

Sun Certified Mobile Application Developer (311-110)  
Sun Microsystems, Inc.

Exam Notes  
Sathya Srinivasan

02/12/2004

## Table of Contents

Performance improvements .....	3
Application size reduction .....	3
GameCanvas .....	4
Difference between GameCanvas and Canvas.....	4
LayerManager.....	5
Layer .....	6
Sprite .....	6
TiledLayer.....	6

## Chapter 9 MIDP Game UI

---

Given a scenario, develop code using the MIDP Game API package to improve performance and reduce application size.

### Performance improvements

- Most of the implementation of the API in the devices will be done at the native code level. So the performance will be significantly better.
- API is set in such a way that hardware acceleration can be done easily in the implementation.
- Images for animation and layers are stored in a single buffer and are rendered after every game cycle. Hence the number of `paint()` calls are reduced.
- Make sure that only the affected pixels are drawn in a screen and not the whole screen.
- If you use only the `GameCanvas.keyStates()` method to query the keys pressed by the user, initialize the `GameCanvas` object using the `GameCanvas(true)` constructor to suppress events listeners generated for game keys.
- Reuse a `GameCanvas` object since a buffer is created for each `GameCanvas` instance. Creating multiple instances will result in multiple buffers, which is bad for performance.
- Call `GameCanvas.getGraphics()` method to get the `Graphics` object BEFORE starting the game and re-use the object again, since each call gets a new instance of the `Graphics` object.
- A `Layer` should not be rendered if it is not visible.
- Use `TiledLayer.setCell(int, int, int, int)`, `TiledLayer.fillCells(int, int, int, int, int, int)` and `TiledLayer.setStaticFileSet(Image, int, int)` methods sparingly as they affect performance.

### Application size reduction

- Amount of work done in Java has been reduced significantly by letting the native code do most of the work. Hence the size of the application will be very small.
- Reuse a `GameCanvas` object to minimize heap usage.

Compare and contrast the use of MIDP's `GameCanvas` class vs. the MIDP low-level `Canvas`.

## GameCanvas

- A dedicated off-screen buffer is created for each `GameCanvas` instance. Whatever is drawn on the canvas is first stored in this buffer and then flushed to the screen by calling the `GameCanvas.flushGraphics()` method.
- Contents of the buffer are modified ONLY by calls to the object and by no-one else (like the system, etc.).
- Buffer size is the maximum size of the `GameCanvas`. But the flush size maybe smaller depending on the presence of other objects on the screen (Ticker, Commands, etc.).
- Calling the `GameCanvas(true)` constructor while creating an instance of `GameCanvas` will only suppress the event listeners for game keys. Event listener methods can be used to listen to other methods. Also the suppression will be effective only when the canvas is visible.
- If keys are pressed while the canvas is being hidden, they will be queued up till the canvas becomes invisible and will be sent to the application when the canvas becomes visible again.
- Some devices MAY NOT detect multiple keys pressed at the same time properly.
- `flushGraphics()` method is a blocking, atomic operation. The application can render the next frame immediately after the return of this method.
- When `flushGraphics(int x, int y, int width, int height)` is called, only the intersecting non-zero region is flushed.
- Flusing the buffer DOES NOT clear the buffer. It only renders the pixels on the screen. The pixels are still there in the buffer.
- Default color of a `Graphics` object is `black` with a `SOLID` stroke, and has the default `Font`.
- Game keys are `UP`, `DOWN`, `LEFT`, `RIGHT`, `FIRE`, `GAME_A`, `GAME_B`, `GAME_C`, and `GAME_D`.
- The return value of the `keyStates()` method can be ANDed (bitwise `&` operator) with the game key constants to find out if a key was pressed or not (If the result is 1, key is pressed).
- If a key is kept pressed while the canvas is being shown (made visible), then that key will not be recorded as pressed. It has to be released and pressed again for the `keyStates()` to take note of the key press.
- Calling the `keyStates()` method will clear the previous state values of all the game keys.

## Differences between GameCanvas and Canvas

- `GameCanvas` has an off-screen buffer to improve performance.
- Extra game key press constants are defined in the `GameCanvas` for the keys described above.
- The new `keyStates()` method can be used to get the state of keys pressed without querying each of them.

Given a set of requirements, develop code using MIDP's `LayerManager` class.

## LayerManager

- `LayerManager` maintains an ordered list of `Layer` objects to be displayed. `Layers` can be added, inserted, or removed from the `LayerManager` when needed.
- The `Layer` at index 0 will be the closest to the user and the `Layer` with the maximum value will be the farthest from the user.
- The indexes are ALWAYS contiguous. If a `Layer` is removed, the others will be rearranged such that the continuity is maintained (much like the behaviour of an `ArrayList`).
- A portion of the `LayerManager` can be made visible through the view window. To pan the `LayerManager`, simply move the view window's co-ordinates. This can be done by invoking the `paint(Graphics g, int x, int y)` method and by changing the `x` and `y` values.
- `append(Layer layer)` can be used to add a `Layer` to the `LayerManager`.
  - The `Layer` appended will have the highest index (farthest from user).
  - If the `Layer` is previously present, it will be removed. If a `Layer x` was at position 2 of a 5-layer `LayerManager`, appending the same `Layer` will result in the `Layer` having a new index of 4 (since the index will reduce when the `Layer` is removed first before being appended).
- `insert(Layer layer, int index)` will insert a `Layer` at the given position. If it is previously present, that `Layer` will be removed.
- `paint(Graphics g)` method will paint the `Layers` in the descending order first (`Layer` with highest index first).
- Only the intersection between the view region and the `Graphics` region will be rendered.
- `Layers` outside the intersection region might not be rendered for performance reasons.
- Default values for the view window are 0, 0, `Integer.MAX_VALUE`, `Integer.MAX_VALUE`.

Given a set of requirements, develop code using MIDP's Layer, Sprite and TiledLayer classes.

## Layer

- This is an abstract class.
- The `paint(Graphics g)` method should be overridden to paint this `Layer`. The implementation should check to make sure that the `Layer` is painted only if it is visible.

## Sprite

- `Sprite` is typically used to display the moving object in a game screen.
- This sub-class of `Layer` can be used to animate images.
- The images in this layer can be flipped (like how your image flips if you stand in front of a mirror - along the vertical axis) and rotated by multiples of 90° (90°, 180°, 270°).
- The frames used by `Sprite` are present in a single `Image` passed while creating the `Sprite` object. The frames can be arranged inside the `Image` in any rectangular format (1x4, 2x2, 4x1, for example). The total width and height of the `Image` SHOULD be a multiple of the width and height of a single frame. All frames should have the same dimensions.
- Frames are numbered starting from 0.
- A frame sequence inside the `Sprite` is the order in which the frames should be rendered. The sequence can be different from the sequence of frames in the `Image`.
- The default frame sequence is the same as the sequence generated from the `Image`. The sequence is generated by counting frames in the `Image` from left-to-right and top-to-bottom.
- Frames should be manually switched using the `nextFrame()` and `prevFrame()` methods. This is circular and so calling `prevFrame()` on the first frame will give the last frame in the sequence and vice versa.
- The size of the sequence need not be the same as that of the total frames. Hence, frames can be repeated, ignored, jumbled, etc.
- A "no-effect" can be created by showing the same frame again and again.
- The `referencePixel` is used to determine the pivot using which image transformations (like rotations) can be made.
- `defineReferencePixel(int x, int y)` defines the pixel which will serve as the reference pixel. `setReferencePixel(int x, int y)` will move the `Sprite` in such a way that that pixel will be displayed in the specified location on the screen.
- There are methods to check if the given `Sprite` collides with an `Image` or a `TiledLayer`. Setting the `pixelLevel` parameter `true` will detect collision only if an opaque pixels of the two objects collide. `Sprite` MUST be visible.
- A collision rectangle defines the area in this `Sprite` that will be used for intersection in the collision detection methods. Default area is the same as that of the `Sprite` itself.
- `getFrameSequenceLength()` gets the frames in the sequence. `getRawFrameCount()` gets the total number of frames available in the `Sprite`. The former may not be equal to the latter.
- The array passed in the `setFrameSequence(int[] sequence)` is copied. Hence changes made to the array after the call will have no impact on the set sequence.
- Calling the `setImage(Image image, int frameWidth, int frameHeight)` will cause a new frame set to be loaded into the `Sprite`.

- If the new `Image` contains more frames than the current `Image`
  - § The current frame remains unchanged.
  - § If the defined frame sequence is a custom sequence, it remains unchanged. If it is default, the frame sequence will be reset to the default sequence of the NEW `Image`.
- If the new `Image` contains less frames than the current `Image`
  - § The current frame will be set to the 0<sup>th</sup> frame.
  - § The frame sequence will be set to the default frame sequence of the new `Image`, regardless of whether the current frame sequence is default or not.
- The reference pixel is unchanged in its definition as well as its location.
  - § If the frame size is different, the top-left co-ordinate will be shifted such that the reference pixel will remain stationary.
  - § If the frame size is different, the collision rectangle will be reset to the new size of the `Sprite` (which would be the size of the frame).

## TiledLayer

- `TiledLayer` is typically used to show the background of a game screen (rolling mountains and stuff).
- The tiles for the object are provided in the same way as in a `Sprite`, through an `Image` containing frames arranged in a rectangular fashion.
- Tiles are indexed, starting from 1. NOTE: This is different from `Sprite` where the indexing starts from 0.
- Tiles can be 'animated' by specifying a virtual index number with a tile and putting it in the sequence. This is useful to show effects like 'rippling water on a lake', etc.
- Animated indexes are always negative, beginning with -1 and should be consecutive. A tile is associated with an animated index by calling the `setAnimatedTile(int animatedIndex, int staticIndex)` method.
- Unlike a `Sprite` which contains a frame sequence, a `TiledLayer` is identified by a grid where the tiles can be placed. The grid dimensions are specified in the constructor.
- Creating an animated tile
  - Set the initial tile using `TiledLayer.createAnimatedTile(int staticTile)` method. This returns the negative 'animation' index for this tile.
  - Set the second tile (to create the animation) for the cell using the `TiledLayer.setAnimatedTile(int animationIndex, int staticTile)`.
  - Example tiles would be one tile having water wave at the top of the image and another tile having water wave at the bottom of the image. Showing them one after the other will create the impression of the wave moving.
- If the index of a cell is 0, then nothing will be displayed on that cell (no image).
- A new set of tiles can be set in the `TiledLayer` by using the `setStaticTileSet(Image image, int tileWidth, int tileHeight)` method.
  - If the number of tiles in the new `Image` is greater than the current one
    - § The animated tiles and the grid will be preserved.
  - If the number of tiles in the new `Image` is less than the current one
    - § All animated tiles will be erased. The grid contents will be reset to 0.