Clean Game Library

Mike Wiering

Katholieke Universiteit Nijmegen
Department of Informatics
Faculty of Physics, Mathematics and Computer Science

August 1999

Abstract

This paper describes a game library for Clean, specially designed for parallax scrolling platform games. The purpose is to make game programming easier, by letting the programmer specify what a game should do, rather than program how it works. By integrating this library with tools for designing bitmaps and levels, it is possible to create complete games in only a fraction of the time it would take to write such games from scratch.

At the moment, the library is only available for the Windows platform, but it should not be too difficult to port the low-level functions to other platforms. This may eventually provide an easy way to create games that run on several platforms.

Acknowledgements

First of all, I would like to thank my supervisors, Rinus Plasmeijer and Marko van Eekelen for their guidance and for allowing me to choose games as a serious subject in the first place.

I would like to thank Peter Achten for his guidance throughout the project, for helping me with all the implementation problems and for his suggestions to correct and improve this thesis.

Furthermore, I want to thank everyone who helped me improve the library by sending remarks, suggestions and bug reports.

Finally, I would like to thank my family and friends for all their support during this project.

Contents

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 TERMINOLOGY **2**

2.1 WHAT IS A PLATFORM GAME? 2
2.2 LEVELS 3
2.3 LAYERS 3
2.4 POSITIONING LAYERS 5
2.5 TILES 6
2.6 TILE SEQUENCES 6
2.7 DEFINING A LAYER 6
2.8 SPRITES AND OBJECTS 7
2.9 THE MAIN CHARACTER 8
2.10 BOUNDS 8
2.11 TEXT ELEMENTS 9

CHAPTER 3 USING TOOLS TO DESIGN GAMES **10**

3.1 CREATING TILES AND SPRITES 10
3.2 EDITING LEVELS 11
3.3 GAME CREATION TOOLS 11
3.3.1 Game Maker *11*
3.3.2 Klik & Play *12*
3.4 GAME LIBRARIES 12
3.4.1 DirectX *13*
3.4.2 Allegro *13*
3.4.3 FastGraph *13*

CHAPTER 4 CREATING A GAME LIBRARY FOR CLEAN **14**

4.1 DESIGN DECISIONS 14
4.1.1 Tools *14*
4.1.2 Library requirements *15*
4.2 IMPLEMENTATION DECISIONS 16
4.2.1 Platform choice and portability *16*
4.2.2 Library choice *16*
4.2.3 Programming language *16*
4.2.4 Performance *16*

CHAPTER 5 SPECIFYING A GAME **18**

5.1 DEFINING A GAME 18
5.2 DEFINING LEVELS 19
5.3 DEFINITION OF A BOUND MAP 20
5.4 DEFINING LAYERS 21
5.5 GAME BITMAPS 22
5.6 DEFINING OBJECTS 23
5.6.1 The complete object definition *23*
5.6.2 States *24*
5.6.3 Sprites *24*
5.6.4 The object record *24*
5.6.5 Object events *30*
5.7 STATISTICS 33

CHAPTER 6 GAME FUNCTIONS **34**

6.1 STARTING THE GAME ENGINE 34
6.2 CREATING OBJECTS 34
6.3 FOCUSING AN OBJECT 35
6.4 BROADCASTING EVENTS 35

6.5 PLAYING SOUNDS 36
6.6 CHANGING THE BOUND MAP 37

CHAPTER 7 IMPLEMENTING THE LIBRARY 38

7.1 STRUCTURE OF THE LIBRARY 38
7.1.1 Levels of information 38
7.1.2 Communication between levels of code 39
7.2 INTERNAL REPRESENTATION OF A GAME 41
7.2.1 GameHandle 41
7.2.2 LevelHandle 41
7.2.3 ObjectHandle 42
7.3 LOW-LEVEL FUNCTIONS 43
7.3.1 Game result codes 43
7.3.2 Screen functions 43
7.3.3 Bitmap functions 45
7.3.4 Sound functions 46

CHAPTER 8 USING THE TOOLS 48

8.1 USING GRED 48
8.1.1 Cursor movement 48
8.1.2 Drawing pixels 49
8.1.3 Editing the palette 49
8.1.4 Loading and saving sprites 49
8.1.5 Creating animation sequences 50
8.1.6 Blocks 50
8.1.7 Some more functions 50
8.1.8 Example 51
8.2 USING EDLEV 51
8.2.1 Cursor movement 52
8.2.2 Drawing tiles 52
8.2.3 Combining tiles 52
8.2.4 Shifting colors 52
8.2.5 Copying tiles 53
8.2.6 Bounds and map codes 53
8.2.7 Tile sequences 53
8.2.8 Other functions in EDLEV 54
8.3 GENERATING CLEAN CODE 54
8.3.1 MAKEDX 54
8.3.2 MAP2ICL 54

CHAPTER 9 SPECIFYING A PLATFORM GAME 57

9.1 GETTING STARTED 57
9.1.1 Directory structure 57
9.1.2 Clean compiler setup 58
9.2 CREATING LAYERS 58
9.2.1 Creating a background layer 58
9.2.2 Creating the foreground layer 59
9.3 CREATING SIMPLE OBJECTS 60
9.3.1 Creating a static coin 60
9.3.2 Creating a falling coin 63
9.3.3 The background clouds 63
9.3.4 Crates with items 65
9.3.5 Creating enemies 66
9.3.6 Creating the bees 67
9.3.7 Creating frogs 68
9.3.8 Other objects 68
9.4 CREATING OUR MAIN CHARACTER 69
9.4.1 Main character actions 69
9.4.2 Object state 69
9.4.3 Initialization 70

9.4.4 Movement control *70*
9.4.5 Collisions *72*
9.5 FINISHING THE GAME 73
9.5.1 Adding statistics *73*
9.5.2 Flow of the game *74*

CHAPTER 10 CONCLUSION **76**

10.1 RAPID GAME DEVELOPMENT 76
10.2 POSSIBILITIES 76
10.3 PERFORMANCE 76
10.4 OVERALL CONCLUSION 77

REFERENCES **79**

Chapter 1
Introduction

Although two-dimensional platform games are more and more being replaced by 3D games, this genre is still loved by very many people. Creating a good (playable) platform game is usually a very difficult task, which involves a variety of skills. Programmers will often have to spend a lot of time optimizing the code, because performance is crucial; these games have to run fast enough in order to be enjoyable. Next to the programming, graphics (drawings, bitmaps), sounds and music have to be created as well. In larger projects, several people work together (for example: a game programmer, an artist and a composer).

Many people want to start creating their own games, but soon give up because of the programming difficulties. For those who don't give up, it usually takes years before they have enough programming experience to really get started. They still spend most of the time solving programming problems, instead of creatively designing their game. A complete platform game, written from scratch in a language such as C or Pascal can easily have more than 10,000 lines of source code.

Is there no easier way to create a platform game? Of course there are many game libraries that can be used to make the programming a lot easier. These libraries usually take care of all the low-level functions, such as displaying bitmaps and playing sounds. Using such libraries will definitely save the programmer a lot of time, but the game itself still has to be programmed.

Another solution is provided by some game creation programs, discussed in Chapter 3. These tools make it possible to create complete games with no programming at all. Instead, the user can create sprites, objects and complete levels by just clicking with the mouse. Although this sounds wonderful at first, the possibilities turn out to be limited. Some types of games can be made, but these tools (current versions, at least) don't seem adequate for serious platform games.

Is there no way to design games easily and still have the flexibility of a real programming language? A possible solution is to let the programmer write a specification of the game in a functional language and have an interpreter for this specification that will run the game. The programmer doesn't have to specify exactly how the game works; only what it does. A standard game definition provides the framework for the specification and the programmer can use functions to describe the game.

The purpose of this research is to find out if it is possible to use such a specification in a functional language to create entire platform games and whether or not writing such a specification has benefits above using 'normal' programming languages.

The strategy is to implement a complete game library for the language *Concurrent Clean* [1]. First, we will create a complete game definition as an abstract data type and then implement an interpreter that can run a game of this type. Finally, we will create a few example games with this library and examine the results.

Another interesting question is whether or not the library can be used together with the existing Clean Object I/O library [2] and of course, how the produced games perform.

Chapter 2
Terminology

There are several types of computer games. The first games were *"action arcade games"*, they usually had a static background image on which a few objects[1] could move around. Well known examples include *Pac-Man* and *Asteroids*. Soon the backgrounds became larger and games could move (*scroll*) the entire screen around over these backgrounds. In a lot of these games we look at the (two-dimensional) crosscut of the situation from the front side, as if we are standing right before it (with the sky above and the ground below). We will call these games *"platform games"*. Newer games such as *Doom* and *Quake* are three-dimensional and show the situation from the player's point of view.

In this chapter we will take a look at the elements in a platform game and discuss some of the terms we use. Normally, a game is divided into *levels* (see §2.2), which are the different stages, areas in the game. These levels have to be completed one by one in order to complete the game. The visual part of a level is made of one or more *layers* (§2.3) that can scroll. These layers are built from small blocks, which we will call *tiles* (§2.5). A level can contain all kinds of moving creatures we will call *objects*. On the screen, these objects are represented by their *sprites* (see 2.8). A special kind of object is the *main character* (see §2.9), which is controlled by the player. A level also contains invisible information about 'walls' in the map that control how objects can move through the level, called bounds (§2.10). We will also take a look at the text elements in a game (see §2.11).

2.1 What is a platform game?

The name *"platform game"* comes from the concept of animated objects (*sprites*) walking and jumping on platforms. In these games the player controls a main character that can walk around on the screen. Most games scroll the screen while the main character moves, so that the area (*the level*) is much larger than one single screen. The view is always two-dimensional, there is no depth dimension like in 3D games. However, there are often background *layers* that scroll at different speeds to suggest depth (*parallax scrolling*). These games are also called *"junp 'n run games"* and *"side scrollers"*, which might be more descriptive terms for this genre [3]. But since most people speak of *"platform games"* we will use this term.

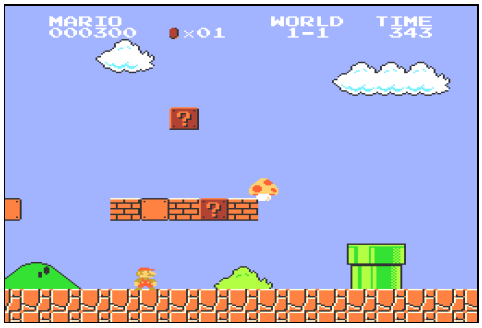

Figure 2-1: Super Mario Bros.

One of the most popular platform games ever is *Super Mario Bros.* (see Figure 2-1) by Nintendo (1985). In this game, the main character, the plumber Mario, must defeat all kinds of enemies and finally rescue the princess from the dragon Bowser. During his quest, Mario can gather coins and other items to survive. Most platform games have been based on a similar idea. A certain main character (a

[1] We will speak of *objects* as "real elements" in a game, not as objects in Object Oriented languages.

person or an animal) has to find his or her way through a number of levels and defeat all kinds of enemies in order to fulfill an important mission. Sometimes the player also has to solve little puzzles to complete a level, but most games do not require the player to think at all.

Even though platform games have a certain goal (i.e. to rescue the world), it usually isn't achieving that goal that makes the game fun to play. After beating a game, we stop playing it. The fun part is to explore the levels and keep wondering what the next one will be like. In the mean time we enjoy the graphics or the background music. The game should remain challenging all the way, and not become too difficult. Variety between levels is very important to make a game challenging.

2.2 Levels

The levels are the different parts (stages) of a game. Usually each level represents a different area with its own scenery and enemies. The main character starts somewhere and has to find the exit. In almost every game the player starts left and travels to the right. The level contains all kinds of dangers the player must pass.

In most games the player has a certain amount of lives. After every fatal mistake the player loses a life and must restart from the beginning of the level. When no lives are left, the game is over. The player often has limited time to complete a level.

Larger games usually have *worlds* in addition to levels. Each world consists of a number of levels. For example, a game could have 7 worlds of 5 levels each instead of 35 separate levels. Each world would have its own characteristics, such as the types of enemies. Typically, each world will start with easy levels in which the player learns new skills and end with a *boss-level* (a very large enemy that must be defeated to pass that world).

Information like the number of remaining lives, the time and the player's score are often shown at the top or bottom of the screen continuously during the game. In Figure 2-1 for example, we can see who is playing (Mario or Luigi), the player's score, the number of coins, the current world and level number, and the remaining time. We will call this kind of information *statistics* here.

2.3 Layers

A platform game is usually built with layers. Figure 2-2 shows a level of the game *Charlie the Duck* by Mike Wiering (1996), which contains four different layers. These separate layers are shown in Figure 2-3.

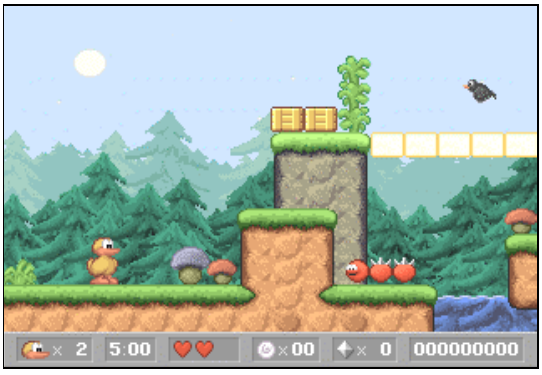

Figure 2-2: Charlie the Duck

The most distant layer is the blue sky together with the sun. Then comes a layer of distant trees and another layer of closer trees. These first three layers all belong to the level's background. The forth layer is on the top and represents the foreground. This is where we find all the moving objects. Actually, the game would also be playable without the three background layers; their only purpose is to make the level prettier.

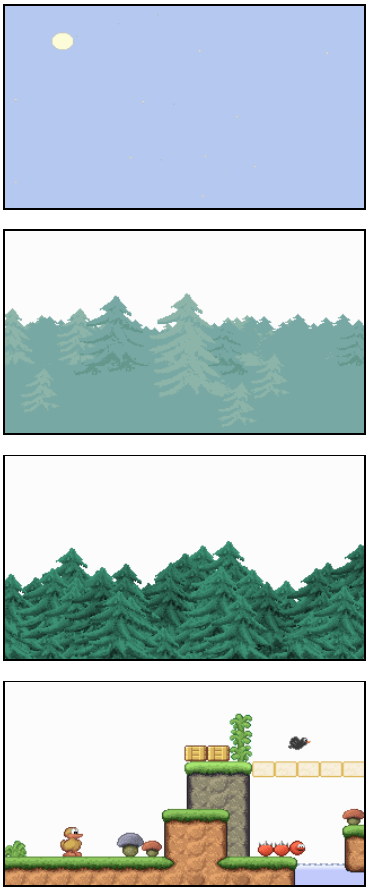

Figure 2-3: The separate layers (back to front)

All these layers (except the first) are partly transparent. By scrolling these layers at different speeds we can add a little depth to the game. It's like looking out of the window from a train at the horizon. The closer each object is, the faster it passes. Objects that are very far away, e.g. the sun do not seem to move at all. This is how we scroll the layers of a platform game. In this example, the most distant layer (the sky with the sun) doesn't scroll at all. The next layer of distant trees scrolls very slowly and the closer layers scroll faster. The foreground layers scroll in such a way that the main character will always be around the center of the screen, so the main character actually controls the scrolling.

If we look more closely at the foreground layer, we see that is built from small blocks (here there are 16 of these blocks horizontally and 13 blocks vertically), named *tiles* (also called *chips* [17] or just *blocks*). Most of the tiles are used at several places, sometimes mirrored or drawn in a different color. That means that we only have to draw a limited set of tiles to create a layer. Each layer has its own set of tiles, so the tiles of the different layers do not need to have the same size. For background layers, it's easier to use larger tiles, sometimes even only one large tile that fills the entire screen.

Figure 2-4 shows how the three background layers are projected on the screen (or the *game window*). Through the transparent areas of the front layers we see parts of those behind them. Usually, these layers are drawn on the screen from back to front, leaving the transparent areas unchanged.

These layers do not have the same size. The most distant layer (the sky with the sun) has the size of the screen. No matter how we move through the level, the sky will never change. The closer layers are larger, because they can scroll. As the player walks through the level to the right, the projection offset of these background layers will also (slowly) move to the right, making these layers seem to move to the left. The foreground layer is not shown here, but it is much wider than all these background layers. All the layers have the same height here, because this game does not scroll up and down.

When we speak of the size of a *level*, we mean the size of the area in which the objects move (defined by the size of the *bound map*, see §2.10). This is usually the same as the size of the foreground layer.

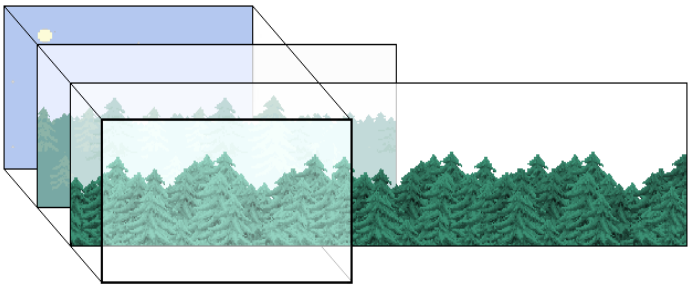

Figure 2-4: Projecting layers on the screen

2.4 Positioning layers

First of all we must define the orientation of a game. We will consider the left-top corner of an entire level to be position (x = 0, y = 0). The x-coordinate increments as we move to the right, the y coordinate increments as we move down. Because levels usually are larger than one screen, we will define the left-top corner of the screen as our *viewpoint* (xview, yview).

We will use this same orientation, with the upper-left corner as (x = 0, y = 0), for all our bitmaps (tiles, sprites) and for the internal representation of layers (layer map) and the bounds in a level (bound map).

We can define how far the player may move from the center of the screen before the viewpoint changes. This viewpoint is actually the player's position in the *bound map*, as described later (see §2.10). We can make a function for each layer that calculates the position of that layer according to this viewpoint. In most cases such a function would be something like:

(x, y) = (xview / C, yview / C)

In this function, C is a constant value associated with the distance of the layer. A layer that is very far away would have a large value for C; the foreground layer(s) would have the value 1.

In some cases we might also want to use time (represented as an increasing integer value T) in such a function. For example, to make a layer with clouds that moves slowly to the left all the time, we could use the following function:

(x, y) = (T / C, 0)

Now C controls the speed in which the layer scrolls to the right. As T increments, the x-position of the layer also increases so that the layer is drawn starting at higher x-position each time. This will make the layer seem to move to the left.

Although layers have a certain (limited) size, they are used as repeating patterns in a level, so the size of a *layer* does not influence the size of the *level*. In fact, we can make a complete background with just a very small layer (which can even be much smaller than the screen). For example, Figure 2-5 shows a small layer (left) and its pattern (right). A layer's projection offset is always calculated modulo its size (both horizontally and vertically).

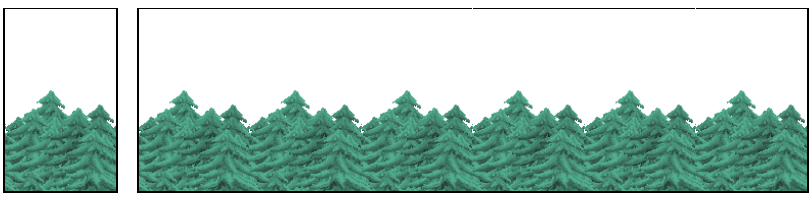

Figure 2-5: A small layer and its pattern

However, we will usually define larger layers, because such repeating layers soon become boring.

2.5 Tiles

A layer can be seen as a two-dimensional array of tiles. These tiles are small bitmaps of the same size. These small bitmaps are usually created separately, but must fit together in such a way that the layer doesn't look too blocky. Another technique is to create larger bitmaps and then split these into smaller tiles.

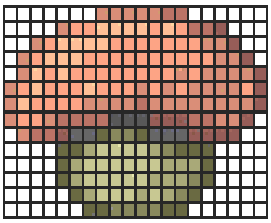

Figure 2-6: Example of a tile

Figure 2-6 shows an example of a tile. This is the same mushroom we saw in the front layer of Figure 2-2. The size of this tile is 20 by 14 pixels. The blank (white) pixels here are transparent. Because there is actually no such thing as a transparent color, we fill the transparent areas with a certain color (here white) and then interpret this color as the transparent color. Usually, one transparent color is defined for a complete layer and not for each tile. This is to make drawing layers less complicated.

When working with 256-color (8 bit) modes, all tiles in all the layers must have the same palette, otherwise the layers cannot be displayed together. In high-color (16 bit) or true-color (24 or 32 bit) modes no palettes are required and every tile may contain any color, defined by red, green and blue values (RGB). Because a transparent color is needed, there will always be one RGB color that cannot be used in the entire layer. It is up to the programmer to choose a transparent color that is not needed anywhere in the layer. This is no problem in 8-bit modes, because we use palette indexes instead of RGB values. For example, if we choose palette index #0, defined as RGB = {0,0,0} (completely black), as our transparent color, we can still use black in our layer by defining another palette index as RGB = {0,0,0} as well.

2.6 Tile sequences

To make our layers more interesting, we might want to add some movement to our layers. For example, water usually has little waves that move and we can let grass move back and forth a little to suggest wind. Of course we could make such things out of real sprites (see §2.8), but these simple movements really belong to the layer and do not affect the rest of the game. Because such movements are very common in platform games, we will define special changing tiles, *tile sequences*.

Figure 2-7 shows two tiles with which we can create a sequence. These two tiles look very similar at first, but if we display both tiles in turn for half a second or so, the cloud will seem to move a little. A sequence is defined by a list of tiles and the time to display each tile. After a sequence ends, it starts over from the beginning. So every time a layer is displayed, some tiles may change.

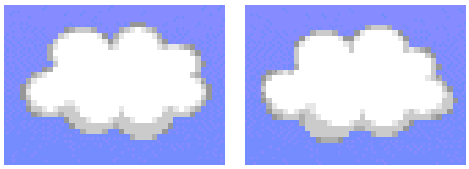

Figure 2-7: A tile sequence

2.7 Defining a layer

The introduction of tile sequences has made layers a little more complicated. Instead of the two-dimensional array of tiles we started with, we will now have to define a new way to store layers.

Because all the tiles in a layer have the same size, why not combine them to one large bitmap? Handling one bitmap is a lot easier than having separate bitmaps for every tile. This way, we get one large bitmap that contains every single tile that occurs anywhere in the layer. We will call this collection of tiles a *game bitmap*. This technique makes it possible to work with all graphics of a layer together. For example, if we want to adjust the colors in one of our layers, e.g. add more contrast or make the entire layer brighter, we can now easily use a drawing program and adjust this large game bitmap.

An example of a game bitmap is shown in Figure 2-8. This game bitmap contains 192 different tiles, all used in one large level. The game bitmap has one transparent color, black in this case.

Before we can define a layer, we will first have to be able to identify the separate tiles stored in our game bitmap. To do this, we will simply number these tiles from left to right, starting at 1 (in this example: 1..16, 17..32, …, 177..192).

Figure 2-8: A game bitmap

Tiles that are used in tile sequences are also stored in such a game bitmap. This means that our definition for tile sequences is now a list of *tile numbers* together with the time each tile must be displayed. Because each layer has only a limited number of tile sequences, we will also number these.

Now it is easy to define our layers (still using a two-dimensional array of integers):

A transparent layer may contain empty spaces, places where there is no tile. We will indicate these empty spaces with the value 0. The numbers 1..n represent the tile with the same number (where n is the number of tiles in the game bitmap). Negative values -1..-m represent a tile sequence number (where m is the number of tile sequneces). These tile sequences are defined by a list of tuples containing a tile number (1..n) and the number of frames to wait before the next frame (0 or more).

Later (in Chapter 5), we will expand this definition of a layer so we can also define horizontal and vertical mirroring.

2.8 Sprites and objects

Another important element of a platform game are the sprites. Sprites are animations that are not part of a layer, and can be drawn anywhere in a level. They can have any size or shape. Like tile sequences, sprites also have sequences of bitmaps which we will call *animation sequences*. A difference with tile sequences is that animation sequences do not always loop. An animation sequence could represent an explosion. This would be displayed only once. However, most animation sequences do loop. To make a character walk for example, we would define one sequence of a single step and repeat this sequence.

An example of a (looping) animation sequence is shown in Figure 2-9. By displaying all these animations in turn, the coin will seem to spin. The coin itself can also be moving, in that case each next sprite will be drawn at a different position.

Figure 2-9: An animation sequence

A sprite is the graphical representation of an *object*. We will use the term *object* for anything we define inside a level that is not part of a layer or part of the bound map. All creatures that move around in a level are objects, so is the main character. Objects don't have to have sprites; they can be invisible. All objects have certain properties (such as their position and their size) and events (such as a collision with another object). Objects can also have changeable states in which their properties and their behavior differ.

2.9 The main character

Most games have one main character, which is the object that the player moves and that is always near the center of the screen. However, there are also games that have two main characters that can be played by two separate players at the same time. For now we will only consider one main character.

A main character can usually perform several actions, like walk, run, jump, shoot and swim. Most games add a lot of special moves to make the game more interesting.

In most cases, the main character is the only object that responds to player input, but again there are exceptions. For example, the player might want to shoot an exploding missile and control the time when the object explodes.

2.10 Bounds

Now we have layers, an interesting problem remains; how do we define where the objects can stand? There are several different methods to do this. In early platform games (like Super Mario Bros., see Figure 2-1), the whole level was clearly divided into blocks, that formed only one layer. These blocks are either solid (like the bricks and pipes) or clear (the sky, the clouds and the mountains).

Another approach is to create a separate map of the *bounds* in a level. This map matches the foreground layer(s). Figure 2-10 shows a picture of a level and its bounds. In the right picture, black lines represent the bounds. The horizontal lines here are platforms on which an object can stand. The vertical line indicates a wall the player cannot pass. This *bound map* is usually defined together with the foreground layer(s) and will therefore have the same block size as the tiles used in these layers. But that is not necessarily so. We could just as well make a game without any layers at all, only a bound map and let sprites walk on invisible platforms in front of a black screen. All level information needed by objects is provided in the bound map.

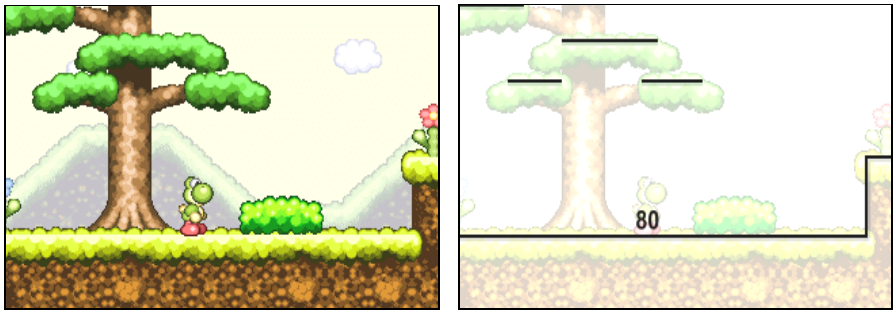

Figure 2-10: Bounds in a level

Just like a layer, the bound map can also be seen as a two-dimensional array of level bounds. Per unit there are four bounds: the upper, lower, left and right bound.

These bounds might change during the game. For example, when the player activates a switch, somewhere else in the level a door opens and the bounds change so that the player can pass through.

In addition to these bounds, the bound map can also contain *map codes*. These are integer numbers that can be used to initialize objects or to mark special areas in the map. This way we can position objects much easier than by entering coordinates in our source code. In the example, we see the number 80, which we can use to initialize the main character. As soon as the main character has been initialized, the map code is removed.

A more advanced way to specify bounds would be to only store a list of vectors. That would make it much easier to create diagonal bounds as well. But here we will just use a bound map with map codes, because objects should be able to make changes to the bounds. Changing a value in a two-dimensional array is usually much easier than constructing a new list of vectors.

The bound map always scrolls with the main character. The position of the bound map is actually the viewpoint of the level, from which the position of each layer is calculated (see §2.4).

2.11 Text elements

As discussed earlier, most games always have some information on the screen (statistics). But there is more text in a game. There can be a menu at the beginning of the game or perhaps a story to read. Usually the game starts with a title screen. Figure 2-11 shows the title screen and the main menu of the game Mickey's Magical Quest, by Capcom (1992).

Although there are games that run in a small window and use standard message boxes to display text or OS-menus to control the game, most games run full-screen and have their own menu system. These menus should be simple. A player doesn't want the game to look like a boring application program.

Figure 2-11: Mickey's Magical Quest

As shown in Figure 2-11, the text on the title screen isn't just written in a certain font and size, but is drawn as a picture, with several colors and shadow. For some games, complete new fonts are developed. Because platform games have little text, all text is sometimes stored as bitmaps. During the game, only the numbers (statistics) are printed as text.

Chapter 3
Using tools to design games

The times when programmers coded their blocky graphics directly into the game's code using hexadecimal values have passed. Everyone uses tools to create their graphics these days. Actually, any bitmap editor or drawing program can be used, but specialized sprite editors have features that make the job a lot easier.

Designing the levels of a game can also be a lot of work. Therefore, programmers will often make simple level editors for games they are writing. Instead of programming a level, they can design it.

In this chapter we will first discuss some of the tools used to create games. We will also take a look at a few game creation tools in which complete games can be designed, without programming. Finally we will look at some game libraries that can be used together with programming languages.

3.1 Creating tiles and sprites

Using a sprite editor to draw tiles and sprites has several advantages compared with ordinary drawing programs, here are some:

- *viewing tile patterns*
  Because tiles are often part of a larger pattern, we will often want to see how this pattern looks while drawing the tile. Figure 3-1 shows an example of a single tile that is used as a pattern. Without the pattern it's hard to see whether the single tile is visually correct. Most sprite editors provide this functionality.

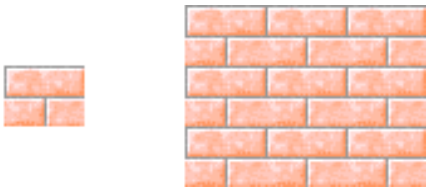

Figure 3-1: A tile and its pattern

- *support for animation sequences*
  Using a normal drawing program, it's very difficult to create a series of bitmaps that result in a smooth animation sequence. Each bitmap (*frame*) has to be almost the same as the previous, but slightly different to suggest movement. A good sprite editor will provide the opportunity to easily go through the sequence while editing it. Some sprite editors can even animate the sequence while the artist is drawing, so that real movement is actually recorded in the animation sequence. For example, by drawing a straight line from left to right will result in an animation sequence of a dot moving from left to right in the same speed as we drew it.

- *transparent color support*
  Because most sprites are not exactly rectangular, we use the transparent color to mark areas in the bitmap that don't belong to the sprite. If we choose pink as our transparent color in a normal drawing program and then smoothen our sprite, we will see that the edges have become pink. And if we make the whole sprite lighter, the transparent color is changed as well. A sprite editor that supports transparent colors will recognize pink as the transparent color here, and not as the color pink.

- *changing and editing palettes*
  When we make games for 8-bit (256-color) modes, we have to choose a palette of 256 colors for

an entire level. By changing the RGB value of a palette entry during the game, this color will change on the entire screen. So we have to make one palette that has all the colors we will need for anything we draw. We will often want to reserve a few palette entries for special effects like fading and for the color of text. Some sprite editors allow the user to select parts of the palette that may be used in a certain sprite. There are also often functions to easily edit the palette (for example create a row of intermediate colors), to reduce the number of colors used in a sprite and match a sprite with a new palette.

3.2 Editing levels

Level editors are much less common than sprite editors and are usually made for a specific game. Using such a level editor we should be able to create and edit the layers and the bound map of a level. While editing, we should also be able to create and edit our tiles. The level editor should then create game bitmaps for the layers with only those tiles that have been used (see §2.7). The layers themselves and the bound map must be stored in a readable format so the programmer can use them.

3.3 Game creation tools

Now we've seen that we can use tools to create sprites and even to create our levels, could we go even further and have a tool that creates an entire game for us, without any programming?

As we have seen in Chapter 2, most platform games have quite a lot in common. Since we can already create our bitmaps and levels, we would only need a tool that finishes the game from here.

Still a lot of work has to be done before we have a game, like defining the behavior of all the objects, including that of the main character; defining how objects interact with each other (what happens when two objects collide?) and defining the flow of the entire game. Finally we will also want to add sound, menus, text screens, etc.

There have been several attempts to make the ultimate game creation tool [5]. We will now look at two of these tools, *Game Maker* [6] and *Klik & Play* [7]. A few more of these tools are *Click & Create* and *The Games Factory* [8], these look a lot like *Klik & Play* and will not be discussed here.

3.3.1 Game Maker

One of the firsts was *Game Maker*, by Recreational Software Designs, with which scrolling 256-color games could be written for DOS. This program lets the user create and edit palettes, images, sounds, blocks (tiles, sprites), maps, monsters and characters. It has an *integrator* to combine these elements to a complete game. In general, Game Maker is easy to use. Figure 3-2 shows the block editor. The only size blocks can have is 20 by 20 pixels.

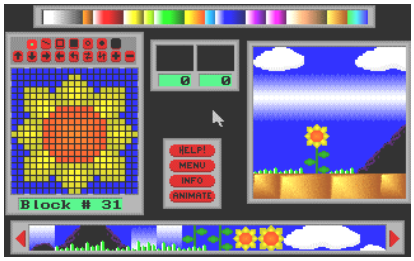

Figure 3-2: Game Maker

We can define a (main) character by making animation sequences for predefined sequences (Idle, Die, Injured, Pick up object and Drop object) and for a number of user defined sequences. These sequences include the movement of the character. Then we enter a list of numbers: the number of lives, inventory at start, money and hit points at start. While editing monsters, we can define what happens when our main character touches these monsters: change the score, number of lives or hit points.

Although it is possible to create some (standard) platform games using Game Maker, the possibilities are rather limited. The way movement is defined (within the animation sequences) does not improve playability. For example, if we make a main character with a jump sequence and a walk sequence, our character cannot both walk and jump at the same time. Instead, it will stop walking, jump in the air and land in exactly the same place where it jumped. The solution here is to make another sequence for our main character in which it jumps and also moves forward. Some games therefore have three jumping keys, one to jump to the left, one to jump straight up and down and the last one to jump to the right.

3.3.2 Klik & Play

Probably the most popular game creation tool is *Klik & Play* for Windows by Europress Software. This program is very easy to use and comes with a lot of predefined object that can be used. The user starts with an empty window in which objects can be placed. These objects automatically start moving according to their default behavior. The user can test the game right away, and whenever an event occurs (for example two objects collide), the game pauses and the user can define what should happen in this case. Of course the user can also create new objects.

For each object the movement can be selected from the following predefined types of movement: mouse controlled; eight directions; racing car; platform; bouncing ball; path movement or static. All these types of movement have their own parameters. Platform movement, for example has the following settings: Speed, Acceleration, Deceleration, Gravity and Jump Strength (see Figure 3-3).

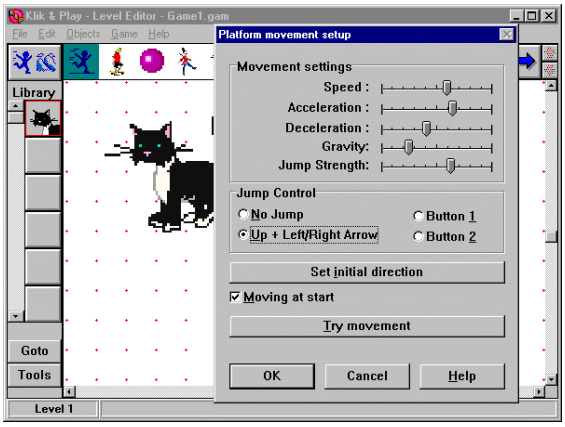

Figure 3-3: Klik & Play

These predefined types of movement make the program very easy to use, but also cause great limitations. We cannot really define how our objects move, we can only configure some parameters.

Games written with Klik & Play also cannot scroll, but consist of one window in which the whole game is played. Therefore, this program is not quite suitable for serious platform game development.

3.4 Game libraries

As we have seen, game creation tools do not provide us with a better way to create platform games than “real” programming languages. It is likely however that such tools will become increasingly better in the future, perhaps by combining them with some kind of (simple) programming language.

Probably the best solution is to use a real programming language in combination with a good *game library*. Such a library should enable us to easily create simple platform games without too much programming, but also give us the freedom to add any amount of code to make the game do *exactly* what we want it to.

An important decision is the level of this library. A very low-level library, that provides only the very basic functions such as drawing a pixel on the screen, will give the programmer a lot of freedom, but will not be very useful. A very advanced library that controls the whole game will often limit the programmer’s freedom.

Most game libraries provide graphics, sound and control functions. These functions are usually highly optimized to get the best results from the hardware. Parts of the library will often be rewritten for specific hardware devices to improve speed even more. The most important graphics functions found in just about every game library include setting the display mode, drawing (and reading) pixels, drawing (transparent) bitmaps, scrolling the screen and page flipping[2]. Higher level game libraries may also include functions to draw entire layers and control sprite movement and detect collisions. Sound functions will typically include playing background music and sound samples. Control functions provide ways to read user input (keyboard, joystick).

We will now look at some of the game libraries available for the PC.

3.4.1 DirectX

DirectX by Microsoft [12, 13, 14, 15] is the most used game library for Windows 95/98. Actually, there are no serious alternatives for making games for Windows. DirectX is integrated with the Operating System and can make use of the specialized video and sound hardware drivers, already installed for the OS.

DirectX contains all the low-level functions for graphics. Functions that are only supported on some hardware are usually emulated on other hardware to provide a universal set of functions. Layers and sprites are not directly supported by DirectX, but are easy to implement.

There are other game libraries that use DirectX and provide higher level functions than DirectX itself. An example is DelphiX by Hiroyuki Hori, which is mainly a DirectX interface for Delphi, but also includes a lot of extra functionality like drawing layers and handling sprites.

3.4.2 Allegro

Allegro [10] is a game-programming library that comes with *djgpp*, a 32-bit protected mode development environment for PC's, distributed under the GNU license [9]. Allegro was set up to be a multi-platform game library; it can be used in DOS, Windows, Linux and OS/2. It is currently one of the most used libraries for DOS games.

In addition to the usual graphics, sound and control functions, Allegro also supports file compression, timers, fixed point and 3D-math functions and even has its own graphical user interface (GUI) manager for applications.

3.4.3 FastGraph

FastGraph [11], by Ted Gruber Software Inc. is also an interesting game library because it is one of the few that also includes a *game editor*, which is actually a combination of a tile/sprite editor and a level editor. The library contains functions to use (display, scroll) such a level in a game. Obviously this combination of an editor and a library is much easier to use than a programming language alone. Since the actual game is still written in C, the programmer has a lot of flexibility to make any kind of game.

[2] Page flipping is a technique used in most games to avoid flickering. First, all graphics are drawn on an off-screen buffer (the *virtual page*), then by swapping (*flipping*) this virtual page with the *visual page* (the screen memory) the new graphics all come together as a new frame. The flipping is synchronized with the screen refresh.

Chapter 4
Creating a game library for Clean

This chapter describes the important choices that have to be made before we can create a game library for Clean. These decisions are divided into design decisions and implementation decisions.

4.1 Design decisions

Our main goal of writing a game library for Clean is to be able to design platform games *easily*. For this reason we want more than a set of low-level graphics functions. We should be able to write a complete game, just by giving a simple *specification* of the game, and without having to write all the standard functions.

The library will primarily be designed to create multi-layer, parallax scrolling platform games, but should also be useful for any other type of game (except 3D games).

There are a few homemade tools, GRED and EDLEV, that have turned out to be very useful for creating platform games over the years. We will design the library to be used together with such tools. Integrating these tools with the library will make designing tiles, sprites and complete layers easier.

4.1.1 Tools

As we have seen in Chapter 3, the best way to create a game is by using a combination of tools and programming. Tools that are very useful are tile/sprite editors and level editors. Later we might need some conversion utilities in order to import the levels into our Clean program.

4.1.1.1 Sprite editor

During this project we will the sprite editor GRED (see Figure 4-1). GRED works with a 256 color palette, of which 0 is always the transparent color. By changing the RGB value of palette entry 0, we can view the sprite against any color background. GRED also supports animation sequences, we can easily go back and forth between frames of a sequence while editing.

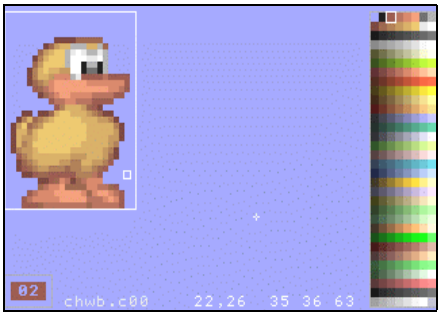

Figure 4-1: Sprite editor GRED

4.1.1.2 Level editor

As our level editor, we will use EDLEV (see Figure 4-2). Current versions of EDLEV can only edit one layer of a level at a time, but it is possible to put tiles in front of other tiles in the same layer. EDLEV will convert these overlapping tiles to new combined tiles in the game bitmap.

Figure 4-2: EDLEV

White lines display the bounds here. As we can see, an object could stand on top of the right palm tree and not on the left one. At the bottom of the screen we see some of our tiles. For creating and editing tiles, EDLEV automatically starts GRED.

EDLEV also supports map codes (see §2.10), although these are limited to a single byte per block. These map codes are displayed as hexadecimal values 01…FF. In the figure we see the map codes 20, B1 and B4, which represent objects that will be started at those positions during the game.

4.1.2 Library requirements

To be useful for serious (platform) game programming, the library must contain at least the following functionality:

- Setting the display mode to a given resolution or opening an area on the screen in which the game can take place. Full screen modes are generally preferred for games above normal game windows, but both should be possible. Of course the original screen must be restored when the program exits.
- Loading, displaying and scrolling complete layers of tiles. These layers can be transparent and can contain tile sequences.
- Handling a collection of objects. These objects are either initialized by a code on the bound map or by the user. The behavior of these objects is entirely defined and programmed by the user. The library must display the objects, run their animation sequences and generate certain events (such as collision events and timer events).
- Play background music and sound samples.
- Generate events for user input, with which the user can control objects.
- Automatic scrolling with the main character. The library should be able to scroll the game along with a selected main character as this character moves. This includes scrolling all the layers according to their movement function (see §2.4).
- Display text and number (statistics) during a game in a specified style.

The programmer should be able to use all this functionality from a high level. Instead of programming how a game works, using library functions, the programmer will give a specification of what the game looks like. The library will interpret this specification to run the game. The specification is based on a standard definition of a platform game. In this specification, the programmer may use functions, which will be evaluated at runtime.

To integrate the library with the design tools GRED and EDLEV, there will be a conversion program that produces Clean code, which can be imported into the game source code.

4.2 Implementation decisions

Now that we have defined what our game library should do, we can decide how to implement it.

4.2.1 Platform choice and portability

Because Clean is a multi-platform language, our game library should actually be made available for any platform. But since such a game library is for a great deal based on I/O calls, the implementation differs a lot for each platform. Implementation for multiple platforms would probably take too much time for this project. Therefore we will only implement the library for one platform, but keep in mind that the library should eventually run on any platform.

This first implementation of the library will be done for the Windows platform.

To improve portability, the design of the low-level library functions should be very general. For example, instead of making a function that can set the display mode to a given VGA mode number (as often seen in PC-game libraries) we would prefer a function that sets the screen mode to a given horizontal and vertical resolution (or opens a window of that size). Such a function can be implemented on any platform.

All these low-level functions have to be well documented, so that anyone can create an implementation of the library for a new platform, without translating the functions line by line. This way, the various implementations can be very well optimized. For example, if we want to port our library to the Mac, the best way would be to have someone with a lot of experience with game programming on the Mac write these functions from the specification, instead of translating the Windows version.

4.2.2 Library choice

Now we have decided to implement the new library for the Windows platform, we have to decide how to implement the low-level functions. Writing all these functions from scratch would be an enormous task so it's best to use another library from within our new library.

As stated in §3.4.1, the most obvious choice for serious game development in Windows 95/98 is DirectX. A disadvantage of DirectX is that it is not available for other platforms than Windows. Using a library such as Allegro would make porting easier. However, current versions of Allegro can only be used to create stand-alone DOS executables and not be called from Windows programs. So we will use DirectX.

4.2.3 Programming language

Because we cannot call DirectX functions straight from Clean, there will have to be some kind of interface. At the moment, the only way to make external calls from a Clean program is by using C calls. Therefore, we must write C functions for the calls to DirectX. Our library functions will generally do a lot more than only call an equivalent DirectX function. This can be done in the C code as well. The library itself will be a combination of Clean modules and C code. Though the library is actually run from the C side, the control of the program will keep going back and forth to the Clean side to handle all sorts of events.

We will call the “heart” of the library, which is part of the C code, the *game engine*. The game engine keeps the game running by updating object movements and generating events. It also handles all the display of the game.

4.2.4 Performance

Good performance of a game is usually crucial for its popularity. For platform games, the frame rate is important. Most platform games run at 70 or 100 frames per second. If drawing the screen takes too long, we get only half (or a third, or less) of this number of frames per second. It can be very annoying for a player if the game suddenly starts moving at half speed every time a new sprite appears. For smooth scrolling games, even loosing only a few frames every second can be annoying.

What would be the performance needed to make this new library useful? For a simple platform game we can set our minimal performance goal at 100 frames per second in mode 320x240, 256 colors (8-

bit) with two layers, background music playing, a main character and a number of moving objects (say five). When the library is finished, we can test such a game and find out what the minimum PC configuration is on which this library can be used.

Chapter 5
Specifying a game

In Chapter 2, we have discussed the elements in a platform game. In this chapter we will set up a complete Clean specification of a game, containing these elements. Later we will write a function that, given such a specification, will interpret the game.

The complete game definition is specified in StdGameDef.dcl.

5.1 Defining a game

We will start at the highest level with the specification of a complete game. So what is a game? Probably the best way to describe a game is by a collection (or a list) of *levels*, because games are clearly divided into separate levels.

In addition to (normal) levels, games also have small parts like the title screen, perhaps a Game Over screen, and more. We could go ahead and define all these types of screens in our game definition. For example we could say that a Game Over screen is a screen with a bitmap and some text, which would be enough in most cases. But what if a programmer some day wants to have a Game Over screen with sprites moving around? Its better to define all these kinds of extra screens as levels as well, even though they usually will contain much less than ordinary levels.

However, levels alone are not enough. There is information that has to be kept throughout the entire game, like the player's number of lives, total score, etc. This kind of information usually controls the flow of the game. For example, the number of lives will often determine whether or not the player may retry a level. The problem is that this kind of information differs very much for each game.

For this reason we will introduce a *GameState*, a type that can be defined by the user. All information that belongs to the game, rather than to one level of the game should be kept here. A game is then parameterized with this GameState. Because this information will often be changed during the levels and not only in between, it should also be available within a level. As we will see later, this state is even passed on to the objects in a level, because they will generally change the state.

Some of the GameState information will be *statistics* that are shown on the screen during a game (like the player's score). These statistics will also differ a lot in various games. Therefore we will add a function to the definition of a game that gives a list of such statistics, which can of course be empty.

There are two more functions that belong to this game definition: quitlevel and nextlevel. The function quitlevel tells the game library when the level can be ended. This is often determined by information from the GameState. After ending one level, we need to know what the next level will be. The function nextlevel returns a level number, position in the list of levels (1…n) or 0 to quit the entire game.

It might look like a good idea to add a main character to the definition of a game. Although most platform games have one main character throughout the entire game, we might not want this for some games. Besides, we probably don't want our main character in every level, since levels now have a broader meaning.

Our definition of a game is now:

```
:: Game gs
   = { levels     :: [Level (GSt gs)]
     , quitlevel  :: (GSt gs) → (Bool, GSt gs)
     , nextlevel  :: (GSt gs) → (Int, GSt gs)
     , statistics :: (GSt gs) → ([Statistic], GSt gs)
     }
```

To use the type GSt, the programmer must include StdGSt. The GSt type is an abstract data type:

```
:: *GSt gs
   = { gs :: gs
     , tb :: !*OSToolbox
     }
```

This type is passed on from one function to the next throughout the library, to make sure the functions are evaluated in the correct order. In a game specification, the programmer defines functions that accept a GSt and also return one. These functions themselves do nothing with this GSt, except use it to call other functions and return it as (part of) their result.

5.2 Defining levels

As we have seen in Chapter 2, a level contains a number of *layers* that can scroll and one *bound map*. The layers are only to make the game visible for the player and do not affect the game itself. Therefore, a level could also have no layers.

The bound map contains information needed to place objects in a level. Of course, the definition of all these objects is also needed. So we will also include a list of objects in our definition of a level. All these objects have an *object code*, which is the number found in the bound map to initialize the object.

Would a level be the place to introduce a main character? Again, for most games we could do that, but to make our complete game definition simpler, and to give the programmer more freedom, we will not define a main character at all! Instead, the programmer can only define *objects*. By giving an object certain properties, it will automatically become the main character.

The main difference between a main character and an ordinary object would be that the user controls it while other objects move by themselves. Besides, the main character is always on screen. Why not let any object be capable of reacting to user input? This will only expand the possibilities for the programmer. The object that the screen should follow can also be set during the game. This way, we could make games for two players and let the screen follow both players in turn.

We can still define a single main character (as an object) and let this object be the only object that reacts to user input and have the screen follow this object.

Another property of a level is the position at which it starts. In most platform games, levels start at the left and scroll to the right, but any starting position is possible. Therefore we will add an initial position to our definition of a level. This is the position in the bound map. The layers will all be positioned according to their movement function (see §2.4).

A level can also have background music, which is played during this level. Usually such music is repeated until the level ends.

The definition of a level is now:

```
:: Level state
   = { boundmap     :: BoundMap
     , initpos      :: Point
     , layers       :: [Layer]
     , objects      :: [Object state]
     , music        :: Maybe Music
     , soundsamples :: [SoundSample]
     , leveloptions :: LevelOptions
     }
```

As we see here, the game state is passed on to the objects.

Point is a predefined type, which contains two integers, w and h. Maybe is also a predefined type and is in this case either *Just* this music or *Nothing*.

The order of the layers is important, because the layers are drawn in this order. The most distant layer must be the first one in the list and the closest layer will be the last.

Music, played in the background, is identified by a file pathname (*musicfile*), which is a standard MIDI file and two flags stating whether the music should restart when it ends (*restart*) and whether the music should continue after the level ends (*continue*):

```
:: Music
   = { musicfile :: String
     , restart   :: Bool
     , continue  :: Bool
     }
```

Each level can contain a list of sound samples that can be played during that level. These sound samples are loaded before the level starts to avoid having to wait every time before hearing a sound sample. Sound samples are identified in the program as follows:

```
:: SoundSample
   = { soundid      :: SoundID
     , soundfile    :: String
     , soundbuffers :: Int
     }

:: SoundID
   :== Int
```

The *soundid* can be any number so we can indicate which sound we want to play during the game. The *soundfile* is either the name of a standard WAV file or the *resource identifier* of the sample. Because Clean does not support resources in the executable (yet), we have to use separate files for the sound samples.

The *soundbuffers* field defines how often the same sound sample can be played at once. If this value is 1, the sample can only be played if it is not already being played.

The *LevelOptions* type contains some settings that apply to a level, especially meant to be used for testing in an early phase.

```
:: LevelOptions
   = { fillbackground :: Maybe Colour
     , escquit        :: Bool
     , debugscroll    :: Bool
     , fadein         :: Bool
     , fadeout        :: Bool
     }
```

By using the *fillbackground* color, the complete background may be set to any RGB color. Normally, no background color is needed, because the layers will fill the screen. However, all the layers can be transparent or there might not be any layers at all. The backgroundcolor has type *Colour*, already defined in the Clean Object I/O library. This can be either a normal color name or an RGB value.

While testing a game, it is often easy to be able to exit very right away. By setting *escquit* to True, hitting the *Escape* button will always quit the level.

Another useful option for testing is *debugscroll*. This makes the arrow keys automatically scroll the level, so the programmer will not have to define a complete object to move in the level only to see if the level is okay.

The *fadein* and *fadeout* options specify whether or not fading is used when the level appears and disappears. Without fading, the screen will suddenly change from one level to the next. If we do use fading, the screen will first slowly turn black and the next level will then appear smoothly.

5.3 Definition of a bound map

A bound map contains the objects in a level and the bounds (see §2.10). Because we will usually use a level editor to design our levels, the bound map will normally be generated by the level editor. But sometimes we might have a very simple level, like a black screen, for which we would like to specify the bound map by hand.

The bound map is actually a two dimensional array of integer values, that we will represent as a list of arrays of the type Int. The arrays represent the horizontal lines (indexed from left to right). The number of these arrays in the list sets the vertical size of the bound map.

The game engine interprets the bound map's integer values as follows:

Bit 0 – Top of the block is solid
Bit 1 – Left side of the block is solid
Bit 2 – Bottom of the block is solid
Bit 3 – Right side of the block is solid

Bit 4…7 – Reserved

Bit 8…31 – Map code

Though the level editor we will use, EDLEV, only supports 8-bit map codes, the programmer can also let objects change the bound map during the game and use any 24-bit integer value.

The information in the bound map is meant to be generated by the level editor and to be read by the game engine and therefore is not very readable. However, the programmer can use functions to get the information from bound map values instead of looking at these bits.

The size of the blocks in the bound map determines the size of the entire level (the size of the layers is not relevant, as they are repeated). The size of a level in pixels is the size of the bound map multiplied with the block size. Usually, the block size of the bound map will be equal to the tile size of the front layer.

Whenever the screen scrolls, new objects may have to be initialized because they are close enough to the visual screen to become active. Another property of the bound map is the area around the visual screen in which the game engine should look for objects that have to be initialized. This distance is defined by *startobjx* and *startobjy*. In some games this distance could be only one block, which makes objects appear at exactly the same place every time as we go through a level. We can also set this value to a very large number and all the objects become active as soon as the game starts. Objects also have their own *forget* property which indicates how far they can move from the screen before becoming inactive. This is discussed later.

We might want to use map codes for more than just initializing objects. For example, we could indicate water with a map code, so every time an object sees this code it will know it is under water. We would not want the game engine to keep trying to initialize new objects because of this map code used for water. Therefore we will introduce a minimum value *objstart*, that is considered to be an object. All codes below this value are ignored when the game engine looks for new objects. However, events are generated when an object touches such a map code.

This concludes our definition of a bound map:

```
:: BoundMap
   = { map       :: [{#Int}]
     , blocksize :: Size
     , objstart  :: Int
     , startobjx :: Int
     , startobjy :: Int
     }
```

The type Size is a predefined type here, containing two integers w and h.

5.4 Defining layers

Just like a bound map, a layer usually contains elements that are generated by the level editor. The only part of a layer that the programmer will always have to define is the movement function (see §2.4).

As we've seen before, a layer is made of *tiles* from one *game bitmap*. These tiles are indexed and a two dimensional array of these index numbers will define a layer. As discussed in §2.7, we can use negative numbers to indicate a *tile sequence*. We will use this coding system to define a layer.

We can also expand the definition to be able to mirror some tiles (or even sequences) in our layer map horizontally or vertically. We can do this by adding once or twice the number of tiles in the entire game bitmap to the value, or in case of a sequence, subtracting once or twice the total number of sequences. The coding of a layer map value x follows:

Value of x	*Meaning*
0	Empty tile, layer is transparent at this position
1 … n	Tile number x (n is the number of tiles in the game bitmap)
n + 1 … 2n	Tile number x - n, mirrored horizontally
2n + 1 … 3n	Tile number x - 2n, mirrored vertically
3n + 1 … 4n	Tile number x - 3n, mirrored both horizontally and vertically
-1 … -m	Sequence number -x (m is the total number of sequences)
-m - 1 … -2m	Sequence number -x - m, mirrored horizontally
-2m - 1 … -3m	Sequence number -x - 2m, mirrored vertically
-3m - 1 … -4m	Sequence number -x - 3m, mirrored both horizontally and vertically

Again, this type of coding is done by the level editor and not by the programmer. However, for very simple levels (like a screen with only one bitmap on it), this system is easy to use.

The definition of a layer is now:

```
:: Layer
   = { bmp       :: GameBitmap
     , layermap  :: LayerMap
     , sequences :: [TileSequence]
     , movement  :: Movement
     }

:: LayerMap
   :== [{#Int}]
```

As we can see here, we use a list of arrays for the layer map, just like we did for the bound map.

The tile sequences are also generated by the level editor and have the following type.

```
:: TileSequence
   = (Int, Sequence)

:: Sequence
   = [(Int, Int)]
```

The first integer value from the TileSequence tuple is the sequence's (negative) number, which corresponds with the number found in the layer map. The sequence is defined as a list of tuples of two integers, the first is the tile number and the second is the number of frames to display this tile.

The movement function is defined as follows:

```
:: Movement
   :== Point GameTime → Point

:: GameTime
   :== Int
```

The first *Point* is the position of the bound map and the *GameTime* is the number of frames that have passed since the level started. This function is called every frame, before the game engine draws the layers, and should return a point at which the layer is to be positioned.

5.5 Game bitmaps

We have seen that a game bitmap is a collection of smaller bitmaps, called tiles. These tiles all have the same size (the *unitsize*). Some bitmaps are transparent and have a transparent color, pixels of this color are always skipped when the bitmap is drawn.

There are different kinds of bitmaps, which makes specifying the transparent color difficult. An 8-bit (256 color) bitmap would use a palette index to indicate the transparent color. An RGB bitmap doesn't use a color palette and therefore needs an RGB color value as it's transparent color. For the rest of the game, the number of colors of the bitmap is not at all important, so we would rather not include it in our game definition. But how can we then specify the transparent color?

The solution is not to specify the color, but a coordinate of the bitmap that contains this transparent color instead. It doesn't matter anymore if the bitmap has 256 colors or more, and besides, we can change the RGB value of this color in the bitmap without having to change our game definition. The position of the transparent point is calculated modulo the size of the bitmap. This makes it easy for the programmer to indicate edges of the bitmap without knowing the exact sizes. For example, the bottom right corner of a bitmap can be indicated by (-1, -1) instead of (*width*-1, *height*-1).

Here is the definition of a game bitmap:

```
:: GameBitmap
   = { bitmapname  :: String
     , unitsize    :: Size
     , dimensions  :: (Int, Int)
     , transparent :: Maybe Point
     }
```

The bitmap name can be either a file name or a resource identifier. Because Clean currently does not support resources, all game bitmaps must be stored in separate files.

The tuple *dimensions* indicates how many smaller bitmaps are used (first horizontal and then vertical).

5.6 Defining objects

The objects are probably the most interesting part of this specification, because the programmer should be able to define their behavior, but the game engine has to make these objects work.

Because objects can be very different, the information needed to control these objects can not all be defined here. So the programmer should be able to create any type of object and add all the information needed to control it. But we still have to be able to control all the objects together in our library. The solution here is to add an object state.

There is one object, the *AutoInitObject*, which is always automatically initialized when a level starts. This object has code 0, so it can not be initialized from the level map. To use the AutoInitObject, we can simply define any object in our game and give it 0 as object code.

5.6.1 The complete object definition

We will start by looking at the complete definition of an object. In this definition, state is the type of the local data used by the object. The parameter *gs* is the game state (see §5.1).

```
:: Object gs
   = E.state:
   { objecttype :: ObjectType
   , sprites    :: [Sprite]
   , init       :: !SubType !Point !GameTime !gs
                   → *(!*(state, ObjectRec), !gs)
   , done       :: !*(state, ObjectRec) !gs
                   → *(!*(ObjectType, SubType), !gs)
   , move       :: !*(state, ObjectRec) !gs
                   → *(!*(state, ObjectRec), !gs)
   , animation  :: !*(state, ObjectRec) !gs
                   → *(!*(state, ObjectRec), !gs)
   , touchbound :: !*(state, ObjectRec) !DirectionSet !MapCode !gs
                   → *(!*(state, ObjectRec), !gs)
   , collide    :: !*(state, ObjectRec)
                   !DirectionSet !ObjectType !ObjectRec
                   → *(!*(state, ObjectRec), !gs)
   , frametimer :: !*(state, ObjectRec) !gs
                   → *(!*(state, ObjectRec), !gs)
   , keydown    :: !*(state, ObjectRec) !KeyCode !gs
                   → *(!*(state, ObjectRec), !gs)
   , keyup      :: !*(state, ObjectRec) !KeyCode !gs
                   → *(!*(state, ObjectRec), !gs)
   , userevent  :: !*(state, ObjectRec) !EventType !EventPar !EventPar
```

```
                    → *(!*(state, ObjectRec), !gs)
    }
```

5.6.2 States

We will define an object's (local) state as an existential quantified type (see §5.6.1). This will allow us to use a different type of state for each object. We can still however create a list of all the objects and work with the separate objects in this list.

This object state is defined by the programmer and can have any type. It will usually be a record, containing several elements that are needed to control the object. However, for simple objects, this state can also be empty.

Because our game engine has to run the objects and check collisions, some information will have to be the same for every object. Every object will have a size, a position and a lot more. The state will be used to store information that is not standard to every kind of object.

Now, we have three kinds of states: the game state (§5.1), the object state (local) and the object record (§5.6.4), which is also used by the game engine.

5.6.3 Sprites

Every object has a list of sprites. We use the following definition of a sprite:

```
:: Sprite
   = { bitmap   :: GameBitmap
     , sequence :: Sequence
     , loop     :: Bool
     }
```

Sprites work just like tile sequences, but they can end. If the *loop* property is set to False, the game engine will generate an *animation* event when the sequence has ended (see §5.6.5.4).

Each sprite can have its own game bitmap. This means that an object can have sprites of different sizes. This has no influence on the size of the object itself.

The *currentsprite* field of the object record (see §5.6.4) contains the index number of the sprites that is displayed. If this number is 0 or larger than the number of sprites in the *sprites* list, the object will become invisible.

5.6.4 The object record

The (standard) object information needed by both the game engine and the Clean programmer is stored in an *object record*. For dealing with most events, such an object record must be sent from the game engine to Clean and back. We will first give the complete definition of an object record and then explain the elements.

```
:: ObjectRec
   = { active          :: Bool
     , subtype         :: SubType
     , size            :: Size
     , pos             :: Point
     , offset          :: Point
     , currentsprite   :: Int
     , displayoptions  :: DisplayOptions
     , ownbounds       :: Bounds
     , bouncebounds    :: Bounds
     , collidebounds   :: Bounds
     , forgetdistance  :: Point
     , framecounter    :: GameTime
     , layer           :: LayerPosition
     , acceleration    :: RealXY
     , speed           :: RealXY
     , bounce          :: FVXY
     , maxspeed        :: RealXY
     , slowdown        :: FVXY
```

```
, skipmove        :: Int
, options         :: ObjectOptions
}
```

5.6.4.1 Active

The boolean value *active* indicates whether the object should remain active (alive). Whenever this flag is set to False, the object will be removed within one frame. Just before an object is removed, it's *done* event will occur (see §5.6.5.2).

5.6.4.2 Subtype

All objects have an ObjectType, which is the number in the bound map that initializes the object. It is possible to have several objects of the same type, with a few differences that we want to define in the level map. The SubType can be seen as an extra parameter of an object in the map. This parameter is a map code that is placed right above the code indicating the ObjectType. The value of the SubType should be always lower than the *objstart* value, specified with the bound map (see §5.3), because otherwise the SubType will be seen as an ObjectType that has to be initialized itself.

Figure 5-1 shows an example of objects defined in a bound map (in the level editor). In this example the *objstart* value is 10, so values lower than 10 are not seen as objects. There are three objects here, all with ObjectType 10, the first (on the left) has SubType 1, the next (in the center) has SubType 0 and the last object has SubType 2.

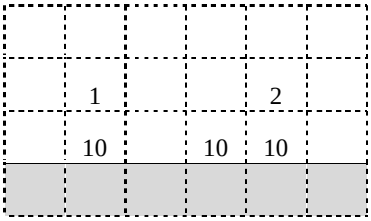

Figure 5-1: Defining object's SubTypes in the bound map

SubType is defined as an integer value:

```
:: SubType
   :== Int
```

5.6.4.3 Size

The size of the object is defined in this value of the type Size, which contains a width (w) and a height (h). This is the virtual size of the object and does not say anything about the size of the sprite on the screen, which is determined by the sprite itself. However, if *stretch* property is set to True, the sprite will be stretched to fit into this size (see §5.6.4.7).

5.6.4.4 Pos

The value of *pos* indicates the position of the object in the entire level (or actually in the bound map) and has the type Point. The x and y coordinates define the number of pixels between the top left corner of the level and the top left corner of the object, so this position does not depend on how the screen scrolls.

However, it is possible to specify the position from the top left corner of the *screen* instead, by setting the *static* flag in the *options*. This can be used for objects that should always stay at the same place on the screen, no matter how the layers scroll.

5.6.4.5 Offset

The *offset* of an object indicates the relative position of the object's sprite to its actual position. We could for example have a little fire as an object in our game. Touching this fire would hurt our main character, touching the smoke above the fire would do no harm. Instead of creating two objects, *fire* and *smoke*, we can make one single object that has the size and position of only the fire. The object's

sprite however is much higher and also shows smoke above the fire. The y value of the offset will be negative in this case, so that the smoke is drawn above the (dangerous) fire.

5.6.4.6 CurrentSprite

The *currentsprite* value is the index of the current sprite in the *sprites* list of the object. If this value is 0 or higher than the number of sprites in the list, no sprite will be displayed.

5.6.4.7 DisplayOptions

There are some advanced options to control the way the object's sprite is displayed. These are found in the *displayoptions* field. The *DisplayOptions* has the following definition:

```
:: DisplayOptions
   = { blink           :: Bool
     , stretch         :: Bool
     , mirrorleftright :: Bool
     , mirrorupdown    :: Bool
     , rotation        :: Rotation
     }
```

By setting *blink* to True, the sprite will only appear every other frame.

The *stretch* property determines whether the sprites are drawn at their actual size or scaled to exactly fit in the size of the object.

The *mirrorleftright* and *mirrorupdown* flags control the mirroring of the sprite.

A sprite can also be rotated at 90, 180 or 270 degrees. Therefore the *Rotation* type is defined as follows:

```
:: Rotation
   = NoRotation | Rotate90 | Rotate180 | Rotate270
```

5.6.4.8 Bounds

One of the events that we will be programming for objects is the *collide* event, which occurs at a collision between two objects. Every object will have its own collide function that can be defined for a collision with any other object. But there are also some collisions that we are not interested in or that could never take place.

For example, if we create a main character with a big gun that can shoot several bullets at the same time in all directions, we will generally be looking for collisions between these bullets and other objects in the level. We are not interested in collisions between two separate bullets. We could make a collide function for bullet that does nothing if the other object is also a bullet, but we could make everything a lot easier (and faster) if this kind of collision is ignored by the game engine in the first place.

For this reason, all objects have *ownbounds*, *bouncebounds* and *collidebounds* values. These are of type *Bound*, which has the following definition:

```
:: Bound
   :== Int
```

We should actually see this integer value as a collection of bits that represent bounds. Using these bounds, we can divide all our objects into separate groups that have similarities. The *ownbounds* field contains the object's own bounds, *bouncebounds* contains the types of bounds the object bounces with (bouncing is discussed later), *collidebounds* indicates the bounds for which the object's collide event is called.

A collide event will only occur for an object if its *collidebounds* field has one or more of the same bits set as the other object's *ownbounds* field. For example, we can define the following bounds:

```
BND_MAIN_CHARACTER :== (1 << 0)
BND_BULLET         :== (1 << 1)
BND_ENEMY          :== (1 << 2)
BND_POWER_UP       :== (1 << 3)
BND_WATER          :== (1 << 4)
```

For our main character, we would only be interested in collisions with enemies and power-up items:

```
maincharacter.ownbounds     = BND_MAIN_CHARACTER
maincharacter.collidebounds = BND_ENEMY + BND_POWER_UP + BND_WATER
```

Bullets, enemies and power-up items can be defined in a similar way:

```
bullet.ownbounds     = BND_BULLET
bullet.collidebounds = BND_ENEMY

enemy.ownbounds     = BND_ENEMY
enemy.collidebounds = BND_MAIN_CHARACTER + BND_BULLET + BND_ENEMY

powerupitem.ownbounds     = BND_POWER_UP
powerupitem.collidebounds = BND_MAIN_CHARACTER

water.ownbounds     = BND_WATER
water.collidebounds = 0
```

As we can see in this example, not all collisions between two objects will result in two separate collide events. Here we have an event for the main character colliding with water, but not for water colliding with the main character. In most cases however, both objects will check for their collisions and handle their own changes.

There are two predefined bounds, which work in the same way:

```
BND_MAP_CODES     :== (1 << 30)
BND_STATIC_BOUNDS :== (1 << 31)
```

These are bounds defined on the bound map. The first, BND_MAP_CODES indicates blocks in the bound map that have a map code. These must be codes lower than the *objstart* value, since higher values will be interpreted as object types. Figure 5-2 shows an example of the use of such map codes.

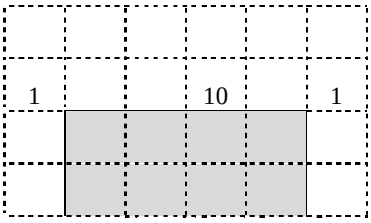

Figure 5-2: Using map codes to define bounds

In this example, the map code 10 indicates a creature walking on top of a small platform. We don't want this object to fall off the edge, so we use code 1 to indicate a bound for this object. During the game, map code 10 will disappear at the moment this creature is initialized. If we set *collidebounds* to BND_MAP_CODES, the collide event will occur every time the object reaches an edge. We could use this event to turn the object around. By setting *bouncebounds* to BND_MAP_CODES, this can be done automatically.

5.6.4.9 ForgetDistance

The property *forgetdistance* contains the *x* and *y* distance that an object can move away from the visible screen before it becomes inactive. The distance is measured from the point at which objects are initialized (see §5.3), defined by (startobjx, startobjy). This x and y values are bound map blocks.

5.6.4.10 FrameCounter

Each object has a *framecounter* which is of type *GameTime* which is defined as an integer value.

```
:: GameTime
   :== Int
```

This time is counted in *frames*. These frames are the updates on the screen, usually 70 or 100 per second. At initialization of an object, its framecounter contains the time since the level started. The

value is incremented automatically by the game engine, but may be changed by the programmer. Whenever the timer reaches the value 0, a *frametimer* event is generated (see §5.6.5.7). This provides a way to schedule an event at a certain number of frames in the future, by setting this timer to a negative value.

5.6.4.11 Layer

The *layer* property of an object determines the depth of the object in the game. The type is *LayerPosition*:

```
:: LayerPosition
   = InFront | AtLayer Int
```

Usually, objects will be placed in front of the layers, *InFront*, but we could for example have a moving cloud in the background that moves somewhere behind our scenery, with the most distant layer. We would then set this cloud at position *AtLayer 1*, so that the object is drawn after the first layer, but before the second.

The integer value used with AtLayer is not restricted to the actual number of layers in the level. It can therefore also be used to place some objects in front of others.

5.6.4.12 SkipMove

We have already seen a few of the events that control our objects, such as the *collide* and the *frametimer* event. Another of these evens is the *move* event, which is generated for each object every frame. The initial setup of the library was to control all normal movement with this move event. Information like the speed, gravity and acceleration of the object could be kept in the object's state. For an object that moves like a bouncing ball, for example, we would have the move function do something like:

```
state.speed = state.speed + state.gravity
objrec.pos = objrec.pos + state.speed
state.speed = state.speed + state.acceleration
```

Whenever this object would touch the ground or a wall, we could have an event make the object bounce by inverting its speed.

Although this does work fine for only a few objects, we see the performance drop fast as we add more objects to the game. Everything starts to move slower, and slower...

Evidently, the number of events we can handle in our game is not unlimited. If we have 100 objects, for example, at 100 frames per second, there will be 10,000 of these move events every second. Each time, the complete *object record* has to be sent from the game engine to Clean and back again every time. Its not that the calculations in the move function take such a long time (similar tests in pure C could handle a lot more moving objects without losing speed). Here, there are two threads, a C thread and a Clean thread. The game engine runs in the C thread, the move event runs in the Clean thread. Every time we generate this event, the C thread suspends itself and the Clean thread continues. After the Clean thread has done those few calculations, the control goes back to the C thread. In the mean time, the game engine still has to draw all the layers and run the game which already takes a lot of time without any of these events.

For this reason we want to limit the number of events that occur during the game. The *skipmove* property provides a solution for objects that move in a rather constant way, like most objects do. It contains the number of frames until the move event is generated again. By setting skipmove to SK_FOREVER, the move event is never generated and the object moves entirely according to the parameters that follow in the next sections.

5.6.4.13 Speed

The x and y speed of an object is defined as two Real values. These form the type RealXY:

```
:: RealXY
   = { rx :: Real
     , ry :: Real
     }
```

These *speed* values, rx and ry, are added to the object's position every frame. For example, an object with speed {rx = 1.0, ry = 0.0} will move to the right, one pixel per frame.

Because objects have to be displayed at actual (Integer) pixel locations, their position does not have the type RealXY, but Point. If the speed value is 0.25 for example, the position will only change once every four frames.

5.6.4.14 Acceleration

The *acceleration* of an object also has the type RealXY. This parameter controls how fast an object's speed increases or decreases. This value is added to the speed every frame.

For example, if we want to make an apple smoothly fall down from a tree, we could set its acceleration (in this case gravity) to {rx = 0.0, ry = 0.1}. This would make the vertical speed 0.1 at first, then 0.2, 0.3, 0.4, 0.5, … The relative position of the apple would be 0.1, 0.3, 0.6, 1.0, 1.5, … (which makes the movement parabolic).

5.6.4.15 MaxSpeed

As the name indicates, *maxspeed* contains the maximum x and y speed an object may ever have. This is the absolute x speed (left or right) or y speed (up or down). This makes it possible to control objects with a large acceleration.

5.6.4.16 SlowDown

By using *slowdown*, we can decelerate an object. There are two ways we can do this: by using a *slowdown factor* or a *slowdown value*.

A *slowdown factor* is a Real number, which is multiplied with the speed every frame. By setting this value to 0.9, for example, the object's speed will be decrease by one tenth every frame.

A *slowdown value* is subtracted from the object's speed every frame.

Because of these two ways we can slow down objects, *slowdown* has a special type FVXY:

```
:: FVXY
   = { fvx :: FV
     , fvy :: FV
     }

:: FV
   = Factor Real | Value Int
```

5.6.4.17 Bounce

Though collisions with bounds generate a *touchbound* event, the *bounce* field can define standard bouncing movement and the *touchbound* event can be turned off, just like the move event.

Just like *slowdown*, bounce can be used either as a factor or a normal value.

Used as a factor, bounce is multiplied with the speed at every collision. A bouncing ball would normally use such a factor, for example 0.5, for the y factor. This would make the ball bounce less high every time and eventually stop it.

Used as a value, bounce indicates a new (absolute) speed the object should have after it collides with a bound on the bound map. This can be used to make objects keep bouncing with a constant speed.

5.6.4.18 Options

The *options* field contains various flags that can be set for an object. Most of these flags help control which of the events are enabled or disabled. The type is *ObjectOptions*, defined as:

```
:: ObjectOptions
   = { ignorelevelbounds   :: Bool
     , checkkeyboard       :: Bool
     , allowkeyboardrepeat :: Bool
     , static              :: Bool
     , hdirection          :: HDirection
     , vdirection          :: VDirection
```

```
      , automirrorleftright :: Bool
      , automirrorupdown    :: Bool
      , freeze              :: Bool
      }

:: HDirection
   = DirLeft | DirRight

:: VDirection
   = DirUp | DirDown
```

When *ignorelevelbounds* is True, nothing happens when the object moves out of the level. If this value is false, all the edges of the level are seen as static bounds.

Because usually only one object in a level will be responding to the keyboard, objects only receive keyboard events when their *checkkeyboard* flag is set. There are two keyboard events, *keydown* and *keyup*. These are discussed later.

The *allowkeyboardrepeat* flag controls whether keys automatically repeat when they are pressed for some time. If set to False, pressing a key will always generate only one *keydown* event and one *keyup* event, no matter for how long the key is pressed. When this value is True, new *keydown* events will come after the key is pressed for some time.

The *static* flag makes an object move relative to the screen rather than to the level. If the static object has no movement, it will stay at the same place on the screen as we scroll through a level.

The *hdirection* and *vdirection* fields contain the objects current horizontal and vertical direction. These can be used for objects that have different sprites for different directions.

When the *automirrorleftright* and *automirrorupdown* flags are set, the object's sprites are automatically mirrored when the horizontal or vertical direction change.

As we have seen, an object's framecounter is incremented every frame and generates an event whenever it reaches zero. By setting the framecounter to a negative value, we can schedule an event. In some cases we would want the object to do nothing at all (to *freeze*) until this event occurs. Setting the *freeze* flag will therefore disable an object's move function whenever its framecounter contains a negative value.

5.6.5 Object events

Just like a normal application in a multitasking environment is event driven, we can run games in the same way. Some of the events are the same in both cases, like keyboard input, but a game also has different events like objects that collide. By programming such events we can easily create the *behavior* of an object.

We will give objects the following events. Of course, most objects will only need to use a few of these.

5.6.5.1 Init event

The *init* event occurs whenever the game engine finds an object code on the bound map that is near enough to the visible screen, or when an object is created by another object.

```
init :: !SubType !Point !GameTime !gs → *(!*(state, ObjectRec), !gs)
```

As we can see, the init function creates a new object state and an object record. The information given to this function is the object's SubType (a map code, less than the bound map's *objstart* value, that is placed above the map code), the location of the new object and the time in frames.

The init event is used to initialize the object record and its object state, however, an object can also use or even change the game state, because of the gs parameter.

5.6.5.2 Done event

The *done* event occurs when an object moves too far away from the visible screen (depending on its *forget* x and y values) or when its active property has been set to False.

```
done :: !*(state, ObjectRec) !gs → *(!*(ObjectType, SubType), !gs)
```

Here, we see that the done function returns an ObjectType and a SubType to the game engine. These are placed back into the bound map, at their original position, so that the object will become active again whenever the screen is close enough to the object. However, this ObjectType can also be set to zero. In that case the object will not return anymore.

The object's state and object record are passed to this function and will then seize to exist. Therefore, there is no point in changing the object's properties. But we can use this function to update the game state or to create new objects.

5.6.5.3 Move event

Whenever the *skipmove* value reaches 0, the *move* event is generated. This can be used to move the object. This movement will be in addition to normal movement, defined by the values of *speed*, *acceleration*, etc.

```
move :: !*(state, ObjectRec) !gs → *(!*(state, ObjectRec), !gs)
```

The move event is meant to alter object's properties, so it returns the same type it accepts. We can actually control anything from the move event, the object state, the object record and also the game state.

By default, the move event will occur every frame. Only by setting the skipmove value, this event can be ignored (see §5.6.4.12).

5.6.5.4 Animation event

The *animation* event occurs when an object has a sprite with an animation sequence that does not loop, and this sequence has ended.

```
animation :: !*(state, ObjectRec) !gs → *(!*(state, ObjectRec), !gs)
```

This event could for example be used for an explosion sprite. The explosion animation sequence will run once and then the object has to be removed. This could be done by setting active to False in the animation event (see §5.6.3).

5.6.5.5 Collide event

Whenever an object collides with another object (and the object's *collidebounds* match the other object's *ownbounds*), the collide event is generated.

```
collide :: !*(state, ObjectRec) !DirectionSet !ObjectType !ObjectRec
           → *(!*(state, ObjectRec), !gs)
```

Just like the move event, collide can change the object state, the object record and the game state.

The DirectionSet indicates the direction(s) of the collision.

```
:: DirectionSet
   = { top    :: Bool
     , left   :: Bool
     , bottom :: Bool
     , right  :: Bool
     }
```

This indicates the other object's bound that the object touched.

The ObjectType and the complete object record of the colliding object are also given to this function, however, these are not returned by the function, so the collide function will only define the collision for the object itself and not for the object it collides with.

If an object collides with several other objects at the same time, new events are generated for each collision. This is because handling a single collision is generally easier for the programmer than handling a list of collisions.

5.6.5.6 TouchBound event

The *touchbound* event is actually a special version of the *collide* event, for collisions with static bounds. In order to receive these touchbound events, the object's collidebounds must include BND_STATIC_BOUNDS and/or BND_MAP_CODES.

The game engine generates *touchbound* events when an object touches a bound or a map code in the bound map.

```
touchbound :: !*(state, ObjectRec) !DirectionSet !MapCode !gs
              → *(!*(state, ObjectRec), !gs)
```

MapCode is the value found in the bound map. As explained in §5.6.4.8, this code can be used to define bounds for objects.

```
:: MapCode
   :== Int
```

The DirectionSet contains the direction in which the object touched the bound. If the object were moving to the right when it hit a wall, the *left* value will be True, because it hit the wall's *left* bound. In some cases, more than one of these directions can be True at the same time.

5.6.5.7 FrameTimer event

The *frametimer* event occurs whenever the object's framecounter becomes zero.

```
frametimer :: !*(state, ObjectRec) !gs → *(!*(state, ObjectRec), !gs)
```

This event will usually be used to schedule an event a number of frames in advance.

5.6.5.8 KeyDown and KeyUp event

The *keydown* and *keyup* event are generated for all objects that have the *checkkeyboard* flag set, every time the user presses a key on the keyboard.

```
keydown :: !*(state, ObjectRec) !KeyCode !gs
           → *(!*(state, ObjectRec), !gs)

keyup   :: !*(state, ObjectRec) !KeyCode !gs
           → *(!*(state, ObjectRec), !gs)
```

The KeyCode is an integer value indicating a game key. These game keys are defined as constants and start with GK_ (for example GK_ESCAPE, GK_LEFT).

```
:: KeyCode
   :== Int
```

The *allowkeyboardrepeat* flag (see §5.6.4.18) controls whether there will be several keydown events if the key is pressed for a long time.

5.6.5.9 User event

The *userevent* event is a kind of broadcast event that objects can generate to send messages to itself or other objects.

```
userevent :: !*(state, ObjectRec) !EventType !EventPar !EventPar
             → *(!*(state, ObjectRec), !gs)

:: EventType
   :== Int

:: EventPar
   :== Int
```

These events are defined entirely by the programmer and can have two parameters. For example, a main character could send (broadcast) its position to all other objects once in a while, so "intelligent" enemies can follow and attack the main character.

5.7 Statistics

Text items in our games will be defined as *statistics*. As we have seen, the type Game contains a function that returns a list of statistics to the game engine to be displayed.

Here is the definition of these statistics:

```
:: Statistic
   = { format    :: String
     , value     :: Maybe Int
     , position  :: Point
     , style     :: Style
     , color     :: Colour
     , shadow    :: Maybe Shadow
     , alignment :: Alignment
     }
```

The first field *format* contains a C-like format string in which the value is displayed, for example: “Coins: %d”. Here, %d will be replaced by the actual number of *value*.

The *value* of a statistic is not required, since a statistics may also be a normal string with only text. In that case this value can be set to Nothing.

The screen position at which the statistic should be displayed is specified by the Point *position*, which contains two coordinates, x and y, relative to the top left corner of the screen.

The *style* parameter can be used to specify how the text is written. The *font name* and *size* can be selected as well as *bold* and *italic*.

```
:: Style
   = { fontname :: String
     , fontsize :: Int
     , bold     :: Bool
     , italic   :: Bool
     }
```

The color of the text is defined by *color*, which has type *Colour*, which can be either a normal color name or an RGB value. By using the *shadow* attribute we can easily create a shadow under our text. The Shadow type is defined as:

```
:: Shadow
   = { shadowpos   :: Point
     , shadowcolor :: Colour
     }
```

Here, *shadowpos* is the relative position of the shadow to the position of the text. This could be for example {x = 1, y = 0} to make shadow only appear on the right side of the letters. The shadow text is written with color *shadowcolor*.

There are also some *alignment* options, which make positioning the statistics easier.

```
:: Alignment
   = { xyfromscreencenter :: (Bool, Bool)
     , xycentered         :: (Bool, Bool)
     }
```

The first tuple, *xyfromscreencenter* indicates that the position is calculated from the center of the screen. If this value is (True, False), for example, a statistic at position (x = 0, y = 0) would be displayed at the top of the screen, starting exactly in the center.

The second, *xycentered* indicates that the position is the center of the text that will be written.

By setting both these parameters to (True, True), text with position (x = 0, y = 0) will be displayed exactly at the center of the screen.

Chapter 6
Game functions

Part of the definition of a game, given in the previous chapter, consists of object events. These events are functions that can change either the object's state, its properties, or the game state. There are also some library functions that can be used in the event code. This chapter describes these functions, which are declared in StdGame.dcl. The first, *OpenGame* is not used in events, but is needed to start the game engine. This is the function that interprets a complete game specification.

6.1 Starting the game engine

Because the game engine is based upon the Clean Object I/O library, the start of a game looks like the start of any Clean I/O program. There is only one new I/O function, OpenGame, which is called with the game specification as a parameter. This function interprets the game definition and runs the complete game. The following example shows the code needed to start the game engine and run a game specification named *gamedef* starting with *initialgamestate* as the game state.

```
import StdEnv, StdIO
import StdGame, StdGameDef

Start world
    = startIO 0 0 [init] [ProcessClose closeProcess] world
where
    init ps
        # (_, ps) = OpenGame initialgamestate gamedef [] ps
        = closeProcess ps
```

Now, we will take a look at the *OpenGame* function, which has the following type definition:

```
OpenGame :: gs (Game gs) [GameAttribute gs] !(PSt .l .p)
            → (ErrorReport, !(Pst .l .p))
```

The game state (gs) can have any type, which allows us to create a unique game state for each different game (see §5.1). This type is used as a parameter for the type Game, so (Game gs) is the complete game definition that can be used together with a game state that has type gs.

There are also some attributes that can be passed to the OpenGame function in the form of a list.

```
:: GameAttribute gs
   = ScreenSize Size
   | ColorDepth Int
```

The first, screensize is the size of the window (or actually the screen) in pixels. The default is {w = 320, h = 240}. Only some sizes can be used, depending on the hardware and the installed drivers. ColorDepth indicates the number of color bits per pixel. The default is 8 (256 colors), but on most systems 16, 24 or 32 can also be used.

6.2 Creating objects

As we have seen before, we can create objects as codes in the bound map. The game engine initializes these objects as we scroll through a level. But in some situations we would like to create new objects during the game, without having to define these in the bound map. For example, if our main character can shoot, we would like to create an unlimited number of bullets as new objects during the game.

To create new objects, we can use the CreateNewGameObject function, which has the following definition:

```
CreateNewGameObject :: !ObjectType !SubType !Point !(GSt .gs)
                       → (GRESULT, (GSt .gs))
```

This function creates a new object with type ObjectType and subtype SubType at position Point and then generate an *init* event for the object.

Objects created with this function can have any value as ObjectType, unlike objects defined in the level map, which can only have the codes 01 … FF.

The result of the function (and most other functions) is returned as GRESULT. This function currently only returns GR_OK.

6.3 Focusing an object

In our game definition there is still no way to determine how the screen should scroll. Usually, we want the screen to follow our main character. In some cases however, we might want the screen to stop scrolling (for example if the main character looses a life and falls off the screen).

We will allow any object to have the *focus*, which makes the game engine follow the object. If there is no object that has the focus, the level will not scroll.

The following function will set the focus to the object that calls it:

```
CreateObjectFocus :: !ObjectFocus !(GSt .gs)
                     → (GRESULT, (GSt .gs))

:: ObjectFocus
   = { scrollleft      :: Int
     , scrollup        :: Int
     , scrollright     :: Int
     , scrolldown      :: Int
     , maxxscrollspeed :: Int
     , maxyscrollspeed :: Int
     }
```

The values *scrollleft*, *scrollup*, *scrollright* and *scrolldown* define the minimum space between the object and the edge of the screen. If the object moves nearer to the edge of the screen, the screen will scroll further (unless the level ends, of course). These values only take effect when the object moves.

The maximum scroll speed is defined by *maxxscrollspeed* and *maxyscrollspeed*.

6.4 Broadcasting events

In some cases we want our objects to be able to send messages to other objects. This can be done by using the function CreateUserGameEvent:

```
CreateUserGameEvent :: !EventType !EventPar !EventPar !EventTarget
                       !GameTime !(GSt .gs) → (GRESULT, (GSt .gs))

:: EventTarget
   = Self
   | AllObjects
   | BoundType Bounds
```

The first parameter, EventType is a user-defined number indicating the type of the event. Two parameters can be supplied. These are the parameters of the *userevent* function that occurs for the objects that receive the message.

An object can send messages to itself, to all the objects or to all objects with a certain bound type. Having an object send a message to itself may seem strange, but this is very useful because

CreateUserGameEvent also accepts a GameTime parameter that defines when the event will take place. This is the number of frames the game engine will wait before generating the event.

This provides a way to easily schedule all kinds of events for objects. For example, if we want a bomb in our game that destroys all surrounding objects, we can let the bomb create a user event with event type EV_BOMB and the x and y position as parameters to all objects. We can define each object's reaction to the explosion by programming its *userevent* event. If we don't want to define this event for every object, we can define a class of objects that respond to such an explosion as a bound type and have the bomb send the event only to objects that have this bound type.

6.5 Playing sounds

We also have a function to play sound samples. As we have seen before, a level contains a list of available sound samples. We can let objects play sounds by using the function PlaySoundSample:

```
PlaySoundSample :: !SoundID !Volume !Pan !Frequency !GameTime
                   !(GSt .gs) → (GRESULT, (GSt .gs))

:: Volume
   :== Int

:: Pan
   :== Int

:: Frequency
   :== Int
```

Each time we play a sample, we can specify its volume, the panning (balance between left and right), the frequency and how long to wait before the sample starts.

The volume can be defined by any value between MIN_VOLUME and MAX_VOLUME, which are predefined constants, defined as 0 and 10,000.

The pan value is any value between PAN_LEFT (-10,000) and PAN_RIGHT (10,000). The constant PAN_CENTER is defined as 0.

By changing the frequency at which we play a sound sample, we can make the sample sound higher or lower. The DEFAULT_FREQUENCY constant indicates the frequency at which the sound sample was recorded. Any other value can be used from approximately 100 to 100,000 Hz.

Although we can already play MIDI files as background music during a level, we can also take a sample of an instrument and make music by playing the sample at different frequencies. However, calculating the exact frequency of a note is not very easy.

A note sounds exactly one octave higher when we double the frequency. There are 12 half notes in an octave, so each half note distance factor is $h = \sqrt[12]{2} \approx 1.05946$. If we take a normal A at 440 Hz, we can calculate the frequency of a B by multiplying 440 with h^2 which makes 466 Hz, and the C by multiplying 440 with h^3 which makes 494 Hz. This is the *Middle C* (the note exactly between the *treble clef* and the *bass clef* in piano music scores).

Because this is rather complicated, we will define a module notes.dcl, which can supply these frequencies. This module contains a function note:

```
note :: Int → Frequency
```

The result is a frequency that can be used with PlaySoundSample. The integer parameter is the half note distance to the Middle C.

By combining several PlaySoundSample calls, we can create little tunes that can be played during our game. The following example shows a list of calls that together play the beginning of the well known Dutch song *"Sinterklaas kapoentje…"*

```
PlaySintSong gs
    (_, gs) = PlaySoundSample piano vol pan  7  0 gs  /* G */
    (_, gs) = PlaySoundSample piano vol pan  7 20 gs  /* G */
```

```
(_, gs) = PlaySoundSample piano vol pan  9 30 gs  /* A */
(_, gs) = PlaySoundSample piano vol pan  9 50 gs  /* A */
(_, gs) = PlaySoundSample piano vol pan  7 60 gs  /* G */
(_, gs) = PlaySoundSample piano vol pan  4 90 gs  /* E */
= gs
```

Of course we can also just record this song as a WAV file and use the sample instead. But in some cases, programming a little song like this is more convenient.

6.6 Changing the bound map

Using the functions GetBoundMap and SetBoundMap, objects can edit values in the bound map. This can be used to make changes to a level during the game. Both the map codes and the directions of the bounds can be changed.

```
GetBoundMap :: !Point !(GSt .gs) →
               (!GRESULT, (!Int, !DirectionSet), (GSt .gs))

SetBoundMap :: !Point (!Int, !DirectionSet) !(GSt .gs) →
               (!GRESULT, (GSt .gs))
```

The Point parameter indicates the location in the bound map. The tuple (Int, DirectionSet) contains the data that will be stored in or read from the bound map. The integer value is the map code and the direction set is contains True or False values for whether the upper, left, lower and right bounds of the block are set. Reading from a position outside of the map will result in map code -1 and all bounds set.

Chapter 7
Implementing the library

In this chapter we will look at the implementation of the game library. We will first look at the global architecture of the library, how the Clean code interacts with the part of the game engine written in C. Then we will discuss the internal representation of the game data type. Finally, we will look at how the C-code is organized and discuss the low-level functions, which are OS specific. These are the functions that will have to be rewritten in order to port the library to other platforms.

7.1 Structure of the library

If we look at a game, written in Clean, using the game library, we could define three levels of code. At the highest level, we have the code of the game itself, as the (game) programmer wrote it. At the second level, we have the part of the game library written in Clean and the lowest level is the C code. Actually, the middle level (the Clean library) is also divided into certain levels itself, but we will now look at the library as one level of code.

7.1.1 Levels of information

The three levels work with different kinds of information. Figure 7-1 shows which information concerning the game is available at these three levels of code:

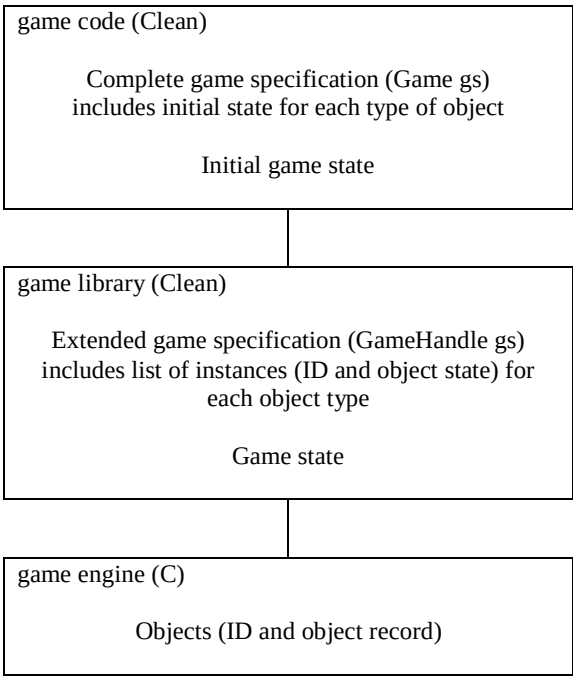

Figure 7-1: Available game information at three levels of code

As we see in this figure, the game code written by the programmer (the top layer) only contains initial values for the objects and for the game state. This is because the programmer only provides a specification of the game and then gives full control to the library by starting the interpreter function

OpenGame. The specification is meant to be as clear as possible for the programmer but does not have the optimal structure for the internal representation of the game. For this reason, there is a second game type, called *GameHandle*. This type is very similar to the normal game definition, but also has information that is not interesting to the programmer. We will discuss this extended game definition type in §7.2.

The most important information that controls a game is separated in three areas: the game state, the object state and the object records. The game state and the object state belong to the Clean part of the library and the object records are stored in the C part.

7.1.2 Communication between levels of code

The communication between the two Clean code levels is very simple, there is one call from the game code to the library (OpenGame) which runs the complete game. The complete game definition is given as a parameter as well as a list of attributes and the initial game state. The OpenGame function will then run all the levels in the game and finally return when the game has ended.

Communication between the Clean library and the C code is much more interesting, because of the call-back functions. Figure 7-2 shows some examples of call-back functions which take place while running a level. We use these call-back functions because all the object's events, written by the programmer are functions, which must be evaluated at runtime.

The Clean code and the C code both run in different threads. Switching between these threads takes time and shouldn't be done more than necessary. Some calls from the Clean thread to the C code can be done without switching these threads, such as the functions to store and retrieve the object records. These functions are called as if they belong to the Clean thread.

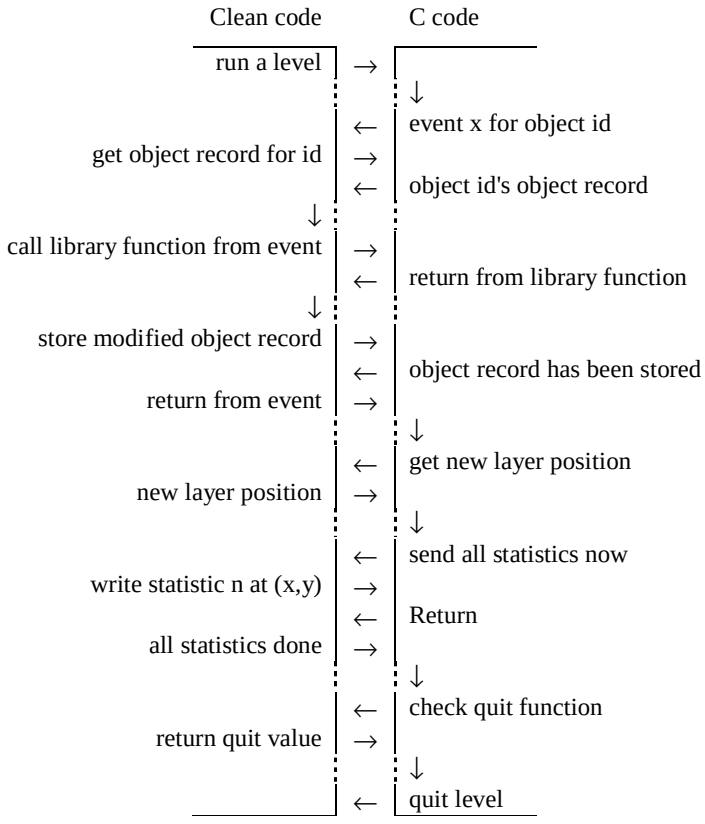

Figure 7-2: Call-back functions

As we see, after the game engine is started, it keeps calling Clean functions to control the game. During every frame, four different types of call-back functions are evaluated: object event handling (§7.1.2.1), layer positioning (§7.1.2.2), statistics (§7.1.2.3) and checking the quit function (§7.1.2.4). Here, we see

only a few examples of these call-back functions, the actual number of calls depends on the number of objects, layers, and statistics.

7.1.2.1 Object event handling

The object events are the most complicated call-back functions. This is because the code for the event handlers, defined by the programmer of the game, may contain calls to library functions (such as CreateNewGameObject and CreateUserGameEvent).

All object events are handled in a similar way. First, the game engine decides that the event will take place for a certain object. It sends a message to Clean telling the type of event, the object type and the ID of the object the event occurred for. In some cases, there is more information. For example, the information for the collide event will also include some information about the other object (object type and ID) and the directions of the collision. Each type of event has its own parameters.

The Clean library has a complete game definition, containing all the events for each object type, as defined by the programmer. In addition to this, the library also keeps a list of instances for each object type for objects that are active, containing their object state.

After receiving the information that a certain event has taken place, the Clean library will request the complete object record from the game engine. The complete object record is then sent to the Clean library and decoded in order to fit in the object record definition. Depending on the type of event, more information might be needed from the game engine. In case of a collide event, the complete object record of the colliding object will also be retrieved.

At this time, the Clean library has all information needed to evaluate the objects event code. The object state, the object record and the game state are the usual parameters for these events. The object's event code will usually change this information, so the new (modified) object record must be sent back to the game engine. To do this, it is first coded into a more simple type, which is sent to the game engine. Of course, the object state will also be stored in the list of instances.

Event handling is rather time consuming, because object records have to be retrieved and stored for every event. Even if the event function is very simple, like in the following example, the object record is still retrieved and stored.

```
\(state, objectrec) gs → ((state, objectrec), gs)
```

This is why we want to reduce the number of events that are actually generated by the game engine as much as possible (as described in Chapter 5).

During evaluation of the object's event functions, several library calls can take place. The game engine always knows which object's event is being handled, so it knows which object calls each function. This is needed for functions like CreateObjectFocus, which sets the focus to the object that calls this function.

7.1.2.2 Layer positioning

We have already seen that each layer has a function *movement*. This function returns a new position for the layer, given the current position in the game and the game time.

During each frame, this function is called after all object events have taken place. At this time, the positions of all objects are known, including that of the object which has the focus.

The movement function is always called for all layers just before the game engine starts to draw these layers.

7.1.2.3 Call-back functions for statistics

After all the layers have been drawn, it is time to write the statistics. As we have seen, the statistics are defined by a function *statistics* in the game definition.

First, the game engine sends a message to Clean, saying that it is now time to send all the statistics. The Clean library will then evaluate the *statistics* function, which returns a list of all the statistics. Instead of sending the complete list to the game engine, all elements are sent to the game engine one by one and directly drawn (written) to the screen.

After that, the call-back function returns, and all statistics are ready.

7.1.2.4 The quit function

Just like the statistics function, the quit function is also defined in the game definition. It only returns True or False (quit or don't quit yet). This function is always called after a complete frame has been displayed.

7.2 Internal representation of a game

When a game starts, its definition is first translated to a new data type, called GameHandle. This type is almost the same as the Game type, but has some additional information, which is not relevant to the programmer. Such information includes object ID's, bitmap ID's, lists of object instances, etc.

In this section, we will take a look at this internal representation.

7.2.1 GameHandle

The definition of *GameHandle gs* is just about the same as that of *Game gs*:

```
:: GameHandle gs
   = { levels`     :: [LevelHandle (GSt gs)]
     , quitlevel`  :: (GSt gs) → (Bool, GSt gs)
     , nextlevel`  :: (GSt gs) → (Int, GSt gs)
     , statistics` :: (GSt gs) → ([Statistic], GSt gs)
     }
```

The only difference (apart from the names) is that levels are defined as *LevelHandle (Gst gs)* instead of *Level (GSt gs)*.

The translation from a *Game gs* to a *GameHandle gs* is very simple. Only the levels change into a new type LevelHandle.

```
createGameHandle :: (Game .gs) → (GameHandle .gs)
createGameHandle {levels, quitlevel, nextlevel, statistics}
    = { levels`     = map createLevelHandle levels
      , quitlevel`  = quitlevel
      , nextlevel`  = nextlevel
      , statistics` = statistics
```

7.2.2 LevelHandle

When we look at the LevelHandle type, again we see that it is almost equal to the Level type, only objects has changed from [Object state] to [ObjectHandle state].

```
:: LevelHandle state
   = { boundmap`     :: BoundMap
     , initpos`      :: Point
     , layers`       :: [Layer]
     , objects`      :: [ObjectHandle state]
     , music`        :: Maybe Music
     , soundsamples` :: [SoundSample]
     , leveloptions` :: LevelOptions
     }
```

The *createLevelHandle* function is also very simple, just like *createGameHandle*, it copies all the fields from the level to the levelhandle. Only the objects are changed:

```
objects` = map CreateObjectHandle objects
```

7.2.3 ObjectHandle

The ObjectHandle is the first type that is really different from the original definition. Two new fields have been added: spriteids and instances.

```
:: ObjectHandle gs
   = E.state:
   { objecttype` :: ObjectType
   , sprites`    :: [Sprite]
   , spriteids`  :: [SpriteID]
   , instances`  :: [(InstanceID, state)]
   , init`       :: !SubType !Point !GameTime !gs
                    → *(!*(state, ObjectRec), !gs)
   , done`       :: !*(state, ObjectRec) !gs
                    → *(!*(ObjectType, SubType), !gs)
   , move`       :: !*(state, ObjectRec) !gs
                    → *(!*(state, ObjectRec), !gs)
   , animation`  :: !*(state, ObjectRec) !gs
                    → *(!*(state, ObjectRec), !gs)
   , touchbound` :: !*(state, ObjectRec) !DirectionSet !MapCode !gs
                    → *(!*(state, ObjectRec), !gs)
   , collide`    :: !*(state, ObjectRec)
                    !DirectionSet !ObjectType !ObjectRec
                    → *(!*(state, ObjectRec), !gs)
   , frametimer` :: !*(state, ObjectRec) !gs
                    → *(!*(state, ObjectRec), !gs)
   , keydown`    :: !*(state, ObjectRec) !KeyCode !gs
                    → *(!*(state, ObjectRec), !gs)
   , keyup`      :: !*(state, ObjectRec) !KeyCode !gs
                    → *(!*(state, ObjectRec), !gs)
   , userevent`  :: !*(state, ObjectRec)
                    !EventType !EventPar !EventPar
                    → *(!*(state, ObjectRec), !gs)
   }

:: SpriteID
   :== Int

:: InstanceID
   :== Int
```

The new field *spriteids`* is a list that corresponds with the list of spites and contains an ID (number needed for communication with the game engine).

The *instances`* field is used to store a list of all instances of the object that are active in the game engine. We need this to store the object state for each object. Objects are identified by their own unique ID (called InstanceID here).

The translation from *Object gs* to *ObjectHandle gs* is still simple, all fields are copied and spriteids` and instances` are initialized with the empty list. The instances` field remains empty until an object is created. The init function is then called, which returns a new state. This new state is stored in the instances` list.

The spriteids` field is filled when all the game bitmaps are initialized. This is done before the game engine starts to prevent delays during the game. When we initialize a new sprite, the game engine provides an ID, which we use to indicate that sprite. This ID is stored in the spriteids` field, at the same position as the original sprite in the sprites` list.

More internal information about the game is kept by the Clean library, but not in the gamehandle. The function that actually calls the game engine, PlayLevel, also keeps lists of ID's for bitmaps and complete layers, because their memory has to be released after the level is finished. We will not discuss this information here.

7.3 Low-level functions

The C-code belonging to the game library, is actually divided into two separate source files, `cGameLib.c` and `cOSGameLib.c`. The first contains the system independent parts of the game engine (moving objects, checking collisions, etc.) and the second contains the OS specific calls. This is to make it easier to port the game library to other platforms.

Declarations for the functions available from Clean are made in `intrface.h`, other functions are called from `cCrossCall.c`. The following source and object files are involved:

```
intrface.h
cCrossCall.c  (cCrossCall.obj)
cGameLib.h
cGameLib.c    (cGameLib.obj)
cOSGameLib.h
cOSGameLib.c  (cOSGameLib.obj)
DDUTIL.H
DDUTIL.CPP    (ddutil.obj)
DSUTIL.H
DSUTIL.C      (dsutil.obj)
```

The DDUTIL and DSUTIL files contain utility functions for working with DirectX (DirectDraw and DirectSound). To use the DirectX functions, the following library files have been included: ddraw_library (ddraw.dll), dsound_library (dsound.dll) and winmm_library (winmm.dll).

In order to port the library, all files beneath cOSGameLib.h will have to be changed. We will now look at the functions that would have to be translated. There will probably also be parts of cGameLib that would have to be changed, because this contains some code which has the same level as cCrossCall (such as setting up window classes and writing text).

7.3.1 Game result codes

Most functions in the game library return either a boolean value (True for success and False for failure) or one of the following result codes:

```
 0     GR_OK
-1     GR_FAILED
-2     GR_OS_ERROR
-3     GR_INVALID_BITMAP_ID
-4     GR_INVALID_SPRITE_ID
-5     GR_INVALID_MAP_ID
-6     GR_NOT_FOUND
```

These result codes are used mainly in the bitmap functions and are returned to the Clean layer. GR_OK indicates that nothing went wrong, GR_FAILED is used when a function fails and none of the other error codes are applicable. GR_OS_ERROR indicates an OS-specific error (in the Windows implementation, this is an error returned by DirectX calls). GR_INVALID_BITMAP_ID, GR_INVALID_SPRITE_ID and GR_INVALID_MAP_ID are returned when a bitmap id, a sprite id or a (layer) map id does not exist. The last result code, GR_NOT_FOUND is returned by functions that load files from disk (or from the resources) if the file cannot be found.

7.3.2 Screen functions

The first group of functions takes care of the screen or the window in which the game runs. To make a game run smoothly, we use double buffering. This means that we don't draw images directly to the (visible) screen, but to a buffer (the virtual screen). Once every frame, the visual and the virtual screen are swapped (this is called a page-flip).

The following screen functions are defined:

7.3.2.1 OSInitGameWindow

This function sets up a game window and initializes everything needed for page flipping. The parameters are global integer values: iScreenWidth, iScreenHeight and iBitsPerPixel, which are

initialized by the cross-call code in cCrossCall.c. The first two define the screen resolution, the third defines the color depth used in the game. There is actually a forth parameter, the boolean value bFullScreen. This indicates whether the game is played on a full screen or in a window. In the Windows implementation, this value is always True.

```
BOOL OSInitGameWindow ()
```

If setting up the game window goes wrong somewhere, the function returns False (the game window could not be opened), otherwise True (success).

7.3.2.2 OSDeInitGameWindow

This function is called before the program terminates and can be used to restore the screen mode etc.

```
void OSDeInitGameWindow ()
```

7.3.2.3 OSClearScreen

OSClearScreen clears the visual screen with black. This function has no parameters and no return value.

```
void OSClearScreen ()
```

7.3.2.4 OSFillBlack

This function clears an area on the visual or virtual screen with black (depending on the first boolean parameter `vis`, True: visual screen, False: virtual screen). The second parameter `r` describes the area on the screen that will be cleared.

```
void OSFillBlack (BOOL vis, RECT r)
```

7.3.2.5 OSClearVirtualScreen

This function clears the virtual screen. Here, a color can be given as parameter and the virtual screen will be filled with this color.

```
void OSClearVirtualScreen (COLORREF c)
```

7.3.2.6 OSBlit

The OSBlit function copies (*blits*) an area from the virtual screen to the visual screen. The only parameter `r` describes the area that will be copied.

```
void OSBlit (RECT *r)
```

7.3.2.7 OSFlip

This is the page-flipping function. This function waits until the video hardware is ready to draw a next frame on the screen and then swaps the visual and virtual screen, making the virtual screen visible. This flipping should be done as fast as possible.

```
void OSFlip ()
```

7.3.2.8 OSGetGameWindowHDC and OSReleaseGameWindowHandle

These two functions are used to get the window handle and release it again. This handle is needed when we want to write text to the window. Because writing text is not very specific for only games, this was not added to cOSGameLib. When porting the game library, there will probably already be a way to write text to a window.

```
BOOL OSGetGameWindowHDC (HDC *hdc)

void OSReleaseGameWindowHandle (HDC hdc)
```

7.3.3 Bitmap functions

The next group of functions contains the bitmap functions. Most of these functions are only used to work with the bitmaps themselves and may not seem very OS dependent. However, some platforms have a way to store these bitmaps in video memory, which will usually make working with them a lot faster. For this reason, all bitmap handling is done in cOSGameLib.

We work with game bitmaps, which are actually collections of small bitmaps stored together in one large bitmap.

7.3.3.1 OSInitGameBitmap

As the name indicates, this function initializes a game bitmap. The bitmap is loaded into memory so it can be displayed on the screen fast. The first parameter is the ID for the bitmap. If this value is 0, OSInitGameBitmap will create a new ID for the bitmap.

```
int OSInitGameBitmap (int id, char *name,
                      int bitmapwidth, int bitmapheight,
                      int blockwidth, int blockheight);
```

The second parameter, `name`, contains the file name or resource name of the bitmap.

The other parameters indicate the size of the bitmap (`bitmapwidth` and `bitmapheight`) and define how the large bitmap is divided into smaller bitmaps (`blockwidth` and `blockheight`).

The result of this function is either a valid bitmap ID (>0) or a Game Result Code.

7.3.3.2 OSFreeGameBitmap

This function frees a game bitmap from memory, given its bitmap ID.

```
int OSFreeGameBitmap (int id)
```

7.3.3.3 OSGetGameBitmapInfo

This function is used to retrieve information about a bitmap. Given a valid bitmap ID, it will store the width, height, block width, block height, number of blocks horizontally (`blockcountx`) and vertically (`blockcounty`) in the six integer parameters.

```
BOOL OSGetGameBitmapInfo (int id, int *width, int *height,
                          int *blockwidth, int *blockheight,
                          int *blockcountx, int *blockcounty);
```

7.3.3.4 OSFreeGameBitmaps

This function frees the complete list of game bitmaps that have been initialized.

```
void OSFreeGameBitmaps ()
```

7.3.3.5 OSSetTransparentColor

OSSetTransparentColor makes a bitmap transparent. There are three parameters: a bitmap ID, and a x and y position of a point in the bitmap which contains the transparent color.

```
int OSSetTransparentColor (int id, int x, int y)
```

7.3.3.6 OSInitBlockSequence

This function initializes a sequence of tiles. These tiles belong to a game bitmap and the sequence indicates a pattern in which these tiles are displayed. There are four parameters: the bitmap ID, the sequence ID, a pointer to the sequence and the length of the sequence.

```
int OSInitBlockSequence (int bitmapid, int seqid, char *seq, int len)
```

The sequence ID is the (negative) value used in the layer map to indicate the sequence. The sequence itself is an array of integer values for tile number and number of frames, in turn.

7.3.3.7 OSRunBlockSequences

This function makes all tile sequences advance one frame. All tile sequences have an internal counter indicating the current position. This counter will be incremented and reset if the sequence has ended. There are no parameters.

```
void OSRunBlockSequences ()
```

7.3.3.8 OSGetCurrentBlock

OSGetCurrentBlock returns the current tile from a tile sequence. This tile is indicated by its number inside the game bitmap and depends on the counter value of the sequence. The two parameters are a bitmap ID and a sequence ID.

```
int OSGetCurrentBlock (int bitmapid, int seqid)
```

7.3.3.9 OSDraw

This is the function that draws part of a game bitmap to the virtual screen. This function should be as fast as possible, because it is usually the bottleneck.

```
void OSDraw (RECT *dst, int id, RECT *src,
             BOOL mirlr, BOOL mirud, int flags)
```

The first parameter, `dst` indicates the area on the virtual screen to which the bitmap will be drawn. Then comes the bitmap ID (`id`) and the source area on the bitmap (`src`). If the sizes of both areas are not equal, the bitmap will be stretched to fit in the destination area.

There are two more boolean parameters for mirroring, `mirlr` and `mirud` (mirror left / right and mirror up / down) and an integer value `flags`, which contains a bit representation of the DisplayOptions type defined in the game definition.

7.3.4 Sound functions

The last group of OS-specific functions is formed by the sound functions. These functions are used to play background music and to play sound samples.

7.3.4.1 OSInitSound

This function can be used to initialize the sound drivers when the program is started. It returns a boolean value which indicates whether or not sound can be used.

```
BOOL OSInitSound ()
```

7.3.4.2 OSDeInitSound

OSDeInitSound is called before the entire program terminates and can be used to close down the sound system.

```
void OSDeInitSound ()
```

7.3.4.3 OSInitSoundSample

This function loads a sound sample into memory, so it can be played during the game (without having to load it at that time). There are three parameters: an ID for the sound sample, a file name (or resource identifier) and the number of buffers that will be reserved for this sample (so it can be played more than once at the same time).

```
BOOL OSInitSoundSample (int id, char *name, int buffers)
```

7.3.4.4 OSFreeSoundSamples

OSFreeSoundSamples releases all the sound samples that have been initialized.

```
void OSFreeSoundSamples ()
```

7.3.4.5 OSPlaySoundSample

This function plays a sound sample that has been initialized before. The first parameter is the ID of the sound sample, which is the same as the ID used with OSInitSoundSample. The other three parameters tell how the sample is played: volume, pan (left / right balance) and the frequency.

```
BOOL OSPlaySoundSample (int id, int volume, int pan, int freq)
```

7.3.4.6 OSPlayMusic

This function can be used to play background music (MIDI) during a game. This function has two parameters: `midifile`: the name of the MIDI file and a boolean value that indicates whether the music should restart when it has finished playing (`restart`).

```
BOOL OSPlayMusic (char *midifile, BOOL restart)
```

7.3.4.7 OSStopMusic

OSStopMusic stops any background music that is playing.

```
BOOL OSStopMusic ()
```

Chapter 8
Using the tools

This chapter describes the use of the tools needed to create sprites and complete levels. First we will look at the sprite editor GRED, then the level editor EDLEV and finally some conversion programs that are needed to convert sprites and complete levels to Clean code. Note that these tools were developed as prototypes and have certain limitations. All these tools work only in DOS. They were primarily designed only to get the job done and can be a little difficult to use. For this reason, one should first read this chapter and experiment a little with the tools, before starting to construct a complete game (which is described in the next chapter).

8.1 Using GRED

The sprite editor GRED will run on any PC (286 or higher) with VGA running DOS. The syntax to start GRED is:

```
GRED [filename] [-w:width] [-h:height]
```

For example:

```
GRED enemy.000 24 20
```

Without any parameters, GRED will start a new file with the size 32 by 24 pixels. The maximum size allowed for sprites is 160 by 160 pixels.

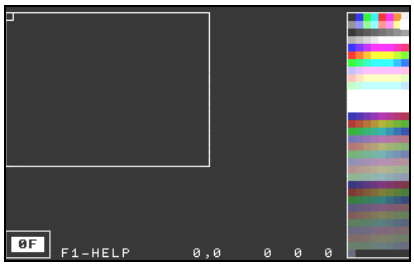

Figure 8-1: GRED

As shown in Figure 8-1, the screen is divided in a drawing area (left) and a color palette (right). On the bottom of the screen, at the left we see the current drawing color (white here, index 0F in the palette). The numbers in the middle indicate the current cursor position (here 0,0) and on the right we have the RGB values of the color at the cursor position (all zero here: black).

If a mouse drive is installed, there will also be a mouse pointer in the center of the screen. But GRED works very well with only a keyboard.

8.1.1 Cursor movement

When we first start GRED, the cursor is at the top-left of the drawing area. By using the arrow keys, we can move the cursor to any position on the matrix. In addition to the normal arrow keys for up, down, left and right, we can also use the **Home**, **End**, **PgUp** and **PgDn** keys for diagonal movement.

By pressing the **Tab** key, we can switch between the drawing area and the palette. In the palette area, we can move the cursor in the same way to select a color. After selecting a new color, we can press **Tab** again (or **Enter**) to select this color as the new (foreground) color or **Esc** to keep the old color. We can

also select a new background color by pressing **Space**. The background color is displayed in the box at the bottom left of the screen around the foreground color (in Figure 8-1, the background color is black).

Another way to select a new color is by simply clicking on the new color in the palette area with the mouse (left button for the foreground color, right button for the background color).

8.1.2 Drawing pixels

After selecting a color, we can now draw pixels. To draw pixels using the mouse, just press the left mouse button. Using the keyboard, we can draw a pixel by moving the cursor to the correct position and pressing the letter **x** (foreground color) or a **Space** (background color). Another way to draw pixels in the foreground color is by holding the **Shift** key down and moving the cursor. This will leave a 'track' of pixels, useful for drawing straight lines.

With the right mouse button, we can select a color from somewhere in the drawing (instead of from the palette). The same can be done by moving the cursor to a pixel with the desired color and pressing the letter **c**.

8.1.3 Editing the palette

Normally, when we start GRED, the palette is filled with the standard VGA palette values (see Figure 8-1). The 256 color mode uses 6-bit palette entries, so all RGB values go from 0 to 63. The number of unique colors we could possibly make with these RGB values is $64^3 = 262{,}144$. However, we are limited to a palette of only 256 colors. Therefore, the default palette is unlikely to be very suitable for our sprites and we will probably want to create our own.

In the palette area, we can edit the RGB values of a color by pressing the letters **r**, **g**, **b** to increment the red, green or blue value or the capitals **R**, **G**, **B** to decrement the values. Alternative keys are (numeric keypad: **Shift**-**7**, **8**, **9** (increment) and **Shift**-**1**, **2**, **3** (decrement).

As we have seen before, there is one color that is used as the transparent color. In GRED, this is palette index 0. Normally, this color is set to black, but by changing the RGB values of the transparent color, we can see what our sprite looks like against different background shades.

In order to use several of the advance functions in GRED (and EDLEV), the palette should be divided into rows of eight shades of the same color, from left to right, dark to light (see Figure 8-2).

Figure 8-2: Palette row in the palette area

GRED has a special function to make a row of smooth shades between two colors. This can be done by selecting the first color (black in this example) as background color (**Space**) and then moving the cursor to the last color (white) and pressing ***** on the numeric keypad. It is also possible to pick up a palette row (**Del**) and paste it somewhere else (**Ins**).

After defining a complete palette for our sprites, we save this palette as `DEFAULT.PAL`. GRED always looks for a palette file with this name at startup and automatically loads the palette if found. We can have one default palette in a directory, so we will usually use separate directories for sprites using different palettes (this is to avoid having to load a palette manually each time we edit sprites). Palette files can be loaded with **Shift**-**F9** and saved with **Shift**-**F10**.

8.1.4 Loading and saving sprites

GRED uses its own file format, which is very simple: the first two bytes indicate the width and height of the sprite and are followed by all the pixels. There are several functions to export sprites and generate Pascal, C or assembly code, but we will not need these here. We only have to be able to load and save our sprites (in GRED's own format), later we will use a conversion program to generate Clean code.

Files can be loaded either by entering the name at the command line, or by pressing the **F3** key. By default, new files are loaded into the current image at the cursor position. By moving the cursor to the

top left corner first, we can load a new sprite and completely forget the old one. If the sprite we want to load is transparent, we can use **Alt**-**F3** to place this transparent sprite on top of our existing image.

After editing the sprite, we use **Alt**-**x** to save our sprite and exit. There are more keys for saving (such as **F4**), but these will also generate extra files we don't need.

8.1.5 Creating animation sequences

As we have seen before, animation sequences are lists of sprites that suggest movement when displayed in sequence. The individual frames only have small differences. How do we create such animation sequences?

We will start by creating the first bitmap and saving it with a name followed by the extension `.000`. This will be our first frame. Then we save the same file again, now with extension `.001`. Now we have a sequence with two frames and we can go back and forth with **Ctrl**-**PgUp** and **Ctrl**-**PgDn**. We can now start editing the second frame and check how the animation between these first two frames looks. Every time we want to add a frame, we save one of the previous with a new number as extension and then make changes to the new frame. While moving through the sequence, the frames are automatically saved to disk.

8.1.6 Blocks

There are some functions in GRED that use blocks, such as drawing rectangles. Blocks are defined as the area between the last position at which **F5** was pressed and the current cursor position. So, to draw a rectangle, we go to one corner, press **F5**, then move to the opposite corner and press the key for a rectangle (**Ctrl**-**F5**). Other block functions include: Mirror left/right (**Shift**-**F1**), Mirror up/down (**Ctrl**-**F1**), Fill block (**Shift**-**F5**), Fill block with random colors (**Alt**-**F5**), Make block lighter (**F7**) or darker (**Shift**-**F7**), Draw a circle (**Alt**-**F7**), Scale (**F8**) and Smoothen area (**Alt**-**F8**). For a more detailed list of available functions, see the `README.TXT` file that comes with GRED.

By using the **F5** key in the palette area, we can make palette rows active or inactive. Some of the block functions, like random fill, scaling and smoothening, will then only use colors that are selected. If no rows have been selected, only colors between the foreground color and the background color are used. For example, if we want to create a block of random pixels using only two colors, we can set select a background color and a foreground color right next to each other, then do the random fill.

8.1.7 Some more functions

There are a few more useful functions in GRED. First of all, by pressing **Alt**-**1**...**9**, we can change the size of the grid. **Alt**-**0** shows the sprite at actual size on an empty screen and **Alt**-**-** shows a complete pattern of the image.

We can change the size of the sprite with **Alt**-**H** (change horizontal size) and **Alt**-**V** (vertical size). It is also possible to insert a line (horizontal: **Ins**, **h** or vertical: **Ins**, **v**). We can move the entire image Up, Down, Left and Right with **Ctrl**-**u**, **d**, **l** and **r**.

There are functions to fill a horizontal or vertical line with the foreground color: **Ctrl**-**h** (horizontal line) and **Ctrl**-**v** (vertical line). The letter **f** fills an area (flood fill).

There is also a way to change color into another color in the entire image. This can be done by moving the cursor to a pixel with the old color and pressing **Alt**-**c**. We can then select a new color and this will be replaced everywhere in the image. Another very useful function is to change a complete group of colors. For example, in Figure 8-3 in the next section we have a drawing of a brown block, which only uses one palette row. We can easily change the color of the entire block to blue, for example, keeping all the original shades. To do this, we select the new color as foreground color (any color blue in this example) and press **Shift**-**Tab**. Then we have a blue block!

GRED even has an Undo function (**F2**), which works after most of the block functions (pressing **F2** again will restore the original image). There is also a Help function (**F1**), which contains a short list of the most important keys.

A complete list of function keys is given in the file `README.TXT`.

8.1.8 Example

To get a little more used to GRED, we will look at the steps needed to create the block shown in Figure 8-3. This block could be used as one of the building blocks for a level of a game. This example shows exactly how this sprite was made.

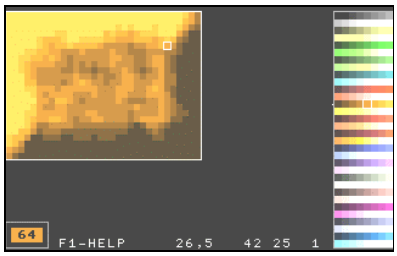

Figure 8-3: Example block

Here, GRED is started with the original `DEFAULT.PAL`, included with the program. First of all, we will select the color of the block (a palette row): **Tab**, ↓ (several times, choose a color), **F5** to select the palette row, and **Tab** again to return to the drawing area.

Now we create a block by pressing **F5** at the top left corner, and then **Ctrl**-**End**, to move to the bottom right corner. Here we press **Alt**-**F5** (random fill block) followed by **Alt**-**F8** (smoothen block).

Next, we will make the top and the left side lighter. We press **Ctrl**-**Home**, ←, then **F7** (lighter), **End**, **F7**, **End**, **F7**, **End**, … several times until the top looks right. The same for the left side: **Ctrl**-**Home**, ↑, then **F7**, **PgUp**, **F7**, **PgUp**, …

Finally, we will make the shadow at the bottom and at the right. We press **Ctrl**-**End**, **F5** to set the new block start position. We press ↓, **Shift**-**F7**, **End**, **Shift**-**F7**, **End**, … and **Ctrl**-**End**, →, **Shift**-**F7**, **PgUp**, **Shift**-**F7**, **PgUp**, … and we're done, we can view a pattern (**Alt**-**-**) and finally save the block and quit (**Alt**-**x**).

This small demonstration shows how fast some kind of sprites can be created in GRED. Using these block functions requires some practice, but will eventually be much faster than drawing the whole sprite with the mouse.

8.2 Using EDLEV

The level editor EDLEV works together with GRED and has the same system requirements. In order to use EDLEV, GRED must be properly installed and there must be a palette file named `DEFAULT.PAL` in the current directory. EDLEV has the following command line syntax:

```
EDLEV filename [blockwidth blockheight]
```

For example:

```
EDLEV level1 20 16
```

The default block size is 32 by 24 pixels, just like in GRED. When started, EDLEV scans the current directory for all existing blocks with extension `.000`, `.001`, `.002`, etc. In order to use a bitmap as a tile in the level editor, its extension must start with `.0`.. and the bitmap must use the `DEFAULT.PAL` palette.

Figure 8-4: EDLEV

In Figure 8-4 we see a screenshot of EDLEV. At the bottom of the screen, we see the blocks (tiles) we can work with. There are many more tiles, but only one row is displayed at a time. At the top, we see the editing area, in which we can create our level.

8.2.1 Cursor movement

The cursor movement in EDLEV is actually very similar to that in GRED. However, EDLEV has no mouse support, everything is done by keyboard. We move the cursor around with the arrow keys, again the **Home**, **End**, **PgUp** and **PgDn** keys control diagonal movement.

By pressing **Tab**, we can select one of our tiles. Here, again we can scroll through all our tiles with the arrow keys. After choosing a new tile, we can press **Enter** or **Tab** to select it or **Esc** to cancel.

8.2.2 Drawing tiles

After selecting a tile, we can draw this tile in the map at the cursor position by pressing the letter **x**. We can pick up a tile directly from the level map by pressing the letter **c** (instead of selecting this tile from the list of tiles).

Tiles are often only small parts of a larger image. For example, look at the top of the palm tree in Figure 8-4, which is actually one drawing, divided into nine separate tiles. We will usually want to draw these tiles in the same order. We can do this with **Alt-9**, which will draw a rectangle of 3x3 tiles, starting at the cursor position, starting with the current tile. We can do the same for 2x1 tiles (**Alt-2**) and 2x2 tiles (**Alt-4**).

We can also create a new tile, by moving to an empty block and pressing **Enter**. This will start GRED with a new empty tile. In the same way, we can edit existing tiles, by pressing **Enter** at the position of the tile we want to edit.

By pressing **Backspace** we can mirror the tile at the cursor position (left/right), **Shift-Backspace** will turn the tile upside-down.

8.2.3 Combining tiles

Although we use EDLEV to edit only one single layer at the same time, it is possible to place one sprite in front of another. These tiles will automatically be converted to one new tile later. To do this, EDLEV always works with two layers, in which tiles can be placed, a front layer and a back layer. We can switch layers by pressing **Space**. The border color around the screen indicates which layer we are editing (blue = background, green = foreground). Normally, we create most of the level as the background layer and occasionally we want to put a tile in front of another somewhere and use the foreground layer for this. By pressing **Alt-f**, we can turn all the foreground tiles on or off and with **Alt-b** we can toggle the background tiles. To swap the foreground and background tiles at a position we can press **Alt--**.

8.2.4 Shifting colors

If we use a palette that is organized in rows of eight colors (as described in §8.1.3), we can easily change the color of our tiles in EDLEV, without having to edit the tiles themselves. For example, if we take the brown block shown in Figure 8-3, we can place it all over our level in different colors. We do

this with the **+** and **-** keys, for shifting the color up and down by 8, or **Shift**-**+** and **Shift**-**-** to shift the color by one, making the tile a little lighter or darker. The left palm tree in Figure 8-4 uses the same tiles as the other one, but has color shift 2, which makes it lighter.

We can pick up a color shift value with **=** and recolor another block with the same value with **/**. Pressing ***** will restore the original color of a tile (set the color shift value to 0).

In the end, new tiles will be generated for all different colors of the same sprite.

8.2.5 Copying tiles

We can select a block in EDLEV by pressing **Shift**-**Space** at one corner and moving the cursor to the other. After selecting an area, we can copy it with **Shift**-**c**. Then we can move the cursor to a new position and press **Shift**-**x** to draw the entire block at the cursor position.

To copy and paste individual tiles, we can use the keys **[** (copy tile at cursor position) and **]** (paste the tile). This copies the entire tile, foreground and background, and color information.

8.2.6 Bounds and map codes

While editing a level, we can also define the static bounds (as described in §2.10). At each position in the map, we can define a Left, Right, Upper and/or Lower bound. These bounds are displayed by dotted lines at the edge of the tiles. We can turn bounds on or off using **Shift** together with the arrow keys on the numeric keypad. For example, **Shift**-↑ creates an upper bound for the tile at the cursor position. While editing, we can turn the bounds on or off with **Alt**-**z**.

We can also insert a map code at any position. Map codes are hexadecimal numbers from 01 to FF and can be used for any purpose, usually for initializing objects (see §2.10).

To enter a map code, we press **Ins** and then two digits. To remove a code, we simply type the code 00, which is never displayed.

8.2.7 Tile sequences

There is one special map code, that is recognized by EDLEV, code FF at the top left corner of the entire level. If we look back at Figure 8-4 again, we see the map code FF (top left), some sunflowers, again code FF and more sunflowers. This is the definition of two tile sequences.

If a level has any tile sequences, these must be defined at the top left of the level. Code FF on an empty tile indicates a sequence. If there are very many of these sequences, the next lines may also be used.

FF	FF					
30	5					
30	5					
FF	5					
20	5					
20						

Figure 8-5: Tile sequences example

Figure 8-5 shows an example of a small level with three tile sequences (the shaded areas indicate tiles). The actual level starts at column 3 and has width 5. The numbers 30, 20 and 5 indicate how long each tile is displayed. The first tile sequence has two tiles that are displayed in turn for 30 frames. Everywhere in the level where the first tile is placed, this sequence is shown. So in the level shown in Figure 8-4, everywhere where we draw a sunflower, it will automatically become a sequence.

8.2.8 Other functions in EDLEV

EDLEV is a little less advanced than GRED, it has no Undo function and no Help screen (yet). But here are some more useful functions in EDLEV.

Using **Alt-h** and **Alt-v**, we can set the horizontal and vertical size of the level to the cursor position. Just like in GRED, we can insert a horizontal line with **Ins**, **h** and a vertical line with **Ins**, **v**. The **F10** key shows a map of the whole level. We can quit EDLEV and save our level by pressing **Alt-x**, or quit without saving with **Esc**.

8.3 Generating Clean code

Now we have learned how to create sprites and complete layers, we will now want to use them in our Clean programs. There are still two conversion programs we need to run. The first will generate a game bitmap for a given level, including only the tiles that are used in this level. The second will translate all this information to a Clean module, which can then be included into our source code. The reason for two separate programs is that the level editor was not only designed for use with Clean, but also for other languages. So after creating a game bitmap we might want to convert the data to other formats instead of Clean code.

However, there is a script file CONV which calls both of these programs. This will be discussed in the next chapter, when we will specify a complete platform game.

8.3.1 MAKEDX

The program MAKEDX is the first of the two conversion programs. It will read a complete level, just like EDLEV, using the palette `DEFAULT.PAL` and the tiles which are located in the current directory. It then searches for all unique tiles and generates a game bitmap, which is a `.BMP` file. In addition to this game bitmap, MAKEDX also produces a map of the layer, but still in binary format. This file has the same name as the level, with extension `.LEV`. If our level file has tile sequences, these will be stored in a new file with the extension `.SEQ`, and if the level contains any static bounds or map codes, the bound map will also be written to a file with extension `.BND`.

These new files contain a representation of the layer, very much as defined in our game definition, but still in binary format.

8.3.2 MAP2ICL

The second conversion program is MAP2ICL. This program takes the output of MAKEDX and generates a Clean module. We'll now look at an example of how a level, made with EDLEV is translated to a Clean module.

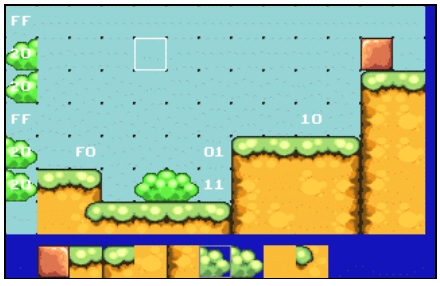

Figure 8-6: Example level TEST

Figure 8-6 shows our small level. It's name is `TEST` and it contains two tile sequences and some map codes. This level also has bounds, but these are not visible in this figure. We have nine different tiles.

Now we will run the first conversion program, MAKEDX:

```
MAKEDX TEST
```

After executing this command, we have four new files: `TEST.BMP`, `TEST.LEV`, `TEST.SEQ` and `TEST.BND`. The first is our game bitmap, displayed in Figure 8-7.

Figure 8-7: The new game bitmap TEST.BMP

After creating our game bitmap, we will now generate the Clean module. The program MAP2ICL accepts two parameters, the first is the name of the level file, the second is the name we want to use in Clean. Here, are level file name is `TEST` and we will let MAP2ICL generate definitions starting with TestLevel (for example: TestLevelBounds, TestLevelBitmap, etc.):

```
MAP2ICL TEST TestLevel
```

After running this program, two new files have been generated: `TEST.DCL` and `TEST.ICL`. The generated files always have the same name as the level (parameter 1) with `.DCL` and `.ICL` as extension. These files have the following contents:

`TEST.DCL`:

```
definition module TEST


import StdEnv, StdGameDef

TestLevelBitmap :: GameBitmap

TestLevelMap :: [{#Int}]

TestLevelBounds :: [{#Int}]

TestLevelSeq001 :: (Int, [(Int, Int)])
TestLevelSeq002 :: (Int, [(Int, Int)])

TestLevelSequences :: [(Int, [(Int, Int)])]
```

`TEST.ICL`:

```
implementation module TEST

import StdEnv, StdGameDef

TestLevelBitmap :: GameBitmap
TestLevelBitmap
  = { bitmapname  = "TEST.BMP"
    , unitsize    = { w = 24, h = 24 }
    , dimensions  = (13, 2)
    , transparent = Just { x = 263, y = 23 }
    }

TestLevelMap :: [{#Int}]
TestLevelMap = [{0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,2,0},
  {0,0,0,0,0,0,0,0,0,0,0,4,5},
  {0,0,0,0,0,0,0,0,0,0,0,6,7},
  {0,0,0,0,0,0,0,4,5,9,10,6,7},
  {9,10,0,-1,-2,0,12,13,7,14,6,13},
  {13,15,16,5,9,10,6,7,13,14,6,7}]

TestLevelBounds :: [{#Int}]
TestLevelBounds = [{0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,15,0},
  {0,0,0,0,0,0,0,0,0,0,0,3,1},
```

```
  {0,0,0,0,0,0,0,0,4096,0,2,0},
  {0,-4096,0,0,0,256,3,1,1,9,2,0},
  {1,9,0,0,0,4352,2,0,0,8,2,0},
  {0,8,3,1,1,9,2,0,0,8,2,0}]

TestLevelSeq001 :: (Int, [(Int, Int)])
TestLevelSeq001 = (-1,[(1,32),(3,32)])

TestLevelSeq002 :: (Int, [(Int, Int)])
TestLevelSeq002 = (-2,[(8,32),(11,32)])


TestLevelSequences :: [(Int, [(Int, Int)])]
TestLevelSequences = [TestLevelSeq001, TestLevelSeq002]
```

We can now include this module in our Clean code with the statement:

```
import TEST
```

Now we have access to the game bitmap, the layer map, the bound map and the sequences.

Chapter 9
Specifying a platform game

In this chapter we will build a complete platform game to give an example of how to use the library we've created. To avoid spending too much time on the graphics, we will remake an existing game: *Charlie the Duck*, a game originally written in Turbo Pascal. In Chapter 2 we have already seen a picture of a level from this game (Figure 2-2).

This new version of *Charlie the Duck* will only have a few small levels, because it is only meant as a demonstration. The most interesting part is the definition of the objects, the main character *Charlie* in particular.

We will start by creating a level that can scroll. Then we will add simple objects to the level and finally create the main character. After we have one complete level, we will build the rest of the game around it (title screen etc.).

Note that current versions of GRED and EDLEV are only available for DOS. Since these programs require startup parameters, the best way to run them is directly from the DOS command line or from a DOS batch file.

In this chapter we will make use of some files in the Clean Game Library ZIP file, CGL.ZIP, which can be downloaded from: http://home.wxs.nl/~mike.wiering/cgl/download.htm.

9.1 Getting started

Before we start programming, we will first set up a useful directory structure. We also have to setup the sprite editor and the level editor to be able to use them efficiently while programming the game.

9.1.1 Directory structure

The level editor EDLEV and the sprite editor GRED are both written for 8-bit modes (256 colors). Both programs use a palette file named `DEFAULT.PAL`, which contains a list of the 256 RGB colors that can be used. Though it is possible to load a new palette while editing a sprite, it is far easier to have one default palette that is automatically loaded. But we probably do want to use more than 256 colors in the entire game. Therefore, we will make a few different directories with their own `DEFAULT.PAL` file. Changes made to the color palette will affect all bitmaps in the directory.

Another reason for creating separate directories is that some bitmaps are only used as blocks in a layer and others are part of an animation sequence of a sprite. Having all these bitmaps mixed together only makes it harder to find the one we're looking for. While editing a layer, it is only possible to use sprites of one particular size. So it would also make sense to have all sprites of each size in a separate directory.

At this point, we probably haven't decided exactly what the size of all our sprites will be yet. Since we will start with the layers anyway, we will use the following directory structure for now. Later we can add directories as we create objects and levels.

We will start by creating a new directory `Charlie` and unpacking the file TEMPLATE.ZIP (which is available in CGL.ZIP) in this new directory. This will create a subdirectory `bin` which contains the programs we will need. We now have the following directory structure:

```
Charlie\
Charlie\bin
```

The highest level directory (`Charlie\`) will be the place for our source code `Charlie.icl`, all code generated by the level editor (include files) and also all the files needed to run the game (final bitmaps, music files and sound samples).

We will create some extra directories for the levels and the sprites:

```
Charlie\Objects
Charlie\Backgrounds
Charlie\Level1
```

We will use the directory `Charlie\Objects` for sprites. Later, we might want to divide these sprites into separate directories if we run out of palette colors, but for now, one directory is enough.

The directory Charlie\Backgrounds will contain bitmaps used for our backgrounds. Because we will simply use one large bitmap for the complete background layer, we can have all background bitmaps together. We will use a normal drawing program (not GRED) to create the background bitmaps.

The directory `Charlie\level1` is for all the blocks used in the front layer of the first level. These blocks all use the same palette and have the same size. Later, when we want to make more levels, we just add new directories.

The easiest way to create the directories is by using the batch (script) file `NEWDIR`, available in `TEMPLATE.ZIP`, which copies all files needed to run GRED, EDLEV and CONV. We have already unpacked this ZIP file, so we can use the batch file `NEWDIR.BAT`. We can now create our directories:

```
NEWDIR Objects
NEWDIR Level1
```

Because we will use normal drawing programs to make the background bitmaps, instead of using GRED, we will simply create the directory Backgrounds with the DOS command:

```
MKDIR Backgrounds
```

In the two directories we created with NEWDIR, we now have four files: DEFAULT.PAL (default palette file for GRED and EDLEV), GRED.BAT, EDLEV.BAT and CONV.BAT. These last three batch files will start the programs from the `BIN` directory.

9.1.2 Clean compiler setup

Before we can compile our game, we must create a project file and setup the Clean compiler. We create a new project named `Charlie.prj`. Then we have to set the link options so that the Game Library can be used. To do this, we add the following library files: ddraw_library, dsound_library and winmm_library. We will also need the following object files: cGameLib.obj, cOSGameLib.obj, ddutil.obj and dsound.obj. Because the game library is built on top of the Object I/O library, CleanIDE must be started with extra memory: `cleanide -h 9M -s 400K`, for example.

9.2 Creating layers

To create our game, we will start with the template `Untitled.icl` and name it `Charlie.icl`. The template contains the startup code and some useful definitions. Using this template will save us time and enable us to start defining levels right away!

9.2.1 Creating a background layer

We will start by creating the background layer. To do this, we will use an ordinary drawing program and create the bitmap shown in Figure 9-1. This bitmap has size 320x320 pixels, we name it `ForestBackground.bmp`.

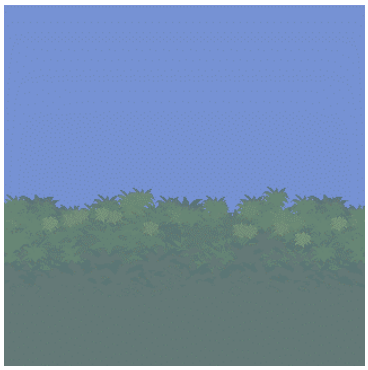

Figure 9-1: Background bitmap

We will now include this game bitmap in our game definition (Charlie.icl):

```
ForestBackground :: GameBitmap
ForestBackground
   = { bitmapname  = "ForestBackground.bmp"
     , unitsize    = { w = 320, h = 320 }
     , dimensions  = (1, 1)
     , transparent = Nothing
     }
```

As we see in this definition, this game bitmap has only one large tile. It also isn't transparent, because it will be used for the most distant layer.

Now we can define our background layer:

```
Level1Background :: Layer
Level1Background
   = { bmp       = ForestBackground
     , layermap  = [{1}]
     , sequences = []
     , movement  = defaultScrollMovement 3
     }
```

Because we only have one large tile which is repeated for the entire background, we define the layer map as [{1}]. This background layer has no tile sequences, so we specify the empty list.

We want our background layer to scroll slower than the foreground layer so it will look more distant. We can use a predefined movement function: `defaultScrollMovement`. Using this function makes a layer scroll at 1/*n* speed, where *n* is the argument. In our definition, the background layer will scroll at a third of the speed.

9.2.2 Creating the foreground layer

The foreground layer will be much more complex than the background layer we just created. We will use the level editor to create this foreground layer. We will call it Level1 (note that we are confusing *levels* with *layers* here, but because all the action takes place in this foreground layer, we are actually defining the level itself, so we call it Level1).

We have already created a directory for the first level: `Charlie\Level1`. We will now start creating our foreground. In DOS, we go to this directory and type: `edlev L1 20 16`, which starts the level editor with a new level.

We use blocks of 20 by 16 pixels here. Choosing a good block size is important, because all tiles will have this size. In some cases it can be useful to have square tiles, but here we will use rectangular ones. Some common block sizes are 16x16, 20x16, 32x24, 32x32, but larger sizes can also be used. In fact, using large blocks of size 40x40, for example, will make the game much faster than using very small blocks, like 8x8. The reason is that there is a lot more overhead when filling a screen with 8x8 pixel tiles than with 40x40 pixel tiles.

Now, we just take our time to design the beginning of a beautiful level. We draw the ground, some plants and trees… (see Figure 9-2).

Figure 9-2: Designing the foreground layer

While we are editing the level, we create the bounds right away. Most ground tiles have bounds, so we define these bounds as we create the tiles and then just copy the complete tiles, together with their bounds. We don't need any map codes yet, except one FF in the top left corner, for if we want to define a tile sequence some time.

Once we have (part of) a level, we save it and return to DOS.

The next thing to do is to convert the level to a format that can be used in our Clean specification. We do this by typing: `conv L1 Level1`. This will convert our level file L1 to a new Clean module (L1.dcl and L1.icl), in which definitions are made. The names of these definitions will start with "Level1".

Because the template file we use is designed to load a level L1 by default, we can now just run the game and it will show the level we have just created against a black background. We can scroll through our level using the arrow keys and exit by pressing **Esc**.

Of course, we want to use our beautiful ForestBackground as our background layer instead of a black screen. To achieve this, we simply change the value of `fillbackground` from `Just Black` to `Nothing` and insert our `Level1Background` at the beginning of the layers list:

```
layers = [Level1Background, Level1Layer]
```

Now we can scroll through our first level and also have a background that scrolls at a third of the speed.

9.3 Creating simple objects

Our level is still very empty, so we will now create some objects. Designing the main character right away would be a little complicated, so we will start with a very simple object: a coin that cannot move and just waits until the main character collects it.

9.3.1 Creating a static coin

We will want to put these coins somewhere in our levels, so we need a code for a static coin, which we can place in the level map. We use the first available object code (not lower than the objstart value) for this coin, which is 10.

We start by just defining these coins in our level as if we can already use them. So we run the level editor and enter the map code 10 at several places in the level. Finally, we save the level and run the conversion script.

If we run the game now, we will see no change. The game engine will find code 10 in the bound map and send a message to Clean, saying that object with object type 10 may be initialized. But of course, nothing will happen.

9.3.1.1 Creating sprites

Now, we will create sprites for the coin. We want this to be a golden rotating coin so we will have to create an animation sequence with frames that show the coin from all sides. Once the main character

grabs the coin, later, we will want it to disappear in a pretty way, with glitter. So we will need a glitter sprite as well.

We have already created a subdirectory `Charlie\Objects` in which we will draw the sprites. Although we are only interested in the bitmaps we will create, we will use the level editor to order these sprites in groups.

We start the level editor, now from the `Objects` directory: `edlev OBJ 20 16`, which will create a new level file named `OBJ`. We will use the same size block size as we did for the level. We draw our coin from all angles and then also draw the glitter (see Figure 9-3), then save this level and type: `conv OBJ Objects`. This will provide us with a game bitmap, named `ObjectsBitmap`, which contains these 12 animation frames (in the same order).

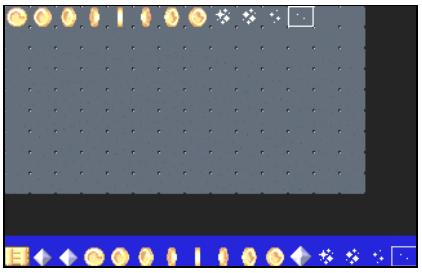

Figure 9-3: Designing animation sequences using EDLEV

As soon as we include the `OBJ` module to our definition, we can make use of our new bitmap and define our sprites. Because we don't want to have to type long sequences of numbers, like [(1,30), (2,30), (3,30), …, (8,30)] for the coin's sequence, we will create a simple function BitmapSequence:

```
BitmapSequence :: Int Int Int → Sprite
BitmapSequence start count speed
  | count == 0 = []
  | otherwise = [(start,speed)] ++
                      (BitmapSequence (start+1) (count-1) speed)
```

Now we can define the coin sprite running at a given speed as follows:

```
CoinSprite :: Int → Sprite
CoinSprite speed
   = { bitmap   = ObjectsBitmap
     , sequence = BitmapSequence 1 8 speed
     , loop     = True
     }
```

In almost the same way, we can define the glitter sprite:

```
GlitterSprite :: Int → Sprite
GlitterSprite speed
   = { bitmap   = ObjectsBitmap
     , sequence = BitmapSequence 9 4 speed
     , loop     = False
     }
```

Instead of looping like the coin sequence, the glitter sequence will run only once.

9.3.1.2 Specifying the object

Now, we will specify the static coin object in our source code. We start by defining the coin's object type (we have already chosen the code 10):

```
OBJ_STATIC_COIN :== 0x10
```

This is also a good time to define some of the bound types. We will have to be able to specify that the our main character can collide with this coin and that this coin collides with the main character. Because there will be several other objects that behave just like coins, we will give them all the name

power up. These are all the "good" items our main character can find in the level. We can now go ahead and define the following bound types:

```
BND_MAIN_CHARACTER :== (1 << 0)
BND_POWER_UP       :== (1 << 1)
```

Because of the size of an integer, we are limited to 32 different bound types (of which two are predefined: BND_STATIC_BOUNDS and BND_MAP_CODES. Because bound types will usually be used by whole groups of object types, this should be more than enough.

Whenever we want to define a new object, we can take a *defaultGameObject* and change some properties, instead of defining all properties one by one. This defaultGameObject is actually an object filled with default values for all properties (no sprites, no movement).

The properties that we will have to change are:

- Sprites
- Collide event (The coin should detect a collision with the main character, so it can change itself into glitter and eventually disappear.)
- Animation event (We will use this event to know when the glitter animation is finished.)
- The object record (We will need to define the bound types.)

Our definition becomes (note that the final code in Charlie.icl is a little different, but this is probably more readable):

```
StaticCoinObject
    # obj = defaultGameObject OBJ_STATIC_COIN size state
    # obj = { obj
            & sprites = [CoinSprite 8, GlitterSprite 25]
            , init = (newinit size state)
            , collide = newcollide
            , animation = killobject
            }
    = obj
where
    newinit size st subtype pos time gs
        # (objrec, gs) = defaultObjectRec subtype pos size time gs
        # objrec = { objrec
                   & ownbounds     = BND_POWER_UP
                   , collidebounds = BND_MAIN_CHARACTER
                   }
        = ((st, objrec), gs)

    newcollide (st, or) bnds objtype objrec gs
        # or = { or
               & currentsprite         = 2
               , options.removemapcode = True
               , ownbounds             = 0
               , collidebounds         = 0
               }
        = ((st, or), gs)

    size = {w = 20, h = 16}
    state = NoState
```

In addition to the properties mentioned above, we also had to redefine `init`, in order to initialize values in the object record.

Our sprite list contains two different sprites. The value of `currentsprite` is set to 1 by default, so we will see the rotating coin at first. When the coin collides with the main character, the `currentsprite` value is set to 2 (the glitter sprite). Because this sprite does not loop, the `animation` event will occur when the animation sequence has finished. This event has been set to `killobject`, which has the following definition:

```
killobject (st, or) gs
    = ((st, {or & active = False}), gs)
```

By setting the `ownbounds` and `collidebounds` fields to 0, we make sure that nothing happens when the main character touches the glitter when the coin is already gone. We also set `options.removemapcode` to True, so the coin will not reappear after the main character has collected it.

The last thing we still have to do before we see the coins in our level, is add our new StaticCoinObject to the list of objects: GameObjectList (which is used in the definition of Level1).

9.3.2 Creating a falling coin

In the previous section we created an object that could not move and just waits until the main character grabs it. Now we will make another type of coin object, but this one falls down and bounces on the ground.

Again, we will start by choosing a code for the object: 11, define some of these objects in our level (far above the ground, so we can see them fall) and define the object type:

```
OBJ_FALLING_COIN :== 0x11
```

Because the falling coin looks very much like the static coin, we will start with a static coin and then change it into a falling coin:

```
FallingCoinObject
    = {StaticCoinObject & init = (newinit size state)}
where
    newinit size st subtype pos time gs
        # (objrec, gs) = defaultObjectRec subtype pos size time gs
        # objrec =
            { objrec
            & acceleration   = {rx = 0.0, ry = 1.0 / 16.0}
            , bounce         = { fvx = Value 0.0
                               , fvy = Factor (4.0 / 5.0)
                               }
            , bouncebounds   = BND_STATIC_BOUNDS
            , forgetdistance = {x = 1, y = 8}
            }
        = ((st, objrec), gs)

    size = {w = 20, h = 16}
    state = NoState
```

The new definition for `acceleration`, will make the object fall. The `bounce` factor will make the coin bounce up with 4/5 of the speed it falls down, so it will eventually stop bouncing. By setting the value of `bouncebounds` to BND_STATIC_BOUNDS, the coin will bounce against the bounds defined in the bound map. Because the coin can now fall down, we make the (vertical) `forgetdistance` a little larger. This will prevent the coin from disappearing right away if we scroll the screen a little.

Now we have created our second type of object. We only have to include it in the `GameObjectList`. In the same way we can create static diamonds and falling diamonds.

As we see here, we can reuse parts of objects we have created earlier and keep creating more complex objects.

9.3.3 The background clouds

To make our level even more interesting, we will create some clouds in the background that do not scroll at all. This will make the game have three layers instead of two. But instead of creating a complete new layer for these three little clouds, which would make the game slower, we will use objects for the clouds.

Unlike most other objects, these clouds must be drawn behind our foreground layer and not in front of it. Actually, the clouds should be behind the background layer as well, but we wouldn't see them if they were, because our background layer is not transparent. So we will just put them in front of the sky (which has only one color anyway) and behind the front layer.

We will start by defining names for our layers:

```
LYR_BACKGROUND  :==  1
LYR_FOREGROUND  :==  2
LYR_PLAYER      :==  3
LYR_INFRONT     :==  4

LYR_STATUS      :== 10
```

Using these layer definitions, we can place objects at different heights in our level. We will usually want our main character in front of everything (`LYR_PLAYER`), but in some cases, an object can be in front of the main character (`LYR_INFRONT`). For example, parts of the palm trees will be drawn in front of Charlie to make it look like Charlie is *in* the tree and not in front of it.

Later, we will add a status line at the top of the screen with the statistics. This will always our highest layer: `LYR_STATUS`.

The objects we have created before (the coins and diamonds) should be in the foreground, so we add the following line to these objects:

```
layer = AtLayer LYR_FOREGROUND
```

9.3.3.1 Using the AutoInit object

Because our clouds do not belong to one position in the map, we don't want to define them there. Instead, we will use the AutoInit object to initialize our clouds:

```
OBJ_AUTOINIT :== 0

AutoInitObjectLevel1
    # obj = defaultGameObject OBJ_AUTOINIT size state
    = {obj & init = (newinit size state)}
where
    size = {w = 1, h = 1}
    state = NoState

    newinit size st subtype pos time gs
        (_, gs) = CreateNewGameObject OBJ_CLOUD 0 {x =  36, y = 50}
        (_, gs) = CreateNewGameObject OBJ_CLOUD 0 {x =  88, y = 28}
        (_, gs) = CreateNewGameObject OBJ_CLOUD 0 {x = 240, y = 43}
        (objrec, gs) = defaultObjectRec subtype pos size time gs
        = ((st, {objrec & active = False}), gs)
```

The AutoInit object is only needed to start the clouds, so `active` is set to False at the end of the `newinit` function.

Each level can have it's own AutoInit object, we call this one `AutoInitObjectLevel1`. In our level definition for level 1, we will then define our objects:

```
objects  = [AutoInitObjectLevel1] ++ GameObjectList
```

This way, we can have a different AutoInit object for each level. Of course, we don’t add this AutoInit object to the GameObjectList as well.

9.3.3.2 Defining the clouds

We still have to define the clouds themselves. To do this, we create a new set of sprites using the level editor, now with large blocks: 40x24 pixels. We call our level CLOUDS and convert it with the command: `conv CLOUDS Clouds`.

Back in our definition, we define a sprite for our clouds: `CloudSprite`, and specify a cloud:

```
CloudObject
    # obj = defaultGameObject OBJ_CLOUD size state
    # obj = { obj
            & sprites = [CloudSprite]
            , init    = (newinit size state)
```

```
            }
    = obj
where
    size = {w = 40, h = 24}
    state = NoState

    newinit size st _ pos time gs
        # (objrec, gs) = defaultObjectRec subtype pos size time gs
        # objrec = { objrec
                   & options.static = True
                   , layer          = AtLayer LYR_BACKGROUND
                   }
        = ((st, objrec), gs)
```

By setting `options.static` to True, the cloud will not move with the layer, but remain at the same place on the screen all the time. The definition of `layer` sets cloud in the background layer.

In a rather similar way, we use objects in front of all the palm trees, which are drawn in front of our main character (`LYR_INFRONT`). These objects are not static though.

9.3.4 Crates with items

Throughout our level, we will have crates containing items for our main character (coins, diamonds, hearts, lives, etc.). These crates break open when Charlie jumps on top of them and the item(s) appear (see Figure 9-4).

Figure 9-4: Charlie opens a crate

As we see in the figure, the crate breaks into six small blocks which then fall off the screen.

First of all, we add the following definitions to `Charlie.icl`:

```
BND_BLOCKS :== (1 << 2)
...
OBJ_INVISIBLE_CRATE :==  0xB0
OBJ_CRATE           :==  0xC0
...
OBJ_CRATE_PART      :== 0x100
```

We name the bound type `BND_BLOCKS` instead of `BND_CRATE`, because we will make more objects with the same collision behavior. We don't want any objects to go through a crate so they must bounce against it. The only interesting collisions are those with the main character. This is the same with a bouncing block, which we will define later, so both will have `BND_BLOCKS` as bound type.

In addition to the normal crates, we also have invisible crates. These are exactly the same, but they are not visible in our level, so the player will find them by coincidence.

For objects we create during the game, we will use higher object types (0x100 and higher). Such codes cannot be entered in the level map.

```
CrateObject          = Crate OBJ_CRATE           True
InvisibleCrateObject = Crate OBJ_INVISIBLE_CRATE False

CrateObject objecttype visible
    # obj = defaultGameObject objecttype size state
```

```
        = { obj
          & sprites = [CrateSprite]
          , init    = (newinit size state)
          , collide = newcollide
          }
    where
        size = {w = 20, h = 16}
        state = NoState

        newinit size st subtype pos time gs
            # (objrec, gs) = defaultObjectRec subtype pos size time gs
            # objrec = { objrec
                       & layer         = AtLayer LYR_FOREGROUND
                       , ownbounds     = if visible
                                            (BND_BLOCKS + BND_STATIC_BOUNDS)
                                             BND_BLOCKS
                       , collidebounds = BND_MAIN_CHARACTER
                       , currentsprite = if visible 1 0
                       }
            = ((st, objrec), gs)
```

As we see here, visible crates have `BND_STATIC_BOUNDS` as one of their bound types. This makes other objects bounce and collide with these crates as they would with a wall or a platform defined in the level map. Invisible crates do not have this bound type, so objects can go through them.

The collide function is the most interesting, of course:

```
        newcollide (st, or) bnds othertype otherobjrec gs
            | ((othertype == OBJ_MAIN_CHAR) && (bnds.bottom))
                # pos1 = or.pos
                # pos2 = {pos1 & y = pos1.y + 8}
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 1 pos1 gs
                # pos1 = {pos1 & x = pos1.x + 4}
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 2 pos1 gs
                # pos1 = {pos1 & x = pos1.x + 4}
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 3 pos1 gs
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 4 pos2 gs
                # pos2 = {pos2 & x = pos2.x + 4}
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 5 pos2 gs
                # pos2 = {pos2 & x = pos2.x + 4}
                # (_, gs) = CreateNewGameObject OBJ_CRATE_PART 6 pos2 gs
                # or = {or & options.removemapcode = True, active = False}
                # obj = case or.subtype of
                    0 -> if visible OBJ_FALLING_COIN OBJ_STATIC_COIN
                    1 -> if visible OBJ_FALLING_DIAMOND OBJ_STATIC_DIAMOND
                    2 -> OBJ_HEART
                    3 -> OBJ_LIFE
                # (_, gs) = CreateNewGameObject obj 0 or.pos gs
                = ((st, or), gs)
            = ((st, or), gs)
```

Here we see how the create breaks into six parts, which are all separate objects. Finally, the item in the crate is created according to the crate's subtype. The crate object itself is destroyed. When we define a crate in our level map, we insert a C0 (or B0 for invisible) and above it, a 0, 1, 2 or 3 for the different kinds of items. Whenever a crate has another crate on top of it, it can only contain a coin (subtype 0).

The CratePart object is not very interesting. According to it's subtype, the speed is set to a standard value and then a random value is added to make the crates open a little differently each time. The object is at the front layer (`LYR_INFRONT`) and is destroyed as soon as it falls off the screen.

9.3.5 Creating enemies

Next, we will create some enemies in our level. These enemies just walk in a straight line and turn around when something is in their way. If our main character jumps on top of such an enemy it will die and fall off the screen. At some places in the map, we will put map codes to make this enemy turn around and not fall off a platform, as shown in Figure 5-2 (it wouldn't fall actually, it would just keep walking in the air with no ground beneath it). We will indicate this enemy on the map by code 80.

We start by drawing sprites for the enemy and adding the following definitions to our `Charlie.icl`:

```
OBJ_ENEMY :== 0x80
...
BND_ENEMY :== (1 << 3)
```

We will only look at the newinit function of the enemy, the rest is very much the same as previous objects.

```
newinit size state subtype pos time gs
    # (objrec, gs) = defaultObjectRec subtype pos size time gs
    # objrec = { objrec
               & speed          = {rx = ~0.5, ry = 0.0}
               , bounce         = {fvx = Factor 1.0, fvy = Value 0.0}
               , layer          = AtLayer LYR_FOREGROUND
               , options        = { objrec.options
                                  & automirrorleftright = True
                                  }
               , ownbounds      = BND_ENEMY
               , bouncebounds   = BND_STATIC_BOUNDS + BND_ENEMY +
                                    BND_MAP_CODES
               , collidebounds  = BND_MAIN_CHARACTER
               , forgetdistance = {x = 6, y = 4}
               }
    = ((state, objrec), gs)
```

This enemy walks half a pixel per frame, so it will move once every two frames. Whenever it touches a wall (vertical bound), it will turn around and continue with the same speed. Because of the `automirrorleftright` value, the sprite will be mirrored after it turns around. The `bouncebounds` value includes map codes, so the enemy will turn around when it finds a code in the map.

The only real collisions are with the main character. The collide function checks if the colliding object is a main character coming from the top (in the same way the crate does). If so, the enemy kills itself.

9.3.6 Creating the bees

The next enemy in our game will be a flying bee. We will make it fly randomly, by adding small random values to it's horizontal and vertical speed a little once in a while. We will do this with the `move` event. This event can be generated every frame if we want it, but we will skip a random number of frames between each `move` event, so we will make a new move function:

```
...
move = newmove
...
```

Again, most part of the bee's code looks a lot like other objects, so we will only look at the most interesting part, the movement.

First of all, we will need a few random functions: one for integer values and one for very small real values:

```
/* get random integer value 0..n */
IRnd n gs = (Rand rem n, gs)

/* get random Real value ~n..n */
RRnd n gs =
    (n * (((toReal Rand) / max) - ((toReal Rand) / max)), gs)
where
    max = (toReal MaxRand)
```

In the bee's `newinit` function, we set its initial `speed` to `{rx = ~0.5, ry = 0.0}` and the `skipmove` value to 0. This will make the `move` event occur as soon as possible.

Our new move event looks like this:

```
newmove (st, or) gs
    # (turn, gs) = IRnd 30 gs
    # (xadd, gs) = RRnd 0.05 gs
    # (yadd, gs) = RRnd 0.085 gs
    # (skmv, gs) = IRnd 25 gs
    # rxv = (if (turn == 1) (~ or.speed.rx) (or.speed.rx)) + xadd
    # ryv = or.speed.ry + yadd + 0.005
    # or = {or & skipmove = skmv, speed = {rx = rxv, ry = ryv}}
    = ((st, or), gs)
```

To change the movement, we take two small random numbers and add them to the speed. This bee will turn around about once in every 30 times. By setting `skipmove` to a random value, the bee will move with the same speed for some time and not be too unpredictable.

9.3.7 Creating frogs

The frogs are actually very easy to make now, they behave just like normal enemies, but they jump instead of walk. We create gravity with the object's acceleration value and then let the object bounce. The horizontal bounce value is `Factor 1.0`, which makes the object turn around with the same speed. The vertical bounce value is `Value 2.0`. This means that the falling frog will start moving in the opposite direction with speed 2.0. This will make the frog always jump with the same strength.

```
objrec = { objrec
         & speed        = {rx = ~1.0, ry = 0.0}
         , acceleration = {rx = 0.0, ry = 1.0 / 16.0}
         , bounce       = {fvx = Factor 1.0, fvy = Value 2.0}
         ...
         }
```

9.3.8 Other objects

Now we have seen several types of object, we will only take a quick look at the other objects in our game, before we look at our main character in the next section.

We have some more power up items: hearts and lives. These work just like the falling coins.

There is also water with moving waves. These waves are objects, which are placed at the `LYR_INFRONT` layer. When our main character falls into the water, we will see the water splash. This is also done with objects, which only show an animation sequence once and then stop existing.

Another object that exists for only a short time is the flash object. We see this flash every time our main character kills an enemy by landing on top of it.

We also have bounce blocks that behave like ordinary blocks to all creatures except the main character, which will be able to jump very high from such a bounce block.

At the end of our level, we have an object that Charlie must touch in order to complete the level, the EndingOjbect. This object rotates for a while after Charlie has touched it and then it stops.

In our second level we have a few more objects: the sharp pins, which kill our main character right away, bounce blocks that move (these are exactly the same as normal bounce blocks, only their speed is different). There are also some wall objects which make it possible for our main character to go behind a wall. These work in the same way as the objects in front of the palm trees.

Finally, we also have a few objects for our status line (see Figure 9-5), the diamond, the coin some hearts and the little 'x' and ':', which are also objects.

Figure 9-5: The status line

We use these objects together with the statistics to make the status line more interesting. All these objects are placed at LYR_STATUS and are static. They are created by the AutoInit object, just like the clouds are.

One last object remains: the menu pointer on the title screen. This is an AutoInit object itself and it is one of the few objects in this game that have an object state different from NoState. This object checks the keys that are pressed, changes its `offset` value to move the visible pointer and ends the level (the title screen) when a choice has been made.

9.4 Creating our main character

Now, we will define our main character, Charlie. We will have to create a much more complex object than the ones we have created so far. At any time, Charlie can be walking, jumping, falling, swimming or just be standing still. All these actions have their own types of movement and their own sprites.

9.4.1 Main character actions

We will start by defining all the actions.

```
MC_IDLE  :==  1

MC_WALK  :==  2

MC_JUMP  :==  3
MC_FALL  :==  4
MC_SWIM  :==  5
MC_DEAD  :==  6
```

We will use the main character's object state to store the current action. In our event functions, we will often have a case structure depending on this current action.

We define a sprite for each action using the bitmaps in Figure 9-6.

Figure 9-6: Main character bitmaps

The first six bitmaps correspond with the defined actions. We use the last two to make our main character open its mouth if the player does nothing for a long time (to get the player's attention back to the game). We define six sprites: CharlieIdleSprite, CharlieWalkSprite, CharlieJumpSprite, CharlieFallSprite, CharlieSwimSprite and CharlieDeadSprite.

In the definition of our main character, the order of the sprites corresponds with the action constants defined above, so we can use these constants with `currentsprite` as well.

```
sprites = [CharlieIdleSprite, CharlieWalkSprite, CharlieJumpSprite,
           CharlieFallSprite, CharlieSwimSprite, CharlieDeadSprite]
```

9.4.2 Object state

We use the object state to store information about the main character that we cannot store in the object record. We have already seen the first element: the current action.

To determine the action at a certain point in the game, it's useful to have information about the previous position of the main character. For example, if the current action is `MC_WALK`, but the current position is exactly equal to the previous position, we know that our character is not walking anymore, so we set the action back to `MC_IDLE`. In the same way, we can find out that the character has reached the ground after a fall. However, remembering only one previous position is not enough. If our character makes a jump, it will stop for one frame at the highest point before starting to fall again. We wouldn't want it to become idle at that point. For this reason we will keep track of the two last positions of our main character.

Charlie's health is also stored in the object state. At the beginning of a level, there are three hearts. Every time Charlie is hurt by an enemy, one of these hearts disappear. When no hearts are left, Charlie loses a life.

During the game Charlie can jump on top of enemies to destroy them. The player receives points for destroying enemies. We will make the points double each time Charlie jumps from one enemy straight to the next. For example, we get 100 points for the first, 200 for the next, then 400, 800, etc. However, Charlie may not touch the ground in between. We will also play a sound that becomes higher each time. This is also something to store in the object state, the note to play next.

We can now define the object state:

```
:: MainCharState
   = { action     :: Int
     , lastspeed1 :: RealXY
     , lastspeed2 :: RealXY
     , enemynote  :: Int
     , health     :: Int
     }
```

At the beginning of a level, the main character's state looks like this:

```
initstate = { action     = MC_IDLE
            , lastspeed1 = zero
            , lastspeed2 = zero
            , enemynote  = 0
            , health     = 3
            }
```

9.4.3 Initialization

The `init` function of the main character has some interesting differences from that of previous objects. First of all, the main character controls the scrolling through the level:

```
# (_, gs) = CreateObjectFocus
     { scrollleft      = 132, scrollup        = 50
     , scrollright     = 132, scrolldown      = 52
     , maxxscrollspeed =   2, maxyscrollspeed =  3 } gs
```

Since we use a screen of 320x240 pixels in this game, our main character will have an area of 56x138 pixels at the center of the screen in which it can freely move, without causing the level to scroll.

Because the main character will be controlled by the player, `options.checkkeyboard` is set to True and `options.allowkeyboardrepeat` to False (we only want to know when a key is pressed and when it is released again).

The main character collides with just about all bound types that we have defined.

```
…
, ownbounds      = BND_MAIN_CHARACTER
, bouncebounds   = BND_STATIC_BOUNDS
, collidebounds  = BND_ENEMY + BND_WATER + BND_ENDING +
                   BND_POWER_UP + BND_BLOCKS + BND_KILL
…
```

9.4.4 Movement control

An important part of the movement is defined by values in the object record. The gravity is defined by: `acceleration = {rx = 0.0, ry = 1.0 / 8.0}`. This will make our main character automatically fall down as soon as there is no solid ground beneath it. Whenever the character walks, it is always slowed down by: `slowdown = {fvx = Factor 1.0 / 16.0, fvy = Value 0.0}`. The maximum (horizontal) speed depends on whether Charlie is walking on the ground or swimming in the water. So we change the value of `maxspeed` whenever Charlie falls into the water or jumps out of it.

The way we can control Charlie's movement is defined in the `keydown` and `keyup` event code. There are three keys that we check: the ← and → arrow keys and **Space**.

```
newkeydown (st=:{action}, or) key gs
    | key == GK_LEFT
        # newaction = if (action == MC_IDLE) MC_WALK action
        = (({st & action = newaction},
           {or & acceleration.rx = or.acceleration.rx - ac,
                 currentsprite = newaction}), gs)
    | key == GK_RIGHT
        # newaction = if (action == MC_IDLE) MC_WALK action
        = (({st & action = newaction},
           {or & acceleration.rx = or.acceleration.rx + ac,
                 currentsprite = newaction}), gs)
    | key == GK_SPACE
        | (isMember action [MC_IDLE, MC_WALK, MC_SWIM])
            # act = action
            # ((st, or), gs) = (({st & action = MC_JUMP},
                {or & speed = (jumpspeed or.speed)
                    , currentsprite = MC_JUMP
                    , offset = normaloffset
                    , maxspeed = maxwalkspeed}), gs)
            | (act == MC_SWIM)
                # (_,gs) = Splash {x = or.pos.x, y = or.pos.y + 32} gs
                = ((st, or), gs)
            = ((st, or), gs)
        = ((st, or), gs)
    | otherwise = ((st, or), gs)
where
    jumpspeed sp=:{rx,ry} = {rx = rx, ry = ry - 4.35 - abs (rx) / 3.0}

newkeyup (st, or) key gs
    | key == GK_LEFT
        = ((st, {or & acceleration.rx = or.acceleration.rx + ac}), gs)
    | key == GK_RIGHT
        = ((st, {or & acceleration.rx = or.acceleration.rx - ac}), gs)
    | otherwise = ((st, or), gs)
```

What we actually do to make Charlie walk left or right is change the value of `acceleration`. The horizontal acceleration can only have three values: *-ac*, 0 and *ac* (*ac* is defined as 1.0 / 5.0), because `options.allowkeyboardrepeat` is False. As long as we keep the left or right arrow key pressed, the `acceleration` value remains the same and Charlie's `speed` increases, until it reaches the `maxspeed` value. When we release the key, the acceleration stops, and the main character will gradually slow down because of the `slowdown` value.

Jumping is done by setting the vertical speed to a negative value, calculated by `jumpspeed`. The gravity will eventually make Charlie fall down again. Such a jump may only start from the ground, so the current action must be `MC_IDLE`, `MC_WALK` or `MC_SWIM`. Jumping out of the water will cause a splash.

The current action is mainly controlled by the `animation` event. By using sprites that have short sequences and do not loop, we make this event occur regularly. Using this `animation` event, we keep checking if the current action should be changed into another action. For example, if the main character is jumping, we will check if it has reached its highest point and has started to fall. When it is falling, we check if it has reached the ground and should be walking again. If it is walking, we check if it is not moving anymore and should be idle, and so on. This is all done by the animation event.

```
newanimation (st=:{action = act}, or=:{offset = ofs}) gs
    # xstuck = ((or.speed.rx == st.lastspeed1.rx) &&
      (or.speed.rx == st.lastspeed2.rx) && ((toInt or.speed.rx) == 0))
    # ystuck = ((or.speed.ry == st.lastspeed1.ry) &&
      (or.speed.ry == st.lastspeed2.ry) && ((toInt or.speed.ry) == 0))
    # oldact = act
    # (act, ofs) = case act of
        MC_WALK   -> if xstuck
                        ((if (or.speed.ry > sp) MC_FALL MC_IDLE), ofs)
```

```
                        ((if (or.speed.ry > sp) MC_FALL MC_WALK), ofs)
            MC_JUMP   -> (if (or.speed.ry > sp) MC_FALL
                           (if (xstuck && ystuck) MC_IDLE MC_JUMP), ofs)
            MC_FALL   -> (if (or.speed.ry > sp) MC_FALL MC_WALK, ofs)
            MC_SWIM   -> (MC_SWIM,
                             {ofs & y = if (ofs.y == 4) 3 (ofs.y + 1)})
            otherwise -> (MC_IDLE, ofs)
        # st = {st & lastspeed2 = st.lastspeed1}
        # st = {st & lastspeed1 = or.speed}
        = (({st & action = act},
                    {or & currentsprite = act, offset = ofs}), gs)
    where
        sp = 1.0 / 4.0
```

The `offset` is used to make Charlie go up and down all the time in the water.

9.4.5 Collisions

The main character's `collide` function is very large. It contains separate parts, which handle collisions with the various kinds of objects in the level. It has the following structure:

```
newcollide (st, or) bnds othertype otherobjrec gs
  |  (othertype == OBJ_WATER)
     = ...
  |  (othertype == OBJ_PIN)
     = ...
  |  (othertype == OBJ_BOUNCE_BLOCK)
     = ...
  |  (othertype == OBJ_ENDING)
     = ...
  |  (othertype == OBJ_HEART)
     = ...
  |  bnds.top // main character lands on top of other object
     | (not (otherobjrec.ownbounds bitand BND_BLOCKS == 0))
        = ...  // other object is a crate
     | (not (otherobjrec.ownbounds bitand BND_ENEMY == 0))
        = ...  // other object is an enemy
     = ((st, or), gs)
  |  (not (otherobjrec.ownbounds bitand BND_ENEMY == 0))
     = ...     // main character is hurt by enemy
  =  ((st, or), gs)
```

We first check for all the special objects by their object type and then handle the rest together, looking at their bound types. Because objects can have several bound types, we cannot simply compare these bounds with `BND_ENEMY` or `BND_BLOCK`. Instead, we have to use the `bitand` operator.

Collisions with all the kinds of enemies are handled in the same way. The main character either lands on top of the enemy, killing it, or touches the enemy from a side or from the bottom and gets hurt.

Note that the heart (`OBJ_HEART`) is the only kind of power up item the main character collides with. The others (coins, diamonds and lives) do collide with the main character and handle these collisions by themselves. Information about the number of coins, diamonds and lives is kept in the game state, because this information has to do with the entire game, not only one level. The number of hearts only belongs to a single level and is stored in the main character's object state.

If the main character gets hurt and doesn't have anymore hearts, it dies. We set the current action to `MC_DEAD` and let it slowly fall off the screen. We also create two events, using the function `CreateUserGameEvent`. The first, `EV_STOP_MOVING` is to make the dead main character stop falling before being forgotten by the game engine. The second, `EV_QUIT_LEVEL` is to quit the level. If there are no lives left, there will also be a third event: `EV_GAME_OVER`, which will cause the update the game state.

There is one more user event: `EV_STOP_BLINKING`. This is set when the main character gets hurt but does have at least one heart. After a such a collision, we give the player some time to get out of the

harmful situation and make the main character blink for a while. During this time, it will not be hurt by other objects. We use the `EV_STOP_BLINKING` event to know when to stop this blinking.

As mentioned before, we play a sound that becomes higher every time the main character jumps from one enemy to another. We do this by playing the sound at a higher frequency each time. We use the `getnotefreq` function to calculate the frequency of the note we want.

```
# (_, gs) = PlaySoundSample SND_HIT HighVolume PAN_CENTER
      (getnotefreq (MIDDLE_C + 76 + 2 * st.enemynote)) 0 gs
# st = {st & enemynote = st.enemynote + 1}
```

9.5 Finishing the game

After defining our main character, our level has finally become playable. To make our first level complete, we will create the statistics, which are shown at the status line. After that, we will define the flow of the game and finally, we can start creating the other levels and finish the game.

We also create a title screen, which is just like a (very small) level. This level has two objects: the menu pointer object and the main character (but without its `keydown` and `keyup` functions). We use statistics for the text on the title screen.

Figure 9-7: The title screen

9.5.1 Adding statistics

In Figure 9-5, we have seen the status line. From left to right it shows the number of lives, diamonds, coins, hearts and the player's score. Most of this information comes from the game state, except the player's health. This is shown entirely with objects, as discussed earlier.

We will first look at the game state:

```
:: GameState
   = { curlevel    :: Int
     , maxlevel    :: Int
     , titlescreen :: Bool
     , statusline  :: Bool
     , exitcode    :: Int
     , lives       :: Int
     , coins       :: Int
     , diamonds    :: Int
     , score       :: Int
     , quit        :: Bool
     , gameover    :: Bool
     }
```

Because the statistics function belongs to the entire game and not to a single level, we will use `statusline` to indicate whether or not the status line should be shown. The title screen does not have a status line, but the normal levels do. We will do the same for the "GAME OVER" text. This should only be displayed when `gameover` is set to True. In the same way, `titlescreen` indicates when text should be written for the title screen.

The result of the `Statistics` function depends on the values of `titlescreen`, `statusline` and `gameover`:

```
Statistics :: GameState -> ([Statistic], GameState)
Statistics gst
    | gst.titlescreen
        = ([ TitleTextShadow, TitleText
           , DemoText
           , Copyright
           , MenuText 0 "Start"
           , MenuText 1 "Exit"
           ], gst)
    | gst.statusline
        = ([ Lives    gst.lives
           , Diamonds gst.diamonds
           , Coins    gst.coins
           , Score    gst.score
           ] ++ (if gst.gameover [GameOver] []), gst)
    = ([], gst)
```

The lives, diamonds, coins and score in the status line are all defined separately as `Int` → `Statistic` functions, for example:

```
Score :: Int -> Statistic
Score n
  = { format    = "Score  %07d"
    , value     = Just n
    , position  = {x = ~10, y = 5}
    , style     = StatStyle
    , color     = White
    , shadow    = Just StatShadow
    , alignment = zero
    }
```

9.5.2 Flow of the game

Every level sets an exit code (`exitcode` in the game state), indicating that the player has managed to complete the level, or that the player was killed, or that the player has simply quit the level. The `exitcode` can have the following values: `EC_NONE`, `EC_QUIT`, `EC_SUCCESS` and `EC_FAILURE`.

Any object can change the exitcode, using the following function:

```
setexitcode newcode gs = appGSt (setgstexitcode newcode) gs
where
    setgstexitcode :: Int GameState → GameState
    setgstexitcode c gst = {gst & exitcode = c}
```

In the initial game state, exitcode is set to `EC_NONE`. The AutoInit object sets it to `EC_QUIT` as soon as a level starts. If our main character looses a life and has to restart the current level, we set `exitcode` to `EC_FAILED` and if our character does reach the end of the level, we set it to `EC_SUCCESS`.

We define the flow of our game by writing a `nextlevel` function, which is part of the game definition. In our case, this function will look at `exitcode` to determine what the next level is.

```
NextLevelFunction :: GameState → (Int, GameState)
NextLevelFunction gst =: {curlevel, maxlevel, titlescreen,
                          exitcode, lives, gameover}
    | exitcode == EC_QUIT
        = (0, gst)
    | titlescreen
        = (next, {gst & titlescreen = False
                      , statusline = True
                      , lives = DEFAULT_LIVES
                      , curlevel = next})
    | exitcode == EC_FAILURE
         | lives > 0
```

```
            = (curlevel, {gst & lives = lives - 1})
        = title
    | exitcode == EC_SUCCESS
        = nextlevel
    = title
where
    title = (1, {gst & titlescreen = True
                     , statusline = False
                     , gameover = False
                     , curlevel = 1})
    nextlevel = if (curlevel + 1 > maxlevel)
                   title (next, {gst & curlevel = next})
    next = curlevel + 1
```

Chapter 10
Conclusion

After implementing the complete game library, we can now look at the results. In the previous chapter, we have already seen that it is possible to create a complete platform game using a Clean specification. But can we create games faster or more easily using this library? Could we use the library for other types of games (other than platform games)? Is the performance of the games we create sufficient to make them playable?

10.1 Rapid Game Development

It took two and a half weeks to create the platform game *Charlie the Duck* (described in Chapter 9) using the new library (about 1,950 lines of code). Writing the original version in Pascal (35,000 lines) took more than six months in 1996. But making such a comparison is actually not fair. The Pascal version includes all the low-level code and was made without using any kind of level editor. Besides, it is a complete game with twelve levels, not just a two-level demo. However, this example should give an idea of how easy it is to use the Clean library together with the tools. To make a good comparison between Clean and another language, we would have to implement a very similar library for this other language (also using the same tools) and then write exactly the same game in both languages.

Using the level editor makes designing the layers and positioning the objects very easy. However, because levels are converted to Clean code, the game must be recompiled every time we change a level. Compiling a game in Clean takes a long time.

10.2 Possibilities

The library was designed for platform games, but also turns out to be very suitable for actually any type of two-dimensional games. The game *Worms* (see Figure 10-1) shows an example of a non-platform game created with the library. Here, the game has only one layer, which cannot scroll and there is no gravity.

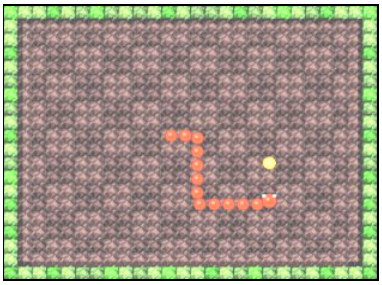

Figure 10-1: Worms

In the same way, we could for example make a racing game (viewed from the top) or a game such as *Pac-Man*. Two-player games can also be made, such as a fighting game.

The library is probably also suitable for most board games, but since there is no mouse pointer, these may be a little difficult to control.

The game library is not designed for 3D games, because these require a complete different set of functions. However, it is of course possible to create drawings of 3D movement and insert these in a game. We could render a rotating cube and create an animation sequence that displays this cube.

In the current version of the library, it is not possible to use I/O functions while a level is running, because the *iostate* is not available throughout the library. However, we can make games with high scores etc., by doing file access before and after the actual game runs. This problem can probably be solved in future versions, making it possible for any game object to perform I/O functions.

10.3 Performance

The performance of the game library is acceptable on most PC's, however there are still problems with certain configurations, on which the games are really slow. This can probably be solved though, because other (similar) demos using the same DirectX functions do run fast on these PC's.

If we make another (unfair) comparison between both versions of *Charlie the Duck* and look at the system requirements, we see great differences. The original version requires only a 486 computer at 66 MHz to run at full speed. The new version requires a computer with a Pentium processor and a fast video card to run properly. This difference is caused in the first place by the use of DirectX, but also has to do with the countless optimizations in the original version.

The code that is produced using a universal library is generally not optimal for a particular game. The produced executables are also very large in comparison with similar games written in other languages. The interpreter function can run any platform game, but will not look for the most efficient way to do this. For example, the *Worms* game (see §10.2) has about the same system requirements as *Charlie the Duck*, even though it doesn't scroll, and there is only one layer with a small worm walking around. We could easily write a much more efficient version of this game.

However, the system requirements of the games produced using the library are not any higher than those of most games written today.

To measure whether the introduction of *skipmove* (see §5.6.4.12) has turned out to be useful for performance optimization, we can create a simple demo of a small level with random bouncing balls (see Figure 10-2) in two versions. In the first version, the *ball* object has an object state containing the current speed and all movement is handled by the ball's *move* event. In the second version, *skipmove* is set to -1, so the *move* event never occurs and the game engine handles the movement entirely. We can then run the demo with a number of balls and scroll through the level. We are interested in the maximum number of balls with which the demo can scroll smoothly without jerking at all.

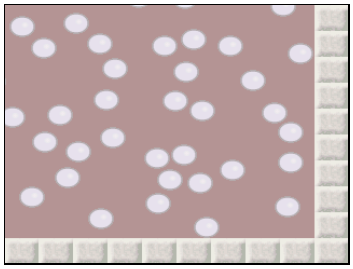

Figure 10-2: Speed test

The first version (*move* event controls objects) runs at full speed (100 frames per second) with up to 28 balls[3]. If we add more balls, the scrolling isn't smooth anymore. The second version (without the *move* event) scrolls completely smoothly with up to 195 balls. From this example, we can conclude that using the move event should be avoided whenever possible to improve performance. The delay in the first version is probably mainly caused by the cross-call overhead (switching between threads).

10.4 Overall conclusion

Using Clean together with the game library provides a new way to actually design games instead of really program them. Using the library together with some tools, we can create high quality platform

[3] Tested on a Pentium II-300 MHz

games in only a fraction of the time needed to program them from scratch. The library is also very useful for most kinds of (two-dimensional) games, not only for platform games.

The produced games are generally larger in size and less efficient than they would be if the game had been written from scratch, but the system requirements are not higher than those of most new games. There are still some performance problems on some PC configurations which make the games very slow, but this can probably be solved.

By avoiding continuous switching between the game engine, written in C and functions in the Clean specification, the performance can be improved.

Distributing final games is still a little awkward, because Clean does not support resources in the executables. All files needed for the game (bitmaps, sound samples and music files) have to be copied along with the game to make it work. The player can also look at (or listen to) these files without playing the game.

Porting the library to other platforms should not be too difficult. This might eventually result in an opportunity to create games that run on several platforms.

References

[1] Plasmeijer, M.J. and van Eekelen, M.C.J.D. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishing Company, 1993.

[2] Achten, P.M. and Wierich, M. *A Tutorial to the Clean Object I/O Library - version 1.1. Internal Report*, Department of Functional Programming, University of Nijmegen, The Netherlands, 1999 (ftp://ftp.cs.kun.nl/pub/Clean/supported/ObjectIO/doc/tutorial.11.ps.gz).

[3] Gruber, D. *Action Arcade Adventure Set*. Coriolis Group Books, 1994 (http://www.fastgraph.com/aaas.html).

[4] Gruber, D. *Modeling Sprite Animation Using Finite State Automata*, published as: "Automata Animation", PC Techniques, Vol. 6, No. 1, 1995.

[5] Game Creation Programs for Non-Programmers: http://www.mindspring.com/~ambrosine/resource.html.

[6] Recreational Software Designs, *Game Maker*, Review by Power Unlimited, VNU Electronic Leisure Publishing, March 1994

[7] Europress, *Klik & Play*: http://www.europress.co.uk/products/kp.html.

[8] Creative Tools For A Creative Age: Click & Create and The Games Factory: http://clickteam.com/.

[9] DJGPP, a complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running DOS: http://www.delorie.com/djgpp/.

[10] Allegro, A Game Programming Library: http://www.talula.demon.co.uk/allegro/.

[11] The Fastgraph Home Page: http://www.fastgraph.com/.

[12] The Microsoft® DirectX® Web site: http://www.microsoft.com/directx/.

[13] The MSDN Library: http://msdn.microsoft.com/library/.

[14] Loirak Developement Group. *DirectX Game Programming*: http://loirak.com/prog/directx/.

[15] Joffe, D. *Game Programming with DirectX*, 1998: http://www.geocities.com/SoHo/Lofts/2018/djdirectxtut.html.

[16] Gruber, D. *Color Reduction for Windows Games*, Visual Developer, Vol. 7, No. 1, 1996.

[17] Hori, H. *DelphiX source code and documentation*: http://www.ingjapan.ne.jp/hori/.