD), it actively seeks its target. Travels in a random path if launched before target is acquired (which usually results in a hit on the sub).
- Decoy torpedo: Has no warhead, but if launched underwater will decoy an enemy vessel that has depth charges, provided the sub is neither moving too fast, nor has active sonar engaged.
- Stinger missile: The aerial equivalent of a dumb torpedo.
- Cruise missile (registered version only): May be wire-guided from the sub.

The straightforward way to perform collision detection between objects in the game is by comparing the location of every object to every other object in the game, which is an inherently quadratic algorithm. In practice we optimized the process as follows.

Firstly, there is no collision detection between ships. The appearance of collision is minimized (but not eliminated) by ensuring that motion planning has a built in sloppiness that prevents sea lanes from becoming too narrow, and by various AI decision such as ensuring that no ship is charging towards the ship while another is broadsiding it (see Section 5).

Secondly, collision detection between torpedoes and ships is simplified by keeping a count of the number of ships located in each tile (see Section 4.5). Only those torpedoes that are in the same tile as a ship actually go on to compute distances from each ship, and may thereby actually record a collision. A side-effect is that ships that are on the edge of a tile are very hard to hit. Since tile edge information is not available to the player, some torpedoes that appear to be hits actually miss the ship, a situation that mimics real life.

7.6 Other Weapons

There are other ship weapons in addition to shells and torpedoes. Some ships have depth charges, which are used when the sub dives below periscope depth. The ship makes repeated passes over the sub's last known position until the sub is either destroyed or becomes invisible for a sufficiently long period of time. The ship's attention may be diverted by decoy torpedoes (see Section 7.5). The depth charges are the only things that are actually visible in the periscope window (aside from a blue glow) when the periscope is underwater.

The EMF burst is a passive weapon that emits a wavefront visible to the player as an expanding ring of foam (see Section 6.3). The wavefront may detonate torpedoes and missiles, and temporarily impede the progress of the sub, or even damage it. Damage may be avoided by outrunning the wavefront or diving beneath it. Fortunately for the player, there is a short window of time while it recharges.

Finally, there are other miscellaneous projectile weapons used in the Leviathan mission set.

7.7 Static Objects

Static objects in *Subhunt* include:

- The battery powerup, which adds battery charge.
- The fuel powerup, which adds diesel fuel.
- The air powerup, which adds air.
- The repair powerup, which reduces repair time.
- The mine, which damages the sub. Mines may be individual or laid in strings, and may be proximity fused or time fused.
- The mystery powerup, which may be any of the above.

The player can activate the above objects by colliding with them. In addition, there are two other objects that the user may encounter:

- The pillbox, which has a substantial gun.
- The lighthouse, which has no function (other than eye-candy).

These objects may only be destroyed by missiles.

7.8 Winning and Losing

The player must destroy all enemy shipping in order to win a level and thus get awarded a medal and permission to proceed to the next level. As an aid, the number of remaining enemy ships is displayed in the top left corner of the control panel. Factors that may prevent the player from winning include:

- Running out of resources
- Taking too much damage
- Failing to hit the enemy.

Subhunt has three levels of difficulty. At the lowest level of difficulty there is no opposition from the enemy vessels, which makes game play a matter of hitting ships and collecting powerups. At the intermediate level of difficulty there is some opposition, but not all enemy vessel types may be represented. At the highest level of difficulty there are no holds barred.

The sub may take damage in several categories:

- Periscope damage is caused by collision with a floating object, or excessive hull damage. Mild damage results in broken glass obscuring the view, and heavy damage renders the periscope unusable.
- Propellor damage is caused by diving into the ocean floor. The effectiveness of the engines is inversely proportional to propellor damage.
- Torpedo launch system damage is caused by excessive hull damage. Any damage prevents the launch of torpedoes and missiles.
- Long range scan damage is caused by collision with a floating object, or excessive hull damage. Mild damage results in broken glass obscuring the view, and heavy damage renders the long range scan unusable.
- Sonar damage is caused by excessive hull damage. Any damage prevents the use of the sonar.
- Diesel engine damage is caused by submerging while the diesel is running. Any damage prevents the use of the diesel engine.
- Hull damage is caused by enemy weapon strikes and collisions with land, ocean bottom, or ships. Excessive hull damage will cause collateral damage to other systems, and will eventually lead to hull rupture and death. The depth at which hull rupture occurs is inversely proportional to hull damage.

Damage is recorded as elapsed time to repair. The player can opt to repair one item at a time. The repair powerup will reduce the repair time of the currently selected damage category if nonzero, or the next nonzero damage category otherwise.

8 Further Information

More information on *Subhunt* can be obtained from the *Subhunt* web page at the University of North Texas: http://hercule.csci.unt.edu/subhunt. In the US, it can be downloaded from Webfoot Games at http://www.webfootgames.com/catalog/subhunt.htm. More information about the publisher can be found at http://www.spectrumpacific.com.au.

9 Acknowledgements

Subhunt benefited immensely from the enthusiastic participation of students in the Department of Computer Sciences, School of Visual Arts, and College of Music at the University of North Texas. We are indebted to Karen Bravo, Keith Burlison, Feras Fanari, Gary Frye, Hillary Han, Brian Higgins, John Gotcher, Michael Howell, Michael Lagocki, Jeremiah Isaacs, Daniel Lara, Andy Lomerson, Mark Mann, Chris Philpot, Rebecca Rodgers, Marco Rosales, Evan Trickett, and Steve Wilson.

References

1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
2. R. Floyd, "Algorithm 97: Shortest Path", Communications of the ACM, 5(6): 345, 1962.
3. C. A. R. Hoare, "Quicksort", Computer Journal, 5(1):10-15, 1962.
4. M. Minsky, *Society of Mind*, Simon and Schuster, 1985.

Appendix A: Screen Shots

Figure 5 *Instrument panel view.*

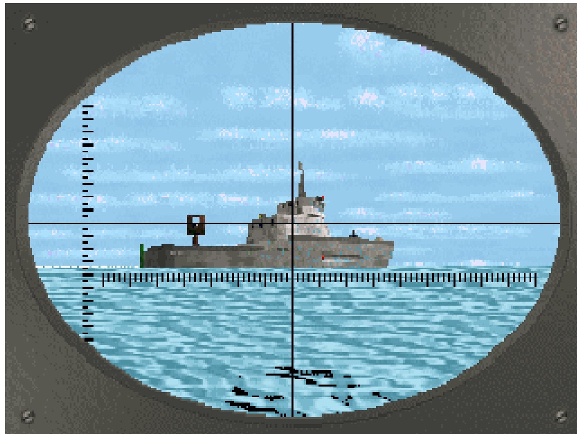

Figure 6 *Full-screen periscope view.*

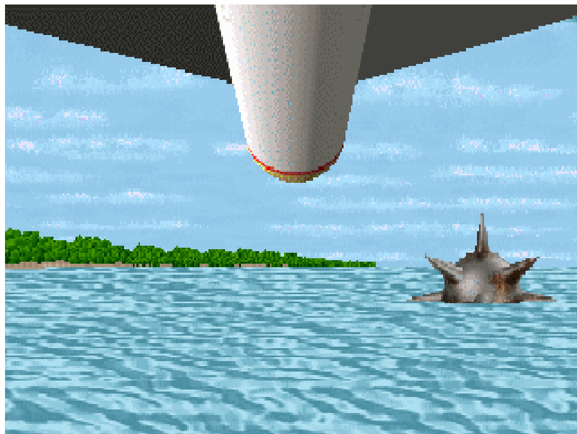

Figure 7 *Cruise missile camera view.*

Appendix B: Controls

Type	Command	Key
Propulsion	Faster	Pad +
	Slower	Pad -
	Stop	Pad *
	Diesel start/stop	E
	Supercharger	Tab
Attitude	Left rudder	Left arrow
	Right rudder	Rt arrow
	Down ailerons	Dn arrow
	Up ailerons	Up arrow
Sonar	Active	A
	Passive	D
	Off	S
Long range scan	Zoom in	Z
	Zoom out	X
	Left	H
	Right	L
	Up	K
	Down	J
	Lock on sub	Left shift
Heads-up display	On/off	T
	Scroll faster	Y
	Scroll slower	R
	Flush buffer	Q
Periscope	Rotate left	Pad 4
	Rotate right	Pad 6
	Stop rotation	Pad 5
	Look	Enter
	Spin to front	Pad 8
	Spin to rear	Pad 2
	Crosshairs on/off	Pad 0
Damage control	Start/stop repair	\
	Total	-
	Next	Backspace
	Previous	=
	Next damaged	[
	Previous damaged	]
Torpedoes	Launch	Space
	Load	Right Alt
	Unload	Right Ctrl
	Next	Left Alt
	Previous	Left Ctrl
	Camera	Right shift

Table 3 *Initial assignments for customizable keyboard commands.*

Type	Command	Key
General	Exit	ESC
	Pause	F1
	Custom settings	F2
	Help	F3
	Redraw screen	F4
	Save game	F5

Table 4 *Fixed keyboard commands.*

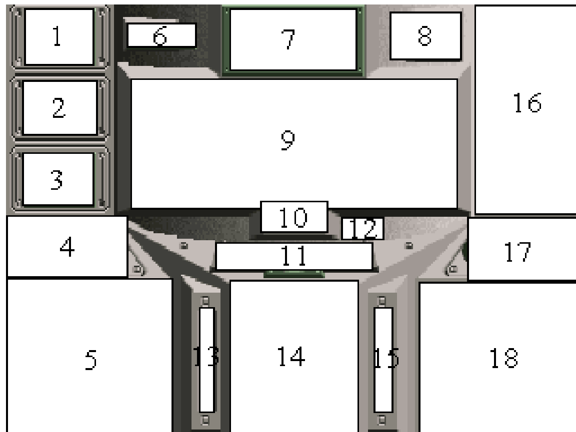

Figure 8 *Control panel: (1) enemy count, (2) fuel, (3) battery charge, (4) air, (5) launch control, (6) elapsed time, (7) hull pressure, (8) damage control, (9) periscope, (10) periscope control, (11) rudder, (12) speed, (13) ailerons, (14) sonar, (15) depth, (16) throttle, (17) compass, (18) long range scan.*

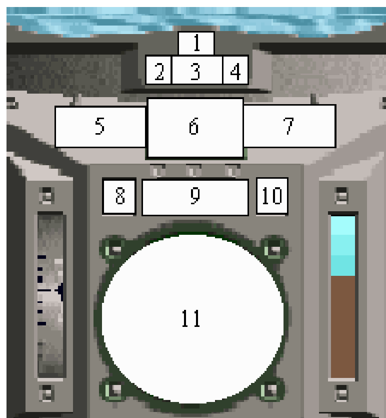

Figure 9 *Detailed view of the periscope, rudder, and sonar controls: (1) periscope stop button, (2) periscope left button, (3) periscope angle, (4) periscope right button, (5) rudder left button, (6) rudder dial, (7) rudder right button, (8) active sonar light, (9) sonar switch, (10) passive sonar light, (11) sonar screen.*

Figure 10 *Detailed view of damage control: (1) indicator light, (2) current item, and (3) repair time.*

Figure 11 *Detailed view of the torpedo launch controls: (1) current rack, (2) launch button, (3) loaded torpedo, (4) load button, (5) status light, (6) unload button, (7) previous rack button, (8) torpedo count, (9) next rack button.*