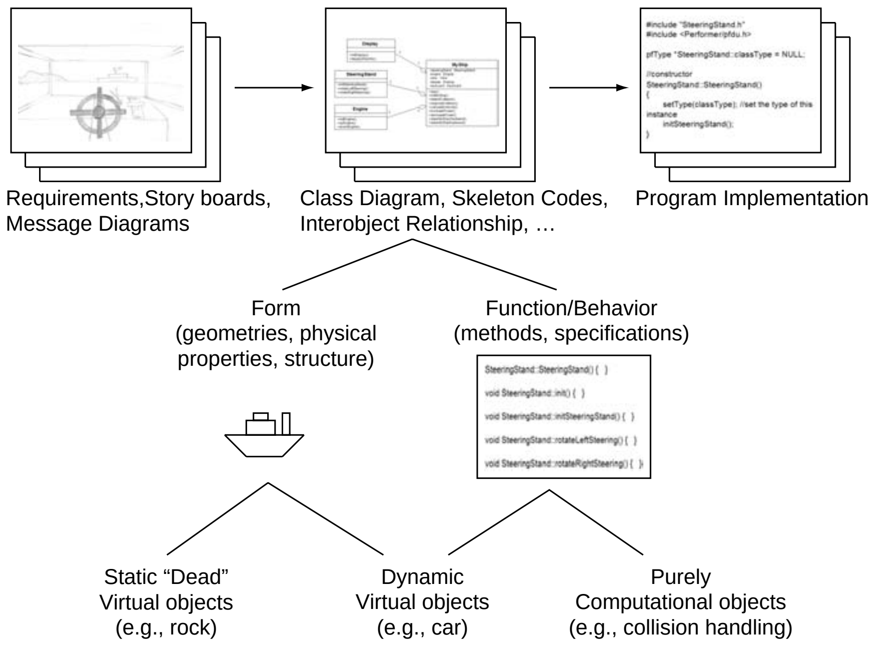
Chapter 2
Requirements Engineering and Storyboarding

Good system engineering practice is vital to the successful development of VR systems, more so than ordinary software systems because VR systems have multifaceted requirements (not just to make correct computations). In fact, a typical development process for VR systems will go through many cycles of revisions, as there is a lack of design guidelines on how to effectively integrate various resource-consuming computations and interactive techniques.

Thus, in building a VR system, we must start with identifying and describing its requirements. Requirements [IEE94] are statements identifying a capability, physical characteristic, or quality factor that bounds a product or process need for which a solution will be pursued. Requirements refer to the desired properties of the system and the constraints under which it operates and is developed. Requirements should be documented and specified as clearly as possible, for ease of revision and later maintenance. Although requirements engineering is a difficult and cumbersome process, it should be done at least for the important core part of the system. These descriptions are best captured and maintained using computational support tools and formalisms, but in actuality, even hand-drawn sketches and documents (such as the storyboards) would be useful [Cim04].

Requirements may be functional or nonfunctional. Functional requirements describe system services or functions. Nonfunctional requirements are constraints on the system or on the development process. There are many ways to go about doing requirements engineering for a VR system. For instance, we start with the functional requirements such as those about the scenes, virtual objects comprising the scene, behaviors, and the style of interaction.

Storyboarding is one way to start off the requirements engineering process. A storyboard is a visual script designed to make it easier for the director and cameraman to “see” the shots before executing them [Cri04]. It saves time and money for the producer and is used for making movies, commercials, and animation. There are structured ways to make storyboards, but for now, informal sketches and annotations suffice for our purpose (See Figure 2.1).



```
#include "SteeringStand.h"
#include <Performer/pfdu.h>

pfType *SteeringStand::classType = NULL;

//constructor
SteeringStand::SteeringStand()
{
      setType(classType); //set the type of this
instance
      initSteeringStand();
}
```

```
SteeringStand::SteeringStand() { }

void SteeringStand::init() { }

void SteeringStand::initSteeringStand() { }

void SteeringStand::rotateLeftSteering() { }

void SteeringStand::rotateRightSteering() { }
```

FIGURE 2.1. Modeling and implementing virtual objects in an object-oriented fashion.

The overall scenario, as represented in the simple form of sequences of "cuts" (or static scenes) in the storyboard, can be further refined and include some dynamics. One useful method is to use the Message Sequence Diagrams (MSD) [DeM79], or use cases [Car98]. The MSD depicts typical scenarios of internal and external behaviors of a VR world in terms of sequences of data or control signals exchanged among objects in the system (See Figure 2.4). Using the MSDs, one can test the system for later model validation, but more importantly, it enables the developer to identify important objects in the system. Constructing MSDs also aids in identifying the sequences of the messages among various objects and picturing how they interact with one another. In particular, external devices can be treated as an object for human–computer interaction. Object classes are then constructed by examining the identified objects and grouping them according to the commonality in their attributes.

Objects, better referred to as "virtual objects," are the constituents of a virtual environment through which the user will obtain the virtual experience. Although there is a natural mapping from virtual objects to the "objects" in the object-oriented programming paradigm, virtual objects are rather just a modeling concept at least at this stage. As these virtual objects are later implemented as "objects" in an object-oriented computational platform (which would be a natural thing to do), they are interchangeably referred to as both a modeling concept and a specific computational implementation. Virtual objects, for their physical connotation, indeed lend themselves naturally to the object-oriented system development methodology, and this book chooses to illustrate the implementation details using the object-oriented platform. Note that the object-oriented approach can be used to model the concrete virtual objects and scenes they compose in the VE, and to abstract various functional services required to execute and manage them, for instance, device management, rendering control, object and scene creation/consolidation/importing, event management and communication, process management, and so forth. We use the OpenSceneGraph [Ope04] and the SGI Performer[1] to illustrate many of the concepts explained in this book (actual code samples may be found on the companion CD). OpenSceneGraph is an open-source high-performance 3D graphics toolkit written entirely in standard C++ and OpenGL. SGI Performer is a popular commercial package for developing virtual reality applications.

For a large-scale virtual environment with many sorts of objects, sketching a rough object class diagram can be useful. A class diagram shows the existence of classes and their relationships in the logical and brief view format. The standard class diagram notation such as that of the Unified Modeling Language (UML) [Fow97] can be used. The diagram includes association, aggregation, composition, and inheritance relationships. Relationships provide a path for communication between objects. It is important to begin the overall modeling process with a consistent view of the object-orientation. With a clear picture of a system configuration in terms of constituent objects and information flows between them, the detailed specification behavior, function, and form for each object can begin.

Virtual objects, just like physical objects, can be characterized by three main aspects: the form, function and behavior. *Form* refers to the outer appearance of virtual objects, and their physical properties and structure.[2] We usually associate "appearance" with the visual sense (how it looks), however, a form or appearance must be judged with respect to ways it can stimulate humans through the display devices. Thus, form may include appearances also in terms of audition, haptics (force feedback), and other modalities that humans possess. For simplicity, we concentrate on the visual part for now, but later in the book, we will talk about modeling and simulation of nonvisual appearances. Other physical properties (which may be required for physical simulation) such as mass, material property, velocity, and acceleration may be included as part of form information.

[1] Performer is a registered trademark of Silicon Graphics, Inc. A free month-long evaluation version of Performer is available at www.sgi.com.

[2] Structure refers to the spatial/logical relationship among component objects in the case where the given object is a composite one.

Function refers to encoding what virtual objects do (i.e., primitive tasks) to accomplish their behavior (defined below), whether autonomously or in response to some external stimuli or event, and *behavior* refers to how individual virtual objects dynamically change and carry out different functions over a (relatively long) period of time, usually expressed through states, exchange of data/events, and interobject constraints. It is somewhat difficult to clearly draw the line between function and behavior. Functions may be viewed as primitive behaviors that are mostly atomic and taking a relatively short amount of time. Separating them, nevertheless, is useful for modular design of object dynamics. The description of objects, as part of a formal or informal specification of the overall application or system, must address these aspects. Note that there may be objects without form (purely computational objects such as device interfaces) or without function or dynamic behavior (e.g., static nonmoving objects such as virtual rocks).

So, for instance, the form specification/description would start by capturing the initial approximate shape/volume as well as the physical configuration of those objects (e.g., a simple hand-drawn sketch will do). As the description gets more mature and goes through a number of refinement iterations, the objects could decompose into smaller components (e.g., by breaking a car into its components, such as body, wheels, doors, etc.). Values of important attributes (e.g., size, color, mass, object type, etc.) may be added to this description as well. These descriptions are best captured and maintained using computational support tools and formalisms, but in actuality, hand-drawn sketches and documents (such as the storyboards) would still prove useful. More detailed explanations of the modeling and initial implementation process are given in Chapters 3 and 4.

Construction of virtual objects and their world often requires many revisions, and changing one aspect of the world will undoubtedly affect other aspects of it. For instance, different shapes and configurations (positions and orientations in space) can result in different dynamic behaviors. A jet fighter has different aerodynamic characteristics from that of a passenger airplane. Form can also affect functionality. For instance, two different robots differing in size may have different work volumes and capabilities. Such a development cycle is difficult to handle when working in a single level of abstraction and considering these design spaces in isolation.

Object functions and behaviors can equally be described using tools as primitive as plain text to more structured and diagrammatic representations such as procedural scripts, state transition diagrams, data flow diagrams, constraint languages, and the like. The choice of representation should be based on the complexity and nature of the object behavior and also on the type of behavior model supported by the VR development platform (so that the description can be easily mapped to and implemented at a later time). For instance, some game engines support state-based automata to express and implement intelligence into objects. Less fancy VR development platforms only support procedural programming for object behavior implementations. See Chapter 4 for more details. Figure 2.1 illustrates this initial modeling process as demonstrated in this book.

Another equally important functional requirement concerns user interaction. The storyboard and the MSD identify the important junctions and events at which user input is required. The task required to be carried out by the user should be refined to some degree and matched with the capabilities of the hardware devices and computational power of the computing hardware. The method of interaction modeling and interface design is treated in Chapter 5.

A related problem to interaction is the designation of the proper display devices. Different display systems are suited for different tasks and situations. For instance, HMDs are more suited for close-range manipulation tasks, whereas large projection displays are suited for navigation and walk-through application. Whether to employ head-tracking, haptics, 3D sound, and so on is an important interaction-related decision to make. Generally, sensors and displays cannot be changed during their use. They are also generally expensive, and one might not have the luxury of choosing the best possible displays and sensors. A clever design of the contents can overcome some of the limits introduced by low-end displays and sensors. Thus, at an early stage, having a rough idea of the nature of the user tasks and interactions (e.g., style of input and response to input) is helpful in determining the right displays and sensors and in recognizing the limits and bounds introduced by the hardware for providing a suitable level of presence and usability. Also note that there may be interaction objects (those that are purely functional such as device polling, or those also with form such as menus) to consider as well. Putting the user in the center of the system design process is very important as many VR systems fail simply because they are not user friendly.

The important nonfunctional requirements to consider at this stage are requirements for the overall system performance and device constraints. The performance requirement is rather simple. A virtual reality system is a real-time system, and must make computations for simulations, synchronize its output with various input devices, and maintain display updates at a rate at which human users will feel comfortable. For instance, for smooth computer graphic animations, the simulation for updates should be made at about at least $15 \sim 20$ times per second. Other input or display devices may require different timing requirements (for instance, haptic equipments ideally require an update rate of up to 1000 Hz for delivery of smooth force feedback). Note that 1/15th second is an amount relative to the capability of the computational and graphics hardware. Thus, if the functional requirement cannot be accommodated by the nonfunctional constraints such as the performance bounds or the devices, they have to be addressed in some way, either by making a business decision to purchase the appropriate equipment or later by designing to overcome the resulting distraction factors through clever content psychology. The important thing is that this be known in the early design stage.

Finally, a developer needs to understand, once again, that making these requirements and implementing them is an iterative process, starting from a rough picture and being refined stage by stage. To what degree should the requirements and implementation be done? That depends on the discretion of the developer.

Example: Ship Simulator Design

We illustrate this initial modeling process more concretely by illustrating the design of a simple virtual ship simulator. The objective of the example application is to assist trainees to navigate in and out of the pier and anchor without colliding with other vessels or the coast. Figure 2.2 lists the initial requirements for the simulator. Given these high-level goals and informal requirements of the system, we start with sketches of the storyboards as shown in Figure 2.3.

Requirements (Level 1)

- The virtual ship simulator (named *Ship Simulator*) helps users (named *User*) operate a vessel (named *My Ship*) and practice docking without colliding with other vessels (named *Other Ship*) or the coast.
- Initial View
  - The default view (named *Camera*) is the scene as seen from the control bridge where the *User* controls its ship (*MyShip*). The *User* can see the outside environment through the windows in the bridge.
- Interaction
  - The control bridge includes a steering wheel (named *Steering Wheel*) and an engine lever (named *EngineTelegraph*) for the *User* to steer and control the velocity of the *My Ship*.
  - The *User* can look around the interior of the bridge and change its view named *Camera*).
  - The basic mode of control via keyboard (named *Keyboard*) and mouse (named *Mouse*) must be supported. *Ship Simulator* shall accept input from the *Keyboard* to control *My Ship*.
- Models
  - The bridge includes a steering wheel (named *Steering Wheel*) and an engine lever (named *Engine Telegraph*).
  - The scene must also include object models for sky, sea, *other ship*, terrain, and pier.
- Simulation
  - *Ship Simulator* controls several automatically navigated vessels (*Other Ship*).
  - *Othership's* initial positions and moving directions are chosen randomly.
  - *Otherships* change their speed and directions every 10 seconds.

FIGURE 2.2. The initial requirements for the virtual ship simulator.

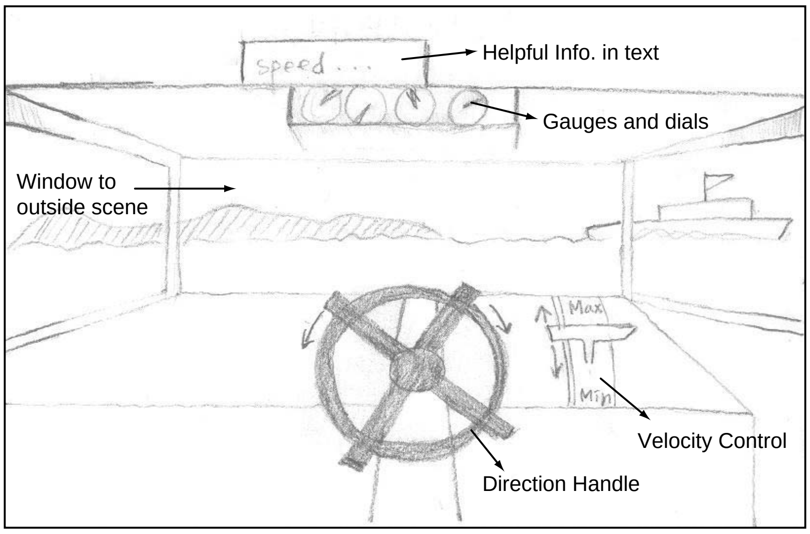


FIGURE 2.3(a). The default starting view of the *ShipSimulator*. The interior of the control bridge is seen with the steering wheel, engine lever, outside view, and gauges. The *User* can look around the control bridge.

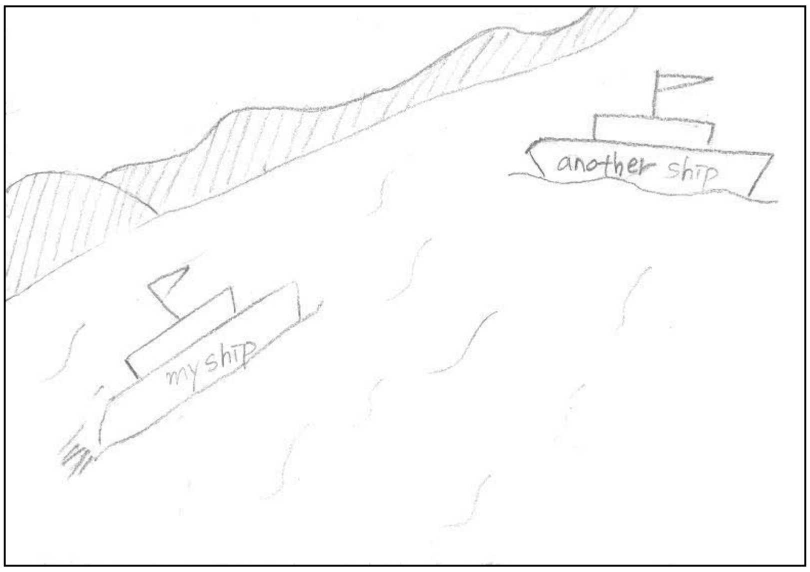


FIGURE 2.3(b). The external view of Figure 2.3a. A number of ships (including *MyShip*) move around the sea. This view can be selected by separate keyboard/ mouse control.

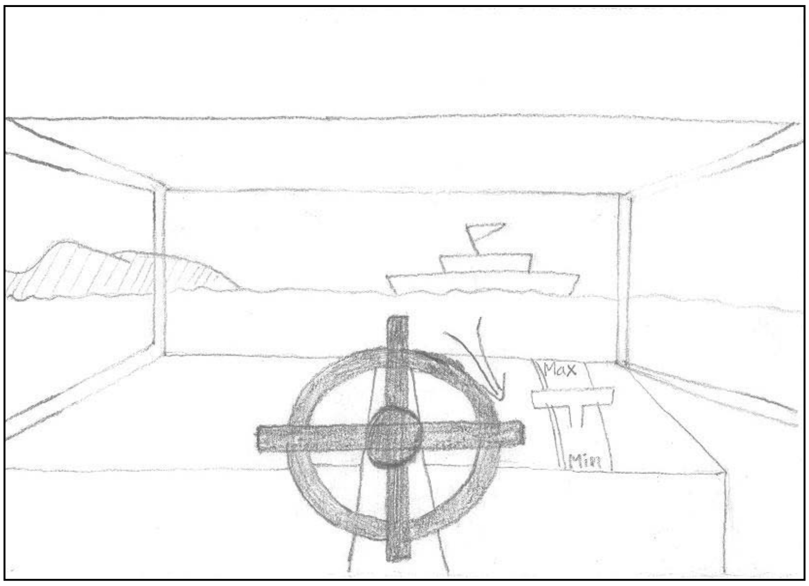


FIGURE 2.3(c). As the *User* steers the ship using the handle (named *SteeringWheel*), the scene through the window is changed accordingly.

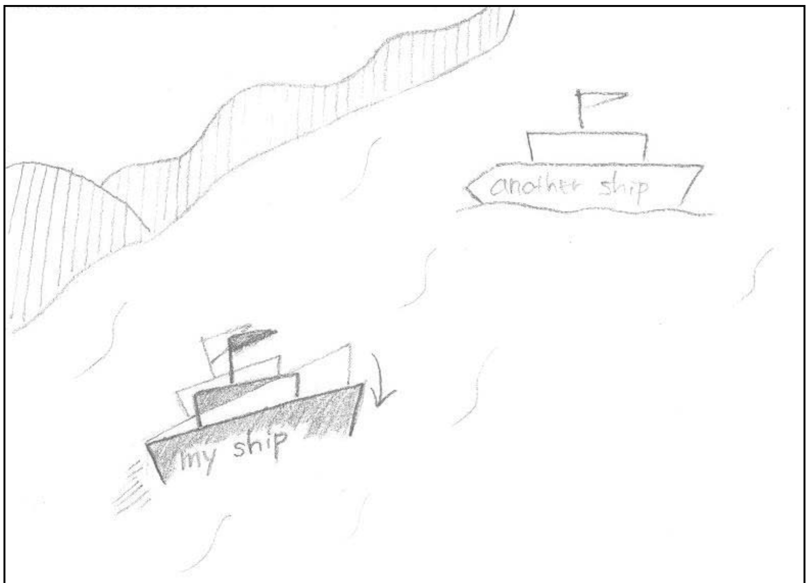


FIGURE 2.3(d). The external view of Figure 2.3c.

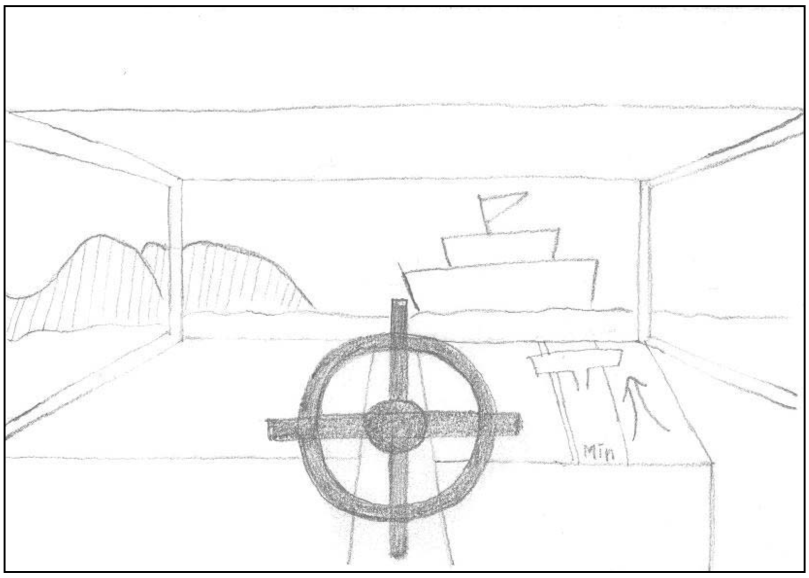


FIGURE 2.3(e). As the *User* manipulates the engine lever (named *EngineTelegraph*) and controls the velocity of *MyShip*, the scene through the window changes accordingly.

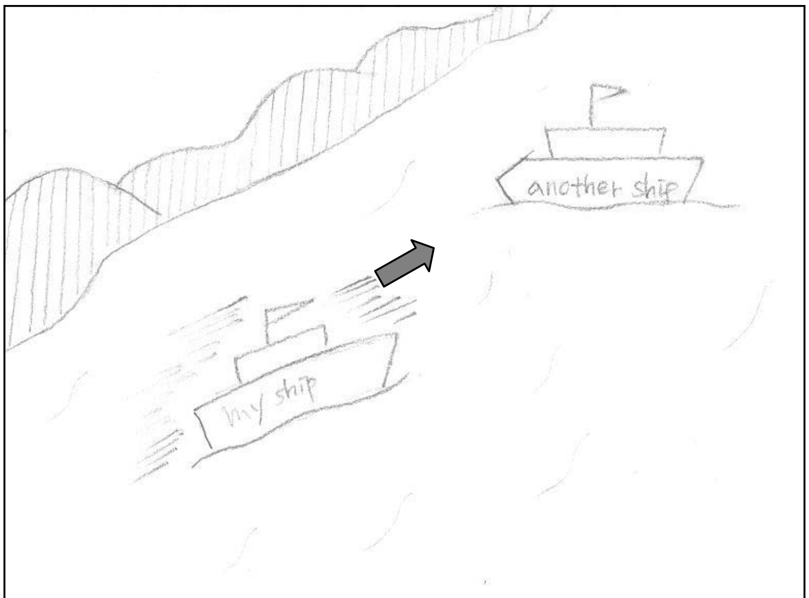


FIGURE 2.3(f). The external view of Figure 2.3e.

As shown in this simple storyboard, the three major objects are identified first: the trainee vessel (called *MyShip*), other automatically controlled vessels (called *OtherShip*), and the central simulation control module (called *ShipSimulato*r). *MyShip* is composed of, among other things, *SteeringWheel and EngineTelegraph* (the user interface for vessel control). We also identify an interface object: the *Keyboard* (for various ship and training control functions) and an object representing the camera position, *Camera*.

The specification starts by creating simple scenarios using the MSD as depicted in Figure 2.4. Figure 2.4a is the first simple example of the MSD, a trainee interaction scenario for "looking around" on the control bridge. When the *User* enters a key, it is stored by the interface object *Keyboard*, and the *User* checks what kind of keys were pressed (e.g., "z" for looking to the left), and the *Camera* is updated accordingly. A similar interaction scenario is given in Figures 2.4b and c where the *User* communicates to the *Keyboard* (pressing the up/down/left/right arrow keys) to control the speed and the course of *MyShip*. In Figure 2.4d, the *OtherShip* sets its own initial position and direction in a random fashion and changes its speed and direction periodically every 10 second.

An initial class definition (with major functionalities specified based on the content of the messages exchanged) and the class diagram is designed as depicted in Figure 2.5. Figure 2.5 shows the simplified class diagram created by constructing various MSDs. Notice that the interaction object

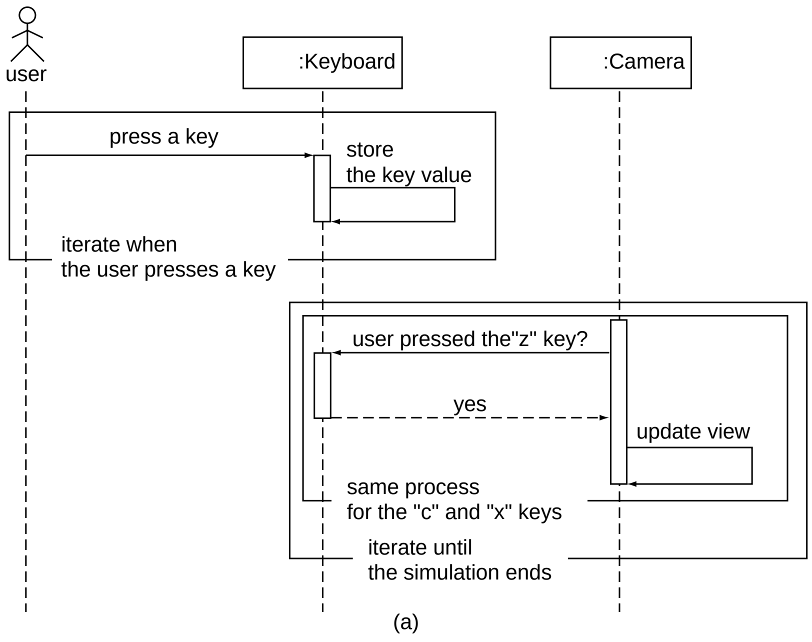
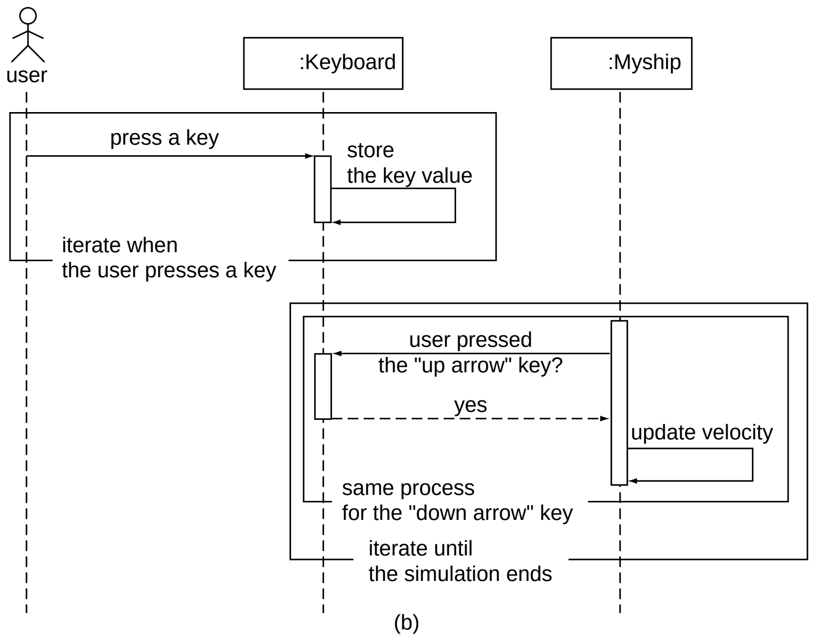


FIGURE 2.4(a). MSD for simple keyboard-based view control.



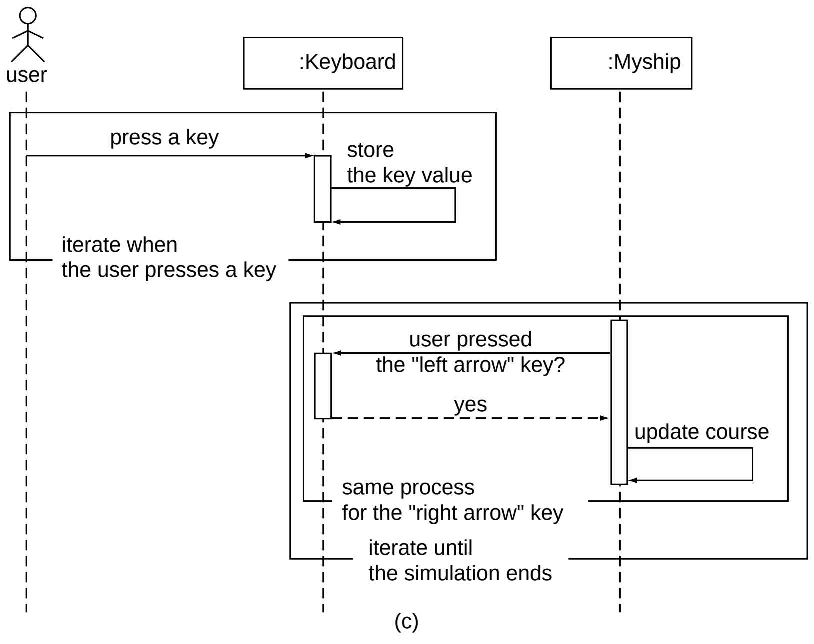


FIGURE 2.4(b),(c). MSD for controlling *MyShip's* velocity and direction using the arrow keys.

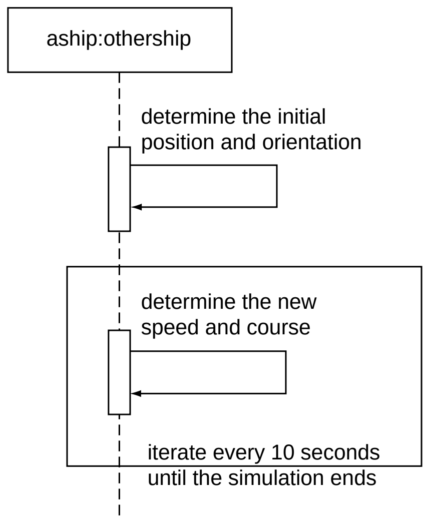

ashiр:othership
determine the initial position and orientation
determine the new speed and course
iterate every 10 seconds until the simulation ends

FIGURE 2.4(d). MSD for initializing and updating an instance of an *OtherShip*.

Keyboard and the *ShipSimulator* are purely "functional" without any form. As noted, the relations between classes are clarified at this stage of the modeling. A trainee can operate *MyShip* through *Keyboard*, then *MyShip* changes *Camera*. He or she can also change the orientation of *Camera* through *Keyboard* but the change in *Camera* does not affect *MyShip*. This initial class diagram will be subject to revision during the next phases of development.

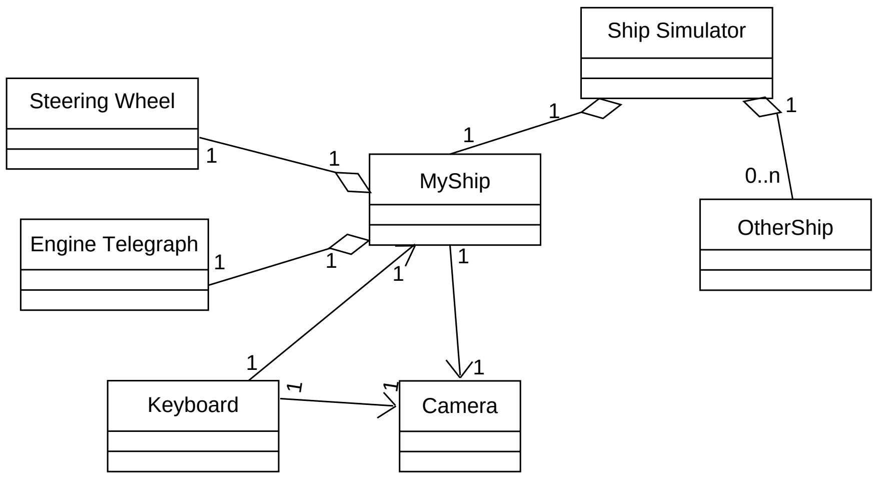


FIGURE 2.5. An initial class definition for the ship simulator.

Summary

VR system design starts with listing the requirements and carefully analyzing them as to whether virtual reality is even needed in the first place. The requirements must be centered around the user's expectation and capabilities. For instance, an experience-oriented requirements will result in a system with emphasis on presence, whereas a task-oriented requirements will place emphasis on efficient interaction. Based on the requirements, the overall scenario can be constructed using storyboards. "Virtual" objects that make up the scene are identified and the basic specifications for their form, function, and behavior should be made. Other aspects of the system such as device constraints, interaction, major special effects, and presence cues are also noted in this early stage of system development. Major interobject relationships are made more explicit by drawing class diagrams and message sequence diagrams.

Pondering Points

- Characterize the form, function, and behavior for a virtual human, virtual rock, virtual airplane, and virtual wind.
- What are possible barriers to making a VR system run in real-time?
- Make a case for, and against, carrying out requirements engineering at all.
- Make a case for, and against, using abstract formalism, support tools, or even documentation for requirements and system specifications.
- Is the object-oriented paradigm most fitting for implementing VR systems?
- Can having too many interaction points in the VR content be detrimental to inducing a good convincing virtual experience?
- In achieving the intended level of virtual experience, how can one make a good decision, for instance, between purchasing a special device for the increased effect, and staying with the less capable one and overcoming its shortcoming using other tricks?