Design of an Object-Oriented, Interactive Animation System

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 1 december 1995 om 14.00 uur

door

Eric Antonius Johannes Peeters

geboren te Zundert
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.dipl.-ing. D.K. Hammer
prof.dr. F.J. Peters

en de copromotor

dr.ir. C.W.A.M. van Overveld
Design of an Object-Oriented, Interactive Animation System
druk: wibro dissertatiedrukkerij, helmond.
Your easy reading is hard writing
[Ernest Hemingway]
Contents

1 Introduction .................................................. 1
  1.1 The DenK project ...................................... 1
  1.2 Computer animation .................................. 3
    1.2.1 Animation techniques ............................ 3
    1.2.2 Overview of animation systems ................. 6
  1.3 Motivation ............................................. 9
  1.4 Overview of this thesis .............................. 9

2 Generalized Display Processor .......................... 11
  2.1 Introduction .......................................... 11
  2.2 Traditional Architectures ........................... 11
  2.3 Generalized Display Processor 0 Architecture .... 13
  2.4 Generalized Display Processor 1 Architecture .... 14

3 Design of Looks ........................................... 17
  3.1 Introduction .......................................... 17
  3.2 Requirements ........................................ 17
  3.3 Useful Concepts ..................................... 19
    3.3.1 Object-Orientation .............................. 19
    3.3.2 Concurrency ..................................... 20
    3.3.3 Combined Object-Orientation and Concurrency .. 22
    3.3.4 Combined Object-Orientation and Concurrency in the GDP's script-language .......... 23
  3.4 Existing Languages ................................... 24
  3.5 Looks ................................................. 25
    3.5.1 Language Aspects ............................... 25
    3.5.2 Run-time Environment ............................ 58
  3.6 Evaluation ........................................... 60

4 Design of the GDP ......................................... 63
  4.1 Introduction .......................................... 63
  4.2 Internal Structure ................................... 63
  4.3 Parser ............................................... 65
  4.4 Interpreter .......................................... 66
  4.5 Renderer .............................................. 66
  4.6 Symbol-table ........................................ 67
    4.6.1 Class-table ...................................... 67
    4.6.2 Object-table .................................... 70
4.6.3 Example ......................................................... 73

5 Implementation Aspects ........................................ 77
  5.1 Introduction .................................................. 77
  5.2 Abstract Syntax Trees ........................................ 77
  5.3 Garbage Collection .......................................... 79
  5.4 Run-time Library Linking ................................. 81

6 Class Library .................................................... 85
  6.1 Introduction .................................................. 85
  6.2 Fundamental classes ........................................ 85
    6.2.1 Basic Classes ........................................... 86
    6.2.2 Data-structure Classes ................................ 89
    6.2.3 Input and Output Classes ............................. 90
    6.2.4 Position and Vector Classes .......................... 94
  6.3 Collection .................................................... 96
    6.3.1 Description .............................................. 96
    6.3.2 Implementation ......................................... 98
  6.4 Events .......................................................... 99
    6.4.1 Description .............................................. 99
    6.4.2 Implementation ......................................... 103
  6.5 Graphical User-Interfaces .................................. 104
  6.6 Rendering ...................................................... 107
    6.6.1 Geometry ............................................... 108
    6.6.2 Topology and primitives ............................... 114
    6.6.3 Light sources ........................................... 116
    6.6.4 Camera .................................................. 120
  6.7 Direct Manipulation ......................................... 123
    6.7.1 Selection ................................................ 123
    6.7.2 Control .................................................. 126
  6.8 (Inverse) Kinematics ......................................... 130
    6.8.1 Introduction ............................................. 130
    6.8.2 Prism .................................................... 134
    6.8.3 Revolute ................................................ 136
    6.8.4 Cylinder ................................................ 137
    6.8.5 Helix .................................................... 138
    6.8.6 Plane .................................................... 139
    6.8.7 Sphere ................................................... 141
    6.8.8 Isomorph ............................................... 143
  6.9 Dynamics ...................................................... 144

7 Sample Animation ............................................... 147
  7.1 Introduction .................................................. 147
  7.2 Synopsis ..................................................... 147
  7.3 Script ........................................................ 148
  7.4 Frames ........................................................ 153
Contents

8 Conclusions and Future Work 157
  8.1 Applications and Conclusions ........................................ 157
  8.1.1 Applications .................................................. 157
  8.1.2 Conclusions .................................................... 158
  8.2 Future Work .................................................................. 158
    8.2.1 Multi-Processor System ......................................... 158
    8.2.2 Distributed Multi-User System ................................. 159
    8.2.3 Rendering ....................................................... 160
    8.2.4 Executable Media .............................................. 160

A Specification 163
  A.1 Syntax ................................................................. 163
  A.2 Semantics ........................................................... 167

B Concrete Syntax of Looks 175

C Abstract Syntax of Looks 177

D Semantics 179
  D.1 Type Definitions .................................................... 179
  D.2 Evaluation Functions ............................................. 179
    D.2.1 Eval ............................................................ 179
    D.2.2 Eval_sml ..................................................... 185
    D.2.3 Eval.sml .................................................... 185
    D.2.4 Eval_program ............................................... 186

E Symbol-table Functions 187

Bibliography 191

Index 197

Summary 203

Samenvatting 205

Acknowledgements 207

Curriculum Vitae 209
Chapter 1

Introduction

1.1 The DenK project

In 1989, the Eindhoven-Tilburg Organization for Inter-University Cooperation (SOU) started the research project DenK (an acronym of the Dutch translation of Dialogue management and Knowledge acquisition; see Sob89 for the original project proposal). The goal of this project is the development of a generic, multi-modal user-interface (GMUI):

development: the DenK research project should result in a demonstrable prototype GMUI.

generic: this GMUI should be applicable in a variety of application domains (e.g. computer assisted operator training, virtual laboratories, and simulators).

multi-modal: in the GMUI, user interaction is supported via several UI-devices or modes, viz. graphical (mouse and graphical screen) and textual (via keyboard and text window, both in formalized and semi-natural language).

As described in [Be93] this user-interface should be based upon the real situation metaphor of two (dialogue-) partners having access to a common workspace. The (human) user of the interface is one of the partners, the other partner and the workspace are both simulated by the interface implemented on a computer. Since the simulated partner tries to fulfill all wishes from its human partner, it will be called the cooperative assistant. The textual communication mode between the two partners consists of fragments in natural language:

- from user to the cooperative assistant: the user can command the assistant to perform a task in the workspace or he can ask the assistant a question concerning the workspace.

- from assistant to the user: the assistant responds to commands by performing the requested task and to questions by giving the correct answer in natural language.
Chapter 1. Introduction

The communication between the human partner and the workspace consists of:

- **from user to the workspace**: selection and direct manipulation of objects in the workspace, e.g. by means of a mouse.
- **from workspace to the user**: up-to-date visual representation of the workspace, via realistic rendering of moving 3-D virtual scenes.

The communication between the cooperative assistant and the workspace consists of:

- **from assistant to the workspace**: the assistant can modify and inspect the workspace.
- **from workspace to the assistant**: the workspace responds to modifications by changing itself and to inspections by giving correct results.

The relation between the human user, the cooperative assistant, and the common workspace is depicted in Figure 1.1.

![Diagram](image)

**Figure 1.1**: The relation between the user, the assistant, and the workspace

In order to tailor the GMUI for a dedicated application domain, both the common workspace and the cooperative assistant should be instantiated properly: the workspace should be informed about the appearance and behavior of the objects in the domain, while the assistant should be informed about the facts and reasoning rules of the domain.

Since the workspace can change over time, either by its own autonomous behavior, direct manipulation by the user, or modification by the cooperative assistant, the (visual) communication from workspace to user can be viewed as an animation of the evolving workspace.

**Definition 1.2**: animation

Animation is a collection of changes over time that have a visual effect.

For example, an animation may consist of changes over time of position, shape, color, lighting, camera position, camera orientation, etc.

**Definition 1.3**: animation system

An animation system is a system that can be used to create animations.
1.2. Computer animation

The workspace can therefore be implemented on an animation system provided this system supports:

- autonomous behavior of the animated objects in the workspace
- modification of the (behavior of) the objects in the workspace by the cooperative assistant
- inspection of the status of these objects by the cooperative assistant
- direct manipulation of these objects by the user.

Furthermore, the animation system should fully comply with the generic character of the multi-modal user-interface of the DenK project.

The design of an animation system that fulfills the above requirements is the main subject of this thesis. In order to be able to compare it with existing animation systems, the next section contains a brief introduction and overview of previous results in computer animation.

1.2 Computer animation

With the advent of fast computers and sophisticated graphical hardware in the last decade, the variety of computer animation applications has grown enormously. Nowadays computer animation is used in a diversity of professions, for example:

- in simulators: to train personnel to react to all kinds of emergencies, thereby reducing the risks when such events really occur
- in architecture: to study the effects of a new design with respect to the environment and to the commodiousness
- in hospitals: by medical staff to study different aspects of a surgery before the actual performance, in order to anticipate to possible complications
- in commercials and the entertainment industry: the film ‘Jurassic Park’ by Steven Spielberg, for example, is one of the greatest box-office hits of all-time, not in the least because of the impressive animation of the dinosaurs.

This section describes the techniques that are used in computer animation (1.2.1) and the incorporation of these techniques in various animation systems (1.2.2).

1.2.1 Animation techniques

The foundation of animation is the discovery in 1824 by Peter Mark Roget that the human eye sees a motion when pictures are shown at an appropriate rate (see Chapter 1 in [Joh89] for a summary of early theories and equipment). In early animation devices, animation was achieved by means of a rotating cylinder that contained a (small) set of pictures. The animations were therefore very short, periodic, and also rather involving to produce. The real breakthrough came in 1889 with the introduction of celluloid film, which allowed for picture-for-picture recording, first carried out by Blackton in 1906. Now, animations could have arbitrary length and were easier to produce.
At first, all pictures (or frames in animation terminology) were drawn by hand before they were recorded on film. Naturally, this method is very time-consuming considering that more than 1000 pictures are necessary for just one minute of animation. To speed up this process, the technique of cel-animation was introduced in 1914 by Earl Hurd. In his approach the main characters and backgrounds are drawn with non-transparent ink on transparent cells, which may be combined to generate a frame. The number of drawings is indeed reduced considerably, but the quality of the animation is poor, since few of the drawings actually change over time: most frames use the same cells - albeit in changing configurations.

To improve the technique of cel-animation, a new technique was developed: key-framing, where the animators drew only the important frames (or key-frames) of the animation. The task of drawing the frames between the key-frames (logically called inbetweening) was carried out by less qualified artists. One of the studios where this technique reached its perfection was the studio of the illustrious Walt Disney, e.g. the first animated feature length motion picture 'Snow White and the Seven Dwarfs' (1938) was produced by his company.

When computers with graphical devices became available (1950s), they were first used to visualize the results of scientific computation [Zaj66]. Later it was realized that computers could also be used to facilitate the elaborate inbetweening process of more traditional animation production [BW71]). Consequently, much research went into the development of methods to specify the automatic inbetweening of key-frames. In computerized inbetweening, the key-frames are represented in the computer, while some numerical attributes (e.g. coordinates and angles) are interpolated in order to generate the other frames. Although more advanced techniques for computer animation were introduced afterwards, computer assisted key-frame animation is still used frequently.

In fact, a somewhat more sophisticated version of inbetweening became quite popular in recent years under the name of morphing, and, for example, it was used to produce the famous sequence of metamorphoses of human heads in the 'Black or White' clip of Michael Jackson (see [Bei92] for a detailed description of the applied technique).

Basically, there are two ways to specify the inbetweening process: script-oriented or graph-oriented. In the former approach the inbetweening is specified by means of a script\footnote{The input of scripted key-frame animation consists thus of two parts: key-frames and a script.}, an animation specification in a formal language. In the latter approach the inbetweening is specified by means of a graph that can be manipulated by the mouse. These approaches are not mutually exclusive, but may be used together [Sch86].

Unfortunately, key-frame animation has two (major) drawbacks. First, the animator has to know the exact sequence of frames in the animation in advance. Nevertheless, the term interactive key-frame animation is used in literature. Normally, this term means either the interactive design of a key-frame animation by means of the repetitive 'trial-and-error' method (i.e. the animator repeatedly views and changes the animation until he or she is satisfied), or it means the interactive control of the inbetweening process. The second drawback is that natural phenomena may be very hard to specify by means of key-frames. In order to overcome these problems, many techniques have been proposed; they can be classified in the following mainstreams: (inverse) kinematics, (inverse) dynamics, constraints, algorithmic animation, and mo-
1.2. Computer animation

tion recording (this classification is based on [Wil87]; a more elaborate introduction to (inverse) kinematics and dynamics can be found in [REG+91]).

In the kinematics approach, an animation is specified by supplying the trajectories, over time, of parameters that describe the characters in an animation. Examples of such parameters are angles, distances, etc. Inverse kinematics, on the other hand, deals with inferring these (parameter) trajectories from given positions of designated parts of the animated characters. Both approaches typically apply to articulated bodies, complex characters that consist of an assembly of elements with restricted relative motion. For example, Girard uses inverse kinematics for these bodies to imitate the motion of legged animals [Gir87]. The following example clarifies these techniques: suppose the position of an arm is parameterized by an elbow angle and a shoulder angle (note: in this example the arm is an articulated body). In the kinematics approach, these two angles vary over time, and this defines the arm movement over time. In the inverse kinematics approach, a trajectory of some end effector of the character (say, the hand) is given, and the angle parameters are inferred from this trajectory.

To facilitate the simulation of natural phenomena, the technique of dynamics can be used. Here, the animated bodies are supplied with masses, inertial tensors, accelerations, etc. Their movements can thus be calculated using Newton's law (i.e. the force applied to a body equals its mass times its acceleration). Like kinematics, dynamics also has a counterpart: inverse dynamics, where forces are computed that are necessary to reach a specified goal. In recent years, many methods were proposed to apply these approaches to articulated bodies (see e.g. [WMS88, ADH89, Ove91]). Again these techniques are clarified by an example: suppose the arm from the previous example has a mass. In the dynamics approach, the forces applied to the arm vary over time, and this defines — by means of Newton's law — the acceleration of the arm over time. Consequently, a numeric integration method can be used to calculate the position over time. In the inverse dynamic approach, again a trajectory of some end effector is input, but this time the forces that should be applied to the arm in order to have the end effector move along the designated trajectory are computed.

Since the specification of the animation using inverse kinematics or inverse dynamics consists of giving the end-positions of the moving bodies (while the movements of the other elements are calculated automatically), these techniques are commonly known as goal-directed animation.

Both kinematics and dynamics are special cases of constraints, i.e. relations between (properties of) bodies that are maintained by the system. This maintenance mechanism adds to the impression of autonomous behavior of animated bodies. For example, constraints have been used in animation to simulate waving cloths [OL92] or other flexible objects [PB88]. Another type of constraint is the property that bodies cannot move through each other. To prevent this from happening, techniques are introduced for collision detection and collision response, i.e. the actions that are taken whenever a collision is detected (see e.g. [MW88, Bar89]).

Sometimes the above techniques are too limited to control an animation, e.g. animating algorithms or growing trees are hard to specify kinematically or dynamically. Moreover, these animations may be easier to specify by means of an algorithm. This approach is therefore called algorithmic animation control. Consequently, animation systems that support this kind of control must have very powerful script-languages (e.g. the script-language ASAS [Rey82] is an extension of the programming
language LISP). Naturally, all of the above techniques can be incorporated in an algorithmic controlled animation system, because they can be implemented using a script-language.

Finally, animations, especially the movement of human figures, can be specified by means of motion recording. It consists of real-time recording the movements of human actors and using these recorded movements in the specification of an animation [MP90]. Furthermore, the recorded motions can be interpolated to increase the usefulness of the animation system. However, since the movements are pre-recorded, interaction with an animation of this kind is rather limited.

1.2.2 Overview of animation systems

In the last decade many animation systems have been introduced. The characteristics of some representatives of this set are summarized in this section. Unfortunately, the designers of these systems seem not always to agree upon the terminology used. Consequently, the terminology has to be defined explicitly.

**Definition 1.4 : universal animation system**

A universal animation system is an animation system that can be used for a great variety of animation domains.

The opposite of a universal animation system, a dedicated animation system, is a system that is tailored for a specific domain, e.g. the animation systems Gesture and Human Factory are dedicated to animation of human movements.

**Definition 1.5 : direct modification**

Direct modification means that a running animation can be controlled by means of script fragments.

**Definition 1.6 : direct manipulation**

Direct manipulation means that a running animation can be controlled by means of a control device, e.g. a mouse.

**Definition 1.7 : interactive animation**

Interactive animation means that a running animation can be controlled by means of direct modification or direct manipulation.

Table 1.8 (page 9) contains the characteristics of current animation systems with respect to: universality, direct modification, direct manipulation, and motion specification (the symbol ✓ that is used in this table means that a particular feature is at least partly supported by an animation system). The following animation systems are included:
1. **Advanced Visualizer [Vin92]**
   Advanced Visualizer is a commercial animation system developed by Wavefront Technologies. It supports key-framing and algorithmic control (even for articulated bodies). Furthermore, Advanced Visualizer supports channels, i.e. sources that provide information which can be used to control the animation; for example, external simulation tools may use channels to control an animation in the Advanced Visualizer.

2. **ASAS [Rey82]**
   ASAS is an animation system that is based on the LISP programming environment. To facilitate the definition of simultaneously moving bodies, ASAS's script-language supports *actors*, i.e. modular animated program structures.

3. **Avenue [DAU88]**
   Avenue is an animation system that supports both key-framing and algorithmic control. The latter is based on the logic programming paradigm (the language Prolog is an exponent of this paradigm). Consequently, an Avenue script consists of facts (e.g. there is a green body somewhere in space) and rules (e.g. green bodies will always be rotating, and rotating bodies become red). By applying the rules to the known facts, Avenue is able to generate animations (e.g. the green body will start to rotate while turning red).

4. **Clockworks [BGA+87, GB90]**
   Clockworks is an integrated environment for modeling, image synthesis, animation, and simulation. All motion control (e.g. key-frames and the corresponding interpolation information) is translated by the Clockworks to a script in an object-oriented language. Furthermore, the Clockworks allows for animations to be partly controlled by external processes.

5. **Controller [Joh89]**
   Controller is an animation system that is aimed at the production of animations for entertainment. Motions in Controller are specified by means of curves (dedicated curves are available in order to mimic dynamics).

6. **Explore [Vin92]**
   Explore is a commercial animation system developed by Thomson Digital Image. It consists of several modules: FACE for modelling, ANIM for animation, and RENDER for rendering. The constraints supported by Explore can only be used to control the inverse kinematics of articulated bodies.

7. **Gesture [MC96]**
   Gesture is an animation system that uses a high-level script-language to generate an animation of human beings. Consequently, the script-language contains specific movements (e.g. walk, scratch, etc.). For each of these movements Gesture has a description how to perform them (e.g. by using dynamics, kinematics, or key-frame data) — the actual motion calculations are carried out by external programs.

8. **Human Factory [MTT87]**
   Human Factory is an animation system whose main purpose is the direction of
synthetic actors. It supports key-framing, kinematics, dynamics, and algorithmic control. The script-language is also aimed at the aforementioned purpose, and supports, for example, procedures to control the muscles of the actors.

9. Kaya [WMS88]
Kaya is an animation system that is developed to explore low-level control of articulated bodies by means of dynamics (although kinematics is also supported). Furthermore, Kaya is able to perform collision detection and response.

10. Miranim [MTTF85]
Miranim is an animation system that consists of two parts: Body-Building for modeling and Animedit for motion specification. The latter is intended for animators who do not know how to program: they can specify an animation by applying pre-defined motions to bodies. However, the system can be easily extended to support more dedicated motion primitives. For this purpose Miranim contains the language Cinemira-2.

11. Pinocchio [MP90]
Pinocchio is an animation system that mainly supports motion control by means of recorded (human) movements. Furthermore, Pinocchio contains a script-language that facilitates both the interpolation between recorded motions and the specification of autonomous actors using these motions.

12. PowerAnimator [Vin92]
PowerAnimator is a commercial animation system developed by Alias Research. It only supports motion control by means of key-framing and kinematics, where the trajectories can be specified by means of curves.

13. SoftImage 3D [Vin92]
SoftImage 3D is a commercial animation system developed by SoftImage. Besides key-framing, it supports kinematics and dynamics for articulated bodies which can be controlled by means of direct manipulation. The supported constraints are restricted to connections between bodies and to collision detection.

14. Solar [CWC88]
Solar is an animation system that contains an object-oriented script-language. Solar motions are specified by means of curves that are stored in ‘motion-objects’. Next, these ‘motion-objects’ are linked to ‘graphical-objects’ which then perform the specified motion (multiple ‘motion-objects’ can be linked to a single ‘graphical-object’ in order to generate complex, hybrid motions).

15. Symbolics [Vin92]
Symbolics is a commercial animation system that supports key-framing animation controlled by means of curves. These curves can be specified manually in a graphics window, or they can be specified by means of script fragments. Since Symbolics is based on a LISP interpreter script fragments can be added without restarting the animation (note that this does not imply that Symbolics supports direct modification: indeed, these fragments cannot be used to control a running animation).
1.3. Motivation

16. Walt [Ov91, Nie92]

Walt is an animation system that supports several kinds of motion control. Furthermore, an animation in Walt can be influenced by means of direct manipulation. It also contains a multi-track recorder which can be used to record and playback the movements of selected bodies. Currently Walt serves as a test-bed for animation algorithms developed at Eindhoven University of Technology, e.g. a constraint algorithm [OL92] and a collision detection algorithm [Kel93b].

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>universal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dir. modification</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dir. manipulation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>key-framing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>kinematics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>inv. kinematics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dynamics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>inv. dynamics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>constraints</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>alg. control</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>motion rec.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1.8: Characteristics of current animation systems

1.3 Motivation

The most important motivation for the research described in this thesis, is the fact that currently no (universal) animation system exists that is capable of both direct manipulation and direct modification. Consequently, no existing animation system can be used to describe the (autonomous) workspace in the DenK project. Furthermore, current animation systems often offer two modes: a modelling mode and an animation mode. It is usually not possible to change the model during the animation. Although this is not a big problem for many animations, it definitely restricts the universality of the animation system (e.g. it hinders the development, over time, of animated objects, such as growing plants).

A similar problem occurs when using articulated bodies: in many animation systems it is only possible to specify the hierarchy of these bodies before the animation. Nevertheless, changing the hierarchy during an animation is sometimes very useful: for example, when a man should throw a ball to another man, the ball should at first be connected with the thrower, then it should move freely through space, and finally it should be connected with the catcher.

1.4 Overview of this thesis

The remainder of this thesis consists of 7 chapters. Chapter 2 discusses the Generalized Display Processor (GDP) architecture, an object-oriented universal animation system
that supports direct manipulation and direct modification and that is therefore suitable for the representation of the common workspace in the DenK project. Chapter 3 presents Looks, the object-oriented script-language of the GDP. Chapter 4 addresses the design of the GDP, while Chapter 5 discusses some aspects of an object-oriented implementation. Chapter 6 presents some classes that are useful in such an animation system. For example, the generic class collection is designed to facilitate existential and universal operations in direct modification. Chapter 7 describes a sample animation, written in Looks, that contains inverse kinematics motion in combination with direct manipulation. Finally, Chapter 8 summarizes the results of this thesis and also presents directions for future research.
Chapter 2

Generalized Display Processor

2.1 Introduction

This chapter discusses the architecture of an animation system that fulfills the requirements from the previous chapter and which is consequently suitable for the representation and visualization of the workspace of the DenK project. First, Section 2.2 presents three traditional architectures that are used to display images on a screen. Second, Section 2.3 explains why these architectures are not very well suited for a universal animation system, and it discusses the Generalized Display Processor architecture from [Ov91] that is devised to be used in such an environment. Finally, Section 2.4 argues why this Generalized Display Processor architecture does not satisfy all of the aforementioned requirements for the DenK project, and it shows how this architecture may be modified in order to fulfill these requirements.

2.2 Traditional Architectures

Modern computer systems use graphical displays based on CRTs (Cathode Ray Tubes), that have the property that the screen-contents decay very quickly. Consequently, the screen-contents should be redrawn (or refreshed) fast enough, in order to suggest the human eye that the screen-contents are permanent (if the refresh-rate is too low, approximately below 60 Hz, this refreshing process is experienced as a ‘flickering’ screen).

Since refreshing the screen is a repetitive, time-consuming process, it should not be performed by the Central Processor Unit (CPU) because otherwise:

- programmers have to anticipate the interruption of their programs due to the refreshment of the screen;

- programs would run slowly on such a CPU, since some of its time the CPU is refreshing the screen.

Therefore, architectures were developed where specialized video hardware draws the screen-contents on the graphical display [FDFH90]. In these architectures the CPU and video hardware share a memory module that contains a description of the screen-contents. Now, the CPU only has to update the contents of the memory module in order to change the contents of the screen. Since the CPU should also be able to
inquire about these contents, a bidirectional communication channel exists from the memory module to the CPU. Figure 2.1 shows the functional components of these architectures.

![Figure 2.1: Specialized Video Hardware](image)

Based on technical developments, several architectures were introduced in the last decades, for example:

1. **vector-graphics**
   When the first of these architectures was developed in the 1960s, memory modules were slow and expensive. Therefore, the minimization of the amount of memory was an important design issue. This goal was reached by restricting the screen-contents to line segments only (hence: vector-graphics), where only the begin- and end-positions of a line segment are stored in memory. Since the memory module contained a list of graphical primitives (i.e. line segments), it became known as a display list (DL); the video hardware that used this DL was named a display processor (DP).

2. **raster-graphics**
   When memory modules became faster and cheaper in the 1970s, the need to minimize the amount of memory became less prominent: it became possible to store the screen-contents as a raster (hence: raster-graphics) of picture elements (pixels). Since each pixel can be controlled individually, raster-graphics allows for much more realistic rendering (on sophisticated hardware even photographic realism can be achieved). The memory module and video hardware of this architecture are known as frame buffer and video controller respectively.

3. **raster-graphics with DP**
   In the raster-graphics architecture the CPU may have to update many pixels in the frame buffer (e.g. filling a large polygon with a color). In order to minimize the amount of time the CPU has to deal with filling the frame buffer, a dedicated processor is introduced (again, the name display processor is used). The communication from CPU to DP is performed in terms of graphical primitives, such as line segments and polygons, whereas the communication from DP to frame buffer consists of the pixels for these primitives. In order to facilitate inquiries about the screen-contents, which are stored in the frame buffer, communication channels are introduced from frame buffer to display processor and from display processor to CPU. This architecture is shown in Figure 2.2.
2.3 Generalized Display Processor 0 Architecture

In [Or91] it is argued that the above architectures are not very suitable for programs that involve many changes of the screen-contents, e.g. animation programs. Indeed, changing the screen-contents requires communication from CPU to memory module or DP (in case of raster-graphics with DP architecture). Depending on the amount of communication, this may severely slow down the program that runs on the CPU. Moreover, the computationally expensive algorithms used in computer animation (e.g. dynamics, constraint solving, or collision detection and response) will hinder the other tasks running on the same CPU. Finally, programs containing animation sequences may be hard to implement on these architectures, because these sequences require regular attention of the CPU (e.g. by using interrupts).

In order to solve these problems, an architecture is proposed where a special processor is used to carry out the animation tasks of a program. This processor is a generalization of an ordinary display processor, since it also supports motions of graphical objects (traditionally, display processors only deal with static graphical objects). Therefore, the name Generalized Display Processor\(^4\) is proposed for this new processor type. Figure 2.3 contains a graphical representation of this architecture.

As can be seen in this figure, an architecture was chosen where a memory module contains a description of the required animation sequence. Since this memory module is a generalization of the traditional DL, it is called a Generalized Display List (GDL). It is used by the CPU to store the description of the animation that should be carried out by the Generalized Display Processor 0. For this purpose, the CPU is linked with the GDL by means of a unidirectional communication channel. The information transferred from CPU to GDL is an animation specification in a formal language: the script-language of the Generalized Display Processor 0. Furthermore, the CPU should be able to update the GDL, while the Generalized Display Processor 0 should react

\(^4\)In this thesis this processor type is called Generalized Display Processor 0.
immediately to these changes in the GDL. Finally, graphics input devices, such as a mouse, are coupled to the Generalized Display Processor 0 for direct manipulation purposes.

In [Ov91] a (software) implementation — called Walt — of such a Generalized Display Processor 0 is presented. Walt supports several motion specifications, e.g. (inverse) kinematics, dynamics [Bar94], track recording [Nie92], and collision detection and response [Kel93b]. A description of the latest version of Walt can be found in [Kel93a].

2.4 Generalized Display Processor 1 Architecture

At first sight, the Generalized Display Processor 0 architecture seems to be suitable for implementation of the workspace of the DenK project1.1, because:

- it supports autonomous behavior of animated objects
- it supports direct modification of these objects;
- it supports direct manipulation of these objects.

However, there are still a few problems:

1. Due to the use of unidirectional communication channels from CPU to GDL and from GDL to Generalized Display Processor 0, inspection of the status of these objects is not possible.

2. The GDL supports the definition of moving objects, each with its own individual motion specification (otherwise, the GDL would obstruct the universality of the Generalized Display Processor 0). The GDL is therefore more complex than an ordinary DL, which contains only objects of a pre-defined type (i.e. line segments). Consequently, updating the GDL necessary in direct modification, may be a very tedious process (resulting, for every frame, in a complicated analysis of the GDL by the Generalized Display Processor 0).

The first problem may be solved by introducing a communication channel from the Generalized Display Processor to the CPU, provided the communication over this channel is asynchronous (i.e. the Generalized Display Processor 0 does not have to wait until the CPU is ready to receive information).

The second problem may be solved by placing the GDL inside the Generalized Display Processor and introducing a direct communication channel from the CPU to the Generalized Display Processor (as in raster-graphics with D'V architecture). Again, the communication over this channel — fragments in the script-language — should be asynchronous (otherwise, the CPU would have to wait until the Generalized Display Processor is ready to receive these fragments). In this architecture the Generalized Display Processor does not have to analyze the GDL for every frame, because the information normally stored in the separate GDL is now contained by the Generalized Display Processor itself. This extended processor is called Generalized Display Processor 1 (Figure 2.4 contains a graphical representation).
Unfortunately, Walt is not based on this Generalized Display Processor 1 architecture, for example Walt does not support:

- direct modification (i.e. Walt processes its GDL only once)
- inspection (i.e. Walt does not support interactive examination of its run-time values)

Furthermore, Walt does not support addition of new functionality (it requires changes to Walt’s script-language, parser, and interpreter in order to add new functions). Consequently, users have to deal with several implementation details of Walt when they want to add special motions.

Upgrading Walt to an animation system that fully supports the Generalized Display Processor 1 architecture (as required for the DenK project) would be very elaborate: its script language, parser, and interpreter have to be changed drastically. Therefore, we have chosen to design a completely new animation system based on the Generalized Display Processor 1 architecture — this system is named GDP (an acronym of Generalized Display Processor)\(^2\) — and an accompanying script-language with the name LOOKS. Of course, the knowledge gained during the development of Walt can be used in the design of the GDP. In this way, Walt serves as a precursor of the GDP; for example, new algorithms for dynamics were implemented and tested in Walt [Bar94] before they were ported to the GDP (as described in Section 6.9).

The issues involved in the design of the script-language Looks and the GDP itself are discussed in Chapters 3 and 4 respectively.

---

\(^2\)To recapitulate: Generalized Display Processor 0 is the name of the processor type as described in [Ov91], with Walt as one of its implementations; Generalized Display Processor 1 and GDP are the names of the processor type and an implementation of this last processor type as described in this thesis.
Chapter 3

Design of Looks

3.1 Introduction

One of the most important aspects of the GDP is the language that is used by an application\(^1\) to control the GDP. Clearly, if it would be very hard — or even impossible — to specify animations that should be carried out by the GDP, the usefulness of the GDP would be very limited. The design of an appropriate script-language is therefore crucial in the development of the GDP. In Section 3.2 the requirements for the script-language are discussed. Section 3.3 introduces two useful concepts that facilitate the fulfillment of these requirements: object-orientation and concurrency. Section 3.4 argues that existing languages are not very suitable to control the GDP. In Section 3.5 Looks is presented: the actual script-language of the GDP. The formal specification of the language Looks can be found in Appendix A.

3.2 Requirements

The first stage in the design of the script-language of the GDP is to investigate the requirements the language should meet. Most of the requirements (numbers 4, 5, 6, and 8 below) for the GDP script-language are based upon Horowitz's guidelines for language design [Hor84]:

1. animation-oriented: the language should facilitate the development of animations.

2. universal: the language should support the universal character of the GDP (i.e. the array of applications of the GDP should be as diverse as possible).

3. concise: the language should support a programming style that results in concise programs.

4. comprehensible: the language should consists of a limited set of simple constructs.

5. reliable: the language should facilitate the development of correct programs.

\(^1\)In the rest of this thesis programs — running on the CPU — that use the GDP are called applications.
6. **extensible**: the language should allow the addition of new functionality to the GDP.

7. **interactive**: the language should support the interactive character of the GDP (especially, direct modification).

8. **well-defined syntax and semantics**: the language should be specified by a mathematical formalism, which acts as an intermediate between the implementor and user of the language.

9. **efficient**: the language should allow for an efficient execution.

Requirement 1 is rather trivial: since the GDP is an animation system, its script-language should facilitate the specification of animations.

Requirement 2 prohibits the use of rather specific primitives. Of course, it could be argued that there is no problem in supplying too many primitives, especially since the language is extensible (requirement 6). Unfortunately, such a language will be quickly unmanageable and does not therefore fulfill requirement 4.

For reasons of efficiency, concise programming should be encouraged (requirement 3). Verbose programs are undesirable to control the GDP because both their transmission (from application to GDP) and their interpretation (in the GDP) take long.

In order to increase the usefulness of the language it should contain a small set of simple language constructs (requirement 4); otherwise, it will deter potential users. Furthermore, a compact and comprehensible language facilitates the design of application generators (i.e. programs that ease the development of applications by automatically generating code in the GDP’s script-language).

The usefulness of the language is also improved if it facilitates the design of correct programs (requirement 5). For example, a language that is strongly typed guarantees that programs will not produce errors due to type mismatches; programs written in a weakly typed languages may produce such run-time errors (since these errors may be dependent on the execution-order of the program, they may be very hard to find).

Most likely the functionality of the GDP will be extended during the lifetime of the GDP (i.e. the GDP will be developed incrementally). The script-language should be able to support these extensions (requirement 6) — preferably with no changes to the language.

Since the application and the GDP communicate interactively, the application may send either an entire script or script-fragments. Immediately after receiving such a fragment the GDP will start its execution. The script-language should therefore support this kind of interactive communication (requirement 7). For example, this requirement makes it impossible to use the concept of a "main routine", found in many — compiled — languages (like C and Pascal).

The language should be specified formally (requirement 8), because a specification in a natural language tends to be vague (e.g. it may contain ambiguities). Especially since the designer, the implementor, and the user of the language will be different persons, a sound interface — by means of a formal specification — is necessary.

The language should also allow for an efficient execution, since interactive animations are typically time-critical.
3.3 Useful Concepts

To fulfill the above requirements two concepts are useful: object-orientation and concurrency, which are introduced in Sections 3.3.1 and 3.3.2, respectively. Some approaches that combine these concepts are presented in Section 3.3.3. Finally, Section 3.3.4 discusses how these concepts should be incorporated in the script-language of the GDP.

3.3.1 Object-Orientation

In order to facilitate the development of simulation programs, Dahl and Nygaard designed the simulation language Simula [DMN70]. This language first featured the concept of a class, a combination of variables and operations. Classes can be considered as implementations of abstract data types (although Simula does not contain language constructs for hiding the internal structure of classes), and they define a set of possible values. Furthermore, a new class may be defined as an extension of an existing class (in object-oriented terminology this concept is known as (single) inheritance), and the new class may redefine the operations it received from the existing class. In Simula, objects are used as the entities in simulations. For example, in a simulation concerning queues, a class may describe the data and behavior (operations) of customers, whereas objects of this class act as individual customers in the simulation. Objects are referred to by reference variables (pointers), and they may exist even if the scope of the reference variable that was associated with the object during instantiation vanishes (indeed: other reference variables could refer to this object). Reference variables are typed, and may refer to objects of that type or to objects of types that are extensions of that type. In the latter case, calling an operation using the reference variable invokes the operation as it is defined in the class of the extended object (which may be a redefinition of the original operation; in object-oriented terminology this concept is known as dynamic binding). The decoupling of the lifetime of an object and the scope of reference variables is supported in Simula, because simulations often deal with entities having independent lifetimes (e.g. one customer will enter the queue, while another customer decides to leave). Since it would be very elaborate for the user of Simula to keep track of the objects that are not referred to anymore (by reference variables) in order to dispose these objects, Simula performs this task automatically by means of a garbage collection mechanism.

The object-oriented concepts introduced in Simula (i.e. class, inheritance, object, and dynamic binding), have been used in many languages afterwards, for example:

- Smalltalk [GR89]: Smalltalk is an untyped language and features an interactive programming environment, containing a debugger, a class browser (a tool to inspect classes), and an extensive library of pre-defined classes.

- Eiffel [Mey92]: Eiffel is a typed language and features, apart from single inheritance, multiple inheritance (a new class may be defined as an extension of several existing classes) and assertions (pre- and post-conditions that are checked at run-time, which facilitate the development of correct programs).

- C++ [Str91]: C++ is a typed, object-oriented extension of C (although it contains constructs that may violate strong typing). In contrast to the previous
languages. C++ does not contain a garbage collector: the programmer is responsible for disposing the objects.

Computer animation is one of the fields of computing science leading itself perfectly for object-oriented programming. Indeed, an intuitive match exists between the entities in an animation and objects in an object-oriented system. This property has been noticed by various researchers and as a result several animation systems exist that are based on the object-oriented paradigm, e.g. Clockworks [GB90] and Solar [CWC88]. As a result, the design of an animation is facilitated because the appearance and the behavior of the characters in the animation can be specified as separate objects by the animator - this approach mimics the traditional film making process where the script contains a description of the characters in the film.

The concept of object-orientation seems therefore very useful for the script-language of the GDP. Furthermore, some aspects of object-orientation help to fulfill the requirements of Section 3.2: (multiple) inheritance and dynamic binding may result in a concise programming style (requirement 3), and a library of pre-defined classes facilitates the addition of new functionality to the GDP without changes to the script-language (requirement 6).

### 3.3.2 Concurrency

Computers are traditionally based on the (sequential) Von Neumann machine. Languages that were developed to program these computers initially reflected this sequential execution mechanism (e.g. Fortran [ANS66], Pascal [Wir71], and C [KR78]). Already in 1965 Dijkstra presented a solution to an (academic) problem based on multiple cooperating sequential programs [Dij65], or processes as they are currently called. Soon it was realized that many practical problems (especially in the field of operating systems) were easier to solve by using processes, because sequential solutions would require very detailed information concerning the execution order of program fragments. The development of programs that are based on processes is known as concurrent programming. In the design of such programs two aspects are important:

- **communication**: information has to be passed from one process to another
- **synchronization**: cooperation of processes has to be regulated

Several mechanisms were developed to deal with these aspects of concurrent programming [And91], for example:

1. **semaphore** [Dij68]: a semaphore is an object\(^2\) that has two methods, \(P()\) and \(V()\), and one (internal) non-negative counter. If a process invokes method \(P()\) of a semaphore and the counter of this semaphore is positive, the counter is decremented by one and the process continues; however, if the counter is zero, the process will be delayed until the counter becomes positive (then the counter is decremented and the process continues). If a process invokes method \(V()\) of a semaphore the counter of this semaphore is incremented by one.

---

\(^2\)Object-oriented terminology is used to explain the concept of a semaphore. Naturally, the original explanation did use other terminology.
2. conditional critical region [Hoa72, Bri72]: this mechanism consists of two parts: resources and regions. A resource is a named collection of shared variables to which mutually exclusive access is required. A region is a statement that consists of a resource-name and a list of statements (that may use the variables of the named resource). Regions that have the same resource-name are executed mutually exclusive.

3. monitor [Hoa74]: a monitor is an abstract data-structure that consists of variables and operations (i.e. a monitor can be considered as an object). Due to data-abstraction, processes may only execute the operations of a monitor. Since a monitor may be shared between multiple processes, communication between these processes can be provided by means of the variables of the monitor. Mutual exclusive access of these variables is ensured by the requirement that operations of a monitor cannot be executed simultaneously. Furthermore, synchronization is achieved by means of semaphores.

4. asynchronous message passing [Bri70]: this mechanism is not based on shared variables between processes (in contrast to the previous mechanisms), but on processes that communicate via channels (i.e. unbounded queues of messages) by means of the statements send and receive. Sending a message (via a channel) to another process does not block the sender. Executing a receive statement, however, delays the process until a message is available on the channel.

5. synchronous message passing [Hoa78]: this mechanism, too, is based on processes that communicate via channels. The send statement has a different semantics, however: the sender process blocks until a receiver process is ready to accept the message (consequently, a channel does not buffer messages). The receiver process still blocks until a message is available on the channel.

6. (remote) procedure call [Bri78]: this mechanism is based on a data-structure (a module) that contains (internal) variables and provides operations on these variables. If a process invokes an operation of such a module, a new process is created that performs the operation. Furthermore, a two-way communication channel is introduced between the process that invokes the operation and the newly created process. Finally, the caller process is delayed until the operation is completed and the results (if any) of the operation are returned.

7. rendezvous [DoD83]: this mechanism is closely related to remote procedure calls. However, a rendezvous does not create a new process to perform an operation, but it uses an existing process that has indicated that it is willing to perform the operation by means of an input statement. Thus it synchronizes two processes, and is in this respect related to synchronous message passing.

These concurrency mechanisms may be implemented on a single-processor system (multi-programming; processes are executed one at a time in an interleaved manner) or on a multi-processor system (parallel programming). Since the number of processes will generally exceed the number of processors (especially on a single-processor system), process-switching is necessary: the process that currently uses a processor is delayed, while another process starts to use the processor. Two strategies of process-switching can be distinguished:
1. **implicit**: process switching is outside the control of the processes themselves.

2. **explicit**: process switching is performed at the request of the active process.

In case of multi-programming these strategies are named *pseudo-parallelism* and *quasi-parallelism*, respectively [Sch89].

Since animations may involve several simultaneous movements, the GDP’s script-language should facilitate the specification of such movements. Otherwise, these animations could be specified only in a complex and verbose way, and the language would not fulfill requirements 3 and 4. Consequently, the GDP’s script-language should support at least one of the above concurrency mechanisms.

### 3.3.3 Combined Object-Orientation and Concurrency

Object-orientation and concurrency are two orthogonal concepts: objects may perform their operations concurrently. Concurrency may be introduced in object-oriented systems in two distinct ways:

- **inter-object concurrency**: objects can perform operations concurrently, but each object must perform at most one operation at a time, i.e. it can host one process at a time.

- **intra-object concurrency**: objects can perform operations concurrently, and each object can perform multiple operations simultaneously, i.e. it can host several processes at a time.

Several mechanisms to merge the concepts of object-orientation and concurrency have been proposed, for example:

1. **Actors** [AH87] which is based on asynchronous message passing (consequently, it supports inter-object concurrency).

2. **Concurrent Smalltalk** [Yak90] which extends standard Smalltalk with constructs for asynchronous message passing, synchronization, and atomic objects (inter-object concurrency).

3. **POOL-T** [Ame87] which is based on synchronous message passing and objects that have autonomous behavior (i.e. each object starts its own process; inter-object concurrency).

4. **Scheduling Predicates** [MWBD92] which presents a formalism to restrict inter-object concurrency by means of predicates that specify for each operation when it may be performed.

5. **Smalltalk-80** [GR89] which is based on fork-statements (to create a new process), remote procedure calls (intra-object concurrency), and semaphores.
3.3.4 Combined Object-Orientation and Concurrency in the GDP’s script-language

As mentioned in Sections 3.3.1 and 3.3.2, an animation can be considered as a collection of cooperating entities which can be mapped straightforwardly to a collection of concurrently operating objects.

In order to successfully incorporate these concepts in the script-language of the GDP, the following design issues are important:

1. The concurrency mechanism should support intra-object concurrency, because entities in an animation may perform several operations concurrently (e.g. a shape may be rotating and translating concurrently).

2. Asynchronous (remote) procedure calls seem a natural way to introduce new processes: when an object (the sender) sends a message to another object (the receiver), a new process is started in which the receiver performs the requested operation, while in the original process the sender continues its own operation (note that this introduces intra-object concurrency, because multiple asynchronous procedure calls can be handled by an object simultaneously).

3. Process switching should be explicit, because some operations (e.g. rotations) require more execution time than other operations (e.g. translations). Using implicit process switching may result therefore in a situation where simple processes will get relatively too much execution time compared to more complicated processes. This problem will be solved when the processes can indicate themselves when process switching should occur.

4. Process switching should be fair; indeed, two (explicit) process switching policies can be distinguished:

   - locally decided: the current process (i.e. the process that explicitly demands a process switch) may be resumed at any relevant event in the future (although every process should be resumed eventually, of course)
   - globally decided: the current process may not be resumed until a certain global event occurs.

   The second option, i.e. global process switching, matches perfectly with computer animation, because in animation an intuitive global synchronization event exists: the generation of a new frame. Furthermore, global process switching prevents (by definition) the possibility that one process deprives another one.

5. The concurrency mechanism should be based on multi-programming (i.e. one-processor system), because a multi-processor system would require a dedicated hardware architecture. However, if a single-processor version of the GDP is successful, a version on a multi-processor may be considered.

Since the script-language of the GDP should support explicit process switching for multi-programming, it supports quasi-parallelism [Sch89].
3.4 Existing Languages

In this section we examine existing languages in order to determine whether they are suitable as script-language of the GDP. For this purpose they should have the following characteristics:

- meet the requirements of Section 3.2
- based on the concepts of Section 3.3

The languages that are examined can be classified in two categories:

1. *animation script-languages*
   
   Among these languages are: Clockworks [GB90], ASAS [Rey82], Solar [CWC88], Cinemira [MTTF85], and Pinocchio [MP90].

   The examined script-languages lack the following three properties:
   - suitability for direct modification
   - extensibility with new functionality (e.g. new rendering primitives using graphical hardware) without significant changes to the interpreter/compiler
   - availability of a concurrency mechanism that is based on quasi-parallelism, in combination with asynchronous procedure calls and global process switching

   Furthermore, they also have other — more individual — drawbacks:
   - not object-oriented, but based on Lisp (ASAS) or Pascal (Cinemira)
   - not universal, but dedicated to the subset of anthropomorphic animations (Pinocchio)

2. *object-oriented programming languages*

   Among these languages are: C++, Eiffel, and Smalltalk.

   The examined object-oriented languages have the following drawbacks:
   - None of them contains a concurrency mechanism that is based on quasi-parallelism, with asynchronous procedure calls and global process switching (C++ and Eiffel do not support a concurrency mechanism at all).
   - C++ and Eiffel are not suitable for the interactive character of the GDP because they both contain a "main-routine" that cannot be changed interactively.
   - C++ is not an elegant, simple, or compact programming language (some of its flaws are discussed in [Sak92]).
   - Smalltalk is not strongly typed, resulting in programs that need run-time checking and that may be hard to debug (i.e. Smalltalk violates requirements 9 and 5).

As shown above, using an existing language — either an animation script-language or an object-oriented programming language — requires (radical) changes to its interpreter/compiler. It is therefore unwise, also with respect to maintainability, to use
such a language as the script-language of the GDP. On the other hand, the other option, i.e. the design of a new language, may require more time. We have chosen for the second option, because it will more likely result in a language that is best suited for the GDP. The script-language that we have designed for the GDP is named LOOKS\(^3\) (acronym of Language for Object-Oriented Kinematic Specifications).

3.5 Looks

In this section, the language concepts of Looks will be presented in some detail to provide an operational introduction of Looks. This section is based on the assumption that the reader has some knowledge of object-orientation\(^4\). Consequently, rather than giving a generic discussion of the basic elements of object-orientation (e.g. inheritance, dynamic binding, etc.) these are presented as Looks-ingredients. So, of all object-oriented notions, only the Looks versions are presented here. This amounts to a bottom-up introduction of Looks, and it is not immediately obvious that this set of Looks concepts is a valid realisation of the requirements and the considerations of Sections 3.2-3.4. Therefore, after the introduction of Looks, Section 3.6 confronts in an \textit{a posteriori} fashion the language concepts of Section 3.5 with these requirements and considerations.

3.5.1 Language Aspects

Looks is the language that is used to send information from an application to the GDP. This information can be split in (atomic) actions, consequently a Looks program can be considered as a list of such actions (grammar rule 1). However, due to the interactive character of the GDP these actions do not have to be sent together, but there may be a delay between them.

1. \texttt{<start> ::= \{ <action> \ | \} \}}\(^5\)

Looks supports the following atomic actions:

- forward class declaration (grammar rule 2a, see page 28)
- class-definition (grammar rule 2b, see page 28)
- statement (grammar rule 2c, see page 47)
- reference-location declaration (grammar rule 2d, see page 27)

These atomic actions are explained in great detail in this section, but since in the field of object-oriented programming currently no real consensus exists on the terminology, all used terms are defined first.

\textbf{Definition 3.1 : Attribute}

An attribute is a variable that may contain data.

\(^3\)Looks is a registered trademark in the Benelux (registration number 511969).
\(^4\)The book [Mey88] offers a general introduction to object-orientation
\(^5\)\{ x \} denotes zero or more occurrences of x.
Definition 3.2 : Method
A method is a generic term for an operation.

Definition 3.3 : Procedure
A procedure is a method that does not return data and that may change the state of an object, i.e. the values of the object’s attributes.

Definition 3.4 : Function
A function is a method that returns data.

It is a good programming practice that a function should not change the state of an object.

Definition 3.5 : Object
An object is an entity that consists of a combination of zero or more attributes (data) and zero or more methods (behavior).

Definition 3.6 : Feature
A feature of an object is either an attribute or a method.

Definition 3.7 : Private Feature
A private feature of an object is a feature that may be used only by the methods of that object.

Definition 3.8 : Public Feature
A public feature of an object is a feature that may be used by the methods of that object, by methods of other objects, or by the application in an action-statement.\(^6\)

Note that the definition of public feature allows that both attributes and methods of an object may be public. In some other object-oriented languages, e.g. Smalltalk, attributes may not be public. In these languages inspecting and updating an attribute has to be done by means of methods.

Definition 3.9 : Message
A message, which is sent to an object, is a request to execute one of the methods of that object.

\(^6\)An action-statement is an action in the form of a statement — grammar rule 2c.
3.5. Looks

A message to an object may be sent by other objects or it may be sent by an application (by means of an action-statement) The application is unaware that the execution of the requested method may involve messages to other objects as well.

To ease the definition of a collection of isomorphic objects, the concept of class is introduced.

Definition 3.10 : Class

A class is a template that specifies the public and private features of isomorphic objects.

Loos is strongly typed: types\(^7\) have to be supplied for every attribute, method argument, and method result. This facilitates the development of correct programs: for every feature that is used in a Loos program (e.g. in a statement), the GDP can check at parse-time if the requested feature is provided by the concerning object. Furthermore, strong typing improves the speed of the GDP, since no type checking has to be performed at run-time.

In order to keep Loos as simple as possible, objects should always be created using a class (even if the class will be used to create only one object). The objects that are created this way are called the instances of the class. Furthermore, Loos is based solely on object-orientation, including basic numerical values, logical values, etc. This facilitates a uniform syntax definition of Loos: no explicit rules are necessary for these ‘basic types’ (indeed, the basic types are given as predefined basic classes).

To understand the semantics of Loos it is important to discriminate between an object, an object-reference (i.e. a pointer to an object), and a reference-location (i.e. a variable that may contain an object-reference). Note that multiple object-references may refer to the same object (this property is known as aliasing), but that each reference-location can contain only one object-reference. Due to the strong typing of Loos, reference-locations are typed: they can contain only object-references to objects of certain types. Grammar rule 2d contains the syntax for the declaration of new reference-locations.

\[
\begin{align*}
2d. \text{<action>} & \quad ::= \text{"OBJECT" } \text{<identifier_list>} \ \text{"\ }\text{<type>}
\end{align*}
\]

28. \text{<identifier_list>}
\[
\begin{align*}
& \quad ::= \text{<identifier>} \ {\text{,}} \ \text{<identifier>}
\end{align*}
\]

Using the above terminology, the attributes of an object can be considered as reference-locations, resulting in a chain of objects that refer to each other. Moreover, the basic values are provided as predefined objects and accompanying object-references (in the remainder of this thesis they are named basic object and basic object-reference, respectively). For example, basic object ONE is an instance of class integer, and is referred to by the basic object-reference with name 1. Figure 3.11 shows a graphical

\[^{7}\text{In this thesis the terms ‘class’ and ‘type’ denote the same concept and are used interchangeably. The term ‘class’ stems from the field of object-orientation, whereas the term ‘type’ has a long tradition in computing science. The term ‘type’ is used in this thesis in order to be able to use traditional names, e.g. ‘strong typing’ instead of ‘strong classing’.}
\[^{8}\text{The keywords of Loos are case-independent.}
\[^{9}\text{The syntax of identifiers is included in Appendix B.}]
\]
representation\(^{10}\) of these basic entities and it also shows the effect of the fragment:

```plaintext
object i,j : integer, i:=1;
```

Note that by default reference-locations are initialized by the null-reference.

![Diagram of reference-locations, object-references, and objects](image)

**Figure 3.11: Reference-locations, object-references, and objects**

If subsequently fragment `i:=i.plus(1);` is executed, another object-reference will be stored in reference-location `i` (the new configuration is shown in Figure 3.12).

![Diagram of reference-locations, object-references, and objects (cont'd)](image)

**Figure 3.12: Reference-locations, object-references, and objects (cont’d)**

As can be seen in grammar rule 2b, a class-definition consists of a heading, inheritance specification (explained on page 33), declaration of features (page 31), and the implementation of methods (page 32). Grammar rule 2a is used to declare a new class heading, without giving the other parts of the class definition. This forward mechanism allows the specification of mutually dependent class-definations.

```
2a. <action> ::= "FORWARD" <heading>
2b. <action> ::= "CLASS" <heading> ["EXTERNAL"] ["<identifier>"] ["<class.kind>"] <inherit.decl> ["<declarations>"] ["<implementation>"

"ENDCLASS"

3. <heading> ::= <identifier> ["<identifier.list>"]
4a. <class.kind> ::= "BASIC"
4b. <class.kind> ::= "OBJECT" ["<identifier>"
```

Furthermore, a class-definition does not have to be given integrally, but it may be split in several parts. The following states of a class-definition can be distinguished:

\(^{10}\)The following graphical conventions are used:

- **boxes (thin edges)**: reference-locations
- **arrows**: object-references
- **single edge**: null
- **circles**: objects

\(^{11}\)[x] denotes at most one occurrence of x.
3.5. **Looks**

1. *declared:* the heading of the class is given by means of a forward declaration,

2. *specified:* the inheritance specification and declaration of features of the class are given,

3. *implemented:* the methods of the class are implemented.

These states should be encountered in the above order, e.g. it is not possible to implement a class before it is specified. Example 3.13 shows how split class-definitions are used in the definition of mutually dependent class-definitions. The declaration and implementation parts in this example will be explained later in this section.

**Example 3.13 : Split class-definitions**

```plaintext
forward B;
// class B is declared

class A
public
  b : B; // class B is declared and may thus be used
  f ( ) : void;
endclass;
// class A is specified

class B
public
  a : A; // class A is specified and may thus be used
  g( ) : void;
implementation
  g( ) : void;
begin
  a.f( ) // class A is specified, thus procedure f may be used
  end;
endclass;
// class B is implemented

class A
  implementation
  f( ) : void;
  begin
  b.g( ) // class B is implemented, thus procedure g may be used
  end;
endclass;
// class A is implemented
```

The heading starts with an `<identifier>` which gives the name of the new class. The optional `<identifier_list>` part in the heading is used to specify the formal generic types of the class (genericity will be discussed below).

The *external* option after the heading in a class-definition is used to specify the name of a library that should be linked during run-time (this concept is explained in Section 3.5.2). Grammar rule 4b is also used for run-time library linking: it specifies the name of a compiled function in the newly linked library that returns a compiled object (again this is elaborated in Section 3.5.2).

---

12 Looks supports three kinds of comments (the first is restricted to one line):

1. `// comment`
2. `{ comment }
3. `/* comment */`
Keyword `basic` in a class-definition is used to indicate the basic classes of Looks: `void`, `boolean`, `integer`, `real`, `char`, and `string`. These classes are characterized by the fact that all possible instances are present initially, and therefore no new instances can be created later on. This keyword may only be used during the initialization of the GDP (classes that are present initially are also specified by means of a Looks script); trying to use keyword `basic` after initialization will result in an error. Example 3.14 shows how this keyword is used in the initialization-script of the GDP to define basic class `integer`.

**Example 3.14 : Keyword basic**

```looks
class integer
  basic
  public
    plus (x : integer) : integer;
    times (x : integer) : integer;
    ...
  endclass;
```

To prevent the definition of several classes with similar structures, a class may be defined *generically*. For instance, a generic list may be defined instead of defining a list of integers, shapes, cars, etc. A generic list could provide methods such as empty, add, concatenate, etc.

**Definition 3.15 : Generic Class**

A generic class is a template for classes that have a similar structure, and only differ in the type of their features.

To define a generic class, the formal generic types are listed after the class name. Formal generic types are identifiers which may be used as a type in the class-definition. When using a generic class, existing types (which will be called *actual generic types*) have to be supplied for each formal generic type. Since a generic class acts as a template for classes, each generic class with actual generic types is a correct class (or type). For example, generic class `A[T]` is a template for the types `A[S]`, for every type `S`.

Grammar rule 29 specifies the syntax of types. The optional part in this rule denotes the actual generic types that should be provided for each generic class.

29. `<type>` ::= `<identifier> | "[" `<type>` ("," `<type>`)?"]`  

Types are used at two distinct places in a Looks program:

- **action level**

  At this level types are used to specify the kind of object-references that may be stored in a reference-location (grammar rule 2d). The types that may be used at this level consist of an existing class-name and, if required, actual generic types (that are also constructed in this way). Of course the number of actual generic types should equal the number of formal generic types of a class.
3.5. Looks

- class-definition level
In a class-definition, the types of ancestors (related to inheritance, see page 33), attributes, method-results, parameters, and local variables have to be specified (due to the strong typing of Looks). The following types may be used:

1. formal generic type
In a generic class, the formal generic types may be used everywhere where a type is required. When the formal generic types are replaced by actual generic types, all occurrences of these formal generic types in the class-definition are updated automatically.

2. class-name (plus actual generic types, if necessary)
If the class-name denotes a generic class, the actual generic types that have to be provided should be types that are allowed in the class-definition (i.e. one of the types in this list: 1 or 2). Note that the class-name may be the name of the class that is currently being defined. Hence, recursive classes may be defined. These classes can be very useful for modeling recursive structures (e.g. chains, lists, and trees).

Example 3.16 shows how a formal generic type may be used in a class-definition and how actual generic types should be provided at action level.

Example 3.16 : Generic types

```)
class A[T]
  public
  w : T;  // formal generic type T used as type of
       // attribute w (class-definition; case 1)
  x : A[T];  // formal generic type T used as actual
             // generic type (class-definition; case 2)
  f (y:T) : void;  // formal generic type T used as type of
                   // parameter y
  g () : T;  // formal generic type T used as result-type
implementation
  f (y:T) : void;
  var z:T;  // formal generic type T used as type of
             // local variable z
  begin
  ...
  end;
endclass;

object a1 : A[integer];  // type integer used as actual generic type
                         // (action level)
object ab : A[boolean];  // type boolean used as actual generic type
object aan : A[integer];  // types integer and A[integer] used as
                          // actual generic types
```

Grammar rules 11, 12, and 13 specify the syntax for the declaration of the public and private features of a class. As rule 11 indicates: the order of public and private features is not important.

11a. `<declaration>` ::= `<public>` | `<private>` |

11b. `<declaration>` ::= `<private>` | `<public>` |

12. `<public>` ::= "PUBLIC" `<attrs.or.methodhead>` | "PRIVATE" `<attrs.or.methodhead`

13. `<private>` ::= "PRIVATE" `<attrs.or.methodhead>` |

Grammar rule 15a specifies the syntax for the declaration of (public or private) attributes. The attribute-names of a class should be unique. As mentioned previously, due to the strong typing of Looks the type of the attributes has to be specified.

15a. \texttt{<attr-or-methodhead>} := \texttt{<identifier>} \texttt{'} \texttt{<type>}

Grammar rules 15b, 16, and 17 specify the syntax for the declaration of (public or private) methods. A method-name may not be used as an attribute-name (and vice versa), because feature-names are used in an inheritance definition to specify features that should become public (inheritance is explained below). If a name could be used both as method-name and attribute-name, the specification of public inherited features would require special attention (especially when only one of the features should become public). However, method-names need not be unique: 	extit{overloading} of method-names is allowed (when an overloaded method is called, it depends on the types of the actual parameters which method is chosen, see page 45).

15b. \texttt{<attr-or-methodhead>} := \texttt{<specifier>} \texttt{'} \texttt{[} \texttt{<form-param-list>} \texttt{]} \texttt{'} \texttt{<type>}

16. \texttt{<form-param-list>} := \texttt{[} \texttt{<param-kind>} \texttt{]} \texttt{<identifier-list>} \texttt{'} \texttt{<type>}

17. \texttt{<param-kind>} := \texttt{VAL'}

17b. \texttt{<param-kind>} := \texttt{VAR'}

As grammar rule 15b indicates: a method-heading consists of a name, formal parameters, and a result-type (if this type is void the method is a procedure, otherwise it is a function). Looks supports two sorts of parameters (grammar rule 11): value-parameters (keyword: val) and var-parameters (keyword: var), they pass object-references and reference-locations as parameters, respectively. By default, a parameter is a value-parameter.

Grammar rules 14, 18, 19, and 20 specify the syntax for the implementation part of a class. Note that it is not necessary to implement methods: the keyword \texttt{implementation} may be used in isolation (thereby promoting the class to state ‘implemented’). This option is useful to complete a class-definition, even when no methods of this class will be implemented (this is convenient in case of inheritance, because only implemented classes may serve as ancestors).

14. \texttt{<implementation>} := \texttt{IMPLEMETATION} \texttt{[} \texttt{<method-def>} \texttt{]} \texttt{<method-def>} \texttt{]}

18. \texttt{<method-def>} := \texttt{<prefix>} \texttt{<identifier>}

19a. \texttt{<method-def-rest>} := \texttt{<loc-variables>} \texttt{<block-statement>}

19b. \texttt{<method-def-rest>} := \texttt{<loc-variables>} \texttt{<block-statement>}

20. \texttt{<loc-variables>} := \texttt{VAR} \texttt{<identifier-list>} \texttt{'} \texttt{<type>}

The implementation of a method (grammar: rule 18) starts with the heading of the method (this heading should match one of the declared public or private method-headings; however, it may also be the heading of an inherited method - hence the possible prefix - this will be explained later on in this section). Furthermore, if the method-name is not overloaded, the parameters and result-type of the heading are optional.

Grammar rule 19b is used for run-time library linking: it specifies the name of compiled code that should be executed when the Looks method is invoked (this concept is explained in Section 3.5.2). Grammar rule 19a specifies how a method-body may be implemented in Looks. First, a method implementation may contain
local variables (grammar rule 20), which should be unique and may not occur as parameters of the method. Second, the body of the method is implemented as a block-statement (the section that presents the statements of Looks starts on page 47). Finally, since the declaration and implementation of methods are always split, mutually recursive methods are allowed.

On page 30 genericity was introduced: a mechanism to facilitate the definition of isomorphic classes. However, genericity cannot be used to ease the definition of classes that are extensions of each other. For this purpose Looks supports inheritance.

Definition 3.17 : Inheritance

Inheritance is a mechanism to define a new class (descendant) as an extension of an existing class (direct ancestor).

This extension may be accomplished either by adding new features or by redefining inherited methods.

The inheritance relation between a descendant and a direct ancestor is indicated as follows: descendant → direct ancestor. Indirect ancestors of a descendant can be found by means of the transitive closure of the inheritance relation (∈️). Direct and indirect ancestors are commonly known as the ancestors of a descendant.

The inheritance relations between classes can be represented by means of an inheritance graph (definition 3.18).

Definition 3.18 : Inheritance graph

An inheritance graph of a class, say A, is a directed graph with:

- vertices: class A and its ancestors
- edges: inheritance relation between these classes

In the inheritance graph of a class A, the path from the root (i.e. class A) to an ancestor is called an inheritance-path.

Three sorts of inheritance can be distinguished (in order of sophistication): single inheritance, multiple inheritance, and repeated inheritance.

Definition 3.19 : Single Inheritance

Single inheritance requires that the inheritance graph of a new class is a chain.

Definition 3.20 : Multiple Inheritance

Multiple inheritance requires that the inheritance graph of a new class is a tree.

Note that in case of multiple inheritance every ancestor is reachable by exactly one inheritance-path.
**Definition 3.21: Repeated Inheritance**

Multiple inheritance requires that the inheritance graph of a new class is acyclic.

Note that in case of repeated inheritance an ancestor may be reachable by several inheritance-paths.

Figure 3.22 contains a graphical representation of the three sorts of inheritance: single, repeated, and multiple.

![Graphical representation of inheritance](image)

Figure 3.22: Graphical representation of inheritance in Looks

However, inheritance is more than just an abbreviation mechanism: classes that have an inheritance relation are related (*conforming*) types.

**Definition 3.23: Conformance**

Type B conforms to type A, if and only if:

- Class A is equal to class B, or
- Class A is an ancestor of class B

In terms of an inheritance-graph the definition of conformance can be restated as follows: type B conforms to type A, if — and only if — A is a node of B's inheritance-graph.

If type B conforms to type A all objects of type B will have at least the same features as objects of type A. Consequently, whenever objects of type A are required in a program, objects of type B may be used. Note that this does not violate the strong
3.5. Looks

typing of Looks (indeed, strong typing requires that it is possible to check at parse-time if a feature is provided by an object). Type conformance is the key to the powerful mechanism of dynamic binding (page 53).

Looks supports the most powerful form of inheritance: repeated inheritance. Furthermore, the inheritance mechanism of Looks is based on the open+closed principle [Mey88]:

- **open**: all (public and private) features of an ancestor are inherited — and may be used — by the descendant

- **closed**: only public features of an object may be used by other objects

By default, all inherited features of an ancestor will become private in the descendant. This may be overruled by explicitly stating that an inherited feature should be public. Consequently, all public features of the descendant are listed in its class-definition. However, explicitly naming all public inherited features may hinder the conciseness of a Looks program (requirement 3). Consequently, there is a conflict between clearness (i.e. explicitly naming public inherited features) and conciseness. Since the estimated overhead of these explicit public inherited feature-names is small compared to the rest of a class-definition, we have chosen to include them in order to obtain 'self-documenting' classes. Grammar rule 5 specifies the syntax of inheritance. The inherited features that should become public in the descendant are listed between parentheses after the type of the ancestor (grammar rules 5 and 6).

5. `<inherit_def>` := [ `INHERIT` <type> `{ `| `[ <feature> `] `| `[ `[ <feature> `] `] `| `] `] `| `[ `[ `<feature> `] `] `| `[ `<feature> `] `]`

6. `<features>` := `<feature>` `{`; `<feature>` `}`

Syntactically there is no difference between multiple and repeated inheritance (the first requires that every ancestor is reachable by exactly one inheritance-path, whereas the latter has no such requirement). Looks requires that ancestors are implemented: thus the situation that a class inherits from an ancestor before the methods of the ancestor are implemented is not possible. In an interactive environment, as the GDD, this would cause severe problems when objects of the descendant were used before the implementation of the ancestor (namely, the inherited methods would not have been implemented).

Example 3.24 shows a simple case of multiple inheritance: class C inherits all features of class A and B. Furthermore, public feature k of class A stays public in C, public feature l becomes private, private feature m becomes public, and private feature n stays private. The features of class B are treated in a similar way. Finally, class C contains five public features (i.e. k, m, p, r, x) and five private features (i.e. l, n, q, s, y). Note, that it is allowed that a descendant decides that an inherited private feature (even an attribute) should become public. This is a result of the open-closed principle: the descendant inherits all features and it may use these features without restrictions.
Example 3.24: Multiple inheritance, public inherited features

```java
class A {
    public
        h, i : integer;
    private
        m, n : integer;
    implementation
    endclass;

class B {
    public
        p, q : integer;
    private
        r, s : integer;
    implementation
    endclass;

class C
    inherit A(3,4), B(p,r)
    // C inherits all features of class A and B, where:
    // A’s features k and m become public, l and n become private
    // B’s features p and r become public, q and s become private
    public
        x : integer;
    private
        y : integer;
    implementation
    endclass;
}
```

In Example 3.24 the features that are named in the inherited feature-lists are all attribute-names. For method-names, however, things can become more complicated since these names may be overloaded. Only specifying the method-name indicates that all methods with this name become public. If this is not intended, argument-types may be specified to indicate the methods that should become public. Grammar rules 7a, 8, 9, and 10 give the full syntax for features that should become public (the options `<prefix>` and renaming — [‘>’ `<identifier>’] — are explained later):

```typescript
7a. `<feature>` ::= `<prefix>` `<identifier>` `<argumenttypes>` [‘>’ `<identifier>’]
8. `<argumenttypes>` ::= [‘{’ `<type,’type>’ }’]
9. `<type_list>` ::= [‘{’ `<type,’type>’ }’]
10. `<prefix>` ::= [‘<type,’type>’]
```

Example 3.25 shows how types may be used to select an overloaded method. In this example class B inherits 4 methods of class A: 3 methods stay public, but the method with name f that requires a boolean argument becomes private.

Furthermore, a descendant may provide new implementations for inherited methods (i.e. redefinition of inherited methods), for example when these methods should use features that are present in the descendant but not in the ancestor. This is demonstrated also in Example 3.25 where the redefined body of method g contains attribute i that is introduced in class B, and was not present in class A was the method was first defined.

Example 3.25: Redefinition of inherited methods

```java
class A {
    public
        f (x : integer) : void;
        g (x : boolean) : void;
        g (x : integer) : void;
        g (x : boolean) : void;
```
3.5. Looks

implementation
g (x : integer) : void ... // body.g.A
;
endclass;

class B
inherit A(f(integer),g)
// f(integer), g(integer), and g(boolean) stay public
// f(boolean) becomes private
public
  i : integer;
implementation
g (x : integer) : void
begin
  // body.g.B
  i := x
  // new body of method g now uses attribute i of class B
end;
;
endclass;

As mentioned previously, attribute-names should be unique in a class. Therefore a problem occurs when two inherited attributes have the same name and should both be public in the descendant. This problem may be solved if at least one of these attributes is renamed. Example 3.26 demonstrates the renaming mechanism of Looks: classes C and D inherit all features of classes A and B (that both contain an attribute with name x); class C causes no problem since only A's attribute becomes public, in class D, however, both attributes should be public, therefore renaming is necessary in order to prevent a name conflict (B's attribute x is known in class D as y).

Example 3.26: Renaming of inherited features

Suppose classes A and B are defined previously, and that these classes contain a public attribute x.

class C
inherit A(x),B
  // no problem: B's x becomes private
  ;
endclass;

class D
inherit A(x),B(x,y)
  // no problem: B's x is renamed to y
  ;
endclass;

The renaming mechanism of Looks may also be used to change the names of public inherited methods (although methods will not cause naming conflicts: if two inherited methods have the same name, and they should both become public in the descendant, they will form an overloaded public method in the descendant). Note that renaming makes a feature public in a descendant.

Since private inherited features may not be used by other objects (due to the open-closed principle), there is no need to rename them explicitly as long as a unique discrimination mechanism exists. For this purpose Looks supports prefixes: in a descendant the private inherited features of an ancestor should always be prefixed with
the type of the ancestor\(^\text{13}\) (although prefixes may also be used for public inherited features). Although the concepts of prefixing and renaming are related, there is a difference: prefixing is used to uniquely identify inherited features within a descendant, whereas renaming is used to create unique names for public inherited features.

Example 3.27 demonstrates the prefixing mechanism of Looks: in class C prefixes are used to access the private inherited attributes of classes A and B. Although these attributes have the same name, the prefixes prevent a name-conflict. This example also shows that prefixes may be used to indicate that a private inherited feature in a class (i.e. attribute \(x\) that class C has inherited from class A) may become public — and even renamed — in a descendant of that class (i.e. attribute \(A_x\) becomes public in class D and is renamed to \(y\)). Note that in class D attribute \(y\) is also known as \(A_x\) (even after renaming). However, outside the class this attribute is only known as \(y\). In the inheritance specification of class D prefix ‘A:’ is mandatory because ‘inherit C(x)’ is impossible since class C does not contain a public or (own) private feature \(x\).

Example 3.27: Prefixing of inherited features

Suppose classes A and B are defined previously, and that these classes contain a public attribute \(x\).

```plaintext
class C
  inherit A, B
  public
  f (): void;
  implementation
  f (): void;
  begin
    A_x := 0; // A's x becomes 0
    B_x := 1; // B's x becomes 1
  end
endclass;

class D
  inherit C(A:x)
  // C's private inherited attribute A_x becomes public and is renamed to y
  public
  x: integer;
  f (): void;
  implementation
  f (): void;
  begin
    A_x := 0; // A's x (= B's y) becomes 0
    B_x := 1; // B's x becomes 1
    x := 2; // D's x becomes 2
    y := 3; // D's y (= A's x) becomes 3
  end
endclass
```

A class inherits the features of an ancestor only once. By default, the first time a class is encountered as an ancestor (either directly if it occurs in the list of inherited types, or indirectly as ancestor of one of these types), its features — including method-bodies — are inherited by the descendant. However, to overrule this default selection (i.e. when the descendant, say C (Figure 3.23), should inherit a method of a repeatedly inherited ancestor, say A, that is redefined in a class, say B, that occurs in another inheritance-path), the same mechanism may be used that

\(^{13}\)In principle also the actual generic types (if any) of an ancestor have to be provided; however, if the descendant has inherited from the ancestor in a unique way, the ancestor-name may be used as prefix.
3.5. Looks

also overrules the private nature of inherited features. Consequently, when a name of an inherited feature occurs after a type in an inheritance-specification, the feature will become public in the descendant, and if the feature is a method, the descendant will use the method-body of that type. This allows for a very flexible specification of repeated inheritance: for every repeated inherited method it can be specified which method-body should be chosen in the descendant (indeed, there may be several implementations of this method in the various inheritance-paths). Consequently, features may be used only once in an inheritance specification (trying to rename a feature in order to get around this condition will not succeed). The mechanism to overrule the default selection of method-bodies is shown in Example 3.28.

Example 3.28 : Overruling default inherited public method-body

class A
public
f () : void;
g () : void;
implementation
f () : void ... // body.f.A
  g () : void ... // body.g.A
endclass;

class B
inherit A(f,B,g)
  // inherited feature f is renamed to b
implementation
b.f () : void ... // body.f.B, note: 'b.f' and 'b' denote the same method
b.g () : void ... // body.g.B
endclass;

class C
inherit A(f,B,g)
  // first C inherits all features of class A, thus
  // * (public) A.f has body body.f.A
  // * (private) A.g has body body.g.A
  // then C inherits all features of class B, and indirectly again of class A
  // (since features can be inherited only once, no new features are inherited).
  // However, one of the defaults is overruled, resulting in the following situation:
  // * (public) A.f has body body.f.A
  // * (public) A.g has body body.g.A
endclass;

class D
inherit A(f,B,h)
  // illegal: feature f of class A and feature h of class B denote the same feature
endclass;

Figure 3.29: Inheritance-graph of the previous example

Note that in the previous example overruling the default implementation of a repeatedly inherited method has the side-effect that the method becomes public in the descendant (in class C the default implementation of (repeatedly) inherited method g is overruled by explicitly naming this method after type B, thus making this method

A

\[ \text{B} \]

C
public in class C). Clearly, this is an unwanted situation if the method should remain — or become — private in the descendant. Therefore, extra syntax is needed to indicate the private status of a method in a descendant: the name of such a method is placed between brackets (grammar rule 7b).

\[ \text{Example 3.30 shows how the default implementation of a private inherited method may be overruled.} \]

Suppose classes A and B are defined as in example 3.28

```java
class C
    inherit A(g),B(h)
    // first C inherits all features of class A, thus
    // • (public) A.f has body f.A
    // • (private) A.g has body g.A
    // then C inherits all features of class B, and indirectly again of class A
    // (since features can be inherited only once, no new features are inherited)
    // However, one of the defaults is overruled, resulting in the following situation:
    // • (public) A.f has body f.A
    // • (private) A.g has body g.B
```

As explained above, the implementation of a method-body consists of a statement. Furthermore, statements can be used as actions in a script. In the rest of this section, the statements of Looks and their constituent expressions are introduced.

**Definition 3.31: Expression**

An expression is a construction that evaluates to a reference-location or to an object-reference.


**Definition 3.32: Null-expression**

The null expression is an expression that evaluates to the special null object-reference.

This null object-reference may be used, for example, to specify that a reference-location does not contain a reference to an existing object (initially, all reference-locations contain the null object-reference).

```java
23a. \langle expression \rangle \quad ::= \text{"NULL"}
```

The null expression is the only expression in Looks that conforms to every type. In theory, such a null object-reference should be present for every type in order to obey Looks' strong typing rules. Since this would be very elaborate, the typing rules of Looks are relaxed in this respect. This is illustrated in Example 3.33 where in the first assignment statement the null object-reference conforms to type A and in the second assignment to type B (actually, the assignments in this example will not change the contents of reference-locations a and b because reference-locations are automatically initialized with the null object-reference).
3.5. Looks

Example 3.33: Conformance of null

Suppose classes A and B are defined previously.

```java
object a : A;
object b : B;
a := null; // null conforms to A
b := null; // null conforms to B
```

Definition 3.34: Self-expression

The self expression is an expression that evaluates to an object-reference that refers to the object in whose method-body this expression occurs.

Of course, a public message, i.e. a message requesting the execution of a public method, may be sent to this object, or its public attributes may be accessed (this may result in a new object-reference, such as in ‘self.x’). The self expression is illustrated in Example 3.35, where it is used in a method to implement the doubling of a list.

```java
23b. <expression> ::= "SELF" <expression_rest>
26. <expression_rest> ::= [ '.' <identifier> [ '(' <act-param_list> ')'] ]
27. <act-param_list> ::= [ <expression> [ ',' <expression> ] ]
```

Self expressions may occur only in method-bodies, because then it is clear to which object they refer. Consequently, when they occur in action-statements, a syntax error will be generated. Note that the self expression does not evaluate to a reference-location, therefore it may not occur on the left-hand side of an assignment statement, i.e. ‘self := ...’ is not allowed.

Example 3.35: Self expression

```java
class list
public
    concat (x : list) : list;
    double () : list;
private
    a : integer;

Implementation
    double () : list
    begin
        result(self.concat(self))
    end;
end:
```

Note that after self only public features are allowed, therefore using ‘self.a’ in example 3.35 is not allowed.

Definition 3.36: Basic object-reference expression

A basic object-reference expression is an expression that evaluates to an object-reference that refers to an object of a basic type.
Again, a message may be sent to this object (this may result in a reference to another object). Note that objects of basic types do not have attributes. The following basic object-references can be supposed to be pre-defined in Looks (see also Section 6.2.1):

- integer : e.g. 0, -1, 1 (rule 24a)
- real : e.g. 0.0, -3.14, 2.72e-5 (rule 24a)
- character: e.g. 'a', 'b', 'c' (rule 24b)
- string : e.g. "", "a", "abc" (rule 24c)
- boolean : true, false (rule 24d)

Definition 3.37 : Reference-location expression

A reference-location expression is an expression that evaluates to a reference-location, which contains a reference to an object.

Again, a message may be sent to this object, or its public attributes may be accessed (this may result in a new object-reference). Reference-locations are identified by means of a name (rule 24d). However, the names true and false may not be used as reference-locations, since they are used as basic object-references of basic type boolean. The following kinds of reference-locations exist:

- object reference-location (no prefix)
- attribute (prefix may be used to identify an inherited, private attribute, see Example 3.27)
- parameter (no prefix)
- local variable (no prefix)

The following grammar rules specify the syntax of reference-location expressions:

```
23c. <expression> ::= <expression.begin> <expression.rest>
24a. <expression.begin> ::= ['+' | '-' | number] [ <realrest> ]
24b. <expression.begin> ::= "" [ <character> '"
24c. <expression.begin> ::= "" [ <character> ]
24d. <expression.begin> ::= <ancestor> <prefix> <identifier> [ '(' <act.param.list> ')' ]
25. <expression.rest> ::= [ '.' <identifier> [ '(' <act.param.list> ')' ]
26. <act.param.list> ::= [ <expression> [ '.' <expression> ] ]
```

In Example 3.38 object reference locations appear on the left-hand side of an assignment, and basic object-references appear on the right-hand side.

Example 3.38 : Reference-locations

```
object i : integer;
object r : real;
object c : character;
object s : string;
object b : boolean;
i := 5;
r := 0.0;
c := 'a';
s := "abc";
b := true;
```
3.5. **Looks**

In order to prevent name clashes the following rules must be obeyed:

1. object reference-locations should be unique
2. attributes should be unique within a class
3. parameters and local variables should be unique within a method

However, this does not disallow a name to be used as object reference-location, attribute, and parameter, simultaneously. In this case, the above scope rules are translated into priorities (with parameter as highest priority). The rationale for this decision is the relation between reference-location and method-body, where the name is used: parameters and local variables are more related to the method-body than attributes and object reference-locations, attributes are more related to the method-body than object reference-locations. This principle is illustrated in Example 3.39.

**Example 3.39 : Scope-rules**

```
object i,j,k : integer;
class A
  public
  l,j : integer;
  g (l : integer) : void;
  implementation
    begin
      l := 0; // parameter
      j := 0; // attribute
      g := 0; // object reference-location
      end;
```

**Definition 3.40 : Message expression**

A message expression is an expression that evaluates to an object-reference that is the result of the invoked method.

If the result-type of the method is void, the object-reference is null; otherwise, it refers to an object. As in the previous cases, a message may be sent to this object, or its public attributes may be accessed (this may result in a new object-reference). The message expression is illustrated in Example 3.41, which is a modified version of Example 3.35 (i.e. in Example 3.41 method concat has become private in class list, and inheritance of this class is used to show how a prefix may be used to send a message to this inherited, private method — to recall: prefixes are used to access inherited, private features). The optional ancestor-part may be used to invoke the original method-body of an ancestor (this is necessary in case the descendant redefines the method as an extension of the original; this concept is demonstrated in Example 3.42). The following grammar rules specify the syntax of message expressions:

23c. `<expression>` ::= `<expression_begin>` `<expression_rest>`

24d. `<expression_begin>` ::= `<ancestor>` `<pref>` `<identifier>` [ `<act_param_list> '.'` ]

25. `<ancestor>` ::= [ `<type>` `.'` ]

26. `<expression_rest>` ::= [ `.'` `<identifier>` [ `<act_param_list> '.'` ] ]

27. `<act_param_list>` ::= [ `<expression>` [ `.'` `<expression>` ] ]
Chapter 3. Design of Looks

The message expression and the self expression are related: self expression 'self.f()'... may be written as message expression 'f()'...'. However, the reverse does not hold: if g is a private method, 'g()'... is a correct message expression, but 'self.g()'... is not a correct self expression (indeed, according to the definition of self expressions only the public features of the object that is referred to by self may be used).

Message expressions may occur only in method-bodies, because then it is clear to which object they are sent, i.e. the object that is referred to by self. Consequently, when they occur in action-statements, a syntax error will be generated. Note that the message expression does not evaluate to a reference-location, therefore it may not occur on the left-hand side of an assignment statement, i.e. 'concat() := ...' is not allowed.

Example 3.41 : Message expression

```plaintext
class list
private
    concat (x : list) : list;
end;
endclass;

class my.list
inherit list()
public
    double () : list;
implementation
    double () : list
begin
result(list:concat(self))
    // self is used as value parameter of function concat
    // the result of function double is the concatenation
    // of self with itself
end;
endclass;
```

As explained previously, a descendant may redefine an inherited method. However, when the descendant wants to extend the inherited method, it needs to have access to the original method-body. For this purpose Looks supports an 'ancestor', which consists of an ancestor-type and two colons (grammar rule 25). This ancestor specifies the class whose method should be invoked. Although the concepts of prefix and ancestor are related, there is a fundamental difference between them: the former is used to indicate a private inherited feature in a descendant, whereas the latter is used to indicate original methods of an ancestor. Example 3.42 demonstrates how ancestors may be used. This example also shows that an ancestor and a prefix may be combined: 'B::A.f()' indicates the execution of the method 'A.f()' as it is implemented in class B; prefix 'A::' is required since method f is a private inherited feature in class B.

Example 3.42 : Ancestors and prefixes in expressions

```plaintext
class A
    public
    f () : void;
implementation
    f () : void ...// body1
endclass;

class B
    inherit A()
implementation
    A.f () : void ...// body2
endclass;
```
3.5. Looks

```plaintext
class C
  inherit B()
public
  g O: void;
Implementation
A: f (): void // body3
  g (): void
begin
  A::f(); // prefix: body3 is executed
  A::f(); // ancestor: body1 is executed
  B::A::f(); // ancestor + prefix: body2 is executed
  // B::f() cannot be used, since f is a private, inherited feature in B
end;
endclass;
```

If the definition of a method contains formal parameters, the message has to supply the actual parameters (in Example 3.41 argument self was used). For value-parameters the types of the formal and actual parameters do not have to be identical: it is sufficient if the type of the actual parameter conforms to the type of the formal parameter (cases 1 and 2 in Example 3.43).

If the formal parameter is a var-parameter, however, the corresponding actual parameter should evaluate to a reference-location and it should have the same type as the formal parameter. This is illustrated in case 3 in Example 3.43, where statement 'd.f2(0)' is just another way of writing 'i := 1.0'.

Basic object-references may not be used as actual var-parameters because these parameters should be reference-locations. This is illustrated in case 4 in Example 3.43, where statement 'd.f2(0.0)' is just another way of writing '0.0 := 1.0'.

Example 3.43: Typing rules

```plaintext
class dummy
  public
    f1 (x : real) : void;
    f2 (var x : real) : void;
  :
implementation
    f2 (var x : real) : void
    begin
      x := 1.0;
    end;
  :
endclass;

object i : integer;
object r : real;
  d.f1(0); // case 1: correct (note: integer conforms to real)
  d.f2(r); // case 2: correct
  d.f2(1); // case 3: not correct, because 1 does not have type real
  d.f2(0.0); // case 4: not correct, because 0.0 is no reference-location
```

Since method-names may be overloaded, and the types of actual value parameters do not have to equal the types of the corresponding formal value parameters (they only have to conform), there may be several method-headings that may qualify for a given message. The following rules are used to decide which method to choose (in order of importance):

```plaintext
1. Types match.
2. Types conform.
3. Most specific matching.
4. Least specific matching.
5. Best matching.
6. Name matching.
```
1. **direct hit**: if the types of the arguments of the message equal the formal parameter-types of exactly one method, that method will be chosen.

2. **matching types**: if the types of the arguments of the message conform to the formal parameter-types of exactly one method, that method will be chosen.

3. **failure**: no method will be chosen, otherwise.

Note that a direct hit does not imply that other methods do not match the message too. But since the direct hit is detected first, these (other) matching methods will not be taken into consideration. Example 3.44 shows how these rules are used to determine which method should be selected for a given message. It might be argued that for case 5 in this example heading 2 is a 'better' match than heading 1 and that therefore heading 2 should be chosen. However, case 6 demonstrates that it is not always possible to indicate a 'best' match. Therefore, if there is no unique match, a failure occurs.

Furthermore, the method is selected on the types of the reference-locations, not on the types of the objects that their object-references refer to during run-time. This facilitates a selection at compile-time, whereas using the types of objects a selection at run-time is required (this run-time selection is used in, for example, Smalltalk). Naturally, run-time method selection would harm the speed of the GDP. This situation is demonstrated in case 7 of Example 3.44.

**Example 3.44 : Overloading**

Suppose the following classes are defined previously:

```java
class A {
    public f (x : A; y : C) : void; // heading 1
    f (x : B; y : C) : void; // heading 2
    f (x : C; y : B) : void; // heading 3
}
```

```java
class dummy;
object a : A; object b : B; object c : C; object d : D; object dm : dummy;
```

```java
dm.f(a,a); // case 1: failure, no direct hit, no match
dm.f(a,c); // case 2: direct hit (heading 1), no match
dm.f(b,c); // case 3: direct hit (heading 2), match (heading 1)
dm.f(a,d); // case 4: match (heading 1)
dm.f(b,d); // case 5: failure, no direct hit,
            // 2 matches (headings 1 and 2)
dm.f(d,d); // case 6: failure, no direct hit,
            // 3 matches (headings 1,2, and 3)
```

```java
a := b;
dm.f(a,c); // case 7: direct hit (heading 1), no match
            // heading 2 is not chosen, although a contains
            // a reference to an object of type B
```

**Definition 3.45 : Comparison expression**

A comparison expression consists of two expressions and evaluates to a reference to the boolean object TRUE if these expressions evaluate to references to the same object; it evaluates to a reference to the object FALSE, otherwise.
In other words, by means of a comparison expression it is possible to detect aliasing. Of course, a message may be sent to the boolean object that is referred to by the returned object-reference. This is illustrated in Example 3.46 where reference-location eq3 will contain a reference to the boolean object TRUE, if (and only if) reference-locations a1 and b1 contain references to the same object and reference-location a2 contains the null object-reference. Note that in this example reference-locations a1 and b1 have different types. The following grammar rules specify the syntax of comparison expressions:

23d. `<expression>` ::= '(' `<expression>` ',' `<expression>` ')' `expression resto`
26. `<expression resto>` ::= { '.' `<identifier>` { { '.' `<act.param_list>` '.' } } }
27. `<act.param_list>` ::= [ `<expression>` { { '.' `<expression>` } ]

Although a comparison-function may be defined in a class, the result of this function may not be the same as the comparison expression. This, too, is illustrated in Example 3.46 where class A contains such a comparison-function (i.e. the function `equal`). If a1 and a2 contain references to the same object, both the comparison expression and the comparison-function return a reference to the boolean object TRUE. However, if they refer to different objects whose attributes have the same value, the comparison expression and the comparison-function return a reference to the boolean objects FALSE and TRUE, respectively. Thus, `(a1=a2)` implies `a1.equal(a2)`, but not vice versa.

**Example 3.46 : Comparison expression**

```plaintext
class A
public
  x : integer;
  equal (x : A) : boolean;
implementation
  equal (x : A) : boolean
  begin
    result((x=x.x)) // true if the argument has the same attribute
  end
endclass;

class B
// inherits all features of class A, but does not redefine function equal
implementation
endclass;

object eq1.eq2.eq3 : boolean;
object a1,a2 : A;
object b1 : B;
new(a1); new(a2); new(b1);
// create new objects and store references in a1, a2, and a3
a1.x:=0; a2.x:=0; b1.x:=0;
eq1 := (a1=a2); // false, because a1 and a2 do not refer to the same object
eq2 := a1.equal(a2); // true, because the x attribute of both objects is 0
a1 := b1; // a1 and b1 now refer to the same object
a2 := null; // a2 now contains the null object-reference
eq3 := (a1=a2).and((a2=null)); // true
```

Table 3.47 summarizes the characteristics of the various expressions in Looks: the ability of beginning with a prefix and the kind of value it returns (reference-location or object-reference).

**Definition 3.48 : Statement**

A statement is a construction to control the flow of a program or to change the value of a reference-location.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Prefix</th>
<th>Reference-location</th>
<th>Object-reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>null expression</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>self expression</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>basic object-reference</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>object</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>attribute</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>parameter</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>local variable</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>message expression</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>comparison expression</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.47: Summary of expressions

Thus, statements are the building blocks to implement the algorithms that specify the behavior of an animation. They may occur as an action-statement (grammar rule 2c) or in the body of a method (grammar rules 14, 18, and 19a). The former is generally used to start or to stop the movement of an object, whereas the latter is generally used to implement the movement of objects.

2c. <action> ::= <statement>
14. <implementation> ::= "IMPLEMENTATION" [<method_def> {"" <method_def> } ]
18. <method_def> ::= <prefix> <identifier> 
["(" [ <form_param_list> ] ")" ] <type> ]
<method_def_post>
19a. <method_def_post> ::= [ <loc.variables> ] <block_statement>

As defined by grammar rule 19a, statements can be grouped in a block-statement:

22. <block_statement> ::= "BEGIN" <statement> { "" <statement> } "END"
21k. <statement> ::= <block_statement>

Intuitively, the statements that are contained in a block-statement are executed in the same order in which they occur in the block-statement. Grammar rule 21k specifies that a block-statement may be used whenever an 'ordinary' statement is expected. Apart from the block-statement Looks contains ten statements: empty, result, allocation, assignment, repetition, selection, synchronous message, checked assignment, asynchronous message, and synchronize.

Definition 3.49: Empty Statement

The empty statement specifies that neither the control flow nor a reference-location is changed.

The empty statement is included to facilitate the definition of empty method-bodies. Grammar rule 21a specifies the syntax of the empty statement:

21a. <statement> ::= ε

14ε denotes the empty production rule and is not an element of the language.
3.5. Looks

Example 3.50 shows how an empty statement may be used to implement an empty method-body. Furthermore, this example shows that empty statements allow for multiple semicolons to be used successively (i.e. without generating a syntax error).

Example 3.50 : Empty statement

```
class A
    :
    implementation
    f (): void;
    begin
        // empty statement
        end;
    :
    endclass;
    ;; // too empty statements
```

Definition 3.51 : Result statement

A result statement returns values (i.e. object-references) from the body of a method to the caller of that method.

A method-body may contain several result statements, but only the result of the result statement that is evaluated last will be returned (i.e. the execution of a method-body does not terminate when a result statement has been encountered; in this respect the result statements differs from 'return' as found in C++). Furthermore, in order to obey the strong typing rules of Looks the type of the returned value has to conform to the result-type of the function (i.e. they do not have to be equal). Grammar rule 21d specifies the syntax of a result statement:

```
21d. <statement> ::= "RESULT" '(' <expression> ')'
```

Example 3.52 shows how a result statement is used in method zero of class A to return an object-reference to the integer object that represents value 0. Note that in this example the first result statement is overruled by a second, and final, one.

Example 3.52 : Result statement

```
class A
    :
    implementation
    zero (): real;
    begin
        result(1.0); // will be overruled
        result(0) // function returns integer 0 (which conforms to real)
        end;
    :
    endclass;
```

Definition 3.53 : Allocation statement

An allocation statement creates a new object and stores a reference to this object in a reference-location

\[\text{\textsuperscript{15}}\text{Note that Looks does not contain a statement to dispose allocated objects: all allocated 'un-referenced' objects (i.e. no object-reference refers to these objects) are automatically disposed by a garbage collection mechanism (see Section 3.5.2).}\]
However, since initially all objects of basic classes are present, no new objects of these classes can be created (trying to do so, will store a null reference in the corresponding reference-location). Grammar rule 21c specifies the syntax of an allocation statement:

\[
\text{Example 3.54 shows several (mis-)uses of allocation statements:}
\]

- case 1 is not allowed 0 is not a reference-location (0 is a basic object-reference),
- case 2 is allowed; because no new objects of basic class integer can be created, null is stored at reference-location i,
- case 3 is allowed and creates a new object of class A and stores a reference to this object in reference-location a,
- case 4 is not allowed because the evaluation of ‘a.f()’ returns an object-reference (instead of a reference-location as required by the allocation statement).

\[
\text{Example 3.54: Allocation}
\]

```java
class A
  public
    f() : A;
  ...
endclass;
...
object a : A;
object i : integer;
...
new(0);  // case 1: not allowed
new(i);  // case 2: allowed; stores null at i
new(a);  // case 3: allowed
new(a.f()); // case 4: not allowed
```

\[
\text{Definition 3.55: Assignment statement}
\]

An assignment statement stores an object-reference in a reference-location.

An assignment statement contains two expressions: the left-hand expression should evaluate to a reference-location, the right-hand expression should evaluate either to a reference-location or to an object-reference. Due to the strong typing of Looks, the types of these expressions should be related: the type of the right-hand expression should conform to the type of the left-hand expression. Grammar rule 21g specifies the syntax of an assignment statement:

\[
\text{Example 3.56 shows several (mis-)uses of assignment statements:}
\]

\[
\text{21g. <statement>} \quad ::= \quad <expression> \quad ' : = ' \quad <expression>
\]

The assignment statement is evaluated as follows: first, the left-hand expression is evaluated; second, the right-hand expression is evaluated; finally, the object-reference of the right-hand expression (or in case this expression evaluates to a reference-location: the object-reference stored in this location) is copied to the reference-location of the left-hand expression.
• case 1 is not allowed because 0 is a (basic) object-reference,

• case 2 is allowed and stores a reference to integer object THREE in reference-location i.

• case 3 is not allowed because reference-location i can only contain references to objects of class integer or descendants of integer (note that class integer has inherited from class real, and not vice versa),

• case 4 is allowed and stores a reference to integer object THREE in reference-location r (note that this reference-location may contain references to real objects, but because class integer has inherited from class real this reference-location may also contain references to integer objects),

• case 5 is not allowed because expression '0.plus(3)' does not evaluate to a reference-location (as explained previously, this expression does evaluate to an object-reference).

Example 3.56 : Assignment

object i : integer;
object r : real;

0 := 3;     // case 1: not allowed
i := 3;     // case 2: allowed
1 := 3.0;   // case 3: not allowed
r := 3;     // case 4: allowed
0.plus(3) := i;  // case 5: not allowed

Definition 3.57 : Repetition statement

A repetition statement executes another statement repetitively.

The number of times this other statement is executed is determined by an expression that evaluates to a boolean value (as long as the expression yields true the statement is executed). Grammar rule 211 specifies the syntax of a repetition statement:

211. <statement> ::= 'WHILE' <expression> 'DO' <statement>

In Example 3.58 a repetition statement is used to execute the assignment statement 'i:=i.plus(1)' ten times.

Example 3.58 : Repetition

object i : integer;

i:=0;
while i.lt(10) do i:=i.plus(1);

Definition 3.59 : Selection statement

A selection statement makes a choice between two statements according to a boolean expression.
If the expression yields true the first statement is executed, otherwise the second statement is executed. The second statement is optional, however. If it is omitted and the expression yields false, the evaluation of the selection statement is finished.

Grammar rule 21j specifies the syntax of a selection statement:

21j. \( \langle \text{statement} \rangle \) := "IF" \( \langle \text{expression} \rangle \) "THEN" \( \langle \text{statement} \rangle \) | "ELSE" \( \langle \text{statement} \rangle \) |

In Example 3.60, an implementation of Euclid's algorithm to determine the greatest common divisor, a selection statement is used to discriminate between the cases 'i greater than j' and 'i not greater than j'.

**Example 3.60 : Selection**

```
oBJECT i,j : integer;
;
\// i > 0, j > 0
\text{while} i \text{ mod}(j) \text{ do}
\text{if} i > (j)
\text{then} i:=(i \text{ div}(j))
\text{else} j:=(j \text{ mod}(i));
```

**Definition 3.61 : Synchronous message statement**

A synchronous message statement invokes a method of an object, while the sender of the message waits for the completion of the method.

```
\text{A synchronous message consists of an expression, which may be a self expression, a basic object-reference expression, a reference-location expression, a message expression, or a comparison expression, as long as it ends with a message to an object. It is not required that the result-type of this method is void (i.e. the method may be a function; in this case the returned value is just ignored). This facilitates a concise programming style if the result of a function is not important (see Example 3.62 where function addpoint adds a point to a shape and returns a corresponding index). Grammar rule 21e specifies the syntax of a synchronous message statement:}
```

21e. \( \langle \text{statement} \rangle \) := \( \langle \text{expression} \rangle \)

Example 3.62 also shows how a synchronous message is used to invoke procedure draw of the object that is referred to by reference-location sh.

**Example 3.62 : Synchronous message**

Suppose classes point and shape are defined previously, and that class shape contains the following methods:

- \text{addpoint} with signature 'addpoint \( (p\text{ point}) : \text{ integer} \)'
- \text{draw} with signature 'draw \( () : \text{ void} \)'

```
oBJECT sh : shape;
object p : point;

sh.draw(); // draw a shape
sh.addpoint(p); // adds point p to shape sh; allowed because the result-type of a
```

// synchronous message may be integer
3.5. **Looks**

Like most object-oriented languages, Looks supports *dynamic binding.* This means that it depends on the type of the object which method will be invoked, not on the (declared) type of the reference-location. Dynamic binding may be introduced by storing an object-reference into a reference-location where the type of the object-reference conforms to the type of the reference-location; in this case the object-reference is said to be cast to the type of the reference-location. There are two ways to introduce casting; either by means of an assignment or by means of actual value-parameters. These two cases are illustrated in Examples 3.63 and 3.64, respectively.

**Example 3.63 : Dynamic binding**

Suppose class `shape` and descendants `square` and `circle` are defined previously, and that class `shape` contains a method `draw()` with signature `draw () : void` which is redefined in classes `square` and `circle`.

```plaintext
object sh : shape;
object sq : square;
object c : circle;
...
sh := sq; sh.draw();  // draws a square (dynamic binding, due to assignment)
sh := c; sh.draw();  // draws a circle (dynamic binding, due to assignment)
```

**Example 3.64 : Dynamic binding**

Suppose the same classes are defined as in the previous example.

```plaintext
class dummy
  public
    show (sh : shape) : void;
  implementation
    show (sh : shape) : void;
    begin
      sh.draw();
    end;
  endclass;
object d : dummy;
object sq : square;
object c : circle;
...
d.show(sq);  // draws a square (dynamic binding, due to actual value-parameter
d.show(c);  // draws a circle (dynamic binding, due to actual value-parameter
```

Clearly, dynamic binding is a very powerful mechanism: statements do not have to contain complicated case analyses. Furthermore, future classes may also inherit from `shape`; due to dynamic binding, method `show` in class `dummy` also works for these future classes. This is a very important feature of a language, especially when used in an interactive programming environment.

**Definition 3.65 : Checked assignment statement**

A checked assignment statement determines the type of an object that is referred to by an object-reference, and stores a copy of this object-reference is stored in a reference-location, provided the strong typing of Looks is not violated.
This statement contains two expressions: the left-hand expression should evaluate to a reference-location, the right-hand expression should evaluate to a reference-location or to an object-reference. Grammar rule 21h specifies the syntax of a checked assignment statement:

\[
21h. \text{ <statement> } := \text{ <expression> 'if' <expression> }
\]

The checked assignment statement is evaluated as follows: first, the left-hand expression is evaluated; second, the right-hand expression is evaluated; finally, if the object that is referred to by the right-hand expression has a type that conforms to the type of the left-hand expression, the object-reference of the right-hand expression (or in case this expression evaluates to a reference-location: the object-reference stored at this location) is copied into the reference-location of the left-hand expression. If the types do not conform, the value null is copied to this reference-location. Subsequently, a comparison expression between the left-hand expression and null may be used to test whether the checked assignment was successful.

Since a checked assignment statement performs its own type-checking, there is no restriction on the types of the expressions that may be used in such a statement (note that this property is much weaker than the typing requirement of an assignment statement).

Example 3.66 shows how a checked assignment may be used to determine whether a real reference-location contains an integer object-reference.

**Example 3.66 : Checked assignment**

\[
\begin{align*}
\text{object } i & : \text{ integer;} \\
\text{object } r & : \text{ real;} \\
\text{:

i \leftarrow 0.0; \\
i \leftarrow r; \\
\text{if } (i = \text{ null}) \\
\text{then } // \text{ a alternative will be selected because } r \text{ did not contain } \\
\text{ a reference to an integer object } \\
\text{:

else } // \text{ this alternative will not be selected } \\
\text{:

i \leftarrow 0; \\
i \leftarrow r; \\
\text{if } (i = \text{ null}) \\
\text{then } // \text{ this alternative will not be selected } \\
\text{:

else } // \text{ this alternative will be selected because } r \text{ did contain a } \\
\text{ reference to an integer object (this reference is now } \\
\text{ copied to reference-location i) } \\
\text{:

}\end{align*}
\]

A checked assignment statement is very useful in combination with genericity, because it introduces a mechanism to access the features of formal generic types. Due to the strong typing of Looks, accessing these features directly is not allowed, since it is not known if the actual generic types do contain these features (for this purpose the language Eiffel supports constrained genericity: actual generic types have to conform to a certain type). Example 3.67 shows how checked assignments and (multiple) inheritance of generic type array and type writable (see Section 6.2 for a description of these classes) may be used to define a generic array that is also writable.
Example 3.67: Combination of genericity and checked assignment

Suppose class writable is defined previously, and that this class contains a method write with signature 'write (os:ostream) : ostream'.

```plaintext
class writable, array[T]
  inherit array[T](get, set, size), writable(write) // see Chapter 6
implementation
  var w : writable;
  begin
    w := x;
    if (w = null)
      then // Error: element x is not writable
        array::set(i, x) // see ancestor 'array::':
    else // set-method
        w.write(os); // see ancestor 'writable::write':
  end;

write (os : ostream) : ostream;
begin
  i := 0;
  while i lt size() do
    w := get(i); // checked assignment will always succeed because method set
    w.write(os); // get(i).write(os) is not allowed because get(i) has (formal)
    i := i.plus(1)
  end;
end;
```

An asynchronous message statement and a synchronize statement are used to specify quasi-parallelism, the form of intra-object concurrency supported by Looks.

Definition 3.68: Asynchronous message statement\(^{16}\)

An asynchronous message statement invokes a method of an object, while the sender of the message does not wait for the completion of the method.

An asynchronous message consists of an expression, which has to end with a message to an object. It is not required that the result-type of this method is void (i.e. the method may be a function; in this case the returned value is just ignored).

Definition 3.69: Synchronize statement

A synchronize statement switches between various asynchronous messages.

Grammar rule 21f and 21b specify the syntax of an asynchronous message statement and a synchronize statement, respectively:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>21f</td>
<td><code>&lt;statement&gt;</code> := <code>*&lt;expression&gt;</code></td>
</tr>
<tr>
<td>21b</td>
<td><code>&lt;statement&gt;</code> := <code>SYNCHRONIZE</code></td>
</tr>
</tbody>
</table>

\(^{16}\)To comply with object-oriented terminology, we have chosen for the name 'asynchronous message' instead of 'asynchronous procedure call'.

The difference between the synchronous and asynchronous message is the behavior of the sender of the message. In case of a synchronous message the sender will wait until the receiver of the message has completed the requested method. In case of an asynchronous message, however, the sender will wait until the receiver has completed the requested method or the execution of the requested method has reached a synchronization point, whichever comes first. The sender then resumes his own operation, even if the receiver has not completed the requested method. To specify the synchronization points in a method, Looks contains a synchronize statement.

The concurrency mechanism of Looks works as follows: all asynchronous messages that have reached a synchronization point are stored in a set (say S); as long as S is not empty, one of its members is selected, say am, and subsequently executed until it is finished or a synchronization point is reached. In the latter case, message am is moved to a new set (say S'). This set will also be filled with the asynchronous messages that were started by the execution of message am and did not finish because they too have reached a synchronization point. Set S' will replace S when all members of S have been selected (i.e., when S is empty). Furthermore, at this time a new frame may be rendered, a new script fragment may be parsed, garbage may be collected, etc. The following algorithm in pseudo-code illustrates the concurrency mechanism of Looks:

```java
do true
	→ S' := Ø;
	do S ≠ Ø
		→ ||		am := "Asynchronous_message";
		am ∈ S;
		S := S \ {am};
	"execute am until termination or synchronization";
	S' := S' \ "asynchronous messages introduced by execution of am (including am itself if it did not finish)"

		// context switch to next thread of control
		||
	on;
	s := S';
	"rendering, parsing, garbage collection, ...";
	on;
```

Example 3.70 shows how a synchronize-statement and asynchronous messages may be used to implement two concurrently moving shapes (to illustrate the concurrency mechanism integer parameters are added that will be printed when a method is executed).

**Example 3.70 : Asynchronous message**

```java
class shape

public

move (i : integer) : void;

implementation
```
3.5. Looks

```
move (i : integer) : void;
begin
  while true do
  begin
    // code to move the shape
    out.write(i); // print i
    synchronize
  end
end:
```

Consider the statements:
- `s1.move(1); // synchronous message to s1`
- `s2.move(2); // synchronous message to s2`

They produce the following list of integers:
```
1 1 1 1 1 ...
```

Since the synchronous message to `s1` does not terminate, the synchronous message to `s2` will never be executed.

Consider the statements:
- `s1.move(1); // asynchronous message to s1`
- `s2.move(2); // asynchronous message to s2`

If the selection (`:=`) is deterministic, these statements produce the following list of integers:
```
1 2 1 2 1 2 ...
```

However, if the selection is non-deterministic, the following list may be produced:
```
1 2 2 1 2 1 1 2 ...
```

Consider the statements:
- `s1.move(1); // asynchronous message to s1`
- `s2.move(2); // synchronous message to s2`

They produce the same lists as in case the message to `s2` was asynchronous.

However, the statements following the synchronous message will never be executed, because the synchronous message does not finish.

In Looks non-deterministic selection is chosen, because it is more general than the deterministic selection.

Since Looks supports multi-threaded quasi-parallelism it is possible to send another asynchronous move-message to one of these objects, e.g. ‘‘s1.move(3)’. Thus, `s1` executes two move methods concurrently. If the move-method is called synchronously as a action-statement (e.g. ‘s1.move(4)’), the system ‘hangs’ because the next action-statement will be executed only after termination of move (note that move contains an infinite repetition, and will therefore never terminate).

There is a restriction with respect to synchronize statements and asynchronous messages: value-parameters and local variables of a method may not be used as var-parameters of an asynchronous message. Indeed, the lifetime of value-parameters and local variables is bounded by the execution of a method and may be exceeded by the lifetimes of the methods that are invoked by this method by means of asynchronous messages. If value-parameters and local variables were used as var-parameters of these asynchronous messages, the corresponding methods could try to use them even though they are already extinct. Example 3.71 demonstrates this (potential) problem with methods `f` and `g`, where method `g` is finished before the assignment of method `f` takes place.
Example 3.71: Problem due to var-parameter of asynchronous message

```lisp
class A
  public:
  f (var x : integer) : void;
  g () : void;
implementation
  f (var x : integer) : void;
  begin
    synchronise;
    x := 0;
  end;
  g () : void;
  var k : integer;
  begin
    f(1) // problem, because reference-location 1
    // disappears before the assignment in method f is
    // executed (location 1 is local for method g and
    // disappears when method g is completed)
  end;
endclass:
```

3.5.2 Run-time Environment

This section describes two aspects of the run-time environment of Looks: garbage collection and run-time library linking. Although they are not really part of the language itself, these aspects improve the ease of programming and the extensibility of the GDP enormously. Therefore, they are treated in combination with the language definition.

Garbage Collection

Since the memory space of a computer is limited, the amount of objects that can exist at the same time during execution of an (object-oriented) program is limited too. Therefore it is wise to deallocate the memory of objects that will not be referenced anymore. Basically, there are two ways in which this deallocation process can take place:

- explicit: the deallocation is specified by the programmer. This results in a tedious programming style, because the programmer has to be very accurate whether objects should be deallocated. If he deallocates an object that is still referred to, these references become dangling (i.e. they do not refer to an object and they are not equal to null. On the other hand, if he forgets to deallocate an object that will not be referenced anymore, the memory space is going to be exhausted gradually (this is known as memory leaking).

- implicit: the deallocation is not specified by the programmer, but is carried out automatically. This automatical deallocation process is known as garbage collection. Naturally, garbage collection reduces the programming effort considerably, since the programmer does not have to implement fussy administrations in order to detect loose objects. In other words: garbage collection facilitates concise programming (requirement 3)

The interpreter for Looks programs, which is contained in the GDP, supports garbage collection. Therefore no explicit deallocation syntax is needed in Looks. Information on how garbage collection is actually implemented in the GDP can be found in Section 5.3
3.5. **Looks**

Run-time Library Linking

To facilitate the extensibility of the GDP (requirement 6) run-time library linking is supported. This means that new **system-classes** (i.e. classes that may use machine-dependent libraries) may be added to the GDP without re-compilation of the GDP. Moreover, these classes may even be added to a — running — GDP. For example, run-time library linking may be used to define a (Looks) class *camera* whose implementation uses machine-dependent libraries, such as the XGL-library [Sun91b] for rendering (see Example 3.72 for the definition of such a camera class).

To specify that a Looks class, say A, uses compiled code\(^\text{17}\) the name of the library that contains this code has to be supplied. This is possible by means of keyword **EXTERNAL** and an (optional) library-name (in case the library-name is omitted, the name of the Looks class (i.e. A) is used as library-name):

\[ 3. \text{<heading> := <identifier> [ "<identifier.list> " ]}
\text{[ "EXTERNAL"] <identifier> ]} \]

The library containing the compiled code is linked to the GDP during parsing of the Looks class. Since parsing Looks fragments takes place when the GDP is already running, the linking also occurs at run-time. Furthermore, the machine-libraries which are used by this compiled code will also be linked at run-time (hence: run-time library linking). In Example 3.72 *camera.library* is the name of the library that contains the compiled code.

This compiled code may include a compiled object which may be incorporated in a Looks object. Thus part of a Looks object may consist of compiled code. In this way, machine-dependent, low-level information (e.g. rendering information for the XGL-library [Sun91b]) can be stored in Looks objects. To specify the compiled object that contains this low-level information the keyword object and an (optional) object-name should be used (in case the name is omitted, the name of the Looks class (i.e. A) is used):

\[ 4b. \text{<class.kind> := "OBJECT" [ <identifier> ]} \]

In Example 3.72 *camera.object* is the name of the compiled object (to be found in library *camera.library*) that will be incorporated in the Looks *camera* objects.

To fully benefit from the possibility to use compiled code in a Looks class, methods of this class should be implementable by means of this compiled code. In this way, the methods can access the low-level information that is stored in the accompanying compiled object. To specify that a method, say f, is implemented by means of compiled code, the symbol "#" and an (optional) function-name should be used (in case the name is omitted, the name of the method (i.e. f) is used):

\[ 19b. \text{<method.def.rest> := "#" [ <identifier> ]} \]

In Example 3.72 *camera.zoom* is the name of a compiled function (to be found in library *camera.library*) that implements the body of the Looks method *zoom*. In this manner function *camera.zoom* provides an interface to the compiled object *camera.object*.

\(^{17}\)Compiled code is executed directly by the hardware on which the GDP runs, i.e. compiled code is not evaluated by the GDP's interpreter.
The evaluation — by the interpreter of the GDP — of a \textit{(Looks)} method that is implemented by means of compiled code, consists of the execution of this compiled code directly by the hardware.

Example 3.72 : \textit{Run-time library linking}

```java
class camera
    extends camera.library
object camera.object
public
    zoom() : void;
implementation
    zoom() = void #camera.zoom;
endclass;
```

Information on how run-time library linking is actually implemented in the GDP, and restrictions on the compiled code can be found in Section 5.4

3.6 Evaluation

Section 3.2 introduced the requirements that the script-language of the GDP has to meet. These requirements are:

1. animation-oriented
2. universal
3. concise
4. comprehensible
5. reliable
6. extensible
7. interactive
8. well-defined
9. efficient

Table 3.73 evaluates to what extent the various concepts of Looks relate to these requirements (the numbers in this table refer to the requirements above). A '+' sign means that the corresponding Looks concept contributes to fulfilling the associated requirement; a '-' sign indicates that the corresponding Looks concept conflicts with the associated requirement. In this table 'fully object-oriented' means that even basic values are provided as objects; the system classes are presented in Chapter 6.

As can be seen from this table, object-orientation and concurrency based on quasi-parallelism, asynchronous messages, and global synchronization match with the requirements for an animation system (see Section 3.3.4). They can be used in a variety of applications and allow for concise programming:

- object-orientation: classes facilitate the definition of isomorphic objects; inheritance and genericity facilitate the definition of related classes; dynamic binding facilitates message passing to related objects.

- concurrency: inter-object concurrency facilitates the definition of entities in an animation that may perform several operations concurrently.

It should be noted that the global synchronization mechanism cannot be used to control concurrency \textit{between} frames.
### Evaluation of Looks concepts

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object-Orientation</strong></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* generiocity</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* multiple and repeated inheritance</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* fully object-oriented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* system classes</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* dynamic binding</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* quasi-parallelism</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* asynchronous message</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* global synchronization</td>
<td></td>
<td></td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* interpreted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* few expressions and statements</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* strong typing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* garbage collection</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* run-time library linking</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.73: Evaluation of Looks concepts

Furthermore, the concepts of object-orientation and concurrency as used in Looks consist of only a few constructs. Although in general these constructs are relatively simple, one combination is quite involving: repeatedly inheriting a private feature. On the other hand, it allows for a very flexible inheritance specification (see Example 3.30). Since basic types are supported in Looks by means of classes, there is almost no dedicated syntax for these types. The small set of simple, yet expressive statements and expressions also contributes to the comprehensibility of Looks.

Object-orientation promotes reuse of existing classes either by creating objects of these classes or by defining new classes that have inherited from them. In both cases the code of the existing classes may be used. Therefore, the availability of a reliable class-library will facilitate the development of reliable programs. Furthermore, since Looks is strongly typed, it is guaranteed that no type errors will occur during the execution of a Looks program. Consequently, no type checking needs to be done at execution-time, which contributes to an efficient execution.

Clearly, defining new classes that have inherited from existing classes allows for the extension of the animation system with new functionality.

The fact that Looks is an interpreted language contributes to the fact that it is very well suited to interactively control the GDP. Unfortunately, interpreted languages are slower than compiled languages. The interactive character of the GDP, however, is more important than its execution speed. Therefore, no compiled language has been chosen. To support efficient execution of time consuming algorithms, Looks supports run-time library linking. This facilitates the GDP to load and execute compiled code. This mechanism consists of a few Looks constructs and allows for extending the GDP with new functionality.
The run-time environment of Looks supports garbage collection. This means that
Looks programmers do not have to specify the deallocation of objects. As explained
in Section 3.5.2, this contributes to concise programming, because the Looks program-
mer does not have to implement a complicated object-administration. Furthermore,
programs will be more reliable, because no dangling references or memory leaks will
occur. Unfortunately, garbage collection will decrease the performance of the GDP
(but as will be argued in Section 5.3 this slowdown is only marginal).
With respect to the syntax and semantics of Looks being well defined we refer to
Appendices A-E.
Chapter 4

Design of the GDP

4.1 Introduction

This chapter describes the design of the GDP, an implementation of a processor according to the definition of the Generalized Display Processor 1 type (see Chapter 2). First, Section 4.2 describes the internal structure of the GDP: its four functional components (i.e., parser, interpreter, renderer, and symbol-table), the communication between these components, and the communication between the environment of the GDP (application, windowing system, and graphics hardware) and these components. The four components of the GDP are discussed in the subsequent sections.

4.2 Internal Structure

Upon receiving a Looks script the GDP has to parse the script, interpret the script, and render the corresponding frames. These tasks are carried out by three distinct components of the GDP: the parser, the interpreter, and the renderer. The fourth and last component of the GDP is a symbol-table that is used by the other three components to store and retrieve information on classes and objects. Figure 4.1 contains a graphical representation of the internal structure of the GDP.

The communication between the various components of the GDP can be classified as follows:

- **Parser → Symbol-table**
  The parser stores class-definitions, declarations, etc. in the symbol-table. It uses this information to analyze new script fragments, for example, class-names should be unique, types should be checked, etc.

- **Parser → Interpreter**
  The parser translates the Looks statements that occur as actions in a script to abstract syntax trees (ASTs); these ASTs are subsequently sent to the interpreter.

- **Interpreter ← Symbol-table**
  The interpreter stores and retrieves information on references that are used during interpretation (e.g., for the interpretation of the Looks statement ‘x:=y’ it retrieves information on object-reference y and it updates object-reference x such
that $x$ will refer to the same object as reference $y$). Furthermore, the interpreter retrieves information on method-bodies (e.g. for the interpretation of the Looks statement `$x.f()$' it retrieves the AST that corresponds with method $f$ of the object that is referred to by object-reference $x$).

- **Interpreter → Renderer**
  The interpreter signals the renderer when a new frame has to be generated (i.e. when all asynchronous messages have reached a synchronization point).

- **Renderer → Symbol-table**
  The renderer retrieves information on cameras (e.g. their position, direction, zoom factor, and associated canvas) and visual objects (e.g. their position and color).

Section 2.4 describes roughly the communication between the GDP and its environment. As shown in Figure 4.1 several communication channels between the GDP and its environment exist:

- **Application → Parser**
  This unidirectional channel is used by the application to send Looks script-fragments to the parser. Since the application should not be blocked whenever it sends Looks scripts to the GDP, this channel should act as a buffer\(^1\).

- **Application → Interpreter**
  This bidirectional channel is used by the application to send information to the interpreter. Normally, the GDP will be waiting for this information. For example, when the GDP executes fragment (classes `file` and `color` are explained in Sections 6.2.3 and 6.6, respectively)

\(^1\)The size of the buffer is implementation dependent.
4.3. Parser

'object in : ifile; object col : color; new(in); new(col); in.open(); in.read(col),' the GDP will wait for the application to send a color over this channel — even possibly concurrent methods do not proceed any further.

In the other direction this channel is used by the interpreter to send information to the application. For example, when the GDP executes fragment

'object out : ofile; new(out); out.open(); out.writeln(red),' the GDP sends the information on an object of type color, called red, to the application. Since the GDP should not be blocked whenever it sends information to the application, the channel should act as a buffer.

• Parser/Interpreter — Application
This unidirectional channel is used by the parser and interpreter to inform the application whenever errors have occurred, e.g. a grammar violation (parser), a context condition violation (parser), or a division by zero (interpreter).

• Interpreter — Windowing System
This bidirectional channel is used by the interpreter to inspect and set properties of windows on the screen. The GDP may use several windows on a screen, most notably are the camera-windows of course, but also user-defined windows to control an animation may be defined in a Looks script (see Section 6.5). The channel is used by the windowing system to inform the interpreter whenever an event — either a keyboard event or a mouse event — has occurred in one of the GDP’s windows.

• Renderer — Hardware
This unidirectional channel is used by the renderer to draw a new frame on the screen. The renderer can retrieve from the symbol-table an identification in which window the frame should be drawn. The latter identification is stored in the symbol-table by the interpreter, which has received the identification from the windowing system.

4.3 Parser

The main task of the parser is to analyze whether a script-fragment that is received from the application conforms to the grammar of Looks (Appendix B). Furthermore, the parser has to check whether the script-fragment obeys the context conditions of Looks, e.g. class-names should be unique, types should be correct, etc. (see [Pee95] for the context conditions of Looks).

As explained in Section 3.5.1 a Looks program consists of actions: forward class declarations (grammar rule 2a), class-definitions (grammar rule 2b), reference-location declarations (grammar rule 2d), and statements (grammar rule 2c). The first three kinds of actions do not have an immediate effect on an animation: they describe classes (2a and 2b) and reference-locations (2d) which may be used later on. For this purpose, they are stored in the symbol-table. However, the fourth kind of actions, i.e. statements (2c), may have an immediate effect on the animation. For these action-statements the parser constructs corresponding abstract syntax trees (ASTs) in the

2 Again, the size of the buffer is implementation dependent.
abstract syntax of Looks (Appendix C). These ASTs are subsequently sent to the interpreter where they are evaluated according to the semantics of Looks. Since the interpreter may be blocked — due to the interpretation of a previous action-statement — the channel between the parser and the interpreter acts like a buffer.

Finally, when a Looks class definition indicates that a (machine dependent) library is required (see Section 3.5.2 for the appropriate Looks syntax), the parser has to link this library to the GDP. Since it has not to be known beforehand which libraries are going to be needed, the parser can link libraries at run-time. See Section 5.4 for an implementation of run-time library linking in a UNIX environment.

4.4 Interpreter

The interpreter is the functional heart of the GDP, because it actually executes the scripts that are received from the application. For this purpose it consists of an implementation of the evaluation functions as presented in appendices A and D. Although the functions EvalStatementList, EvalSimpleExpressionList, EvalSml, EvalAml, and EvalProgram have been defined in a recursive manner, we choose not to implement them this way, because the evaluation functions could then cause stack overflows. Instead, the evaluation functions are implemented imperatively. An object-oriented implementation of the abstract syntax in combination with the corresponding evaluation functions is presented in Section 5.2. The following algorithm shows the imperative version of function EvalProgram, i.e. the main loop of the interpreter:

```plaintext
do true
    "process asynchronous messages by means of EvalAml"
    "garbage collection"
    "rendering"
    "parsing"
    gd;
```

As can be seen from this algorithm, a garbage collector is invoked after all asynchronous messages are synchronized (garbage collection is discussed in Section 5.3). Subsequently, the renderer is signalled that a new frame has to be drawn. Finally, the interpreter processes the ASTs that are sent by the parser.

Like the parser, the interpreter may not crash due to errors in the script (e.g. divisions by zero, uninitialized references, etc.). Whenever such an error occurs, the interpreter informs the application — by means of the error channel from GDP to application — that a run-time error has occurred.

4.5 Renderer

The renderer is the part of the GDP that deals with the visual side of an animation: presenting the frames of the animation on a screen. What and where the renderer should draw is specified by means of virtual cameras (see Section 6.6.4); for example, a virtual camera contains the eye-position, the view direction, the up direction, and a view angle (which are necessary to specify what the renderer should draw), and it contains a reference to a canvas widget (which is necessary to specify the screen
4.6 Symbol-table

Information that is stored in the symbol-table can be divided into two categories: class information and object information. The class information is mainly used by the parser in order to analyze Looks fragments. The interpreter uses this information for accessing attributes of cast object-references (i.e., when the types of the object and reference-location are not identical) and for fetching method-bodies. The object information is mainly used by the interpreter to create new objects and to store and retrieve object-references. The parser only uses the object information to store and retrieve information on object reference-locations.

The class and object information are explained in Sections 4.6.1 and 4.6.2, respectively. Section 4.6.3 shows how class and object information are used to evaluate a Looks script.

4.6.1 Class-table

One of the tasks of the symbol-table is to register information on the classes that are defined in a Looks script. For each class, the following information has to be stored: name, formal generic types, ancestors, attributes, methods, and compiled object. For example, consider class A that is shown in Example 4.2.

Example 4.2: Sample class

```looks
class A[S,T]
  external library
  object dummy_object
  public
    f (var i : integer; b : boolean) : void;
  private
    x : A [integer, integer];
    y : T;
  endclass;
```

The information that has to be stored for this class consists of:

- name: "A"
- formal generic types: "[S,T]"
- ancestors: none
- attributes: "x : A [integer, integer]" and "y : T"
Chapter 4. Design of the GDP

- methods: 
  
  "f (var i : integer; b : boolean) : void"

- compiled object: the CPPObject returned by function "dummy_object"

For storing this information the class-table (CT) uses several types:

- \( CT = \{\text{cn:Name} \times \text{ft:FGT} \times \text{at:ANC}\}^* \), with:
  
  - cn: class name
  - ft: table of formal generic types
  - at: table of ancestors

- \( FGT = \text{Name}^* \)

- \( ANC = \{\text{t:Type} \times \text{a:ATT} \times \text{os:Index} \times \text{m:MET} \times \text{c:CP?Object}\}^* \), with:
  
  - t: ancestor type
  - a: attributes from ancestor t
  - os: offset of these attributes in the attribute-table of an object
  - m: methods from ancestor t
  - c: compiled object from ancestor t (see Section 3.5.2 for a discussion on compiled objects)

- \( Type = \{\text{i:Index} \times \text{c: Boolean} \times \text{at:AGT}\} \), with:
  
  - i: indicates an entry in the class-table (CT) or in the generic-type table (FGT)
  - c: specification whether i is an index in the class-table
  - at: table of actual generic types (as explained in Section 3.5, actual generic types have to be supplied for each formal generic type)

- \( AGT = \text{Type}^* \)

- \( ATT = \{\text{n:Name} \times \text{t:Type} \times \text{pub:Boolean}\}^* \), with:
  
  - n: attribute name
  - t: attribute type
  - pub: specification whether the attribute is a public or private feature

- \( MET = \{\text{n:Name} \times \text{p:PAR} \times \text{t:Type} \times \text{pub:Boolean} \times \text{a:AST} \times \text{ci:Index}\}^* \), with:
  
  - n: method name
  - p: table of parameters
  - t: result type
  - pub: specification whether the method is a public or private feature
  - a: body (which is an abstract syntax tree, AST)
  - ci: index used to cast an object-reference to the class in which the body is defined (used for inherited methods that are not redefined in a descendant; see Section 4.6.3 for an example)
4.6. Symbol-table

- PAR = (n:Name × t:Type × var:Boolean)*, with:
  
  n : parameter name
  t : parameter type
  var : specification whether the parameter is a var- or value-parameter

For each class, the first entry of its ancestor-table is reserved for the class itself. Consequently, the attributes and methods of a class are handled similarly to the inherited attributes and methods.

The following values show how these types can be used to store the information of class A from Example 4.2:

c = <...,("A",ft,at)>, say class "A" is stored at position 20 in ct
ft = <S,T>
at = <t0,a,0,m,dummy_object>, note: this entry refers to class "A"
a = <("x",t1,false),("y",t2,false)>
m = <("z",p,t3,true,ast,0)>, say ast represents the body of method "f"
p = <("i",t4,true),("b",t5,false)>
t0 = (20,true,<t6,t2>), t0 represents "A[S,T]"
t1 = (20,true,<t4,t4>), t1 represents "A[integer,integer]"
t2 = (1,false,<>,), t2 represents (formal generic type) "T"
t3 = (0,true,<>,), t3 represents "void" (say class "void" is stored at position 0 in ct)
t4 = (2,true,<>,), t4 represents "integer" (say class "integer" is stored at position 2 in ct)
t5 = (1,true,<>,), t5 represents "boolean" (say class "boolean" is stored at position 1 in ct)
t6 = (0,false,<>,), t6 represents (formal generic type) "S"

Figure 4.3 contains a graphical representation of this class table.

![Graphical representation of class table](image)

Figure 4.3: Tables used to store information of class A

Inheritance from a type, say anc, consists of appending a copy of the ancestor-table of anc, say ATanc, to the ancestor-table of the descendant, say ATdes. As a result, conforming the descendant to type anc only amounts to using an offset in ATdes; this offset indicates where the copy of ATanc starts in ATdes. In case of multiple repeated
inheritance from anc, several copies of ATanc will occur in ATdes. Naturally, these copies will share one attribute-table and one method-table. For example, consider the following case of multiple repeated inheritance:

**Example 4.4 : Multiple Repeated Inheritance**

```plaintext
class A ... endclass;
class B inherit A() ... endclass;
class C inherit A() ... endclass;
class D inherit B(),C() ... endclass;
```

The classes from Example 4.4 have the following ancestor-table structures:

- class A : <A>
- class B : <B,A>
- class C : <C,A>
- class D : <D,B,A,C,A>

For example, conforming class D to C amounts to using offset 3 in the ancestor-table of class D. Indeed at offset 3 starts a copy of the ancestor-table of class C. This example also shows that multiple repeatedly inherited classes, such as class A, have to exist several times in the ancestor-table of the descendant. Otherwise, type-conformance cannot be implemented by simply using an offset in the ancestor-table of the descendant.

The interpreter uses the class-table to fetch the bodies (ASTs) of the methods that are invoked by means of messages. Of course the interpreter should be as fast as possible, therefore fetching method-bodies should not introduce a big overhead. As can be inferred from the type definitions above, every method ("body") of a class can be fetched by means of two indirections (i.e. constant time): the first indirection in the ancestors-table, and the second indirection in the method-table. The two indices that specify these indirections form type fexPair:

```
IndexPair = Index × Index
```

As shown in Section 4.6.2 fetching method-bodies of cast object-references can be performed in constant time by adding an offset to the first index of an .indexPair.

### 4.6.2 Object-table

The symbol-table also has to register information on reference-locations, object-references, and objects. For each reference-location the following information has to be stored: its name, its type, and its value. For storing this information the object-table (OT) uses several types:

- \( OT = (\text{In:Name} \times \text{It:Type} \times \text{lv:Value})^* \), with:
  - \( \text{In} \) : name of a reference-location
  - \( \text{lt} \) : location type
  - \( \text{lv} \) : object-reference stored at the location with name \( \text{In} \)

- \( \text{Value} = \{\text{null}\} \cup \text{Basic.Value} \cup \text{Object.Value} \)
4.6. Symbol-table

- Basic.Value = (vt:Type×ci:Index× bv:(Boolean|Real|Integer|Char|String)),
  with:

  vt : value type
  ci : cast index (explained below)
  bv : basic value

- Object.Value = vt:type × ci:Index × att:Attributes, with:

  vt : value type
  ci : cast index
  att : attribute-table of an object (note that all — own and inherited —
       attributes of an object are stored in one table)

- Attributes = Value–

As mentioned in Section 3.5.1, Looks supports aliasing of objects (i.e. multiple object-
references may refer to the same object). In terms of the tables above, this means that
multiple Object.Values may refer to the same attribute-table.
Since the interpreter should be as fast as possible, accessing the attributes of an object
should not introduce a big overhead. The type definitions above seem to indicate that
one index is enough to access an attribute of an object (indeed, all attributes of an
object are stored in one table for that object). However, due to repeated inheritance
the order of attributes in a descendant may differ from the order in an ancestor. For
example, suppose each class of example 4.4 defines two integer attributes x and y. The
attribute-tables of objects of these classes would then have the following structures:

- object A : <A:x,A:y>
- object B : <B:x,B:y,A:x,A:y>
- object C : <C:x,C:y,A:x,A:y>
- object D : <D:x,D:y,B:x,B:y,A:x,A:y,C:x,C:y>

Indeed, the order of the attributes is not fixed during multiple repeated inheritance
(e.g. consider the structures of classes C and D). This is caused by the fact that
inherited attributes (and methods) are added only once to a descendant.
Consequently, two indices (i.e. an IndexPair) are needed to access an attribute: the
first index indicates an ancestor, the second index indicates an attribute from that
ancestor. For example, IndexPair (3,1) would access attribute C.y of an object of
class D, because 3 is the index of ancestor C in the ancestor-table of class D, and 1 is
the index of attribute y in class C. However, C.y is stored in the attribute-table of class
D at position 7 (see the structures above). Therefore, there has to be a mapping from
IndexPair (3,1) to position 7. As explained in Section 4.6.1, each entry in the ancestor-
table contains an index os that indicates the offset of the ancestor’s attributes in the
attribute-table of an object. For example, the classes above would have the following
offsets:
• class A: \(<(A,0)\rangle\), i.e. ancestor A has offset 0

• class B: \(<(B,0),(A,2)\rangle\)

• class C: \(<(C,0),(A,2)\rangle\)

• class D: \(<(D,0),(B,2),(A,4),(C,6),(A,4)\rangle\)

Indeed, the attributes of ancestor C start at position 6 of the attribute-table of an object of class D. Consequently, accessing an attribute that is indicated by an IndexPair (ai,aj) consists of retrieving the offset of the ancestor with index ai and adding aj to this offset (note that this operation requires constant time). For example, consider IndexPair (3,1) for an object of class D: index 3 in the ancestor-table results in an offset 6, adding 1 to this offset results in 7 (the position of C:y in the attribute-table of D).

A last complication is caused by casting: object-references may be stored at reference-locations of a different type (as long as the types conform). For example, consider the following Looks fragment:

'object c:C; object d:D; new(d); c:=d;'

After execution of this code, the reference-location c and d contain the following information:

• OT: \(<...,c,C,cv),(d,D,dv)\rangle

  \(cv = (D,3,attr)\), note: the current ObjectValue type of c is D

  \(dv = (D,0,attr)\)

• attr = \(<D:x,D:y,B:x,B:y,A:x,A:y,C:x,C:y\rangle\)

So the assignment 'c:=d;' has as a side effect that the value of the cast index of c is updated.

Next, suppose attribute (1,1) of c should be accessed (note: IndexPair (1,1) indicates A:y in class C). Value cv has a value type D and cast index 3, indicating that object-reference is cast to the third ancestor of type D. An attribute (ai,aj) is accessed by retrieving the offset of the ancestor with index (ai + cast index), and adding index aj. For attribute (1,1) of c this comes down to: index (1 + 3) in the ancestor-table of class D results in an offset 4, adding 1 to this offset results in 5 (the position of A:y in the attribute-table attr).

Because Looks supports dynamic binding, the method that will be fetched does not depend on the type of the reference-location, but on the type of object that is referred to. This is similar to accessing attributes, as described above, where the value type is used to compute the position of an attribute (e.g. type D was used to compute the position of an attribute of c, because c contained a reference to an object of type D).

The following two procedures describe how methods can be fetched and how attributes can be accessed in constant time even though repeated inheritance, redefinition, or casting may have occurred (let val be a Value, val\#null, and class the entry in CT that corresponds with val.vt):

• for IndexPair (mi,mj) the following method is fetched:

\(\text{class.at}[mi+\text{val.ci}][m]MJ]\)

\(^3\text{During execution of the method, its cast index ci replaces val.ci.}\)
For IndexPair \( (ai,aj) \) the following attribute is accessed:
\[
\text{val.at}(\text{class.at}[\text{val.ci}+ai].\text{os}+aj)
\]
As can be seen method fetching and attribute access require 2 indirections and 1 respectively 2 additions.

### 4.6.3 Example

This section demonstrates how some simple Looks classes are stored in the symbol-table. It also shows how the procedures from the previous section are used to fetch methods and access attributes.

First, consider the following example.

**Example 4.5 : Simple Looks Classes**

```haskell
class A
  public
  x, y, z : integer;
  f () : void;
  g () : void;
  implementation
  f () : void
  begin
    z := 0;
    // attribute: (0,2), index 0 indicates ancestor A, index 2 indicates attribute z
  end
endclass;

class B
  inherit A(f,g)
public
  p, q : integer;
  implementation
  f () : void
  begin
    x := 0;
    // attribute: (1,1), index 1 indicates ancestor A, index 1 indicates attribute y
  end
endclass;

class C
  inherit B(f,g)
public
  b, k, m : integer;
  h () : void;
  implementation
  h () : void
  begin
    l := 0;
    // attribute: (0,1), index 0 indicates ancestor C, index 1 indicates attribute l
  end
endclass;

object a, ac : A; object b, bc : B; object c : C;
new(a); new(b); new(c);
ac := c; bc := c;
```

The class-table as produced by the parser is shown in Figure 4.6 (suppose classes void, boolean, and integer are stored at the first three entries of the class-table; classes A, B, and C are stored at positions 40, 41, and 42 of the class-table; fB, gA, and hC are the ASTs that correspond to the bodies of methods f, g, and h, respectively).

Example 4.5 contains two statements that introduce casting: 'ac:=c' and 'bc:=c'. As explained in the previous section, casting results in values that have a cast index ci
greater than 0 (to recall: this cast index indicates where the location type can be found in ancestor-table of the value type). The object-table as produced by the parser and interpreter after parsing and interpreting the script of Example 4.5 is shown in Figure 4.7.

Table 4.8 shows the method and attribute fetching for various messages. For every message the following information is listed: the IndexPair of the method as returned by the parser, the IndexPair of the method as computed by the fetching mechanism, the class where the fetched method is defined, the cast index during evaluation of the method, the attribute that will be accessed during evaluation of the method.

For example, consider message ‘ac.g()’. The parser returns IndexPair (0,1), because g is the second method of ancestor 0 of type A — the type of reference-location ac. The fetching mechanism returns IndexPair (2,1) and type C, because ac contains a reference to an object, say obj, of type C, where method g is fetched by means of IndexPair (2,1). Furthermore, since class C has not redefined method g, obj has to
be cast to type A (where method g is implemented). This casting is indicated by cast index 2. Finally, the offset of attribute z, indexed by (ai,aj) = (0,2), which is accessed during the execution of method g, is computed using the procedure described in Section 4.6.2. Namely:

- \( \text{val.ci} + ai = 2 + 0 = 2 \)
- \( \text{CT}[42].at[2].os = 5 \), i.e. the attributes of the third ancestor, i.e. A, start at position 5 in the attribute-table of an object of class C.
- \( \text{CT}[42].at[2].os + aj = 5 + 2 = 7 \), i.e. attribute z is stored at position 7 in the attribute-table.

---

4Method g is implemented in the third ancestor of class C. Therefore objects of class C have to be cast to this ancestor before method g can be executed.

5Method f is not implemented in class A.
Chapter 5

Implementation Aspects

5.1 Introduction

This chapter presents some issues in the implementation of the GDP on a UNIX system using the programming language C++ [Str91] (see also [Pee93]). Although these issues are explained in context of the GDP, they are relevant for the implementation of a variety of interpreters on such a system. Aspects that are only useful for the GDP itself, such as the implementation of the Looks symbol-table, are not discussed in this chapter.

First, Section 5.2 presents an object-oriented implementation model of the terms and evaluation function as introduced in Appendix A. Second, Section 5.3 deals with a garbage collection mechanism that can be used in the GDP. Finally, Section 5.4 shows how run-time library linking can be implemented.

5.2 Abstract Syntax Trees

In the GDP Abstract Syntax Trees (ASTs) are used to represent statements that occur in a Looks script (for a discussion of ASTs in combination with programming language implementation, see [Wat93]). ASTs are generated by the GDP’s parser, which sends them to the GDP’s interpreter where they are evaluated according to the semantics of Looks.

In traditional interpreters this evaluation mechanism is specified as a combination of recursive evaluation functions. In general, each different node in an AST results in one evaluation function. Sometimes a couple of these functions are combined in one function that contains a case-statement in order to distinguish the different nodes. This concept is shown in Example 5.1 by means of the evaluation functions for expressions, block-statements, and selections (the syntax in Example 5.1 is based on ANSI C).

Example 5.1: Traditional Evaluation Functions

```c
typedef ... expression;
typedef ... block;
typedef struct {
    expression exp;
    block b1,b2;
} selection;

int Evaluate_expression (expression e)
{
    ...;
}
```
void evaluate.block (block b)
{ ... };

void evaluate.selection (selection s)
{ if (Evaluate.expression(e.exp))
   Evaluate.block(c.bl);
   else
   Evaluate.block(e.bl);
}

This example clearly demonstrates some drawbacks of the traditional approach:

- Type definitions are tightly coupled (e.g. the type definition of selection is composed of the types expression and block).

- Evaluation functions are tightly coupled (e.g. the evaluation function Evaluate.selection uses the evaluation functions Evaluate.expression and Evaluate.block).

- Evaluation functions and type definitions are tightly coupled (e.g. the evaluation function Evaluate.selection uses type definition expression).

As a result of these drawbacks, extending or changing an AST requires changes at several places in the code — an undesirable property with respect to the maintainability of the interpreter.

These problems may be solved by using an object-oriented approach to specify ASTs: all nodes of an AST are objects of derived classes of a common base class, say AST. In this base class a method evaluate is defined that is re-implemented in the different derived classes (e.g. the evaluate method of class expression will differ from the evaluate method of class selection). Example 5.2 shows how the object-oriented approach may be used to specify the ASTs and evaluation functions from Example 5.1 (the syntax in Example 5.2 is based on C++).

Example 5.2 : Object-oriented Evaluation Functions

```cpp
class Container {
    ...
}
// base class for returned values of the evaluation functions

class Boolean_container : public Container {
    public:
        bool boolean();
    ...
};

class AST {
    public:
        AST's descendants;
        virtual Container* evaluate();
    };

class expression : public AST {
    public:
        Container* evaluate() {
            // return a Boolean_container with the value of the expression
        }
    };
```
5.3 Garbage Collection

```java
class block : public AST {
    public:
        container* evaluate() { ... };
};

class selection : public AST {
    public:
        container* evaluate() {
            if (((Boolean_container*)descendants[0].evaluate())->getboolean())
                return descendants[1].evaluate();
            else
                return descendants[2].evaluate();
        }
};
```

This object-oriented approach has been used in the implementation of the interpreter of the GDP: every operator of the abstract syntax of Looks (see Appendix C) corresponds with a derived class of base class AST. The evaluation functions of these derived classes are implemented according to the semantics of the corresponding operator (see Appendix D).

5.3 Garbage Collection

As explained in Section 3.5.2 garbage collection is a mechanism that relieves the programmer from the task to deallocate objects that are not used anymore. Several techniques for garbage collection have been proposed (e.g. see [Wil92] for a survey of uniprocessor garbage collection techniques). Some of the most popular approaches are:

- **reference-counting**
  This technique is based on maintaining the number of pointers that refer to each object. Whenever the last pointer to an object is about to be removed, the object can be deallocated.

- **copying**
  This technique is based on dividing memory space in two areas. At any time, exactly one of these areas is ‘active’. Garbage collection consists of copying all actual objects, i.e. the objects still in use by the program, from the ‘active’ area to the ‘passive’ area. After the copying process the ‘active’ and ‘passive’ area are switched.

- **mark-and-sweep**
  This technique is based on two phases:
  1. mark-phase: all actual objects are marked
  2. sweep-phase: all objects are checked and the unmarked objects are deallocated

Note that this technique requires that all objects, either actual or non-actual, can be reached in some way.
Reference-counting is very easy to implement: objects have to be extended with a reference-counter and pointer assignments have to be extended. Indeed, after the assignment the counter of the object that is originally referred to by the left-hand expression has to be decremented, while the counter of the object that is referred to by the right-hand expression has to be incremented. Furthermore, if the reference-counter of an object reaches zero, the object has to be deallocated. Because of the extension, the execution of pointer assignments becomes slower. Furthermore, reference-counting has a more serious drawback: it cannot deal with objects that have circular references. For example, if two objects refer to each other their reference-counters can never become zero, and consequently they will never be deallocated. Since Looks objects may refer to each other, reference-counting seems not to be the appropriate garbage collection mechanism for the GDP.

A copying garbage collector has a big advantage: as a result of copying objects from one area to another, there will be no 'memory holes' between the objects in the new area. However, at the same time it also introduces a drawback: objects do not have a fixed memory location (therefore an extra indirection may be needed). The most simple implementation of such a garbage collector really needs two memory areas, consequently it is rather demanding with respect to available memory. A more complicated implementation may relocate the actual objects within one memory area.

A mark-and-sweep garbage collector is related to a copying garbage collector: they are both based on traversing the reachable objects. However, they do not have exactly the same complexity: copying is proportional to the number of reachable objects, whereas mark-and-sweep is proportional to the total number of objects. On the other hand, copying a reachable object is more time-consuming than marking such an object.

Since mark-and-sweep garbage collection is able to collect circular references (as may occur in a Looks program), it is 'memory-friendly', and is also easy to implement; it has been chosen as the garbage collector of the GDP. Note that in the GDP the following objects are actual:

- all objects that can be reached via a chain of object-references starting at a reference-location
- all objects that can be reached via a chain of object-references starting at a local variable of an asynchronous message (indeed, the garbage collector may be called before an asynchronous message is finished)
- all objects that can be reached via a chain of object-references starting at a parameter of an asynchronous message

Example 5.3 shows in C++ style pseudo-code the mark-and-sweep garbage collection algorithm as used in the GDP. As shown in Section 4.4 this garbage collector is invoked after every frame.

Example 5.3: Mark-and-sweep Garbage Collection

```cpp
void MarkObject (Object obj) {
    if "obj is not marked" {
        "Mark obj":
        "for every attribute x of obj"
        MarkObject(x);
    }
}
```
Several techniques have been proposed to accelerate garbage collection [Wil92], for example:

- **Generation scavenging**: acceleration based on the notion that 'new' objects are more likely to be garbage than 'old' objects.

- **Incremental collection**: acceleration based on performing garbage collection interleaved with program execution.

As mentioned previously the complexity of the mark-and-sweep garbage collection as used in the GDP is proportional to the number of objects. However, this is not as bad as it may seem: many of the objects in the GDP will be topological primitives (see Section 6.6) and the cost of mark-and-sweep garbage collection is considerably less than the cost of rendering (tests with the prototype of the GDP have shown that garbage collection costs less than 1% of the execution time). Therefore accelerations as presented above are not really necessary in the garbage collector of the GDP.

### 5.4 Run-time Library Linking

Section 3.5.2) presents the syntax for adding new system-classes to a running GDP. This allows for using new machine-dependent libraries, or implementing part of the animation in C++. Run-time library linking consists of the following parts:

- opening a library
- fetching a compiled object from the library
- fetching a compiled method from the library
Library opening

When a Looks class uses compiled code (i.e., when the class-definition contains keyword `external` and an (optional) library-name), the corresponding library is opened by means of function `dlopen`1. Due to some limitations of UNIX the name of a library has to begin with 'lib' and has to end with 'so.0.0' (or some other numerals). However, the Looks programmer does not have to bother with this convention, because the GDP automatically adds the necessary pre- and postfix. For example, when the parser recognizes `external camera.library`, library `libcamera.library.so.0.0` is opened.

Compiled Object Fetching

As discussed in section 4.6.2 a Looks object may contain a pointer to a (compiled) C++ object. Due to implementation details discussed below, the class of such a C++ object has to be derived from a common base class, say CPPObj ect. Example 5.4 shows the outline for a C++ camera class that is to be used in a Looks camera.

**Example 5.4 : C++ camera class**

```cpp
#include <GL.h>
class camera : public CPPObj ect {
    // xgl-related data
    private:
        void setup() {
            ... // code to change view
        }
};
```

When the parser recognizes the keyword `object` and an (optional) object-name in a class definition, a corresponding C++ object should be created and added to the appropriate Looks class. However, function `dlsym` returns the address binding of a symbol, which is not very useful for a class (indeed, we need to create an object of the class). Consequently, for every C++ class that has to be linked at run-time to the GDP there has to be a function that creates an object of this class (by means of calling the constructor of the class). Due to implementation details discussed below, these functions may not have arguments and their result-type should be `CPPObj ect*`. Example 5.5 shows such a function for the camera class.

**Example 5.5 : Camera Creation Function**

```cpp
cpobject* camera_object() {
    return new camera();
}
```

---

1UNIX supports library linking mainly by means of two C-functions [Sun91a]:

- `void* dlopen(char*, int)`;
  
  This function provides access to the library that is named in the first argument [Sun91a]. It returns a descriptor that can be used for later reference to the library i.e. calls to `dlsym`. The second argument is reserved for future use (currently it has to be 1).

- `void* dlsym(void*, char*)`;
  
  This function returns the address binding of the symbol that is named in the second argument as it occurs in the library that is identified by the descriptor in the first argument.
5.4. Run-time Library Linking

Actually, the object-name in a class-definition is therefore not the name of a class but the name of a function that creates an object of that class. Example 5.6 shows the code that is used in the GDP for calling a function that creates a C++ object.

Example 5.6 : Object Fetching

```c
extern void* handle; // initialized by dispen
extern char* object.name; // initialized by the parser
typedef CPPObject* (*signature)();
// signature is a type of a pointer to a function with no arguments
// and result-type CPPObject*

signature function = (signature)dispen(handle, object.name);
// 'function' is a pointer to the function with no arguments and result-type CPPObject*
// read from the library by means of function dispen

CPPObject* cpp = (*function)(); // the function that is read from the library is executed
// cpp can now be added to the appropriate Looks class
```

As can be seen in the previous example it is necessary to indicate the signature of the functions that are used to create the C++ objects in order to be able to execute these functions. Consequently, these functions should have the same signature. However, these functions should create objects of various classes. This seems to be a contradiction, which can be be solved if these classes have a common base class. This class can subsequently be used as the result-type of the functions. Therefore, the classes of C++ objects that are going to be included in Looks objects have to be derived from a common base class CPPObject.

Compiled Method Fetching

As explained in section 3.5.2 it is also possible to specify that a Looks method is implemented by means of compiled code. Since the body of an ordinary Looks method is translated by the parser to an AST (the implementation of ASTs is discussed in section 5.2), it would be convenient if the compiled code is also an AST. Indeed, this provides a uniform way in which the interpreter can deal with both kinds of methods. Consequently, for every Looks method that is implemented by means of compiled code, a derived class of AST has to be created, where the compiled code is implemented as its evaluation method. Example 5.7 shows how the C++ implementation of (Looks) method zoom is incorporated in a derived class of AST.

Example 5.7 : AST for zoom method

```c
extern int invar; // initialized by the interpreter
extern symbolable st; // initialized by the interpreter
class zoom : public AST {
  public:
    void evaluate() { 
      camera* cam = (camera*)GetCPPObject2(Self(invar, st), 0);
      cam->zoom(); // execute the zoom method of C++ object cam
    }
};
```

For the same reasons as above, there have to be functions with identical signatures to create objects of these classes. Example 5.8 shows such a function for the zoom class.

\[\text{GetCPPObject is a function with signature: CPPObject* GetCPPObject(Value, int).}\]

It returns a CPPObject that is associated with the Looks object that is indicated by the first argument. The second argument indicates the index of the ancestor whose CPPObject has to be returned (see section 4.6).
Example 5.8: Zoom Creation Function

```plaintext
AST* camera_zoom() {
    return new zoom();
}
```

Example 5.9 shows the code that is used in the GDP for calling a function that creates an AST object with the implementation of a (Looks) method. This code is very similar to the code for creating a C++ object (Example 5.6), they only differ in the result-type of the signature.

Example 5.9: Method Fetching

```plaintext
extern void handle; // initialized by dpopen
extern char* method_name; // initialized by the parser
typedef AST* (*signature()); // signature is the type of a pointer to a function with no arguments
// and result-type AST*

signature function = (signature)disym(handle, object_name);
// 'function' is a pointer to the function with no arguments and result-type AST* as
// read from the library by means of function disym

AST* ast = (*function)(); // the function that is read from the library is executed
// ast can now be added as the body of a method in the appropriate Looks class
```

Library Creation

The final step in writing C++ code that has to be linked at run-time to the GDP, is to create a library. For example, suppose the C++ code from Examples 5.4, 5.5, 5.7, and 5.8 is stored in file ‘camera.cc’. Using a C++ compiler on a UNIX system this file is compiled to object-file ‘camera.o’. This object-file (or a combination of object-files) can be stored in a library by means of link editor ‘ld’:

`ld -o libcamera_library.so.0.0 camera.o`

Now, library libcamera_library.so.0.0 is ready for run-time linking to the GDP (see Example 3.72 on page 60).
Chapter 6

Class Library

6.1 Introduction

An object-oriented language is not very powerful in isolation. However, together with a class library it may provide a very useful programming environment. This chapter presents a class library for the GDP. Among the classes that are presented are: fundamental classes (6.2), a collection class (6.3), events and responders (6.4), widgets (6.5), rendering classes (6.6), direct manipulation classes (6.7), and classes for kinematics and rigid body dynamics (6.8 and 6.9).\footnote{The GDP automatically interprets an initialization file at start-up time. Many of the classes that are discussed in this chapter are included in this file. Consequently, users of the GDP do not have to define similar classes; on the contrary, they may use the classes that are provided in the initialization file (if these classes do not provide the required functionality, the inheritance mechanism of Looks may be used to implement more appropriate classes). Only the basic classes, presented in Section 6.2.1, are essential for the GDP and may not be removed from the initialization file; all other classes may be removed.}

An important issue in the design of this library is the concept of localization: operations that work on a collection of objects (e.g. event passing, rendering, and dynamics) are carried out automatically, i.e. they do not have to be programmed in Looks (therefore, methods have a local functionality for an object).

Note that rendering and direct manipulation are handled by toolkits such as Iris Inventor [SC92]. These toolkits cannot be used directly in the GDP, because they are designed for (compiled) C or C++, and not for Looks. This does not mean, however, that these toolkits cannot be used to implement rendering and direct manipulation classes of the GDP.

6.2 Fundamental classes

This section describes the fundamental classes of the GDP, i.e. classes that are essential for a majority of applications. The fundamental classes can be categorised as follows: basic classes (6.2.1), data-structure classes (6.2.2), input and output classes (6.2.3), position and direction classes (6.2.4).
6.2.1 Basic Classes

As mentioned in Section 3.5.1 basic classes are classes whose objects are initially present. Consequently, no new objects of basic classes may be created later on, and basic classes may also not be used as ancestors of non-basic classes. Since basic classes are used in the grammar of Looks, they must occur in the initialization file. For example, class void is used to indicate procedures and class boolean is used in the guards of the selection and repetition statement. The following basic classes are supported:

1. void
   This class is used to indicate procedures; no objects of this class are created.

2. boolean
   This class implements logical expressions in Looks; there are two objects of this class: TRUE and FALSE, which are referred to by basic object-references true and false, respectively. Some methods of this class are:
   - 'not():boolean' for negation
   - 'and(b:boolean):boolean' for conjunction
   - 'or(b:boolean):boolean' for disjunction
   For example, the logical expression '\( \neg (b0 \lor (b1 \land b2)) \)' is written in Looks as:
     b0.or(b1.and(b2)).not()
   Note that unary prefix operators become postfix messages in Looks. However, infix and postfix operators remain at the same position in a Looks expression.

3. real
   This class implements floating point expressions in Looks. The following methods are supported:
   - numerical operations (result-type real):
     a. no argument: abs, neg (negative), random, and sqrt. For example, expression \( \sqrt{x} \) is written in Looks as 'x.sqrt()'. Looks expression 'x.random()' returns a random real value in the domain \([0,x)\) or \((x,0]\) using a uniform distribution.
     b. one argument of type real: div, max, min, minus, plus, pow (involution), and times. These functions have one parameter of type real and their result-type is real.
   - logarithmical and trigonometrical functions: exp, ln, 10log, acos, asin, atan, cos, sin, and tan. These functions have no parameters and their result-type is real. For example, the expression \( e^x \) is written in Looks as 'x.exp()'.
   - comparison functions: eq (=), ge (≥), gt (>), le (≤), lt (<), and ne (≠).

---

\(^2\)Of course, the user may change the basic classes in the initialization file. For example, methods may be added, removed, implemented differently, etc.

\(^3\)Changebars are used to indicate rather straightforward classes. However, for reasons of completeness these classes are included in this chapter. Readers who have a background in object-orientation may decide to skip the fragments indicated by changebars.
These functions have one parameter of type real and their result-type is boolean.

- rounding functions: ceil, floor, and round. These functions have no parameters and their result-type is integer. For example, the expression \([x]\) is written in Looks as ‘\(\text{ceil}\)’.

4. integer

This class has inherited from class real (i.e. references to integer objects may be used whenever references to real objects are expected). Nearly all inherited methods from class real are made public, with the exception of methods neg, random, and abs. However, new methods with these names are declared public in class integer, this time with result-type integer. Furthermore, the basic numerical operations and the comparison functions are redefined to take into account that the operation may involve only integer values. For example, consider the following Looks fragment.

```looks
object i : integer; object r : real;
r:=3;
i:=2.plus(r);
// i refers to object FIVE
```

Since argument \(r\), with declared type real, refers to integer object THREE, the method plus uses only integer calculations. Note that the inherited signature of method plus has result-type real, therefore the statement \(i:=2.plus(r)\) will be rejected by the type-checking of Looks. Although these methods can also be used for integer arguments, e.g. \(2.plus(3)\), it is rather strange that these expressions have result-type real. Therefore, the numerical operations and comparison functions are also defined with integer arguments, and in case of numerical operations, integer result-types. For example, consider Looks fragment \(i:=2.plus(3)\). Both methods 'plus(x:real):real' and 'plus(x:integer):integer' apply. But, as explained on page 43, the method 'plus(x:integer):integer' will be selected because the type of the argument (3) equals the type of the declared parameter (i.e. it is a direct hit).

Finally, class integer contains the following two methods:

- fac():integer
  This function returns a reference to the integer object that corresponds to the factorial of the object whose function fac is invoked.

- chr():char
  This function returns a reference to the char object whose number (in the ASCII table) corresponds to the integer object. In case the number does not occur in the ASCII table, null is returned

5. char

This class provides character values in Looks. The following methods are supported:

- ord():integer
  This function returns the number of of the char object. For every char c the
following property holds: \( (c . \text{ord}() . \text{chr}() = c) \).
Note that the reverse 'i . \text{chr}() . \text{ord}() = i' only holds if i is a valid index in the
ASCII table.

- iteration functions: \text{pred} and \text{succ}. These functions have no parameters and
their result-type is \text{char}. They return the predecessor and successor of the
char object, respectively. For example, 'a'.\text{succ}() returns 'b'.

- conversion functions: \text{toupper} and \text{tolower}. These functions have no param-
ters and their result-type is \text{char}. They return the upper-case and lower-case of
the char object, respectively. For example, 'a'.\text{toupper}() returns 'A'.

- test functions: \text{isalnum} (test for letter or digit), \text{isalpha} (test for letter), \text{isdigit}
(test for digit), \text{islower} (test for lower-case), \text{isspace} (test for white space,
i.e. space, tab, carriage return, line feed, or form feed), and \text{isupper} (test
for upper-case). These functions have no parameters and their result-type
is \text{boolean}.

- comparison functions: \text{eq} (=), \text{ge} (\geq), \text{gt} (>), \text{le} (\leq), \text{lt} (<), \text{and ne} (\neq).
These functions have one parameter of type \text{char} and their result-type is
\text{boolean}.
The functions \text{max} and \text{min} are related to these functions. These functions
have one parameter of type \text{char} and their result-type is \text{char}.

\text{string(x:integer):string}
This function returns a reference to the string object that consists of x
characters. For example, 'a'.\text{string(5)} returns "aaaaa".

6. \text{string}
This class provides string values in Looks. The following methods are supported:

- \text{length():integer}
  This function returns the number of characters in the string.
  For example, "abcde".\text{length()} returns 5.

- \text{get(x:integer):char}
  This function returns the character at the x\text{th} position in the string (note:
  the first element in a string is numbered 0). In case x is out of range
  [0,\text{length}) null is returned. For example, "abcde".\text{get(2)} returns 'c'.

- \text{set(x:integer, y:char):string}
  This function returns the string that has character y at position x. In case x
  is out of range [0,\text{length}) null is returned. For example, "abcde".\text{set(2, 'X')}
  returns "abXde".
  Note that in Looks strings are atomic values: they cannot be changed (i.e.
  setting a specific character in a string results in a reference to another
  string).

- \text{concat(x:string):string}
  This function returns the concatenation of two strings.
  For example, "ab".\text{concat("cd")} returns "abcd".

- \text{substring(from, length:integer):string}
  This function returns the substring of length length that starts at position
6.2. Fundamental classes

For example, "abcde".substring(1,2) returns "bc".

- **string(x:integer):string**
  This function returns a reference to the string object with length x. For example, "abc".string(10) returns "abcabcabcabc" (this is a generalization of initializing a string with just one character).

- **comparison functions**: eq (=), ge (≥), gt (>), le (≤), lt (<), and ne (≠). These functions have one parameter of type string and their result-type is boolean.
  The functions max and min are related to these functions. These functions have one parameter of type string and their result-type is string.

6.2.2 Data-structure Classes

In the first prototype of the GDP only two data-structures are supplied:

1. **array[T]**
   Generic class array[T] implements a resizable array of references to objects of type T (of course, other data-structures — which may even use this resizable array — may be implemented in a Looks class). The following methods are supported in class array[t]:

   - **index(i:integer):boolean**
     This function returns whether i is inside the array-bounds.

   - **get(i:integer):T**
     This function returns a copy of the reference stored at position x of the array. If i is outside the array-bounds null is returned.

   - **set(i:integer; x:T):void**
     This procedure stores a copy of reference x at position i in the array. If i is not within the array-boundaries, an error-message is printed. Note that both entry i of the array and reference x refer to the same object.

   - **size(i:integer):void**
     This procedure sets the size of the array to i positions (starting at 0). If i is greater than the old size of the array, they array is resized and the positions from the old size to position i-1 are initialized with null.

   - **size():integer**
     This function returns the size of the array.

2. **list[T]**
   Generic class list[T] implements a list of references to objects of type T. This class is not an implementation of a universal list data-structure (e.g. it lacks concatenation and insertion at a given index), but it is rather dedicated to be used in other system-classes. The following methods are supported in class list[T]:

   - **'add(x:T):integer** for adding a copy of reference x to the end of the list. The returned integer is an index that may be used later to remove the reference from the list.
• index(integer):boolean
  This function returns whether 0≤i<size().
• `del(integer):void` for removing the reference with index i (at this position
  null will be stored).
• `get(integer):T` for retrieving the reference with index i.
• `valid(integer):boolean` for testing whether i is an index of a reference (note:
  at this position also null could have been stored).
• `first()`:integer for retrieving the first index of a reference (-1, if no such
  index exists).
• `next(integer):integer` for retrieving the index following i (-1, if no such
  index exists).
• `size()`:integer for retrieving the number of references including null's stored
  in the list.
• `amount()`:integer for retrieving the number of non-null references stored
  in the list.

6.2.3 Input and Output Classes

Input Classes

The following classes are used to handle input to the GDP:

1. readable
   This class should be used as an ancestor for every class that needs to read
   information from a source (as is the case of most classes presented in this
   chapter). The class contains only one method: `read(in:istream):istream`, which should
   be implemented in each descendant (class istream is explained below). This is
   demonstrated for class point on page 96.

2. istream
   This class is the base-class for all classes that provide reading information from a
   source (descendants that deal with reading from a file or a string are introduced
   below"). The following methods are supported:

   • `open(s:stream):istream` for specifying that information should be read
     from the source with name s.
   • `close()`:void for specifying that no further information can be read from
     the source (opening the source is implemented in the various descendants
     of class istream).
   • `read(var x:boolean):istream`, `read(var x:integer):istream`, `read(var
     x:real):istream`, `read(var x:char):istream`, and `read(var x:string):
     istream` for reading basic objects. These functions return a reference to the
     istream object, thereby allowing for multiple reads in one expression.

4In the future new descendants that deal with reading from sockets may be introduced; this will
facilitate multiple GDP’s communicating with each other.
6.2. Fundamental classes

- "read(x:readable): istream" for reading an object of a class that has inherited from class readable. This method is implemented as follows:

```c
read(x:readable):istream
begin
  if x.eq(null).not() then x.read(self);
  result(self)
end;
```

As can be seen in this implementation the actual reading is carried out by the specialized method 'read(in:istream):void' of the readable object. It is not possible to use this method to read basic objects. For example, suppose class boolean would have inherited from class readable. Calling the above method with a boolean argument b would result in the execution of 'b.read(self)'. However, the read method is executed by the object that is referred to by object-reference b: this method can therefore never change object-reference b. Furthermore, this configuration would allow for using constants such as true as argument for method read. To prevent these problems, specialized read methods for the basis classes are used (true cannot be used as argument for these specialized methods, because it is not a reference-location — as required by the formal var-parameter).

- "eos(): boolean" for testing whether the end of the input source has been reached.

- "fail(): boolean" for testing whether the last read operation has failed.

3. file

This descendant from class istream provides reading information from a file. Class file redefines method 'open(s:string):istream': it specifies that information should be read from the file with name s. When using an empty string (i.e. "") as argument, the input is taken from the I/O channel between the GDP and an application (see Figure 4.1).

4. istring

This descendant from class istream provides reading information from a string. Class istring redefines method 'open(s:string):istream': it specifies that information should be read from string s.

The following code fragment shows how to use class istring:

```c
object is : istring; new(is); object i : integer;
is.open("123 abc");

is.read(i); // read an integer from string "123 abc"
// i = 123, is.fail() will return false,
// and is.eos() will return false

is.read(i); // try to read another integer from string "123 abc"
// i = null, is.fail() will return true,
// and is.eos() will return true
```
It is suggested that these input classes are always accompanied by the following object: `object in; ifstream new(in); in.open(" ");` in order to read from the I/O channel between the GDP and an application.

Output Classes

The following classes are used to handle the output from the GDP:

1. writeable
   This class should be used as an ancestor for every class that needs to write information to a destination (as is the case for most classes presented in this chapter). The class contains only one method: `write(out:ostream):ostream`, which should be implemented in each descendant.

2. ostream
   This class is the base-class for all classes that provide writing information to a destination (descendants that deal with writing to a file or a string are introduced below\(^2\)). The following methods are supported:

   - `open(x:string):ostream` for specifying that information should be written to the destination with name s.
   - `append(x:string):ostream` for specifying that information should be appended to the destination with name s.
   - `close():void` for specifying that no further information can be written to the destination (opening the destination is implemented in the various descendants of class ostream).
   - `write(x:writeable):ostream` for writing an object of a class that has inherited from class writeable. Similar to the read method of class istream this method is implemented by means of the specialized method `write(out:ostream):void` of the writeable object.

   ```
   write(x:writeable):ostream
   begin
     if (x=null)
       then write("null")
     else x.write(self);
     result(self)
   end;
   ```

   - `writeln():ostream` for writing a NEWLINE symbol.

\(^2\)In the future new descendants that deal with writing to sockets may be introduced.
6.2. Fundamental classes

(x:writeable) : ostream' that combine a write and a writeln method. For example, 'writeln(x:boolean):ostream' is implemented as follows:

```
writeln(x:boolean):ostream
begin
  result(write(x).writeln());
end;
```

- 'width(i:integer) : ostream', 'justification(left:boolean) : ostream', 'fill(c:char) : ostream', and 'precision(i:integer) : ostream' for initializing formatting information: width specifies the number of places that will be used for writing basic objects, justification specifies whether the basic objects will be written on the start or at the end of these places, fill specifies the character that will be used to fill the places that are not used by the basic objects themselves, and precision specifies how many digits of the fractional part of real objects will be written.

- 'width():integer', 'justification():boolean', 'fill():char', and 'precision():integer' for retrieving this formatting information.

3. ofstream

This descendant from class ostream provides writing information to a file. Class ofstream redefines the following methods:

- 'open (s:string) : ostream' for specifying that information should be written to the file with name s. When using an empty string (i.e. "") as argument, the information is written to the I/O channel between the GDI and an application (see Figure 4.1).

- 'append (s:string) : ostream' for specifying that information should be appended to the file with name s.

4. ostream

This descendant from class ostream provides writing information to a string. Class ostream supports the following additional methods:

- 'open() : ostream' for specifying that information should be written to an ostream whose initial string is empty.

- 'append(s : string) : ostream' for specifying that information should be appended to an ostream with initial string s.

- 'str():string' for retrieving the (internal) string of the ostream.

---

6 Strictly speaking writing information to a string is not possible, because strings are atomic values. Therefore, class ostream can be considered to contain a string attribute that is replaced with another string every time information is written to the ostream.
The following code fragment shows how to use classes ofstream and ostringstream:

```cpp
object of : ofstream; new(of);
of.append("filename"); // append to file "filename"
of.write("def"); // "def" is appended to the file

object os : ostringstream; new(os);
os.append("123 abc"); // append to string "123 abc"
osp.write("def"); // "def" is appended to string "123 abc"
// os.str() will return "123 abcdef"
```

It is suggested that these output classes will always be accompanied by the following object: `ofstream outofile; ofstream out; ofstream out.;' in order to write to the I/O channel between the GDP and an application.

### 6.2.4 Position and Vector Classes

The following classes deal with the specification of positions and directions in the GDP:

1. **point**

   This class defines positions in 3-D space. In order to facilitate reading and writing of point objects, this class has inherited from classes readable and writable, respectively. Class point contains the following methods.

   - `init(c0,c1,c2:real):void`
     This procedure initializes the coordinates of the point object.

   - `set(i:integer,p:real):void`
     This procedure sets the \( i^{th} \) \((0\leq i<3)\) coordinate of the point object to value \( p \) (if \( i \) is outside this range, nothing happens).

   - `get(i:integer):real`
     This function returns the \( i^{th} \) \((0\leq i<3)\) coordinate of the point object (if \( i \) is outside this range, \textbf{null} is returned).

   - `plus(v:vector):point`
     This function returns the result of adding vector \( v \) (see below) to the point object (note that the result is again a point object).

   - `eq(p:point):boolean`
     This function returns a \textbf{boolean} that indicates whether the coordinates of the point objects referred to by \textit{self} and \( p \) are piecewise identical.

   - `tvector():vector`
     This function returns the \textbf{vector} object that has the same coordinates as the \textbf{point} object.

2. **vector**

   This class defines vectors in 3-D space. For a similar reason as above it has inherited from classes readable and writable. Class vector contains the following methods.
6.2. Fundamental classes

- init(c0,c1,c2: real): void
  set(i: integer; p: real): void
  get(i: integer): real
  Similar to class point.

- norm(): real
  This function returns the norm — or length — of the vector object.

- normalize(r: real): vector
  This function returns the vector object that has length 1 and has the same
direction as self.

- inprod(v: vector): real
  This function returns the inner product of the vector object (referred to by
self) and v.

- crossprod(v: vector): vector
  This function returns the cross product of the vector object (referred to by
self) and v.

- neg(): vector
  This function returns the vector object whose coordinates are the negative
coordinates of self.

- times(r: real): vector
  This function returns the vector object whose coordinates are the coordi-
nates of self multiplied by r.

- plus(v: vector): vector
  minus(v: vector): vector
  These functions return the addition respectively subtraction of vector ob-
jects referred to by self and v.

- equal(v: vector): boolean
  This function returns a boolean that indicates whether the coordinates of
the vector objects referred to by self and v are piecewise identical.

- topoint(): point
  This function returns the point object that has the same
coordinates as the vector object.

3. axis

This class defines an axis, consisting of a point and a direction, in 3-D space.
It has inherited from classes readable and writable. Furthermore, it contains the
following methods:

- init(p: point; v: vector): void
  This procedure initializes the axis object: it will contain a reference to a copy of the point object referred to by p, and a
reference to the vector object that is obtained by normalizing a copy of the
object referred to by v to length 1.

- position(): point
  This function returns a reference to the point object that is
referred to by the axis object.

- direction(): vector
  This function returns a reference to the vector object that
is referred to by the axis object.
These classes have all inherited from readable and writable. For example, method `readable:read(in:istream) : void` of class point is implemented as follows:

```plaintext
readable:read(in:istream) : void
    var r,s,t : real;
    begin
        in.read(r).read(s).read(t);
        init(r,s,t);
    end;
```

Consequently, a point object may be initialized in the following way:

```plaintext
object in : iFile; object p : point;
new(in); new(p);
in.open("geometry");
in.read(p);
```

6.3 Collection

6.3.1 Description

For some applications it is necessary to have access to all objects of a certain type: to perform a method of or to inspect their attributes. This is especially true in context of the DenK project (Section 1.1). For example, in a domain of geometrical shapes the sentences "Are all cubes blue?" and "Does a blue cube exist?" are instances of universal and existential questions, respectively.

The mechanism to perform a method of all objects of a certain type comes in two forms in the GDP:

- **explicit**: the programmer asks to perform such a method
- **implicit**: the GDP automatically performs such a method (this is related to the concept of localization; examples of such methods are event passing, rendering, and dynamics).

A mechanism to program explicit universal operations\(^7\) is not yet available in most object-oriented programming languages. In those languages the programmer may have to create dedicated data-structures that contain information on the existing objects of a type. However, since a programming language should ease the task of a programmer, this approach is not very adequate.

Iterating over all objects of a given class could have been supported in Looks by means of a special language construct, say a for-statement. The syntax of such a statement could be given by means of the following grammar rule:

```plaintext
<statement> ::= 'FOR' <identifier> 'V' <type> 'DO' <statement>
```

The semantics of such a for-statement would be rather straightforward. For example, consider for-statement `for c : cube do S`. This statement has the effect that for

\(^7\)Note that existential operations can be implemented by means of universal operations.
6.3. Collection

each cube statement $S$ will be performed — to be able to refer to a cube, statement $S$
may contain object-reference $c$.
However, one of the design goals of Looks was to keep the language as simple as
possible. Therefore, the number of language constructs should be as small as possible.
Since the universal and existential operations may also be implemented by means of
a generic collection class, there is no need to include a for-statement in Looks. This
collection class collection[T] supports the following methods:

- **collect():void;**
  This procedure stores object-references to all objects whose type conforms to the
  actual generic type of the collection\(^8\) (note that objects that are created later
  are not added automatically to the collection).

- **size():integer;**
  This function returns the number of object-references that are stored in the
  collection.

- **index(i:integer):boolean**
  This function returns whether $0 \leq i < \text{size()}$.

- **get(i:integer):T;**
  This function returns a copy of the $i^{th}$ object-reference that is stored in the
  collection (or null, if $i < 0 \lor i \geq \text{size()}$)

- **reset():void;**
  This procedure clears the collection.

Example 6.1 shows how the collection class may be used to determine whether a blue
cube exists.

**Example 6.1: Existential Operation**

Suppose classes color and cube are defined previously, and that these classes
contain methods 'eq(c:color):boolean' and 'color():color', respectively.
Furthermore, suppose reference-location blue of type color has been defined
previously.

```
object blue.exists : boolean;
object cc : collection[cube];
object i : integer;
i:=0; blue.exists:=false; new(cc); // initialization
cc.collect(); // $cc$ now contains references to all existing cubes
while i.1s(cc.size()) && (blue.exists.not()) do
begin
  blue.exists:=cc.get(i).color().eq(blue);
i:=i+1;
end;
// boolean blue.exists indicates whether a blue cube exists
```

\(^8\)If the actual generic type is real, integer, or string, no references are stored in the collection.
6.3.2 Implementation

Obviously the implementation of method collect of class collection needs to have access to the symbol-table in order to get references to all objects of a certain type.

The most simple way to implement this method is to iterate over all objects while testing if their type conforms to the actual generic type of the collection (this iteration is very similar to the sweep-phase of mark-and-sweep garbage collection, see Section 5.3). This solution is shown in example 6.2.

Example 6.2: Collecting Objects (simple)

```java
void Collect() {
    "for every object obj"
    "if obj's type conforms to the actual generic type of collection"
    "store a reference to obj in collection"
}
```

However, a drawback of this solution is its complexity: it is proportional to the number of all objects. This can be improved if the information in the symbol-table is changed in the following way: every class in the class-table contains references to all its objects (note that this information can also be used in the sweep-phase of the garbage collection, as is shown in Example 6.4). As can be seen in Example 6.3, the complexity of collecting objects in this case is proportional to the number of classes and the number of objects whose type conforms to the actual generic type of the collection.

Example 6.3: Collecting Objects (ideal)

```java
void Collect() {
    "for every class C in the class-table"
    "if C conforms to the actual generic type of collection"
    "for every object obj of C"
    "store a reference to obj in collection"
}
```

Example 6.4: New Sweep-phase()

```java
void Sweep-phase() {
    "for every class C in the class-table"
    "for every object obj of C"
    "if obj is marked"
    "unmark obj"
    "else"
    "deallocate obj"
}
```

The type-conformance as used in Example 6.3 is not correct for generic classes, e.g. array[T] - the generic class in the class-table — does not conform to array[cube] — the actual generic type of a collection. For this purpose "pseudo-conformance" should be used: conformance without paying attention to actual generic types. However, this introduces incorrect results for collections of generic types. For example, consider class array[T]. References to all instances of this class (e.g. objects of type array[integer], array[boolean], array[cube], etc.) are stored together at class array[T]. Obviously, this will cause problems when collecting arrays of a certain type (e.g. collection[array[cube]] is only interested in objects of type array[cube]). Two alternatives can be devised to solve this problem:
6.4 Events

6.4.1 Description

As presented in Chapter 3 there are two ways to invoke a method of an object: synchronous message and asynchronous message. Both messages require that the sender knows the object to whom the message should be sent. Although this might be applicable in many situations, there are situations in which this condition does not hold, i.e., it is unknown to which object(s) a message should be sent. For this kind of situations the GDP supports an event-mechanism, which provides for a decoupling between sender and receiver of messages. In the event-driven approach an object may indicate that it is interested in certain types of events. Furthermore, it will provide a method (namely an event-handler) that will be invoked every time such events occur. Now the 'sender' object may introduce an event, and all objects interested in this event will automatically execute their event-handler method (note that the 'sender' object indirectly sends a message to the interested objects). Furthermore, these 'sender' objects are normal Looks objects, in other words: events may be 'signalled' in Looks. However, Looks events may also be signalled by the run-time system of the GDP, e.g., events that occur in the windowing system will result in raising Looks events of type UIEvent (explained below).

Events are also modelled as objects. They have one common base class: event. This class supports one method 'signal(void)', which may be used to introduce an event. Objects that are interested in events must be instances of classes that have inherited from class responder. This class supports the following methods:

1. Every generic class in the class-table contains a list with all actual generic types that are used with this class. Furthermore, each entry in this list contains references to all objects that have the corresponding actual generic types.

2. The ideal solution from Example 6.3 is modified by adding an extra conformance-check before a reference to an object is stored in a collection.

A drawback of the first alternative is that allocations (by means of new) become more complex because a reference to each new object has to be stored in the appropriate list. A drawback of the second alternative is that collections become more complex because all objects whose type "pseudo-conforms" to the actual generic type of the collection, have to be checked by means of ordinary type-conformance whether they should become members of the collection (which is slower than the ideal case, but faster than the simple case). Because collecting objects is rare compared to the allocation of objects, the second alternative is chosen in the GDP. Example 6.5 shows the implementation as used in the GDP.

Example 6.5: Collecting Objects (final)

```java
void Collect() {
    "For every class C in the class-table",
    if "C pseudo-conforms to the actual generic type of collection"
    "For every object obj of C"
    if "obj type conforms to the actual generic type of collection"
        "store a reference to obj in collection"
}
```
• `eventhandler(e: event):void` for handling event e, i.e. events that are signalled are passed as argument to interested objects (i.e. responders).

• `watch(e: event):void` for specifying that the responder is interested in events of e's type (note that this type will be a descendant of class event).

• `forget(e: event):void` for specifying that the responder is no longer interested in events of e's type.

• `forget_all():void` for specifying that the responder lost interest in all event.

One type of events is special: events that are generated by the user by means of mouse or keyboard. For this kind of events class UIEvent (a descendant of class event) is supported. Furthermore, UIEvents typically occur in widgets (i.e. elements of a graphical user interface – class widget is introduced in Section 6.5). Responders do not have to be interested in UIEvents of all widgets, but they may be interested only in UIEvents of several special widgets. For example, a camera (see Section 6.6.4) is typically only interested in UIEvents that are generated in its associated canvas_widget. For specifying these special widgets class UIEvent has inherited also from list[widget]9. Class UIEvent supports the following methods:

• `allwidgets(all: boolean): void` for specifying whether it is not important in which widget the event has occurred (this overrules the list of widgets described above)10.

• `widget(): widget` for retrieving the widget in which the UIEvent has occurred.

• `widget(w: widget): void` for setting the widget in which the UIEvent has occurred (this provides for simulation of key or mouse events — indeed, a UIEvent may be created and initialized in a L00ks program and it may subsequently be signalled).

• `time()::integer` for retrieving the time at which the UIEvent occurred.

• `dt()::integer` for retrieving the time since the last UIEvent.

The following derived classes of UIEvent are supported:

1. `key_event` is the class that models events that are generated by the keyboard. For specifying the condition when a key_event should be generated, the following methods are supported:

   • `Normal(pressed, released: boolean): void` for specifying that an event should be generated for normal keys that are pressed or released (according to the value of pressed and released, by default both are false).

   • `Ctrl(pressed, released: boolean): void` for specifying that an event should be generated for Ctrl-keys that are pressed or released.

9It would be more natural to have the list of widgets as an attribute of class UIEvent, but rather low-level implementation details of the event-handling mechanism of the GDP have lead to the approach that uses inheritance.

10This option is useful for recording all mouse and keyboard events.
6.4. Events

- `Shift(pressed, released: boolean): void` for specifying that an event should be generated for Shift-keys that are pressed or released.
- `Meta(pressed, released: boolean): void` for specifying that an event should be generated for Meta-keys that are pressed or released.

The Normal, Ctrl, Shift, and Meta condition are combined by means of a logical or. For example, `Normal(true, true); Shift(true, false);` specifies that a key_event should be generated when a normal key is pressed or released, or when a shift-key is pressed (but not when it is released). If subsequently statement `Normal(true, false);` is executed, key_events are only generated when a normal key or a shift-key is pressed.

For creating a key_event in Looks, the following methods are supported:

- `Normal_down(down: boolean): void`, `Ctrl_down(down: boolean): void`, `Shift_down(down: boolean): void`, and `Meta_down(down: boolean): void` for specifying the state of the keyboard (default value is false). For example, `Normal_down(true), Shift_down(true);` specifies that a Shift-key and a Normal key are down at the same time.
- `key(c: char): void` for specifying that character c is (de)pressed (only useful for `Normal_event()`; default is character `'`).
- `Normal_event(): void`, `Ctrl_event(): void`, `Shift_event(): void`, and `Meta_event(): void` for specifying which key caused the event (default is a Normal key). These procedures overrule each other. For example, `Normal_event(); Shift_event();` specifies that the Shift-key caused the event (Shift_down indicates whether the event was caused by pressing or releasing the Shift-key).

For retrieving information from the key_event, the following methods are supported:

- `Normal_down(): boolean`, `Ctrl_down(): boolean`, `Shift_down(): boolean`, and `Meta_down(): boolean` for retrieving the state of the various keys (true means that they are down).
- `key(): char` for retrieving the (normal) key that is (de)pressed (this function returns null, if function Normal returns false).
- `Normal(): boolean`, `Ctrl(): boolean`, `Shift(): boolean`, and `Meta(): boolean` for retrieving which key caused the event (exactly one of these functions will return true). The other retrieval functions may be used to inspect the state of the various keys.

2. mouse_event is the class that models events that are generated by the mouse. For specifying the condition when a mouse_event should be generated, the following methods are supported:

- `button(i: integer, pressed, released: boolean): void` for specifying that an event should be generated whenever button i is pressed or released (0≤i<3; by default both pressed and released are false).
• `motion(boolean):void` for specifying whether an event should be generated whenever the mouse moves (by default false).

For creating a mouse_event in Leks, the following methods are supported:

• `mouse(mouse):void` for setting the state of the mouse at the time the event was signalled.
• `dx(integer):void` and `dy(integer):void` for retrieving the displacement of the mouse since the last signalled mouse_event.
• `button_event(integer):void` and `motion_event():void` for specifying what caused the mouse_event: pressing/releasing a button, or moving the mouse.

For retrieving information from the mouse_event, the following methods are supported:

• `mouse():mouse` for retrieving the state of the mouse at the time the event was signalled.
• `dx():integer` and `dy():integer` for retrieving the displacement of the mouse since the last signalled mouse_event.
• `motion():boolean` for retrieving whether the mouse_event was caused by moving the mouse.
• `getbutton():integer` for retrieving the button that caused the event (this function returns -1, if the mouse_event was caused by moving the mouse).

The following example shows how to create objects that are interested in mouse movements.

**Example 6.6 : Dealing with events**

```plaintext
class mouse_follower
  inherit responder()
public
  init () : void
  implementation
  init () : void
    var mouse : mouse_event
    begin
      new(mouse); // by default, not interested in any event
      mouse.motion(true); // only interested in motion events
      mouse.alldown(true); // from any widget
      responder.watch(mouse);
      // mouse_follower only interested in motion events on any widget
    end;
```

11 Class mouse is used to set and retrieve information on the state of the mouse. It supports the following methods:

- `button_down(integer, down: boolean):void` for specifying the state of button i (0≤i<3, default is false).
- `button_down(integer):boolean` for retrieving whether button i is down.
- `x():integer` and `y():integer` for specifying the position of the mouse.
- `x():integer` and `y():integer` for retrieving the position of the mouse.
- `refresh():void` for updating the mouse object according to the physical mouse, i.e. this allows for polling the physical mouse.
6.4. Events

```hs
responder: eventhandler(e: event): void
  var me: mouse.
  begin
    me ?= e;  // try if e is a mouse.
    // mouse.follower is only interested in mouse.events.
    // (see method init above), that this checked assignment
    // will always succeed, consequently me is not null.
    ...
    // use me.dx() and me.dy() to retrieve the mouse displacement
    ...
  end
endclass;
```

object m0, m1: mouse.follower;
new(m0); m0.init();  // m0 is interested in mouse movements
new(m1); m1.init();  // m1 is interested in mouse movements
// whenever the user moves the mouse, the evthandler method of m0 and m1 will be executed
object m: mouse; new(m); m.x(100); m.y(100);
object me: mouse.event; newme(m);
me.mouse(a);  // create an event because the mouse has moved to (100, 100)
me.motion.event();
me.signal();  // and let this event occur
// the evthandler method of m0 and m1 will now be executed
```

6.4.2 Implementation

The event handling mechanism of Looks is carried out immediately after all asynchronous messages have been evaluated. This is shown in the next algorithm, which represents the main loop of the GDP (the algorithm is a simplification of the algorithm presented on page 66.

```hs
do true
  "process asynchronous messages"
  "event handling";
  "garbage collection";
  "rendering";
  "parsing";
  od;
```

This event handling mechanism is carried out automatically, i.e. it is not programmed explicitly in Looks. Therefore, events can be considered as being local in Looks: they do not have to know which responders are interested in them.

For the event handling mechanism the following datastructures are used:

1. eventlist : event* (a list of all events that are signalled). This datastructure is filled by physical mouse or keyboard events and by the method signal.

2. responders: event → responder* (a function that returns the list of interested responders for every event type). This datastructure is manipulated by methods watch, forget, and forget.all of class responder.

3. widgets: UI.event × responder → widget* (a function that returns the list of widgets from which the responder is interested to receive the UI.event). This datastructure is manipulated by methods add, del, and allwidgets of class UI.event.
Now the event handling mechanism can be described by means of the following algorithm (in pseudo-code):

\[
\text{do eventlist\#<>}
\quad\text{\textbf{for each} eventlist-e}
\quad\text{\textbf{if} "e is a UI\_event"
\quad\quad\text{\textbf{if} "e's widget in widgets(e,responder)"
\quad\quad\quad\text{"responder\_eventhandler(e)"
\quad\quad\quad\text{"e's widget not in widgets(e,responder)"
\quad\quad\quad\text{skip
\quad\quad\text{fi}
\quad\quad\text{fi}
\quad\quad\text{if} "e is not a UI\_event"
\quad\quad\quad\text{"responder\_eventhandler(e)"
\quad\quad\text{fi}
\quad\text{od}
\]

As can be seen in this algorithm, there is a difference between a UI\_event and other events: these last events always cause the execution of the eventhandler of interested responders, whereas the former only cause this execution when the responder is interested in receiving events from UI\_event's widget. The reason for this is that for an arbitrary non-UI\_event there is no obvious choice for associated widgets.

### 6.5 Graphical User-Interfaces

In recent years Graphical User-Interfaces (GUIs) have proven to be a promising way to control applications. Therefore, it is desirable that animations running in the GDP can also be controlled by means of a GUI. For this purpose Looks supports classes that are based on the Motif widget set [Hel91].

These widget classes (widgets) have one common ancestor: class widget. This class has methods for setting and retrieving properties that are common for all widgets:

- position of a widget
- size of a widget
- size of a border
- color of interior and border
- 'manage(b:boolean):void' for specifying that the widget is ready to take part in a GUI or that it does not take part in a GUI anymore.

The default behavior of the widgets with respect to mouse-events is specified by the Motif library. However, a Looks application may decide to overrule this default behavior. For this purpose class widget is a descendant of class responder: by implementing
a dedicated eventhandler method, a widget may overrule the default behavior.

Class widget has the following descendants:

1. simplewidget is the base class of all widgets that cannot have children\footnote{Widgets introduce a new type of hierarchy, which represents the 'container' relation between widgets. For this hierarchy the following terminology is used: widgets that are contained in widget \( w \) are called the children of \( w \), for these widgets their container \( w \) will be called their parent. Thus child and parent are used in context of widgets, whereas ancestor and descendant are used in context of inheritance.}. Methods include setting and retrieving visualization information (e.g. foreground color, shadow color, highlight color), and 'childof(c.compositewidget):void' for specifying the parent of the widget.

Class simplewidget has the following descendants:

- label for displaying a text or a pixmap. Methods include setting and retrieving text, font, pixmap, position, and formatting information.

  Class label has the following descendants:

  - pushbutton for executing a method, known as a callback method, whenever the button is activated\footnote{Note that such a callback method could also have been implemented by means of the eventhandler of the pushbutton. However, this would require some effort of the Looks programmer. Furthermore, invocation of a method is the natural behavior of a pushbutton. Consequently, most pushbuttons will have this behavior. Therefore, a dedicated callback method is available in order to minimize the programming effort.}. This callback method is one of the widget's methods: 'callback()::void'.

  - togglebutton for switching between two states (on/off). Methods include setting and retrieving the state, the color that indicates state on, and a callback method that is executed every time the button is activated.

  - cascadebutton for introducing a pulldown menu whenever the button is activated. Methods include initialization of the associated pulldown menu, and a callback method that is executed every time the button is activated. Due to the underlying Motif implementation, the parent of a cascadebutton has to be a rowcolumn widget (or one of its descendants). This would conflict with calling the public inherited method 'childof(c.compositewidget):void', therefore, for a cascadebutton this method is made private, while a new public method 'childof(r:rowcolumn):void' is introduced. In this way, the type checking of Looks is used to make sure that the parents of cascadebuttons are allowed widgets.

- separator for separating other widgets. Methods include setting and retrieving orientation and drawing style.

- list for selecting one or more entries from an array of textual items. Methods include adding, deleting, selecting, and deselecting items; setting a selection policy (e.g. one item or multiple items); retrieving information on selected items.

- text for displaying and editing text. Methods include setting and retrieving
2. composetwidget is the base class of all widgets that may have children. Class composetwidget has the following descendants:

- **window** is the only widget that does not have a parent. Methods include initialization of the window (a window may even be initialized on remote machines), closing the window, setting and retrieving the title of the window, raising the window (i.e. other windows will not overlap the window), and lowering the window (i.e. no other window will be overlapped by this window).

- **manager** is the base class for composetwidgets that must have parents. It contains method `childof(c:compositewidget) void` for specifying the parent of the widget and methods for formatting (e.g. foreground, highlight, shadow).

Class manager has the following descendants:

- **rowcolumn** is the base class of all menu classes. As such it acts like a container for the various buttons that may be used in a menu; in this approach buttons are children of a menu. Methods of class rowcolumn include formatting methods (for positioning the buttons within a menu) and a callback method.

Class rowcolumn has the following descendants:

- **popupmenu** is a menu that is normally not visible. It pops up whenever a dedicated event is generated in its parent widget. Class popupmenu does not have additional methods.

- **pulldownmenu** is also a menu that is normally not visible. It becomes visible when its associated cascadebutton in a menubar, popupmenu, or other pulldownmenu is activated. Class pulldownmenu does not have additional methods.

- **optionmenu** is a menu for choosing among a set of usually mutually exclusive options. It consists of a cascadebutton and an associated pulldownmenu. The cascadebutton contains the label of the currently selected option; when the button is activated the pulldownmenu appears and subsequently an option from this menu may be chosen. Class optionmenu supports methods to associate a pulldownmenu and to initialize the label of the cascadebutton.

- **menubar** is a menu that can only contain cascadebutton objects, which are placed, if possible, in a single row. Class menubar does not have additional methods.

- **scrolledarea** for displaying only a part of a child-widget, i.e. a scrolledarea object acts like a viewport onto its child-widget. Furthermore, a scrolledarea object provides scrollbars for moving the viewport. The placement of these scrollbars can be specified by methods of the object.

- **canvas** for displaying computer generated images. The class supports various 2-D drawing methods for generating such images.
— scale for displaying and changing a value within a range by means of a slider. The class supports methods for setting and retrieving the range, precision, value, formatting (orientation, width, height, title), and a callback. This callback method is executed whenever the value changes, either by moving the slider or by setting the value by means of a method.

— bulletinboard is the base class for dialog widgets, i.e. widgets that provide a way for providing information or asking a question. It supports methods for setting and retrieving margins.

Class bulletinboard has the following — specialized — descendants:

* messagebox for showing a text. This text has to be acknowledged by activating one of the three associated buttons of a messagebox: "OK", "Cancel", or "Help". Class messagebox has methods for formatting and for setting its text.

Class messagebox has the following descendants (they only differ in representation):

  - errorbox
  - informationbox
  - questionbox
  - templatebox
  - warningbox
  - workingbox

* selectionbox for selecting a choice from a list. It automatically creates the following child-widgets: list, text for displaying and editing the choice, and three buttons ("OK", "Cancel", or "Help"). Class selectionbox supports methods for retrieving these child-widgets and for setting and retrieving the choice. There is one specialized selectionbox:

  - fileselectionbox for selecting a file from a directory. It creates the following child-widgets: 2 lists (one for filenames, and one for subdirectories), 2 texts (one for a file search patterns, and one for the selected filename), and three buttons (see above). Class fileselectionbox also supports methods for setting and retrieving filename, (sub-)directory name, and file search pattern.

All widgets, except window, need to have a parent. Consequently, applications that support a GUI will have at least one window.

6.6 Rendering

Being an animation system, the GDP of course supports classes to visualize (or: render) 3-D objects. An important issue in the design of the rendering classes of the GDP is the distinction between:
• **geometry**: a set of 3-D points

• **topology**: a set of topological primitives, such as lines, polygons, etc.

The separation of geometrical and topological information is made because it allows for an intuitive mechanism to display objects in various ways. For example, the position of a cube may be specified by means of one geometry consisting of 8 points, whereas its appearance may be specified by means of a topology consisting of 12 edges, a topology consisting of 6 polygons, a topology consisting of 4 edges and 2 polygons, etc.

With respect to colors, the GDP supports the RGB color model, i.e. every color consists of three components: red, green, and blue. The values of these components are limited to the range [0,1]. These colors are modeled as objects of class color, a descendant from readable and writable, that has the following methods:

• 'init(r,g,b:real):void' for initializing the red, green, and blue components simultaneously (arguments greater than 1 or less than 0 are truncated to 1 or 0, respectively)

• 'r():real', 'g():real', and 'b():real' for setting the red, green, and blue components individually

• 'r():real', 'g():real', and 'b():real' for retrieving the red, green, or blue component, respectively

• 'eq(c:color):boolean' for testing whether two colors have the same components

Furthermore, the following reference-locations of type color are pre-defined: black (refers to a color object with rgb-components: (0,0,0)), red (rgb: (1,0,0)), green (rgb: (0,1,0)), blue (rgb: (0,0,1)), yellow (rgb: (1,1,0)), cyan (rgb: (0,1,1)), magenta (rgb: (1,0,1)), and white (rgb: (1,1,1)).

Classes for the specification of a geometry and topology are presented in Sections 6.6.1 and 6.6.2, respectively. Classes to illuminate a 3-D scene are introduced in Section 6.6.3. Finally, a class that implements a virtual camera is described in Section 6.6.4. Again the concept of localization is important in the design of these classes: the Looks programmer should not be bothered by the actual rendering process, instead this should be carried out automatically by the GDP. For example, cameras will automatically render all visible topological primitives. Therefore, the Looks programmer only has to specify the motion of geometries; he does not have to deal with rendering the associated topological primitives in the various cameras.

### 6.6.1 Geometry

A **geometry** object models a set of 3-D points. These points can be stored in two ways:

1. **world coordinates**: the 3-D points are given in a fixed basis, say W.

2. **local coordinates**: the 3-D points are given in a local basis, say L; furthermore, there is a transformation, say \( M_WL \), that transforms the coordinates of a point given in basis L to coordinates in basis W.
The relation between the world coordinates and local coordinates of a geometry is shown in Figure 6.7. In this figure the following names are used: \( p_0 \) denotes point \( p \) in basis \( L \), and \( p_1 \) denotes point \( p \) in basis \( W \). The relations between \( L \) and \( W \), on the one hand, and \( p_0 \) and \( p_1 \), on the other hand, is given by means of the following equations:

\[
L = M_{WL} \cdot W \\
p_1 = M_{WL} \cdot p_0
\]

![Diagram](image)

Figure 6.7: Relation between world coordinates and local coordinates of a geometry

For rigid bodies, i.e. objects whose local 3-D points remain fixed with respect to a body-fixed frame, the approach using local coordinates seems to be ideal: moving objects only consists of changing their \( L \) basis (and updating their \( M_{WL} \) accordingly). Therefore, this approach has been taken in the GDP.

Furthermore, specialized geometries (which will be introduced in Section 6.8) may form a tree (the root of this tree is considered to be basis \( W \)). In this situation there are transformations, say \( M_{FL} \), that transform the coordinates of a point in basis \( L \) to the coordinates in the basis of its father in the geometry-tree (note that for geometries at depth \( 1 \) \( M_{FL}=M_{WL} \)). Hence, the (world) coordinates of a point \( p \) given in basis \( L_0 \) at depth \( n \) can be calculated as follows:

\[
M_{FL_1} \cdots M_{FL_0} \cdot p \quad (\text{with } L_0 \text{ the father of } L_{n-1})
\]

However, for efficiency reasons the product \( M_{FL_1} \cdots M_{FL_0} \) is cached in a matrix \( M_{WL_0} \). This relation is specified by means of the following invariant:

**WL-invariant:** \( M_{WL_0} = M_{WL_1} \cdot M_{FL_0} \)

Note that the WL-invariant is not required for obtaining local coordinates or father coordinates of a point (in these cases it may even be violated). However, it should hold whenever world coordinates of a point are required (this is especially true for rendering, therefore the WL-invariants of all geometries are restored before rendering). For this purpose, the main loop is modified as follows:
do true
    → "process asynchronous messages"
    "event handling"
    "restore WL-invariants"
    "garbage collection"
    "rendering"
    "parsing"
    od;

The automatical restoring of the WL-invariants can be described by means of the following repetition (in pseudo-code):

    for each geometry g
        → g.restore.WL-invariant(); // this procedure is defined on page 132

Figure 6.8 shows a geometry-tree with depth 2, and the relations between the three bases.

![Diagram](image)

**Figure 6.8: Relation between hierarchical geometries**

For dealing with basis L and transformation $M_{WL}$ a geometry object supports the following methods:

- 'basis(o:point;xdir,ydir,zdir vector):void' for initializing L and $M_{WL}$ (xdir, ydir, and zdir should be orthogonal and right-handed)

- 'origin(tobasis:geometry):point' for retrieving the origin of basis L in local coordinates of geometry tobasis:

  $$(\text{tobasis}.M_{WL}^{-1})(\text{self}.M_{WL}^{-1})(0,0,0,1)$$

---

14 When transforming points, the translation of the transformation-matrix also has to be taken into account — this is specified by the fourth coordinate.
6.6. Rendering

(The returned point only uses the first three coordinates of the result of this expression.)

There are two special cases for this expression:

- \( \text{tobasis}=\text{self} \): the origin of \( L \) is returned in local coordinates, hence \((0,0,0)\) is returned
- \( \text{tobasis} \cdot M_W = L \): the origin of \( L \) is returned in world coordinates

- 'direction(integer; tobasis:geometry):vector' \((0\leq i<3)\) for retrieving the \( i \)th basis vector of \( L \) in local coordinates of geometry tobasis:

\[
(\text{tobasis} \cdot M_W^{-1}) \cdot (\text{self} \cdot M_W) \cdot (x_i, y_i, z_i, 0)^T
\]

\[(x_0=1, y_0=0, z_0=0; x_1=0, y_1=1, z_1=0; x_2=0, y_2=0, z_2=1)\]

Obviously, this expression has the same two special cases.

- 'axis(integer; tobasis:geometry):axis' \((0\leq i<3)\) for retrieving the \( i \)th axis of basis \( L \) (this axis consists of the origin of \( L \), i.e. a point, and the \( i \)th basis vector of \( L \)).

Naturally, a geometry object also supports methods for adding and deleting points. For this purpose class geometry is a descendant of type list[point], where the inherited methods are renamed as follows: \( \text{add}>\text{addpoint}, \text{get}>\text{getpoint}, \text{valid}>\text{validpoint}, \text{first}>\text{firstpoint}, \text{next}>\text{nextpoint}, \text{and amount}>\text{amountpoint} \). Furthermore, class geometry supports the following methods:

- 'tolocal(point; frombasis:geometry):point' returns the following point:

\[
(\text{self} \cdot M_W^{-1}) \cdot (\text{frombasis} \cdot M_W) \cdot (p.x, p.y, p.z, 1)^T
\]

In case 'frombasis=\text{self}' a copy of point p is returned. Note that argument p is not necessarily a member of the geometry.

- 'getpoint(integer; tobasis:geometry):point' for retrieving a copy of the point with index i:

\[
(\text{tobasis} \cdot M_W^{-1}) \cdot (\text{self} \cdot M_W) \cdot (p.x, p.y, p.z, 1)^T,
\]

with \( p \) : the \( i \)th point, which is represented in local coordinates.

Note that there is a difference between functions get and getpoint: the former returns a reference to a point of a geometry (therefore this point will always be in local coordinates), whereas the latter returns a copy of such a point.

Sometimes it may be necessary to access designated points of a geometry, e.g. the center of a cube may be such a special point, or the four points at the top of the cube, etc. A geometry object supports named subsets to store indices of points (subsets do not have to be disjoint). In this way the geometry object takes care of the book-keeping of these designated points. For this purpose class geometry is a descendant of type list[subset]\(^{16}\), where the inherited methods are renamed as follows: \( \text{add}>\text{addsubset}, \text{get}>\text{getsubset}, \text{valid}>\text{validsubset}, \text{first}>\text{firstsubset}, \text{next}>\text{nextsubset}, \text{and amount}>\text{amountsubset} \). Furthermore, class geometry has the following methods for dealing with subsets:

\(^{15}\)When transforming vectors, the translation of the transformation matrix may not be taken into account.

\(^{16}\)Similar as in the situation for class UI: event, it would have been more natural to have the list of subsets as an attribute of class geometry. But again low-level implementation details have lead to the approach that uses inheritance.
• 'getindex(s:string):integer' for retrieving the index of the subset with name s (-1, if no such subset exists).

Class subset is a descendant of list[integer] and has the following additional methods:

• 'name(s:string):void' and 'name():string' for setting and retrieving the name of the subset.

• 'member(s:pi:integer):boolean' for testing whether point index pi is stored in subset si.

Example 6.9 shows how these methods may be used to create a geometry of a cube with two subsets: one subset containing the points of the cube's top and one subset containing the points of the cube's front.

Example 6.9: Definition of a cube

| object g : geometry; new(g); g.subsets(); |
| object s : point; new(s); s.xinit(0,0,0); |
| object x : vector; new(x); x.xinit(1,0,0); |
| object y : vector; new(y); y.xinit(0,1,0); |
| object z : vector; new(z); z.xinit(0,0,1); |
| object s : subset; |
| object p : point; g.rename(x,y,z); |

// the ML transformation of geometry g is initialized with the standard basis
new(p); p.xinit(-1,-1,-1); g.addpoint(p);
new(p); p.xinit(-1,1,-1); g.addpoint(p);
new(p); p.xinit(1,1,-1); g.addpoint(p);
new(p); p.xinit(-1,-1,1); g.addpoint(p);
new(p); p.xinit(-1,1,1); g.addpoint(p);
new(p); p.xinit(1,1,1); g.addpoint(p);
new(p); p.xinit(-1,1,1); g.addpoint(p);
new(p); p.xinit(-1,-1,1); g.addpoint(p);
new(p); p.xinit(-1,1,1); g.addpoint(p);
new(p); p.xinit(-1,-1,1); g.addpoint(p);

// Note that we make use of the fact that no other points are stored in g, so therefore we
// infer the indices of the points p from the order in which they are added.
// A safer implementation would store the returned indices in an array
new(s);
s.name("top");
s.add(4); // add point 4, i.e. (-1,-1,1), to subset "top"
s.add(5); // add point 5, i.e. (1,-1,1), to subset "top"
s.add(6); // add point 6, i.e. (-1,1,1), to subset "top"
s.add(7); // add point 7, i.e. (1,1,1), to subset "top"
g.addsubset(s); // add subset s to geometry g

// another way to fill a subset is shown below
new(s);
s.name("front");
g.addsubset(s); // add (empty) subset s to geometry g (it will get index 1)
g.getsubset(1).add(0); // add point 0, i.e. (-1,-1,-1), to subset 1, i.e. "front"
g.getsubset(1).add(1); // add point 1, i.e. (1,-1,-1), to subset 1, i.e. "front"
g.getsubset(1).add(4); // add point 4, i.e. (-1,-1,-1), to subset 1, i.e. "front"
g.getsubset(1).add(5); // add point 5, i.e. (1,-1,-1), to subset 1, i.e. "front"

As can be seen in the previous example, initializing a geometry can be rather involved. Obviously, it becomes even worse for complex geometries. In order to prevent this kind of program fragments, geometries may be initialized using 3-D object files. Furthermore, there are tools available to convert 3-D files from various formats to the GDP 3-D object file format. Consequently, a variety of modelling systems can be used to generate 3-D models that are to be animated by the GDP.

The following methods are supported by a geometry object to deal with 3-D object files:
6.6. Rendering

- `name(s: string): void` and `name(): string` for initializing and retrieving the name of the geometry. This name is used for writing the contents of the geometry to a file.

- `read3Dfile(s: string, var a: array[topology]): void` for initializing the geometry using the 3-D object file with name `s`. This method also initializes topologies that are based on points of this geometry (class topology is explained in the next section).

- `write3Dfile(s: string, a: array[topology]): void` for saving the contents of the geometry (and of the topologies of array `a`) in the file with name `s`.

Example 6.10 contains the contents of a 3-D object file. It starts with a heading that indicates that the geometry contains two subsets, i.e., "top" and "front". The two lines after the "#" specify basis `L` of the geometry. The lines thereafter specify the points of the geometry. The two lines that start with `S` specify the subsets of the geometry. Finally, the lines after `&` specify the topological information.

Example 6.10: File "cube.gdp"

```plaintext
cube(top,front!)<topo>
   *
   0 0 0
   1 0 0 0 1 0 0 0 1
   # -1 -1 -1
   # 1 -1 -1
   # -1 1 -1
   # -1 -1 -1
   # 1 -1 -1
   # 1 1 1
   # -1 1 1
   # 4 5 0 7
   # 0 1 4 6
   #
   FLUTTER
   # 0 1 2 3
   # 0 1 5 4
   # 1 2 0 5
   # 2 3 7 6
   # 3 0 4 7
   # 4 5 0 7
   ^
```

Using file "cube.gdp" from Example 6.10, the initialization of a cube becomes very compact. This is shown in Example 6.11.

Example 6.11: Definition of a cube using file "cube.gdp"

```plaintext
object g : geometry, new(g);
object a : array[topology]; // class topology is explained in the next section
g.read3Dfile("cube.gdp", a); // geometry g is completely initialized
```

Finally, class geometry contains methods 'selectable(s : selectable) : void' and 'selectable() : selectable'. These methods are used for direct manipulation purposes and are explained in more detail in Section 6.7.
6.6.2 Topology and primitives

A topology object models a set of topological primitives, i.e. the basic rendering elements of the GDP. These primitives are defined by means of references to points of geometries, and provide a way to visualize geometries. Topological primitives have a common base class: primitive, which supports the following methods:

- 'attribute(a:attribute):void' for setting the rendering attributes of the primitive (class attribute is presented in the next section).
- 'attribute():attribute' for retrieving these rendering attributes.

The following classes of topological primitives have inherited from class primitive:

1. marker
   This topological primitive visualizes one point of a geometry, which may be useful for visualizing the selection of points of a geometry. Class marker adds the following methods:
   - 'init(g:geometry; i:integer):void' for specifying that point i of geometry g has to be visualized.
   - 'point(tobasis:geometry):point' for retrieving a copy of this point, expressed in coordinates with respect to the local coordinate system of tobasis.

2. line
   This topological primitive visualizes a line between two points, which do not necessarily have to be elements of the same geometry. Class line adds the following methods:
   - 'init(fg:geometry; fp:integer; tg:geometry; tp:integer):void' for specifying the begin-point and end-point of the line.
   - 'beginpoint(tobasis:geometry):point' and 'endpoint(tobasis:geometry):point' for retrieving the begin-point or the end-point of the line, respectively, in coordinates with respect to the local coordinate system of tobasis.

3. variable.primitive
   This class is a base class for all topological primitives that have a variable amount of point-references (in contrast to a marker or line that always have one or two point-references, respectively). For manipulating with point-references class variable.primitive has inherited from list(point), i.e. class variable.primitive is a case of multiple inheritance. Furthermore, class variable.primitive has the following additional method:
   - 'getpoint(i:integer; tobasis:geometry):point' for retrieving a copy of the point with index i.

The following topological primitives have inherited from variable.primitive, therefore they may contain a variable amount of point-references:

(a) polyline
   This topological primitive visualizes a line between each pair of points given by consecutive valid indices.
6.6. Rendering

(b) polygon

This topological primitive visualizes a plane surface that is outlined by the points that are referred to by the indices contained in the primitive (if the points are not located in a plane, the visualization is undefined).

It should be noted that the actual position of points is controlled by geometries; topological primitives only contain references to these points. Therefore, changing the position or orientation of a geometry has the effect that all primitives that refer to points of this geometry are visualized using this new position or orientation automatically. This is shown in Example 6.12, where changing the position of a geometry results in an automatic update of the end-point of a line.

Example 6.12: Automatic updating of primitives

```plaintext
object g0 : geometry; new(g0);
object g1 : geometry; new(g1);
object o0 : point; new(o0); o0.init(0,0,0);
object o1 : point; new(o1); o1.init(0,0,0);
object x : vector; new(x); x.init(0,0,1);
object y : vector; new(y); y.init(0,1,0);
object z : vector; new(z); z.init(0,0,1);
object l : line; new(l);
object p : point;
g0.basis(o0,x,y,z);
g1.basis(o1,x,y,z);
new(p); p.init(26,26,26); g0.addpoint(p);
new(p); p.init(50,50,50); g1.addpoint(p);
l.init(g0,g1,0); // represents a line between (26,26,26) and (50,50,50)
o1.init(50,50,50);
g1.basis(o1,x,y,z); // place geometry g1 at point (50,50,50)
// l is updated automatically to represent a line between (26,26,26) and (100,100,100)
```

For storing such primitives class topology is a descendant of type list[primitive]. Further more, class topology supports the following methods:

- 'attribute(a:attribute):void' and 'attribute():attribute' for setting and retrieving the (default) rendering attributes of the primitives stored in the topology (in case the rendering attributes of a primitive are not specified, the attributes of the topology are used)

- 'name(string):void' and 'name():string' for initializing and retrieving the name of the topology (used for saving the contents of the topology)

- 'illuminationmode(mode:integer):void' and 'illuminationmode():integer' for setting and retrieving the illumination model[FDFH90] that is used to render the polygons that are stored in the topology. This illumination mode may have the following values:

  0. Surface color

  In this mode no illumination calculation is done. The polygons are filled with the color of its attribute.

  1. Vertex color\(^7\)

  In this mode no illumination calculation is done. The polygons are filled by interpolating the colors of the vertices across the surface.

\(^7\)In the class library as described in this thesis points cannot have associated colors; hence there is no difference between mode 0 and 1. Later on, however, derived classes of class point may be introduced that have extra information, e.g. color, normal, or both.
2. Flat shading
   In this mode the illumination calculation is done only once per surface, using the surface normal. The resulting color of this calculation is then used to fill the polygons.

3. Gouraud shading
   In this mode illumination calculations are done at each vertex. The resulting colors of the calculations are then interpolated across the surface.

4. Phong shading
   In this mode illumination calculations are done at each pixel. The normal at each pixel is calculated by interpolating the vertex normals across the surface.

Finally, class topology contains methods 'selectable(s : selectable) : void' and 'selectable() : selectable'. These methods are used for direct manipulation purposes and are explained in more detail in Section 6.7.

6.6.3 Light sources

The visualisation of a 3-D scene may be drastically influenced by light sources. This influence can be categorized into the following reflection models [FDFH90]:

- **ambient**: models isotropic reflection of isotropically incoming light. Consequently, this model is independent on the orientation or position of a surface and is also independent on the viewing direction.

- **diffuse**: models isotropic reflection of anisotropically incoming light (typically the incoming light is modelled as a spot-source or a directional source). Consequently, this model is independent on the viewing direction, but it depends on the orientation or position of a surface.

- **specular**: models anisotropic reflection of anisotropically incoming light. Consequently, this model is dependent on the orientation and position of a surface and also on the viewing direction. Furthermore, the color of the reflected light may be different from the color of the surface, thereby allowing for effects such as highlights.

The amounts in which these models contribute to the visualization of a primitive, is specified by the primitive itself (these amounts are known as *material properties of a primitive*). The material properties are stored in an attribute object. Furthermore, each primitive object has an attribute of type attribute, and can therefore refer to such an attribute object. This approach allows for the situation where many primitive objects refer to the same attribute object, therefore the material properties of these primitives can be changed simultaneously. An attribute object has the following methods:

- 'color(c:color):void' and 'color():color' for setting and retrieving the color of a primitive

- 'linestyle(l:integer):void' and 'linestyle():integer' for setting and retrieving the style in which lines are rendered (e.g. solid, dashed, dotted, etc.)
6.6. Rendering

- "filstyle:integer:void" and "filstyle:integer" for setting and retrieving the style in which polygons are filled (e.g. solid, stippled, hatched, etc.)

- "ambient(b : boolean) : void", "diffuse(b : boolean) : void", and "specularon (b : boolean) : void" for specifying which reflection models should be used for the visualization of a primitive

- "ambient() : boolean", "diffuse() : boolean", and "specularon() : boolean" for inspecting which reflection models are used

- "ambient(r : real) : void", "diffuse(r : real) : void", and "specular(r : real) : void" for setting the factors (amounts in which the reflection models contribute to the visualization of a primitive (see Table 6.13)

- "ambient() : real", "diffuse() : real", and "specular() : real" for retrieving these factors

- "specularcolor(c : color):void" and "specularcolor():color" for setting and retrieving the color of reflected light in case of a specular reflection model (see Table 6.13)

- "specularpower(r : real):void" and "specularpower():real" for setting and retrieving the exponent that controls how fast specular reflectance falls off, as the angle increases between the view direction and the direction of the reflected incoming light increases (see Table 6.13)

- "ambientlight(b : boolean) : void", "directionallight(b : boolean) : void", "positionallight (b : boolean) : void", and "spotlight(b : boolean) : void" for specifying which kinds of light sources contribute to the visualization of a primitive (these 4 types of light sources are introduced below)

- "ambientlight() : boolean", "directionallight() : boolean", "positionallight() : boolean", and "spotlight() : boolean" for retrieving which kinds of light sources contribute to the visualization

The light sources that are supported by the GDP have inherited from a common base class light. This class has inherited from readable and writable, therefore light sources can be initialized and saved using files. This class has the following methods:

- "switch(b : boolean) : void" for switching the light source on/off

- "switch() : boolean" for retrieving the state of the switch

- "color(c : color):void" for setting the color of the light source

- "color() : color" for retrieving the color of the light source

The following four light source classes may be used to illuminate a 3-D scene:

1. light.ambient

This light source models light coming from all directions uniformly; it contributes only to the ambient reflections of a primitive. It has only one property: a color. Because this attribute is already specified by base class light, no new attributes or methods are introduced in class light.ambient.
2. light\_directional
   This light source models parallel light rays; it contributes to the diffuse and specular reflections of a primitive. The following methods are introduced in class light\_directional:
   - 'direction(d:vector):void' for setting the direction of the parallel light rays
   - 'direction():vector' for retrieving this direction

3. light\_positional
   This light source models light that is radiated uniformly from a single point; it contributes to the diffuse and specular reflections of a primitive. Furthermore, the intensity of the light received by a primitive depends on the distance from the light source to the primitive. This dependence may be controlled by means of two attenuation factors (see Table 6.13). The following methods are introduced in class light\_positional:
   - 'position(p:point):void' and 'position():point' for setting and retrieving the position of the light source
   - 'attenuation1(r:real) : void', 'attenuation1() : real', 'attenuation2(r:real) : void', and 'attenuation2() : real' for setting and retrieving the attenuation factors that control the intensity profile of the light source.

4. light\_spot
   This light source models light that is radiated in the form of a cone; it contributes to the diffuse and specular reflections of a primitive. This cone is specified by a position (i.e. the top), a direction, and an angle. Similar to a light\_positional, the intensity of the light received by a primitive depends on the distance from the top of the cone to the primitive. This intensity also depends on the position of a primitive in the cone: the intensity is maximum at the principal axis of the cone, and falls off with the cone angle. The following methods are introduced in class light\_spot:
   - 'position(p:point):void' and 'position():point' for setting and retrieving the position of the light source
   - 'attenuation1(r:real) : void', 'attenuation1() : real', 'attenuation2(r:real) : void', and 'attenuation2() : real' for setting and retrieving the attenuation factors that control the intensity profile of the light source.
   - 'direction(d:vector):void' and 'direction():vector' for setting and retrieving the direction of the cone
   - 'angle(r:real):void' and 'angle():real' for setting and retrieving the angle of the cone
   - 'exponent(r:real):void' and 'exponent():real' for setting and retrieving the exponent that controls the concentrical intensity attenuation (see Table 6.13)

Table 6.13 shows how the four types of light sources contribute to the three reflection models\[1\]. The specular reflection model that is used in this table is based on the Phong illumination model [FDFH90, Sun91b].

---

\[1\] In the table operator '*' is used both to multiply two colors and to multiply a color and a real factor. These multiplications are defined as follows:
6.6. Rendering

<table>
<thead>
<tr>
<th>light.ambient</th>
<th>Ambient</th>
<th>Diffuse</th>
<th>Specular</th>
</tr>
</thead>
</table>
| light.directional | $F_a * C_p * C_l$ | $V_d * C_p * C_l * |V_p, V_l||$
| light.positive | $F_d * C_p * A$ | $V_d * C_p * A * |V_p, V_l||$
| light.spot | $F_d * C_p * A$ | $V_d * C_p * A$ |$V_d * C_p * A$ |

Table 6.13: Light sources and reflection models

In this table the following abbreviations are used:

$A = \frac{1}{F_a + F_d + F_s * (P_p - P_i)}$

$V_r = 2 * (V_p, V_l) * V_p - V_l$

Furthermore, the following symbols are used:

$A$ : attenuation
$A_s$ : spread angle (light.spot property)
$C_l$ : color of light source (general light property)
$C_p$ : color of primitive (material property)
$C_r$ : color of reflected light (material property)
$E_c$ : exponent that controls the light source concentration (light.spot property)
$E_s$ : exponent that controls the fall of the specular reflectance (material property)
$F_a, F_d, F_s$ : factors for ambient, diffuse, and specular reflection model (material properties)
$F_a_1, F_a_2$ : attenuation factors (properties of light.positive and Light.spot)
$P_p$ : position of primitive
$P_l$ : position of light source
$V_l$ : direction of light source (unit vector)
$V_p$ : unit normal vector to primitive
$V_p e$ : unit vector from primitive to eye point
$V_p l$ : unit vector from object to light source
$V_r$ : unit reflection vector from primitive

For the illumination of a 3-D scene multiple light sources of the aforementioned types may be used. The contributions of these light sources to the visualization of a primitive are added. However, this may result in colors whose components exceed the maximum value of 1. This overflow may be solved by the following solutions [FDFH90]:

$(r, g, b) * (r, g, b) = (r^2 + g^2 + b^2)$

$f * (r, g, b) = (f^2 * r, f^2 * g, f^2 * b)$

$[r, g, b] * f = (r * f, g * f, b * f)$

$[v_1, v_2] = (v_1, v_2)$, if $(v_1, v_2) \geq 0$

$0$, if $(v_1, v_2) < 0$
1. truncate the 'offending' component to 1
2. scale the components in such a manner that the 'offending' component is scaled to 1
3. compute all colors before display, and perform image-processing to bring the components within the range [0,1]

In the GDP the first alternative is chosen.

6.6.4 Camera

A camera object visualizes a 3-D scene from a specified point and direction in this 3-D space. This visualization is displayed on an associated canvas object (see Section 6.5), which may be presented on a computer that supports the X protocol. Class camera has inherited from class responder; consequently, a camera may be controlled by means of events. For the initialization of a camera object the following methods are supported:

- 'init(c:canvas):void' for associating canvas c with the camera.
- 'canvas():canvas' for retrieving the canvas that is associated with the camera.
- 'eye(e:point):void' and 'eye():point' for setting and retrieving the position of the camera.
- 'dir(v:vector):void' and 'dir():vector' for setting and retrieving the view direction (procedure dir automatically normalizes vector v).
- 'up(v:vector):void' and 'up():vector'

Specifying only the position and the view direction of a camera is not sufficient for a unique camera orientation: it may be rotated around this axis. To remove this degree of freedom a direction has to be specified that indicates the vertical direction as seen the camera. The aforementioned methods may be used for setting and retrieving this direction. Note that the up direction and the view direction may not be identical. The argument of method up does not have to be perpendicular to the view direction: this is done automatically by projecting the argument onto the plane perpendicular to the view direction.

- 'hor():vector' for retrieving the horizontal direction of the camera. This direction is calculated by taking the cross product of the projected up vector and the view direction. Consequently, the view direction, the horizontal direction and the projected up direction form a right-handed basis.
- 'fov(angle:real):void' and 'fov():real' for setting and retrieving the field-of-view: the angle of the pyramid that is further specified by the position and view direction of the camera.
- 'near(near:real):void', 'near():real', 'far(far:real):void', and 'far():real'

To skip the rendering of primitives that are either too close or too far from the camera, the pyramid is truncated by two clipping planes: the near and far clipping planes. The aforementioned methods may be used for setting and retrieving the distance of these planes (the distance is measured to the eye position).
Class camera has inherited from classes readable and writable. The read and write methods initialize respectively store the following information of a camera object: eye, dir, up, fov, near, and far.

Rendering a 3-D scene may be rather time consuming. In order to speed this process up z-buffering may be turned off. Especially for rendering wire-frames, which consist of lines only, this may be beneficial since z-buffering is used to determine visible surfaces (the result of rendering polygons with z-buffering turned off is undefined). For controlling z-buffering a camera object supports the following methods:

- `zbuffer(on:boolean):void` for specifying whether z-buffering should be used in the rendering process
- `zbuffer():boolean` for retrieving the state of z-buffering control

As explained in Section 6.6.2 topological primitives are stored in a topology. When not all cameras render the same topologies it becomes possible to render the same scene in various ways. For example, one camera may render a cube as a wire-frame, whereas another camera may render the same cube as a solid. This situation has a real world analogy: for example, consider the different ways in which one object is shown using either an X-ray camera or an infra-red camera. To facilitate this kind of rendering, a camera has to know which topologies it has to render. For this purpose class camera is a descendant of type list[topology], where the inherited methods are renamed as follows: `add` -> `addtopology`, `get` -> `gettopology`, `valid` -> `validtopology`, `first` -> `firsttopology`, `next` -> `nexttopology`, and `amount` -> `amounttopology`.

For similar reasons, it may be interesting to render the same topologies on various cameras, but with different lighting. Therefore, a camera has to know which light sources are relevant. For this purpose class camera is a descendant of type list[light], where the inherited methods are renamed as follows: `add` -> `addlight`, `get` -> `getlight`, `valid` -> `validlight`, `first` -> `firstlight`, `next` -> `nextlight`, and `amount` -> `amountlight`.

Furthermore, a camera may be attached to a geometry. This means that the basis of the camera, i.e., eye-position, view direction, horizontal direction, and up direction, is taken from a geometry. This is particularly useful when using geometries that have (inverse) kinematics (see Section 6.8) or dynamics behavior (see Section 6.9), because it will facilitate the specification of complex camera movements. For example inverse kinematics will allow for cameras that automatically follow a certain moving object. For specifying this kind of attachment a camera object supports the following methods:

- `carrier(g:geometry):void` for specifying that the camera has to be attached to geometry g.
- `carrier():geometry` for retrieving the geometry to which the camera is attached.
- `releasecarrier():void` for releasing the attachment of the camera to a geometry; the camera will remain at the current position and orientation.

This is shown in Example 6.19 (page 135, where a camera is attached to a geometry that is kinematically controlled.

Finally, class camera supports methods for setting and retrieving a selector (Section 6.7.1) and a cursor (Section 6.7.2)\(^{20}\).

\(^{20}\)A camera can have at most one selector and one cursor.
• ‘selector(s:selector):void’ and ‘selector():selector’ for setting and retrieving the selector of the camera.
• ‘cursor(s:cursor):void’ and ‘cursor():cursor’ for setting and retrieving the cursor of the camera.

The selector and cursor of a camera are used for direct manipulation via that camera. Method ‘init(c:canvas):void’ of class camera automatically creates (inactive) selector and cursor objects. Choosing between these two direct manipulation objects may be carried out by the eventhandler of the camera, for example (with key_events):

• pressing ‘s’:
  - activate the selector (if selector and cursor were both inactive)
  - activate the selector and deactivate the cursor (if the cursor was active)
  - deactivate the selector (if the selector was active)

• pressing ‘c’:
  - activate the cursor (if selector and cursor were both inactive)
  - activate the cursor and deactivate the selector (if the selector was active)
  - deactivate the cursor (if the cursor was active)

In other words: the cursor and the selector can never be active simultaneously. Note that this is only a suggested default behavior of a camera. Naturally, derived classes of camera may implement a different regime.

As presented in Section 4.4, rendering is carried out after garbage collection:

```pseudocode
do true
  → "process asynchronous messages"
  "event handling";
  "restore WL-invariants";
  "garbage collection";
  "rendering";
  "parsing";
```

This rendering mechanism is carried out automatically, i.e., it is not programmed explicitly in Looks. Therefore, rendering can be considered as being local in Looks: the relation between the various rendering objects (such as geometries, topologies, lights, and cameras) is taken care of by the GDP, resulting in automatic updates of the contents of cameras, whenever one of the rendering objects has changed.

The automatic rendering mechanism can be described by means of the following algorithm (in pseudo-code):

```pseudocode
for each camera c
  → for each topology t attached to camera c
    → for each topological primitive p stored in topology t
      → "render primitive p in the canvas associated with camera c, using the light sources attached to camera c and the rendering attributes of primitive p and topology t"
```

21In a multi-processor system it could be carried out simultaneously with garbage collection
6.7 Direct Manipulation

As defined in Chapter 1 direct manipulation consists of controlling a running animation by means of a control device, such as a mouse. Since direct manipulation is only useful for visible objects, these objects have to be selected using camera objects. Indeed, camera objects provide the only way to make objects visible. Basically, direct manipulation comes down to two tasks: first, selecting the objects that should be controlled; second, controlling these objects. These two tasks are explained in Sections 6.7.1 and 6.7.2, respectively.

6.7.1 Selection

Selecting visible objects is carried out by a specialized object: a selector. This object will present a square around the 2-D locator whenever this locator is inside the canvas of a camera. By means of this square visible objects may be selected: every visible object that is covered by the square will be selected by the selector. Class selector has inherited from responder, and has the following methods:

- 'init(c:camera):void' for setting the selector object. Since selection is only possible via a camera object, initialization of the selector requires a reference to a camera, indicating that the selector will be used for selection via that camera.

- 'camera():camera' for retrieving the camera to which the selector is connected.

- 'active(b:boolean):void' and 'active():boolean' for setting and retrieving the state of the selector. If the selector is made active its square is presented on the canvas of the camera and it may be used to select visible objects; if it is made passive its square is removed and the selector cannot be used to select visible objects anymore.

- 'position(x,y:integer):void' for setting the center of the 'selection' square.

- 'setx():integer' and 'getx():integer' for retrieving the center.

- 'size():integer' for setting and retrieving the size of the 'selective' square.

- 'select():void' for selecting the visible objects that are covered by the 'selection' square of the selector.

- 'deselect():void' for deselecting the already selected visible objects that are covered by the 'selection' square of the selector.

- 'deselect_all():void' for deselecting all selected visible objects (regardless whether they are covered by the 'selection' square).

- 'primitives(t:topology):list[integer]' returns the indices of the primitives of topology that are selected by means of the selector.

For example, controlling the selector by means of a mouse may be implemented as follows:
1. The selector has to specify in which mouse events it is interested. Note that mouse events are UI events which are generated in widgets. Therefore, the selector also has to specify from which widget it wants to receive the mouse events. Since the selector is used in combination with a camera, it is obvious that this widget should be the canvas widget associated with the camera:

```plaintext
init(c:camera):void
    var mouse : mouse_event
    begin
        new(mouse);
        mouse.motion(true); // interested in mouse motion
        mouse.button(0,true,false); // interested in button 0 pressed
        mouse.button(2,true,false); // interested in button 2 pressed
        mouse.add(camera().canvas()); // event via the camera’s canvas
        responder:watch(mouse);
        // remember that we are writing a method for a class
        // which inherits from class selector, which in turn
        // inherits from class responder
    end;
```

2. The eventhandler of the selector specifies how the selector responds to these mouse events:

```plaintext
eventhandler(e:event):void
    var me : mouse_event;
    button : integer
    begin
        me=e;
        // e will always be a mouse_event (due to method init)
        button=me.button
        // retrieve button-number from event
        if button.eq(-1)
            then // button=-1 : motion event
                position(me.mouse().x(),me.mouse().y()) // the square is moved to the position of the mouse
        if button.eq(0)
            then // button=0 : button event
                select() // the objects covered by the square are selected
        if button.eq(2)
            then // button=2 : button event
                deselect() // the objects covered by the square are deselected
    end
```
This mechanism allows for a very flexible way to deal with selection of objects by means of a mouse: the actual mouse control (i.e. the mapping of mouse events to selector operations) is not fixed, but may be specified by an application. It even allows for a configuration where selection is carried out by means of keyboard events.

As explained in Section 6.6.2 the only visible objects are topological primitives. So theoretically only primitives can be selected by means of a selector object. Obviously, this is not exactly what is wanted: for example, when the user tries to select a cube, he or she is probably not interested in the primitives that constitute this cube. Consequently, a technique has to be introduced to select objects via constituting topological primitives. For this purpose class selectable is introduced. This class has inherited from responder, and has the following methods:

- `select(s:selector):void` for specifying that the selectable is selected by selector s.
- `selected(p:selector):boolean` for testing whether the selectable has been selected by selector s.
- `deselect(s:selector):void` for specifying that the selectable is not selected by selector s.
- `deselect_all():void` for specifying that the selectable is not selected by any selector.

As presented in Sections 6.6.1 and 6.6.2 classes geometry and topology both have an attribute of type selectable. These attributes are used as follows in method select of class selector (in pseudo-code):

```java
for each topological primitive p that is covered by selector s1's square
    if p.selectable is not selected yet
        p.selectable.select(s1);
        // the selectable attribute of a topology can be set by means of
        // method 'selectable(s : selectable) : void'
        // it is up to the implementation of this attribute's method
        // 'select' what the effect of this selection will be; in
        // example 6.14 method 'select' specifies that a rotating
        // cube should reverse its rotation.
    for each geometry g that is referred to by p
        if g.selectable is not selected yet
            g.selectable.select(s1);
            // the selectable attribute of a geometry can be set by
            // means of method 'selectable(s : selectable) : void'
            // again the behavior is up to the implementation of
            // this attribute's method 'select'.
```

Note that in this approach geometries and topologies have not inherited from class selectable, but have an attribute of class selectable instead; this allows for an easy way to change the behavior of geometries and topologies to selection (and control): changing this attribute will also change their behavior.

---

22 Method deselect is implemented in a similar way.
The reaction to a selection is totally up to the implementation of the `select` method of a `selectable` object. For example, the following implementations of this method may be considered:

- change visualization attributes of the selected topology
- change attributes (e.g. color) of the selected topological primitives (note: not all primitives of a topology may have been selected)
- change the motion behavior of the associated geometry (this is shown in Example 6.14 where selection of a rotating cube has the effect that the cube reverses its rotation).
- specify that an event should be watched, e.g. `mouse_events` or `cursor_events` (see the next section).

**Example 6.14: Selection of a Cube**

```plaintext
class Cube
  inherit selectable
  public
    init(): void
  private
    g : geometry;
    t : topology;
    a : axis;
    angle : integer;
    rotate() : void;
  implementation
    init(): void
    begin
      "Initialization of g, t, and rotation axis a":
      g.a := self; // select g results in selection of the cube
      t.a := self; // selection of t results in selection of the cube
      angle := 10; // rotate over 10 degrees per frame
      rotate();
    end;
    rotate() : void;
    begin
      while true do
        begin
          "rotate over a degrees";
          synchronize;
        end
      end;
    selectable(): void
    begin
      angle := angle.neg(); // reverse rotation angle
    end
  endclass
```

### 6.7.2 Control

Controlling visible objects is already partly possible by means of `mouse_events`. For example, the objects may respond to buttons and/or mouse motions. However, mouse motions fail in one aspect: they are 2-D motions. Clearly, this is insufficient for controlling the 3-D objects presented in this chapter. For specifying such 3-D motions, a 3-D cursor is introduced, which is based on a perspective cursor, such as described in [Ove89]. This cursor consists of three orthogonal axes; one of them will be called the main axis. Furthermore, the cursor supports two modes:
6.7. Direct Manipulation

1. **construction** mode: the cursor is allowed to move in the direction of the main axis or to rotate around the main axis (note: each of the 3 axes can be made the main axis)

2. **trace** mode: the cursor is allowed to move in a plane perpendicular to the main axis and incident with the central point.

Class `cursor` has inherited from `responder`, and supports the following methods:

- `'init(c:camera):void` for initializing the cursor object. Since control is only possible via a `camera` object, initialization of the cursor requires a reference to a `camera`, indicating that the cursor will be used for control via that `camera`. Furthermore, the position of the cursor is initialized to a point on the view axis of the `camera`, and the axes of the cursor are aligned to the `dir`, `up`, and `hor` of the `camera`. Finally, the cursor specifies that it is interested in `mouse.events` via the `camera`'s `canvas`.

- `'camera():camera` for retrieving the `camera` to which the cursor is connected.

- `'active(b:boolean):void` and `'active():boolean` for setting and retrieving the state of the cursor. If the cursor is made active its axes are visible on the canvas of the `camera` and it may be used to control objects; if it is made passive its axes are hidden and the cursor cannot be used to control objects anymore.

- `'position(p:point):void` and `'position():point` for setting and retrieving the 3-D position of the cursor.

- `'prev.position():point` for retrieving the previous position of the cursor, used to drive cursor displacement.

- `'direction(i:integer:d:direction):void` and `'direction(i:integer):direction` for setting and retrieving the direction of the i\textsuperscript{th} axis of the cursor (0\leq i<3).

- `'prev.direction(i:integer):direction` for retrieving the previous direction of the i\textsuperscript{th} axis of the cursor (0\leq i<3).

- `'main(i:integer):void` and `'main():integer` for setting and retrieving the index of the main axis (0\leq i<3).

- `'construction(b:boolean):void` and `'construction():boolean` for setting and retrieving the mode of the cursor (if the mode is not construction-mode, it will be trace-mode).

- `'size(r:real):void` and `'size():real` for setting and retrieving the size of the cursor (i.e. the length of its axes).

- `'translate(r:real):void` for translating the cursor along its main axis over distance r.

- `'translate(r,s:real):void` for translating the cursor in the plane perpendicular to the main axis.

- `'rotate(r:real):void` for rotating the cursor around its main axis over angle r.
• 'scale(r:real):void' for scaling the cursor.

• 'reset():void' for returning the cursor to its initial position and orientation.

Similar to a mouse, a cursor may generate events: cursor.event. Since cursor.events
are generated by a cursor that belongs to a specific camera (and hence to a specific
canvas), class cursor.event can be considered as a special kind of class UI.event. Class
cursor.event supports the following methods:

• 'cursor(c:cursor):void' for creating a cursor.event in Looks.

• 'cursor():cursor' for retrieving the cursor that caused the cursor.event.

The following methods of class cursor will generate a cursor.event:
A cursor should also be controllable by means of a mouse. For this purpose class cursor
supports the following private methods:

• 'translate(x,y:integer,dx,dy:integer):void' for translating the cursor (in either con-
struction-mode or trace-mode) according to a mouse motion.

• 'rotate(x,y:integer, dx,dy:integer):void' for rotating the cursor (in construction-
mode) according to a mouse motion.

For example, controlling the cursor by means of a mouse may be implemented as
follows:

1. Similar to the selector, the cursor has to specify in which mouse events it is
interested:

```pascal
init(c:camera):void
  var mouse : mouse_event
  begin
    :
    new(mouse);
    mouse.motion(true); // interested in mouse-motion
    mouse.add(camera().canvas());
    // event via the camera's canvas
    responder:watch(mouse);
    // only interested in motion events with button pressed
    :
  end;
```
2. The `eventhandler` of the `cursor` specifies how the cursor responds to these mouse events:

```plaintext
eventhandler(e: event): void
    var me: mouse_event;
    ce: cursor_event;
    x,y,dx,dy: integer;
    begin
        me? e;
        // e will always be a mouse_event (due to method init)
        x:=me.mouse().x(); y:=me.mouse().y();
        dx:=me.getdx(); dy:=me.getdy();
        // retrieve position and displacement from the event
        if me.mouse().button_down(1)
            then // button 1: trace mode
                translate(x,y,dx,dy);
                // trace mode: only translate
                if me.mouse().button_down(2)
                    then // button 2: construction mode
                        if dx.abs().gt(dy.abs())
                            then translate(x,y,dx,dy)
                                // transl. over main axis
                            else rotate(x,y,dx,dy)
                                // rotation around main axis
                            new(ce); // the cursor has moved
                            ce.cursor(self);
                            // therefore a cursor_event has to be generated
                            ce.signal();
            end
    end
```

Again it should be noted that the above is just a suggested default behavior of a cursor. Derived classes of class `cursor` may implement another behavior.

Finally, Example 6.15 shows how the classes that are presented in this section may be used to control a cube by means of direct manipulation. This example demonstrates various aspects of direct manipulation in Looks:

- selection is performed via the selectable attribute of geometry and topology objects,
- cursor movements are available by means of a `cursor_event`,
- an object that should be controllable by means of a cursor has to specify that it is interested in such a `cursor_event`\(^\text{23}\) (in the example this is carried out when the cube gets selected).

\(^{23}\)If the direct manipulation also involves mouse-buttons, the object should also specify that it is interested in mouse_events.
Example 6.15: Direct Manipulation of a Cube

class cube

inherited selectable
public
init():void
private
  g: geometry;
  t: topology;
implementation
init():void
begin
  "initialization of g and t":
  g := self; // selection of g results in selection of the cube
  t := self; // selection of t results in selection of the cube
end;
selectable:select(e, cam) : void
begin
  var ce : cursor.event;
  new(ce);
  ce.add(c);
  responder:watch(ce);
  // selection of the cube has the effect that it will watch cursor
  // events that are generated in the canvas of camera e
end;
responder:eventhandler(e, event) : void
begin
  var ce : cursor.event;
  org : point;
  ndir, ydir, zdir : vector;
  ce:=a; // test whether a is a cursor.event
  "retrieve position (org) and orientation (ndir, ydir, zdir) from ce":
  g.basis(org, ndir, ydir, zdir); // align cube with cursor
end

endclass

6.8 (Inverse) Kinematics

6.8.1 Introduction

Chapter 1 introduced the techniques of (forward) kinematics and inverse kinematics. The former deals with specifying an animation by means of supplying trajectories, over time, of parameters that describe the characters in an animation. For example, in Figure 6.16 angles α and β specify the position and orientation of object rigid1 and rigid2 (and thus of point s, that lies on rigid2). In this figure revolute joints are used; these are kinematic elements that allow for a rotation.

Inverse kinematics deals with inferring the parameters from given positions of designated parts of the characters. For example, in Figure 6.16 a motion specification based on inverse kinematics may consist of specifying that point s should move to point t; the angles α and β are subsequently calculated in such a way that point s (on rigid2) reaches point t. Often, inverse kinematics has to cope with underspecification. This is demonstrated in Figure 6.17 where two solutions are shown that both meet the specification.

The numerical aspects of inverse kinematics are treated by several authors, see e.g. [Hau89, GM85]. The canonical solutions typically take all kinematic objects into consideration in order to infer the parameters (e.g., by computing a "generalized inverse" of a Jacobian matrix). Note that this obstructs the concept of localization. Furthermore, the number of kinematic objects and the relations between them do not
6.8. (Inverse) Kinematics

![Diagram of kinematics using revolute joints]

Figure 6.16: Kinematics using revolute joints

![Diagram of underspecification of inverse kinematics]

Figure 6.17: Underspecification of inverse kinematics

have to be fixed in the GDP (due to direct modification). Such solutions are therefore not very well suited for the GDP, because they are designed for a static configuration of objects and relations. The main goal of this section is to show how (inverse) kinematics may be specified in an object-oriented way. The underlying algorithms are rather naively implemented: the numerical aspects were not elaborated upon.

Before we give a more detailed description of the Looks classes for (inverse) kinematics, we explain the general approach towards articulated bodies and (inverse) kinematics that is taken in the GDP.

**Articulated Bodies**

In general kinematics deals with a configuration of rigid objects that are linked by means of joints that allow for relative motion (such a configuration is also known as an articulated body). For example, in Figure 6.16 the configuration consists of two rigid objects that are linked by means of a revolute joint (whose relative motion consists of a rotation). Furthermore, rigid object rigid1 is also linked to the world by means of a revolute. Consequently, the configuration for general kinematics can be considered as a graph, with rigid objects as nodes and joints as edges (indeed: joints specify a binary relation between nodes).

In order to simplify the calculations that are needed for (inverse) kinematics, the GDP restricts this graph to be a tree. If such a tree would be implemented using object-oriented technology, both the joints and the rigid objects would be implemented as objects.
(as can be seen in Figure 6.16). It can be noted that in terms of object relations, rigid
and joints will always alternate. Furthermore, rigid objects are only used for position and
orientation purposes (cf. a geometry), whereas joint objects are only used to specify
the relative motion between rigid objects. Consequently, a rigid and a joint may be combined
in one object: this object has a position and orientation, and it also has a relative motion
with respect to its father in the "kinematics configuration" tree. This kind
of object is specified by means of class kinematic (specialized classes with dedicated
relative motion, such as revolutes, prisms, cylinders, etc. are presented in Sections
6.8.2 to 6.8.8).
Since kinematic objects have a position and orientation (and points for visualization
purposes), class kinematic can be considered as a descendant of class geometry. There-
fore, each element has its own local basis L and corresponding transformation matrix
M_{WL} (that transforms the coordinates of a point given in L to coordinates in world-
basis W). As explained in Section 6.6.1 a geometry can be moved by changing basis
L and transformation M_{WL}. However, this would be troublesome for an element of
an articulated body: its children in the "configuration" tree would have to move too
(resulting in updates of the L basis and M_{WL} of all its children). Therefore, another
transformation is used, say M_{FL} that transforms the coordinates of a point given in
basis L to coordinates in the basis of its father in the "configuration" tree. The relation
between M_{FL} and M_{WL} is specified by means of the WL-invariant (see page 109).
Now the motion of an element of an articulated body may be specified by means of
an update of transformation M_{FL}. Note that this will result in a violation of the
WL-invariant. For example, the movement of the lower arm can be expressed in terms
of the basis of the upper arm (the actual world coordinates of the lower arm are not
relevant for this motion; indeed: they only come into play for rendering).
As said above, changing the M_{FL} due to a motion, will result in a violation of WL-
invariant. The kinematic object will automatically register whether such a violation
has occurred, and will restore the invariant, whenever world coordinates are required,
by means of the following procedure (in pseudo-code):

```plaintext
restore_WL-invariant(): void;
begin
  if violated
    if "father does not exist"
      M_{WL} := M_{FL}
      // M_{FL} contains the correct M_{WL} of the root
    else "father exists"
      father.restore_WL-invariant();
      // make sure father's WL-invariant is not violated
      M_{WL} := father.M_{WL} \cdot M_{FL}
      // update own M_{WL}
    fi
  fi
  violated := false;
end
```

For manipulating the hierarchy of elements class kinematic supports the following meth-
ods:
6.8. (Inverse) Kinematics

- 'childof(father:kinematic:integer)' for specifying that the kinematic object is a child of element father in the hierarchy. Furthermore, the \( M_{FL} \) transformation of the kinematic object is initialized as follows: 
  \[ M_{FL} = (\text{self}.\text{father}.M_{WL})^{-1} \cdot \text{self}.M_{WL}. \]
  This function returns the unique index of the kinematic object in its father's offspring. In case this function would violate the tree-structure, '-'1' is returned.

- 'getfirstchild(integer)' and 'getnextchild(i:integer):integer' for retrieving the indices of the kinematic object's children.

- 'getchild(i:integer):kinematic' for retrieving the child with index \( i \).

- 'ancestor(i:integer):kinematic' for retrieving the \( i^{th} \) ancestor of the kinematic object (0 represents the kinematic object itself, 1 its father, 2 its grandfather, etc.).

- 'disconnect():void' for removing the kinematic object from its father's offspring.

- 'promote():void' for promoting the kinematic object to be the root of the hierarchy (i.e. the element linked directly to basis \( W \)). This may be useful, e.g. for implementing a walking figure where the left and right feet will be alternatively root of the hierarchy. This procedure is implemented as follows (in pseudo-code):

  ```
promote():void
  begin
    if "father does not exist"
      skip
        // no father: the kinematic object is already root
    else "father exists"
      geometry temp = father;
      father.promote();
      // now father is the root of the hierarchy
      disconnect();
      // remove the object from father's offspring
      temp.childof(self);
      // now the kinematic object is root
  end

Kinematics

For performing kinematic behavior class kinematic supports one method: 'step():void'. In the implementation of this method transform \( M \) is used, which defines the specialized relative motion of the various joints. Furthermore, the motion specified by transform \( M \) is given in a basis of a certain geometry, say frombasis. Method step is implemented as follows:

\[ M_{FL} := \text{self}.\text{father}.M_{WL}^{-1} \cdot \text{frombasis}.M_{WL} \cdot M \cdot \text{frombasis}.M_{WL}^{-1} \cdot \text{self}.M_{WL}. \]

Notice that the right-hand expression can be seen as a matrix transformation which expresses the effect of \( M \) in my father's base.

For some cases this expression can be simplified:
• `frombasis=self` (M is given in local coordinates):
  \( (\text{self.father.M}_{wL}^{-1} \cdot \text{frombasis.M}_{wL}) = M_{FL} \) and \( (\text{frombasis.M}_{wL}^{-1} \cdot \text{self.M}_{wL}) = I \), thus:
  \( M_{FL} := M_{FL} \cdot M \)

• `frombasis=self.father` (M is given in father coordinates):
  \( (\text{self.father.M}_{wL}^{-1} \cdot \text{frombasis.M}_{wL}) = I \) and \( (\text{frombasis.M}_{wL}^{-1} \cdot \text{self.M}_{wL}) = M_{LF} \), thus:
  \( M_{FL} := M \cdot M_{FL} \)

• `frombasis.M_{wL}=1` (M is given in world coordinates):
  \( M_{FL} := \text{self.father.M}_{wL}^{-1} \cdot M \cdot \text{self.M}_{wL} \)

Inverse Kinematics

Inverse kinematic behavior is specified by means of a sight-point and a target-point. The motion of the element consists of moving the sight-point as closely as possible to the target-point. In Figure 6.17 the sightpoint and targetpoint are named s and t, respectively. For specifying the sight-point and the target-point class kinematic supports the following methods:

• `'sight(g:geometry;index:integer):void'` for specifying that the sight-point of the kinematic object is the index\(^{th}\) point of geometry g. Note that it is not required that the sight-point is one of the points of the kinematic object itself.

• `'target(g:geometry;index:integer):void'` for specifying the target-point.

For performing inverse kinematic behavior class kinematic supports one method: `'follow():void'`. This method will be redefined by the specialized elements that are defined in the following sections.

6.8.2 Prism

A prism is an element whose restricted relative motion consists of a translation (see Figure 6.18 for a drawing of a prism element). Class prism supports the following methods:

• `'init(d:real; v:vector; frombasis:geometry):void'` for specifying translation direction (argument v), which is given in local coordinates of geometry frombasis. Furthermore, d specifies the distance that will be used for one kinematic step.

• `'direction(v:vector; frombasis:geometry):void'` and `'direction(tobasis:geometry):vector'` for setting and retrieving the translation direction.

• `'distance(d:real):void'` and `'distance():real'` for setting and retrieving the translation distance of the kinematic motion.

Kinematic motion step will translate the prism object along its translation direction over distance d. The inverse kinematic motion follow translates the prism is such a way that the distance between the sight-point and the target-point will be minimized. The prism element supports some methods to control this motion:
6.8. (Inverse) Kinematics

- 'range(min, max: real): void', 'minrange(): real', and 'maxrange(): real' for setting and retrieving the range of allowed positions of the prism object.

- 'velocity(min, max: real): void', 'minvelocity(): real', and 'maxvelocity(): real' for setting and retrieving the maximum and minimum velocity of the prism object.

- 'acceleration(min, max: real): void', 'minacceleration(): real', and 'maxacceleration(): real' for setting and retrieving the minimum and maximum acceleration of the prism object.

![Diagram of a prism element](image)

Figure 6.13: 3-D view of a prism element

Example 6.19 shows how a prism may be used — as an attribute — to implement a camera-trolley that can move over a straight rail. Furthermore, this camera-trolley will automatically follow a specified geometry by translating over this rail. Figure 6.20 is a graphical representation of this configuration.

**Example 6.19: Automatic Camera Movement**

```java
class trolley
{
    public
        init(ccamera): void; // camera c is put on a trolley
        follow(g: geometry, i: index): void; // the trolley will follow geometry g
        stop(): void; // stop the follow motion

    private
        cont_follow(): void; // provides continuous follow motion
        p: prism; // provides the movement of the trolley
        following: boolean; // indicates whether the trolley is following g
        a: array(topology); // the topologies of the trolley

    implementation
        init(ccamera): void
        begin
            following = false; // initially the trolley is not following g
            p = new prism(p.reachfile("trolley.xml"));
            p.basis(c.eye(), c.dirx(), c.hor(), c.up());
            // make a new prism whose basis is identical to the basis of c
            p.init(l, c.hor(), -l);
            // the trolley may move in the horizontal direction of c
            c.carrier(p);
            // attach the camera to the prism
        end;
        follow(g: geometry, i: index): void
        begin
            following = true; // the trolley starts following
            p.sight(p, 0);
            p.target(g, 12);
            // the center of the camera is aimed at point 1 of geometry g
            cont_follow(); // start continuous follow motion of trolley
        end;
}
```

6.8.3 Revolute

A revolute is an element whose restricted relative motion consists of a rotation (see Figure 6.21 for a drawing of a revolute element). Class revolute supports the following methods:

- 'init(ang: real; a: axis; frombasis: geometry): void' for specifying rotation axis (argument a), which is given in local coordinates of geometry frombasis. Argument ang specifies the angle that will used for one kinematic step.

- 'axis(a: axis; frombasis: geometry): void' and 'axis(tobasis: geometry): axis' for setting and retrieving the rotation axis.

- 'angle(a: real): void' and 'angle(): real' for setting and retrieving the rotation angle of the kinematic motion.

Kinematic motion step will rotate the revolute object around its rotation axis over angle degrees. The inverse kinematic motion follow rotates the revolute in such a way that the distance the sight-point and the target-point will be minimized. Similar to class prism, class revolute supports methods range, velocity, and acceleration to control the calculation of this rotation angle.
6.8. (Inverse) Kinematics

Figure 6.21: 3-D view of a revolute element

6.8.4 Cylinder

A cylinder is an element whose restricted relative motion consists of an independent translation and rotation over one axis (i.e., the translation direction is identical to the direction of the rotation axis). Hence, a cylinder element can be considered as an independent combination of a prism and revolute object (see Figure 6.21 for a drawing of a cylinder element). Class cylinder supports the following methods:

- `init(d, ang: real; a: axis; frombasis: geometry): void` for specifying the axis (argument `a`), which is given in local coordinates of geometry from basis. Arguments `d` and `ang` specify the translation distance and rotation angle that will be used for one kinematic step.

- `axis(a: axis; frombasis: geometry): void` and `axis(tobasis: geometry): axis` for setting and retrieving the axis.

- `distance(d: real): void` and `distance(): real` for setting and retrieving the translation distance of the kinematic motion.

- `angle(a: real): void` and `angle(): real` for setting and retrieving the rotation angle of the kinematic motion.

Kinematic motion step will rotate the cylinder object around its rotation axis over angle degrees, while it will simultaneously translate the object along its translation direction over distance `d`. The inverse kinematic motion follows translates and rotates the cylinder in such a way that the distance between the sight-point and the target-point will be minimized. The calculation of the rotation angle and translation distance may be controlled by the following methods:

- `range (n: integer, min, max: real): void`, `minrange (nr: integer): real`, and `maxrange (nr: integer): real` for setting and retrieving the range of the cylinder object (`nr=0` sets/retrieves the range of the translation; `nr=1` sets/retrieves the range of the rotation).\footnote{Note that the looks programmer will typically use more meaningful names for argument `nr`, e.g., `object distance angle integer, distance=0, angle=1;`. Now the range of the angle may be specified as follows: `range(angle,-90,90);`.}
• ‘velocity(nr : integer; min, max : real) : void’, ‘minvelocity(nr : integer) : real’, and ‘maxvelocity(nr : integer) : real’ for setting and retrieving the minimum and maximum velocity of the rotation and translation.

• ‘acceleration(nr : integer; min, max : real) : void’, ‘minacceleration(nr : integer) : real’, and ‘maxacceleration(nr : integer) : real’ for setting and retrieving the minimum and maximum acceleration of the rotation and translation.

![cylinder](image)

Figure 6.22: 3-D view of a cylinder element

6.8.5 Helix

A helix is an element whose restricted relative motion consists of a coupled translation and rotation over one axis (i.e. the translation direction is identical to the direction of the rotation axis). Hence, a helix element can be considered as coupled combination of a prism and revolute object (see Figure 6.23 for a drawing of a helix element). Class helix supports the following methods:

• ‘init.turns(pitch, turns : real; a : axis; frombasis : geometry) : void’ for specifying the axis (argument a), which is given in local coordinates of geometry frombasis. Argument pitch specifies the translation distance of one complete rotation. Argument turns specifies the number of rotations that will be carried out during one kinematic step (the translation distance of the kinematic motion will be calculated using the pitch).

• ‘init.distance(pitch, d : real; a : axis; frombasis : geometry) : void’ for specifying the axis and pitch. Argument d specifies the translation distance that will be carried out during one kinematic step (the rotation angle of the kinematic motion will be calculated using the pitch).

• ‘axis(a : axis; frombasis : geometry) : void’ and ‘axis(tobasis : geometry) : axis’ for setting and retrieving the axis.

• ‘pitch(p : real) : void’ and ‘pitch() : real’ for setting and retrieving the pitch of the helix element.

• ‘distance(d : real) : void’ and ‘distance() : real’ for setting and retrieving the translation distance of the kinematic motion.
6.8. (Inverse) Kinematics

- 'turns(t: real): void' and 'turns(): real' for setting and retrieving the number of rotations in the kinematic motion.

The kinematic and inverse kinematic motions of a helix are similar to the motions of a cylinder. The only exception is the calculation of the angle and distance in the inverse kinematic motion of a cylinder: this calculation has to take into account that the rotation and translation are coupled. Controlling this calculation is similar to class cylinder.

![helix]

Figure 6.23: 3-D view of a helix element

6.8.6 Plane

A plane is an element whose restricted relative motion consists of two independent translations and one rotation (with rotation axis perpendicular to the translation directions). Hence, a plane element can be considered as independent combination of two prism objects and one revolute object (see Figure 6.24 for a drawing of a plane element). Class plane supports the following methods:

- 'init(d0,d1 : real; 10.11 : vector; frombasis : geometry; g : geometry; index : integer; ang : real) : void' for specifying the spanning vectors of the plane, both are given in local coordinates of geometry frombasis, and one of the points of the plane (this point is specified by means of geometry g and integer index - this point partly defines the rotation axis of the plane). Arguments d0, d1, and ang specify the translation distances and rotation angle that will be used for one kinematic step.

- 'distance(nr: integer; d: real): void' and 'distance(nr: integer): real' for setting and retrieving the translation distances and the rotation angle of the kinematic motion; argument nr is used as follows:
  nr=0 : first translation distance
  nr=1 : second translation distance
  nr=2 : rotation angle

- 'position(frombasis: geometry; index: integer) : void' and 'position(tobasis: geometry): point' for setting and retrieving the position.

- 'direction(nr : integer; v : vector; frombasis: geometry) : void' and 'direction(nr : integer; tobasis: geometry) : vector' for setting and retrieving the translation directions (0 ≤ nr < 2).
- `axis(tobasis:geometry):axis` for retrieving the rotation axis.

![Diagram of a plane element](image)

Figure 6.24: 3-D view of a plane element

Naturally, the two translations may be combined into one joint translation. This joint translation and the rotation of the plane are independent, which results in an underspecified kinematic motion. This is demonstrated in Figure 6.25, which shows an initial situation, the situation after a translation and a subsequent (clockwise) rotation, and the situation after a rotation and a subsequent translation. Consequently, the order of translation and rotation has to be specified for each plane. For this purpose class `plane` supports methods `translatefirst(b:boolean) : void` and `translatefirst() : boolean` for setting and retrieving whether the translation precedes the rotation.

![Diagram of order of translation and rotation](image)

Figure 6.25: The effect of the order of translation and rotation

The inverse kinematic motion of a plane has to cope with underspecification: if the sight-point has reached the target-point, the plane element may rotate around the axis through the sight-point. This is demonstrated in Figure 6.26, which shows an initial situation of a target-point t and a plane with sight-point s; this figure also shows two solutions for the inverse kinematic motion that moves the sight-point of the plane to target-point t (many other possible solutions exist).

This underspecification may be removed by specifying a (new) sight-point (`sight.aim`) to be aimed at a (new) target-point (`target.aim`). From these two points, a rotation is derived (cf. a revolute) such that the angle \(\angle(sight.aim,sight,target.aim)\) is minimized. For specifying these two points class `plane` supports the following methods:
6.8. (Inverse) Kinematics

Figure 6.26: Underspecification of the inverse kinematic motion

- 'sight\_aim (g:geometry, index:integer) : void' for specifying the aiming sight-point.
- 'target\_aim (g : geometry, index : integer) : void' for specifying the aiming target-point.

Figure 6.27 shows how an aiming sight-point and an aiming target-point specify a unique inverse kinematic motion. Finally, controlling the calculation of the rotation

\[
\begin{align*}
\text{angle and translation distance is similar to class cylinder.}
\end{align*}
\]

6.8.7 Sphere

A sphere is an element whose restricted relative motion consists of three rotations around axes that have one rotation point in common. Hence, a sphere element can be considered as combination of three revolute objects. Class sphere supports the following methods:

- 'init(g:geometry, index:integer, dir:up:vector, frombasis:geometry y, ang0,ang1,ang2: real) : void' for specifying the sphere: the rotation point of the sphere specified by means of geometry g and integer index). Argument dir specifies the main axis of the sphere, while up specifies a direction in the plane perpendicular to dir (note: up does not have to be perpendicular to dir, because it is projected automatically onto the plane perpendicular to dir); both vectors are given in local coordinates of geometry frombasis. Arguments ang0, and ang1 specify a 'spherical' rotation\(^{25}\).

\(^{25}\)Consider orthonormal base (hor,up,dir), with hor = up \times dir (note: up and dir are normalized automatically by method 'init'). The rotation angles ang0 and ang1 specify the rotation that trans-
Argument `ang2` specifies a rotation (twist) around the axis through the rotation point and the origin of the sphere.

- `'position(frombasis:geometry:index:integer) : void'` and `'position(tobasis:geometry) : point'` for setting and retrieving the common point of the axes.

- `'direction(nr:integer;v:vector;frombasis:geometry) : void'` for setting dir and up ($0 \leq nr < 2$; $nr=0$ specifies vector dir, $nr=1$ specifies vector up).

- `'direction(nr:integer:tobasis:geometry) : vector'` for retrieving vectors dir, up, and hor ($0 \leq nr < 3$; $nr=2$ retrieves vector hor).

- `'angle(nr:integer;ang:real):void'` and `'angle(nr:integer):real'` for setting and retrieving the rotation angles ($0 \leq nr < 3$).

![3-D view of a sphere element](image)

Figure 6.28: 3-D view of a sphere element

Similar to class `plane`, the order of the 'spherical' rotation and the twist may affect the final situation. This order may be specified by means of methods 'twistfirst (b:boolean) : void' and 'twistfirst() : boolean' for setting and retrieving whether the twist precedes the 'spherical' rotation.

The inverse kinematic motion of a sphere also has to deal with underspecification. Again this is solved by introducing methods for an aiming sight-point and target-point.

Finally, the calculation of the angles of the 'spherical' rotation and twist is controlled in a way that is similar to class `cylinder` ($nr=0$ controls the angle for the 'spherical' rotation; $nr=1$ controls the angle for the twist).

Forms $(0,0,1)$ to $(\sin(\text{ang0})\sin(\text{ang1}),\sin(\text{ang0})\cos(\text{ang1}),\cos(\text{ang0}))$. In other words: `ang0` specifies the angle between dir and the position vector of this new point, and `ang1` specifies the angle between up and this position vector.
6.8.8 Isomorph

A isomorph is a rather peculiar element, in the sense that its motion is not restricted and can therefore not be specified by means of a constant transform M. The kinematic motion of a isomorph element, i.e. method step, is based on integration for position and velocity. Class isomorph supports the following methods to control this integration:

- 'init(v:vector;frombasis:geometry):void' for specifying the velocity and acceleration, both vectors are given in local coordinates of geometry frombasis

- 'velocity(v:vector;frombasis:geometry):void' and 'velocity(tobasis:geometry):vector' for setting and retrieving the velocity.

- 'acceleration(a : vector; frombasis : geometry) : void' and 'acceleration(tobasis : geometry) : vector' for setting and retrieving the acceleration.

Besides kinematic motion based on integration, as carried out by method step, class isomorph also supports three other methods to perform kinematic motions:

- 'translate(d:real;v:vector;frombasis:geometry):void' for translating the isomorph object along vector v over distance d.

- 'rotate(ang:real;a:axis;frombasis:geometry):void' for rotating the isomorph object around axis a over angle ang.

- 'scale(f:real; frombasis:geometry; index:integer):void' for scaling the isomorph object with factor f with point index as the origin of the scale operation.

The inverse kinematic motion of class isomorph, i.e. method follow, simulates that the sight-point and the target-point are connected by a damped spring (this motion is based on function move() in [Ov91]); it may be controlled by means of two parameters:

- 'spring(s:real):void' and 'spring():real' for setting and retrieving the spring constant (this constant specifies that, in case of a fixed target-point, the sight-point reaches the target-point in approximately s steps).

- 'damping(d:real):void' and 'damping():real' for controlling the damping.

Class isomorph supports a second inverse kinematic motion that is similar to the aim functionality of classes plane and sphere (this motion is based on function follow() in [Ov91]). This motion may be controlled as follows:

- 'sight_aim (g:geometry; index:integer) : void' for specifying the aiming sight-point.

- 'target_aim (g : geometry; index : integer) : void' for specifying the aiming target-point.

- 'steps(s:integer):void' and 'steps():integer' for setting and retrieving the number of steps in which the sight_aim should be aimed at target_aim.

- 'aim():void' for performing one step of the aim motion of a isomorph.
6.9 Dynamics

As introduced in Chapter 1 the technique of dynamics deals with supplying animated bodies with masses and applying forces to these bodies in order to make them move. This section presents class dynamic, which is a geometry that has a mass and which supports methods for applying forces. This section also presents classes for constraints between such dynamic objects. The (dynamic) motion and constraint solving are both computed by means of an algorithm that is based on relaxation; the mathematical foundation of this algorithm can be found in [Bar94]. Again it should be noted that this section is not intended to elaborate on the various (mathematical) approaches to deal with dynamics in computer animation; it merely presents a way to specify dynamics in an object-oriented environment that is based on localization (i.e. the Looks programmer does not have to deal explicitly with dynamics and constraints: these calculations are carried out automatically by the GDP).

Class dynamic is basically a collection of points that have a mass. Therefore, class dynamic has inherited from class geometry. By default, points will have a unit mass. For the calculation of the motion due to the inertia each dynamic object has a private basis containing the main inertial axes. This basis is updated automatically whenever it may be violated, e.g. when new points are added or when the mass of a point is updated. Class dynamic supports the following methods:

- `mass(integer,m:real):void` and `mass(integer):real` for setting and retrieving the mass of the point with index i.

- `velocitydamping(v:real):void` and `velocitydamping():real` for setting and retrieving the velocity damping factor. With this factor the motion due to inertia will have dissipation (i.e. lose kinematic energy over time). For example, a damping factor of 1.0 will result into a continuous motion of the dynamic object once a force is applied to it, whereas a damping factor of less then 1.0 will result into a motion that gets slower over time.

- `relax(integer):void` and `relax():integer` for setting and retrieving the (maximum) number of relaxation steps that will be used to restore rigidness (which may have been violated by motion due to discretization of the motion equations, see [Bar94]).

- `applyforce(integer,f:vector):void` for specifying that a force, characterized by vector f, should be applied to the dynamic object at the point with index i. Note that this force is applied during exactly one time step.

The constraints between the dynamic objects have one common base class: constraint. This class supports the following methods:

- `active(a:boolean):void` and `active():boolean` for setting and retrieving the state of the constraint.

- `relax(integer):void` and `relax():integer` for setting and retrieving the (maximum) number of relaxation steps that will be used to satisfy the constraint.
6.9. Dynamics

- `maxforce(f:real):void` and `maxforce():real` for setting and retrieving the maximum force that may be used to solve the constraint. If during the calculation of the constraint a force occurs that exceeds this maximum value, the constraint will become inactive and as a result the motion of the associated objects will develop as if the objects break loose.

The following specialized constraints are supported:

1. **ptp**
   
   This constraint is used to specify that two points (one of which should be a point of a dynamic) are required to have the same coordinates. Class `ptp` supports the following method:
   
   - `init(d:dynamic; i1:integer; g:geometry; i2:integer) : void` for specifying that the `i1`\textsuperscript{th} point of `d` should be connected to the `i2`\textsuperscript{th} point of `g`. Note that `g` may be a geometry, a (specialized) kinematic object, or another dynamic object.

2. **linehinge**
   
   This constraint is used to specify that two combinations of two points are required to have the same coordinates per combination\textsuperscript{26}. Class `linehinge` supports the following method:
   
   - `init(d:dynamic; i1,i2:integer; g:geometry; i3,i4:integer) : void` for specifying that the `i2`\textsuperscript{th} point of `d` should be connected to the `i3`\textsuperscript{th} point of `g` and that the `i2`\textsuperscript{th} point of `d` should be connected to the `i4`\textsuperscript{th} point of `g`.

The calculations necessary for dynamics motion are carried out automatically in the main loop of the GDF. For this purpose, the main loop is modified as follows:

```plaintext
do true
   "process asynchronous messages"
   "event handling"
   "restore W-L-invariants"
   "dynamics"
   "garbage collection"
   "rendering"
   "parsing"
od;
```

The calculation of the dynamics motion, mainly consists of the following stages:

```plaintext
for each dynamical object
   calculate main axes of inertia (if necessary)
   apply motion due to inertia
   restore rigidness (relaxation)
```

\textsuperscript{26}Naturally, a linehinge can be considered as a combination of two ptp constraints. However, this introduces an extra degree of freedom between the two forces concerned: components of the two forces along the line hinge, but in different directions, cancel each other. This may result in unstable calculations. To overcome this problem, a specialized linehinge should be used whenever two dynamic objects should be connected by means of two point-pairs.
for each relaxation step
  for each active constraint
    estimate constraint forces
      for each dynamical object
        calculate motion due to constraint forces
        restore rigidness (relaxation)
Chapter 7

Sample Animation

7.1 Introduction

This chapter shows how some of the topics that have been discussed in the previous chapters can be combined to create an animation. The following topics are covered by this sample animation:

- Looks constructs: object-orientation, asynchronous message, synchronize
- Inverse kinematics
- Direct manipulation
- Camera initialization

The next section contains a synopsis of the sample animation. The actual (annotated) Looks script of the animation is presented in Section 7.3. Several frames of this animation can be found in Section 7.4.

7.2 Synopsis

The animation consists of an array of kite-shaped objects which have to follow each other, while they are trying to orient in the direction they are moving. Furthermore, the first of these kite objects can be controlled by means of direct manipulation. The geometry and topology (i.e. a single polygon) of such a kite object are shown in the next figure.

Figure 7.1: Geometry and topology of a kite

147
Now the motion conditions of a kite object can be specified more precisely:

1. the point with index 0 of a kite has to follow the point with index 2 of the next kite in the array

2. the main axis of a kite, i.e. the line through point 0 and point 2, should follow the main axis of the next kite in the array

Furthermore, there are some motion constraints with respect to the magnitudes of velocity, acceleration, and angular motions of the kites. Initially, the last kite will follow the first one. It should be clear that in this case the two conditions of the motion specification cannot be fulfilled.

7.3 Script

```c
// ******************************************************************************
// "max" contains the number of kites
// ******************************************************************************
object max : integer; max:=10;

// ******************************************************************************
// initialization of a window and a canvas
// ******************************************************************************
object win:window; new(win); win.open();
win.position(50,50); win.size(400,400);
// the window is ready to take part in the GUI
win.manage(true);
// the canvas is a child of the window and is also ready to take part in the GUI
object can:canvas; new(can); can.childof(win); can.manage();

// ******************************************************************************
// initialization of a camera
// ******************************************************************************
object eye:point; new(eye); eye.init(0,0,-600);
object dirvec:vector; new(dirvec); dirvec.init(0,0,1);
object upvec:vector; new(upvec); upvec.init(0,-1,0);
object cam:camera; new(cam);
// the camera is located at (0,0,-600) and looks along the positive z-axis; (0,1,0) indicates
// the top of the camera
cam.eye(eye); cam.up(upvec); cam.dir(dirvec);
// the camera has a field-of-view of 75 degrees
cam.fov(75);
// the camera sees every object at a distance between 1 and 10000
.cam.near(1); cam.far(10000);
// the camera shows its image in canvas 'can'
cam.init(can);
```
7.3. Script

// 'rainbow' contains the three possible colors of a kite
object rainbow : array[color]; new(rainbow); rainbow.size(3);
  rainbow.set(0, red); rainbow.set(1, green); rainbow.set(2, blue);

// 'kites' contains the geometries of all kite objects (in reverse order)
// note: due to the desired motion, every kite is a plane joint
object kites : array[plane]; new(kites); kites.size(max);

// 'topar' contains the topologies of all kite objects, because every kite can have several topologies,
// the type of 'topar' is array[array[topology]].
object topar : array[array[topology]]; new(topar); topar.size(max);

// initialization of 'kites' and 'topar' (i.e. the geometry and topologies of each kite)
object i : integer; object k : plane; object topo : topology;
object ta : array[topology]; object a : attribute;
i := 0;
while i < max do
begin
  new(k); new(topo); new(ta);
  // ta.size(1) indicates that each kite has only one topology
  ta.set(0, topo);
  // read geometrical and topological information for each kite
  k.read3dfile("kite", ta);
  // specify the color for each kite
  new(a); a.color(rainbow.get(i.mod(3))); topo.attribute(a);
  // specify that each kite is re-ordered using only its color (i.e. no illumination
  // calculation is performed)
  topo.illuminationmode(0);
  // register the topology of each kite at the camera; as a result the camera will
  // render these topologies
  cam.addtopology(topo);
  // store geometrical and topological information in 'kites' and 'topar'
  kites.set(i, k); topar.set(i, ta);
  i := i.plus(1);
end;

// initialization of the motion parameters of each kite

//
object W : geometry; new(W);
object t0 : vector; new(t0); t0.init(1,0,0);
object t1 : vector; new(t1); t1.init(0,1,0);
object p : point; new(p); p.init(0,0,0); W.addpoint(p);
i:=0;
while i.lt(max) do
begin
  k=kites.get(i);
  // each kite is initially positioned at the origin
  // Each kite is allowed to move in the Z=0 plane, specified by geometry W and index 0
  // A kinematics 'step' will result in a translation over (2,2,0) and a rotation
  // around the Z-axis with 2 degrees
  k.init(2.0, 2.0, t0, t1, W, 0, 2.0);
  // velocity constraint (for inverse kinematics)
  // 0: translation velocity bounded by -20 and 20
  // 1: angular velocity bounded by -10 and 10
  k.velocity(0,-20,20); k.velocity(1,-10,10);
  // acceleration constraint (for inverse kinematics)
  // 0: translation acceleration bounded by -10 and 10
  // 1: angular acceleration bounded by -5 and 5
  k.acceleration(0,-20,20); k.acceleration(1,-5,5);
  // the sight-point of each kite is its point with index 0 (note: an inverse kinematics
  // 'follow' tries to move the kite in such a way that its sight-point reaches a
  // designated target-point)
  k.sight(k,0);
  // the sight.aim-point of each kite is its point with index 4 (note: an inverse kinematics
  // 'follow' tries to rotate the kite around its sight-point in such a way that the angle
  // between the sight.aim-point, the sight-point, and a designated target.aim-point is minimized)
  k.sight.aim(k,4);
  // the target-point of each kite is the point with index 2 of the next kite in array 'kites'
  k.target(kites.get(i.plus(1).mod(max)),2);
  // the target.aim-point of each kite is the point with index 0 of the next kite in array 'kites'
  k.target.aim(kites.get(i.plus(1).mod(max)),0);
  // the rotation of each kite is performed before its translation
  k.translatefirst(false);
  i := i.plus(1);
end;

// ******************************************************************************
// initialization of a 3-D cursor
// ******************************************************************************
object cur:cursor; new(cur);
  // the cursor is attached to camera 'cam', meaning that the cursor can be used to control the
  // animation via the cursor associated with 'cam'
cur.init(cam);
  // the cursor is placed at the (0,0,0)
cur.position(p);
7.3. Script

// the cursor is allowed to move in the plane perpendicular to the Z-axis
cur.main(2); cur.construction(false);

/***********************************************
// definition of class 'mover' which performs the continuous inverse kinematics motion of all kites;
// class 'mover' also implements the direct manipulation of the first kite
/***********************************************

class mover

// class 'mover' has to handle events, therefore it has inherited from class 'responder'

inherit responder()

public

// initialization of event-handling
init() : void;

// continuous inverse kinematics motion of all kites
motion() : void;

private

// position of the 3-D cursor at the last event
prev_p : point;

// indices of target and target.aim in geometry W
t, ta : integer;

implementation

init() : void;

var mouse : mouse_event;

begin

// add target point and target.aim point to geometry W
new(prev_p); t := W.addpoint(prev_p);
new(prev_p); ta := W.addpoint(prev_p);

// create a new mouse.event
new(mouse); mouse.motion(true);

// 'mover' is only interested in motion events
// and only if they are generated in canvas 'can'
mouse.add(can);

// note: method watch is inherited privately, therefore a prefix is needed
responder:watch(mouse);

// 'prev_p' is initialized with the current position of the cursor
prev_p := cur.location();
end;

slave:eventhandler(e: event) : void

// method 'eventhandler' is invoked automatically whenever
// a motion event occurs in canvas 'can'

var me : mouse_event; p, pp : point; k : plane;

begin

// (checked) assign event 'e' to mouse_event 'me'; this assignment will always
// succeed because 'mover' is only interested in mouse movements
me := e;

// the first kite only follows the 3-D cursor if the mouse was moved while button 0
// was pressed
if me.mouse().button_down(0)
begin
    // 'W' is the first kite
    k := kites.get(max.minus(1));
    // 'W' is the current position of the 3-D cursor 'cur'
    p := cur.position();
    // store this position in geometry 'W' at position '1'
    W.getpoint(t).init(p.get(0), p.get(1), p.get(2));
    // the new target of the first kite is the current position of cursor 'cur'
    k.target(W, t);
    // the first kite should be aimed in the same direction as the last cursor motion;
    // therefore the target.aim point can be computed by adding the displacement
    // of the cursor (p.prev.p) to the current position of the cursor
    pp := p.plus(p.minus(prev.p));
    // store this position in geometry 'W' at position 'ta'
    W.getpoint(ta).init(pp.get(0), pp.get(1), pp.get(2));
    // the new target.aim of the first kite is stored in 'W' at position 'ta'
    k.target.aim(W, t);
    // store the position of the cursor
    prev.p := p;
end
end;
motion():void
var i : integer;
begin
    // never-ending repetition
    while true do
begin
    // the first kite moves first
    i := max.minus(1);
    // every kite has to move
    while i.ge(0) do
begin
    // kite with index 'i' performs its inverse kinematics motion
    kites.get(i).follow();
    i := i.minus(1);
end;
    // when every kite has performed its inverse kinematics motion, events are
    // handled, and subsequently a new frame is drawn
    synchronize;
end
end;
endclass;

```

Chapter 7. Sample Animation

if me.mouse().button_down(0)
begin
    // 'W' is the first kite
    k := kites.get(max.minus(1));
    // 'W' is the current position of the 3-D cursor 'cur'
    p := cur.position();
    // store this position in geometry 'W' at position '1'
    W.getpoint(t).init(p.get(0), p.get(1), p.get(2));
    // the new target of the first kite is the current position of cursor 'cur'
    k.target(W, t);
    // the first kite should be aimed in the same direction as the last cursor motion;
    // therefore the target.aim point can be computed by adding the displacement
    // of the cursor (p.prev.p) to the current position of the cursor
    pp := p.plus(p.minus(prev.p));
    // store this position in geometry 'W' at position 'ta'
    W.getpoint(ta).init(pp.get(0), pp.get(1), pp.get(2));
    // the new target.aim of the first kite is stored in 'W' at position 'ta'
    k.target.aim(W, t);
    // store the position of the cursor
    prev.p := p;
end
end;
motion():void
var i : integer;
begin
    // never-ending repetition
    while true do
begin
    // the first kite moves first
    i := max.minus(1);
    // every kite has to move
    while i.ge(0) do
begin
    // kite with index 'i' performs its inverse kinematics motion
    kites.get(i).follow();
    i := i.minus(1);
end;
    // when every kite has performed its inverse kinematics motion, events are
    // handled, and subsequently a new frame is drawn
    synchronize;
end
end
endclass;

```
7.4 Frames

This section contains some frames of the animation that is specified in the previous section. The first 70 frames show how the animation evolves from its initial configuration: gradually the kites perform a circular motion. At frame 70 the 3-D cursor starts to move (the cursor is not shown in these frames). Immediately, the first kite starts to follow the cursor; consequently, the others will follow too. The cursor moves from the center (frame 70) to the bottom-left part of the screen (frame 100), makes a turn (frames 105-120), and moves to the top-right part of the screen (frame 160), where it stops. As can be seen from the frames 161-195 all kites gradually connect to each other and also get the same direction, thereby fulfilling the two conditions of the motion specification.

![Frames](image)
This animation runs on a Sun Sparc-Station 5, with a Solaris 2.4 operating system and ZX graphics accelerator, at 29 frames per second.
Chapter 8

Conclusions and Future Work

8.1 Applications and Conclusions

The work presented in this thesis is based on the observation that in current animation and simulation systems the animation tasks are tightly coupled to the other tasks of an application, e.g. the actual simulation calculations. As a result, programs for such systems tend to be complex because of the interleaving of the various tasks. Furthermore, the usually complicated motion calculations of the animation tasks will slowdown the performance of the system.

Chapter 2 introduces an animation system, the Generalized Display Processor (GDP), that does take into account this decoupling of animation tasks and other tasks of an application. Of course there has to be some communication from the application to the GDP. For this purpose the script-language Looks is designed (Chapter 3).

8.1.1 Applications

The combination of GDP and Looks has been used in context of the DenK project (Section 1.1) to implement Winograd's SHRLU [Win72], a program that can reason about a world consisting of geometrical objects. In contrast to SHRLU, the geometrical objects in the DenK system may have autonomous behaviour. In this example the GDP is used to model and visualize the world; the application of the GDP, i.e. the DenK cooperative assistant, may send commands (e.g. "translate cube c1 to position p1 in 5 seconds") or questions (e.g. "how many cubes are blue?") to the GDP. Recently, a more involved application domain for the DenK project has been realized: the development of a multi-modal interface for an electron microscope [BF95]. For this application, the GDP has to simulate the geometric attributes (lens positions, slits, diaphragms in the microscope), as well as the geometric boundary of the electron beam as it propagates through the lens system. In this simulation, both visualisation and interaction were implemented by the GDP.

Another environment where the GDP and Looks have been used is the visualization of distributed algorithms, in particular communication protocols. In this context the application is a simulator for distributed algorithms [Gli94] that uses the GDP to visualize their behaviour [Sch94]. Furthermore, the interaction facilities of the GDP are used to control the simulation, e.g. to specify that simulated processors have crashed, that simulated packets are lost, etc.
The GDP and Looks have also been used as a system for modeling features, which are defined as physical parts of a product that are mappable to generic shapes, and that have functional significance [KDB95]. Examples of such features are holes, slots, stiffeners. In Looks, generic features are specified as classes, feature instances as objects. Furthermore, a hierarchy of generic features can be built, and extended run-time, using the GDP interpreter. For this modeling system system-classes for feature-constraints and feature-shapes have been added to the GDP. Currently, the GDP is used as an environment to model railroad yards. This environment is to be used in the context of an introductory course in technical applications of computer science at the Eindhoven University of Technology [SO].

8.1.2 Conclusions

The experiments with the GDP and Looks indicate that the decoupling of animation tasks and other tasks of an application is a fruitful approach in a variety of circumstances. Furthermore, these experiments also indicate that a script-language that is based on object-orientation in combination with quasi-parallelism is very well suited for specifying the animation tasks.

Another result of these experiments is that the performance of the GDP prototype is in some cases insufficient. This is mainly caused by the fact that several design decisions that underlay the current implementation of the GDP emphasised on functionality and a clean programming style rather than utmost performance. Profile information of the GDP reveals that much time is spent on constructors and destructors of C++ objects, therefore implementing a smart allocation strategy for these objects will improve the performance of the GDP considerably. The profile information also indicates that the simple mark-and-sweep garbage collection mechanism, as presented in Section 5.3, causes only a small overhead.

As explained in Section 3.5, Looks requires that all public inherited features are named explicitly in the class definition of a descendant in order to facilitate 'self-documenting' class. The rationale behind the inclusion of this requirement was that the estimated overhead of naming the public inherited feature-names was small compared to the rest of a class-definition. It turns out that for several classes of the class library from Chapter 6 this is not true. Consequently, this requirement may be absent in a next version of Looks.

The object-oriented implementation of abstract syntax trees and their evaluation functions, as presented in Section 5.2, has proven to be very useful for implementing an interpreter. Especially in combination with run-time library linking, it provides for a simple mechanism to extend the functionality of an interpreter.

8.2 Future Work

8.2.1 Multi-Processor System

As presented in Section 3.3.4, the concurrency mechanism of Looks is based on multi-programming, i.e. on a single-processor system. An interesting direction for future work consists of adapting the GDP to a multi-processor system in order to improve the
8.4. Future Work

processing power of the GDP. Naturally, this may have some effect on the semantics of Looks. Especially because it would allow for real parallel execution of asynchronous messages.

But even without parallel execution of asynchronous messages, a multi-processor system may be useful for the GDP. For example, in a system containing 3 processors, the various elements of the "main loop" of the GDP can be attributed to the processors in the following way:

1. "process asynchronous messages"
   "event handling";
   "restore WL-invariants";
   "dynamics";
   "parsing";

2. "rendering";

3. "garbage collection";

The first processor deals with parsing and interpreting a Looks script, the second processor renders the various frames, and the third processor collects the garbage created by the interpretation. Obviously, processors 1 and 2 have to be synchronized: when processor 1 has executed one step of an animation, processor 2 can start to render the corresponding frame, while simultaneously processor 1 can start to execute the next step of the animation (this is similar to the technique of double-buffering). The third processor does not have to be synchronized with the other two: it can collect the garbage at its own pace.

8.4.2 Distributed Multi-User System

The widgets presented in Section 6.5 are not confined to one workstation. On the contrary: they can be shown on any workstation that can communicate with the GDP via the X-protocol. Furthermore, UI events generated on these remote workstations will be sent to the GDP. Consequently, the GDP supports visualization and direct manipulation of an animation at remote workstations.

However, direct manipulation is only one element of interactive animation. The GDP as described in this thesis does not handle direct modification from multiple applications. One way to facilitate such a multiple direct modification mechanism is by introducing a switch-box. This switch-box acts as an intermediate between the GDP and the various applications: it merges the input and output channels of the applications into one input channel and one output channel for the GDP. In other words, a switch-box allows for using one GDP by multiple applications simultaneously. For example, this functionality is required for a situation where several users are cooperating in one animated domain.

One step further is the situation of cooperating GDPs, i.e. the GDPs are linked and send Looks fragments to each other. This can be achieved by adding sockets as input and output channels to the GDP, which has been shown recently to be feasible [HW].
8.4.3 Rendering

For applications where visual realism is not a crucial requirement, the quality of the rendering as currently supported by the GDP can be acceptable, in particular when using Gouraud shading in combination with multiple light-sources. However, the rendering capabilities can be improved by means of the following extensions [FDFH90]:

1. normals at polygons and vertices: those normals allow for rendering polygons as if they are curved surface segments (the current implementation of the GDP estimates plane equations of the polygons only, hence no smooth shading can be achieved).

2. texture mapping: an image is mapped onto a polygon, thus facilitating improved visual realism.

Furthermore, providing an adequately parameterized implicit surface as a topological primitive facilitates the animation of objects that have flexible surfaces, e.g. clothes and skin. However, due to their complexity, the canonical methods for implicit surfaces [WMW86, LC87] cannot be used in a real-time graphics system. But by making the generated polygon mesh parametrically dependent on a (moving) skeleton, implicit surfaces become feasible for real-time systems [Sta94].

8.4.4 Executable Media

In [BHN*95] it is proposed to extend Multi-Media (MM) to Executable Media (EM), similar to HotJava (see below). EM is characterized by the following aspects:

• **dynamic**: the content of an EM document can vary:
  1. by means of computation, e.g. an interactive simulation
  2. by means of (global) hyperlinks, e.g. retrieving actual weather conditions in a simulation

• **multi-directional**: several users can explore and modify the same application domain where they are real-time visually informed about each other's actions

• **3-D**: an EM document can contain a 3-D model of the geometry of the simulated scenery

• **programmable**: an EM player is freely programmable, changeable and extensible even during a running simulation. To this aim, an EM document contains standard MM-items but also code fragments for the EM player. Consequently, an EM-document may contain extensions or modifications to a 'standard' EM player in order to customize it for obtaining task-specific behavior.

Most of these aspects are already implemented in the GDP. Therefore, the GDP would be very suited as a basis for such an EM player.
8.4. Future Work

HotJava

In 1995 Sun introduced the Java language [Sun95], whose concepts are remarkably similar to those of Looks. Both languages are interpreted and both support object-orientation, concurrency, garbage collection, and dynamic linking. Furthermore, one of the design goals for these languages was simplicity. In Looks this was achieved by defining a new syntax and semantics, while Java took C++ as a basis and removed unnecessary complexities.

The Java language is demonstrated by means of the HotJava World-Wide Web (WWW) browser. Whereas conventional WWW browsers manipulate static HTML documents, HotJava adds dynamic behavior to these documents by means of Java fragments, e.g. to create animated icons.

It would be interesting to port the GDP classes to HotJava, thereby promoting the HotJava functionality to the current functionality of the GDP. This is a significant improvement for HotJava, since up to now it offers only very little support for interactive 3-D animation. It is to be expected, however, that the transparent handling of classes in the main-loop of the GDP (e.g. event handling, restore WL-invariants, dynamics, and rendering) has to be coded explicitly in Java.
Appendix A

Specification

A.1 Syntax

When specifying a programming language two aspects of the language have to be dealt with: its syntax, which describes the representation of programs, and its semantics, which describes the meaning of these programs. The syntax of a language can be subdivided even further: into concrete syntax, that specifies the textual representation of programs, and abstract syntax, that specifies the structural representation of programs. For example, if a language contains an addition operator, the concrete syntax specifies the textual representation of this operator (e.g. "++") and the way it is used (e.g. prefix, infix, or postfix), the abstract syntax, on the other hand, only specifies that the language contains an addition operator and that this operator requires two operands. It is the task of a parser to generate the structural representation — or abstract syntax tree — of a program written in the concrete syntax.

A formalism to specify the concrete syntax of a programming language was one of the research topics in the early days of computing science. In 1959 Backus introduced such a formalism [Bac59]. Later it was used to specify the concrete syntax of ALGOL 60. Since the report that describes this language was edited by Naur [Nau63], the formalism became known as Backus-Naur-Form (BNF). Nowadays, BNF and its extended version EBNF [Bac79] are widely used to specify the concrete syntax of programming languages. The concrete syntax of Licks is also specified by means of EBNF and can be found in Appendix B. An augmented version of this syntax with attributes and context conditions, e.g. concerning static type checking, is included in [Pee96].

The EBNF-formalism is also often used to describe the abstract syntax of programming languages. For example, this may lead to the following definitions of the concrete and abstract syntax of the selection statement:

concrete: <statement> ::= if <expression> then <statement> else <statement> fi

abstract: <statement> ::= if <expression> then <statement> else <statement>

Note that the abstract definition contains too much information: the key-words if, then, and else are not really necessary. In order to guarantee that an abstract syntax cannot contain this abundant syntactical information, another formalism is needed.
to specify an abstract syntax. Since the structural representation of a program may be considered as a tree, the abstract syntax should specify which nodes may be used and how these nodes may be used in combination with (sub-)trees to generate new trees. The well-known concepts (in the theory of algebraic specification [EM85]) of signature and term may be used to specify the abstract syntax and the structural representations, respectively.

First, $S$-sorted sets are defined; they are used in the definitions of both signature and term.

**Definition A.1 : $S$-sorted set**

Let $S$ be a set.

An $S$-sorted set $X$ is a collection of $|S|$ pairwise disjunct sets, where each set is indicated by means of an element of $S$: $X_s$ ($s \in S$).

$\Box$

The requirement that the sets of an $S$-sorted set $X$ should be pairwise disjunct can be specified formally:

$$\left( \forall s_1, s_2 \in S : s_1 \neq s_2 \Rightarrow X_{s_1} \cap X_{s_2} = \emptyset \right).$$

**Example A.2 : $S$-sorted set**

Let $S=\{a, b, c\}$.

Let $X_a=\{k, l\}$, $X_b=\{p, q, r\}$, $X_c=\emptyset$.

Let $Y_a=\{k, l\}$, $Y_b=\{l, m, n\}$, $Y_c=\emptyset$.

Then $X$ is an $S$-sorted set. However, $Y$ is not an $S$-sorted set because $Y_a \cap Y_b \neq \emptyset$.

$\Box$

Next, the definition of a signature is given:

**Definition A.3 : Signature**

A signature $\Sigma$ is a tuple $(S, \Gamma)$, with

- $S$ a set (known as the set of sorts of $\Sigma$)
- $\Gamma$ a $S^\times S$-sorted set (known as the set of operator sets of $\Sigma$)

$\Box$

The set of sorts of signature $\Sigma$ specifies the types of trees that can be generated by means of $\Sigma$, whereas the set of operator sets specifies the set of nodes that can be used. The set of operator sets is $S^\times S$-sorted which means that it is indexed by means of a tuple that consists of a list of sorts (known as the arity) and a single sort (known as the result sort). For example, the abstract syntax of the previous selection statement can be defined by means of signature $(\{expression, statement\}, \Gamma)$ as follows:

- selection $\in \Gamma(\{expression, \text{statement} \})$.

Furthermore, if the language would also contain a repeat-until statement with the following concrete syntax:

- $<\text{statement}> ::= \text{repeat} <\text{statement}> \text{ until } <\text{expression}>$,
then for a corresponding operator (say: repeat) holds:

\[
\text{repeat} \in \Gamma_{\langle \text{statement, expression}, \text{statement} \rangle}.
\]

Note that the order of the sorts in the arity is important (indeed, the arity is a list, not
a set or a bag): e.g. selection \( \notin \Gamma_{\langle \text{statement, expression, statement}, \text{statement} \rangle} \). Indeed, signatures facilitate abstract syntax definitions that do not contain abundant syntactical
information.

As stated above, the abstract syntax of a programming language defines the structural
representation (or abstract syntax tree) of programs written in the language. The next
definition specifies how this abstract syntax may be used to generate abstract syntax
trees.

**Definition A.4 : Terms**

Let \( \Sigma = (S, \Gamma) \) be a signature.

The \( S \)-sorted set \( T_\Sigma \) of terms of \( \Sigma \) is the smallest \( S \)-sorted set \( Y \) such that for
each operator \( \gamma \in \Gamma_{(\text{arity}, \text{result})} \) (arity \( \in S^* \), result \( \in S \)) and (operand) \( y_j \in Y_{\text{arity}(j)} \)
\((1 \leq j \leq \text{arity}()) \) the term \( \gamma(y_1, \ldots, y_n) \in Y_{\text{result}} \).

\( \square \)

In other words, given signature \( \Sigma = (S, \Gamma) \), \( (T_\Sigma)_s \) (with \( s \in S \)) is the set of all terms — or
abstract syntax trees — whose root has sort \( s \). For example, a repeat-until statement
that contains a selection statement corresponds with the term \( \in (T_\Sigma)_{\text{statement}} \):

\[
\text{repeat}(\text{selection}(e_1, s_1, s_2), e_2), \text{ with } e_1, e_2 \in (T_\Sigma)_{\text{expression}}, \text{ and } s_1, s_2 \in (T_\Sigma)_{\text{statement}}.
\]

Example A.5 shows how terms are created given a simple signature.

**Example A.5 : Terms**

Let \( \Sigma = (S, \Gamma) \) be a signature with

- \( S = \{A, B\} \)
- \( \Gamma_{(\langle\cdot, A\rangle)} = \{\text{leaf1, leaf2}\} \)
- \( \Gamma_{(\langle\cdot, B\rangle)} = \{\text{leaf3}\} \)
- \( \Gamma_{(\langle A,A, A\rangle, A)} = \{\text{node1}\} \)
- \( \Gamma_{(\langle B, B, A\rangle, B)} = \{\text{node2}\} \)
- \( \Gamma_{(\langle A, B, A\rangle, B)} = \{\text{node3, node4}\} \)
- \( \Gamma_{(\cdot, s)} = \emptyset, \text{ other } r \in S^*, s \in S \)

The recursion in the definition of \( T_\Sigma \) is started by creating terms using operators
whose arity is \( \langle\cdot, \cdot\rangle \):

- \( \{\text{leaf1, leaf2}\} \subseteq (T_\Sigma)_A \)
- \( \{\text{leaf3}\} \subseteq (T_\Sigma)_B \)

Next, these terms may be used as arguments of operators in order to create new
terms:

- \( \{\text{node1(leaf1, leaf2), node1(leaf1, leaf2), node1(leaf2, leaf1), node1(leaf2, leaf2)}\} \subseteq (T_\Sigma)_A \)
- \( \{\text{node2(leaf3, leaf3)}\} \subseteq (T_\Sigma)_B \)
- \( \{\text{node3(leaf1, leaf3, leaf1), node4(leaf1, leaf3, leaf1), node3(leaf1, leaf3, leaf2), \ldots}\} \subseteq (T_\Sigma)_B \)

Again, these newly constructed terms may be arguments, etc.

\( \square \)
However, as described in [Pee90], using an ordinary signature may result in terms that contain too much structural information because the arity of the operators of the signature is fixed. The next example illustrates this drawback by means of an abstract syntax tree of '1+1+1'.

Example A.6 : Signature

Let (S, Γ) be a signature with

- S = {expression}
- one ∈ Γ(< 2, expression>)
  add ∈ Γ(< expression, expression >, expression>)

The term add(add(one, one), one) is an element of (T_Γ)_{expression} and represents the abstract syntax tree of '1+1+1'.

This term consists of 5 operators, whereas only 4 operators were needed if the operator set of the signature contained a ternary operator:

add3 ∈ Γ(< expression, expression, expression >, expression>)

Now, the abstract syntax tree of '1+1+1' may be represented by the term:

add(add(one, one), one).

This idea may be generalized by using a uniform abbreviation mechanism to specify operators that only differ in the number of sorts in the arity.

Definition A.7 : Indexed Operator

Let Σ = (S, Γ) be a signature.

An indexed operator γ^i ∈ Γ(S^i, result) (i ≥ 0, s ∈ S, result ∈ S) denotes the set of operators: γ^0 ∈ Γ(< result >, result), γ^1 ∈ Γ(< s, result >, result), γ^2 ∈ Γ(< s, s, result >, result), etc.

Naturally, definition A.7 may be extended by using multiple indices. For example:

γ^{i,j} ∈ Γ(S^{i,j}, result) (i ≥ 0, s_0 ∈ S, s_1 ∈ S, result ∈ S)

denotes the set of operators: γ^0,0 ∈ Γ(< result >, result), γ^0,1 ∈ Γ(< s_0, result >, result), γ^0,1,i ∈ Γ(< s_0, s_1, result >, result), etc. Indexed operators are used in the abstract syntax of Looks (Appendix C) at various places: e.g. to specify that a block-statement may contain any number of statements. Multiple indexed operators are used too: e.g. in the selection operator. It could be argued that:

- selection ∈ Γ(< expression, statement, statement >, statement>)

However, since most statements that are used in a selection statement will be block-statements, the selection operator would be used mainly in the following construction:

selection(e, block_statement^1(s_1, ..., s_n_1), block_statement^2(s_2_1, ..., s_2_n_2)),

with n_1, n_2 ≥ 0, e ∈ (T_Γ)_{expression}, s_1, ..., s_n_1, s_2_1, ..., s_2_n_2 ∈ (T_Γ)_{statement}

In order to minimize the total number of operators in a term, the abstract syntax of Looks has adopted the following selection operator:

3In the remainder of this thesis the parentheses of terms that do not have sub-terms will be omitted, e.g. leaf instead of leaf().
A.2. Semantics

\[ \text{selection}^{i,j} \in \Gamma_{\langle \text{expression} \rangle, \langle \text{statement}^{i,j} \rangle, \langle \text{statement} \rangle}, \ i, j \geq 0 \]

Now the term can be rewritten as: selection(e, s_{11}, ..., s_{1n}, s_{21}, ..., s_{2n}), which contains two operators less. The operators repetition and invocation are treated in a similar way.

Example A.8: Natural numbers

Let \((\{\text{expression}\}, \Gamma)\) be a signature. Natural numbers may be included in this signature in two distinct ways:

1. \[ \text{zero} \in \Gamma_{\langle \text{expression} \rangle}, \text{succ} \in \Gamma_{\langle \text{expression}, \text{expression} \rangle} \]
2. \[ \text{zero} \in \Gamma_{\langle \text{expression} \rangle}, \text{one} \in \Gamma_{\langle \text{expression} \rangle}, \text{two} \in \Gamma_{\langle \text{expression} \rangle} \]

Example A.8 shows how natural numbers may be specified in a signature. Clearly, the first alternative introduces many operators in terms, e.g., the term that corresponds with value 3: \(\text{succ(succ(succ(zero)))}\). On the other hand, the second alternative results in terms that consist of one operator for each value. However, this alternative requires to provide the signature with operators for all values that may appear in a program. Since it would be an arduous task to supply these operators explicitly, again an abbreviation mechanism is needed.

Definition A.9: Parameterized Operator

Let \(C = \{c_i \mid i \geq 0\}\) be a set of constants.
Let \(\Sigma = (S, \Gamma)\) be a signature.
A parameterized operator \(\gamma \in \Gamma_{\langle \text{expression}, \text{result} \rangle} (c \in C, \text{result} \in S)\) denotes the set of operators: \(\gamma_{c_i} \in \Gamma_{\langle \text{expression}, \text{result} \rangle}, \gamma_{c_i} \in \Gamma_{\langle \text{expression}, \text{result} \rangle}, \) etc.

The abstract syntax of the statements and expressions of Loox, defined by means of a signature that contains indexed and parameterized operators, can be found in Appendix C.

A.2 Semantics

According to [Win93] three branches of semantics can be distinguished:

- **Operational semantics**: the meaning of a programming language is described by how it executes on an abstract machine. For example, the semantics of a repetition can be specified as follows:

  \[
  \langle c, s \rangle \rightarrow \text{false} \quad \text{and} \quad \langle c, s \rangle \rightarrow \text{true} \rightarrow \langle c', s' \rangle \quad \langle \text{while } b \text{ do } o \rangle \rightarrow \langle c', s' \rangle
  \]

  The first rule specifies that if expression \(b\) evaluates to false in state \(\sigma\), the
execution of the repetition does not change the state of the abstract machine. The second rule specifies that if expression $b$ evaluates to true in state $\sigma$ and the evaluation of statement $c$ changes the state of the abstract machine from $\sigma$ to $\sigma'$ and the evaluation of the repetition changes state $\sigma''$ to $\sigma'$, then the evaluation of the repetition changes the state from $\sigma$ to $\sigma'$. In other words, if the expression evaluates to false the state is not changed, if it evaluates to true then statement $b$ is executed followed by the repeated execution of the repetition.

- denotational semantics: the meaning of a programming language is described by means of partial functions on states. For example, the semantics of a repetition can be specified as follows:
  \[ C[[\text{while } b \text{ do } c]] = \bigcup_{n \in \mathbb{N}} C^n(\emptyset), \quad \text{with} \]
  \[ \Gamma : \mathcal{P}(\text{state} \times \text{state}) \rightarrow \mathcal{P}(\text{state} \times \text{state}) \text{ and} \]
  \[ \Gamma(\emptyset) = \{(\sigma, \sigma') : B[[b]] \sigma = \text{true} \land (\sigma, \sigma') \in \varphi \circ C[[c]] \} \cup \{(\sigma, \sigma') : B[[b]] \sigma = \text{false}\} \]
  A function $C$ is defined that maps the repetition to the least fixed point of function $\Gamma$, resulting in the set of tuples of all possible state-transitions which can be considered as a function that maps states to states. Function $B$ maps a boolean expression to a set of tuples that consist of a state and a boolean value -- this set can be considered as a function that maps states to boolean values.

- axiomatic semantics: the meaning of a programming language is described by giving its proof rules within a program logic. For example, the semantics of a repetition can be specified as follows:
  \[ (A \land B) \mathcal{C} (\neg A) \]
  \[ \{A\text{while } b \text{ do } c\} (A \land \neg A) \]
  This definition states that if condition $A \land b$ holds prior to the execution of $c$ and condition $A$ still holds after execution of $b$ (therefore condition $A$ is called an invariant of statement $b$), and condition $A$ holds prior to the execution of the repetition, then condition $A \land \neg b$ holds after the execution of the repetition. The conditions that hold prior to and after the execution of a statement are called the precondition and postcondition, respectively. The combination of precondition, statement, and postcondition, as used in the above definition, is known as a Hoare triple.

For every programming language these three formalisms are related: they all describe the meaning of the language. Therefore, relations exist between these formalisms. For example, the relation between operational and denotational semantics can be stated as:
  \[ C[[c]] = \{(\sigma, \sigma') : c, \sigma > \neg \sigma'\} \]
  This relation describes that the denotational semantics of statement $c$ can be considered as the abstraction of the operational semantics with respect to the state. The relation between the denotational and axiomatic semantics, given Hoare-triple \(\{A\}c\{B\}\) (for every condition $A$), can be stated as:
  \[ (\Delta \sigma : \sigma \in L : (\sigma \models A) \Rightarrow (C[[c]] \sigma \models B)) \]
  This relation describes that for every state $\sigma$ where condition $A$ holds, the execution of statement $c$ results in a state where condition $B$ holds. These states are related by means of the denotational semantics of statement $c$.\footnote{\(B[[b]] \sigma = \text{true} \) means that \((\sigma, \text{true}) \in B[[b]]\)}
These formalisms, however, are not redundant, but they are each apt for a specific goal: operational semantics is useful for describing the semantics of a language that has to be implemented by means of an interpreter, denotational semantics is useful for comparing the semantics of languages, and axiomatic semantics is useful for deriving and proving programs.

Since the GDP has to be controlled interactively by Looks, and since interactive operation is achieved intuitively using an interpreter, the GDP will contain an interpreter for Looks. Therefore, the semantics of Looks will be defined by means of operational semantics\(^3\). The abstract machine that is used in this operational semantics corresponds with the symbol-table of the GDP (in concert with its operations), which is described in Section 4.6. Consequently, in this approach the states that are used in operational semantics correspond to the states of the symbol-table.

Because the concrete syntax of Looks is not important for its (operational) semantics, the semantics is based on its abstract syntax, which is defined by means of the signature presented in Appendix C. The remainder of this section explains the operational semantics, whose formal definition can be found in Appendix D.

As can be seen in the explanation above, operational semantics is basically a function that maps (the abstract representation of) a statement and a state to a new state:

\[
\text{Eval} : \text{Codetree} \times \text{Symboltable} \rightarrow \text{Symboltable}
\]

Here type Codetree denotes the set of abstract representations of all Looks statements.

As mentioned before, these abstract representations are described by means of terms that are generated by signature \((S, \Gamma)\) (see Appendix C). Consequently, type Codetree can be specified as follows:

\[
\text{Codetree} = \left( \bigcup_{s \in S} : (T_s)_s \right)
\]

Type Symboltable denotes the set of possible states of the abstract machine that is used to describe the operational semantics (see Section 4.6).

However, to specify the semantics of expressions that occur in method bodies, information is needed on intermediate results and also information is needed on parameters, local variables, etc. of the method. This information is supplied to the Eval-function by augmenting the domain of the function with types Reference and Index, respectively (both types are explained in detail in [Pee95]).

In order to model returned values, quasi-parallelism, and blocking of the GDP, the codomain (or range) of function Eval has to be augmented too:

1. To return the value of an evaluated expression, the codomain should contain a field of type Reference.

2. To account for the effect of synchronization in the quasi-parallel form of concreteness in Looks, the Eval-function should indicate whether interpretation halted at a synchronization point or the complete codetree could be interpreted; therefore, the codomain should contain a field of type B.

\(^3\)As mentioned in Section 3.2 a formal specification of a language serves as an interface between the designer, the implementor, and the user of the language. The interface between the designer and the implementor may need more details than the interface between the designer and the user. The formal specification of Looks, as presented in this thesis, has been designed as the interface between designer and implementor. The denotational and axiomatic semantics of Looks remain subject of further research.
3. To account for the blocking of the GDP due to the interpretation of a statement, the codomain should contain a field of type $B$. For example, consider the Loox fragment `S0 ; S1`. Clearly, $S1$ is to be executed after $S0$, even if $S0$ is of the following form:

```plaintext
object i : integer;
i:=0;
while i<100 do
  i:=i+1;
synchronize;
end;
```

Obviously, this repetition halts after 100 frames. Thus, if statement $S1$ is received by the GDP before the repetition is terminated, $S1$ should be postponed until the GDP has been released by the repetition. Otherwise, the semicolon between $S0$ and $S1$, that is used to indicate sequential processing, gets different semantics when it is used in action-statements.

4. To account for the effect of a synchronization in the middle of a block-statement, the codomain should contain a field that contains the statements that have to be postponed until all quasi-parallel processes are synchronized. For example, consider the following Loox program:

```plaintext
begin
  "S1";
synchronize;
  "S2";
end
```

Obviously, statement $S2$ may be executed only after all quasi-parallel processes are synchronized. However, when this execution is resumed, all related reference-locations (i.e. local variables, parameters, and self) should have the values they had before the synchronize. Therefore, the remaining statements (in this case statement $S2$) and the actual values are stored together. Furthermore, since the synchronization may occur several messages 'deep', this particular information for all these methods has to be stored in a list. This results in type $SnlType$ (shorthand for synchronous-message-list-type), which should be a field of the codomain of function Eval:

```
SnlType = (ct:Codetree × invr:Index)*
```

5. To account for the effect of a synchronization within an asynchronous message, the list of postponed statements has to be stored: because the statements following an asynchronous message will always be executed (even if the asynchronous message contains a synchronization). For example, consider the following Loox program:

```plaintext
begin
  *object1.message();
  *object2.message();
end
```
A.2. Semantics

If the methods that are invoked by sending a message to object1 and object2 contain a synchronization, their evaluation returns a list of the postponed statements (see the previous discussion). However, since the message to object1 is asynchronous (note the asterisk before the message), the message to object2 is not postponed. Since both methods contain a synchronization, two lists of postponed statements are the result of the evaluation of this block-statement. This results in type AmlType (shorthand for asynchronous-message-list-type), which should be a field of the codomain of function Eval:

\[ \text{AmlType} = \text{SmlType}^* \]

The augmentation of the domain and the codomain of function Eval results in the following signature:\(^4\):

\[ \text{Eval} : \text{Codetree} \times \text{Reference} \times \text{Index} \times \text{SymbolTable} \]
\[ \quad \rightarrow \text{Reference} \times \text{B} \times \text{B} \times \text{AmlType} \times \text{SmlType} \times \text{SymbolTable} \]

The definition of function Eval can be found in Appendix D.2.1.

As mentioned before, if during the interpretation of a method body a synchronization is encountered, the call-list of that method is stored in a value of type SmlType. This call-list is related to a stack in ordinary function-calling (in Looks stacks for function-calls are a result of the recursive definition of the Eval-function). Stacks are used in the following way: every function-call pushes a new element on top of the stack (and function completion removes the top-element of the stack). However, during interpretation of Looks programs the stack information has to be saved only if the interpretation of a method is halted in a synchronization. Using an ordinary stack would result in too much stack operations. Therefore, this stack is only created when it is really needed. Consequently, it is not a stack in the traditional way, because it is filled in reverse order: the information on the 'deepest' function-call is stored first, whereas the information on the 'top-level' function-call is stored last. The difference between a call-list and a stack is demonstrated in Example A.10.

Example A.10 : Call-list and stack

Consider the following method-calls:

\[ f_0() \]
\[ f_1() \]
\[ f_2() \]
\[ f_3() // \text{terminates normally} \]
\[ f_4() // \text{halts in a synchronize} \]

Using an ordinary stack would result in the following stack history (top of stack is on the left):

\[ <> \]
\[ <f_0> \]
\[ <f_1,f_0> \]
\[ <f_2,f_1,f_0> \]
\[ <f_3,f_2,f_1,f_0> \]

\(^4\)The word 'signature' is also used in Computing Science to specify the domain and codomain of a function.
<f2,f1,f0>
<f4,f2,f1,f0>

Interpretation of method f4 halts in a synchronize, therefore a copy of 
<f4,f2,f1,f0> has to be stored, in order to be able to resume execution of this 
method after all asynchronous methods have reached their synchronize.

Due to the recursive character of the interpreter, this information can also be 
generated only when it is really necessary:
f4 halts in a synchronize
// control is given back to the evaluation of f2
// (due to recursive character of Eval)
f2 is not completed because f4 is halted
// control is given back to the evaluation of f1
f1 is not completed because f2 is not completed
// control is given back to the evaluation of f0
f0 is not completed because f1 is not completed

Now consider the following asynchronous messages:
*f0();
*f0();

Both messages will be halted in a synchronization (due to a call to method f4), 
therefore two stacks should be stored (resulting in a list of call-lists: AmlType).
In this example this list would contain: <f4,f2,f1,f0>,<f4,f2,f1,f0>>.

Asynchronous messages will never halt their caller (even if the interpretation of those 
messages is halted in a synchronize). This is demonstrated in the previous example 
where the second asynchronous message '*'f0()' is executed although the first was halted 
in a synchronize. Consequently, an asynchronous message should store its call-list in 
the list of call-lists, create an empty call-list and indicate that it is not halted in a 
synchronize (thus its caller may proceed its execution). This is formally specified in 
Appendix D on page 180.

If the call-list is resumed after all asynchronous methods have reached their synchro-
nize, the body of the method that contained the synchronization is resumed first. If 
the method does terminate completely (i.e. no synchronization is encountered), its 
caller is resumed. This process is specified by function Eval.sml:

Eval.sml : SmlType × Symboltable → AmlType × B × Symboltable

The codomain of this function contains fields of type AmlType and B because the 
evaluation of the method bodies may introduce new postponed asynchronous messages 
or may release a blocking CDP (i.e. if the CDP may execute new statements that are 
provided interactively). The definition of function Eval.sml can be found in Appendix 
D.2.2.

The previous paragraph introduced a function that evaluates the call-list of one post-
poned asynchronous message. However, during interpretation of a Looks program 
several of these postponed asynchronous messages may occur (information on these 
messages is stored in a value of type AmlType, as mentioned previously). A trivial 
definition of a function that evaluates a value of AmlType would consist of the evalua-
tion of each postponed asynchronous message that is stored in this list in order
A.2. Semantics

of appearance. This results in a deterministic way to evaluate the postponed asynchronous messages. If the order of these messages is not relevant, the function may evaluate the messages in random order. Consequently, several states (i.e., values of type Symboltable) may be reached. As in the previous case, the codomain contains values of type AmtType and B to account for new postponed asynchronous messages and release of a blocking GDP, respectively. This results in the following signature:

\[ \text{Eval}.\text{aml} : \text{AmtType} \times \text{Symboltable} \rightarrow \mathcal{P}(\text{AmtType} \times \mathcal{B} \times \text{Symboltable}) \]

The definition of function \text{Eval}.\text{aml} can be found in Appendix D.2.3. In this definition a value Max is used to discriminate between the deterministic and non-deterministic version (in the deterministic version the returned set is a singleton).

Finally, the semantics of a complete Looks program can be given. Such a program can be considered as a row of symbols of the concrete Looks syntax that is presented in Appendix B. If the set of terminal symbols is denoted by set \( V_T \), a Looks program is an element of \( (V_T)^* \). However, since the GDP is an interactive system, not only the textual representation of a program is important, but also the time at which the various program actions are received by the GDP. For example, consider the following Looks program:

\[
\text{object.start.move();} \\
\text{object.stop.move();}
\]

If the message \'object.stop.move()\' is received immediately after message \'object.start.move()\' the object will not have moved very long. However, if there is a delay between these messages, the object will have moved longer. To model this effect properly, the various actions of a Looks program are provided with an integer that indicates the delay between the release of the GDP and the reception of that action by the GDP. For this purpose, the following type is introduced:

\[ \text{ProgramType} = (\text{text}(V_T)^* \times \text{delay}: \mathbb{N})^* \]

Intuitively, the evaluation function for Looks programs has to map a value of type ProgramType \times Symboltable to a value of Symboltable. However, to obtain a recursive definition of this function, the domain of the function is augmented with a list of postponed asynchronous messages (initially this list is empty) and a boolean that indicates if the GDP is released (initially this boolean is TRUE). Furthermore, since the GDP is intended to be used to develop animations, it is not the value of the Symboltable after evaluation of the Looks program that we are interested in, but the row of (changing) Symboltable values, starting with the initial one. Because the evaluation of a value of type AmtType may be non-deterministic, the evaluation of a Looks program may be non-deterministic too. Therefore, the codomain consists of the set of all possible rows of Symboltables. This results in the following signature of the evaluation function:

\[ \text{Eval}.\text{program} : \text{ProgramType} \times \text{AmtType} \times \mathcal{B} \times \text{Symboltable} \rightarrow \mathcal{P}(\text{Symboltable}^*) \]

The definition of function \text{Eval}.\text{program} can be found in Appendix D.2.4. The working of this function is shown in the following pseudo-code (the numbers refer to the formal definition in the Appendix):

\[
\begin{align*}
\text{if } & \text{"no user input available anymore" } \land \text{ "no postponed asynchronous messages" (1)} \\
& \rightarrow \text{ "do nothing" } \\
\text{\hphantom{\text{if}}} & \text{"no user input can be received" (2)} \\
& \rightarrow \text{ "execute postponed asynchronous messages"}
\end{align*}
\]
□ "new user input has been received"
    \rightarrow "block the GDP and execute the new statement
    and the postponed asynchronous messages (if any)"
□ "new user input has not been received"
    \rightarrow "execute postponed asynchronous messages"
\fi

As shown in the formal definition of this function (case 3), the GDP is released by a codetree that is appended automatically (by means of a sequential blockstatement) to each statement that occurs in a action-statement. In case (4) there is at least one postponed asynchronous message: the one that contains the release codetree. To visualize the actual animation, a function Render is introduced that maps a Symboltable to ScreenContents (ScreenContents is a type that denotes the actual contents of the screen):

\text{Render} : \text{Symboltable} \rightarrow \text{ScreenContents}

The actual animation can be defined by means of a function that maps a Looks program (with delays) to the set of all possible rows of ScreenContents:

\text{Animation} : \text{ProgramType} \rightarrow \mathcal{P}(\text{ScreenContents}^*[\ast])$, with
\begin{align*}
\text{Animation}(\text{program})
&= (\{ \text{row} \in \text{Eval}_{\ast} \text{program}(\text{program},<>,\text{TRUE},\text{Initial},\text{Symboltable}) \\
  &\quad : (\ast i : 0 \leq i \leq |\text{row}| : \langle \text{Render}(\text{row}[i]) \rangle \} \\
\}
\end{align*}

To recapitulate the main topics of this appendix: the concrete syntax of Looks is specified by means of EBNF (Appendix B), the abstract syntax of Looks is specified by means of a signature (Appendix C), and the semantics of Looks is specified by means of operational semantics (Appendix D).
Appendix B

Concrete Syntax of Looks

(1)  <start>         ::= { <action> ';'}
(2a) <action>       ::= 'FORWARD' <heading>
(3)   | 'CLASS' <heading> <inherit_def>
(4)   | 'ENDCLASS'
(5)   | <statement>
(6)   | 'OBJECT' <identifier_list> ';' <type>
(7)   | <identifier> [']' <identifier_list> ']' 
(8)   | 'EXTERNAL' [ <identifier> ]
(9)   | <class_kind> [ <identifier> ]
(10)  | 'SIMPLE'
(11)  | 'OBJECT' [ <identifier> ]
(12)  | 'INHERIT' <type> [ '(' [ <features> ] ')']
(13)  | '{', 'type' [ '(' [ <features> ] ')'] }
(14)  | <feature> [ ',', <feature> ]
(15)  | <prefix> <identifier> <argumenttypes>
(16)  | ['>', <identifier> ]
(17)  | [', <prefix> <identifier> <argumenttypes> ']
(18)  | ['(', <type_list> ')']
(19)  | [',', <type> ]
(20)  | [',', <type> ]
(21)  | [',', <type> ]
(22)  | [',', <type> ]
(23)  | [',', <type> ]
(24)  | [',', <type> ]
(25)  | [',', <type> ]
(26)  | [',', <type> ]
(27)  | [',', <type> ]
(28)  | [',', <type> ]
(29)  | [',', <type> ]
(30)  | [',', <type> ]
(31)  | [',', <type> ]
(32)  | [',', <type> ]
(33)  | [',', <type> ]
(34)  | [',', <type> ]
(35)  | [',', <type> ]
(36)  | [',', <type> ]
(37)  | [',', <type> ]
(38)  | [',', <type> ]
(39)  | [',', <type> ]
(40)  | [',', <type> ]
(41)  | [',', <type> ]
(42)  | [',', <type> ]
(43)  | [',', <type> ]
(44)  | [',', <type> ]
(45)  | [',', <type> ]
(46)  | [',', <type> ]
(47)  | [',', <type> ]
(48)  | [',', <type> ]
(49)  | [',', <type> ]
(50)  | [',', <type> ]
(51)  | [',', <type> ]
(52)  | [',', <type> ]
(53)  | [',', <type> ]
(54)  | [',', <type> ]
(55)  | [',', <type> ]
(56)  | [',', <type> ]
(57)  | [',', <type> ]
(58)  | [',', <type> ]
(59)  | [',', <type> ]
(60)  | 

(6)   <argumenttypes> ::= [',', <prefix> <identifier> <argumenttypes> ']
(7)   <type_list>    ::= [',', <type> ]
(8)   <prefix>       ::= [',', <type> ]
(9)   <declarations> ::= <public> [ <private> ]
(10)  | <private> [ <public> ]
(11)  <public>       ::= 'PUBLIC' <attrs_or_methodheader>
(12)  | '{', <attrs_or_methodheader> }
(13)  <private>      ::= 'PRIVATE' <attrs_or_methodheader>
(14)  | '{', <attrs_or_methodheader> }
(15)  <implementation> ::= 'IMPLEMENTATION'
(16)  | <method_def> [ '{', <method_def> }]
(17)  <attrs_or_methodheader> ::= <identifier_list> ';'
(18)  | <identifier> [ '(', <form_param_list> ')'] 
(19)  | <type>
\begin{align*}
<\text{form.param.list}> &::= [ <\text{param.kind}> | <\text{identifier}.list> :: ' ; ' <\text{type}> \\
& \qquad \{ '; ' | [ <\text{param.kind}> ] <\text{identifier}.list> :: ' ; ' <\text{type}> \} \\
<\text{param.kind}> &::= '\text{VAL}' \\
& \quad | '\text{VAR}' \\
<\text{method.def}> &::= [ <\text{prefix}> <\text{identifier}> \\
& \quad \{ (\{ [ <\text{form.param.list}> | ')' :: ' ; ' <\text{type}> \} \\
\quad <\text{method.def.rest}> \} \\
<\text{method.def.rest}> &::= [ <\text{loc.variables}> | <\text{block}.statement> \\
& \quad | # [ <\text{identifier}> ] \\
<\text{loc.variables}> &::= '\text{VAR}' <\text{identifier}.list> :: ' ; ' <\text{type}> \\
& \quad \{ '; ' <\text{identifier}.list> :: ' ; ' <\text{type}> \} \\
<\text{statement}> &::= '\text{SYNCHRONIZE}' \\
& \quad | '\text{NEW}' (\{ <\text{expression}> :: ')' \\
& \quad | '\text{RESULT}' (\{ <\text{expression}> :: ')' \\
& \quad | <\text{expression}> \\
& \quad | '\text{WHILE}' <\text{expression}> 'DO' <\text{statement}> \\
& \quad | '\text{IF}' <\text{expression}> 'THEN' <\text{statement}> \\
& \quad | '\text{ELSE}' <\text{statement}> ] \\
& \quad <\text{block}.statement> \\
<\text{block}.statement> &::= '\text{BEGIN}' <\text{statement}> \{ '; ' <\text{statement}> \} \\
& \quad '\text{END}' \\
<\text{expression}> &::= '\text{NULL}' \\
& \quad | '\text{SELF}' <\text{expression}.rest> \\
& \quad | ( <\text{expression}.begin> <\text{expression}.rest> \\
& \quad | ( <\text{expression}.begin> :: ' ; ' <\text{expression}> :: ')' \\
& \quad <\text{expression}.rest> \\
<\text{expression}.begin> &::= [ '+' | '-' ] <\text{number} > [ <\text{realrest}> ] \\
& \quad | '\text{'} ' <\text{character} > '\text{'} ' \\
& \quad | '\"' ' <\text{character} > '\"' \\
& \quad <\text{ancestor}> <\text{prefix}> <\text{identifier}> \\
& \quad \{ (\{ <\text{act.param.list}> :: ')' \} \\
<\text{act.param.list}> &::= [ <\text{type}> :: ' ; ' ] \\
<\text{expression}.rest> &::= [ ' ; ' <\text{identifier}> \{ (\{ <\text{act.param.list}> :: ')' \} ] \\
<\text{act.param.list}.begin> &::= [ <\text{expression}> \{ ' ; ' <\text{expression}> \} ] \\
<\text{identifier}.list> &::= [ <\text{identifier}> \{ ' ; ' <\text{identifier}> \} ] \\
<\text{type}> &::= [ <\text{identifier}> \{ ' ; ' <\text{type}> \} ] \\
<\text{Identifier}> &::= [ (\{ 'a' .. 'z' \} | ('A' .. 'Z')) \\
& \quad \{ (\{ '+' | '-' ) | (\{ '0' .. '9' \} | (\{ '0' .. '9' \} \\
& \quad <\text{realrest}> &::= [ ' ; ' <\text{number} > [ (\{ '+' | '-' ) | (\{ '0' .. '9' \} ] \\
<\text{number}> &::= ( '0' .. '9' ) \{ '0' .. '9' \} \\
<\text{character}> &::= '\"' ' \text{'} ' \"' ' \\
\end{align*}
Appendix C

Abstract Syntax of Looks

Signature $\Sigma = (S, \Gamma)$, with:

$S = \{ \text{STATEMENT, CASTED_EXPRESSION, EXPRESSION, SIMPLE_EXPRESSION} \}$

$\Gamma(\text{STATEMENT}, \text{STATEMENT})$
$= \{ \text{block}_i, \text{statement}_i \}, \quad i \geq 0$

$\Gamma(\langle , \text{STATEMENT} \rangle$
$= \{ \text{synchronize, release} \}$

$\Gamma(\langle \text{CASTED_EXPRESSION}, \text{STATEMENT} \rangle$
$= \{ \text{result} \}$

$\Gamma(\langle \text{EXPRESSION}, \text{CASTED_EXPRESSION}, \text{STATEMENT} \rangle$
$= \{ \text{assignment} \}$

$\Gamma(\langle \text{EXPRESSION}, \text{EXPRESSSION}, \text{STATEMENT} \rangle$
$= \{ \text{rev_assignment} \}$

$\Gamma(\langle \text{EXPRESSION}, \text{STATEMENT} \rangle$
$= \{ \text{synch_message, async_message, new} \}$

$\Gamma(\langle \text{EXPRESSION}, \text{STATEMENT} \rangle^*$
$= \{ \text{selection}^i \}, \quad i, j \geq 0$

$\Gamma(\langle \text{EXPRESSION}, \text{STATEMENT} \rangle^+$
$= \{ \text{repetition}^i \}, \quad i \geq 0$

$\Gamma(\langle \text{EXPRESSION}, \text{CASTED_EXPRESSION} \rangle$
$= \{ \text{cast}_{\text{ind}} \}, \quad \text{ind} \in \text{Index}^1$

$\Gamma(\langle , \text{EXPRESSION} \rangle$
$= \{ \text{null} \}$

$^1$Type 'Index' is used in the symbol-table manager to identify an entry in one of the tables [Pe9S].
\[ \Gamma(SIMPLE\_EXPRESSION,EXPRESSION) = \{ \text{expression}^i \}, \quad i \geq 0 \]

\[ \Gamma(<CASTED\_EXPRESSION\#<CASTED\_EXPRESSION\#SIMPLE\_EXPRESSION\#>,EXPRESSION) = \{ \text{compare}^i \}, \quad i \geq 0 \]

\[ \Gamma(<>,SIMPLE\_EXPRESSION) = \{ \text{object}_\text{ind}, \text{attribute}_\text{aip}, \text{parameter}_\text{pip}, \text{variable}_\text{vlp}, \text{integer}_\text{i}, \text{boolean}_\text{b}, \text{real}_r, \text{char}_c, \text{string}_s \}, \quad \text{ind} \in \text{Index}, \ aip,\text{pip, vlp} \in \text{IndexPair}^2, \ i \in \mathbb{N}, \ b \in \mathbb{B}, \ r \in \mathbb{R} \]

\[ \Gamma(<CASTED\_EXPRESSION,EXPRESSION>) = \{ \text{invocation}_\text{eip}, \text{ancestor}_\text{invocation}_\text{ind,si,imp} \}, \quad i \geq 0, \ \text{mip} \in \text{IndexPair} \]

\[ \Gamma(r,s) = \emptyset, \quad \text{other } r \in S^*, \ s \in S \]

\[ ^2 \text{Type 'IndexPair' is used in the symbol-table manager to identify an entry in a table that is accessible by means of one indirection [Pee95]} \]
Appendix D

Semantics

D.1 Type Definitions

\[
\text{Codetree} = (\bigcup_{s \in S} (\text{T}_E)_s) \\
\text{SmlType} = (\text{ct: Codetree} \times \text{invnr: Index}^1)^* \\
\text{AmiType} = \text{SmlType}^* \\
\text{ProgramType} = (\text{text: (V_T^2)^*} \times \text{delay: N} )^*
\]

D.2 Evaluation Functions

D.2.1 Eval

\[
\text{Eval} : \text{Codetree} \times \text{Reference}^3 \times \text{Index} \times \text{Symboltable} \\
\rightarrow \text{Reference} \times \text{B} \times \text{B} \times \text{AmiType} \times \text{SmlType} \times \text{Symboltable}
\]

\[
\text{Eval}(\text{ct}, \text{result}, \text{invnr}, \text{st}) = \\
\text{if } \text{ ct: block statement}^*(s_1, \ldots, s_n) \\
\rightarrow \text{Eval statement list}(\langle s_i \rangle_{i=1}^n, \text{result}, \text{invnr}, \text{st}), \\
\text{ with } n \geq 0, s_i \in (\text{T}_E)\text{STATEMENT } (1 \leq i \leq n) \\
\text{The sub-terms } (s_i) \text{ of the block-statement are stored in a list, which is evaluated by function Eval statement list.}
\]

\[
\Box \text{ ct: result (ce)} \\
\rightarrow \text{Eval(ce, Self)(invnr, st), invnr, st),} \\
\text{ with } ce \in (\text{T}_E)\text{CASTED_EXPRESSION} \\
\text{The evaluation of the result statement consists of the evaluation of its (cast) expression. Since the expression of a result statement (which occurs in the method-body of an object) may begin with attributes of that object, a reference to that object is supplied to the evaluation of the expression (this reference is created by function Self).}
\]

\[\text{index}^1\text{Type 'Index' is used in the symbol-table manager as an index to a block that contains information on a method invocation, e.g. the values of arguments, local variables, etc. [Pee95]}\]

\[\text{V}_T^2\text{is the set of terminal symbols of the (concrete) Looks syntax that is presented in Appendix B.}\]

\[\text{Reference}^3\text{Type 'Reference' is used in the symbol-table manager to store information on object-references or reference-locations[Pee95]}\]
\(\text{ct::assignment}(e,cc)\)
\[
\rightarrow (\text{result}, \text{FALSE}, \text{FALSE}, \text{aml}_1 \oplus \text{aml}_2, \langle \rangle, \text{Assign}(r_1, r_2, st_2)),
\]
with \(e \in (T_c)\text{expression}\), \(cc \in (T_c)\text{casted-expression}\)
\[
(r_1, \text{synch}_1, \text{release}_1, \text{aml}_1, \text{smi}_1, st_1) = \text{Eval}(e, \text{Self}(\text{invnr}, st), \text{invnr}, st)
\]
\[
(r_2, \text{sync}_2, \text{release}_2, \text{aml}_2, \text{smi}_2, st_2) = \text{Eval}(cc, \text{Self}(\text{invnr}, st), \text{invnr}, st_1)
\]
\text{Property}^4: \neg \text{synch}_1 \land \neg \text{sync}_2 \land \neg \text{release}_1 \land \neg \text{release}_2 \land \text{smi}_1 = \langle \rangle \land \text{smi}_2 = \langle \rangle
\]

First, the left-hand expression is evaluated (which results in \(r_1\), a reference-location). Second, the right-hand expression is evaluated (which results in \(r_2\), a reference-location or object-reference). Third, \(r_2\) is copied into \(r_1\) by means of function \text{Assign}. Furthermore, an assignment statement cannot change the value of intermediate results (of expressions or method-bodies), therefore argument 'result' is returned unchanged. Since the expression cannot be halted in a synchronise, the boolean value \text{FALSE} and an empty (smi-) list are returned. Also, the assignment statement cannot release the GDP, therefore the boolean value \text{FALSE} is returned. Finally, the evaluation of the expressions may have introduced new asynchronous messages that are halted in a synchronise; the corresponding information (i.e. smi) is returned jointly by means of a concatenation.

\(\text{ct::rev_assignment}(e,cc)\)
\[
\rightarrow (\text{result}, \text{FALSE}, \text{FALSE}, \text{aml}_1 \oplus \text{aml}_2, \langle \rangle, \text{RevAssign}(r_1, r_2, st_2)),
\]
with \(e, cc \in (T_c)\text{expression}\)
\[
(r_1, \text{synch}_1, \text{release}_1, \text{aml}_1, \text{smi}_1, st_1) = \text{Eval}(e, \text{Self}(\text{invnr}, st), \text{invnr}, st)
\]
\[
(r_2, \text{sync}_2, \text{release}_2, \text{aml}_2, \text{smi}_2, st_2) = \text{Eval}(cc, \text{Self}(\text{invnr}), \text{invnr}, st_1)
\]
\text{Property}: \neg \text{synch}_1 \land \neg \text{sync}_2 \land \neg \text{release}_1 \land \neg \text{release}_2 \land \text{smi}_1 = \langle \rangle \land \text{smi}_2 = \langle \rangle
\]

\(\text{ct::synch_message}(e)\)
\[
\rightarrow (\text{result}, \text{synch}, \text{FALSE}, \text{FALSE}, \text{aml}, \text{smi}, st')
\]
with \(e \in (T_c)\text{expression}\)
\[
(r, \text{synch}, \text{release}, \text{aml}, \text{smi}, st') = \text{Eval}(e, \text{Self}(\text{invnr}, st), \text{invnr}, st)
\]
\text{Property}: \neg \text{release}

The evaluation of the synchronous message consists of the evaluation of its expression. Since this expression cannot release the GDP, the boolean value \text{FALSE} is returned. Furthermore, an synchronous message cannot change the value of intermediate results, therefore argument 'result' is returned unchanged. Finally, the evaluation of the expression may be halted in a synchronise, consequently the values of 'synch' and 'smi' of this evaluation are returned unchanged (indicating that the synchronous message may have been halted in a synchronise).

\(\text{ct::async_message}(e)\)
\[
\rightarrow (\text{result}, \text{FALSE}, \text{FALSE}, \text{aml} \equiv < \text{smi}, \langle \rangle , st')
\]
with \(e \in (T_c)\text{expression}\)
\[
(r, \text{synch}, \text{release}, \text{aml}, \text{smi}, st') = \text{Eval}(e, \text{Self}(\text{invnr}, st), \text{invnr}, st)
\]
\text{Property}: \neg \text{release}

The evaluation of the asynchronous message also consists of the evaluation of its expression. However, in contrast to the synchronous message, an asynchronous message cannot be halted in a synchronise. Therefore, the boolean value \text{FALSE} and an empty (smi-) list are returned. Still, the evaluation of the expression could have been halted in a synchronise. In this case, 'smi' contains information on the halted expression and is stored in the list of asynchronous messages that are halted in a synchronise (i.e. this list is extended with one element: the current asynchronous message).
D.2. Evaluation Functions

□ ct::selection$^{m,n}(e,s_1,\ldots,s_m,s_2,\ldots, s_n)$
  → if GetBoolean$(r_1, st_1)$
    → (r$_2$, sync$_1$, release$_1$, aml$_1@aml_1$, sml$_1$, st$_2$),
    with (r$_2$, sync$_1$, release$_1$, aml$_2$, sml$_2$, st$_2$) =
    Eval.statemlist(<s$_1$, >$^m_{i=1}$, result, invnr, st),
  □ ~GetBoolean$(r_1, st_1)$
    → (r$_3$, sync$_1$, release$_1$, aml$_1@aml_1$, sml$_1$, st$_3$),
    with (r$_3$, sync$_1$, release$_1$, aml$_2$, sml$_3$, st$_3$) =
    Eval.statemlist(<s$_2$, >$^n_{j=1}$, result, invnr, st),

  with $m \geq 0$, $n \geq 0$, $e \in (T_E)EXPRESSION$, $s_1, s_2, j \in (T_E)STATEMENT$ ($1 \leq i \leq m$, $1 \leq j \leq n$)

  (r$_1$, sync$_1$, release$_1$, aml$_1$, sml$_1$, st$_1$) = Eval(e, Sel((invnr, st), invnr, st)

  Property: ~sync$_1 \land$ ~release$_1 \land$ sml$_1$ = <>

First, the expression $e$ is evaluated (which results in a reference to a boolean object). According to the value of this object, either statements $s_1$, or statements $s_2$, are evaluated (by means of function Eval.statemlist).

Finally, both the evaluation of the expression and the evaluation of the statements may introduce new asynchronous messages that are halted in a synchronize; therefore the corresponding information (i.e., aml$_1$) is returned jointly by means of a concatenation.

□ ct::repetition$^n(e,s_1,\ldots,s_n)$
  → Eval(selection$^{n+1,0}(e,s_1,\ldots,s_n,ct)>$, result, invnr, st)
  with $e \in (T_E)EXPRESSION$, $s_i \in (T_E)STATEMENT$ ($1 \leq i \leq n$)

First, the expression $e$ is evaluated (which results in a reference to a boolean object). According to the value of this object, either statements $s_1$, followed by the repetition itself are evaluated (in case the object represents TRUE) or the default values are returned (in case the object represents FALSE). In other words: the evaluation of a repetition statement is specified by means of unfolding the repetition.

□ ct::synchronize
  → (result, TRUE,FALSE,<>><, <> st)

The evaluation halts in a synchronize, therefore the corresponding field is set to TRUE. Furthermore, since no statements are postponed due to this synchronize yet, the (aml-) list is empty. Finally, this evaluation has not introduced new asynchronous messages, consequently the (aml-) list is empty too.

□ ct::new(e)
  → (result,FALSE,FALSE,aml,<>><, New(r$_1$, st$_1$)),
  with $e \in (T_E)EXPRESSION$

  (r$_1$, sync$_1$, release$_1$, aml$_1$, sml$_1$, st$_1$) = Eval(e, Sel((invnr, st), invnr, st)

  Property: ~sync$_1 \land$ ~release$_1 \land$ sml$_1$ = <>

First, the expression is evaluated (which results in a reference-location). By means of function New a new object is created and an object-reference to this object is stored in the reference location.

□ ct::expression$^n(se_1,\ldots,se_n)$
  → Eval.simpleexpressionlist(<$se_i$, >$^n_{i=1}$, result, invnr, st),
  with $n \geq 0$, $se_i \in (T_E)SIMPLE_EXPRESSION$ ($1 \leq i \leq n$)

The simple expressions (se$_i$) of the expression are stored in a list, which is evaluated by function Eval.simpleexpressionlist.
\[\text{ct::compare}(c_1, c_2, s_1, \ldots, s_n) \rightarrow (r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1, s_{\text{ml}1}}, s_1, s_2)\]
with \(n \geq 0, c_1, c_2 \in (T_\text{E})\text{CASTED\_EXPRESSION}\), \(s_i \in (T_\text{E})\text{SIMPLE\_EXPRESSION}\)
\((1 \leq i \leq n)\)

\((r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1, s_{\text{ml}1}}, s_1) = \text{Eval}(c_1, \text{Self}(\text{invnr}, st), \text{invnr}, st)\)
\((r, \text{sync} \_2, \text{release} \_2, a_{\text{ml}2, s_{\text{ml}2}}, s_2) = \text{Eval}(c_2, \text{Self}(\text{invnr}, st).\text{invnr}, st)\)
\((r, \text{sync} \_3, \text{release} \_3, a_{\text{ml}3, s_{\text{ml}3}}, s_3) = \text{Eval}(\text{simple expression list}(s, s_1, \ldots, s_n), \text{BoolenToValue}(\text{EqualValues}(r_1, r_2, s_2), \text{invnr}, st)\)\]

\text{Property:} \neg \text{sync} \_1 \land \neg \text{sync} \_2 \land \neg \text{release} \_1 \land \neg \text{release} \_2 \land \neg \text{release} \_3 \land \neg \text{sync} \_4 \land a_{\text{ml}4} = <> > \land a_{\text{ml}5} = <> >
\]

First, the left-hand expression is evaluated (which results in \(r_1\), an object-reference or a reference-location).
Second, the right-hand expression is evaluated (which results in \(r_2\)). Third, the function \text{EqualValues}\ is used
to determine whether \(r_1\) and \(r_2\) refer to the same object.
The result of this function is a boolean which is
transformed by means of function \text{BoolenToValue} to a reference to the corresponding boolean object.
This reference is subsequently used as intermediate result for the evaluation of the simple expressions \(\{s_i\}\).

\[\text{ct::invocation}(c_1, \ldots, c_n) \rightarrow (r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1}, \ldots, a_{\text{ml}n}, s_{\text{ml}1}, s_1)\]
with \(n \geq 0, c_i \in (T_\text{E})\text{CASTED\_EXPRESSION}\)
\((1 \leq i \leq n)\)

\[s_0 = s \]
\[(r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1}, s_{\text{ml}1}, s_1) = \text{Eval}(c_1, \text{Self}(\text{invnr}, st), \text{invnr}, st - 1)\]
\((1 \leq i \leq n)\)

\[(r, \text{sync} \_2, \text{release} \_2, a_{\text{ml}2}, s_{\text{ml}2}, s_2) = \text{CallMethod}(\text{result}, \text{UnDefIndex}(s_{\text{ml}1}), \text{mip}, <r_1 >_{s_{\text{ml}1}}, s_{\text{ml}1}, s_1)\]
\[(r, \text{sync} \_3, \text{release} \_3, a_{\text{ml}3}, s_{\text{ml}3}, s_3) = \text{Eval}(\text{body}, \text{UnDefReference}(s_1'), \text{invnr}, st)\]

\text{Property:} \neg \text{sync} \_1 \land \neg \text{release} \_1 \land a_{\text{ml}1} = <> > (1 \leq i \leq n)\]

First, the arguments of the invocation are evaluated (from left to right). These arguments are cast expressions:
the results of the expressions are cast to the type of the corresponding formal parameter.
Second, function \text{CallMethod} fetches the body of the method that is indicated by IndexPair mop.
Since an undefined index is
used as second argument of function \text{CallMethod} the selected method will be a feature of the object that is
referred to by argument \(\text{result}'\) (note, this introduces dynamic binding).
Furthermore, function \text{CallMethod} also creates a new invocation-block that contains the actual arguments (i.e., \(r_i\)), reference-locations for the local variables of the method, and a reference to the object whose method is invoked (in case the value of self is
needed during the evaluation of the method-body). Finally, the fetched method-body is evaluated.

\[\text{ct::ancestor\_invocation}(c_1, \ldots, c_n) \rightarrow (r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1}, \ldots, a_{\text{ml}n}, s_{\text{ml}1}, s_1)\]
with \(n \geq 0, c_i \in (T_\text{E})\text{CASTED\_EXPRESSION}\)
\((1 \leq i \leq n)\)

\[s_0 = s \]
\[(r, \text{sync} \_1, \text{release} \_1, a_{\text{ml}1}, s_{\text{ml}1}, s_1) = \text{Eval}(c_1, \text{Self}(\text{invnr}, st), \text{invnr}, st - 1)\]
\((1 \leq i \leq n)\)

\[\text{result}' = \text{if EqualValues} (\text{result}, \text{NullToValue}(s_1')) \rightarrow \text{result}\]
\[\neg \text{EqualValues} (\text{result}, \text{NullToValue}(s_1')) \rightarrow \text{scale} (\text{result}, \text{ind}, s_1') \]
\[\text{fi}\]
\[(\text{body}, \text{new invnr}, st') = \text{CallMethod} (\text{result}', \text{ci}, \text{mip}, <r_1 >_{s_{\text{ml}1}}, s_{\text{ml}1}, s_1)\]
D.2. Evaluation Functions

\[(r, \text{synch}, \text{release}, \text{aml}, \text{sml}, st')\]
\[\text{Eval(body, UndefReference(st'), newinvnr, st')}\]

**Property:** \(\neg \text{synch} \land \neg \text{release} \land \text{sml} = \langle \rangle \ (1 \leq i \leq n)\)

This case also deals with method-invocation, but whereas in the previous case the fetched method was a feature of the object that was referred to by reference 'result', the method is now fetched from an ancestor-class (which is indicated by index 'i') of that object. In order to correctly evaluate this method-body, the object that is referred to by 'result' should be cast to this ancestor-class (the casting is indicated by index 'ind').

\[\Box \text{ct::cast}_{\text{ind}}(e)\]
\[\rightarrow (r, \text{FALSE}, \text{FALSE}, \text{aml}, \text{sml}, st')\]
with \(e \in (T_\text{c}) \text{expression}\)
\[\text{Eval(e, Self(invr, st), invar, st)}\]
\[r = \text{if EqualValues}(r', \text{NullToValue(st'))} \rightarrow r'\]
\[\Box \neg \text{EqualValues}(r', \text{NullToValue(st'))} \rightarrow \text{Scale}(r', \text{ind}, st')\]

**Property:** \(\neg \text{synch} \land \neg \text{release}\)

First, the expression is evaluated (which results in \(r'\)). In case \(r'\) differs from null, \(r'\) is cast (i.e. \(r'\) refers to a part of an object). The actual casting is performed by means of function \(\text{Scale}\), where argument 'ind' specifies the casting.

\[\Box \text{ct::release}\]
\[\rightarrow (\text{result}, \text{FALSE}, \text{TRUE}, \langle \rangle, \langle \rangle, \text{st})\]

The DOP is released, therefore the corresponding field in the result is set to TRUE. The other fields have a default value.

\[\Box \text{ct::object}_{\text{ind}}\]
\[\rightarrow (\text{AccessObject}(\text{ind}, st), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

This evaluation returns the reference-location that is indicated by index 'ind'.

\[\Box \text{ct::attribute}_{\text{aip}}\]
\[\rightarrow (\text{AccessAttr}(\text{result, aip, st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

This evaluation returns the reference-location of the attribute that is indicated by index 'aip' and is also a feature of the object that is referred to by reference 'result'.

\[\Box \text{ct::parameter}_{\text{pip}}\]
\[\rightarrow (\text{AccessParam}(\text{invar, pip, st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

This evaluation returns the reference-location of the parameter that is indicated by index 'pip' and that occurs in the invocation-block indicated by 'invar'.

\[\Box \text{ct::variable}_{\text{vip}}\]
\[\rightarrow (\text{AccessLocVar}(\text{invar, vip, st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

\[\Box \text{ct::integer}_{i}\]
\[\rightarrow (\text{IntegerToValue}(i, \text{st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

This evaluation returns a reference to the object that corresponds with integer i.

\[\Box \text{ct::boolean}_{b}\]
\[\rightarrow (\text{BooleanToValue}(b, \text{st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]

\[\Box \text{ct::real}_{r}\]
\[\rightarrow (\text{RealToValue}(r, \text{st}), \text{FALSE}, \text{FALSE}, \langle \rangle, \langle \rangle, \text{st})\]
\( \text{ct}:: \text{char_c} \\
\quad \rightarrow \text{CharToValue}(c, \text{ct}, \text{FALSE, FALSE, }\text{>, }\text{>, st}) \\
\text{ct}:: \text{string_s} \\
\quad \rightarrow \text{StringToValue}(s, \text{ct}, \text{FALSE, FALSE, }\text{>, }\text{>, st}) \\
\text{ct}:: \text{null_null} \\
\quad \rightarrow \text{NullToValue}(\text{st}, \text{FALSE, FALSE, }\text{>, }\text{>, st}) \\
\) 

This evaluation returns a null-reference.

\( \bar{s} \)

\textbf{Eval\_statementlist}

\textbf{Eval\_statementlist : Codetree* × Reference × Index × Symboltable} 
\quad \rightarrow \text{Reference × } \text{B × B × AmiType × SmiType × Symboltable}

\textbf{Eval\_statementlist(<ct, >_{n=1}, result, invnr, st)} 
\textbf{=} 
\textbf{if } n=0 
\quad \rightarrow (\text{result, FALSE, FALSE, }\text{>, }\text{>, st}) 
\textbf{with } n>0 
\quad \rightarrow \text{if } \text{synch_0} \rightarrow (\tau_0, \text{synch_0, release_0, aml_0, sml_0}) \text{ with } (\tau_0, \text{synch_0, release_0, aml_0, sml_0, st_0}, st_0) \text{ and } \text{Eval\_statementlist(}<ct, >_{n=1}, result, invnr, st_0)} 
\quad \text{fi, with } n>0, \text{ ct } ∈ (S)\text{STATEMENT } (1 ≤ i ≤ n)

This (recursive) function evaluates a list of statements in order of appearance. If the evaluation of one of these statements halts, synchronize the remaining statements are stored in a block-statement, which is then added to the smi-list (i.e. the list that contains information on the halted statements). Note that the returned value of an evaluated statement is passed to the next evaluation.

\textbf{Eval\_simpleexpressionlist}

\textbf{Eval\_simpleexpressionlist : Codetree* × Reference × Index × Symboltable} 
\quad \rightarrow \text{Reference × } \text{B × B × AmiType × SmiType × Symboltable}

\textbf{Eval\_simpleexpressionlist(<se, >_{n=1}, result, invnr, st)} 
\textbf{=} 
\textbf{if } n=0 
\quad \rightarrow (\text{result, FALSE, FALSE, }\text{>, }\text{>, st})
D.2. Evaluation Functions

\[ n > 0 \]
\[ \to (r_1, \text{sync}_1, \text{release}_1, \text{release}_0, \text{release}_1, \text{aml0} \oplus \text{aml}_1, \text{sm10} \oplus \text{sm1}_1, \text{st}_1) \]
with \((r_0, \text{sync}_0, \text{release}_0, \text{aml0}, \text{sm10}, \text{st}_0) = \text{Eval}(se_i, \text{result}, \text{invnr}, \text{st})\)
\[ (r_1, \text{sync}_1, \text{release}_1, \text{aml1}, \text{sm1}_1, \text{st}_1) = \text{Eval} \cdot \text{simpleexpressionlist}(<se_i >^n_{i=1}, \text{result}, \text{invnr}, \text{st}_0) \]

\[ \text{fi} \]
with \(n \geq 0\), \(ct_i \in (T_D)\text{SIMPLE_EXPRESSION (}1 \leq i \leq n)\)

This function evaluates a list of simple-expressions in order of appearance. In contrast to the previous function, the evaluation of simple-expressions cannot be halted in a synchronize. Again, the returned value of an evaluated simple-expression is passed to the next evaluation.

D.2.2 Eval_sml

\[ \text{Eval.sml} : \text{SmIType} \times \text{Symboltable} \to \text{AmlType} \times \text{B} \times \text{Symboltable} \]
\[ \text{Eval.sml}(<cb_i >^n_{i=1}, \text{st}) \]
\[ = \]
\[ \text{if} \ n=0 \]
\[ \to (<>., \text{FALSE}, \text{st}) \]
\[ \text{fi} \]
\[ \text{if} \ n>0 \]
\[ \to \]
\[ \text{if} \ \text{sync} \to (<\text{asm}1 \oplus <\text{cb}_i >^n_{i=2} \oplus <\text{aml}1', \text{release}_1, \text{st}_1) \]
\[ \text{fi} \]
\[ \to \]
\[ \text{release} \to (\text{aml}1', \text{stm}, \text{release}_1, \text{vrelease}_2, \text{st}_2). \]
\[ \text{fi} \]
\[ \text{with} \ (\text{aml}, \text{release}_2, \text{st}_2) = \text{Eval.sml}(<\text{cb}_i >^n_{i=3}, \text{st}_1) \]
\[ \text{fi} \]
\[ \text{with} \ (r, \text{sync}, \text{release}_2, \text{aml}, \text{sm}, \text{st}) \]
\[ = \text{Eval}(cb_i, ct, \text{UndefinedReference}(\text{st}), \text{cb}_1, \text{invnr}, \text{st}) \]

\[ \text{fi} \]
with \(n \geq 0\), \(cb_i \in (ct: \text{Codetree} \times \text{invnr: Index}) (1 \leq i \leq n)\)

This function evaluates a list of statements that are halted in a synchronize (as explained in Appendix A this list can be considered as a stack). If the evaluation of one of these statements halts in a synchronize, the halted statement together with the remaining statements are stored in a new sml-list. This sml-list is then returned in combination with the information on new asynchronous messages that are halted in a synchronize (i.e. aml_i).

D.2.3 Eval_aml

\[ \text{Eval.aml} : \text{AmlType} \times \text{Symboltable} \to \text{P}(\text{AmlType} \times \text{B} \times \text{Symboltable}) \]
\[ \text{Eval.aml}<\text{aml} >^n_{i=1}, \text{st}) \]
\[ = \]
\[ \text{if} \ n=0 \]
\[ \to \{(<>., \text{FALSE}, \text{st})\} \]
\[ \text{fi} \]
\[ \text{if} \ n>0 \]
\[ \to \]
\[ \{(u_j : 1 \leq j \leq \text{MAX} \leq n, (u_{\text{aml}, \text{release}_2, \text{st}_2} : (\text{aml}1', \text{release}_2, \text{st}_2) \in \text{Eval.aml}<\text{aml} >^n_{i=1, \text{st}}, \text{st}_1) \}
\[ \quad \text{with} \ (\text{aml}1', \text{release}_2, \text{st}_2) = \text{Eval.sml}(\text{aml}, \text{st}) \]
\[ \quad \text{fi} \]
\[ (\text{MAX}=1 \Rightarrow \text{deterministic scheduler}) \]
\[ \text{fi} \]
with \(n \geq 0\), \(\text{aml} \in \text{SmIType} (1 \leq i \leq n)\)
This function evaluates a list of aml-lists, where each aml-list is evaluated by means of function Eval.aml. If 'MAX' equals 1, the aml-lists are evaluated in order of appearance; otherwise, they may be evaluated in random order.

### D.2.4 Eval.program

**Eval.program**: ProgramType × AmlType × B × Symboltable → P(Symboltable')

\[
\begin{align*}
\text{Eval.program}(\langle sn_1 \rangle_{i=1}^{n}, \text{aml}, \text{released}, \text{st}) &=
\begin{cases}
\text{if } n=0 \land \text{aml}='>' \\
\rightarrow \{ '<>st' \} & (1)
\end{cases} \\
& \sqcup n>0 \lor \text{aml}=''< \\
& \rightarrow (\langle \lambda_{\text{aml}}, \text{release}_1, \text{st}_1 \rangle : (\text{aml}_1, \text{release}_1, \text{st}_1) \in \text{Eval.aml(aml',st')} \\
& \rightarrow (\langle \text{row} : \text{row} \in \text{Eval.program}(\text{pt'}, \text{aml}_1, \text{released}', \text{st}_1), \\
& \rightarrow <>'st'@\text{row} \rangle
\}
\end{align*}
\]

\[
\text{with: (aml',pt',released',st')} =
\begin{cases}
\text{if } \neg(n>0 \land \text{released}) \\
\rightarrow (\text{aml}, <sn_1, \rangle_{i=1}^{n}, \text{released'\text{Yrelease}_1}, \text{st}) & (2) \\
\sqcup n>0 \land \text{released} \land sn_1, \text{delay}=0 \\
\rightarrow (\text{aml} \oplus <\langle \text{block.statement}(ct, \text{release}), \text{UndefIndex(st')} \rangle > \\
\langle <sn_1, \rangle_{i=2}^{n}, \text{release}_1, \text{st}'' \\
\rangle, \text{with } (ct,st'') = \text{Parse}(sn_1, \text{text}, \text{st}) & (3)
\end{cases} \\
\sqcup n>0 \land \text{released} \land sn_1, \text{delay}>0 \\
\rightarrow (\text{aml}, <(sn_1, \text{text}, \text{sn}_1, \text{delay}=1)> \sqcup <sn_1, \rangle_{i=2}^{n}, \text{release}_1, \text{st}) & (4)
\end{cases}
\]

This function evaluates a Looks program, that is modelled by means of list of fragments and delays (to account for the interactive character of the GDP). Four cases can be distinguished:

1. All fragments have been processed and no asynchronous messages are halted in a synchronise: nothing happens.
2. No fragment can be received (either because all fragments have been processed, or because the GDP is blocked): the asynchronous messages that are halted in a synchronise are evaluated by means of function Eval.aml.
3. The GDP is released and the next fragment has been received: this fragment is evaluated in combination with the asynchronous messages that are halted in a synchronise (if any). Furthermore, the GDP will be blocked for a long as the evaluation of this fragment. It will be released again by the evaluation of the release-term that is appended to the fragment by means of a block-statement.
4. The GDP is released and the next fragment has not been received: the asynchronous messages that are halted in a synchronise (if any) are evaluated. Furthermore, the delay of the next fragment will be decremented.

\footnote{Parse is the function that analyses a Looks program; it stores the class-definitions and object-declarations of this program in the symbol-table, while the statements of this program are returned as terms of the term-algebra presented in this appendix. Function Parse has the following signature: Parse : (Vp)^n × Symboltable → Codetree × Symboltable}
Appendix E

Symbol-table Functions

This appendix describes functions of the symbol-table that are used in the formal semantics of Appendix D. A complete formal specification of the symbol-table can be found in [Pee95].

- **AccessAttr**: $Reference \times IndexPair \times SymbolTable \rightarrow Address$

  $addr := AccessAttr(ref,ip,st)$
  This function returns the reference-location of an attribute (that is indicated by IndexPair 'ip') of an object (that is referred to by 'ref').

- **AccessLocVar**: $Index \times Index \times SymbolTable \rightarrow Address$

  $addr := AccessLocVar(invnr, nr, st)$
  This function returns the reference-location of the local variable with index 'nr' of the invocation with index 'invnr'.

- **AccessObject**: $Index \times SymbolTable \rightarrow Address$

  $addr := AccessObject(oi, st)$
  This function returns the reference-location with index 'oi'.

- **AccessParam**: $Index \times Index \times SymbolTable \rightarrow Address$

  $addr := AccessParam(invnr, nr, st)$
  This function returns the reference-location of the parameter with index 'nr' of the invocation with index 'invnr'.

- **Assign**: $Address \times Value^2 \times SymbolTable \rightarrow SymbolTable$

  $st' := Assign(addr, val, st)$
  This function copies the object-reference 'val' into reference-location 'addr'.

---

1 Type Address is used in the symbol-table manager to store information on reference-locations [Pee95]

2 Type Value is used in the symbol-table manager to store information on reference-locations or object-references [Pee95]
Appendix E. Symbol-table Functions

- **BooleanToValue**: boolean × SymbolTable → Value

  \[
  \text{val} := \text{BooleanToValue}(b, st)
  \]
  This function returns a reference to the (Looks) boolean object that corresponds with boolean 'b'.

- **CallMethod**: Reference × Index × IndexPair × Reference* × SymbolTable → CodeTree × Index × SymbolTable

  \[
  (\text{codetree, invn, st'}) := \text{CallMethod}(\text{ref, ci, ip, aps, st})
  \]
  This function fetches the body of a method (that is indicated by IndexPair 'ip') of an object O (that is referred to by 'ref'). Furthermore, the values of the actual parameters are stored in an "invocation-block" that is referred to by index 'invn'. This block also contains reference-locations for the local variables of the method and it contains a — possibly cast — reference (self) to object O. In case the method is not implemented in O's type (say T) but in one of its ancestors (say S), reference self (that is stored in the block) is cast to the S-part of object O. Finally, index 'ci' indicates whether O's own body should be used ('ci'=UndefIndex(st)) or the body of an ancestor ('ci'≥0).

- **CharToValue**: character × SymbolTable → Value

  \[
  \text{val} := \text{CharToValue}(c, st)
  \]
  This function returns a reference to the (Looks) character object that corresponds with character 'c'.

- **EqualValues**: Reference × Reference × SymbolTable → boolean

  \[
  b := \text{EqualValues}(\text{ref0, ref1, st})
  \]
  This function indicates if 'ref0' and 'ref1' refer to (parts of) the same object.

- **GetBoolean**: Reference × SymbolTable → boolean

  \[
  b := \text{GetBoolean}(\text{ref, st})
  \]
  This function returns the boolean value that corresponds with the Looks boolean object (that is referred to by 'ref').

- **IntegerToValue**: integer × SymbolTable → Value

  \[
  \text{val} := \text{IntegerToValue}(i, st)
  \]
  This function returns a reference to the (Looks) integer object that corresponds with integer 'i'.

- **New**: Address × SymbolTable → SymbolTable

  \[
  \text{st'} := \text{New}(\text{addr, st})
  \]
  This function creates a new object and stores a reference to this object at reference-location 'addr'.
• **NullToValue**: $SymbolTable \rightarrow Value$

  \text{val} := \text{NullToValue(st)}

  This function returns a (Looks-) value null.

• **RealToValue**: $real \times SymbolTable \rightarrow Value$

  \text{val} := \text{RealToValue(r, st)}

  This function returns a reference to the (Looks) real object that corresponds with real ‘r’.

• **RevAssign**: $Address \times Value \times SymbolTable \rightarrow SymbolTable$

  \text{st}' := \text{RevAssign(addr, val, st)}

  This function checks whether ‘val’ refers to an object whose type conforms to the type of reference-location ‘addr’. If the types conform, a reference to this object is stored at ‘addr’; otherwise a null-reference is stored.

• **Scale**: $Reference \times Index \times SymbolTable \rightarrow Value$

  \text{val} := \text{Scale(ref, distance, st)}

  This function returns a reference ‘val’ that contains a (cast) reference to the object that is referred to by reference ‘ref’. The casting is specified by means of index ‘distance’ that indicates to which part of the object ‘val’ should refer.

• **Self**: $Index \times SymbolTable \rightarrow Value$

  \text{self} := \text{Self(invnr, st)}

  This function returns a reference to an object (say O), where ‘invnr’ indicates an "invocation-block" of one of O’s methods.

• **StringToValue**: $String \times SymbolTable \rightarrow Value$

  \text{val} := \text{StringToValue(s, st)}

  This function returns a reference to the (Looks) string object that corresponds with string ‘s’.

• **UndefIndex**: $SymbolTable \rightarrow Index$

  \text{ind} := \text{UndefIndex(st)}

  This function returns an undefined index.

• **UndefReference**: $SymbolTable \rightarrow Reference$

  \text{ref} := \text{UndefIndex(st)}

  This function returns an undefined reference.
Bibliography


Bibliography


Bibliography


Index

A
abstract syntax, 163
   Looks, 177
abstract syntax tree (AST), 63, 77
action, 25
actual generic type, 30
algorithmic animation, 5
aliasing, 27, 47
allocation, 49
ambient reflection model, 116
ancestor (inheritance), 33, 35
ancestor (in expression), 44
animation, 2
animation system, 3
   universal, 6
application, 17
array, 89
articulated body, 5
   Looks, 131
assignment, 50
AST, 63, 77, 83, 163
asynchronous message, 55
asynchronous message passing, 21
attribute, 116
attribute, 25
   access, 72
   declaration, 32
axiomatic semantics, 168
axis, 95

B
basic class, 27, 30, 86
basic object, 27
basic object-reference, 27
subitem expression, 41
block-statement, 48
boolean, 86

C
camera rendering, 122
camera, 120
canvas, 106
casting, 53
cell-animation, 4
char, 87
checked assignment, 53
class, 19, 27
   basic, 30
   definition, 28
   split, 28
   heading, 29
   implementation, 32
   recursive, 31
class-table, 67
   attribute-access, 72
   definition, 68
   inheritance, 69
   method-fetching, 70, 72
collection, 96
collision detection, 5
collision response, 5
color, 108
comments, 29
communication
   application and interpreter, 64
   application and parser, 64
   interpreter and application, 65
   interpreter and renderer, 64
   interpreter and symbol-table, 63
   interpreter and windowing system, 65
parser and application, 65
parser and interpreter, 63
parser and symbol-table, 63
renderer and graphics hardware, 65
renderer and symbol-table, 64
comparison expression, 46
compiled code, 59
compiled method fetching, 83
compiled object fetching, 82
concrete syntax, 163
Looks, 175
concurrency, 20
conditional critical region, 21
conformance, 34, 40
constraint, 144
constraints, 5
CPPObject, 82
cursor, 126
curved surfaces, 160
cylinder, 137

D

dangling reference, 58
DenK, 1, 157
denotational semantics, 168
descendant, 33
diffuse reflection model, 116
direct hit, 46
direct manipulation, 6
Looks, 123, 147
control, 126
selection, 123
multiple users, 67

direct modification, 6
display list (DL), 12
display processor (DP), 12
distributed multi-user system, 159
DL, 12
DP, 12
dynamic binding, 19, 53
dynamic, 144
dynamics, 5
inverse, 5
Looks, 144
implementation, 145

E

empty, 48
event handling, 103
event, 99
events, 99

key_event, 100
mouse, 102
mouse_event, 101
responder, 99
UI_event, 100

Executable Media, 160
expression, 40
basic object-reference, 41
characteristics, 47
comparison, 46
message, 43
null, 40
reference-location, 42
self, 41

F

feature, 26
declaration, 31
inheritance, 35
private, 26
public, 26
flat shading, 116
formal generic type, 30
forward kinematics, 5
Looks, 133
frame, 4
frame buffer, 12
function, 26

G

garbage collection, 58, 79
copying, 79
GDP, 80
mark-and-sweep, 79
reference-counting, 79
GDL, 13
GDP, 15
Generalized Display List (GDL), 13
Generalized Display Processor 0, 13
Generalized Display Processor 1, 14
genericity, 30
geometry, 108
goal-directed animation, 5
Gouraud shading, 116
Index

H

helix, 138
HotJava, 161

I

ifile, 91
illumination model, 118
implicit surfaces, 160
inbetweening, 4
  graph-oriented, 4
  script-oriented, 4
inheretance, 33, 35
  multiple, 33
  repeated, 34, 38
  single, 33
instance, 27
integer, 87
inter-object concurrency, 22
interactive animation, 6
interpreter, 66
inverse dynamics, 5
inverse kinematics, 5
  Looks, 134, 147
isomorph, 143
istream, 90

J

Java, 161

K

key-framing, 4
key.event, 100
kinematic, 132
kinematics, 5
  forward, 5
  inverse, 5
  Looks, 133

L

library
  creation, 84

linking, 59, 81
opening, 81
light, 117
light.ambient, 117
light.directional, 118
light.positional, 118
light.spot, 118
line, 114
linehinge, 145
list, 89
local variables, 33
Looks, 15
  requirements, 17

M

main-loop GDP, 66, 103, 110, 145
mark-and-sweep, 79
marker, 114
matching types, 46
memory leaking, 58
message, 26
  asynchronous, 55
  expression, 43
  synchronous, 52
method, 26
  declaration, 32
  fetching, 53, 72
  heading, 32
  implementation, 32
  public inherited, 36
  selection, 45
  direct hit, 46
  matching types, 46
monitor, 21
morphing, 4
motion recording, 6
mouse, 102
mouse.event, 101
multi-processor system, 158
multi-programming, 21
multi-user system, 159
multiple inheritance, 33

N

null expression, 40
O
object, 26
object-orientation, 19
object-reference, 27
object-table, 70
  attribute-access, 72
definition, 70
method-fetching, 72
repeated inheritance, 71
ofile, 93
operational semantics, 167
  Looks, 179
ostream, 92
ostream, 93
overloading, 32, 36, 45

P
parallel programming, 21
parameter, 32, 45
parser, 65
Phong illumination model, 118
Phong shading, 116
plane, 139
point, 94
polygon, 115
polyline, 114
prefix, 37
primitive, 114
prism, 134
private feature, 26
procedure, 26
process, 20
process-switching, 21
pseudo-parallelism, 22
ptp, 145
public feature, 26

Q
quasi-parallelism, 22

R
raster-graphics, 12
with DP, 12
readable, 90
real, 86
recursive class, 31
redefinition, 36
reference-counting, 79
reference-location, 27
expression, 42
reflection model, 116
  ambient, 116
diffuse, 116
specular, 116
remote procedure call, 21
renaming, 37
renderer, 66
rendezvous, 21
repeated inheritance, 34, 38
repetition, 51
responder, 99
result, 49
revolute, 136
rigid body, 109
run-time library linking, 59, 81
  run-time compiled method fetching, 83
  run-time compiled object fetching, 82
  run-time library creation, 84
  run-time library opening, 81

S
scope rules, 43
script, 4
selectable, 125
selection, 51
selector, 123
self expression, 41
semantics, 167
  axiomatic, 168
denotational, 168
  Looks, 179
operational, 167
  Looks, 179
semaphore, 20
shading
  flat, 116
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gouraud, 116</td>
<td>var parameter, 32</td>
<td></td>
</tr>
<tr>
<td>Phong, 116</td>
<td>vector, 94</td>
<td></td>
</tr>
<tr>
<td>signature, 164</td>
<td>vector graphics, 12</td>
<td></td>
</tr>
<tr>
<td>single inheritance, 33</td>
<td>void, 86</td>
<td></td>
</tr>
<tr>
<td>specular reflection model, 116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sphere, 141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>statement, 47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocation, 49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignment, 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asynchronous message, 55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>block-statement, 48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>checked assignment, 53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty, 48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repetition, 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result, 49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>selection, 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronize, 55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronous message, 52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string, 88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strong typing, 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbol-table, 67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>functions, 187</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronize, 55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronous message, 52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronous message passing, 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syntax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstract, 163</td>
<td></td>
<td></td>
</tr>
<tr>
<td>concrete, 163</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system-class, 59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>term, 165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>texture mapping, 160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>topology, 115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type, 27, 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conformance, 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEvent, 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>universal animation system, 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value parameter, 32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

The work presented in this thesis originates in the observation that in current animation and simulation systems the animation tasks are tightly coupled to the other tasks of an application, e.g. the actual simulation calculations. As a result, programs for such systems tend to be complex because of the interleaving of the various tasks. Furthermore, the usually complicated motion calculations of the animation tasks will slowdown the performance of the system.

Chapter 2 introduces an animation system, the Generalized Display Processor (GDP), that does take into account this decoupling of animation tasks and other tasks of an application. Of course there has to be some communication from the application to the GDP. For this purpose the script-language Looks is designed (Chapter 3). Since an animation can be considered as a collection of autonomous moving objects, the language Looks is based on the paradigms of object-orientation and concurrency. With respect to object-orientation Looks supports concepts such as multiple and repeated inheritance, genericity, and dynamic binding. The concurrency mechanism of Looks is based on pseudo-parallelism, because this synchronization principle perfectly matches the global synchronizations within an animation (i.e. rendering the various frames).

The design of the GDP and its four constituents — parser, interpreter, renderer, and symbol-table — is explained in Chapter 4. The next chapter discusses the implementation of three important aspects of the GDP: abstract syntax trees with their interpretation, garbage collection, and run-time library linking.

Chapter 6 presents a class library that can be used in combination with the language Looks. Since the primary goal of the GDP is interactive computer animation, the class library most importantly deals with classes for events, graphical user interfaces, direct manipulation, rendering (geometry, topology, light sources, camera), kinematics, inverse kinematics, and dynamics.

Chapter 7 shows how Looks and its accompanying classes can be used to create a simple animation that contains inverse kinematics motion and direct manipulation.
Samenvatting

Het werk dat beschreven wordt in dit proefschrift vindt zijn oorsprong in de observatie dat in huidige animatie- en simulatiesystemen de animatietaken nauw verwoven zijn met de overige taken van een applicatie, zoals bijvoorbeeld de feitelijke simulatieberekeningen. Deze verwering heeft tot gevolg dat programmatuur voor dergelijke systemen complex is. Daarnaast zullen de vaak ingewikkelde bewegingsberekeningen, zoals die gebruikt worden ten behoeve van de animatietaken, de snelheid van het gehele systeem nadelig beïnvloeden.

In hoofdstuk 2 wordt een animatie-systeem, de Generalized Display Processor (GDP), geïntroduceerd dat gebaseerd is op een ontkoppeling van de animatie-taken en de overige taken van een applicatie. Uiteraard dient er enige communicatie te zijn van een applicatie naar de GDP. Voor dit doel is de script-taal Looks ontwikkeld (hoofdstuk 3). Aangezien een animatie beschouwd kan worden als een verzameling autonoom bewegende objecten is de taal Looks gebaseerd op de paradigma’s van object-orientation en concurrency. Met betrekking tot object-orientation ondersteunt Looks concepten zoals multiple en repeated inheritance, genericity en dynamic binding. Het concurrency mechanisme van Looks is gebaseerd op pseudo-parallelisme aangezien dit synchronisatie mechanisme uitstekend aansluit bij de globale synchronisatie binnen een animatie, namelijk het tekenen van de diverse plaatjes.


Hoofdstuk 6 beschrijft een class bibliotheek die gebruikt kan worden in combinatie met de taal Looks. Aangezien het hoofddoel van de GDP bestaat uit interactieve computer animatie richt deze bibliotheek zich voornamelijk op classes voor events, grafische user interfaces, directe manipulatie, tekenen (geometrie, topologie, lichtbronnen, camera), kinematica, inverse kinematica en dynamica.

In hoofdstuk 7 wordt gedemonstreerd hoe Looks en de bijbehorende class bibliotheek gebruikt kunnen worden om een eenvoudige animatie te maken die bestaat uit inverse kinematica en directe manipulatie.
Acknowledgements

Writing a PhD thesis is a solitary venture, but it cannot succeed without the support and help from many people. At this point I want to thank the people who have contributed, in one way or another, to my PhD thesis:

- Kees van Overveld, my copromoter, for his guidance during the last five years,
- Dieter Hammer and Frans Peters for being promoters of my thesis,
- Erik Jansen and Martin Rem for being members of the "reading committee",
- Huub van de Wetering for reading the various drafts of my thesis and for helping to implement the GDP during evenings, weekends, and even holidays,
- Rita Roels, for helping to implement the GDP,
- Sobu, especially Alice Pillot, Marianne Wagemans, and Karen Leurs, for the financial support,
- my parents for, well, everything.
Curriculum Vitae

Eric Peeters was born on October 29, 1966 in Zundert. In 1985, after his pre-university education (VWO) at the Katholieke Scholengemeenschap in Etten-Leur, he started to study Computing Science at the Eindhoven University of Technology. In 1990 he wrote a thesis on an algebraic description and implementation of a circuit compiler, and in September of that same year he received his Master's degree. From 1990 until 1994 he was employed by the Eindhoven-Tilburg Organization for Inter-University Cooperation (SOBU) where he worked on the project Dialogue Management and Knowledge acquisition (DenK). During these years he was stationed at the Department of Mathematics and Computing Science of the Eindhoven University of Technology. The work carried out in context of the DenK project has lead to this thesis. Since January 1, 1995 he works as a research scientist at Philips Research.
Stellingen
behorende bij het proefschrift
Design of an Object-Oriented, Interactive Animation System
van
Eric Peeters

1. Interactieve animatie [1] dient niet verward te worden met interactieve ontwikkeling van animatie [2].


3. Huidige interactie technieken schieten tekort voor selectie en directe manipulatie van autonoom bewegende 3-D objecten.


5. De algemene oplossing voor het vinden van een lijn m door een punt P en loodrecht op een lijn l, zoals beschreven in [1], is niet algemeen. De oplossing is namelijk niet gedefinieerd als P op l ligt.


6. Copy-and-paste mechanismen, die de gebruiker in staat stellen om programmafragmenten snel te dupliceren, zijn niet bevorderlijk voor het verkrijgen van de juiste abstracties in een programma.
7. Het feit dat een belangrijk deel van de levenscyclus van computerprogrammaterur bestaat uit coderen, testen en onderhoud weerspiegelt zich niet in het curriculum van de studie Technische Informatica aan de Technische Universiteit Eindhoven [1].


8. De vaak dure, specialistische apparatuur van een universiteit kan doelmatiger worden indien zij 24 uur per dag en 7 dagen per week toegankelijk is voor medewerkers en studenten.


11. Gooien het rendement van het aandelenpakket dat is samengesteld door een fictieve aap, zoals beschreven in The Wall Street Journal [1], zou over inkomsten verkrengen uit beleggingen kanspelbelasting geheven dienen te worden.


12. De combinatie van een promotie en een volledige betrekking heeft als voordeel dat men zich geen zorgen hoeft te maken over de kenze van een vakantiebestemming.