15

Motion modeling

This chapter is dedicated to discussing some geometric techniques for *animation modeling* and *motion planning*. In particular, the reader is introduced to the degrees of freedom of a moving system, and to the central issue of *configuration space* (CS), the set of numeric k-tuples that determine the placement and orientation of all the system components. A curve in configuration space, parametrized on the time domain, completely specifies the *motion* of the system. When the animated scene is too complex, it is necessary to project the configuration space onto coordinate subspaces related to the various actors, and to adopt a graph-theoretic approach to global *choreography*, allowing for motion coordination of interacting actors. The chapter also presents a general geometric technique to compute a polyhedral approximation of *free configuration space*, that encodes all the feasible motions of a mobile system in presence of obstacles. From a geometric programming viewpoint, the reader will learn that `PLaSM` provides FLASH animations based on 2D keyframes and gives a good support, based on CS sampling, to the symbolic generation of *animated* VRML of complex *storyboards*.

15.1 Degrees of freedom

Consider a moving system $R = \{R_i\}$, as a set of rigid bodies, either mutually constrained or not. The motion of R is performed within a working space $\mathrm{WS} \subset \mathbb{E}^d$, that contains an environment $E = \{E_j\}$, which is a set of rigid obstacles, that are either stationary or move along known trajectories. *Dimension* of the motion planning problem is the minimal dimension d of an Euclidean space such that $R \cup E \subset \mathbb{E}^d$.

The moving system R has also a number k of *degrees of freedom* (DOFs) which is equal to the minimal number of scalar parameters that uniquely determine the *configuration* of R, i.e. the placement and orientation of all the R elements with respect to a reference coordinate frame.

The degrees of freedom of a moving system are also called *generalized coordinates* [FL87], and constitute the main ingredient for the computation of the system dynamics using either the Lagrange-Euler or the Newton-Euler formulations. But system dynamics is beyond the scope of this book. The interested reader may

Geometric Programming for Computer-Aided Design Alberto Paoluzzi
 ISBN 0-471-89942-9

study a good Robotics book, for example [FL87]. A classical reading in Mechanics is [LCA26]. The realistic simulation of behavior of physical systems is one of the main achievements of computer graphics in the last decade [CS89].

Rigid 2D body The *translational motion* of a planar body B moving amidst a collection of polygonal obstacles has geometrical dimension 2 and 2 degrees of freedom, because the orientation of B does not change, and the body position is determined by the position of one of its points, fixed in advance.

The more *general motion* of a single rigid planar figure has dimension 2 and 3 degrees of freedom, including two translational and one rotational degree. The rotational degree determines the orientation of the figure with respect to a reference coordinate frame. It is usually chosen as the angle between two corresponding axes from a fixed reference frame and from a frame attached to the object.

Example 15.1.1 (2D body with 3 DOFs)
The simplest example is given in Script 15.1.1, where `body2D` is a rectangle. The `x` and `y` coordinates of its bottom-leftmost corner are assumed as the translational DOFs, whereas the `angle` between its bottom edge and the x axis of the fixed frame in $\mathbb{E}^2$ is taken as the rotational DOF. The 2D cuboid configurations are generated as the images of the `body2D` function, depending on three real parameters. The four configurations produced by the last expression are displayed in Figure 15.1. The reader should notice that the object `CUBOID:<2,1>` is generated in a local frame centered on the bottom leftmost corner and aligned with the fixed reference frame. Notice also that the rotation tensor must be applied *before* the translation tensor.

Script 15.1.1 (2D body with 3 DOFs)

```
DEF body2D (x,y,angle::IsReal) =
   (T:<1,2>:<x,y> ∼ R:<1,2>:angle): (CUBOID:<2,1>);

(STRUCT ∼ AA:body2D):
   < <0,0,0>, <2,2,PI/4>, <4,2+2*SIN:(PI/4),0>, <6,0,3*PI/4> >
```

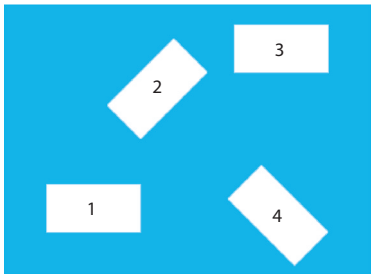

Figure 15.1 Four configurations of the 2D cuboid. The fixed reference frame is aligned with the bottom and left edges of configuration 1

Rigid 3D body The general motion of a single rigid body B moving in a 3D space has 6 degrees of freedom. In this case there are 3 *translational* DOFs, corresponding to the position of a body point, and 3 *rotational* DOFs, corresponding to a triplet of *Euler angles*, that determine the orientation of a reference frame attached to the body, with respect to a fixed coordinate frame.

A common choice for the triplet of Euler angles ϕ, θ, ψ, in this case called *roll*, *pitch* and *yaw* angles, respectively, leads to a triplet of elementary rotation tensors $\boldsymbol{R}_z(\phi)$, $\boldsymbol{R}_y(\theta)$ and $\boldsymbol{R}_x(\psi)$, where $\boldsymbol{R}_z$, $\boldsymbol{R}_y$ and $\boldsymbol{R}_x$ respectively denote rotations about the z, y and x axes. Their composition in the order specified below gives to the general 3D rotation tensor that gives the body its orientation with respect to the fixed frame:

$$\boldsymbol{R}(\phi, \theta, \psi) = \boldsymbol{R}_z(\phi) \circ \boldsymbol{R}_y(\theta) \circ \boldsymbol{R}_x(\psi).$$

Some different choices for the Euler angles are quite common in Robotics, but are unusual in graphics and animation systems.

Our well-trained reader is certainly aware at this point that the above rotation tensor is applied to the moving system B *before* the translation tensor, in order to get a given body configuration, i.e. the orientation and position corresponding to a 6-tuple of generalized coordinates.

Example 15.1.2 (3D body with 6 DOFs)
The function `body3D` given in Script 15.1.2 produces a position and orientation of the `CYLINDER` with radius 0.2 and height 0.8, originally aligned with the z axis and with the basis centered on the origin of the coordinate frame. The new configuration depends on the 6 generalized coordinates `dx`, `dy`, `dz`, `roll`, `pitch` and `yaw`.

Script 15.1.2 (3D body with 6 DOFs)

```
DEF body3D (dx, dy, dz, roll, pitch, yaw::IsReal) =
   (T:<1,2,3>:<dx,dy,dz> ~ R:<1,2>:roll ~ R:<1,3>:pitch ~ R:<2,3>:yaw):
      (CYLINDER:<0.2,0.8>:24);

STRUCT:< MkFrame, body3D:< 0.5,0.5,0, PI/2,0,PI/4 > STRUCT MkFrame,
   MKvector:<0,0,0>:<0.5,0.5,0> >;
```

The graphical assembly generated by the last expression, where `MkFrame` is the generator of the model of the standard 3D frame, and `MKvector` is the generator of the vector difference of two assigned points, is shown in Figure 15.2.

A given configuration of our cylinder, say, without reference frames, is clearly generated by the application of the generating function `body3D` to the 6-tuple of DOFs:

```
body3D:< 0.5,0.5,0, PI/2,0,PI/4 >
```

Mechanical manipulators A mechanical *manipulator*, more often called a *robot arm*, is an ordered set or rigid bodies, called *links*, pairwise connected by either revolute or prismatic *joints*. The first link is attached to a supporting basement, whereas the last link may handle a tool. A fixed reference frame is associated with the arm basement, and another reference frame is considered as rigidly attached to each link.

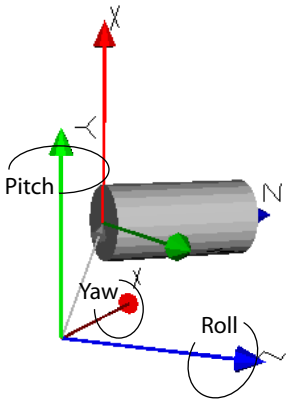

Figure 15.2 Position and orientation of a 3D body, depending on 6 degrees of freedom. Both the attached and the fixed frame are shown

Joint transformations Various kinds of joints may allow for different combinations of rotational and translational DOFs. Usually, a *spherical* joint permits either 2 or 3 rotational degrees of freedom, a *cylindrical* joint allows for 1 rotational and 1 translational DOF, whereas a *prismatic* joint gives 1 translational degree. Notice that each joint is associated with a *link pair* and constrains the relative movement of the second link with respect to the first one.

A transformation tensor depending on the allowed degrees of freedom can hence be associated with each joint, i.e. to each ordered link pair (R_i, R_j). This tensor will specify the position and orientation of the reference frame attached to the R_j link with respect to the coordinate frame associated to the R_i link.

Kinematics chains and bones When analyzing the structure of a manipulator $R = \{R_i\}$ as a graph $G = (N, A)$, whose nodes are the links and whose arcs are the joints, i.e. with $N = R$ and $A \subset R^2$, the resulting graph is most often acyclic,[1] i.e. is a set of *open kinematics chains*.

Models of anthropomorphic robots also contain a tree made of links and joints. In computer animation a simplified representation and visualization of system links, used to visualize the degrees of freedom and to specify the *configuration space paths* that define the system motion, is usually given as a set of pairwise jointed *bones*.

Example 15.1.3 (Plane robot arm)
Let us consider a simplified 2D robot arm, as a kinematics chain with four links and three rotational joints. We also assume that (a) the links are rectangles; (b) they are equal to each other; (c) the first link is rigidly connected to the embedding space, i.e. it cannot move.

In Script 15.1.3 we give the `link` object as a rectangle with the main axis aligned with the negative y axis, and with the position of the rotational joint positioned on the origin. The rotational `joint` is a function of a real parameter. When applied to an

[1] Or, even better, is a *tree*.

arbitrary value α of the argument in degrees, `joint:`α returns an affine transformation tensor. Finally, `arm` is a function of the three joint angles, respectively denoted `a1`, `a2` and `a3`.

To fully understand the meaning of the `arm` body expression, the reader should remember the semantics of hierarchical structures from Section 8.2.1. Two different configurations of the plane robot, produced by application of the `arm` generating function to different triples of actual parameters, are displayed in Figure 15.3.

Script 15.1.3

```
DEF link = (T:<1,2>:<-1,-19> ~ CUBOID):<2,20>
DEF joint (alpha::IsReal) = T:2:-18 ~ R:<1,2>:(alpha * PI/180);

DEF arm (a1,a2,a3::IsReal) = STRUCT:
   < link, joint:a1, link, joint:a2, link, joint:a3, link >;
```

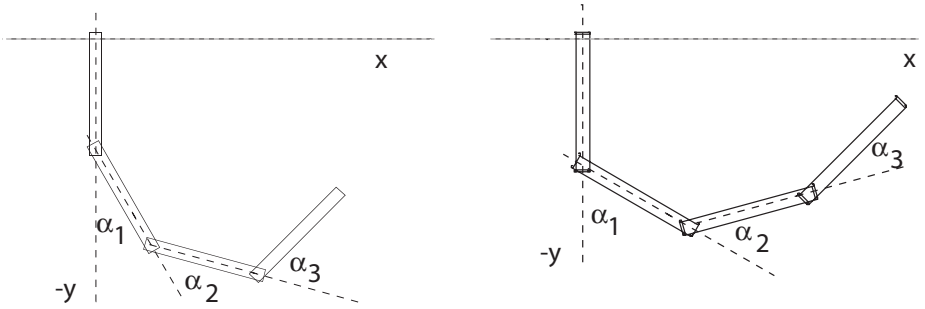

Figure 15.3 Two plane robot configurations generated by `arm:<30,45,60>` and `arm:<60,45,30>`

15.2 Configuration space

Each one of the generalized coordinates ξ_i of a moving system with k degrees of freedom, may vary continuously within a real interval

$$\Xi_i = [\xi_i^{min}, \xi_i^{max}] \qquad 1 \leq i \leq k.$$

where ξ_i^{min} and ξ_i^{max} are often called *joint limits* in robotics applications.

The Cartesian product CS of the k intervals Ξ_i is called the *configuration space* of the moving system:

$$CS := \Xi_1 \times \Xi_2 \times \cdots \times \Xi_k \subset \mathbb{R}^k.$$

Each point $(\xi_1, \ldots, \xi_k) \in CS$ corresponds to a different *configuration* of the moving system, i.e. to a different placement and orientation of all its parts.

Motion A continuous curve

$$\gamma : [0, 1] \to \mathbb{R}^k,$$

such that $\gamma[0,1] \subset CS$, completely defines a *feasible motion* of a mobile system.

The image of a smooth CS curve is also known as *configuration space path*. Clearly, $\gamma(0)$ and $\gamma(1)$ respectively correspond to the starting system configuration and to the goal configuration of the motion.

When reparametrized[2] in a time interval, such curve gives the configurations as a function of time, so that the curve itself and its first and second derivatives represent the configuration space *displacement*, *velocity* and *acceleration*.

Since a continuous acceleration is produced by a continuous force or torque, and standard actuators operate continuously, a feasible motion actually requires a configuration space curve at least of class C^2. More often, smooth CS curves are used to represent a motion, and in particular polynomial or rational curves. Non-uniform splines provide a very simple control of the motion acceleration.

Example 15.2.1 (Configuration space path)
A continuous curve in configuration space for the 2D robot `arm` defined in Script 15.1.3 is produced in Script 15.2.1 by the cubic Bézier generating function `CSpath`, where the `intervals` generator of 1D polyedral complexes is given in Script 11.2.1. The curve image in CS with reference axes labeled as α_1, α_2 and α_3, is displayed in Figure 15.4a. Notice that the sampling function produce the following values:

```
Sampling:6 ≡ < 0 , 1/6 , 1/3 , 1/2 , 2/3 , 5/6 , 1 >
```

A motion representation by graphical aggregation of the robot placements corresponding to the generated CS sampling is given in Figure 15.4b.

Script 15.2.1 (Bézier configuration space path)

```
DEF CSpath = (CONS ∼ Bezier:S1):<<0,0,0>,<90,0,0>,<90,90,0>,<90,90,90>>;
DEF Sampling (n::IsInt) = (AA:/ ∼ DISTR):< 0..n, n >;

(MAP:CSpath ∼ Intervals):18;
(STRUCT ∼ AA:(arm ∼ CSpath ∼ [ID]) ∼ Sampling):18;
```

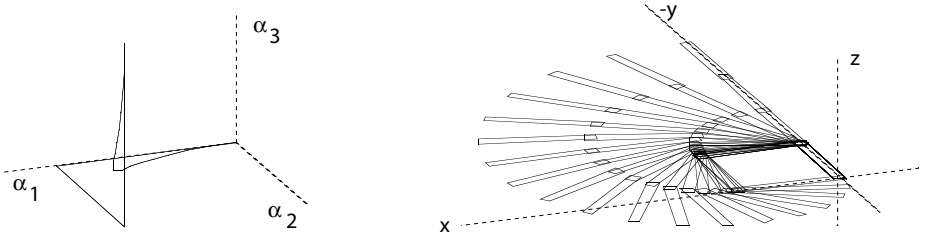

Figure 15.4 (a) Configuration space path produced as a cubic Bézier curve
(b) Set of configurations corresponding to a sampling of the path

[2] See Section 5.1.2.

Example 15.2.2 (Different CS paths)
Two different configuration space paths for the plane robot arm with 3 rotational DOFs are given in Script 15.2.2 and displayed in Figure 15.5. Notice that both the corresponding motions have the same start and goal configurations, whereas the two intermediate control points of the configuration space path are different.

Script 15.2.2

```
DEF CSpath1 = Bezier:S1:<<0,0,0>,<90,0,0>,<90,90,0>,<90,90,90>>;
DEF CSpath2 = Bezier:S1:<<0,0,0>,<0,90,90>,<90,90,0>,<90,90,90>>;
```

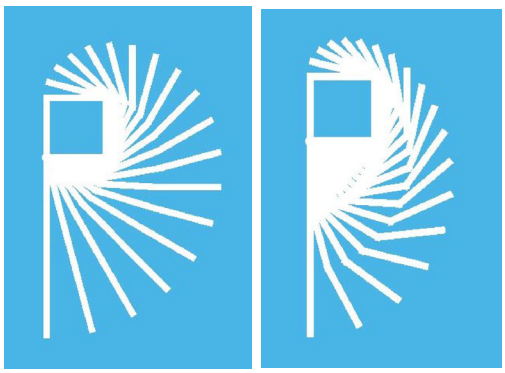

Figure 15.5 Two movements of the plane arm, corresponding to different configuration space paths, i.e. to different behaviors

Free configuration space The set FP of *free positions* for a moving system B with k degrees of freedom is the compact subset of $CS \subset \mathbb{R}^k$ constituted by all points whose corresponding placement of the moving system is *free*, in the sense that B neither intersects any obstacle in its working space WS nor self-intersects. The set FP is more often known as *free configuration space*.

A goal configuration r_2 is reachable from a starting configuration r_1, i.e. there exists a feasible motion between them, if they are both contained in a same connected component of FP. To compute FP is therefore equivalent to deciding in advance all the possible reachability problems for a mover B and a set of obstacles E. Clearly, in order to plan the motions from a given starting configuration r_1 of the mover, it is sufficient to determine just the connected component of FP that contains r_1.

The reader interested to algorithmic robot motion planning is referred to the Latombe book [Lat91].

15.3 Animation with PLaSM

In the last few years, PLaSM has been extended to support colors, textures, cameras and animations. In particular, PLaSM was aimed at giving symbolic support to the design of very complex animations.

For this purpose the language semantics were extended, and both an animation methodology and an animation server were developed [BBC+99]. The animation server, used only for display, was first implemented using *OpenInventor* as the animation engine. More recently, the language interpreter started to directly export *animated* VRML files, and some simple FLASH animations.

In this section we discuss the basic concepts of the `PLaSM` animation model, based on CS sampling, and give several simple examples. The design and implementation of complex animations with more than one animated *actor* are discussed in the next sections.

15.3.1 Some definitions

Let us start our discussion of the `PLaSM` approach to the generation of computer animations by giving some definitions.

Scene A *scene* is here defined as a function of some real parameters, with values in some suitable data type, that we called *animated hierarchical polyhedral complex* (AHPC).

Configuration Each feasible set of parameters defines a *configuration* of the scene.

State A corresponding pair <*time, configuration*> is a *state* of the animation. The product of the time domain times the configuration space gives the *state-space* of the animation.

Behavior The animation *behavior* is a curve in animation state-space. Each part of the scene may change in time with respect to position, orientation and modeling parameters, and may even change its internal assembly structure.

Animation An animation is a pair <scene, behavior>.

15.3.2 Generating FLASH animations

The `PLaSM` language may give basic support to the generation of FLASH animations, exported as `.swf` files,[3] that are rendered by a FLASH viewer, usually embedded by default as a *plug-in* in web browsers.

Animations of this kind are based on the explicit generation of sequences of *frames* either by *inbetweening* of *key-frames* or by configuration space *sampling*.

The reader should remember that FLASH animations are, by definition, 2D only. The animation of a 3D scene would require a preliminary projection of the scene data base. Consequently, in the remainder of this section we assume we are dealing with 2D data only.

---

[3] Where the file suffix stands for *ShockWaveFlash* by Macromedia.

Key-frame inbetweening An important technique of computer generated animation, originating from traditional animation methods, consists in setting from scratch some pictures, called *key-frames*, with important postures of the characters in the scene, and in deriving a sufficient number of intermediate figures by interpolation, that in this context is called *inbetweening*.

With PLaSM, this animation technique may be instanced by defining the either open or closed polygonal contours of the objects in two or more subsequent keyframes, and by generating the intermediate postures by the *shape interpolation* or approximation methods defined in Section 7.3.2, that the interested reader is invited to review at this point.

Notice that a *shape* is there defined as an equivalence class of congruent figures, where the class representative is chosen with a vertex in the origin of the reference frame. Hence, in order to instance a shape in a plane configuration, two translational and one rotational parameter values are required. Such parameters, needed for placement and orientation of characters generated by shape interpolation, are equally derived by interpolation of the placements of two shape points, using 3D (three DOFs) curves or splines parametrized on the time domain.

Configuration space sampling When the shape as well as the placement of an actor may be generated by using a PLaSM generating function depending on actual parameters of numeric type, the easiest way to produce a FLASH animation consists of a variation of the CS sampling method used to export VRML animations.

The only difference in this case regards the need for explicit generation of a sequence of polyhedral complexes, one for each frame of each moving actor. As it is easy to understand, such an approach is quite space-inefficient, but is the only approach allowed by the *Macromedia* API to produce `.swf` files. It follows that only quite simple FLASH animations can be safely exported and efficiently rendered.

PLaSM primitives for Flash animation

The PLaSM primitives that affect the FLASH rendering of an exported `.swf` file are listed and discussed in this section. To load a FLASH file in a web browser from an HTML page requires some specialized tags. They are given in Script 15.3.1 for the sake of user comfort, because the two more diffuse browsers require some horrible tag attributes. Below we give a solution for an HTML *anchor* that works on *both* browsers, at least at the time of this writing!

Notice that in Script 15.3.1 the filename is `flashExample.swf`, supposed to be in the same directory of the HTML page, and that the *anchor* name is `flashExample`.

A FLASH file may contain either a *movie* or a single static *picture*, that is also considered a movie. Hence we discuss here the primitives used to generate and export from PLaSM both single pictures and simple animations.

RGBACOLOR The RGBACOLOR primitive of the `psmlib/flash.psm` predefined library, applies to number quadruples in $[0,1]^4$, i.e. in RGBA space, where $\alpha \in A$, the last color coordinate, denotes (the opposite of) *transparency*, ranging from full transparency ($\alpha = 0$) to full opacity ($\alpha = 1$). Two examples follow:

Script 15.3.1 (Embedding a flash file into html)

```
<A NAME="flashExample">
<OBJECT CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" WIDTH="100%"
CODEBASE="http://active.macromedia.com/flash5/cabs/
   swflash.cab#version=5,0,0,0">
<PARAM NAME="MOVIE" VALUE="flashExample.swf">
<PARAM NAME="QUALITY" VALUE="high">
</OBJECT>

<EMBED SRC="flashExample.swf"
WIDTH="100%" PLAY="true" LOOP="true" QUALITY="high"
PLUGINSPAGE="http://www.macromedia.com/shockwave/download/
   index.cgi?P1_Prod_Version=ShockwaveFlash">
</EMBED>
</A>
```

```
DEF darkGrey = RGBACOLOR:<0.1, 0.1, 0.1, 1>;
DEF trasparentDarkGrey = RGBCOLOR:<0.1, 0.1, 0.1, 0.7>;
```

FILLCOLOR is used to fill the object *interior* with the specified RGBACOLOR:

```
DEF name = pol2D FILLCOLOR RGBAcolor;
```

LINECOLOR is used to associate the given color property to the *boundary* edges of the object:

```
DEF name = pol2D LINECOLOR RGBAcolor;
```

ACOLOR is used to give a color property both to the edges and to the interior of the object:

```
DEF name = pol2D ACOLOR RGBAcolor;
```

LINESIZE is the operator used to specify the `lineWidth` in pixels of the object edges. In this case we have:

```
DEF name = pol2D LINESIZE lineWidth;
```

Flash is the exporting directive for single pictures, that exports the `pol2D` object, visualized in a display area of `areaWidth`, to the file with name `fileName.swf`

```
Flash:pol2D:areaWidth:'fileName.swf';
```

ACTOR is applied to the sequence `polComplexSequence` of frames related to the same object, and then to the `startingTime` integer, that specifies the ordinal time when the object must start being visible in the movie. The duration of the action of the actor in the movie will depend on the `frameRate` parameter specified in the exporting `FlashANIM` statement.

```
DEF name = ACTOR:polComplexSequence:startingTime;
```

FRAME is used to generate a given 2D polyhedral complex to be visible in the movie in the (ordinal) time interval [`startingTime`, `endingTime`].

```
DEF name = FRAME:polComplex:startingTime:endingTime;
```

FlashANIM is the statement used to export a movie. It must be successively applied to an *animation2D* value, that must be either (a) a sequence of sequences of 2D polyhedral complexes, or (b) a sequence of `ACTOR` and `FRAME` expressions. The resulting function is applied to the `areaWidth` parameter (in pixels), to the `'fileName.swf'` string, and finally to the `frameRate` integer parameter:

```
FlashANIM:animation2D:areaWidth:'fileName.swf':frameRate
```

Example 15.3.1 (A first example)
A very simple but complete example of generation and exporting of a FLASH animation is given in Script 15.3.2. In particular, a default colored square moves translationally over a background of a yellow rectangle with black borders. As stated by the `FlashANIM` primitive, the animation rendering is planned for a display area `300` pixel wide at a framerate of `10` frames per second. Also, the life cycle of the `FRAME` generated object (the yellow rectangle), corresponds to the ordinal time interval $[1, 30]$, whereas the mover object (the cyan square) appears at ordinal time $t = 21$ and completes its motion at ordinal time $21 + 10$. The actual movie duration is 3.1 *sec*, depending on the given framerate.

Script 15.3.2 (Translating square)

```
DEF mover (tx::isInt) = T:<1,2>:<tx,5>:(CUBOID:<1,1>);
DEF moverSequence = (AA:mover:(1..10));
DEF static_rectangle = (CUBOID:<10,20> fillcolor yellow);

DEF background = FRAME:static_rectangle:1:30;
DEF actor = ACTOR:moverSequence:21;

FlashANIM:< background, actor >:300:'animation1.swf': 10;
```

Example 15.3.2 (Umbrella animation)
Here we animate the opening of the wire-frame umbrella with curved rods given in Script 11.5.2. So, in the following Script 15.3.3 we start by loading the `flash`, `vector` and `viewmodels` libraries. The latter is needed to set-up a `proj` projection of the 3D wire-frame umbrella in 2D space. Two *clips* to be displayed are then defined, corresponding to the opening umbrella, and to the same but reversed frame sequence. Finally, our FLASH animation, to be rendered at a `framerate` $= 20$ in a screen area wide 200 pixels, is exported to the `umbrella.swf` file.

Script 15.3.3 (Animated umbrella)

```
DEF proj = projection: parallel: dimetric;
DEF umbrellaFun = proj ∼ t:1:5 ∼ umbrella:10;
DEF umbrellaFrames= aa:umbrellaFun:(2 scalarVectProd 11..80);
DEF clip1 = ACTOR:umbrellaFrames:1;
DEF clip2 = ACTOR:(REVERSE:umbrellaFrames):41;

FlashANIM:< clip1, clip2 >:200:'umbrella.swf':20;
```

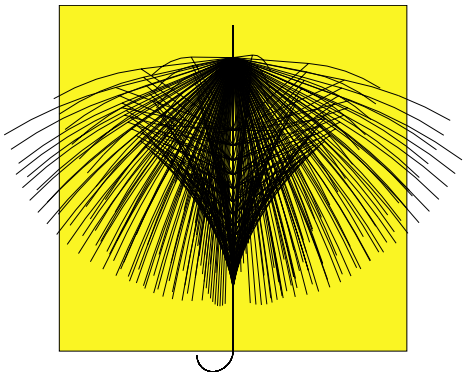

Figure 15.6 The set of frames used for the Flash umbrella animation

15.3.3 Generating VRML animations

Unlike FLASH animations, where only a basic support is currently given, to be possibly combined with the importing and editing of the generated `.swf` files within the FLASH interactive development environment (IDE), `PLaSM` offers quite sophisticated support to the design of complex VRML animations. The language may support the development of animations at three different levels:

1. by offering primitives resulting in a *hierarchical animation*, where movie clips may contain other movie clips, and are translated and scaled hierarchically on the time axis;
2. by supporting the *choreography* methodology discussed in the next section, that allows for automatical coordination of the time interaction of independently defined actors, by using network programming techniques, and by defining each clip *storyboard* as an oriented graph;
3. at a finer modeling level, a functional programming approach may define *actors as chains of operators* going from parameter spaces with smaller dimension, to spaces with higher dimension, up to the actor configuration space, even of very high dimension. The actor postures and motion can therefore be controlled at the more appropriate level, with the minimal animator effort.

Last but not least, the use of a scripting language in designing and implementing animations, allows for reuse of characters and for automatic generation of different behaviors. It is interesting to notice that the top-level animation systems contain a scripting language, like, e.g., *Maya* and *MEL*. Actually, *every* action performed

in Maya runs based on *MEL* scripts. *MEL* is an integral part of Maya's overall design [LG02].

Animation data structure

In order to insert an animation support into PLaSM, a new primitive data type *Animated Hierarchical Polyhedral Complex*, completely transparent to the user, was added to the language. We often refer in the following sections to such a data type using the abbreviation *AnimPolComplex*. We also call *clip* a data object of this type. An animated polyhedral complex is defined as a quadruple

$$\texttt{animpolc} := \texttt{< polc, id, t}_{start}\texttt{, t}_{end}\texttt{ >}$$

where

1. polc := is a *hierarchical polyhedral complex* annotated with properties;[4]
2. id := is the unique *identifier* of the clip;
3. $\texttt{t}_{start}$:= is the clip *starting time*;
4. $\texttt{t}_{end}$:= is the clip *ending time*.

Timeline A clip *timeline* is the interval $[\texttt{t}_{start}\texttt{, t}_{end}]$ of the clip representation as *AnimPolComplex*.

Animation primitives

The set of specialized PLaSM primitives for animation modeling is introduced below. MOVE and FRAME are respectively dedicated to generating animated polyhedral complexes with a given behavior and to background objects on the scene in a specified time interval. ANIMATION is a container for non-linear editing of hierarchical animations. LOOP has the obvious meaning, SHIFT and WARP are used for hierarchical timeline translation and scaling, respectively.

FRAME is used for display of *static objects* is used for display of *static objects* $\texttt{polc}_i$ that are present on the scene only in a time interval $[\texttt{t}_i\texttt{, t}_{i+1}]$. Two patterns of usage are allowed:

$$\texttt{FRAME : polc : < t}_{start}\texttt{, t}_{end}\texttt{ >;}$$
$$\texttt{FRAME : < polc}_1\texttt{, polc}_2\texttt{, ... , polc}_n\texttt{ > : < t}_1\texttt{, t}_2\texttt{ , ... , t}_n\texttt{ , t}_{n+1}\texttt{ >;}$$

MOVE must be orderly applied to (a) generator of *geometric data*, given as a function of real parameters (degrees of freedom); (b) *configuration data*, given as n-sequence of CS points; (c) *timing data*, given as an increasing sequence of n time values.

---

[4] In particular, annotated with the sequence of CS points to be pairwise linearly interpolated during the polyhedral complex animation, and with the name of the generating function.

```
MOVE : objfun : < par1, par2, ... , parn > : < t1, t2, ... , tn >
```

where the function `objfun` generates the object placement $\texttt{objfun:par}_i$ at $\texttt{t}_i$ time, and where the object configurations are linearly interpolated between the CS points $\texttt{par}_i$ and $\texttt{par}_{i+1}$ within the time interval $[\texttt{t}_i, \texttt{t}_{i+1}]$. Notice that `PLaSM` curves or splines can be used to generate a proper sampling of generic behavior curves.

ANIMATION is used as a *container* that allows *hierarchical* aggregation of both standard and animated polyhedral complexes, including nested `ANIMATION` invocations. It may also contain hierarchical operators over both standard and animated polyhedral complexes, say `STRUCT` operators and affine transformations, as well as `LOOP`, `SHIFT` and `WARP` operators discussed below.

```
ANIMATION : (pols::isseqof:(OR~[isanimpol,isfun,ispol])) → isanimpol
```

The reader should notice that the `STRUCT` operator is overloaded to the behavior of the `ANIMATION` operator, so that they are fully interchangeable. What to actually use may be mainly a matter of code self-documentation.

LOOP and OUTERLOOP are used to *repeat* `times` times the animated polyhedral complex `anim`. In particular, `LOOP` repeats the content of the timeline $[\texttt{t}_{start}, \texttt{t}_{end}]$, whereas `OUTERLOOP` works on $[0, \texttt{t}_{end}]$.

```
LOOP : (times::isint)(anim::isanimpol) → isanimpol
OUTERLOOP : (times::isint)(anim::isanimpol) → isanimpol
```

SHIFT applies a *timeline translation* to the animated polyhedral complex `anim` from $[\texttt{t}_{start}, \texttt{t}_{end}]$ to $[\texttt{t}_{start} + \texttt{t}, \texttt{t}_{end} + \texttt{t}]$.

```
SHIFT : (t::isnum)(anim::isanimpol) → isanimpol
```

WARP and OUTERWARP produce a *timeline scaling* of the `anim` parameter. In particular, `WARP` scales $[\texttt{t}_{start}, \texttt{t}_{end}]$ with the time origin as fixed point of the scaling, whereas `OUTERWARP` scales $[\texttt{t}_{start}, \texttt{t}_{end}]$ with $\texttt{t}_{start}$ as fixed point.

```
WARP : (t::isnum)(anim::isanimpol) → isanimpol
OUTERWARP : (t::isnum)(anim::isanimpol) → isanimpol
```

Overloading of `PLaSM` primitives

Some important predefined `PLaSM` geometric operators have been extended to work also with *animated* polyhedral complexes. They include:

1. affine transformation tensors: `T`, `S`, `R`;
2. hierarchical assembly: `STRUCT`
3. geometric constructors: `CUBOID`, `MKPOL`;
4. function mapping over polyhedral constructors: `MAP`.

The remainder of this section is dedicated to discussing some implementation details of the `PLaSM` animation subsystem. The standard reader may go directly to the next section for some easy animation examples.

Animated behavior The `MOVE` primitive is implemented by evaluating the geometry generation function `objfun` in a modified primitive `PLaSM` environment named `*anim_env*`. In such an environment a new type is defined:

$$\texttt{AnimBehaviour} := \texttt{<< par}_1\texttt{, par}_2\texttt{, ... , par}_n\texttt{ >, < t}_1\texttt{, t}_2\texttt{, ... , t}_n\texttt{ >>}$$

This data type is used to manage (via the redefinition of the `APPLY` combinator) the parameter propagation in expression evaluation. In particular, a primitive redefined only in `*anim_env*`, in order to accept both its standard parameters and the ones of `AnimBehaviour` type, *absorbs* this parameter and creates some basic *AnimPolComplexes*. In particular, the primitive `PLaSM` *application*

```
f:a = APPLY:< f,a >
```

was redefined in such a way that:

$$\texttt{f : <<par}_1\texttt{, par}_2\texttt{, ... , par}_n\texttt{>, <t}_1\texttt{, t}_2\texttt{, ... , t}_n\texttt{>>} \rightarrow \texttt{AnimPol}$$

if the function `f` can accept `AnimBehaviour` values in `*anim-env*`; otherwise we have

$$\texttt{f : << par}_1\texttt{, par}_2\texttt{, ... , par}_n\texttt{ >, < t}_1\texttt{, t}_2\texttt{, ... , t}_n\texttt{ >>} \rightarrow$$
$$\texttt{<< f:par}_1\texttt{, f:par}_2\texttt{, ... , f:par}_n\texttt{ >, < t}_1\texttt{, t}_2\texttt{, ... , t}_n\texttt{ >>}$$

if `f` is a standard function. With this approach, the animation parameters are normally processed by all the standard `PLaSM` functions, until they encounter a function that supports the generation of animations, i.e. a function which can be applied to `AnimBehaviour` values.

Specialized methods Some specialized methods work with values of type `Anim-Behavior`:

1. `anim-arg(arg::TT)` returns `arg` if it is of `AnimBehaviour` type, else, if `arg` is a sequence `<a1, ... , an>`, then an `AnimBehaviour` with parameters `<anim-arg:a1, ... , anim-arg:an>` is returned; The purpose of this method is to homogenize `AnimBehaviour` with respect to sequences.
2. `anim-pred(pred::IsFun)(arg::IsAnimBehaviour)` is used to check if all `arg` parameters verify the `pred` predicate.
3. Finally we have:

$$\texttt{IsAnimBehaviourOf(pred::IsFun)(arg::TT)} \equiv \texttt{anim-pred:pred:(anim-arg:arg)}$$

Affine transformations The elementary affine tensors `T`, `S` and `R` are redefined as follows, so that they can be applied to standard polyhedral complexes as well as to animated polyhedral complexes:

```
{ T | S | R }(index::OR ~ [IsInt,IsSeqOf:IsInt])
   (par::IsAnimBehaviourOf:(OR ~ [IsNum,IsSeqOf:IsNum]))
   (pol::OR ~ [IsPol, IsAnimPol]) → IsAnimPol
```

Geometric constructors The geometric constructor `CUBOID` may be animated by giving a value of type `AnimBehaviour` to its numeric parameters. Analogously, an animated behavior can be given to the `points` parameter of `MKPOL` constructor. Anyway, at least at the time of writing, the internal structure of a polyhedral complex can be animated only by animating the parameters of the embedded transformations.

```
CUBOID (par::IsAnimBehaviourOf:(IsSeqOf:IsNum)) → IsAnimPol

MKPOL (points::IsAnimBehaviourOf:(IsSeqOf:(IsSeqOf:IsNum)))
   (cells::(IsSeqOf:(IsSeqOf:IsIntPos)))
   (pols::(IsSeqOf:(IsSeqOf:IsIntPos))) → IsAnimPol
```

Function mapping A (only internal) variation `MAPC` of the `MAP` primitive operator can be similarly animated, by ...

```
MAPC (fun::((IsSeqOf:IsNum) → IsAnimBehaviourOf:(IsSeqOf:IsFun)))
   (pol::IsPol) → IsAnimPol
```

Simple examples

In this section we show some simple animation examples based on CS sampling. In each case the moving `object` must be defined as a *function* of its degrees of freedom. When some *animPolComplex* value is exported, an *animated* VRML file is appropriately generated. In particular, the VRML animation will produce a piecewise linear interpolation of adjacent configuration space points.

If a non-linear *behavior* is needed, i.e., if the object motion must correspond either to a non-linear CS path or to a non-uniform velocity, then it is always possible to linearly approximate the desired behavior with arbitrary precision by either generating a uniform point sampling of a nonlinear CS curve, or by using a non-uniform time sampling for equally spaced CS points, respectively.

Example 15.3.3 (Rotated cube (1))
A very simple animation example is given in Script 15.3.4, where the standard unit `cube` is rotated about the z axis. In this case we have only 1 rotational DOF, associated with the `alpha` parameter. In this case, the CS space is 1D, so that `cube` must be a function of one real parameter, and we have $(\alpha_i) = (0, \pi, 0) \subset CS$, and $(t_i) = (0, 3, 6)$. The animated VRML generated by the `PLaSM` interpreter gives a piecewise linear interpolation between such CS points.

Script 15.3.4 (Rotated cube)

```
DEF cube (alpha::IsReal) = (R:<1,2>:alpha ∼ CUBOID):<1,1,1>;
DEF out = MOVE:cube:<0,PI,0>:<0,3,6>;

VRML:out:'out.wrl';
```

It is very hard to give an appropriate rendering of an animation sequence in a book. Figure 15.7 gives a frame sequence from a DV rendering of the animation of Script 15.3.4 left to right and top to bottom.

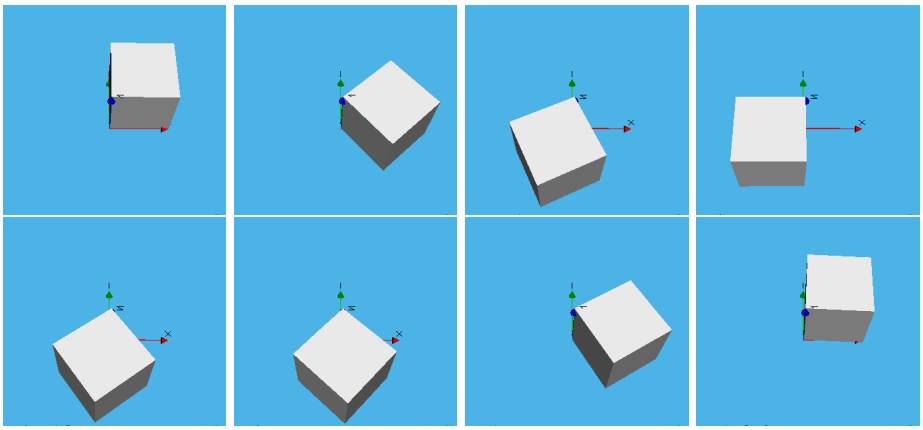

Figure 15.7 A sequence of frames from the animated rotation of the unit cube about the z axis

A rotation axis parallel to z and passing for the cube centroid is used in Script 15.3.5. The reader should notice that standard tensor composition applies, where only one of the rotation parameters is used as an argument of the generating function `cube`, according to the fact that the animation has only 1 degree of freedom.

Script 15.3.5 (Rotated cube (2))

```
DEF cube (alpha::IsReal) =
   (R:<1,2>:alpha ~ T:<1,2>:<-1/2,-1/2> ~ CUBOID): <1,1,1>;
DEF out = MOVE:cube:<0,PI,0>:<0,3,6>;

VRML:out:'out.wrl';
```

Example 15.3.4 (Planar robot arm)
The planar robot arm with 3 DOFs defined in Scripts 15.1.3 and 15.2.1 is animated in Script 15.3.6 by using a CS path that is a piecewise approximation with 8 linear segments of a cubic Bézier curve defined by four CS points. Notice that:

1. the expression `Sampling:8` generates a sequence of 9 values in $[0, 1]$;
2. the `[ID]` function transforms each of them in a 1D vector;
3. the above vectors are finally mapped by the `CSpath` function into a sequence of CS points, according to the Bézier curve defined in Script 15.2.1.

Script 15.3.6 (2D arm)

```
DEF CSpoints = (AA:(CSpath ~ [ID]) ~ Sampling):8
DEF out = MOVE:arm:(CSpoints:(0..8));

VRML:out:'out.wrl';
```

Example 15.3.5 (Planar motion of cube)
A general planar motion with 3 degrees of freedom of the unit cube is produced by Script 15.3.7, where the cube is translated along a circular path in 2D while rotated about its own vertical axis. In particular, the `CSpoints` sequence used by the `MOVE` primitive is generated by applying the function

$$\texttt{CScurve} : \mathbb{R} \to \mathbb{R}^3$$

to the elements of the number sequence in $[0, 2\pi]$ produced when evaluating the expression `2*PI scalarVectProd Sampling:16`. The `scalarVectProd` operator, given in Script 2.1.21 returns the product of a scalar times a vector, i.e. in this case:

$$2\pi\,(0, 1/16, \ldots, 15/16, 1).$$

Analogously, `TimePoints` contains a sampling with 17 elements of the interval $[0, 4]$, so that the animation generated as value of the `MOVE` expression has a duration of 4 seconds. Notice, looking at the `CScurve` function, that

1. the circular path of the motion has radius $r = 2$;
2. the rotational parameter α is bound to the interval $\Xi_\alpha = [0, -4\pi]$.

Script 15.3.7 (Planar motion)

```
DEF movingCube (tx,ty,alpha::IsReal) = T:<1,2>:<tx,ty>:(cube:alpha);

DEF CScurve = [K:2 * COS, K:2 * SIN, - * K:2];
DEF CSpoints = AA: CScurve: (2*PI scalarVectProd Sampling:16);
DEF TimePoints = 4 scalarVectProd Sampling:16;
DEF WSpath = (polyline ~ AA:[S1,S2,K:0]): CSpoints;

DEF out = STRUCT:<
   MOVE: movingCube: CSpoints: TimePoints,
   WSpath >;

VRML:out:'out.wrl';
```

It is also important to note in this example that either `STRUCT` or `ANIMATION` primitives could equally be used to define the *AnimPolComplex* `out` to be exported to a VRML file.

Example 15.3.6 (Clock animation)
In Figure 15.9 we show four keyframes from the VRML animation generated by Script 15.3.8.

The *animPolComplex* `movie` is generated using the `clock3D` function defined in Script 6.4.4. The animation *behavior* is described in this case by two `<h,m>` (*hour*, *minute*) pairs corresponding to the starting and ending configurations, to be assumed at `0` and `10` seconds, respectively. We note the *extreme ease* of the `PLaSM` approach to animation definition.

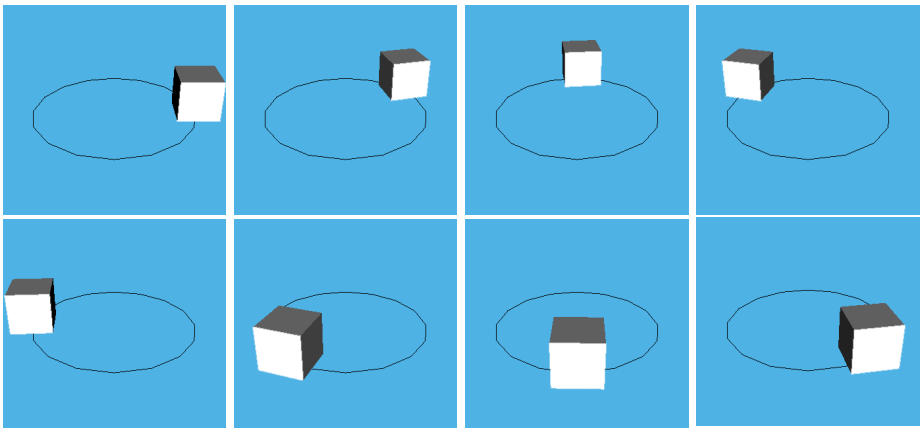

Figure 15.8 A frame sequence from the rotating and translating cube animation

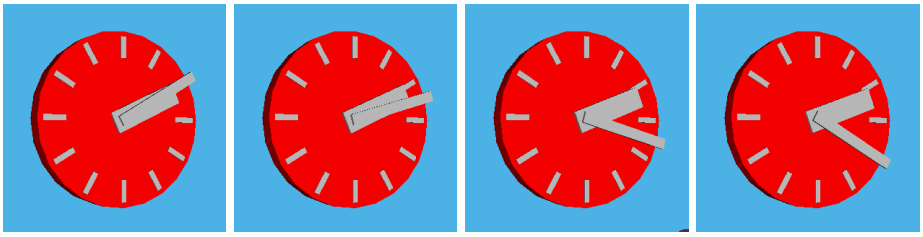

Figure 15.9 Ten minutes in ten seconds ...

15.4 Motion coordination

The graph-theoretic model introduced in [BBC+99] for the design of complex animations is discussed in this section. It provides both a computer representation for the animation *storyboard* as an acyclic directed graph, and a computational technique for time coordination of independently defined animation *segments*. The graph representation and the timing algorithm make reference to the network programming method known as PERT (Program Evaluation and Revision Technique) [Rob63, Ste71] and to the CPM (Critical Path Method), respectively. Both are management techniques for complex projects that are well known to industrial engineers and production managers.

15.4.1 Non-linear animation

Let us start by giving some definitions of terms that are useful to connect the *choreography* approach for the control of very complex animations here presented to the methods currently used in computer animation.

Script 15.3.8 (Clock animation)

```
DEF movie = MOVE: clock3D:<<2,10>,<2,20>>:<0,10>

VRML:movie:'out.wrl';
```

Background The scene part which is time-invariant is called the scene *background*.

Foreground The time-varying portion of an animated scene is called the scene *foreground*.

Storyboard The high-level description of the animation behavior is called the *storyboard*. It is represented as a hierarchical a-cyclic graph with only one node of in-degree zero (called *start* or *source* node) and only one node of out-degree zero (called *end* or *sink* node). The source node will represent the animation *start*. The sink node will represent the animation *end*. The nodes and arcs of the storyboard are also called *events* and *animation segments* (or simply *segments*), respectively.

Segment An animation *segment* is an arc of the storyboard. It represents a foreground portion characterized by the fact that every interaction with the remaining animation is concentrated on the starting and ending events.

Hierarchical animation Hence each segment may be modeled *independently* from the others, by using a *local* coordinate frame for both space and time coordinates. The concepts of storyboard and segment are interchangeable: each complex segment of an animation can be modeled by using a local storyboard, which can be decomposed into lower-level segments.

Event An *event* is a storyboard node. Segments starting from it may begin only when *all* the segments ending in it have finished their execution.

Segment The segment configuration space is defined as

$$XCS = T \times CS$$

where CS is the product space of the interval domains of segment DOFs, and $T = [0, \infty)$ is the time domain.

Geometric model The *geometric model* of a segment is a description of both the assembly structure and the geometry of its elementary parts.

Behavior A continuous curve in segment configuration space XCS is called a *behavior* of the segment. Let

$$b : [0, 1] \rightarrow XCS, \quad \text{with } b(u) = (b_0(u), b_1(u), \ldots, b_d(u))$$

be a behavior curve. Then it must be

$$b_0(u_k) > b_0(u_h) \qquad \text{for each } u_k > u_h.$$

The $d + 1$ dimension of XCS is related to the number d of free parameters of the geometric model of the segment. In order to represent a behavior, sampled curves or splines of suitable degree, parametrized in the $[0, 1]$ interval, are used. A simple and

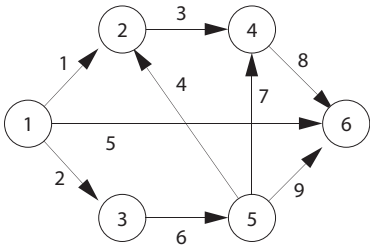

Figure 15.10 Storyboard representation as oriented graph

often useful choice is to use Bézier curves to represent behaviors. In such a case the behavior is completely specified by a point subset in XCS. In particular, any Bézier curve interpolates the first and last points and approximates the other ones. The curve degree is defined by the number of points minus one. So a linear behavior will simply be described by giving two extreme points in XCS.

Actor We call an animation *actor* (or *character*) a connected chain of segments with the same geometric model and with different behaviors. So, at each given time each actor has a unique fixed set of parameters, i.e. a unique configuration.

Choreography In order to independently edit the developed animation segments as a whole, network programming techniques are used. In particular, the dynamic programming algorithm of critical path method is used to compute *minimal* and *maximal times* of events as well the *completing time* of the whole animation *clip*. Such an algorithm is used to compute the timing of actors in each animation segment.

15.4.2 Network programming

The PERT (Program Evaluation and Review Technique) [Rob63, Ste71], also known as *Critical Path Method*, is well-known for managing, i.e. programming and controlling, very complex projects and in particular for scheduling and optimum allocation of resources. Projects may have tens or hundred of thousands of activities and events.

Deterministic PERT for computation of critical activities is probably the most well known variation of network programming techniques, in which a bundle of inter-dependent activities is represented as a directed acyclic graph. Such a graph model is used as the computational basis for project analysis and forecasts.

Storyboard representation A storyboard is represented as a network, i.e. as a directed acyclic graph with only one source node and one sink node. The source node represents the animation start; the sink node the animation end. The arcs represent animation segments; the nodes represent the events of completion of *all* the entering arcs. Notice that segments (arcs) exiting from a node may start only when all the segments (arcs) entering that node have finished.

Minimal and maximal spanning time The *minimal spanning time* (t_k) of a node k is the *minimal* time for completing the segments entering the node k. Notice that

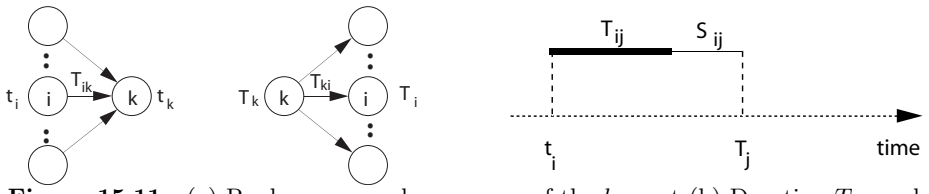

Figure 15.11 (a) Predecessors and successors of the k event (b) Duration $T_{i,j}$ and slack $S_{i,j}$ of the (i, j) segment

such segments *can* be completed into this time. The *maximal spanning time* (T_k) of a node k is the *maximal* time needed for completing the segments entering the node k. Notice that such segments *must* be completed into this time.

The algorithm to compute both minimal and maximal spanning times of nodes is very simple. The computational approach can be classified as an example of dynamic programming. Formally we have:

$$t_k = \max_{i \in \text{pred}(k)} \{t_i + T_{ik}\}, \qquad T_k = \min_{i \in \text{succ}(k)} \{T_i - T_{ki}\} \tag{15.1}$$

The corresponding algorithm can be decomposed into a *forward computation* step and a *backward computation* step.

Forward and backward computation Forward computation of minimal times t_k. Let 0 be the (unique) source node of the network. Set $t_0 = 0$. Then try to compute the minimal time t_e of ending node, i.e. the completion time of the whole project. The recursive formula allows computing of the minimal times t_k of all nodes.

Backward computation of maximal times T_k. Let e be the (unique) sink node of the network. Set $T_e = t_e$. Then try to compute the maximal time T_0 of starting node. The recursive formula allows for computing the maximal times T_k of all nodes.

Segment slacks The *slack* S_{ij} of the segment (i, j) is defined as the quantity of time which may elapse without a corresponding slack of the completion time of the animation. The segment slack S_{ij} is given by the formula

$$S_{ij} = (T_j - t_i) - T_{ij},$$

where T_{ij} is the expected duration of segment (i, j).

Notice that the so-called *critical segments* have null slacks, i.e. $S_{ij} = 0$.

Implementation of CPM

A `PLaSM` implementation of CPM (Critical Path Method) is given in this section. We start (1) by preparing a small toolbox of basic functions and predicates in Script 15.4.1, then (2) we implement in Script 15.4.2 two operators `inarcs` and `outarcs` that return, respectively, the inward and outward arcs of a given node, then (3) we give in Script 15.4.3 two operators `tmin` and `tmax` to compute the minimum and maximum spanning times of nodes; and finally (4) we provide the computation of a small *storyboard* graph.

The graph is represented as follows. It is supposed that only one arc is allowed between each pair of nodes, so that the arc is identified by such ordered pair. We also suppose that the graph contains only one node of indegree 0 (the storyboard start) and only one node of outdegree 0 (the storyboard end).

The graph is here described as a set of triples, one-to-one associated with the arcs. Each triplet (n_i, n_j, t_{ij}) respectively contains the indices of the starting n_i and ending n_j node of the arc, and the scheduled duration t_{ij} of the associated animation segment.

Toolbox Binary predicates `bigger`, `smaller`, `biggest` and `smallest` respectively return `true` if: (a) `b` is larger than `a`; (b) `b` is smaller than `a`; (c) `b` is the largest of `seq` elements; (d) `b` is the smallest of `seq` elements. They return `false` otherwise. Let us remember that the `TREE` primitive is a combinator that recursively applies a binary function over a sequence of arguments of any length.

Script 15.4.1 (CPM toolbox)

```
DEF bigger  (a,b::IsReal) = GT:a:b;
DEF smaller (a,b::IsReal) = LT:a:b;
DEF biggest (a,b::IsReal) = IF:< bigger, s2, s1 >:<a,b>;
DEF smallest (a,b::IsReal) = IF:< smaller, s2, s1 >:<a,b>;
DEF RMAX (seq::IsSeqOf:IsReal) = TREE: biggest: seq;
DEF RMIN (seq::IsSeqOf:IsReal) = TREE: smallest: seq;
```

Operators The partial function `inarc:`n_i, when applied to `arc` $\equiv$ `<`n_k, n_i, t_{ki}`>` triplet, returns the sequence `<`n_k, t_{ki}`>`, denoting the arc as *entering* n_i, and the empty sequence `<>` otherwise. Analogously, partial function `outarc:`n_i, applied to `arc` $\equiv$ `<`n_i, n_j, t_{ij}`>` triplet, returns the pair `<`n_j, t_{ij}`>` for the *outgoing* arc, and `<>` otherwise. The subsets of `graph` arcs entering or leaving n_i is returned by the `inarc:`n_i`:graph` and `outarc:`n_i`:graph` expressions, respectively.

The implementation in Script 15.4.2 was quick for the authors to write, but is pretty inefficient, since its complexity is $O(n^2)$, where n is the number of graph nodes, so it makes sense to use only on small graphs. The reader trained in computer science may develop a more efficient solution.[5]

Script 15.4.2 (Network analysis)

```
DEF inarc (node::IsInt)(arc::IsSeq) =
   IF:< C:EQ:node ~ S2, [[s1,s3]], K:<> >:arc;
DEF outarc (node::IsInt)(arc::IsSeq) =
   IF:< C:EQ:node ~ S1, [[s2,s3]], K:<> >:arc;

DEF inarcs (node::IsInt)(graph::IsSeq) = (CAT ~ AA:(inarc:node)):graph;
DEF outarcs (node::IsInt)(graph::IsSeq) = (CAT ~ AA:(outarc:node)):graph;
```

[5] Hint for the other readers: just sort the arcs on either the second or first nodes, respectively.

Algorithm The recursive algorithms coded in Script 15.4.3 are just a direct PLaSM translation of Formula (15.1) for the forward and the backward network computations.

Script 15.4.3 (Algorithm)

```
DEF tmin (graph::IsSeq)(node::IsInt) = RMAX:predecessorTimes
WHERE
   predecessors = inarcs:node:graph,
   predecessorTimes = IF:< C:EQ:0 ~ LEN,
      K:< Tstart >,
      AA:(+ ~ [tmin:graph ~ S1,S2]) >:predecessors
END;

DEF tmax (graph::IsSeq)(node::IsInt) = RMIN:successortimes
WHERE
   successors = outarcs:node:graph,
   successorTimes = IF:< C:EQ:0 ~ LEN,
      K:< Tstop >,
      AA:(- ~ [tmax:graph ~ S1,S2]) >: successors
END;
```

Example 15.4.1 (Storyboard)
The computation of the sequences of minimal and maximal spanning times (t_k) and (T_k) for the oriented graph coded by the `storyBoard` set of arc triples is provided by Script 15.4.4. The explicit statement of the `lastNode` is needed, as well as the statement of values for `Tstart` and `Tstop` times. Notice that there are some arcs with scheduled duration zero. They are called *dummy arcs* and are used to introduce coordination constraints. The reader should (1) draw the `storyboard` graph; (2) label the arcs with their durations; and (3) annotate the nodes with the spanning times computed below. Notice that the total duration of our storyboard is 19 time units.

Script 15.4.4 (Storyboard example)

```
DEF storyBoard = <<0,1,2>,<1,2,5>,<2,3,3>,<3,4,4>,<1,5,0>,<6,2,0>,
   <2,7,0>,<8,3,0>,<5,6,10>,<6,7,5>,<7,8,2>>;

DEF lastNode = 4;
DEF Tstart = 0;
DEF Tstop = tmin:storyBoard:lastNode;

AA:(tmin:storyBoard):(0..8) ≡ < 0, 2, 12, 19, 23, 2, 12, 17, 19 >
AA:(tmax:storyBoard):(0..8) ≡ < 0, 2, 16, 19, 23, 2, 12, 17, 19 >
```

15.4.3 Modeling and animation cycle

Different clips can be edited (aggregated, time scaled and translated, looped, back-looped, etc.) on the movie *timeline* using the high level animation primitives discussed in Section 15.3.3. A simple but quite complete exercise with such editing primitives is

given in Example 15.6.2.

The modeling and animation methodology summarized below concerns the single animation clip.

1. Clip decomposition into animation segments, and definition of the clip storyboard as a graph.
2. Modeling of geometry and behavior of the animation segments. Each segment will be modeled, animated and tested independently from each other.
3. Non-linear editing of segments by describing their events and time relationships. Segment coordination is computed by using the critical path method.
4. Simulation and parameter calibration of the animation as a whole.
5. Feedback with possible storyboard editing.
6. Starting a new cycle of modeling, editing, calibration and feedback, until a satisfying result is obtained.

Animated lamps

In this section a complete example of scene modeling and animation is discussed. The example aims to resemble the famous *Luxo lamp* animation by Michael Kass and Andrew Witkin (see Foley *et al* [FvDFH90]). In our case two simpler lamps are moving together in animated VRML by describing a quite complex path in their configuration spaces.

The geometric models of the lamp components are generated, and the lamp assembly is defined in local coordinates. Then the storyboard of the animation is given, where the movements of the two actors are both specified and coordinated. The specification of a CS paths for one of animation segments is also given. Notice that the provided code is a quite complete working example, that runs under a `PLaSM` interpreter. If the reader provides the 6 missing CS curves, it may be exported to be displayed by a VRML plug-in supporting the rendering of animations.[6]

Geometry modeling

First of all, some design parameters are defined in Script 15.4.5, in order to easily parametrize the resulting models with respect to some important design dimensions. The two lamps in our storyboard can be thus made different with respect to `basis`/`head` ratio, as happens for humans depending on age.

Script 15.4.5 (Some design parameters)

```
DEF rodHeight = 20;          DEF basisRadius = 20;
DEF rodSide = SQRT:2;        DEF basisHeight = 2;
```

A small toolbox of operators is given in Script 15.4.6. The function `convert` will transform a number sequence from degrees to radiants. The `XCAT` function is

[6] A complete coding may be found in the installed folder `plasm/examples/anim/luxo.psm`

a generalized version of the CAT concatenation operator. A further variation of the circle definition allows generation of *circle arcs* with variable angle a, radius r and number of approximating segments n. The truncated cone generator TrunCone depends on the bottom and top radiuses r1, r2 and on the height h, as well as on the number n of approximating facets. In addition, the Q generalized shortcut for QUOTE given in Script 1.5.5 is also used in the following scripts.

Script 15.4.6 (Toolbox)

```
DEF XCAT = CAT ~ AA:(IF:<IsSeq,ID,LIST>);
DEF convert (seq::IsSeq) = (AA:* ~ DISTL):< PI/180, seq >;
DEF circlesector (a::IsReal) (r::IsReal) (n::IsInt) =
   (S:<1,2>:<r,r>~JOIN):(MAP:([cos,sin]~s1): (intervals:a:n);

DEF TrunCone (r1,r2,h::IsReal)(n::IsInt) =
   MAP:[ x * cos ~ s2, x * sin ~ s2, z ]:
      (QUOTE:<1> * (QUOTE ~ #:n):(2*PI/n))
WHERE
   x = K:r1 + s1 * ( K:r2 - K:r1 ),
   y = K:0,
   z = s1 * K:h
END;
```

Then the geometric model of both the maleJoint and femaleJoint and the doubly JointedRod of the lamp are specified in Script 15.4.7, starting from the 2D halfcircular shape of halfHinge2D. Notice that the chain of infix operators in JointedRod is left-associative.

Script 15.4.7 (Subcomponent modeling)

```
DEF halfHinge2D = circlesector:PI:1:12;
DEF hinge2D = STRUCT:< halfHinge2D, T:<1,2>:<-1,-3>,Q:2 * Q:3 >;
DEF hinge = (MKPOL ~ UKPOL):(hinge2D * Q:0.5);
DEF DoubleHinge = STRUCT:<hinge, T:3:1.2, hinge>;
DEF hbasis = circle:1.2:<24,1> * Q:2;
DEF femaleJoint = STRUCT:<
   T:3:-5:hbasis, T:2:0.85,R:<2,3>:(PI/2):DoubleHinge>;
DEF maleJoint = STRUCT:< R:<2,3>:PI,
   T:3:-5:hbasis, T:2:0.25,R:<2,3>:(PI/2):Hinge>;
DEF rod = T:<1,2>:<rodSide/-2,rodSide/-2>:
   (CUBOID:<rodSide,rodSide,rodHeight>);
DEF JointedRod = maleJoint TOP rod TOP femaleJoint COLOR GREEN;
```

The basis object and the head generating function are given in Script 15.4.8. The function head depends on the integer parameter that specifies the number of approximating facets of TrunCone. The strange design choices of specifying basis and head so much differently, is motivated by the desire to show the great pervasiveness of AnimBehaviour type parameters (see Setion 15.3.3) through the language constructs.

And finally the Luxo generating function depending on three joint angles a1, a2

Script 15.4.8 (Part modeling)

```
DEF basis =
   (circle:basisRadius:<32,1> * Q:basisHeight)
   TOP femaleJoint;

DEF head = STRUCT ~ [ K:maleJoint, K:(T:3:5),
   embed:1 ~ circlesector:(2*PI):4,
   TrunCone:<4,4,8>, K:(T:3:8),
   TrunCone:<4,20,20> ];
```

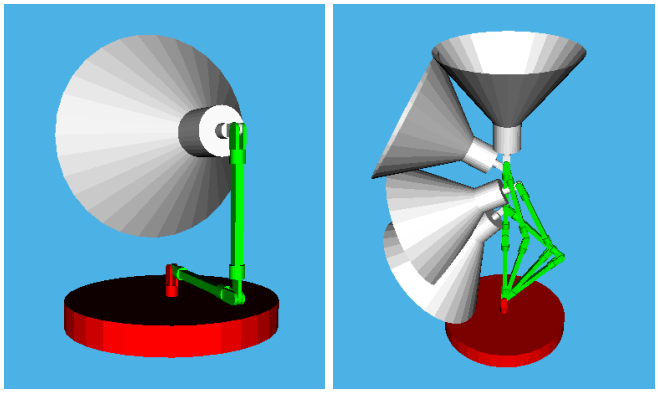

Figure 15.12 (a) The lamp configuration generated by `(Luxo ~ convert):<-90,90,90>)` (b) Some superimposed key-frames

and `a3` and with red `basis` and white `head` is given in Script 15.4.9 and displayed in Figure 15.12. Remember that `convert` provides the degrees to radians numeric conversion.

Script 15.4.9 (Lamp assembly modeling)

```
DEF Luxo (a1,a2,a3::IsReal) = STRUCT:<
   basis COLOR RED,
   T:3:(basisHeight+5), R:<1,3>:a1, JointedRod,
   T:3:(rodHeight+10), R:<1,3>:a2, JointedRod,
   T:3:(rodHeight+10), R:<1,3>:a3, head:32 COLOR WHITE
>;

DEF out = Luxo:(convert:<-90,90,90>);
VRML:out:'out.wrl'
```

Motion modeling

Actor definition As we know, a mobile object on the plane has 3 degrees of freedom: a rotation and two translations, to be applied in this order. The other 3 degrees of freedom are the internal joint angles. The two actors of our clip are defined in Script 15.4.10. Notice that the scaling tensor in `LuxoSon` is applied to the lamp model

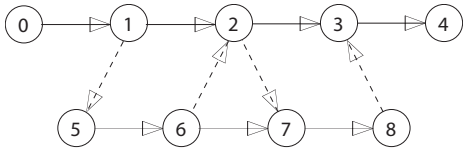

Figure 15.13 Storyboard representation as oriented graph. Dummy arcs ($t_{ij} = 0$) are dashed

before applying the rotation and translation tensors.

Script 15.4.10 (Mobile lamps)

```
DEF LuxoFather (a1,a2,a3, a4,a5,a6::IsReal) = STRUCT:<
   T:1:a1, T:2:a2, R:<1,2>:a3, Luxo:<a4,a5,a6>
>;

DEF LuxoSon (a1,a2,a3, a4,a5,a6::IsReal) = STRUCT:<
   T:1:a1, T:2:a2, R:<1,2>:a3, S:<1,2,3>:<0.7,0.7,0.7>, Luxo:<a4,a5,a6>
>;
```

Storyboard definition The clip storyboard is given as an oriented acyclic graph, with animation segments associated with the arcs and (coordination) events associated with the nodes. The expected duration of some animation segments are directly given as `PLaSM` definitions. In Figure 15.13 the storyboard representation as an abstract directed graph is given. A projection in $\mathbb{E}^2$ of the storyboard embedded in configuration space, corresponding to the 2 translational degrees of freedom, is given in Figure 15.13.

Script 15.4.11 (Segment durations)

```
DEF Time_0_1 = 3;        DEF Time_5_6 = 10;
DEF Time_1_2 = 5;        DEF Time_6_7 = 5;
DEF Time_2_3 = 3;        DEF Time_7_8 = 2;
DEF Time_3_4 = 4;
```

Animation timing The minimal spanning times t_i needed for starting and ending animation segments are denoted as `t0, ... ,t8`; The maximal spanning times T_j are denoted as `tt0, ... ,tt8`.

Forward computation of minimal spanning times of coordination events is given in Script 15.4.12, where `RMAX`, `RMIN` are pre-defined `PLaSM` operators to compute maximum and minimum values of a set of reals, and are defined in Script 15.4.1. The maximal spanning times T_i of nodes may be analogously computed by the functional environment of the language. The use of an explicit CPM implementation, given in Section 15.4.2, is not actually needed for moderately simple animation projects.

Script 15.4.12 (Min forward and max backward times)

```
DEF t0 = 0;                          DEF tt0 = tt1 - Time_0_1;
DEF t1 = t0 + Time_0_1 ;             DEF tt1 = RMIN:<tt2 - Time_1_2, tt5>;
DEF t2 = RMAX:<t1 + Time_1_2, t6>;   DEF tt2 = RMIN:<tt3 - Time_2_3, tt7>;
DEF t3 = RMAX:<t2 + Time_2_3, t8>;   DEF tt3 = tt4 - Time_3_4;
DEF t4 = t3 + Time_3_4 ;             DEF tt4 = t4;
DEF t5 = t1;                         DEF tt5 = tt6 - Time_5_6;
DEF t6 = t5 + Time_5_6 ;             DEF tt6 = RMIN:<tt7 - Time_6_7, tt2>;
DEF t7 = RMAX:<t6 + Time_6_7, t2>;   DEF tt7 = tt8 - Time_7_8;
DEF t8 = t7 + Time_7_8 ;             DEF tt8 = tt3;
```

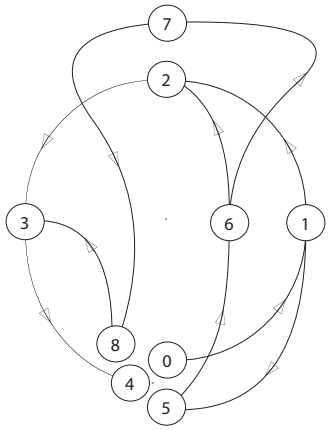

Figure 15.14 Projection of CS paths in the coordinate subspace of basis translation parameters

Fluidity constraint A smooth and gracefully flowing of all animation segments in a clip is achieved if we use as starting and ending times of each `MOVE` expression the averages $\texttt{tm}_i$ of minimal and maximal spanning times of nodes. Such a fluidity constraint of the whole animation clip *holds for every possible choice of scheduled durations of animation segments* [BBC+99]. Such average times are computed explicitly in Script 15.4.13. This approach greatly helps in designing complex choreographies by using independently developed actors.

Script 15.4.13 (Scheduled times)

```
DEF tm0 = (t0 + tt0) / 2;      DEF tm5 = (t5 + tt5) / 2;
DEF tm1 = (t1 + tt1) / 2;      DEF tm6 = (t6 + tt6) / 2;
DEF tm2 = (t2 + tt2) / 2;      DEF tm7 = (t7 + tt7) / 2;
DEF tm3 = (t3 + tt3) / 2;      DEF tm8 = (t8 + tt8) / 2;
DEF tm4 = (t4 + tt4) / 2;
```

Segment's CS paths For each animation segment the configuration space path must be given. This can be done, e.g., as a Bézier curve. One of them, for segment $(0, 1)$, is given in Script 15.4.14. The other CS paths can be specified similarly, by

giving some points in 6-dimensional configuration space of the animation. The reader is challenged in giving by itself such paths for each non-dummy arc in animation graph, and by looking carefully at the clip results. Notice that the `CS_0_1` object given below is a Bézier generating function of degree 3.

Script 15.4.14 (CS paths)

```
DEF CS_0_1 = AA:((Bezier:S1 ~ AA:XCAT):<
   <100,0, convert:<0, 0,0,0> >,
   <150,0, convert:<30, 30,0,-10> >,
   <200,50, convert:<-150, -20,90,0> >,
   <200,100, convert:<90+180, -60,105,60> >
> );
```

Clip definition Our Luxo's `clip` concerning both mobile lamps is exported to `vrml` by the last expression of Script 15.4.15. Notice that the `xtime` function is used to transform a sequence of time points into a sampling of the 1D Bézier curve generated by that points. Notice also that each non-dummy segment of the storyboard shown in Figure 15.13 corresponds to a `MOVE` expression in the `ANIMATION` container.

Script 15.4.15 (Animation scripting)

```
DEF xtime (tseq::IsSeqOf:IsReal) = (CAT ~ AA:(Bezier:(AA:LIST:tseq)));
DEF father = MOVE:LuxoFather;
DEF son = MOVE:LuxoSon;
DEF points = sampling:10;

DEF clip = ANIMATION:<
   father:(CS_0_1:points):(xtime:<tm0,tm1>:points),
   father:(CS_1_2:points):(xtime:<tm1,tm2>:points),
   father:(CS_2_3:points):(xtime:<tm2,tm3>:points),
   father:(CS_3_4:points):(xtime:<tm3,tm4>:points),

   son:(CS_5_6:points):(xtime:<tm5,tm6>:points),
   son:(CS_6_7:points):(xtime:<tm6,tm7>:points),
   son:(CS_7_8:points):(xtime:<tm7,tm7+0.6*(tm8-tm7),
      tm7+0.8*(tm8-tm7),tm8>:points)
>;
```

The VRML exporting of the Luxo's `clip` mounted over a static and textured plane support is done in Script 15.4.16. Both `TEXTURE` and `SIMPLETEXTURE` operators are contained in the `psmlib/colors.psm` library. Some frames from the rendering of the exported VRML `clip` are given in Figure 15.15. The reader should notice that `ANIMATION` and `STRUCT` primitives may be freely nested into each other. The path of the file `gioconda.jpg` is relative to the `plasm` folder installed on the user machine. It should be suitably changed if the user has no writing permissions on the `plasm` folder, where the `luxo.wrl` is going to be exported by Script 15.4.16.

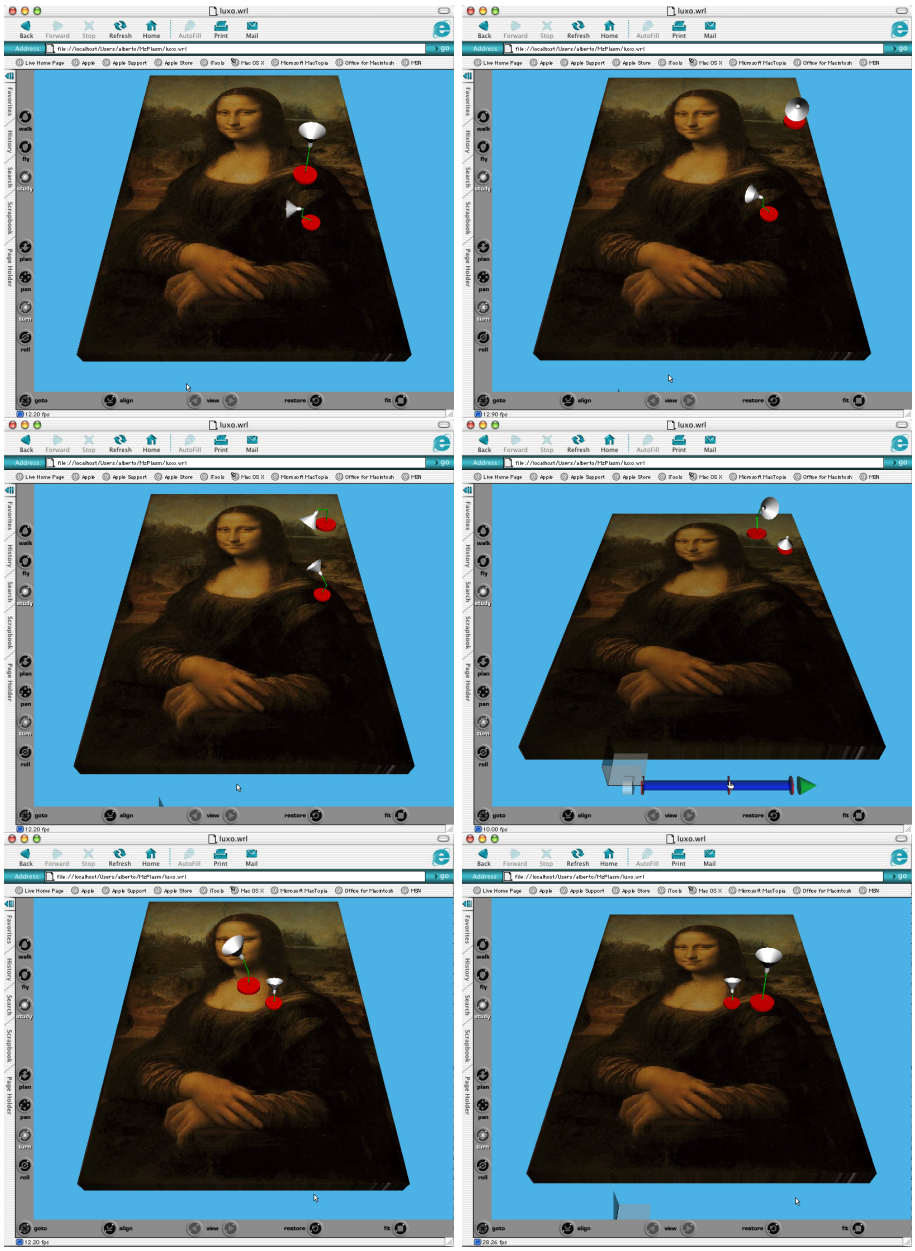

Figure 15.15 Some frames of our *Luxo*'s parody over Mona Lisa. In one on frames the widget for interactive control of the animation rendering is also shown. The animated VRML code generated by `PLaSM` is displayed using on *MacOS X* the *Cortona* plug-in by Parallelographics and Microsoft's *Internet Explorer*.

Script 15.4.16 (VRML exporting)

```
DEF Gioconda = SIMPLETEXTURE:'examples/color/img/gioconda.jpg';

DEF out = STRUCT:< clip,
   (T:<1,2,3>:<-8,-12,-1> ~ S:<1,2>:<16,24> ~ CUBOID):<1,1,1>
   TEXTURE Gioconda >;

VRML:out:'luxo.wrl';
```

15.5 Extended Configuration Space

In this section we discuss, implement and exemplify a general and simple geometrical method for solving a difficult problem: the computation of a polyhedral approximation of the *free configuration space* FP for a robot system R moving in a working space containing obstacles E. This method was introduced in [Pao89].

15.5.1 Introduction

Let the *mobile system* $R \cup E \subset \mathbb{E}^d$ be composed of a set $R = \{R_i\}$ of mobile rigid parts with k degrees of freedom, that we will call the *robot*, and a set $E = \{E_i\}$ of rigid *obstacles*. Both the robot and the obstacles are usually 2D or 3D. If the obstacles move along known trajectories, the whole system can be statically modeled taking time into account as an additional dimension, and by embedding the system in a 3D or 4D spacetime, respectively.

The symbol P^d will be used to refer to both R_i and E_i, being them represented as d–dimensional polyhedra.

Consider the configuration space $CS \subseteq \mathbb{R}^k$ and the working space $WS \subseteq \mathbb{E}^d$ of the robot. We define the *extended configuration space map* as a function

$$ECS : WS \to WS \times CS \subseteq \mathbb{R}^{d+k}.$$

The set FP of free positions of R may be straightforwardly computed when $ECS(R)$ and $ECS(E)$ are provided. The first term is a polyhedral encoding of all the possible placements of R allowed by its degrees of freedom, whereas the second term is the result of a straight k-extrusion of obstacles, which do not depend on the degrees of freedom.

We like to remark that the distinction between robot and obstacles is softened in this approach. It does not make any difference if either the polyhedron P^d is part of an articulated manipulator or it is an obstacle: the distinctive property is the association between objects in the scene and degrees of freedom.

15.5.2 Rationale of the method

The extrusion operations are defined in such a way that the point

$$q = (x_1, \ldots, x_d, t_1, \ldots, t_k) \in \mathbb{R}^{d+k}$$

belongs to $ECS(P^d) = P^{d+k}$ if and only if the projection of q within the workspace, denoted as $\Pi_{\mathrm{WS}}(q) = (x_1, \ldots, x_d)$, belongs to the placement of P^d corresponding to

the parameters value $(t_1, \ldots, t_k)$. We can write:

$$q \in P^{d+k} \Longleftrightarrow \Pi_{\mathrm{WS}}(q) \in P^d \mid_{(t_1,\ldots,t_k)}$$

It follows that, if the ECS images P_i^{d+k} and P_j^{d+k} of $P_i^d, P_j^d \in R \cup E$ have a common point q, then P_i^d and P_j^d overlap or touch in the configurations corresponding to the last k coordinates of q:

$$q \in P_i^{d+k} \cap P_j^{d+k} \Longleftrightarrow \Pi_{\mathrm{WS}}(q) \in P_i^d\big|_{(t_1,\ldots,t_k)} \cap P_j^d\big|_{(t_1,\ldots,t_k)}$$

Imagine computing the intersection set of each pair

$$(P_i^{d+k}, P_j^{d+k}) \in \mathrm{ECS}(R \cup E) \times \mathrm{ECS}(R \cup E),$$

to project such sets onto CS and to take the union of the projections. This yields the *configuration space obstacles* CSO, i.e. the set of points in configuration space which correspond to prohibited configurations of the system. Recalling that *free configuration space* FP coincides with the difference between configuration space and configuration space obstacles, we conclude that FP can be computed by looking for intersections in ECS, and then by subtracting their projection from configuration space [Pao89].

In a single formula we can write:

$$\mathrm{FP} = \mathrm{CS} - \mathrm{CSO} = \mathrm{CS} - \bigcup_{i,j} \Pi_{\mathrm{CS}}(ECS(P_i^d) \cap ECS(P_j^d))$$

15.5.3 ECS algorithm

The conceptual skeleton of the ECS method for the computation of the free configuration space is summarized in Figure 15.16. A more detailed statement of the algorithm follows.

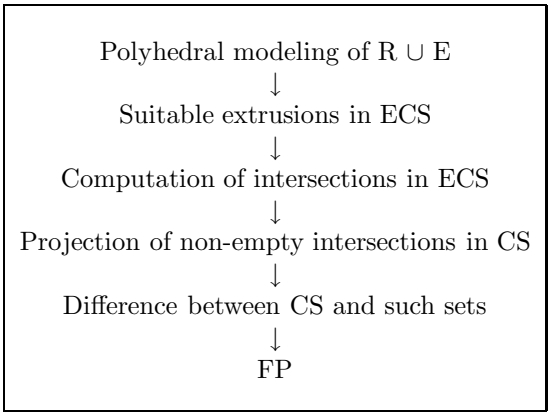

Figure 15.16 Conceptual skeleton of the ECS method.

Modeling step A suitable solid model of $R \cup E$ is given in the workspace WS. Convex decompositions are used to represent the system; this choice allows us to easily compute the ECS image of each part in the scene though the convex hulls generated by the `JOIN` operator.

Extrusion step The image of each $P^d \in R \cup E$ in the Extended Configuration Space is computed. This results in a set of higher dimensional polyhedra P^{d+k}, where for each P^{d+k} a convex decomposition within ECS is available.

Intersection step The set intersection of each pair P_i^{d+k}, P_j^{d+k} is computed. Such sets correspond either to auto-intersections of an articulated robot or to intersections of the latter with obstacles. Since a quasi-disjoint decomposition of the operands is given, intersection can be distributed and performed by pairwise intersecting convex cells. The intersection of convexes is a convex set, so that this step results in a set of quasi-disjoint convex cells, $\{c_i^{d+k} | c_i^{d+k} \subset \mathrm{ECS}\}$.

Projection step Each convex cell $c_i^{d+k} \subset \mathrm{ECS}$ is projected onto CS. This process can be thought of as a sequence of d elementary projections, each one performed along a coordinate direction. The result is the set

$$\left\{ c_i^k | c_i^k = \Pi_{\mathrm{CS}}(c_i^{d+k}) \right\}.$$

Since the projection of a convex set is convex, a collection of convex sets in configuration space is obtained. Unfortunately, these are no longer quasi-disjoint, but give a set covering of configuration space obstacles.

Difference step The free configuration space for the moving system $R \cup E$ is finally obtained as:

$$\mathrm{FP} = \mathrm{CS} - \bigcup_i c_i^k.$$

15.5.4 Encoding degrees of freedom

In this subsection it is discussed how to generate $ECS(R)$ and $ECS(E)$, i.e. how to encode the degrees of freedom of the mobile polyhedral system and the polyhedral scene itself within higher dimensional polyhedra.

Each one of the rigid parts which compose the robot, say P^d, may move according either to translational degrees of freedom only, or to rotational only, or to both translational and rotational degrees of freedom. If the position of P^d depends on the value of a parameter t_i (α_i), we say that P^d is *subject* to t_i (α_i). The representation P^{d+k} of P^d, which encodes the whole set of k degrees of freedom, is computed by performing an appropriate sequence of k suitable extrusion operations.

Types of extrusion

A definition of straight, linear and screw extrusions, recalled from [PBCF93], is given in the following. For each translational degree of freedom (with parameter t_i) and for

each rotational degree of freedom (with parameter α_i) in the scene, the following rules are applied:

1. If P^d is subject to t_i, then an appropriate *linear extrusion* is required, which generates the whole set of positions corresponding to the translational degree of freedom (see Figure 15.17b).
2. If P^d is subject to α_i, an appropriate *screw extrusion* is performed, which generates the whole set of positions and orientations corresponding to the rotational degree of freedom (see Figure 15.17c).
3. If P^d is subject neither to t_i nor to α_i, an appropriate *straight extrusion* is required, which encodes the independency of P^d from the considered degree of freedom (see Figure 15.17a).

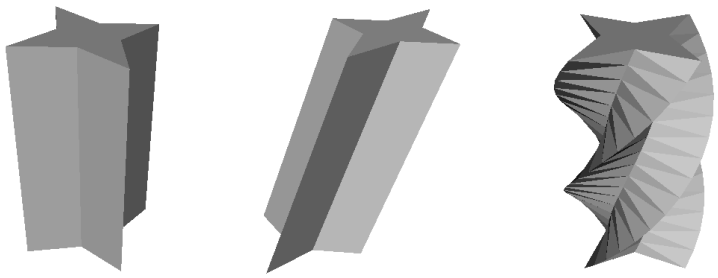

Figure 15.17 3D polyhedral encoding of a degree of freedom of a 2D object: (a) straight extrusion (independence on parameter) (b) linear extrusion (translational DOF) (c) screw extrusion (rotational DOF)

Notice that, according to the above rules, the sequence of extrusions to be applied to P^d is partly determined by the degrees of freedom of P^d itself, and partly by the degrees of freedom of other bodies in the scene.

Definitions of the straight, linear and screw extrusion operators (E_h, $LE_{\mathbf{v},h}$ and $SE_{\theta,i,j,h}$, respectively), are given below. As is shown, a linear extrusion can be computed by composition of a straight extrusion and of an affine transformation.

Straight extrusion A *straight extrusion* is a mapping

$$E : \mathcal{P}^{d,d} \rightarrow \mathcal{P}^{d+1,d+1}$$

between polyhedral spaces, such that $P^d \mapsto P^d \times [0, 1]$.

Linear extrusion A *linear extrusion* is a mapping

$$LE : \mathcal{P}^{d,d} \times \mathbb{R} \rightarrow \mathcal{P}^{d+1,d+1}$$

such that $(P^d, t_i) \mapsto (\boldsymbol{H}_{d+1}(t_i) \circ E)(P^d)$, where $\boldsymbol{H}_{d+1}(t_i) \in \text{lin}\,(\mathbb{R}^{d+1})$ is a *shearing* tensor with matrix (η_{ij}), where:[7]

$$\eta_{ij} = \begin{cases} t_i, & i = 1, j = d+1 \\ \delta_{ij}, & \text{(Kröneker symbol) elsewhere} \end{cases} \tag{15.2}$$

Screw extrusion A *screw extrusion* is a mapping

$$SE : \mathbb{R} \times N^+ \times \mathcal{P}^{d,d} \to \mathcal{P}^{d+1,d+1}$$

where N^+ is the set of positive integers, and such that

$$(\alpha, h, P^d) \mapsto SE(P^d) = P^{d+1},$$

where $SE(P^d)$ is produced by the algorithm given below.

Algorithm A sequence of computations needed to generate the dimension-independent *screw extrusion* $SE(P^d)$ of the P^d polyhedron is discussed in the following.

First, the set of polyhedra

$$\mathcal{S}^{d,d+1} = P^d \times \left\{ \frac{i}{h},\ i = 0, \ldots, h \right\},$$

with $\mathcal{S}^{d,d+1} \subset \mathcal{P}^{d,d+1}$, called a set of *polyhedral d-slices* in $\mathbb{E}^{d+1}$ space, is subject to the action of a discrete family of parametric rotation tensors

$$\mathcal{R} = \{ \boldsymbol{R}_{d-1,d} \left(\alpha \frac{i}{h} \right),\ i = 0, \ldots, h \}$$

depending on the same parameter i, thus giving rise to the family of *rotated d-slices*

$$\mathcal{RS}^{d,d+1} = \{ Q_i^d = \boldsymbol{R}_{d-1,d} \left(\alpha \frac{i}{h} \right) \left(P^d \times \left\{ \frac{i}{h} \right\} \right),\ i = 0, \ldots, h \} \subset \mathcal{P}^{d,d+1}.$$

Second, a *joinCells* operation is applied to each element of the set of pairs of rotated slices with adjacent indices, thus generating a set of *solid* $(d+1)$-dimensional *layers* S_i^{d+1} as the pairwise convex hulls of convex cells belonging to different hyperplanes:

$$\mathcal{S}^{d+1,d+1} = \{ S_i^{d+1} = joinCells(Q_i^d, Q_{i+1}^d),\ i = 0, \ldots, h-1 \} \subset \mathcal{P}^{d+1,d+1}.$$

Finally, the screw-extruded polyhedron $SE(P^d)$ is generated by quasi-disjoint union of solid $(d+1)$-slices:

$$SE(P^d) = \bigcup\ \mathcal{S}^{d+1,d+1} \in \mathcal{P}^{d+1,d+1}.$$

---

[7] For the Kröneker symbol see Section 3.3.3.

Implementation (1) A dimension-independent implementation of the set of extrusion operators defined above is implemented in Scripts 15.5.1 and 15.5.2.

A slightly different strategy is used in the implementation of the *SE* operator. In particular, the input `pol` is decomposed into a set $\{c_k\}$ of convex cells by the `SpliCells` operator given in Script 10.8.4, and a single

$$\texttt{slice} = \{c_k,\ k = 1, \ldots, i_k\} \times \{0\}$$

is produced. Then a single solid `layer` is generated by pairwise `JOIN` of corresponding cells of `slice` and of a copy of it properly rotated and translated by a suitable `tensor`. Finally, the polyhedral result is obtained by `STRUCT` aggregation of properly rotated and translated copies of the single `layer`, exploiting at this purpose the `STRUCT` semantics.

The (homogeneous version of) shearing tensor $\boldsymbol{H}_{d+1}(t_i) \in \mathrm{lin}\,(\mathbb{R}^{d+1})$ described by equation (15.2), and needed to implement the *LE* operator, is quite straightforwardly given by the `Shear` function. The `IDNT` operator, that yields the identity matrices of arbitrary dimensions, is given in Script 3.3.5.

Script 15.5.1 (Extrusion toolbox)

```
DEF Extrusion (angle::IsReal)(h::IsInt)(pol::Ispol) =
   (STRUCT ~ CAT ~ #:h):< layer, tensor >
WHERE
   slice = (AA:(EMBED:1) ~ SpliCells):pol,
   tensor = T:(d+1):(1/h) ~ R:< d - 1,d >:(angle/h),
   layer = (STRUCT ~ AA:JOIN ~ TRANS ~ [ID, AA:tensor]):slice,
   d = DIM:pol
END;

DEF Shear (t::Isreal)(pol::Ispol) = (MAT ~ Update ~ IDNT):(d+1):pol
WHERE
   update = < S1, newrow > (CONS ~ CAT) AA:SEL:(3..d+1),
   newrow = K:(<0,1> CAT #:(d - 2):0 AR t),
   d = DIM:pol
END;
```

Implementation (2) A revised definition of the *straight*, *linear* and *screw* extrusion operators is used for the implementation of `EX`, `LEX` and `SEX` operators given in Script 15.5.2. The purpose is to directly allow for encoding the degrees of freedom in a non-normalized fashion, using directly the *joint limits* associated with each feasibility interval $[\xi_i^{min}, \xi_i^{max}]$ of the t_i or α_i parameters. Therefore we have:

$$\texttt{EX} : \mathbb{R}^2 \times \mathcal{P}^{d,d} \to \mathcal{P}^{d+1,d+1}$$

such that $((\xi_i^{min}, \xi_i^{max}), P^d) \mapsto (\boldsymbol{T}_{d+1}(\xi_i^{min}) \circ \boldsymbol{S}_{d+1}(\xi_i^{max} - \xi_i^{min}) \circ E)(P^d)$, where $\boldsymbol{T}_n(a)$ and $\boldsymbol{S}_n(b)$ respectively denote the translation and scaling tensors along the n-th coordinate direction. Analogously, for the linear extrusion we set

$$\texttt{LEX} : \mathbb{R}^2 \times \mathcal{P}^{d,d} \to \mathcal{P}^{d+1,d+1}$$

such that $((\xi_i^{min}, \xi_i^{max}), P^d) \mapsto (\boldsymbol{T}_{d+1}(\xi_i^{min}) \circ \boldsymbol{S}_{d+1}(\xi_i^{max} - \xi_i^{min}) \circ \boldsymbol{H}_{d+1}(\xi_i^{max} - \xi_i^{min}) \circ E)(P^d)$, where $\boldsymbol{H}_n(a)$ is the shearing tensor (15.2). Finally, the screw extrusion is redefined as:

$$\texttt{SEX} : \mathbb{R}^2 \times N^+ \times \mathcal{P}^{d,d} \rightarrow \mathcal{P}^{d+1,d+1}$$

such that $((\xi_i^{min}, \xi_i^{max}), h, P^d) \mapsto (\boldsymbol{R}_{d-1,d}(\xi_i^{min}) \circ \boldsymbol{S}_{d+1}(\xi_i^{max} - \xi_i^{min}) \circ SE)(\xi_i^{max} - \xi_i^{min}, h, P^d)$.

The above definitions of the `EX`, `LEX` and `SEX` operators are implemented very easily in Script 15.5.3. The reader should remember that a sequence of compositions is applied, as always, in reverse order.

Script 15.5.2 (Extrusion operators)

```
DEF EX (x1,x2::IsReal)(pol::Ispol) =
    ( T:(DIM:pol+1):x1
   ~ S:(DIM:pol+1):(x2 - x1)
   ~ Extrusion:0:1 ): pol;

DEF LEX (x1,x2::IsReal)(pol::Ispol) = ( MKPOL ~ UKPOL
   ~ T:(DIM:pol+1):x1
   ~ S:(DIM:pol+1):(x2 - x1)
   ~ shear:(x2 - x1)
   ~ Extrusion:0:1 ): pol;

DEF SEX (x1,x2::IsReal)(h::IsIntPos)(pol::Ispol) =
    ( R:<d, d - 1>:x1
   ~ S:(d + 1):(x2 - x1)
   ~ Extrusion:(x2 - x1):h ): pol
WHERE d = DIM:pol END;
```

Example 15.5.1 (Straight, linear and screw extrusion)
Three simple examples of straight, linear and screw extrusion of an empty square are produced by Script 15.5.3 and are displayed in Figure 15.18.

Script 15.5.3

```
DEF pol1 = (T:<1,2>:<-5,-5> ~ CUBOID):<10,10>;
DEF pol2 = S:<1,2>:<0.9,0.9>: pol1;
DEF pol3 = pol1 - pol2;

VRML:( EX:<0,10>: pol3):'out1.wrl';
VRML:(LEX:<0,10>: pol3):'out2.wrl';
VRML:(SEX:<0,PI>:16: pol3):'out3.wrl';
```

15.5.5 Computation of FP

In this section we discuss and implement some examples of FP computation where dim $ECS = d + k$ is equal to $3, 4$ and 5, respectively. In actual cases the embedding

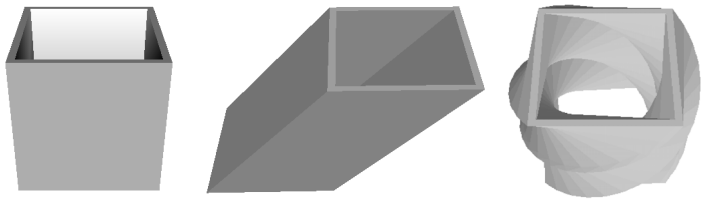

Figure 15.18 Encoding of a DOF via straight, linear and screw extrusions produced by `EX`, `LEX` and `SEX` operators

geometrical space has dimension $d \leq 4$, where $d = 4$ arises when modeling with a spacetime approach. Conversely, $k = \dim\ CS$ may take an arbitrary (small) value, depending on the structure of the moving system.

1D moving system ($d = 1,\ k = 2$)

Let us consider a one-dimensional moving system $R = \{R_1, R_2\}$, with both elements equal to the unit segment $[0, 1] \subset \mathbb{R}$, and constrained to translate inside the interval $\Xi = [0, 4]$ of the x axis. Such a system has 2 degrees of freedom, defined by the translations t_1 and t_2 of the first point of the segments.

ECS representation A representation of this system in the extended space (x, t_1, t_2) is obtained as follows. The set of placements of the R_1 segment is given on the plane (x, t_1) by a parallelogram generated by linear extrusion of R_1. Since it does not depend on t_2, its extended representation in (x, t_1, t_2) space is given by straight extrusion.

The computer representation with `PLaSM` is produced in Script 15.5.4 by applying a suitable shearing tensor to the parallelepiped of size `<1,4,4>`. The dimetric projections of the volumes $\mathcal{S}(R_1), \mathcal{S}(R_1) \subset ECS$ are produced by the last two expressions of the script, having provided the loading of the `viewmodels` library.

Script 15.5.4 (ECS example (1))

```
DEF shear1 = MAT:<<1,0,0,0>,<0,1,3/4,0>,<0,0,1,0>,<0,0,0,1>>;
DEF shear2 = MAT:<<1,0,0,0>,<0,1,0,3/4>,<0,0,1,0>,<0,0,0,1>>;
DEF R1 = (shear1 ~ CUBOID):<1,4,4>;
DEF R2 = (shear2 ~ CUBOID):<1,4,4>;

projection:parallel:dimetric:R1;
projection:parallel:dimetric:R2;
```

A projected view of the polyhedral volume encoding the whole set of placements of R_1 is shown in Figure 15.19a. The polyhedral encoding of the set of placements of R_2 is derived analogously, and is shown in Figure 15.19b.

A parallel projection of the set union and intersection of extended objects $\mathcal{S}(R_1)$ and $\mathcal{S}(R_2)$, produced by Script 15.5.5, respectively, is given in Figure 15.20.

Finally, in Script 15.5.5 a polyhedral representation of free positions FP is generated, by set difference of configuration space CS and the intersections of extended obstacles

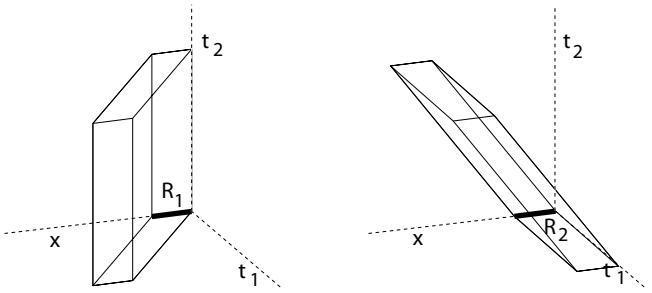

Figure 15.19 Representations $\mathcal{S}(R_1), \mathcal{S}(R_2) \subset ECS$ of the moving elements R_1 and R_2

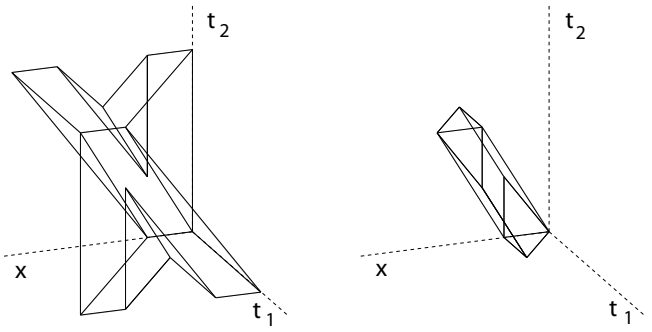

Figure 15.20 Set union and intersection of extended objects $\mathcal{S}(R_1)$ and $\mathcal{S}(R_2)$

projected in CS. The `project` operator, used to project a higher dimensional object onto the coordinate subspace defined by the `coords` subset of coordinate indices is also given in Script 15.5.6. The resulting FP is shown in Figure 15.21.

1D moving system with obstacle ($k = 2$)

Finally, we compute FP with the same approach in presence of a 1D obstacle in working space WS $= [0, 4]$. First, we assume as obstacle the interval $[2, 2.5] \subset$ WS, whose extended representation in WS $\times$ CS is given by the extended configuration space obstacle `ECSO` of Script 15.5.7. The union of intersections between the (extended representations of) mobile components and between them and `ECSO` is given by `UFP`. This extended representation of *unfeasible positions* is shown in Figure 15.22.

The FP set is finally computed in Script 15.5.7 and shown in Figure 15.23a for the $[2, 2.5]$ obstacle and in Figure 15.23b for the $[2.25, 2.75]$ obstacle.

Script 15.5.5 (ECS example (2))

```
projection:parallel:dimetric:(R1 + R2);
projection:parallel:dimetric:(R1 & R2);
```

Script 15.5.6 (ECS example (3))

```
DEF project (coords::IsSeq) =
   MKPOL ~ [AA:((CONS ~ AA:SEL):coords) ~ S1, S2, S3] ~ UKPOL;

DEF CS = CUBOID:<4,4>;
DEF FP = CS - project:<2,3>:(R1 & R2);

VRML:((STRUCT ~ [ID, @1]):FP):'out.wrl';
```

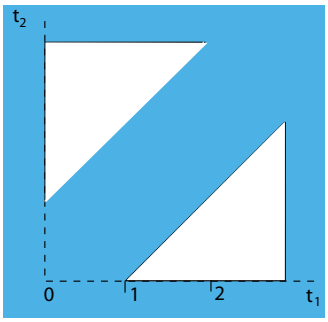

Figure 15.21 Free configuration space FP (the white areas) for the unconstrained 1D example.

15.6 Examples

Some simple examples of animation and motion planning are given in this section. In particular we discuss: (a) a possible animation of or the umbrella modeling that we developed through the book; (b) a fast modeling of a *non-holonomic*, i.e. with non independent degrees of freedom, *planar motion* of a car; (c) the motion modeling of an *anthropomorphic robot* with several DOFs; (d) the *motion planning* problem that arises when solving a 2D labyrinth, by using the extended configuration space method;

15.6.1 Umbrella modeling (5): animation

An animation of the umbrella model defined in Script 8.5.14, and later refined in 11.5.2 and 12.6.1, is given in Script 15.6.1. Some keyframes from the generated movie are shown in Figure 15.24.

We believe it may be interesting to note that the animation effects of the umbrella model, including the extension of surface canvas and the changing curvature of rod curves, are obtained from the second line of the script, by animating *only* the umbrella opening angle, and *without any change* to the umbrella defining code given in the above

Script 15.5.7 (ECS example (4))

```
DEF ECSO = (T:1:2 ~ CUBOID):<0.5,4,4>;

DEF UFP = (R1 & R2) + (R1 & ECSO) + (R2 & ECSO);
DEF FP = CS - project:<2,3>: UFP;

VRML:(FP):'out.wrl';
```

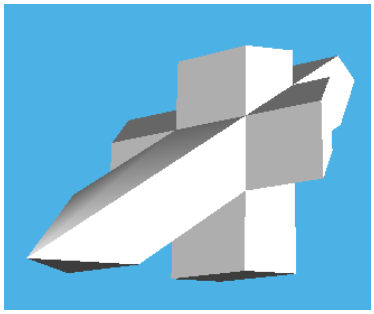

Figure 15.22 Extended configuration space representation of unfeasible configurations: $UFP = ECS(CSO)$

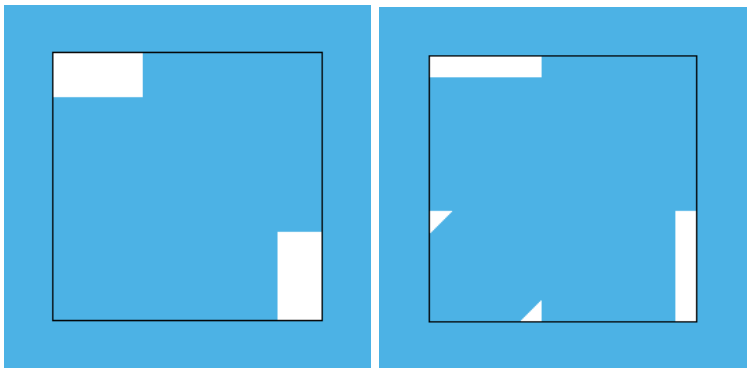

Figure 15.23 Free configuration space FP (the white areas) for the constrained example: (a) 1D obstacle located at $x = 2$ (b) obstacle at $x = 2.25$

mentioned scripts. Such effects are due to the pervasiveness of values of *animBehaviour* type trough the `PLaSM` code at evaluation time. We remember that the rods were defined as Bézier curves depending on cortrol points depending in turn from the opening angle, and analogously the canvas are Coons' patches defined by boundary Bézier curves that are functions of the umbrella opening angle.

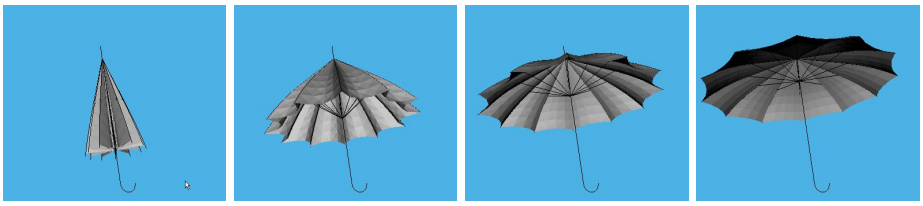

Figure 15.24 A sequence of frames from the animated umbrella opening

15.6.2 Non-holonomic planar motion

The example discussed in this section is aimed at illustrating a quite frequent case of planar motion, where the degrees of freedom are non-independent. In this type of

Script 15.6.1 (Singing in the rain)

```
DEF clip = MOVE:(umbrella ∼ [K:10, ID]):<10,50,70,80>:<0,1,2,3>;
DEF movie = (LOOP:10 ∼ ANIMATION): < clip, SHIFT:3, WARP:-1:clip >;

VRML:movie:'out.wrl';
```

motion, typical of cars, the orientation of the moving body depends on the derivative of the displacement vector, i.e. on the direction of the velocity vector.

In other words, the orientation of the body, i.e. the rotational degree $\alpha(u)$ along the body's trajectory, depends on the derivative of the translational degrees $(c_x(u), c_y(u))$, seen as the smooth coordinate functions of a point-valued map $\boldsymbol{c}(u)$ of a real parameter, i.e. as a smooth curve. In particular, we have:

$$\boldsymbol{c} : [0,1] \to \mathbb{R}^2, \quad \text{with} \quad \boldsymbol{c}(u) = (c_x(u), c_y(u)), \quad \text{and} \quad (\phi \circ \boldsymbol{c}')(u) = \alpha(u)$$

where

$$\phi : \mathbb{R}^2 \to [-\pi, \pi], \quad \text{with} \quad \phi(\boldsymbol{c}(u)) = \text{sign}\left(\frac{c'_y(u)}{\|\boldsymbol{c}(u)\|}\right) \text{acos}\left(\frac{c'_x(u)}{\|\boldsymbol{c}(u)\|}\right).$$

Implementation The `PLaSM` implementation of this kind of planar motion is quite easy, and is given in Script 15.6.2. The `Vect2DToAngle` operator is a direct implementation of the ϕ function above. When ϕ is applied to a vector in $\mathbb{R}^2$, it returns the angle that the vector gives with the x axis. Notice that the `UnitVect` operator is given in Script 3.2.4, and that the `ACOS` predefined function returns the angle whose cosine is equal to the actual function argument.

The `tangent` operator, where `D` is the Fréchet derivative given in Script 5.2.15, when applied to a sequence of coordinate functions of a curve map, returns the sequence of coordinate functions of the derivate map.

Finally, the `Curve2CSpath` operator, which stands for "*from curve to configuration space path*", when applied to a 2D `curve` generating sequence $\boldsymbol{c}$, returns a function that, applied to some u point in $[0, 1]$, gives back the $(\alpha(u), c_x(u), c_y(u))$ configuration space point.

Script 15.6.2 (From 2D curve to CS)

```
DEF Vect2DToAngle = SIGN ∼ S2 * ACOS∼S1 ∼ UnitVect;
DEF tangent (f::IsSeqOf:IsFun)(a::IsSeqOf:IsReal) = CONS:(D:f:a):<1>;
DEF Curve2CSpath (curve::IsSeqOf:IsFun) =
   AL ∼ [Vect2DToAngle ∼ tangent:curve, CONS:curve];
```

Note We note that the `Curve2CSpath` operator, when applied to some *arbitrary* 2D curve generating sequence $\boldsymbol{c}$, returns the vector-valued function $(\phi \circ \boldsymbol{c}', \boldsymbol{c}) : \mathbb{R} \to \mathbb{R}^3$, such that $(\phi \circ \boldsymbol{c}', \boldsymbol{c})(u) \in CS$ for each $u \in [0, 1]$. In conclusion, we can say that the discussed planar non-holonomic motion is completely modeled by the map

`Curve2CSpath` given below, from the set of *trajectory* curves in working space[8] to the set of CS curves:

$$\texttt{Curve2CSpath} \equiv (\phi \circ \mathrm{D}, \mathrm{id}) : ([0,1] \to \mathbb{R}^2) \to ([0,1] \to \mathbb{R}^3),$$

where D is the derivative operator.

Example 15.6.1 (Our red "Ferrari")
A (very) simplified *dream car* is modeled and animated in Scripts 15.6.3 and 15.6.4, respectively. In particular, `Ferrari` is the mobile object generating function depending on three real parameters, where `mover` is a properly transformed instance of the extruded `car` defined in Script 6.2.5. In this case the reader is asked to give his/her own implementation of the mobile object, since the `Curve2CSpath` operator suitably applies to every planar motion of a rigid body. The `background` object is a container for the image of the `trajectory`, given in Script 15.6.4, and a supporting white rectangle.

Script 15.6.3 (Ferrari example (1))

```
DEF Ferrari (alpha,tx,ty::IsReal) =
   (T:<1,2>:<tx,ty> ~ R:<1,2>:alpha):mover
WHERE
   mover = (T:2:0.15~R:<2,3>:(PI/2)~S:<1,2>:<-1/8,1/8>~T:1:-1.5):car
END;

DEF background = STRUCT:<
   (T:<1,2>:<-1,-1> ~ CUBOID):<7,5> COLOR WHITE,
   MAP:trajectory:(Intervals:1:40)
>;
```

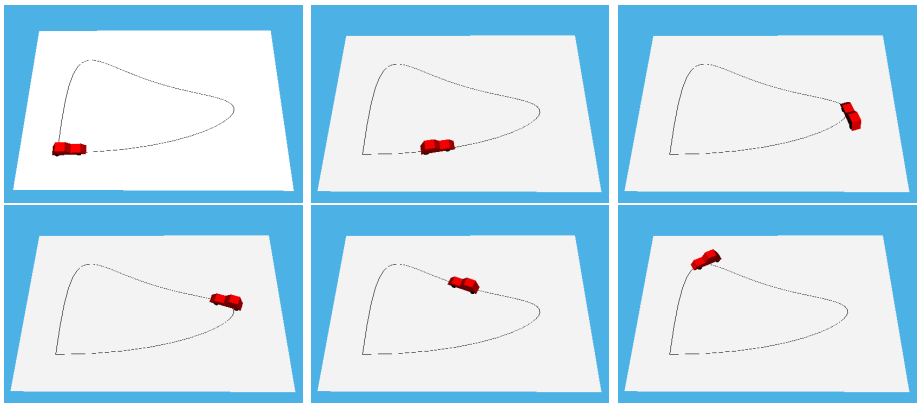

Figure 15.25 A frame sequence from the motion of a planar mover (our red "Ferrari") on a given trajectory

The motion of our red `Ferrari` is modeled in Script 15.6.4, where the *animated*

[8] Here we use the Robotics term for the space where the mover acts.

polyhedral complex produced by the evaluation of the `out` symbol is exported into the VRML file named `out.wrl`. A sequence of frames from the animation rendering produced by the *Cortona* Plug-in and *MS Internet Explorer* is shown in Figure 15.25.

The example `trajectory` is a Bezier curve of degree 5 generated by six 2D points. The `CSpath` is a piecewise linear approximation with 20 segments of the corresponding CS curve. The `sampling` operator is given and discussed in Script 15.2.1. Finally, the uniform timing of the animation is specified by the `Timepath` real sequence, for a total duration of 5 seconds. The `scalarVectProd` operator, for the product of a scalar times a vector, is given in Script 2.1.20.

Script 15.6.4 (Ferrari example (2))

```
DEF trajectory = Bezier:S1:<<0,0>,<8,0>,<5,5>,<2,-3>,<0,8>,<0,0>>;
DEF CSpath = (AA:(Curve2CSPath:trajectory ~ [ID]) ~ Sampling):20;
DEF Timepath = 5 scalarVectProd Sampling:20;

DEF out = STRUCT:< background,
   MOVE: Ferrari: CSpath: Timepath >;

VRML:out:'out.wrl';
```

Project hint It might be interesting to note, as a project hint to the reader, that by reparametrizing the trajectory for $s \in [0, L]$, where s is the *curvilinear abscissa* and

$$L = \int_{trajectory} ds$$

is the total length of the trajectory, the animation would linearly approximate the actual speed of the mobile object, for every arbitrary time `sampling` distribution into the increasing sequence `Timepath`.

15.6.3 Anthropomorphic robot with 29 *degrees of freedom*

The aim of this section is to show an easy strategy to reduce the number of parameters that an animation depends on, and to discuss the very high-level `PLaSM` operators used to develop a *hierarchical animation* (see Section 15.3.3). For this purpose we consider a 3D articulated moving system with several degrees of freedom, say the `Robot` object defined in Script 15.6.5, that we already introduced as the hierarchical `body` structure given in Script 8.5.9.

Implementation The `Robot` function given in Script 15.6.5 is an explicitly parametrized version of the `body` object of Script 8.5.9. In this case, each formal parameter is a sequence of real numbers, denoting the subset of degrees of freedom associated with each body joint. The reader is referred to Section 8.5.2 for a discussion of the semantics of the articulated system. Just note here that the `Torso` is used as root of the tree of `Robot` links, and that there are 5 kinematics chains. The functions `top_limb`, `upper_limb` and `lower_limb` can be found in Section 8.5.2.

Script 15.6.5 (Robot definition)

```
DEF Robot( dofNeck, dofHead,
   dofShoulderR, dofElbowR, dofWristR,
   dofShoulderL, dofElbowL, dofWristL,
   dofHipR, dofKneeR, dofAnkleR, dofToeR,
   dofHipL, dofKneeL, dofAnkleL, dofToeL::IsSeqOf:IsNum ) =
STRUCT:< torso,
   (joint1 ~ top_limb):< dofNeck, dofHead >,
   (joint2 ~ upper_limb):< dofShoulderR, dofElbowR, dofWristR >,
   (joint3 ~ upper_limb):< dofShoulderL, dofElbowL, dofWristL >,
   (joint4 ~ lower_limb):< dofHipR, dofKneeR, dofAnkleR, dofToeR >,
   (joint5 ~ lower_limb):< dofHipL, dofKneeL, dofAnkleL, dofToeL >>
WHERE
   torso = T:<1,2>:<-5,-5>:torso_shape,
   joint1 = T:3:30,
   joint2 = T:<1,3>:<6,30>,
   joint3 = T:<1,3>:<-6,30>,
   joint4 = T:1:4,
   joint5 = T:1:-4
END;
```

Interpolators In some animation system the variables, defining a mapping from the standard unit interval to the interval of joint limits of some degree of freedom, are called *interpolators*.

Actually, an interpolator can be seen as a higher-level degree of freedom, and a complex motion, say a curve living in a high dimensional space $\mathrm{CS} = \Xi^k$, can always be reduced to a curve living in a lower-dimensional configuration space $[0,1]^h$ defined by $h << k$ interpolators.

Two quite simple motions of our `Robot` with 29 degrees of freedom, specified by using a single interpolator, are defined in the following Example 15.6.2. Clearly, a finer control of the motion could be achieved by using an higher number of interpolators.[9]

Example 15.6.2 (Gymnastic exercises)
A function `Exercise1` of a single real parameter `cx` is defined in Script 15.6.6 by referencing the `Robot` function given in Script 15.6.5 with actual parameters depending on `cx`, supposed to be in the interval $[0,1]$. It is clear that `cx` is an *interpolator* variable, as defined above. Notice that only 4 joint variables of `Robot` are affected by the `cx` interpolator, whereas the other 25 are set to some fixed values, and in this case are all set to zero. It is quite easy, by looking at `Robot` definition and at Figure 15.26 to recognize the type of motion.

Another gymnastic exercise of our `Robot` is defined in Script 15.6.7, where a single interpolator `cx` now defines a (linear) curve in a 16-dimensional CS subspace.

A hierarchical animation of the `Robot` generating function is defined bottom-up in Script 15.6.8:

[9] And, for the sake of animator' happiness, by connecting them to interactive widgets in the animation system interface.

Script 15.6.6 (Exercise 1)

```
DEF Exercise1(cx::IsNum) = Robot:<
   <0,0>,<0,0,0>,<0,150*cx,0>,<0,0>,<0>,<0,-150*cx,0>,<0,0>,<0>,
   <0,45*cx,0>,<0>,<0>,<0>,<0,-45*cx,0>,<0>,<0>,<0> >;
```

Script 15.6.7 (Exercise 2)

```
DEF Exercise2(cx::IsNum) = Robot:<
   <0,5*Cx>,<0,5*Cx,20 - 40*Cx>,<-30 + 60 * Cx,0,0>,<30 - 30*Cx,0>,<0>,
   <30 - 60 * Cx,0,0>,<30 * Cx,0>,<0>,
   <20 - 40 * Cx,0,0>,<-20 * Cx>,<-20 + 40 * Cx>,<20 * Cx>,
   <-20 + 40 * Cx,0,0>,<-20 + 20 * Cx>,<20 - 40 * Cx>,<20 - 20 * Cx> >;
```

1. By setting, at the deepest hierarchical level, two clips of `AnimPolComplex` type, called `clip1` and `clip2`, by using the `MOVE` animation primitive.
2. Defining, at a higher level, two more complex clips, produced by joining, within two adjacent *time segments* $[0, 1]$ and $[1, 2]$, one instance of `clip1` and one back-reversed copy of it, and analogously for `clip2`. The *reflection* of timeline is produced by the `WARP` primitive with `-1` argument.
3. Each one of such double clips with timeline $[0, 2]$ is looped 5 times, yielding two longer clips with timeline $[0, 10]$.
4. At the highest hierarchical level, three clips with timeline $[0, 10]$, $[0, 1]$ and $[0, 10]$, respectively, are mounted together. The second one is generated by the `FRAME` primitive, locking the `Robot` for 1 seconds in the configuration produced by `Exercise1:0`. The pasting of the three clips is performed by the timeline translations produced by the `SHIFT` primitive.

Script 15.6.8 (Hierarchical animation)

```
DEF clip1 = MOVE: Exercise1 :<0,1>: <0,1>;
DEF clip2 = MOVE: Exercise2 :<1,0>: <0,1>;

DEF movie = LOOP:10:(
   ANIMATION:<
      LOOP:5:(
         ANIMATION:< clip1, SHIFT:1, WARP:-1:clip1 >),
      SHIFT:10,
      FRAME:( Exercise1:0 ):<0,1>,
      SHIFT:1,
      LOOP:5:(
         ANIMATION:< clip2, SHIFT:1, WARP:-1:clip2 >)
   >);

VRML:movie:'out.wrl';
```

Note It is important to remark that the semantics of the `ANIMATION` primitive closely resembles the one of the `STRUCT` primitive.

In particular, a *timeline transformation*, say a `SHIFT:`t or `WARP:`t tensor, can be either directly applied to a clip (*animPolComplex*), or inserted within the sequence argument of some `ANIMATION` primitive. In the latter case each timeline transformation applies to all clips that follow it in the sequence, exactly as done by affine transformations within a `STRUCT` sequence. The *animPolComplex* object called `movie` in Script 15.6.8 is a good example of this semantics.

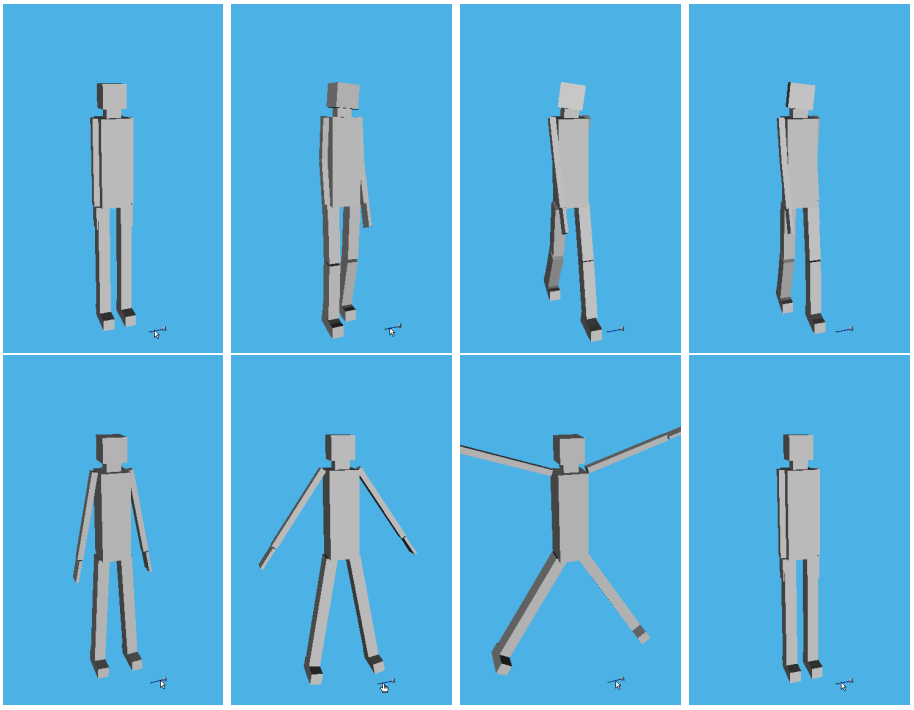

Figure 15.26 A frame sequence from robot's gymnastic exercises

15.6.4 Solving a 2D labyrinth

A 2D labyrinth, where a path connecting the entrance and the exit must be found, provides an interesting example for planning the motion of a mobile object moving amidst obstacles.

Labyrinth modeling

In Script 15.6.9 we give a general method to produce geometric models of rectangular labyrinths by assembly predefined rectangular cells. In particular four cells models `c1`, `c2`, `c3` and `c4` are defined, that are displayed in Figure 15.27. A `coding` matrix using integers from $\{1, 2, 3, 4\}$ is then given, that codifies the desired labyrinth layout. The operators `Q` and `optimize` are defined in Scripts 1.5.5 and 6.5.2, respectively.

The transformation from the `coding` integer scheme to the 2D polyhedral complex

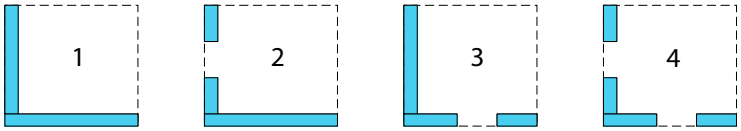

Figure 15.27 The four cells used for labyrinth assembly

Script 15.6.9 (Labyrinth (1))

```
DEF c1 = STRUCT:< Q:10 * Q:1, Q:1 * Q:<-1,9> >;
DEF c2 = STRUCT:< Q:10 * Q:1, Q:1 * Q:<-1,3,-3,3> >;
DEF c3 = STRUCT:< Q:<4,-3,3> * Q:1, Q:1 * Q:<-1,9> >;
DEF c4 = STRUCT:< Q:<4,-3,3> * Q:1, Q:1 * Q:<-1,3,-3,3> >;

DEF cellTypes = <c1,c2,c3,c4>;

DEF coding = <
   <1,1,4,2,4,1,2,1,2,2,3,1>,
   <3,2,2,3,3,4,4,4,3,4,3,2>,
   <2,1,2,2,2,3,3,1,2,1,2,1>,
   <1,2,1,2,2,2,3,3,2,4,2,2>,
   <3,4,2,4,3,4,1,1,4,3,3,2>,
   <1,1,4,3,3,3,3,4,3,1,2,2>,
   <2,2,2,3,3,3,2,3,3,1,4,2>,
   <1,1,2,1,2,1,2,1,2,1,2,1>>;
```

`Labyrinth` is given in Script 15.6.10. In this case, the `coding` matrix is first transformed into a `CellArray` matrix, whose elements are taken from the `cellTypes` set. Then, a polyhedral complex `Assembly` is generated from `CellArray`, by suitably inserting horizontal and vertical translation tensor generators `T:1` and `T:2` and `STRUCT` operators between `CellArray` elements. Finally, the `Labyrinth` model is completed by adding the topmost `border1` and right-hand `border2`.

Script 15.6.10 (Labyrinth (2))

```
DEF CellArray = (CONS ∼ AA:(CONS ∼ AA:SEL)): coding: cellTypes;

DEF Assembly = (assemblyRows ∼ AA:singleRow): CellArray
WHERE
   singleRow = STRUCT ∼ CAT ∼ AA:[ID, T:1 ∼ SIZE:1],
   assemblyRows = STRUCT ∼ CAT ∼ AA:[ID, T:2 ∼ - ∼ SIZE:2]
END;

DEF border1 = Q:120 * Q:<-80,1>;
DEF border2 = Q:<-120,1> * Q:<34,-3,44>;
DEF Labyrinth = (optimize ∼ STRUCT):< border1, T:2:70:Assembly, border2 >;

VRML:Labyrinth:'out.wrl'
```

The `PLaSM` code in Script 15.6.10 can easily be abstracted to give labyrinths of arbitrary dimensions starting from arbitrary sets of cells, with the only constraint that cells on the same row have the same height. We leave this task to the advanced

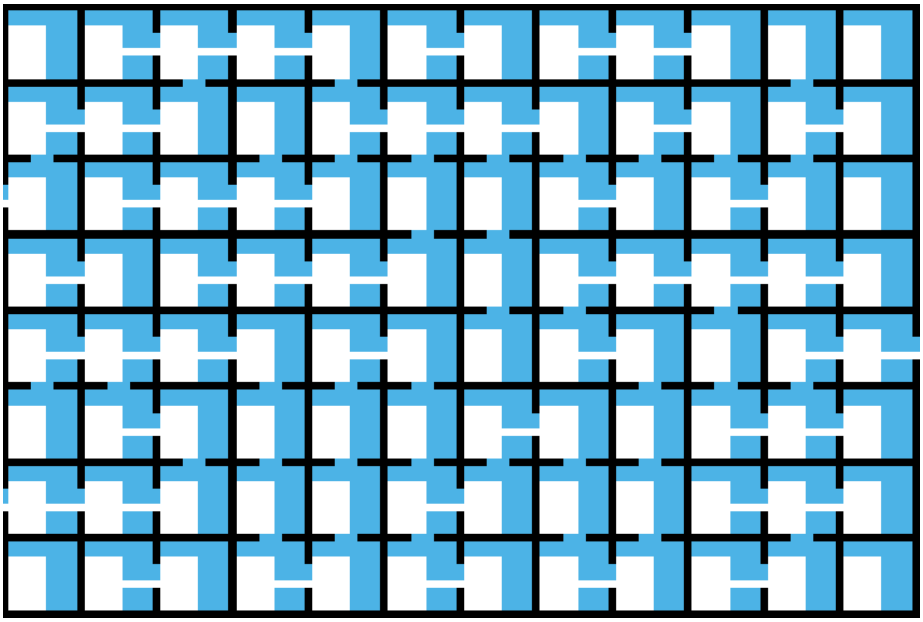

Figure 15.28 A 2D labyrinth (black) and the free configuration space (white) for a moving rectangle `CUBOID:<4,2>` with 2 translational DOFs *only*. The gray areas are the grown obstacles. The labyrinth's entrances are unconnected in FP, hence an admissible motion between them does not exist

reader of the book. We just note that the borders and the parameter of tensor `T:2:70` must be suitably computed at this purpose.

Translational motion

An early geometric technique, was developed by Lozano-Perez and Wesley [LPW79] for planning the collision-free motion of a 2D convex body translating amidst polygonal obstacles. Their approach reduces to planning the motion of a *single point* moving amidst a *grown* version of the obstacles, that is obtained as Minkowski sum of the obstacles and the moving body.

In Section 14.6 we defined the `OFFSET` operator as the Minkowski sum of a polyhedral complex with a rectangular parallelotope. Hence, if we suppose our mobile body to be defined as `CUBOID:<4,2>`, we can compute the *grown obstacles* and the free configuration space FP as done in Script 15.6.11.

Script 15.6.11 (Labyrinth (2))

```
DEF fp = (BOX:<1,2> - OFFSET:<-4,-2>):labyrinth ;

DEF out = STRUCT:< fp COLOR white, labyrinth COLOR black >;
VRML: out:'out.wrl';
```

The resulting `out` object is shown in Figure 15.28. As it is easy to see, FP is not

path-connected, so that no solution exists for a translational motion of our horizontal rectangle with size 4×2. The reason is quite obvious: the width of the mobile body is larger than the passages between vertically adjacent labyrinth cells, so that only horizontal translations are allowed.

General 2D motion

Let us suppose now that our mobile rectangle is also allowed to rotate about its local origin, with angle $\alpha \in [-\pi/2, +\pi/2]$, i.e. between -45 and $+45$ degrees. In this case it will be able to pass across both horizontal and vertical doors in the labyrinth.

In Script 15.6.12 we compute FP under this new assumption. First, we compute the extended representations `ECSmover` and `ECSobstacles` of the mobile body and the Labyrinth, respectively. Then we compute by intersection the set `UFP` of unfeasible positions. Finally, FP is given by subtracting from CS the projection of the unfeasible position set.

The `EX`, `LEX` and `SEX` operators, for straight, linear and screw extrusion, respectively, are given in Script 15.5.2. Every path in the path-connected FP component between the labyrinth entrances gives a feasible motion between the start and goal configurations of our mover object.

Script 15.6.12 (Labyrinth (3))

```
DEF ECSmover = (dof3 ∼ dof2 ∼ dof1): mover
WHERE
   dof1 = R:<1,2>:(pi/-2) ∼ SEX:<pi/-2,pi/2>:6,
   dof2 = LEX:<0,121>,
   dof3 = LEX:<0,81>,
   mover = CUBOID:<4,2>
END;

DEF ECSobstacles = (dof3 ∼ dof2 ∼ dof1):Labyrinth
WHERE
   dof1 = EX:<pi/-2,pi/2>,
   dof2 = EX:<0,121>,
   dof3 = EX:<0,81>
END;

DEF UFP = ECSmover & ECSobstacles;
DEF CS = (T:3:(PI/-2) ∼ CUBOID):< 121, 81, PI >;
DEF FP = CS - project:<3,4,5>: UFP;

VRML:(FP struct @1:CS):'out.wrl';
```

The reader should note that our result is only a polyhedral — and regularized — approximation of the true FP set. Also, as it is well known to computer scientists and computational geometers [HSS84, SS90], the FP computation is $PSPACE$ hard for an arbitrary number of moving rectangles. Thus, it remains tractable only for very simple settings, such as the one discussed in this section.