SIEMENS
mobile

Siemens Mobility Toolkit for Java™ Development

Developing Games with the Game API

Version 1.0, 11/2003

Cleverson Schmidt and SMTK Team

Table of Contents

Table of Figures

1 - Introduction

1.1 - About this white paper

The purpose of this white paper is to describe the Siemens Game API and the general steps in a game development process. This paper intends to be a beginner's guide on how to develop a game.

It is assumed that the reader has Java™ and MIDP1.0 programming knowledge.

1.2 - System requirements

To start developing games with SMTK the following requirements must be met.

SMTK Core must be installed.

At least one Emulator Pack must be installed.

J2SE™ SDK should be installed. It is used for debug purposes.

This configuration is intended for use on Windows 98 (1st or 2nd edition), Windows NT 4.0 (Service Pack 5 or higher), Windows 2000 or Windows XP operating systems running on Intel hardware.

- Intel® Pentium® 166MHz or compatible processor;
- RAM 64 MB (minimum requirement);
- 110-230 MB hard disk space;
  - 40 MB – SMTK;
  - 70 MB – J2SE™ SDK;
  - 120 MB – J2SE™ SDK documentation.

2 - Siemens Game API

The Game API is composed of a set of high performance classes that were designed specially for implementing games. Using the classes from the Siemens Game API makes your game run much faster than just use MIDP1.0. Trying to do the same using MIDP1.0 would result in a loss of performance.

Game entities can be easily mapped to objects of the Siemens Game API like sprites, layers and tiled maps (TiledLayer). Moreover the API offers a game centric version of the Canvas class that makes the task of writing games easier.

All these features are implemented in the `com.siemens.mp.game` and `com.siemens.mp.color_game` packages. The first is for black and white devices while the second has color support. Further details on each object can be found in the next chapters.

3 - Game Canvas

Game Canvas is a special and game centric version of Canvas class. It extends Canvas functionality providing extra features for game developers. In addition to the features inherited from Canvas (commands, input events, etc.) it also provides game-specific capabilities such as an off-screen graphics buffer and the ability to query key status.

One of the main advantages in using Game Canvas is its built-in off-screen buffer. But what is an off-screen buffer? An off-screen buffer is a copy of the screen that is not displayed until the developer requests for that. All changes to a canvas are not directly drawn to the screen but drawn to a second screen which is not visible (the off-screen buffer). Painting operations are faster because they are performed in memory rather than directly in the device' screen, and they can be reflected to the screen at once with a single method call, the `flushGraphics()` method.

Due to the off-screen buffer, a well know technique used by game developers called double buffering requires less effort to be implemented in a Game Canvas than in an ordinary Canvas. This buffer can be accessed using the `getGraphics()` method and all painting operations are performed under this graphics context. This approach leads to performance gains and it is a general recommendation to remove screen flickering. More info on the double buffering technique can be found in the chapter 7 (Performance considerations).

Another difference between Canvas and Game Canvas is how they handle key events. In Canvas we have to override key events like `keyPressed(..)` and `keyReleased(..)` in order to know when and which keys were pressed and released. Game Canvas offers an additional method for querying key status. At any given time we are able to know which keys were pressed calling the `getKeyStates()` method. It returns an integer where each bit represents a specific key on the device. A key's bit will be 1 if the key is currently down or has been pressed at least once since the last time this method was called. The bit will be 0 if the key is currently up and has not been pressed at all since the last time this method was called. The four lower bits are defined by UP_KEY, DOWN_KEY, LEFT_KEY, RIGHT_KEY, etc; the remaining bits may be optionally mapped to device-specific keys.

Here is how you could find out if the left key is pressed:

```
int keyState = getKeyStates();
if ((keyState & LEFT_KEY) != 0) {
    System.out.println("Left Key pressed");
}
```

4 - Game Logic

The main logic of a game could be written in the class which extends the `GameCanvas` class. We can divide the logic in 2 steps. The initialization step, where we set up and get some objects that will be further used in the game implementation and the loop step, where we make the game come true.

The first step, initialization, can be easily embodied in the `GameCanvas` constructor. The constructor is the right place to put some instructions like getting the `GameCanvas` offscreen buffer, getting the canvas dimensions, creating Sprites, tile maps (`TiledLayer`), and layer managers as well as game state initialization. Here is how we could implement the initialization step:

```
public class MyGameCanvas extends GameCanvas {
    private Graphics buffer;
    public MyGameCanvas(){
        super(true);
        buffer = getGraphics();
        createSprites();
        createMap();
        initializeGameState();
    }
}
```

An optional instruction would be starting the game loop (the second step) as the last instruction of the constructor.

After the game initialization, it's time to make the things move. We need to add the second step. We need to add a loop to control our game. As it is a CPU-bound task, it should be implemented in a separated thread, so that our game doesn't lose its responsiveness. One alternative is just implement the `Runnable` Interface in our Game Canvas class and consequently its `run()` method, that will be the place for our game loop. This loop will be responsible for carrying out the game. A typical loop is found in the Figure 4.1.

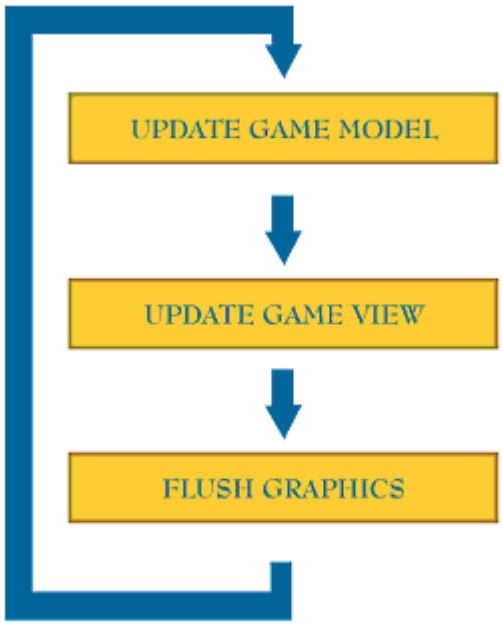

Figure 4.1: Game loop

There are three main tasks to be done inside the loop: update the game model, update the game view and flush the changes to the display's graphics context.

Update Game Model

Here is where we change the state of the objects that live in our game. Sprites, layers, tiles and every object which represents the game model have its state changing methods called from here. All these changes are made in the logical objects, therefore they are not perceived by the game player yet. They are just in memory level.

As key presses and releases reflect in changing the state of some objects, the querying of key status is done in this step as well.

Update Game View

After modifying the game state, the changes must be propagated to the end user. This is done through updating what the player sees, in other words the game view.

When we use the Siemens Game API this is easily achieved calling the paint method of each game entity (`Layers`, `Sprites` and `TiledLayers`) or the `LayerManager` as seen in the chapter 5. We pass the off-screen buffer as parameter to every paint call and afterwards the game view is ready to be reflected

to the game player. It is worth to mention that, at this point just the off-screen buffer is updated.

Flush Graphics

Until we reach this step, the game player actually does not receive any feedback on the changes performed so far. Changes that were first applied to the game model, its objects and then propagated to the game view through the off-screen buffer. To finalize the cycle we flush the buffer to the current displayable Game Canvas at once. This is done with a single method call, the `flushGraphics()` method. It automatically flushes the off-screen buffer to the display and does not return until the flush has been completed.

Code

The Figure 4.1 can be translated to the code as follows

```
public class MyGameCanvas extends GameCanvas implements Runnable{
    // constructor
    // other stuffs
    public void run(){
        while (running){
            updateGameModel();
            updateGameView()
            flushGraphics();
            try {
                Thread.sleep(GAME_SPEED);
            }
            catch (InterruptedException ie) {}
        }
    }
}
```

As said before, one of the alternatives to make the `MyGameCanvas` class run in a separated thread is implement the `Runnable` Interface. We could also make our class extends the Thread class or even use a `TimerTask`. Choosing implement the `Runnable` interface, we must provide an implementation to the `run()` method and then place the game loop inside of it.

We start a new thread in the constructor or in response to some user action like a command action. The code to create and start a new thread can be written in a single line:

```
new Thread(this).start();
```

After this line is executed, the game loop will start to run

5 - Game Elements

5.1 - Sprites

5.1.1 - What is it?

Every game is made of moving items. An item represents an object from the real world. Balls, soldiers, space ships, cars, all of them, at any given time, have a well defined position in terms of x and y coordinates in a 2D world. Besides a position, an item has also some other features that describe it like height and width.
All of these features and behavior are expressed by a Sprite object. Sprites have a set of attributes and methods that make the task of writting games a lot easier. We can move, show, hide, animate, check for collisions and much more using Sprites.

5.1.2 - Constructing sprites

There are three ways to create a sprite:

- **from another sprite**

```
public Sprite(com.siemens.mp.color_game.Sprite s)
       throws NullPointerException
```

All instance attributes (raw frames, position, frame sequence, current frame, visibility) of the source Sprite are duplicated in the new Sprite. It is just a clone method.

- **using an image**

```
public Sprite(javax.microedition.lcdui.Image image)
       throws NullPointerException
```

Creates a new non-animated Sprite using the provided mutable Image. By default, the Sprite is visible and positioned at (0,0).

```
Ex:
Image carImage = Image.createImage("car.png");
Sprite s = new Sprite(carImage);
```

- **using an image containing frames**

```
public Sprite(javax.microedition.lcdui.Image image,
              int frameWidth, int frameHeight)
       throws NullPointerException, IllegalArgumentException
```

Sprites can also be used as an animation as explained later on. This third constructor creates a new animated Sprite using frames contained in the provided mutable Image. The frames must be equally sized, with the dimensions specified by frameWidth and frameHeight. They may be laid out in the image horizontally, vertically, or as a grid. The width of the source image must be an integer multiple of the frame width, and the height of the source image must be an integer multiple of the frame height

In order to exemplify how a multiple frame sprite is created consider the following png file.

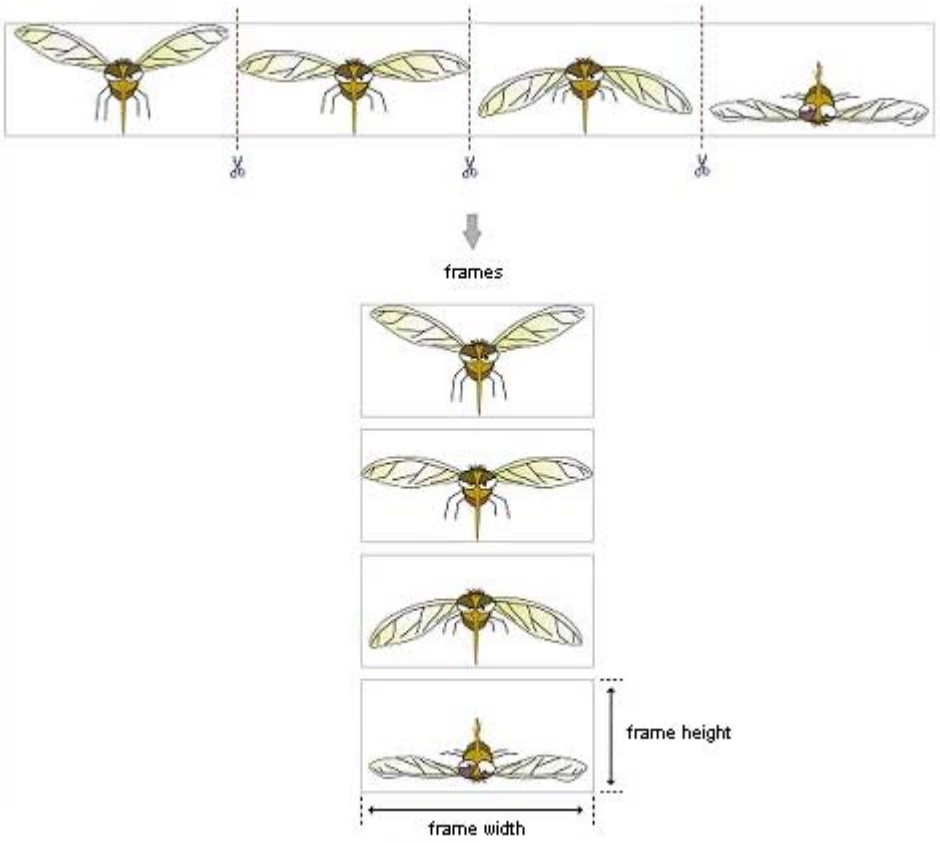

Figure 5.1: sprite frames

Considering that the original image is 400 pixels width and 40 pixels height then we could create an equally-sized frames sprite containing a sequence of 4 frames as follows.

```
Image flyImage = Image.createImage("fly.png");
Sprite s = new Sprite(flyImage, 100, 40);
```

5.1.3 - Moving sprites

Sprites can be easily moved using the move method. We just need to determine how many pixels we want the sprite move along horizontal and vertical axis and then pass these values to the move method like the following example.

```
sprite.move(2,3);
```

The parameters can be negative; in this case the sprite will move to the left or up, depending on which parameter is negative.
Notice that just calling the move method does not update the sprite in the canvas. We must call sprite's paint method and then `flushGraphics()` to reflect the new position in the screen.
The good news is that when a sprite is moved and repainted through its paint method, we don't need to erase the old version already painted in the canvas. The redraw is performed by the implementation. This leads to smooth animation and movement of a sprite.

5.1.4 - Animating sprites

When a sprite is created based in a multi frame image, it can be animated by showing different frames in a loop, timer, thread, etc. The frames contained in the sequence are mapped to an index and this index can be used to animate the sprite.
The Sprite class has a set of specific methods that are used in order to animate a frame sequence: nextFrame, prevFrame, setFrame, setFrameSequence.
Through these methods it is possible to traverse the frame sequence back and forth as well as define a sequence of frames different from the original one. In this case an array of integers is supplied containing the new sequence of frames to be shown.
To exemplify how the animation works consider the image below.

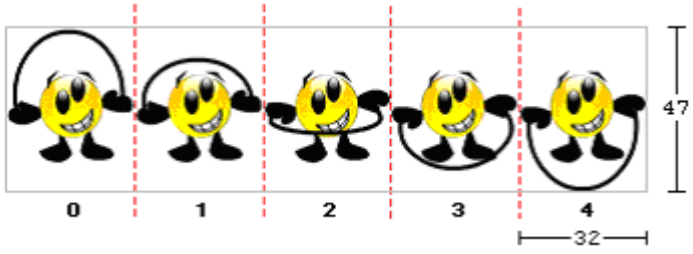

Figure 5.2: animating a frame sequence

The image is divided in the sprite constructor in 5 frames. If we don't set a frame sequence then the animation will be performed using the default sequence that displays the Sprite frames in a cyclic order, from 0 to 4. But if we want to properly animate this sequence we have to define a new sequence different from the default one. The sequence shown in the Figure 5.3 represents the correct sequence of frames to be displayed.

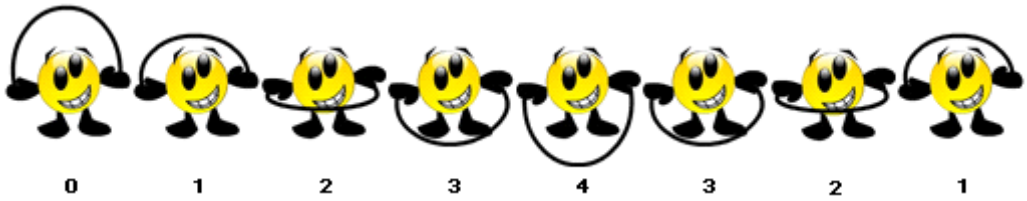

Figure 5.3: defining a frame sequence

Here is how you would create an animated sprite using the Figure 5.2:

```
Image smileyImage = Image.createImage("sourceImage.png");
Sprite smile = new Sprite(smileyImage, 32, 47);
int[] FRAME_SEQUENCE = {0, 1, 2, 3, 4, 3, 2, 1};
smile.setFrameSequence(FRAME_SEQUENCE);
```

The animation can be easily implemented in a separeted Thread:

```
public void run(){
    while (true){
        smile.nextFrame();
        smile.paint(buffer);
        flushGraphics();
        try {
            Thread.sleep(150);
        }
        catch (InterruptedException ie) {}
    }
}
```

Each time `nextFrame()` is called, it updates the sprite with the next frame contained in the frame sequence. This new frame is then painted in the off-screen buffer and afterwards copied to the screen using `flushGraphics()`.
The complete source code is available in the Appendix A - "A.1 - Animating Sprites"

5.1.5 - Detecting collisions

The basic task of collision detection is to determine whether two objects are in contact with each other. It is a feature that can be found in almost every game. A

starship hitting an asteroid, a soccer player kicking a ball or a monster being killed by a bullet are typical examples of collision between 2 game entities. We don't need to concern with checking x and y coordinates of 2 objects because this essential feature is already implemented for us. The Sprite class has 2 methods for detecting collisions.

The first one checks if 2 Sprites are colliding each other. It does so simply by checking if the Sprites' collision rectangles intersect.

The method signature is as follows:

```
public boolean collidesWith(com.siemens.mp.color_game.Sprite s,
                            boolean pixelLevel)
```

The second method for collision detection is used to check if a Sprite is colliding with an Image positioned at a given location. This task is performed checking if the Sprite's rectangle intersects with the Image rectangle.

Here is the method signature.

```
public boolean collidesWith(javax.microedition.lcdui.Image image, int x,
                            int y, boolean pixelLevel)
```

Both methods have a boolean parameter indicating the pixel-level detection. If pixel-level detection is used, a collision is detected only if opaque pixels collide. That is, an opaque pixel in the Sprite would have to collide with an opaque pixel in Image, or another Sprite, for a collision to be detected. Only those pixels within the Sprite's collision rectangle are checked. But take care with this parameter because currently it is not implemented in Siemens Game API.

To exemplify the concept let's use the example below:

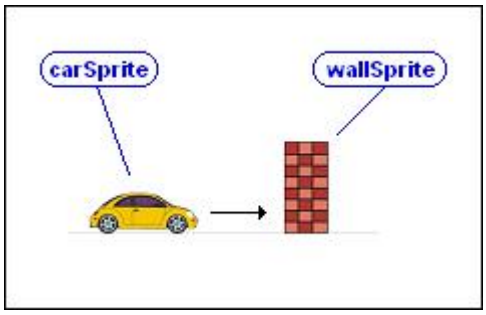

Figure 5.4: collision detection

We have a car and a wall. Both are mapped to Sprite objects.

```
Image carImage = Image.createImage("car.png");
Sprite car = new Sprite(carImage);
```

```
Image wallImage = Image.createImage("wall.png");
Sprite wall = new Sprite(wallImage);
```

From time to time the car moves to the right:

```
public void run() {
      while (true) {
          car.move(2,0);
          car.paint(buffer);
          flushGraphics();
          try {
              Thread.sleep(150);
          }
          catch (InterruptedException ie) {}
      }
}
```

If the above code were executed, the car would pass over the wall because there is not collision prevention. We need to add code to prevent unexpected behavior. The following code would fit:

```
public void run(){
        while (true){
            if (!(car.collidesWith(wall, false))){
                car.move(2,0);
                car.paint(buffer);
                flushGraphics();
                try {
                    Thread.sleep(150);
                }
                catch (InterruptedException ie) {}
            }
        }
}
```

The complete source code is available in the Appendix A - “A.2 - Detecting collisions”

5.2 - Tiled Layer

5.2.1 - What is it?

Game backgrounds are usually large. Maps, cities, streets are all examples of game scenarios. If backgrounds were represented by images, we would not have much more room for other things.

A common technique used by game developers for creating game backgrounds without wasting memory is construct them based in small parts, parts which are repeated over and over. The background is divided in a grid of equally sized cells

which are then filled with the small image fragments, these parts are called tiles. When we group the tiles together in a layer to construct a game background we have a Tiled Layer.

5.2.2 - Constructing Tiled Layers

It is not difficult to construct a game map based in a Tiled Layer. Suppose we want to create a city with houses, trees, lakes, gardens and so on, like the Figure 5.5.

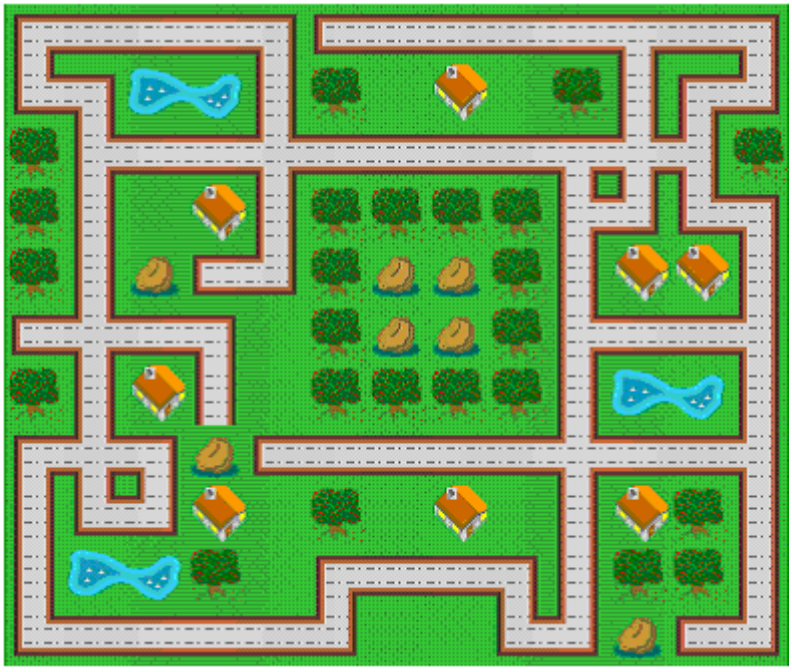

Figure 5.5: game map

The first task is to identify the entities or objects that will compose our map. Search the map for repetitive pieces and promote them as “tiles”. The whole map will be reduced in a few tiles resulting in a lot of saved memory.
It is important to mention that all tiles must be equally sized. Figure 5.6 shows candidate tiles.

After identifying the tiles we need now to group them together in a single image that will be further used as a source for the Tiled Layer construction.

For each tile in the source image is assigned an index number. They are numbered consecutively in row major order (indexes are assigned across the first row,

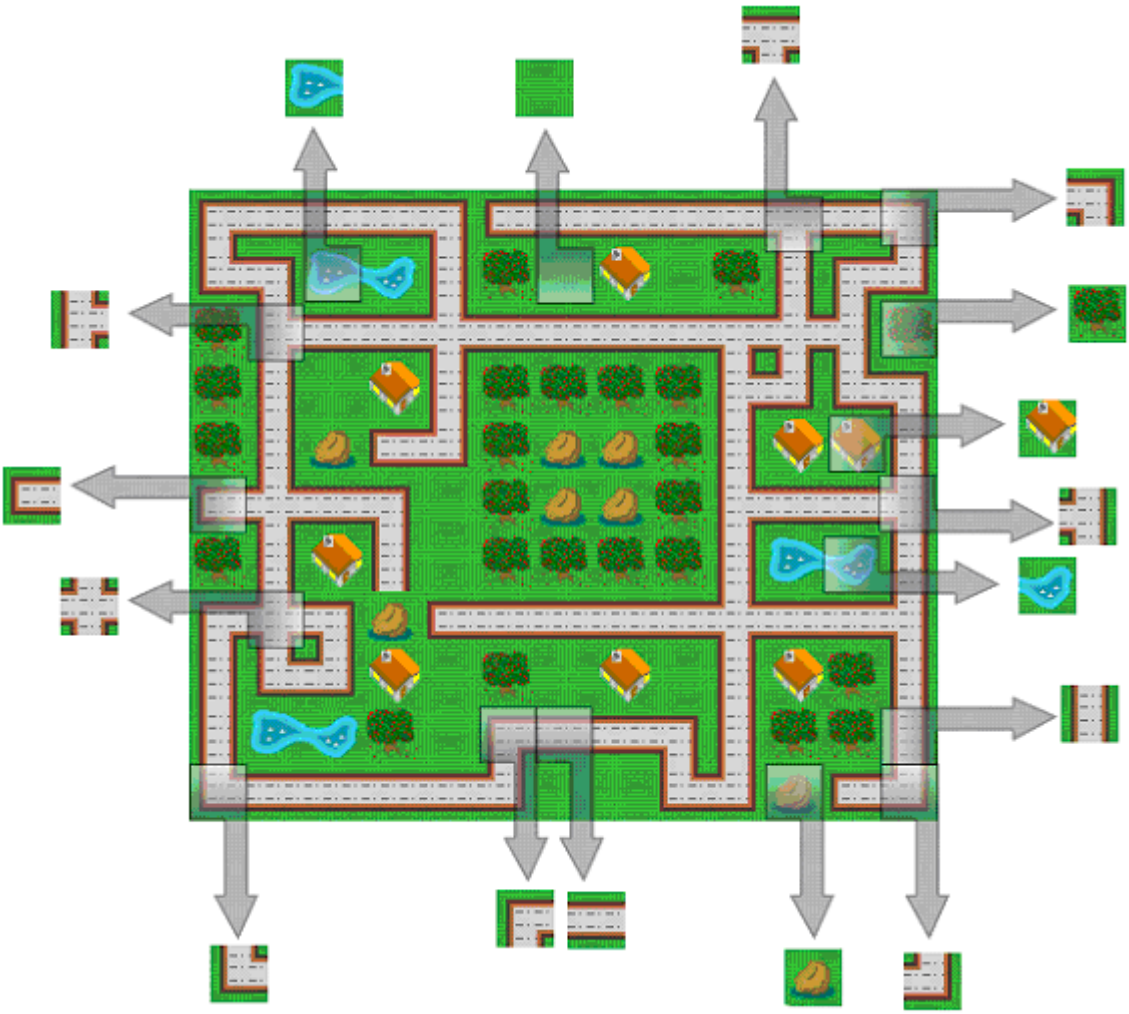

Figure 5.6: identifying tiles

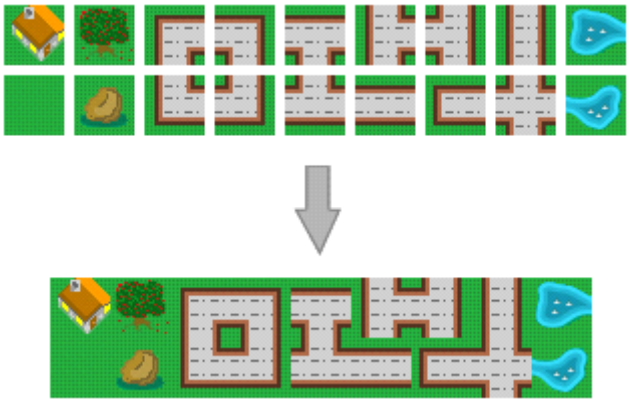

Figure 5.7: grouping the tiles in a single image

then the second row, and so on). These tiles are regarded as static tiles because there is a fixed link between the tile and the image data associated with it.

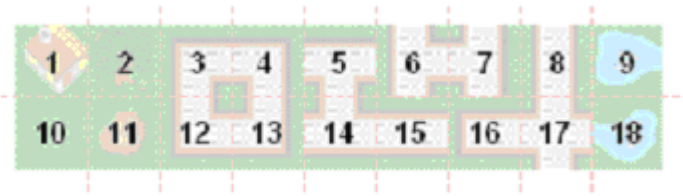

Figure 5.8: assigning indexes to the tiles

At last with the source image in our hands, we can now create our Tiled Layer. There is just one way to do so.

```
public TiledLayer(int columns, int rows,
                  javax.microedition.lcdui. Image image,
                  int tileWidth, int tileHeight)
     throws NullPointerException, IllegalArgumentException, OutOfMemoryError
```

And here is how you would construct it (Supposing each tile is 15 pixels wide and 15 pixels high and the map has 13 columns and 11 rows).

```
Image mapImage = Image.createImage("sourceImage.png");
TiledLayer map = new TiledLayer(13, 11, mapImage, 15, 15);
```

5.2.3 - Filling the Tiled Layer cells

All cells in the Tiled Layer grid are initially empty, they contain tile index 0. We need now to fill this grid using the tiles provided in the source image.
The grid should be filled with corresponding tile indexes so that we reach the result shown in the Figure 5.9

The TiledLayer class offers 2 methods for filling cells.
The first one binds a single cell, giving its row and column coordinates, to a static tile. The tile is referenced by its index in the source image.

```
public void setCell(int col, int row, int tileIndex)
            throws IndexOutOfBoundsException
```

The second method is a group version of the first. It binds a group of cells to a given tile.

```
public void fillCells(int col, int row, int numCols, int numRows,
                      int tileIndex)
```

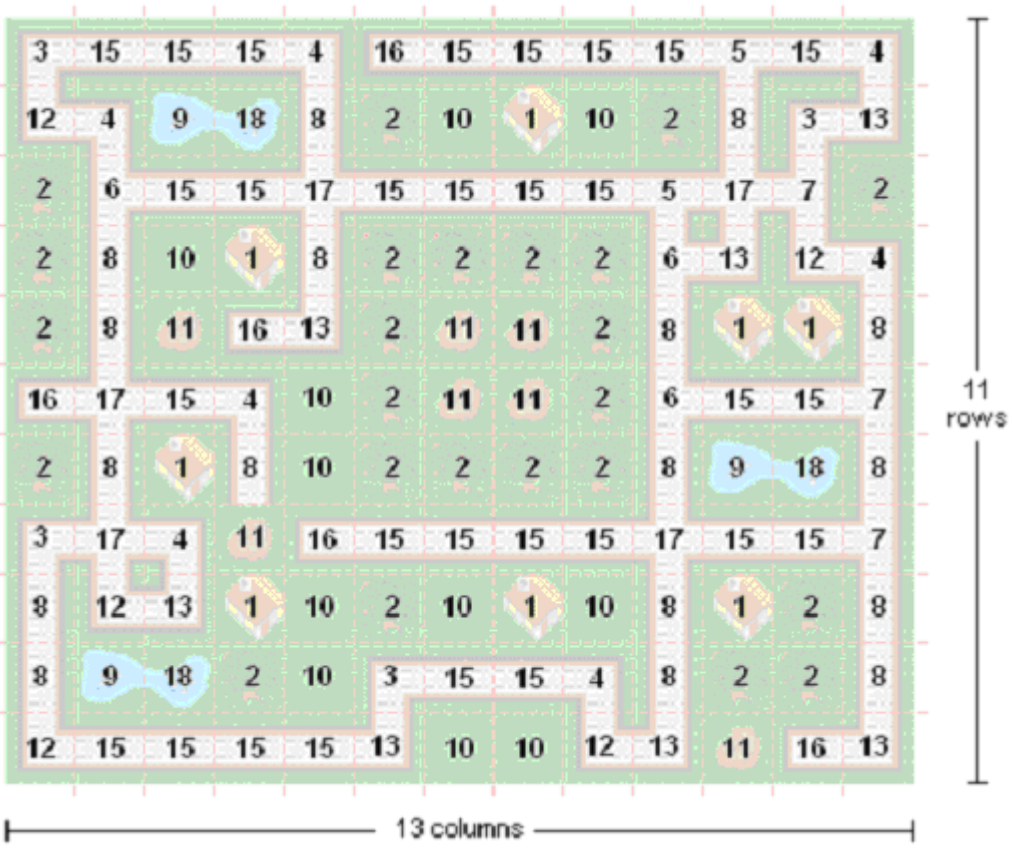

Figure 5.9: filling the Tiled Layer

In order to fill our Tiled Layer we declare an int array and initialize it with the indexes shown in the Figure 5.9.

```
int[][] mapTiles = {{ 3, 15, 15, 15,  4, 16, 15, 15, 15, 15,  5, 15,  4},
                    {12,  4,  9, 18,  8,  2, 10,  1, 10,  2,  8,  3, 13},
                    { 2,  6, 15, 15, 17, 15, 15, 15, 15,  5, 17,  7,  2},
                    { 2,  8, 10,  2,  8,  2,  2,  2,  2,  6, 13, 12,  4},
                    { 2,  8, 11, 16, 13,  2, 11, 11,  2,  8,  1,  1,  8},
                    {16, 17, 15,  4, 10,  2, 11, 11,  2, 06, 15, 15,  7},
                    { 2,  8,  1,  8, 10,  2,  2,  2,  2,  8,  9, 18,  8},
                    { 3, 17,  4, 11, 16, 15, 15, 15, 15, 17, 15, 15,  7},
                    { 8, 12, 13,  1, 10,  2, 10,  1, 10,  8,  1,  2,  8},
                    { 8,  9, 18,  2, 10,  3, 15, 15,  4,  8,  2,  2,  8},
                    {12, 15, 15, 15, 15, 13, 10, 10, 12, 13, 11, 16, 13}
 };
```

Finally we bind the array data to the Tiled Layer using the `setCell(..)` method in a loop construction.

```
int i,j = 0;
for (i = 0; i < map.getRows(); i++){
    for (j = 0; j < map.getColumns(); j++ ){
```

```
            map.setCell(j, i, mapTiles[i][j]);
        }
}
```

The complete source code is available in the Appendix A - “A.3 - Constructing Tiled Layers”

5.2.4 - Animated Tiles

It is quite often to have moving tiles in a game map. What If some of the trees or houses in our map start catching fire? We need a way to replace the tree tiles by burning tree tiles. The first thought that comes in mind is just replace the tiles one by one in the game map through the `setCell(…)` method. But this approach would require a separated `setCell(…)` call to each burning tree. A better way for doing this is using animated tiles. They allow the developer to change the appearance of a group of cells very easily. With the group of cells all filled with the animated tile, the appearance of the entire group can be changed by simply changing the static tile associated with the animated tile. This technique is very useful for animating large repeating areas without having to explicitly change the contents of numerous cells.

We can create an animated tile using the `createAnimatedTile(int)` method. This method returns the index to be used for the new animated tile, which is always a negative and consecutive number. The first animated tile has the index -1, the second -2 and so on. After creating the animated tile, we can change the static tile associated with it using the `setAnimatedTile(int, int)`. This call will reflect in all tiles that are associated to the animated tile.

One example is worth more than a thousand words. So let’s burn some trees!

Suppose a game with a small village as depicted in Figure 5.10.

Figure 5.10: small village

Suddenly this village is attacked by flying saucers and all its trees start to burn.

Figure 5.11: small village on fire

How to code that?

First we do what we have already done in the last section. We identify the tiles, create the source image and construct the Tiled Layer. Once created, we need to fill the Tiled Layer cells but now using a slightly approach. The animated tiles, trees on fire, need to be created separately.

The initial steps would be as follows.

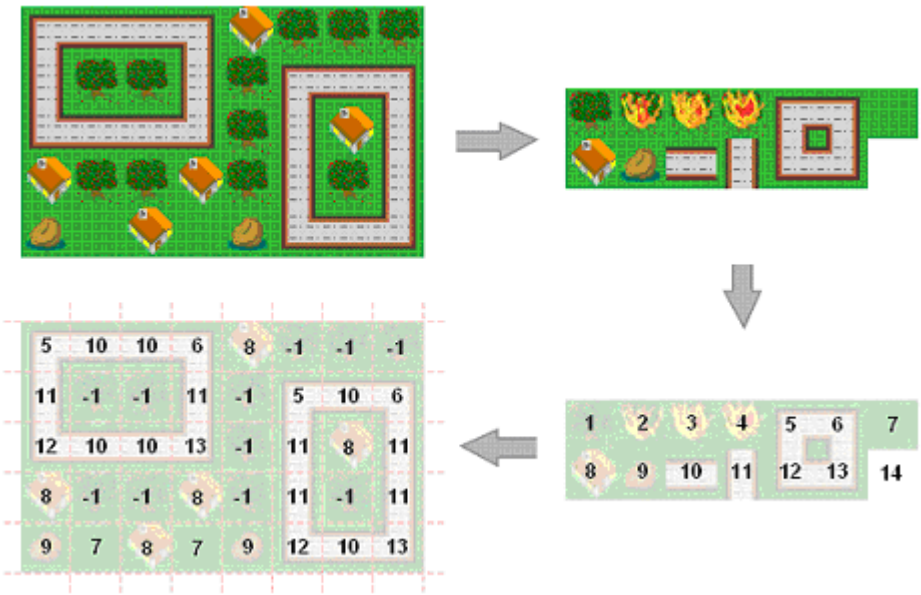

Figure 5.12: initial steps to construct the small village on fire

Notice that all trees in the map are bound to -1. This number is the animated tile index that is created through the createdAnimatedTile(..) method call.

```
tiledLayer.createAnimatedTile(1);
```

The parameter 1 means that the static tile 1, the one which represents the tree, is initially associated to the animated tile. Therefore whenever call to the setAnimatedTile(..) method will be propagated to all trees in the map.
If we wish to burn the trees, the only thing we need to do is call the `setAnimatedTile(int animatedTileIndex, int staticTileIndex)`. In this case we want to replace all normal trees by burning trees. The normal trees are associated to the animated tile referenced by the index -1, so we just need to change this animated tile to another one. Making the following call would fit:

```
tiledLayer.setAnimatedTile(-1, 2)
```

This call replaces all tiles referenced to the animated tile index -1 for the static tile 2, the burning tree.
As we have 3 frames to construct the animation, we just iterate over them in a threaded loop:

```
public void run() {
        try {
            while (true) {
                switch (map.getAnimatedTile(-1)){
                    case 1:
                        map.setAnimatedTile(-1, 2);
                        break;
                    case 2:
                        map.setAnimatedTile(-1, 3);
                        break;
                    case 3:
                        map.setAnimatedTile(-1, 4);
                        break;
                    case 4:
                        map.setAnimatedTile(-1, 2);
                        break;
                }
                map.paint(buffer);
                flushGraphics();
                Thread.sleep(100);
            }
        }catch(Exception e) {
            System.out.println("ops: "+e.getMessage());
        }
}
```

The loop first checks which is the current displayed tile, through `getAnimatedTile(-1)`, and then performs the tile's exchange calling `setAnimateTile(-1,nextTile)`.

The complete source code is available in the Appendix A "A.4 - Animating Tiled Layers"

5.3 - LayerManager

5.3.1 - What is it?

A game, in its final stage of development is composed of many and many layers. Maps, balls, soldiers, space ships, cars, all of them are normally represented using Sprite and Tiled Layer objects. When the display needs repainting we have to call the `paint(..)` method once for each layer. What if our game has 20 Sprites and 1 Tiled Layer? Would we have to perform 21 `paint(..)` calls?
Yes, if you want. But there is a better way for doing so. Use a Layer Manager, an object that works as a layer container, it manages a series of layers simplifying the process of rendering the layers that have been added to it.
Layers that are added to a Layer Manager are stored in an ordered list. The indexes in the list are related to the z-order of the layer, this means that the layer at index 0 is closest to the user while the layer with the highest index is furthest away from the user. This creates an idea of depth to the game. Layers can be added, removed and inserted in the Layer Manager's list.

5.3.2 - Using a Layer Manager

There is no secret on using a Layer Manager. Just think about the college's days when you were learning how to implement lists. A Layer Manager is layer list. It manages a group of layers through common list methods:

```
public void append(com.siemens.mp.color_game.Layer l)
            throws NullPointerException

public void insert(com.siemens.mp.color_game.Layer l, int index)
            throws NullPointerException, IndexOutOfBoundsException

public void remove(com.siemens.mp.color_game.Layer l)
```

Besides the lists methods, the `LayerManager` class provides several features that control how the game's Layers are rendered on the screen.
One of the good features of a Layer Manager is the view window. It controls the size of the visible region and its position relative to the LayerManager's coordinate system. Changing the position of the view window enables effects such as scrolling or panning the user's view. For example, to scroll to the right, simply move the view window's location to the right. The size of the view window controls how

large the user's view will be, and is usually fixed at a size that is appropriate for the device's screen.

6 - Sounds and special effects

6.1 - Playing sounds and tones

Sounds and games are tied together. A good game must be able to play sounds. MIDP version 1.0 was released without media support but Siemens implemented a set of classes that make possible to play sounds and tones. All classes you need are available under the `com.siemens.mp.media` and `com.siemens.mp.media.control` packages.
The Manager class is the start point for playing sounds. It is through a Manager that you obtain a Player for music playback or play a single tone.

6.1.1 - Player

A Player is the object responsible for the media rendering. It provides the methods needed to manage the media been played, methods like start, stop and close.
A Player object is obtained from a Manager with a static call to the method `createPlayer()` that returns a Player from the given Player source.
If we want to play a midi file we just supply the location of the midi like the example as follows:

```
Player p = Manager.createPlayer("music.mid");
```

Another alternative is to create a Player to play back media from an InputStream. After creating the player we are able to play its content, in this case the midi file. This can be done right after the Player creation but there are a few time-consuming tasks that the Player must do before starting. It needs to obtain the information required to acquire the media resources. The Player may have to communicate with a server, read a file, or interact with a set of objects. Besides that it still needs to fill buffers with media data, or perform other start-up processing. So if we call the start method right after the Player creation, all these tasks need to be carried out.
A good approach is to split the time consumed calling intermediary methods like `realize()` and `prefetch()`. The first method constructs portions of the Player without acquiring the scarce and exclusive resources while the second acquires the scarce and exclusive resources and processes as much data as necessary to reduce the start latency.
And finally to start the music we just call the `start()` method:

```
Player p = Manager.createPlayer("music.mid");
// realize() and prefetch() could be called to reduce the media starting
p.start();
```

6.1.2 - Tones

Simple tone generation is very easy when you use the Manager class. It has a static method, `playTone()` that given the note and duration will produce the specified tone:

```
public static void playTone(int note, int duration, int volume)
                    throws MediaException
```

A note is given in the range of 0 to 127 and the volume is given in the range from 0 to 100 where 100 represent the maximum volume. The summary of MIDI note numbers for different octaves can be seen in the Table 6.1.

Octave #	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Table 6.1: MIDI note numbers table

The MIDI specification only defines note number 60 as "Middle C", and all other notes are relative.

Note: the volume in the Siemens devices is divided in 4 groups: low, mid-low, mid-high and high. MIDlets use the application volume that is set through Setup - Audio - Volume – Applications

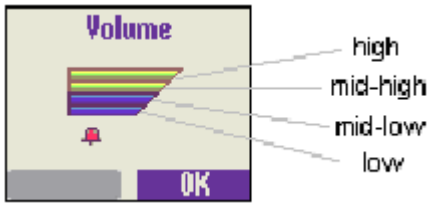

Figure 6.1: volume steps

When you set the volume from a MIDlet, this new value is mapped to one of the volume steps, 0-24 low, 25-50 mid-low, 51-75 mid-high and 76-100 high.
Bear in mind that the application volume takes precedence over the volume set from a Java™ application. You can not set programmatically a volume higher than the global application volume. So if the application volume is set to mute and you set it programmatically to 100 (high) the phone continues mute. Or if the application volume is mid-high then you can just set it to low, mid-low or mid-high.

6.2 - Vibrator

Siemens gives the possibility of adding intensity to the game play experience through the implementation of the Vibrator class. Explosions, punches, and car crashes all come alive with this feature. It gives basic access to the phone's vibrator. You can activate the vibrator either for a given period of time by using triggerVibrator method, or you can control the duration yourself by using the startVibrator and stopVibrator methods. All of them are static methods.
The Vibrator class is found in the com.siemens.mp.game package.
Here is how you would make your games vibrate:

```
Vibrator.startVibrator();
Thread.sleep(1000);
Vibrator.stopVibrator();
```

Another way to get the same result:

```
Vibrator.triggerVibrator(1000);
```

6.3 - Light

All Siemens Java™-enabled devices have a common behavior when the phone detects a long period (15-20 seconds) of user inactivity. The LCD back light fades

out. This may be undesired for games that demand user inactivity. For example, there are some board games like chess that needs reasoning and therefore more time between user interactions. In these cases the player wouldn't be much happy if he had to push some button from time to time to avoid the display's fading out. The solution for that is to disable that feature through a simple static method call:

```
public static void setLightOn()
```

This call will change the default behavior causing the display to stay permanently on.
A call to `setLightOff()` restores the original behavior:

```
public static void setLightOff()
```

Notice that this call will not result in the display to fade out immediately.

6.4 - Leds

The Siemens M55 model has leds on both sides as shown in the Figure 6.2. These leds can be controlled by a Java™ application through a specific class designed only for the M55. Using the LedControl class allows you to switch on and off the LEDs on both sides of the phone. The method names are very intuitive as seen below:

```
public static void switchOFF(int led)
public static void switchON(int led)
```

The led parameter is either LED_TOP or LED_BOTTOM.

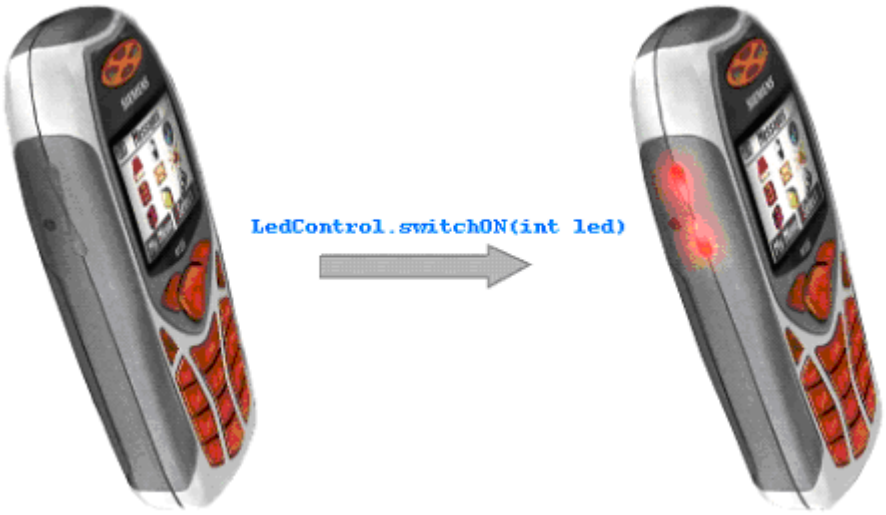

Figure 6.2: M55 leds

Besides switching on and off the leds, this class also allows to play a light pattern. A light pattern is a sequence of flashes so that the LEDs can flash in different “rhythms” depending on the current pattern. To play a light pattern, just call the method as follows:

```
public static void playPattern(int pattern)
```

There are several patterns defined in the LedControl class as constant fields. Patterns like P_BEAT, P_PULSATING, P_WAVE etc. To stop playing a pattern call the `stopPattern()` method.

7 - Performance Considerations

7.1 - General performance tips

7.1.1 - Rule of thumb

There is a basic principle of tuning: "**Do not fix what is not broken**". Do not try to increase the performance of your application if the gains are not significant. A small increase in performance with a large increase in the complexity and obscurity of your technique is a bad trade. Not merely because complex code is more likely to be a source of bugs, but also because complex code will be harder to read for future maintainers. So resist the temptation and remember this rule.

7.1.2 - Measuring your code

The market is plenty of good profiling tools but all of them slow down the application they are profiling. There is only one reliable way to determine the time taken by each part of the application. The technique consists in one method call:

```
public static long currentTimeMillis()
```

It is a static method from the System class and it is quick and has no effect on application timing (see further note).
The currentTimeMillis() method returns the current time in milliseconds. To measure a block of code just write it between 2 currentTimeMillis() calls, like the example below.

```
long start, finish, total;
start = System.currentTimeMillis();
...//the code you want to measure
end = System.currentTimeMillis();
total = finish - start;
```

Note: According to Jack Shirazi in his Java™ Performance Tuning book, System.currentTimeMillis() can take up to half a millisecond to execute. Any measurement including the two calls needed to measure the time difference should be over an interval greater than 100 milliseconds to ensure that the cost of the calls are less than 1% of the total measurement.

Besides time measurement we can also check application's memory consumption. The Runtime class has the 2 methods we need for that:


```
public long freeMemory()
public long totalMemory()
```

The first one returns the amount of free memory in the system while the second returns the total amount of memory in the Java™ Virtual Machine. They can be used together to
discover how much memory an object or a routine is consuming.
Below is the code fragment to check the memory used by a piece of code.

```
Runtime r = Runtime.getRuntime();
long before, after, total;
before = r.freeMemory();
...// the code you want to measure
after = r.freeMemory();
total = before - after;
```

7.1.3 - Object Creation

With the introduction of low-level memory management and a garbage collector, Java™ developers tend to be lazier regarding memory usage. If you fall in this group, take care.
Remember that cell phones are a resource constrained environment and object creation affects directly the performance of your program. It is a CPU intensive operation and implies in memory use, which is limited, and in more frequent calls to the Garbage Collector, which consumes CPU time every time it runs. There are several ways to create and use objects more effectively, saving memory, CPU time and effort:

- Avoid creating objects in frequently used routines. Try to reuse the object created in these routines. Consider how much time you'll need that object. If you are going to use it several times during your routine, keep a reference to it.
- When several instances of a class need access to a particular object, turn this object to static. This saves memory since each instance of this class won't have a copy of this object.

Bear in mind that the more objects you create, the more time the garbage collector uses to finish its task. And the garbage collector is not our friend, it is a time and CPU consumer. You can use it sparingly, but do not do that during gameplay. If your game is processor intensive then you have to avoid calling the garbage col-

lector in the main game loop. The right thing to do in that case is to choose the right moment when to let the garbage collector do its work. Usually the right moment is when the game reaches the end of a level or expects some user interaction. Otherwise running the garbage collector will obviously deteriorate the game performance.

Note: Calling `System.gc()` method **suggests** that the Java™ Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java™ Virtual Machine has made a best effort to reclaim space from all discarded objects. This call is just a suggestion, there is no guarantee that it will be performed instantly.

7.1.4 - Strings

Strings are immutable objects, once created they can't be modified. Methods that look like to change the content of a String actually create another String due to the fact that a String is immutable. For example, if you use the `trim()` method, a new String is created, causing the use of more memory and CPU.
String class is final, meaning you can't subclass it and modify its methods. Every String has internally a char array, which holds the Strings content and it is not accessible, meaning that whatever you want to do with a String has to be done through its methods.
A common String manipulations mistake is shown in the following fragment:

```
String s = " ";
for (int i = 0; i < 10; i++)
   s = s + i;
```

String concatenation performed this way results in lots of extra objects to be generated. Each time the concatenation is performed, a new String is created to hold the new value. The solution of this problem is very simple, just use a StringBuffer and perform the concatenations calling the `append()` method. This saves memory avoiding the creation of more objects. However, if the StringBuffer object is modified, a new char array is created losing the reference with the original String. One way to increase the performance is throw the Strings away and use char arrays directly. Since char arrays are a primitive type, no objects will be created when altering it, saving memory and CPU. The downturn comes down to the amount of work needed to deal with char arrays to manipulate the data.

Some advices when using Strings:

- Some methods of the String class does not copy the contents of the String to another object like `String.substring()`.
- Use StringBuffer to create Strings at runtime.
- To manipulate characters, copy the String char array into a new char array and manipulate from there.

7.1.5 - I/O

When performing I/O operations, it's important to release any resources as soon as they were used. Use the finally clause to do the job.

```
HttpConnection con = null;
InputStream in = null;

try {
   con = (HttpConnection)Connector.open(url);
   in = con.openInputStream();
} catch(IOException ioe) {
   //handle the exception
}
finally {
   try {
      if(in != null)
         in.close();
      if(con != null)
         con.close();
   } catch(IOException e){
      //handle the exception
   }
}
```

The code looks a little messy with the try catch inside the finally block. One option to clean up the code is put this code inside a method that throws an IOException.

```
public void makeConnection(String url) throws IOException {
   HttpConnection con = null;
   InputStream in = null;
   try {
      con = (HttpConnection)Connector.open("http://foo.com");
      in = con.openInputStream();
   }
   finally {
      if(in != null)
         in.close();

      if(con != null)
         con.close();
   }
```

```
}
```

No matter what happens, the connection will be closed. If an exception occurs or if everything runs fine, the finally block will always be executed, closing the connection.

7.1.6 - Exceptions

Throwing exceptions involves high performance costs. This is similar to running hundreds of lines of code.

An exception when not thrown doesn't use CPU time and therefore it does not slow down the application. Otherwise, when thrown, it generates a significant overhead.

Exceptions should be thrown only if they are absolutely necessary. Consider replacing exceptions with good design practices.

An efficient approach is to reuse the same exception instance in several places. It speed up the application execution saving time because it does not have to create new objects for each exception being thrown and registering the object in the stack trace.

The technique consists of the following: create an exception object and reuse it across the code. This saves CPU time.

```
Exception GENERAL_EXCEPTION = new Exception();
try {
   //...
   throw GENERAL_EXCEPTION;
} catch(Exception e) {}

try {
   //...
   throw GENERAL_EXCEPTION;
} catch(Exception e) {}
```

The disadvantage of reusing an exception is that the correct stack trace of the instance is lost because it holds the stack trace from the last exception only.

Depending on your necessity, you may need to use the correct stack trace, so be careful when using this technique.

7.1.7 - Tools

There are several tools available that help you tune and find the bottlenecks of your program. Some of them are free and are available on the Internet and others are commercial.

A bytecode obfuscator can be used to reduce the size of your class files. An obfuscator is a tool that makes your code difficult to understand. Its main purpose is to avoid the exposition to reverse engineering but you can also use it for reducing the footprint of your class files. It is not difficult to reduce a class in around 15 to 20 percent. There are two good obfuscators, RetroLogic's RetroGuard which can be found at http://www.retrologic.com and ProGuard which can be found at http://proguard.sourceforge.net. But take care! Be sure that you preverify your class before obfuscate it, otherwise the application manager will have problems to load it. One more thing: the size reduction is realized in applications that don't use extensively external resources as media files or pictures.
Another useful toll is a profiler. A profiler analyzes the amount of time and memory usage your code spends performing various tasks. With this information, you can determine the bottlenecks in your code and correct it.

7.2 - Game performance tips

7.2.1 - Siemens Game API

Using the classes from the Siemens Game API makes your game run much faster than just use MIDP1.0. Game API has special classes and methods that were designed specially for implementing games. A workaround using MIDP1.0 would result in a high loss of performance. The only drawback is loss of portability but even this disadvantage is not that bad. The standard Game API included in the upcoming MIDP version 2.0 is very similar to the current Siemens Game API, so the effort needed to port your games to MIDP2.0 will be minimal compared to what you would have to do if you just use MIDP1.0.

7.2.2 - Game design

When implementing games using J2ME™ there is no way to get rid of Object Oriented Programming (OOP). It is an inherent Java™ feature and it is high recommended in software development. In spite of its countless advantages, OO has a slight cost in terms of efficiency. As objects are references allocated dynamically, there is a small overhead to store and later on retrieve them from the heap. Moreover, when we use polymorphic methods there is an additional small time penalty, as the method to be executed is chosen at run time by searching through the object hierarchy. These disadvantages seem to be of relatively little importance compared with OO benefits but in game development and particularly mobile

game development, we are always searching for ways to speed up the overall performance.

What is being said in this white paper is not to throw away OO concepts but to have an awareness of its costs. So remember that sometimes good OO practices could lead a game to CPU and memory wastes. One of the good practises that can be avoided is using getters and setters to manipulate class attributes. But just do that when you were really desperate and there are no more alternatives. They are used mostly for encapsulating private data. If you make the data public you will no longer need these methods because you will have direct access to the data. This approach results in a smaller class file and therefore a faster loading time. But do that only in special cases as using getters and setters is a good OO practice.

One common mistake is using dynamic code inappropriately. Take a look at the following example where several lines are drawn based in the canvas width and height:

```
public void paint(Graphics g) {
   for (int i = 0; i < canvas.getHeight(); i++)
      g.drawLine(0, i, canvas.getWidth(), i);
}
```

This construction leads to a bad performance because the methods `getHeight()` and `getWidth()` are called in each loop iteration and they always return the same value. A better approach is to move the calls out of the loop:

```
private int height = canvas.getHeight();
private int width = canvas.getWidth();

public void paint(Graphics g) {
   for (int i = 0; i < height; i++)
      g.drawLine(0, i, width, i);
}
```

7.2.3 - Graphics and double buffering

Graphics is probably the most important component in a game, especially when involves animation. The animations should run smoothly without affecting the game experience.

One technique that is almost an obligation when developing games is Double Buffering. This technique ensures that the animation will be displayed correctly, reducing screen flickering. Double Buffering, like the name says, consists of having two buffers to make an animation. One of the buffers is displayed on the

screen, while the other is being drawn. This off-screen buffer will only be displayed when it's completely drawn, replacing the image on the screen while another image is being drawn in the back and so on.

The Siemens GameCanvas class already comes with an off-screen buffer, making the developer's life a little bit easier. You just request this buffer calling the `getGraphics()` method and perform all draw operations with it.

Every time the off-screen Graphics object is used to draw an image, sprite or layer, they are not displayed on the screen until the `flughGraphics()` method from GameCanvas is called. This method guarantees that only when all objects are properly drawn, they will be displayed.

In the example below, two Sprites are created and the off-screen Graphics object is obtained from the GameCanvas, using the `getGraphics()` method. After manipulating the images, the Sprites are painted using the paint() method from the Sprite class, but they are not displayed on the screen until the `flushGraphics()` method is called.

```
Sprite s1 = new Sprite(img1);
Sprite s2 = new Sprite(img2);
Graphics g = getGraphics(); // get the offscreen buffer
.../manipulate the sprites
s1.paint(g);      // paint s1 in the offscreen buffer
s2.paint(g);      // paint s2 in the offscreen buffer
flushGraphics(); // display the sprites on the screen
```

8 - Conclusion

The game developing process is not an easy task. In additional to all game inherent difficulties like animation, collision detection and graphics operations we also have to face one more constraint, the phone is limited in its processing power, memory, connectivity, and display size. To overcome the challenge and simplify the process Siemens supplies a Game API providing features that help developers and improve game performance. The intention of this White Paper was to introduce the Siemens Game API and encourage developers to start in this exciting area.

A - Source Code

A.1 - Animating Sprites

```
/*
This example is compound of 2 class files. A main MIDlet class that has the
responsibility of creating a GameCanvas object and put it visible. The Game-
Canvas handles the sprite creation and animation.
*/

// File SmileMIDlet.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SmileMIDlet extends MIDlet {

    private Command exitCommand;
    private Display display;

    public SmileMIDlet() {
        display = Display.getDisplay(this);
    }

    public void startApp() {
        SmileGameCanvas canvas = new SmileGameCanvas(this);
        display.setCurrent(canvas);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void quit(){
        destroyApp(false);
        notifyDestroyed();
    }

}

// File SmileGameCanvas.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import com.siemens.mp.color_game.*;

public class SmileGameCanvas
            extends GameCanvas implements CommandListener, Runnable{
```

```
    private SmileMIDlet midlet;
    private Sprite smile;
    private Graphics buffer;
    private Command exitCommand;
    private Command startAnimationCommand;
    private final int[] FRAME_SEQUENCE = {0, 1, 2, 3, 4, 3, 2, 1};
    private final int SPEED = 150;

    public SmileGameCanvas(SmileMIDlet midlet){
        super(true);
        this.midlet = midlet;
        buffer = getGraphics();
        exitCommand = new Command("exit", Command.EXIT, 1);
       startAnimationCommand = new Command("start animation",Comand.SCREEN,
0);
        addCommand(exitCommand);
        addCommand(startAnimationCommand);
        setCommandListener(this);
        createSprite();
    }

    private void createSprite(){
        try {
            Image smileyImage = Image.createImage("smile.png");
            smile = new Sprite(smileyImage, 32, 47);
            smile.setPosition(34, 15);
            smile.setFrameSequence(FRAME_SEQUENCE);
        }catch (Exception e){
            System.out.println("Ops: "+e.getMessage());
        }
    }

    private void animateSprite(){
        Thread t = new Thread(this);
        t.start();
    }

    public void run(){
        while (true){
            smile.nextFrame();
            smile.paint(buffer);
            flushGraphics();
            try {
                Thread.sleep(SPEED);
            } catch (InterruptedException ie) {}
        }
    }

    public void commandAction(Command c, Displayable s) {
```

```
        if (c == exitCommand) {
            midlet.quit();
        }else if (c == startAnimationCommand){
            animateSprite();
        }
    }
}
```

A.2 - Detecting collisions

```
/*
This example is compound of 2 class files. A main MIDlet class that has the
responsibility of creating a GameCanvas object and put it visible. The Game-
Canvas creates 2 sprites, a car and a wall, then it starts moving the car
towards the wall. Each time the car is moved, a collision detection is per-
formed to verify if the car can move without crash to the wall
*/

// File CarHittingWallMIDlet.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class CarHittingWallMIDlet extends MIDlet {

    private Command exitCommand;
    private Display display;

    public CarHittingWallMIDlet() {
        display = Display.getDisplay(this);
    }

    public void startApp() {
        CarHittingWallGameCanvas canvas = new CarHittingWallGameCan-
vas(this);
        display.setCurrent(canvas);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void quit(){
        destroyApp(false);
        notifyDestroyed();
    }

}
// File CarHittingWallGameCanvas.java
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import com.siemens.mp.color_game.*;

public class CarHittingWallGameCanvas extends GameCanvas implements Com-
mandListener, Runnable{

    private CarHittingWallMIDlet midlet;
    private Sprite car;
    private Sprite wall;
    private Graphics buffer;
    private Command exitCommand;
    private Command moveCarCommand;
    private final int SPEED = 150;

    public CarHittingWallGameCanvas(CarHittingWallMIDlet midlet){
        super(true);
        this.midlet = midlet;
        buffer = getGraphics();
        exitCommand = new Command("exit", Command.EXIT, 1);
        moveCarCommand = new Command("move car", Command.SCREEN, 0);
        addCommand(exitCommand);
        addCommand(moveCarCommand);
        setCommandListener(this);
        createSprites();
    }

    private void createSprites(){
        try{
            Image carImage = Image.createImage("car.png");
            car = new Sprite(carImage);
            car.setPosition(10, 30);
            car.paint(buffer);

            Image wallImage = Image.createImage("wall.png");
            wall = new Sprite(wallImage);
            wall.setPosition(80, 30);
            wall.paint(buffer);

            flushGraphics();

        }catch (Exception e){
            System.out.println("Ops: "+e.getMessage());
        }
    }
    private void moveCar(){
        Thread t = new Thread(this);
        t.start();
    }
```

```
    public void run(){
        while (true){
            if (!(car.collidesWith(wall, false))){
                car.move(2,0);
                car.paint(buffer);
                flushGraphics();
                try {
                    Thread.sleep(SPEED);
                }
                catch (InterruptedException ie) {}
            }
        }
    }

    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            midlet.quit();
        }else if (c == moveCarCommand){
            moveCar();
        }
    }
}
```

A.3 - Constructing Tiled Layers

```
/*
This example is compound of 2 class files. A main MIDlet class that has the
responsibility of creating a GameCanvas object and put it visible. The Game-
Canvas creates a Tiled Layer based in a source Image and then fills it based
in a int array which contains the indexes to construct a game map
*/

// File CreatingATiledLayerMIDlet.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class CreatingATiledLayerMIDlet extends MIDlet {

    private Command exitCommand;
    private Display display;

    public CreatingATiledLayerMIDlet() {
        display = Display.getDisplay(this);
    }

    public void startApp() {
        CreatingATiledLayerCanvas canvas = new CreatingATiledLayerCan-
vas(this);
        display.setCurrent(canvas);
    }
```

```
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void quit(){
        destroyApp(false);
        notifyDestroyed();
    }
}

// File CreatingATiledLayerCanvas.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import com.siemens.mp.color_game.*;

public class CreatingATiledLayerCanvas extends GameCanvas
                                       implements CommandListener{

    private CreatingATiledLayerMIDlet midlet;
    private Sprite car;
    private Sprite wall;
    private Graphics buffer;
    private Command exitCommand;
    private Command createTiledLayerCommand;
    private TiledLayer map;

    int[][] mapTiles = {{ 3, 15, 15, 15,  4, 16, 15, 15, 15, 15,  5, 15,  4},
                        {12,  4,  9, 18,  8,  2, 10,  1, 10,  2,  8,  3, 13},
                        { 2,  6, 15, 15, 17, 15, 15, 15, 15,  5, 17,  7,  2},
                        { 2,  8, 10,  2,  8,  2,  2,  2,  2,  6, 13, 12,  4},
                        { 2,  8, 11, 16, 13,  2, 11, 11,  2,  8,  1,  1,  8},
                        {16, 17, 15,  4, 10,  2, 11, 11,  2, 06, 15, 15,  7},
                        { 2,  8,  1,  8, 10,  2,  2,  2,  2,  8,  9, 18,  8},
                        { 3, 17,  4, 11, 16, 15, 15, 15, 15, 17, 15, 15,  7},
                        { 8, 12, 13,  1, 10,  2, 10,  1, 10,  8,  1,  2,  8},
                       { 8,  9, 18,  2, 10,  3, 15, 15,  4,  8,  2,  2,  8},
                      {12, 15, 15, 15, 15, 13, 10, 10, 12, 13, 11, 16, 13}
                    };


    public CreatingATiledLayerCanvas(CreatingATiledLayerMIDlet midlet){
        super(true);
        this.midlet = midlet;
        buffer = getGraphics();
        exitCommand = new Command("exit", Command.EXIT, 1);
        createTiledLayerCommand=new Command("Tiled Layer", Command.SCREEN,
0);
```

```
        addCommand(exitCommand);
        addCommand(createTiledLayerCommand);
        setCommandListener(this);
    }

    private void fillTiles(TiledLayer map){
        int i,j = 0;
        for (i = 0; i < map.getRows(); i++){
            for (j = 0; j < map.getColumns(); j++ ){
                map.setCell(j, i, mapTiles[i][j]);
            }
        }
    }

    private void createTiledLayer(){
        try{
            Image mapImage = Image.createImage("SourceImage.png");
            map = new TiledLayer(13,11, mapImage, 15, 15);
            fillTiles(map);
            map.paint(buffer);
            flushGraphics();
        }catch (Exception e){
            System.out.println("ops: "+e.getMessage());
        }

    }

    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            midlet.quit();
        }else if (c == createTiledLayerCommand){
            createTiledLayer();
        }
    }
}
```

A.4 - Animating Tiled Layers

```
/*
This example is compound of 2 class files. A main MIDlet class that has the
responsibility of creating a GameCanvas object and put it visible. The Game-
Canvas creates a Tiled Layer based in a source Image and then fills it based
in a int array which contains the indexes to construct a game map. The ani-
mated tiles are represented by negative numbers in the Tiled Layer's matrix
of integers. The animation is performed in a threaded loop that just
replaces the current tile by the next tile of the animation.
*/

// File SmallVilleBurningMIDlet.java
import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;

public class SmallVilleBurningMIDlet extends MIDlet {

    private Command exitCommand;
    private Display display;

    public SmallVilleBurningMIDlet() {
        display = Display.getDisplay(this);
    }

    public void startApp() {
        SmallVilleBurningCanvas canvas = new SmallVilleBurningCanvas(this);
        display.setCurrent(canvas);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void quit(){
        destroyApp(false);
        notifyDestroyed();
    }
}

// File SmallVilleBurningCanvas.java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import com.siemens.mp.color_game.*;

public class SmallVilleBurningCanvas extends GameCanvas
                              implements CommandListener, Runnable{


    private SmallVilleBurningMIDlet midlet;
    private Sprite car;
    private Sprite wall;
    private Graphics buffer;
    private Command exitCommand;
    private Command burnCommand;
    private TiledLayer map;

    private static int[][] mapTiles = {{ 5, 10, 10,  6,  8, -1, -1, -1},
                                       {11, -1, -1, 11, -1,  5, 10,  6},
                                       {12, 10, 10, 13, -1, 11,  8, 11},
                                       { 8, -1, -1,  8, -1, 11, -1, 11},
                                       { 9,  7,  8,  7,  9, 12, 10, 13}
```

```
                                        };

    public SmallVilleBurningCanvas(SmallVilleBurningMIDlet midlet){
        super(true);
        this.midlet = midlet;
        buffer = getGraphics();
        exitCommand = new Command("exit", Command.EXIT, 1);
        burnCommand = new Command("burn! burn!", Command.SCREEN, 0);
        addCommand(exitCommand);
        addCommand(burnCommand);
        setCommandListener(this);
        createTiledLayer();
    }

    private void fillTiles(TiledLayer map){
        map.createAnimatedTile(1);
        int i,j = 0;
        for (i = 0; i < map.getRows(); i++){
            for (j = 0; j < map.getColumns(); j++ ){
                map.setCell(j, i, mapTiles[i][j]);
            }
        }
    }

    private void createTiledLayer(){
        try{
            Image mapImage = Image.createImage("/SourceImageTreeMap.png");
            map = new TiledLayer(8,5, mapImage, 15, 15);
            fillTiles(map);
            map.paint(buffer);
            flushGraphics();
        }catch (Exception e){
            System.out.println("ops: "+e.getMessage());
        }

    }

    private void burnTrees(){
        new Thread(this).start();
    }

    public void run() {
        try{
            while (true){
                switch (map.getAnimatedTile(-1)){
                    case 1:
                        map.setAnimatedTile(-1, 2);
                        break;
                    case 2:
                        map.setAnimatedTile(-1, 3);
```

```
                        break;
                    case 3:
                        map.setAnimatedTile(-1, 4);
                        break;
                    case 4:
                        map.setAnimatedTile(-1, 2);
                        break;
                }
                map.paint(buffer);
                flushGraphics();
                Thread.sleep(100);
            }
        }catch(Exception e){
            System.out.println("ops: "+e.getMessage());
        }
    }

    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            midlet.quit();
        }else if (c == burnCommand){
            burnTrees();
        }
    }
}
```