Java™ Platform, Micro Edition

Part 5 – Game API

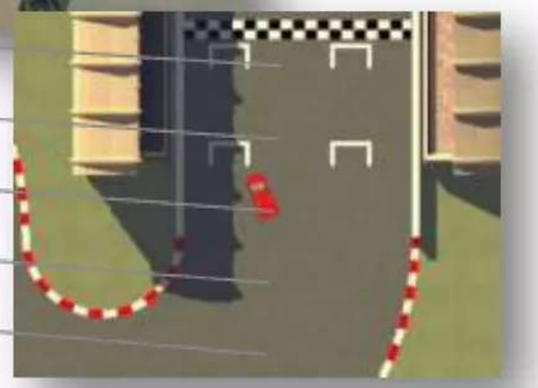
Contents

- Performance
- Game API
  - GameCanvas
  - Bitmaps
  - Sprite
  - Layer and TiledLayer
  - LayerManager

Benchmarks

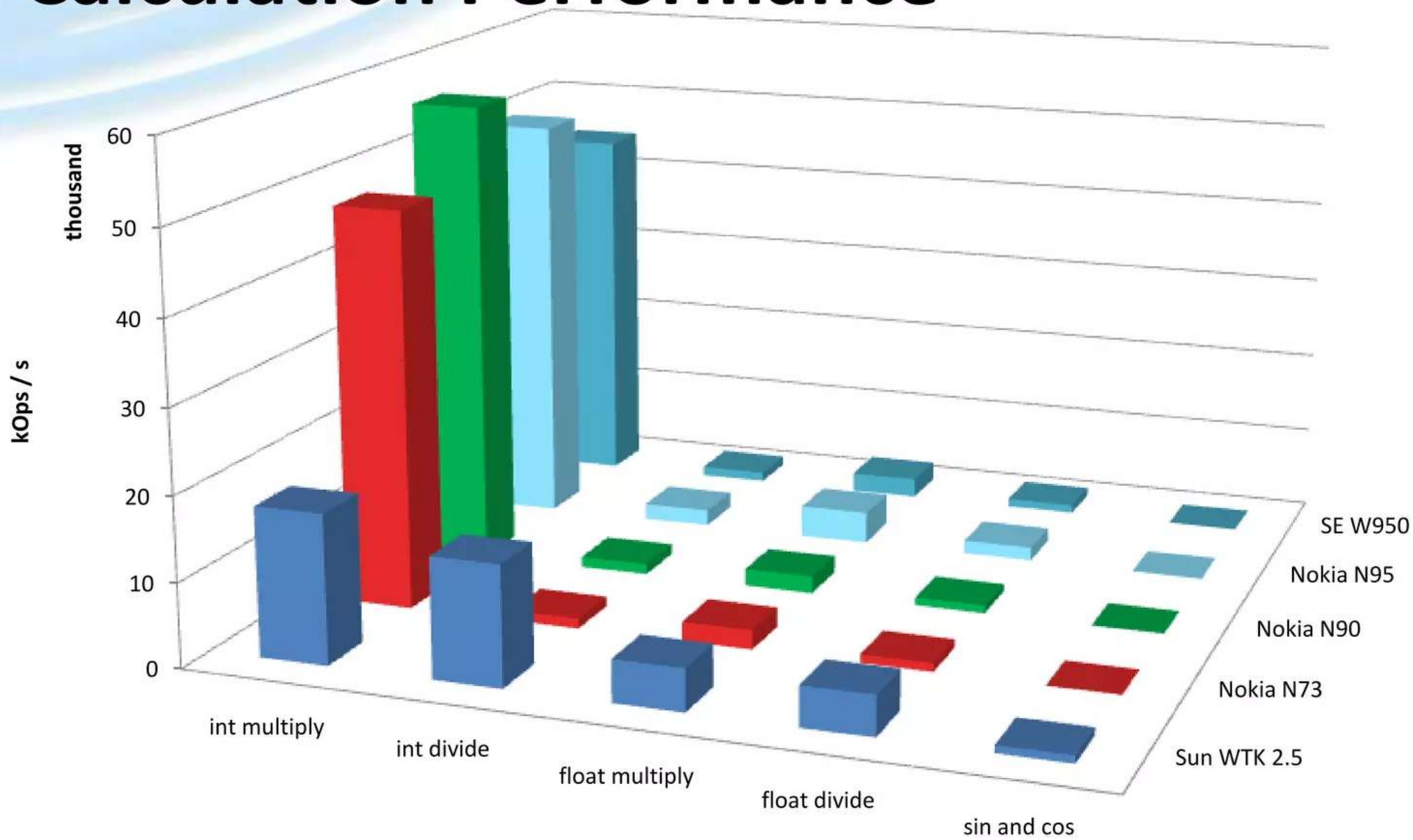
- Take care of **performance**
  - **Multiply** a lot faster than **divide**
  - No hardware **floating point** support
    (mostly SW-emulated, available since CLDC 1.1)
  - **PC-Emulator no reference** for phone performance
  - **Performance of different phones** can differ a LOT
- **Benchmark** results
  - Based on SPMark Java06 Basic Edition
  - http://www.futuremark.com/benchmarks/mobilebenchmarks/

Game Performance



	Sun WTK 2.5	Nokia N90	Nokia N73	Nokia N95	SE W950
2D FPS	2.00	0.80	8.90	29.10	11.80
3D FPS	85.80	7.00	22.20	35.10	18.40

Calculation Performance



	int multiply	int divide	float multiply	float divide	sin and cos
Sun WTK 2.5	17730	14284	5021	4750	894
Nokia N73	47637	1170	2318	1005	25
Nokia N90	56114	1265	2072	932	10
Nokia N95	50100	1958	3856	1619	41
SE W950	44614	1122	2362	984	25

Optimized for game loops

GameCanvas

Games using a Canvas

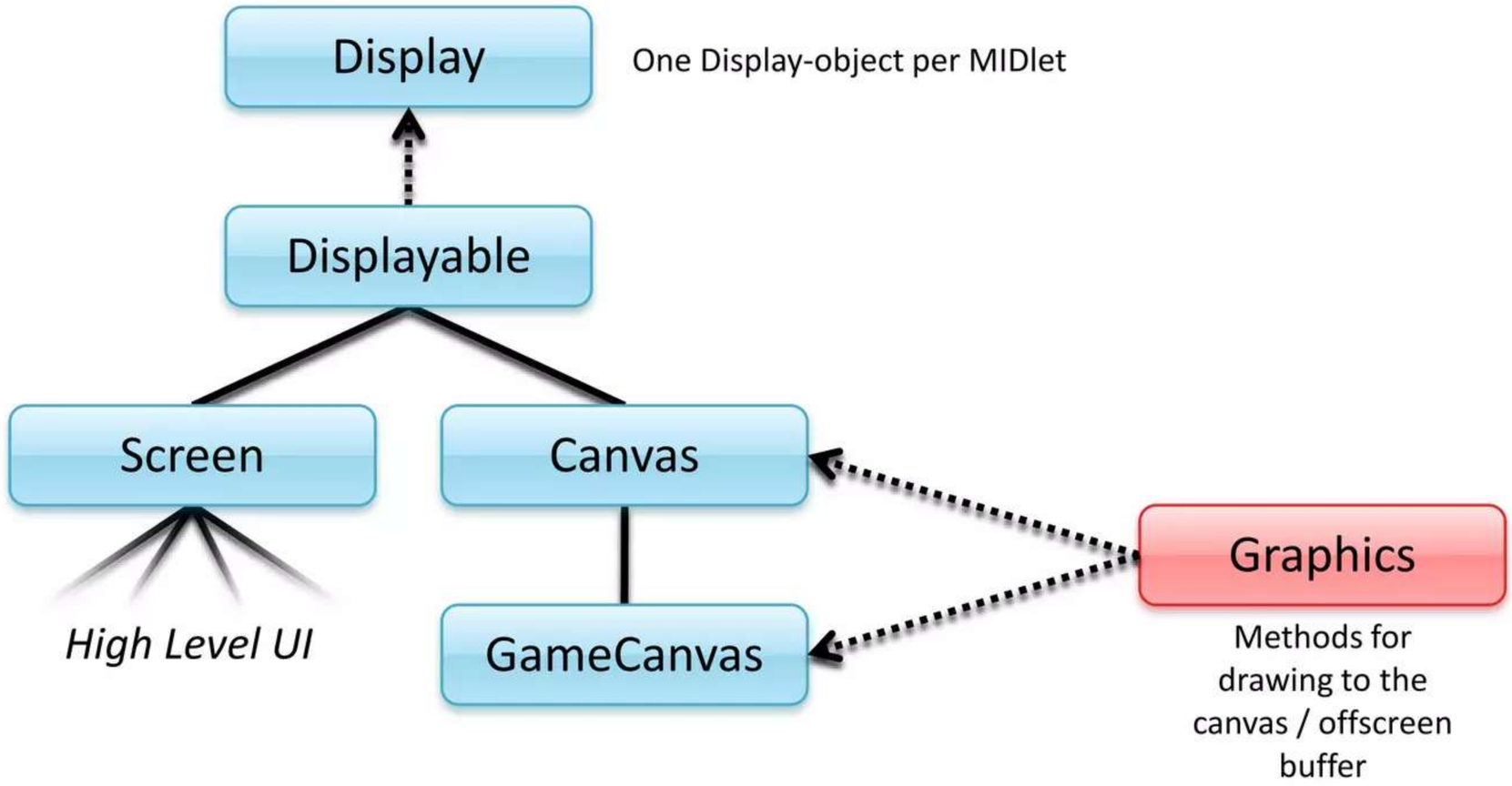
```
public void myCanvas extends Canvas implements Runnable {
        public void run() {
                // Main game loop
                while (true) {
                        ...                     // Do processing
                        repaint();              // Request repaint
                }
        }

        public void paint (Graphics g) {
                // Painting code
        }

        protected void keyPressed (int keyCode) {
                // Handle user input
        }
}
```

Thread

Thread

Thread

GameCanvas


Display
One Display-object per MIDlet
Displayable
Screen
Canvas
Graphics
High Level UI
GameCanvas
Methods for drawing to the canvas / offscreen buffer

Canvas vs. GameCanvas

- **Canvas**
  - **Keys:** sent as events, asynchronous processing
  - **Drawing:** only in `paint()`-method, own thread
  - Suited for event-based games and applications
- **GameCanvas**
  - **Keys:** query state in the game loop for every iteration
  - **Drawing:** everywhere, graphics shown when calling `flushGraphics()` from GameLoop
    - (Both new features are optional in the `GameCanvas`!)
  - Suited for games

GameCanvas Construction

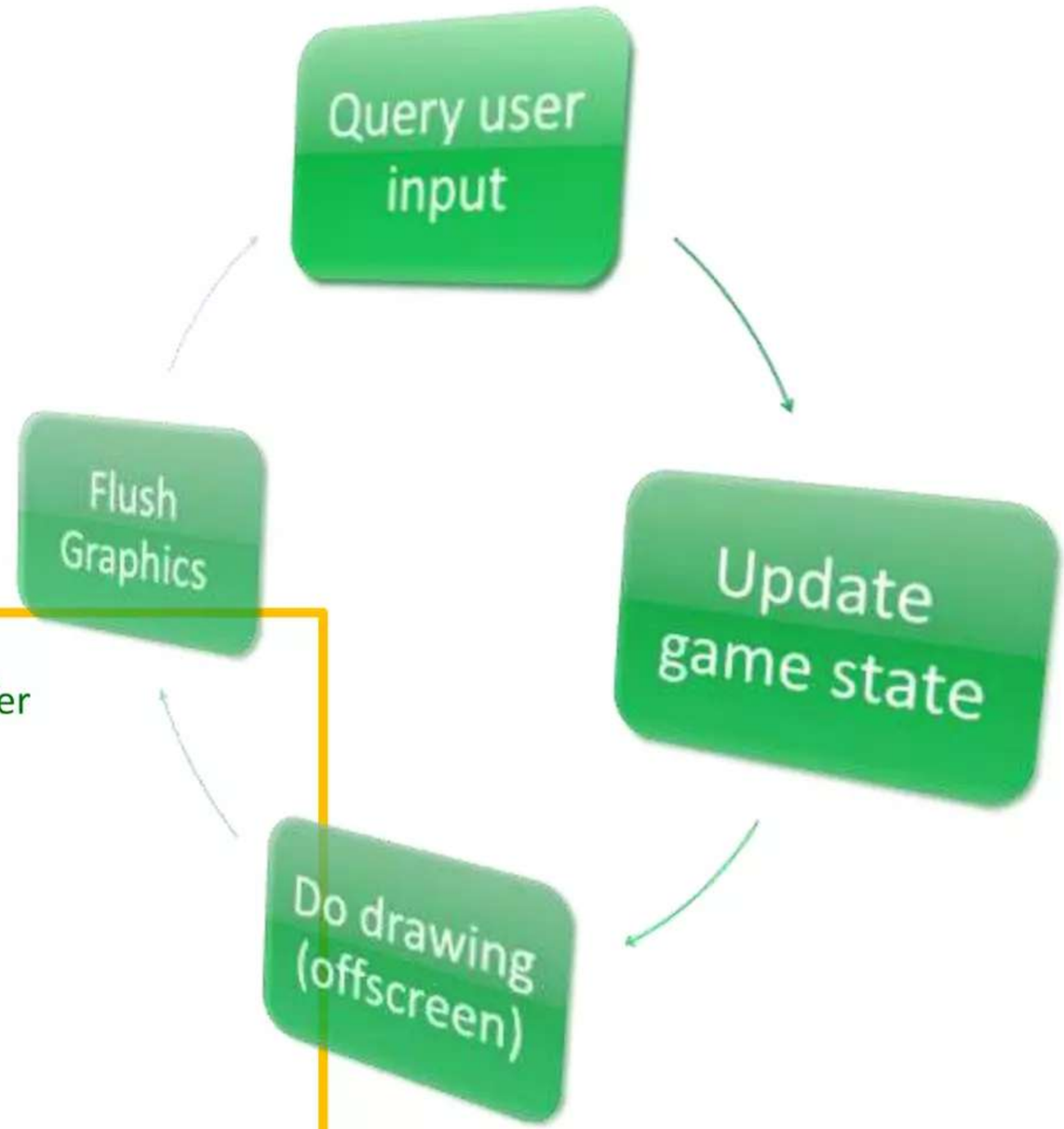
- **Key Event delivery**
  - Call constructor of GameCanvas base class in first line of own constructor
  - **Requires boolean parameter:**
    - **true:** if using getKeyStates() to query currently pressed keys. Suppresses key event delivery to keyPressed(), keyRepeated() and keyReleased() → increases performance.
    - **false:** deliver key events to the methods stated above, like for normal Canvas.
- **Full screen** (also available for Canvas)
  - Call setFullScreenMode(bool) for (de)activating full screen mode
  - Hides softkey and title area
  - **Note:** UI guidelines usually require you to highlight at least one softkey that can be used for exiting the application, even during a game!

Games using a GameCanvas

```
public void myGameCanvas extends GameCanvas implements Runnable {
        public PlayCanvas()
        {
                super(true);                    // Suppress key events -> query with getKeyStates()
                setFullScreenMode(true);  // Use full screen mode for the game
        }
        public void startGame() {
                Thread t = new Thread(this);
                t.start();                      // Start the game loop in its own thread
        }
        public void run() {
                Graphics g = getGraphics();
                // Main game loop
                while (true) {
                        int keyStates = getKeyStates();
                        // ... process key input and update the world ...
                        flushGraphics(); // Method from GameCanvas base class -> do not override!
                        Thread.sleep(20);       // Make sure the other system threads get some time
                }
        }
}
```

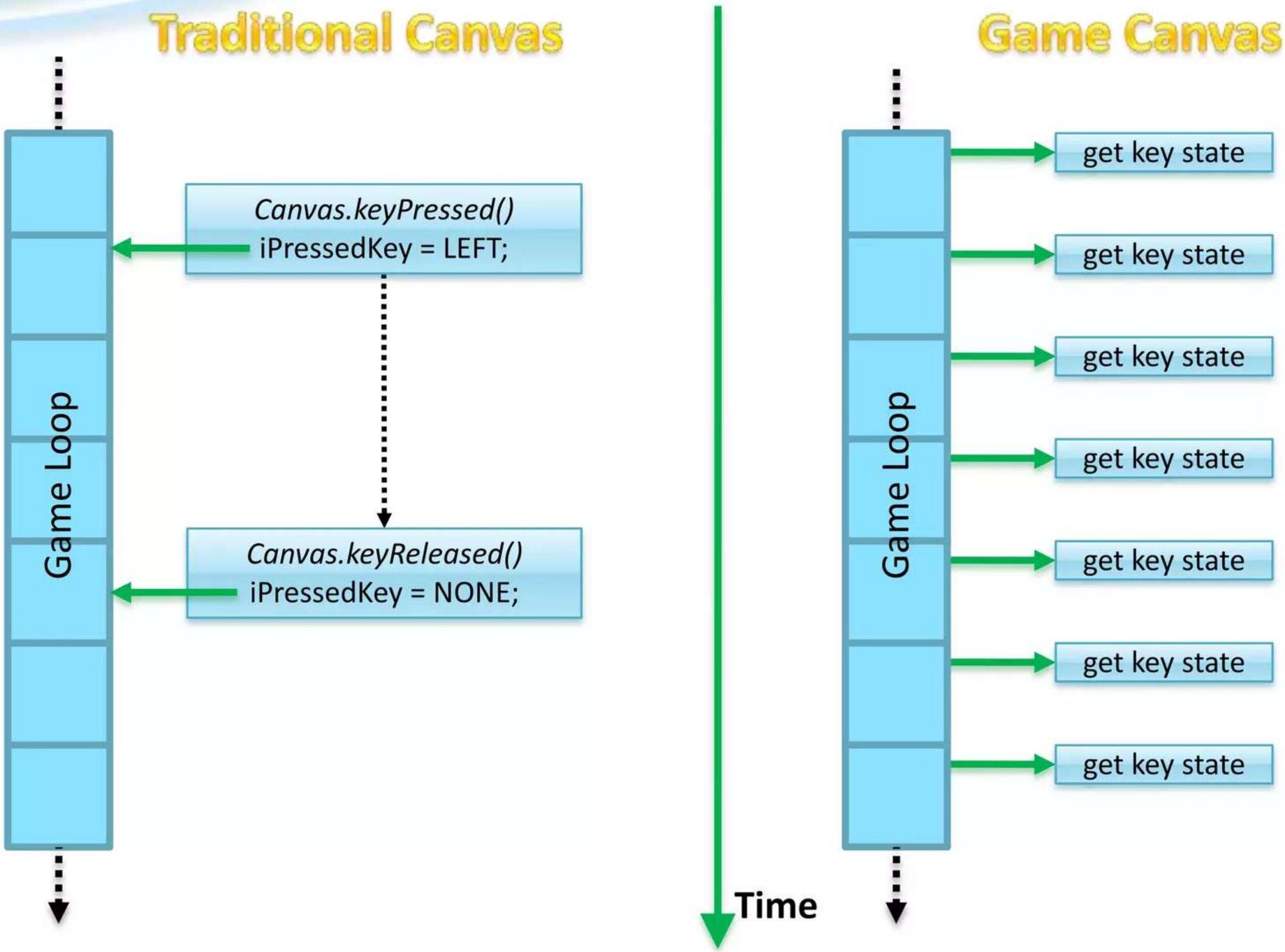
Thread

Game Loop




```
public void run() {
        // Get the Graphics object for the off-screen buffer
        Graphics g = getGraphics();

        while (true) {
                // Query user input
                …
                // Update game state
                …
                // Do drawing
                …
                // Flush the off-screen buffer
                flushGraphics();    // Method from GameCanvas base class
                                    // -> do not override!
        }
}
```

Polling Input



Advantages of Key Polling

- **Multiple keypress**
  - More than one button held **simultaneously**? (if supported by the hardware)
  - Query **all buttons** in one call
- Update state at **defined position**
  - Easier management of action handling
- **Performance**
  - May **increase** – no event handling required

Game Actions

keyStates	0	0	0	1	0	0	1	0	0	0	0	0	0
Position	12	11	10	9	8	7	6	5	4	3	2	1	0

GAME_D_PRESSED

GAME_C_PRESSED

GAME_B_PRESSED

GAME_A_PRESSED

FIRE_PRESSED

DOWN_PRESSED

RIGHT_PRESSED

LEFT_PRESSED

UP_PRESSED

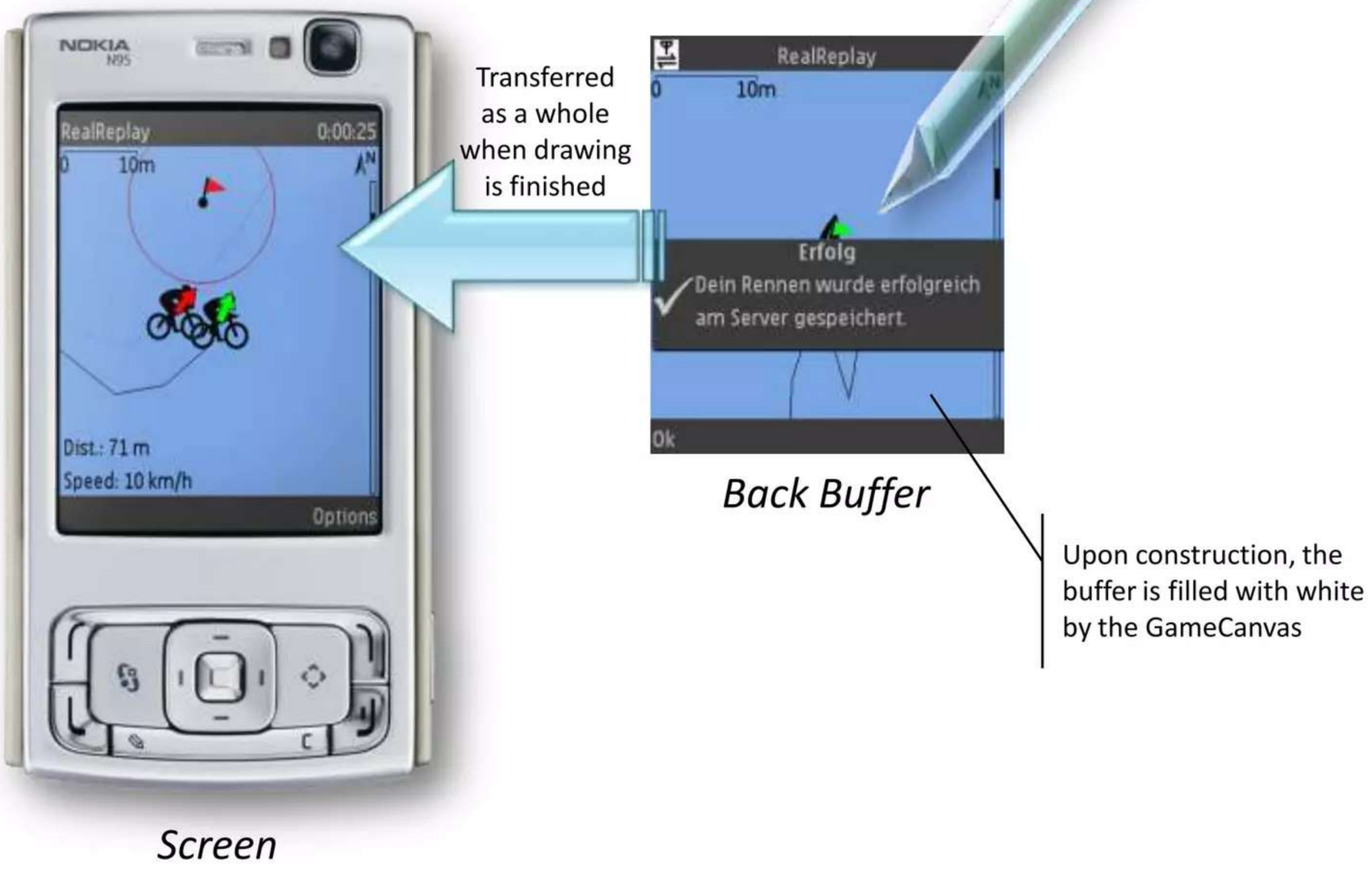
```
// Get state of all keys
int keyStates = getKeyStates();
// Handle all required keys
if ((keyStates & UP_PRESSED) != 0) {
        // Handle up...
} else if ((keyStates & LEFT_PRESSED) != 0) {
        // Handle left...
} else if ...
```

Using

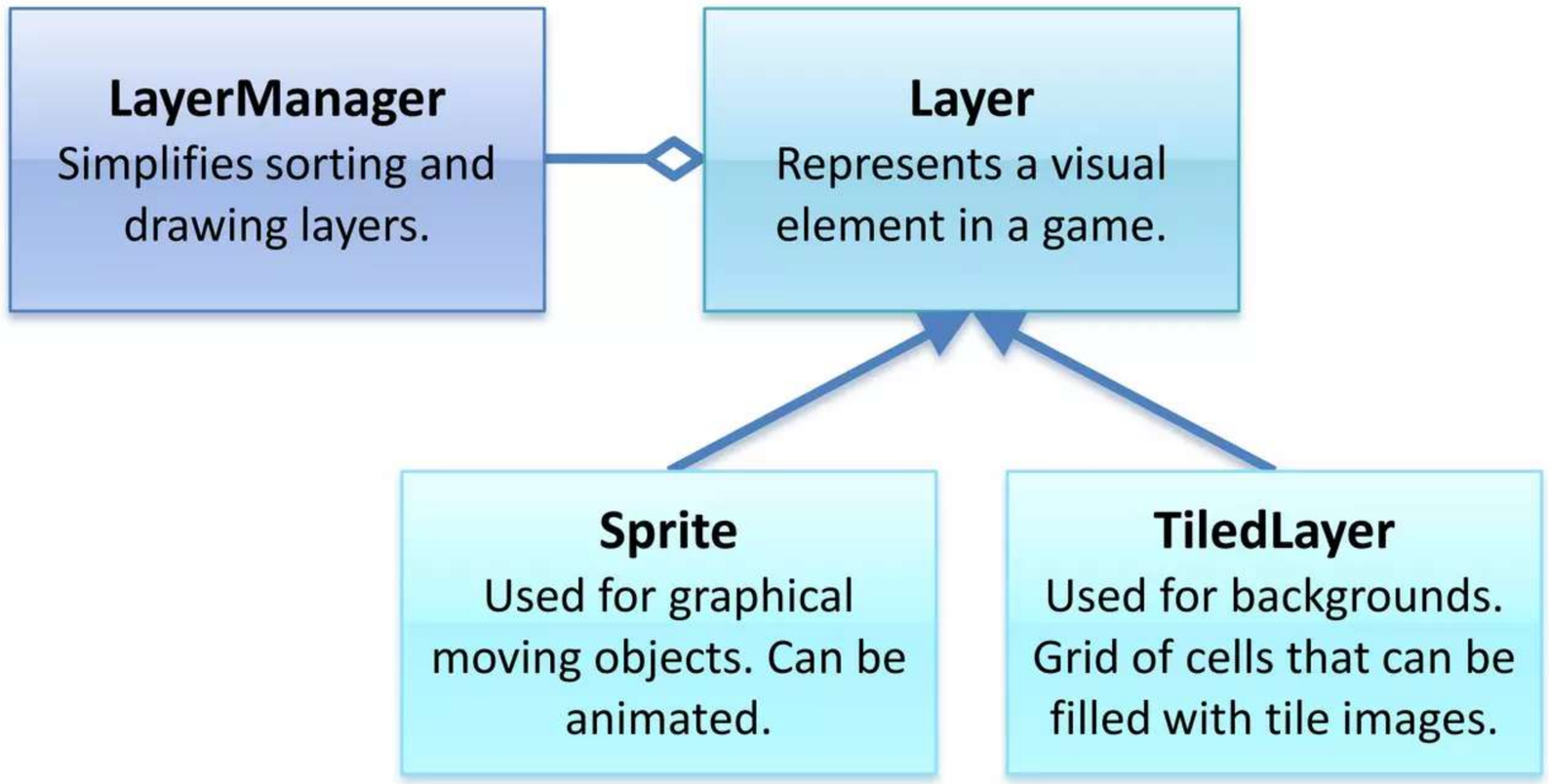
Game Graphics

Double Buffering

- Supported by the GameCanvas



Utility Classes for Graphics

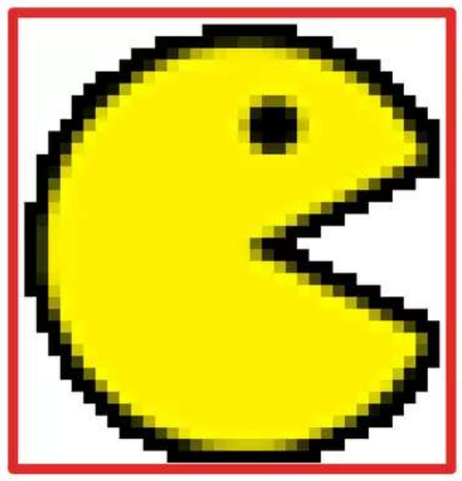


Layer

- Abstract **base class** for a visual element
- Defines **paint()**-method
- **Properties:**
  - Position (upper-left corner)
  - Size
  - Visibility

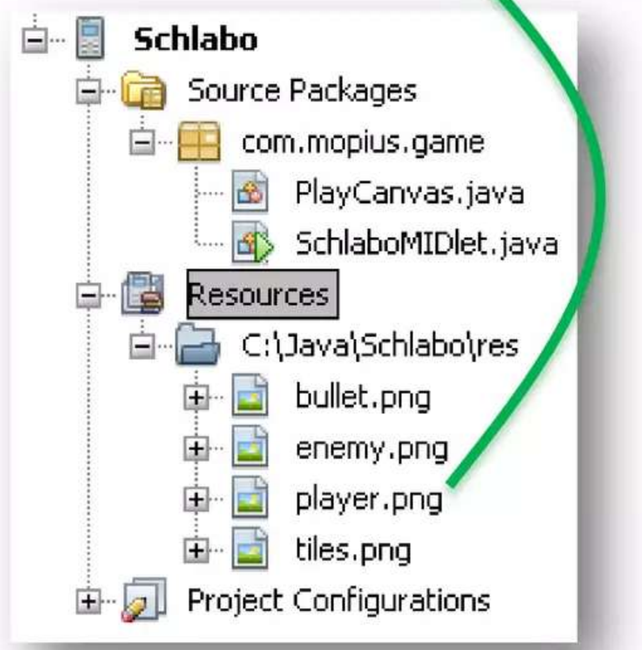
Sprite

- Pre-rendered 2D figure, usually with transparency, integrated into larger scene
- **Early video gaming**
  - Sprite: hardware feature built into graphics subsystem
  - Limited number of sprites on the screen (GBA, SNES, ...)
- **Software-based Sprite in Java ME**
  - Drawing
  - Moving, rotating, flipping
  - Animation
  - Collision detection



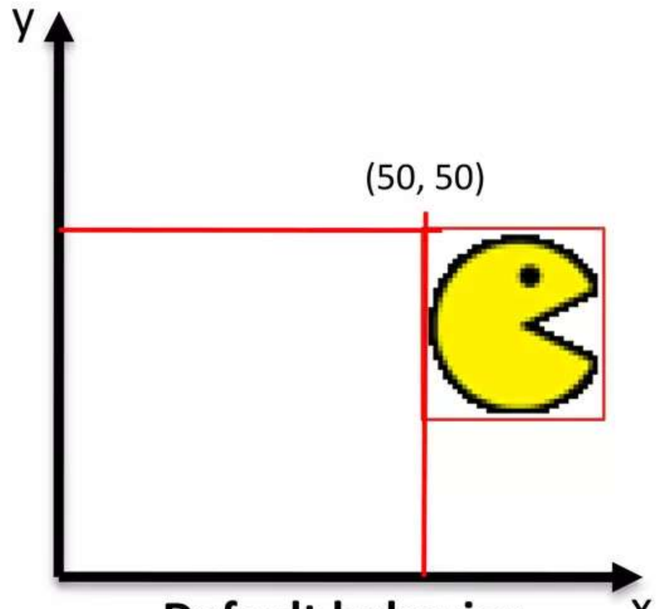
Loading

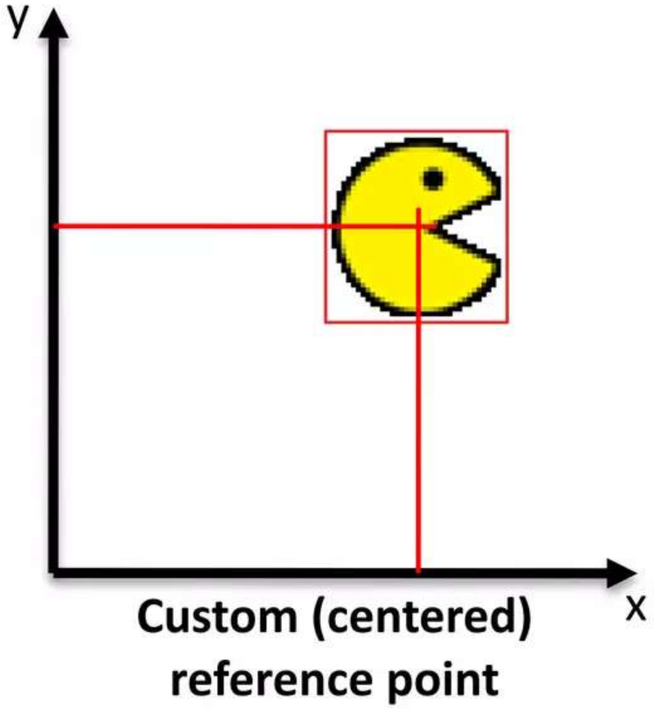
- **Create using:**
  - Image img = Image.createImage(“/player.png”);
    Sprite mySprite = new Sprite(img);
- **NetBeans 6+:**
  - Add a resource folder to the project



Reference Pixel

- **Default**
  - Positions reference sprite at its upper left corner
- **Custom anchor point**
  - **Define reference point:**
    - Relative to un-transformed upper left corner of sprite
    - May be outside of sprite's bounds
    - mySprite.defineReferencePixel(int x, int y);
  - **Position sprite:**
    - Reference pixel is located at pos x/y on painter's coordinate system
    - mySprite.setRefPixelPosition(int x, int y);
  - **Query sprite position:**
    - In the painter's coordinate system
    - int x = mySprite.getRefPixelX(); // getRefPixelY()





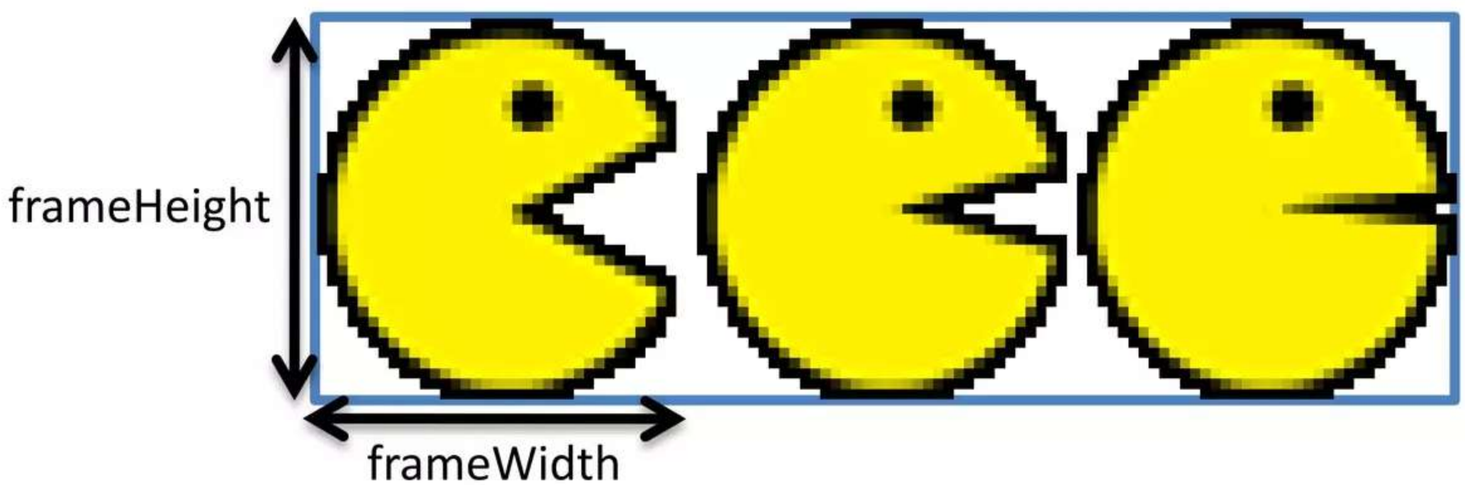
Transformations

- Possible in 90°-steps, no smooth rotation
  - mySprite.setTransform(int transform);

Transformation	Result
TRANS_NONE	
TRANS_ROT90	
TRANS_ROT180	
TRANS_ROT270	
TRANS_MIRROR	
TRANS_MIRROR_ROT90	
TRANS_MIRROR_ROT180	
TRANS_MIRROR_ROT270	

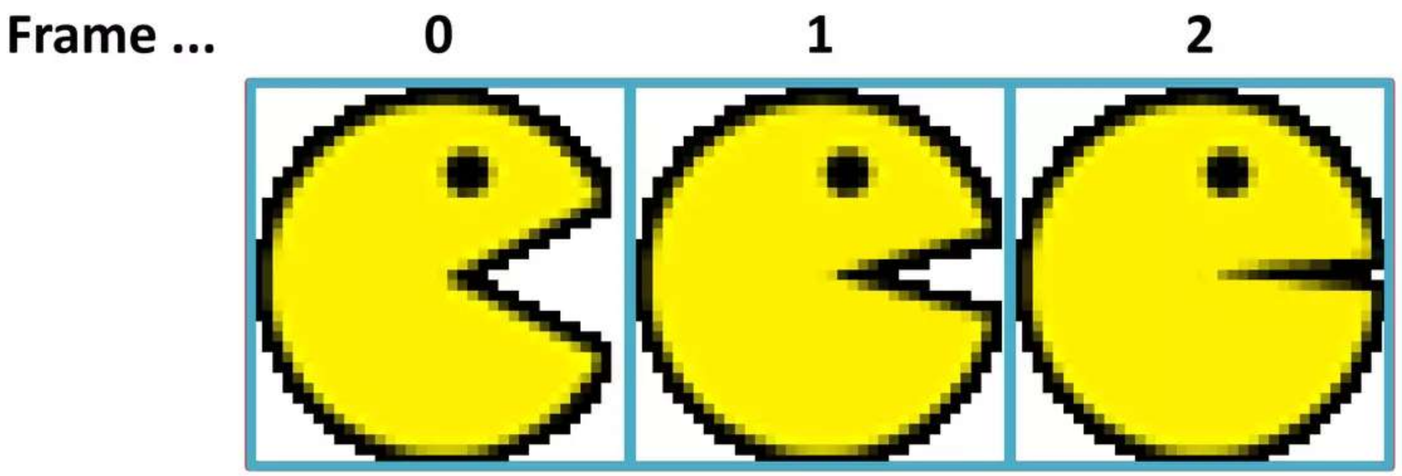
Animation

- Several instances of the sprite that are slightly different
- All frames have the **same size**
- Only **one frame displayed** at the same time
- **Create using:**
  - Sprite mySprite = new Sprite(img, int frameWidth, int frameHeight);



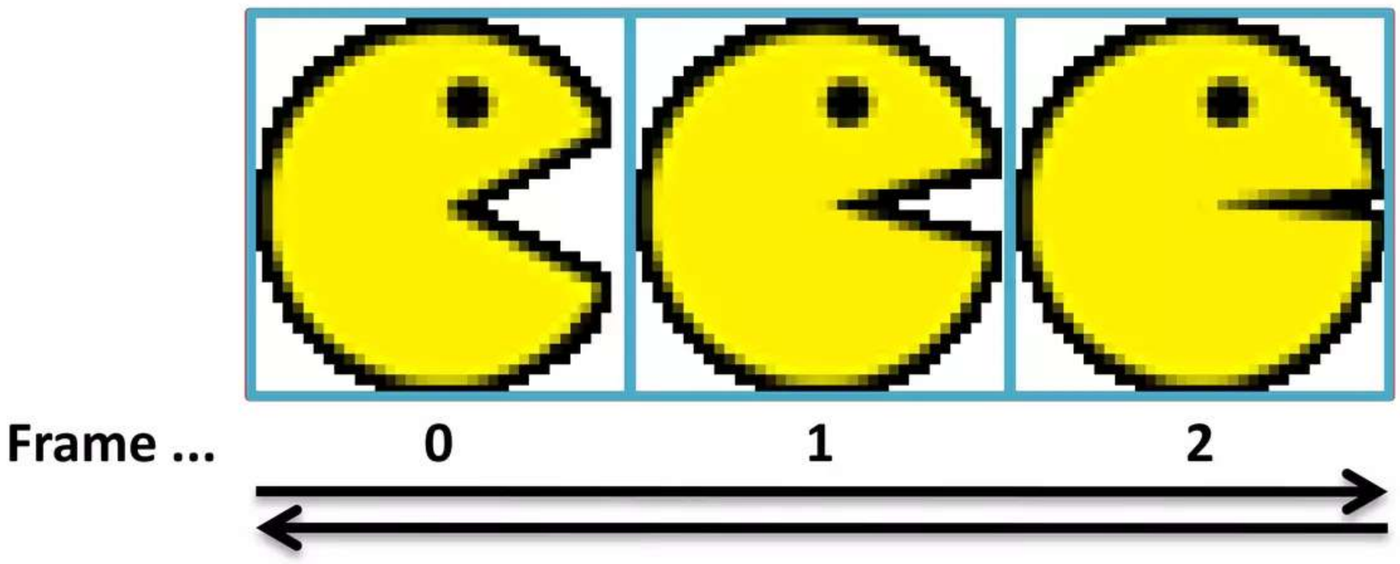
Animation – Frames

- **Set frame to display**
  - mySprite.setFrame(int sequenceIndex);



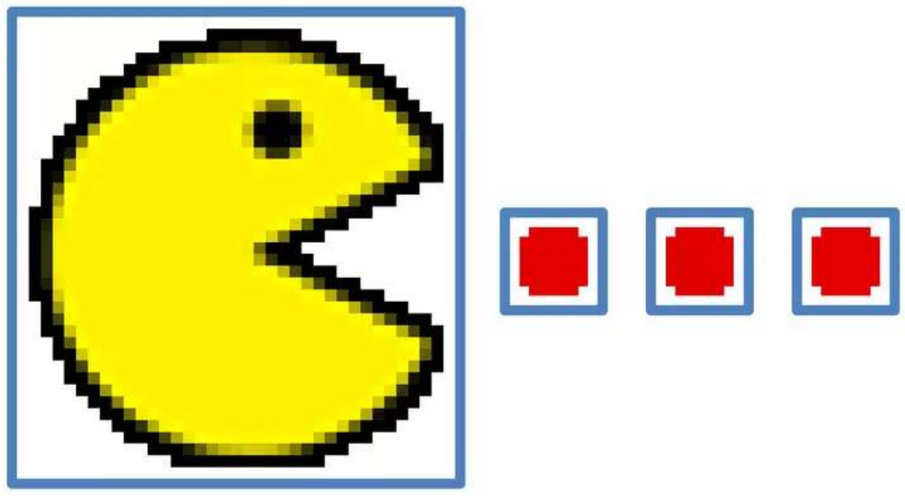
Animation – Frame Sequence

- For continuous animations:
  - int[] frameSequence = {0, 1, 2, 1};
    mySprite.setFrameSequence (frameSequence);
    …
    mySprite.nextFrame();
- Wraps automatically



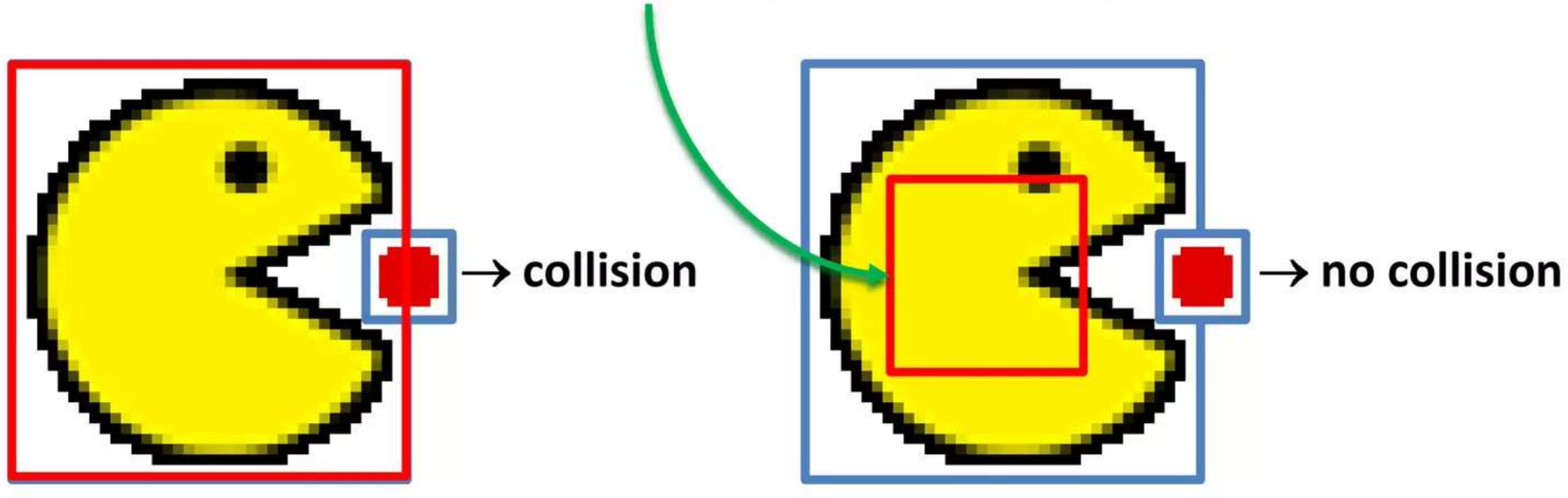
Collision

- Checking for collision between a Sprite and ...
  - another Sprite
  - an Image
  - a TiledLayer



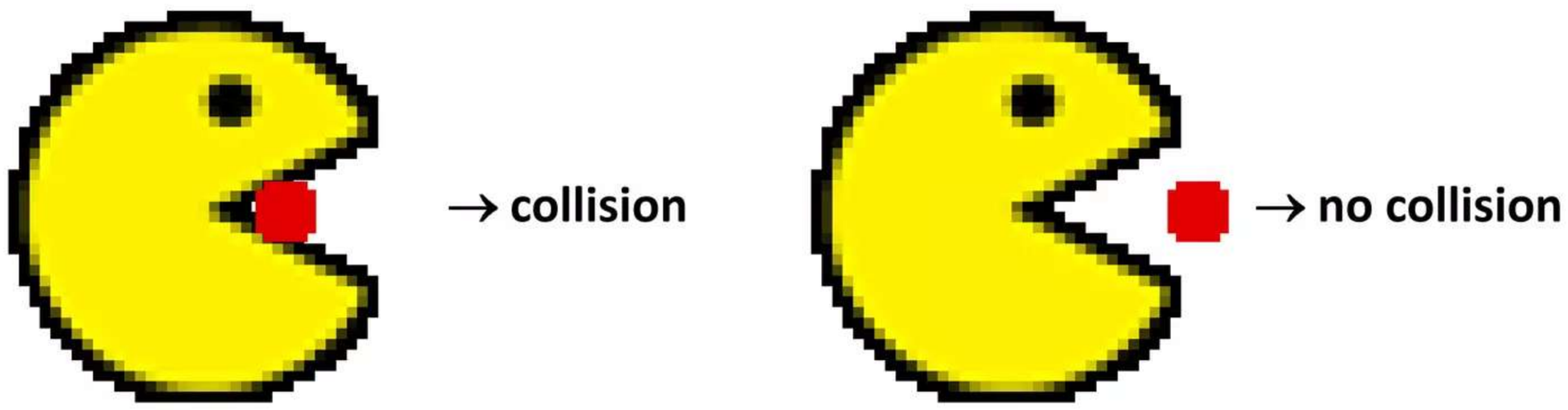
Rectangle Collision

- Checks collision based on **frame boundary**
  - if (mySprite.collidesWith(yourSprite, **false**)) { ... }
- **Fast** and efficient
- Possible to define collision rectangle size
  - mySprite.defineCollisionRectangle (int x, int y, int width, int height);



Pixel Collision

- Checks for overlapping, non-transparent pixels
  - if (mySprite.collidesWith(yourSprite, **true**)) { ... }
- Considerably more expensive than rectangle check



Create beautiful backgrounds

TiledLayer

TiledLayer

- For displaying **background graphics**
- Made of **repeating elements** (=tiles)
  - Create variety of backgrounds with less space
- What is a **tile**?
  - **Square bitmap**
  - Arranged in “**tile maps**”

Diamond Rush
Copyright 2006 Gameloft

How does it work?

- **Original image** (contains all tiles):

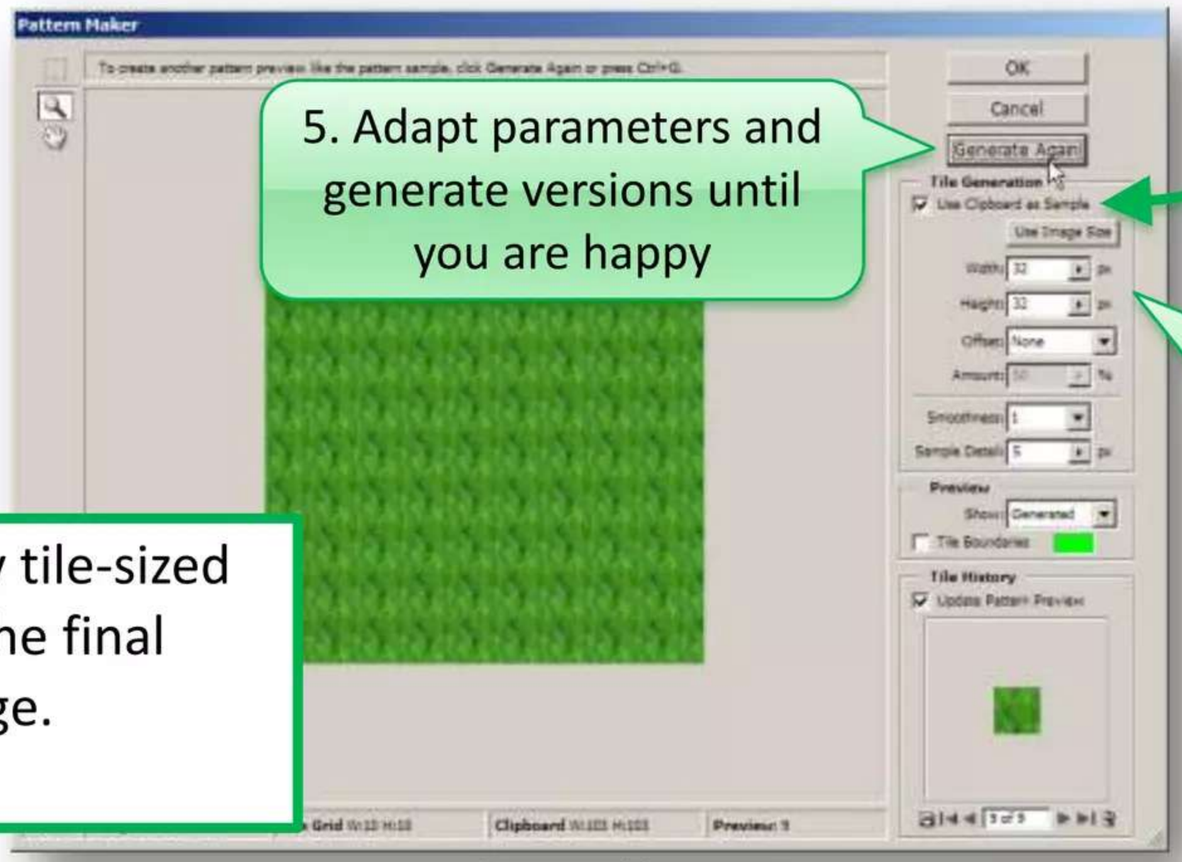
- **Tile Map:**

```
private static final char bg_map [] =
{
4, 14, 1, 3, 5,
10, 8, 4, 7, 5,
3, 2, 2, 9, 11,
12, 6, 6, 11, 1,
1, 1, 1, 1, 1
};
```

Creating Tileable Tiles

Non tileable graphic:

1. Create **empty image** in Photoshop (eg. 300x300 px)
2. Copy part of a **texture** to the **clipboard**
3. Filter → **Pattern Maker**...



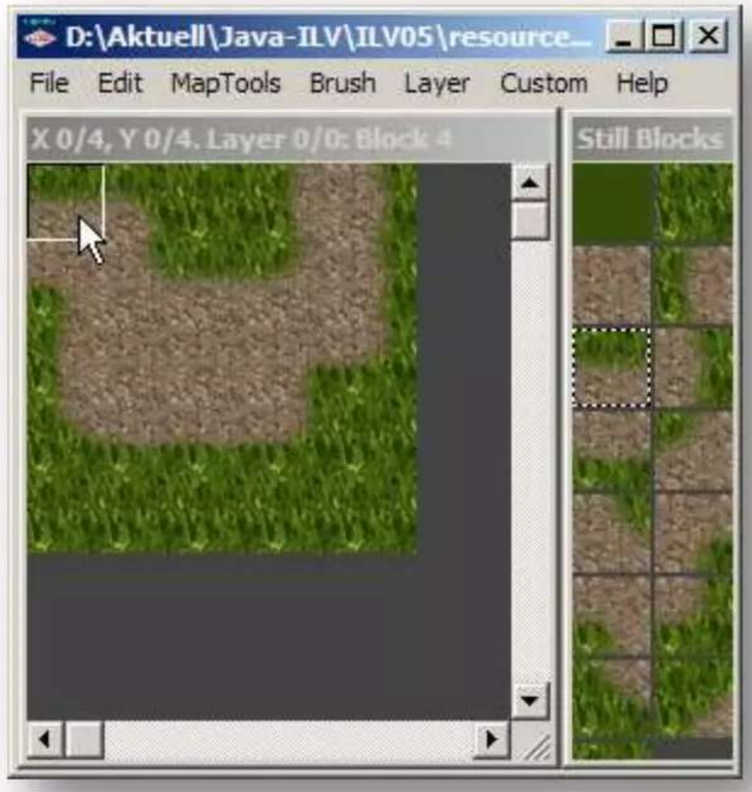
5. Adapt parameters and generate versions until you are happy

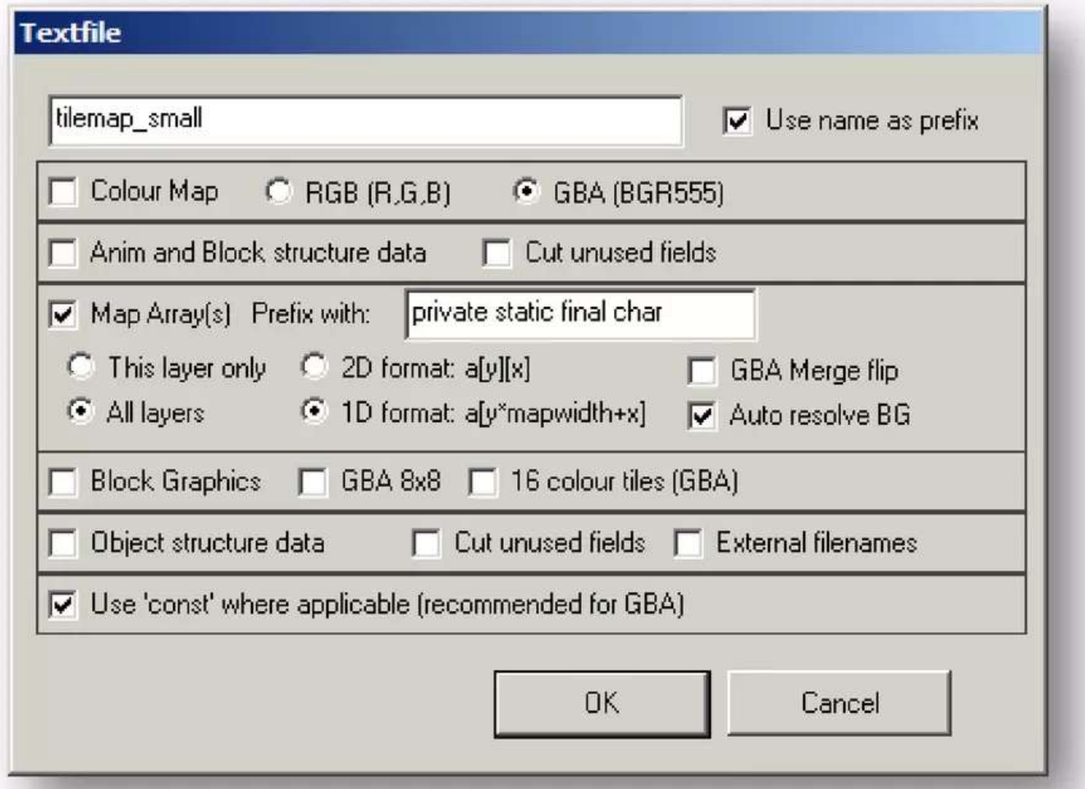
4. Specify width and height of the tiles you want to create (eg. 32x32)

6. OK; then copy any tile-sized part (eg. 32x32) of the final image to a new image.
This is your tile!

Creating a TileMap

- **Free tool: Mappy (or use NetBeans GameBuilder)**
  - http://www.tilemap.co.uk/ (install .png-support as well)
  - **Export as text file** for use with JavaME





The JavaME-Side

1. **Create TiledLayer-Object**
   - TiledLayer myBg = **new TiledLayer** (int colums, int rows, Image img, int tileWidth, int tileHeight);
   - All **tiles** in the image have the **same size**
     - imageWidth = multiple of tileWidth
     - imageHeight = multiple of tileHeight

- **Set Tile Map**
  - **Single cell:** myBg.**setCell** (int col, int row, int tileIndex);
  - **Fill area:** myBg.**fillCells** (int col, int row, int numCols, int numRows, int tileIndex);

tileIndex 0 = empty (transparent) cell

Assign Tile-Array

- **Assign 1D Tile-array to tile map:**

```
final int tileCols = 5;                    // Columns (width) of the tile map
final int tileRows = 5;                    // Rows (height) of the tile map
final int tileSizePixelsW = 32;            // Width (pixels) of the individual tiles
final int tileSizePixelsH = 32;            // Height (pixels) of the individual tiles
TiledLayer bgLayer = new TiledLayer (tileCols, tileRows, Image.createImage ("/res/tiles.png"),
                                     tileSizePixelsW, tileSizePixelsH);
for (int i = 0; i < bg_map.length; i++)    // Go through 1D array
{
   int column = i % tileRows;              // Current column
   int row = (i - column) / tileCols;      // Current row
   bgLayer.setCell (column, row, bg_map[i]);   // Assign cell
}
```

Drawing

- **Manually drawing** the TileMap (without LayerManager)
  - bgLayer.paint(Graphics g);
- **Move layer** (scrolling, ...)
  - bgLayer.move(int xOffset, int yOffset);
  - eg. calling *move(-3, 0);* moves layer 3 pixels to the right

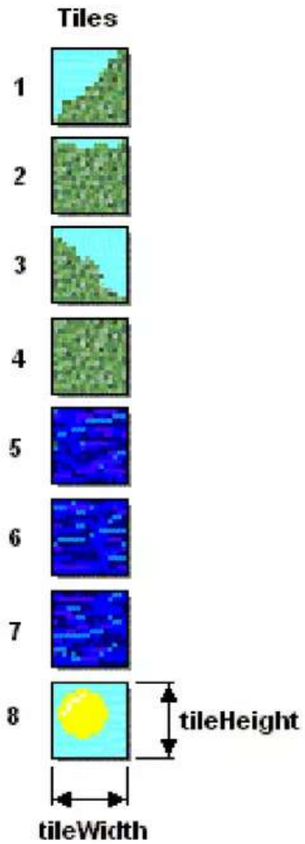
Animating Tiles

- For effects like water, wind brushing through trees, etc.

1. **Generate animated tile index**
   - int animTileIndex = bgLayer.**createAnimatedTile**(1);
   - **Parameter:** Initial static tile index that will be displayed
   - **Returns negative index**, this can be **assigned to cells** that should be **animated**
2. **Switch to different tile**
   - bgLayer.**setAnimatedTile**(animTileIndex, 2);
   - Causes static tile 2 to be displayed instead of 1 for all cells that have the animTileIndex as content

Animated Tiles – Example

1. Tiles:



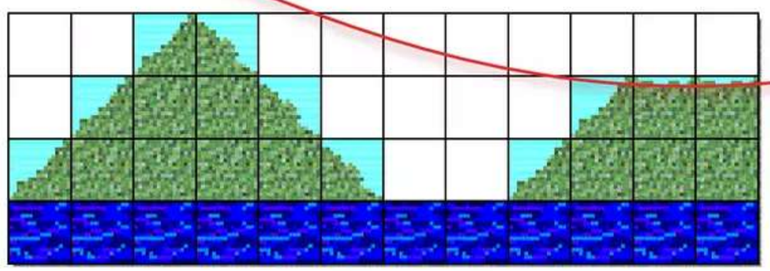
2. Array used to create the tile map:

Cells

0	0	1	3	0	0	0	0	0	0	0	0
0	1	4	4	3	0	0	0	0	1	2	2
1	4	4	4	4	3	0	0	1	4	4	4
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Animated Tiles

-1	=	7



3. Code to create animated tiles:

```
// Indicate that the tile should be animated
int animatedIndex = bg.createAnimatedTile(7);
// animatedIndex should be -1 (first assignment)
// -> Java ME will display tile 7 instead of -1
```

4. Code to animate tiles:

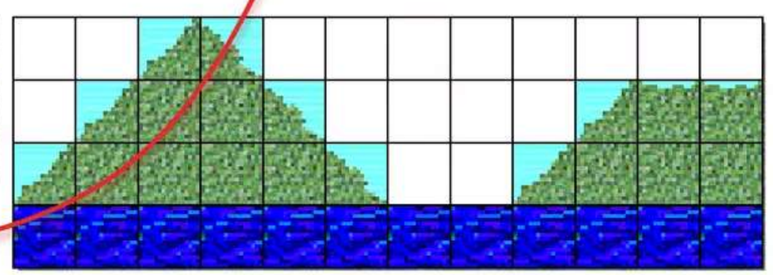
```
// Switch the animated tile
bg.setAnimatedTile(animatedIndex, 5);
// -> Java ME will display tile 5 instead of 7
```

Cells

0	0	1	3	0	0	0	0	0	0	0	0
0	1	4	4	3	0	0	0	0	1	2	2
1	4	4	4	4	3	0	0	1	4	4	4
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Animated Tiles

-1	=	5



Images taken from the Sun Java ME documentation

Don’t want to do all the work yourself?

LayerManager

LayerManager

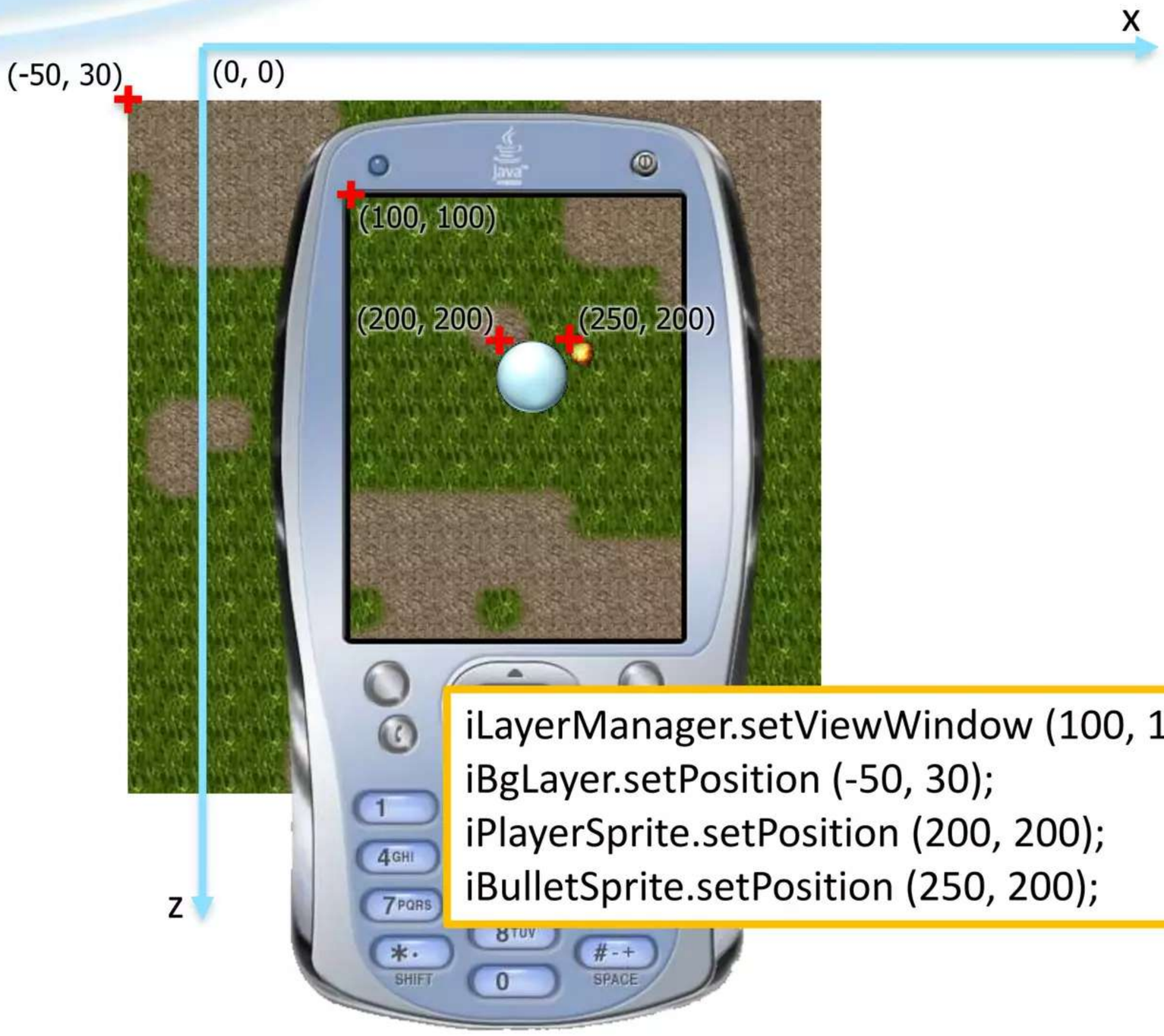
- Manages **multiple layers**
  - Sprites, TiledLayers and own classes derived from those
- **Paints all layers** with a single call to LayerManager.paint(Graphics g, int x, int y)
- **Manages z-order** of layers:

```
iLayerManager = new LayerManager ();
iLayerManager.append (iPlayerSprite);
iLayerManager.append (iEnemySprite);
iLayerManager.append (iBulletSprite);
iLayerManager.append (iBgLayer);
```

z-order	Layer
0	iPlayerSprite
1	iEnemySprite
2	iBulletSprite
3	iBgLayer

visibility ↓ / draw order ↑

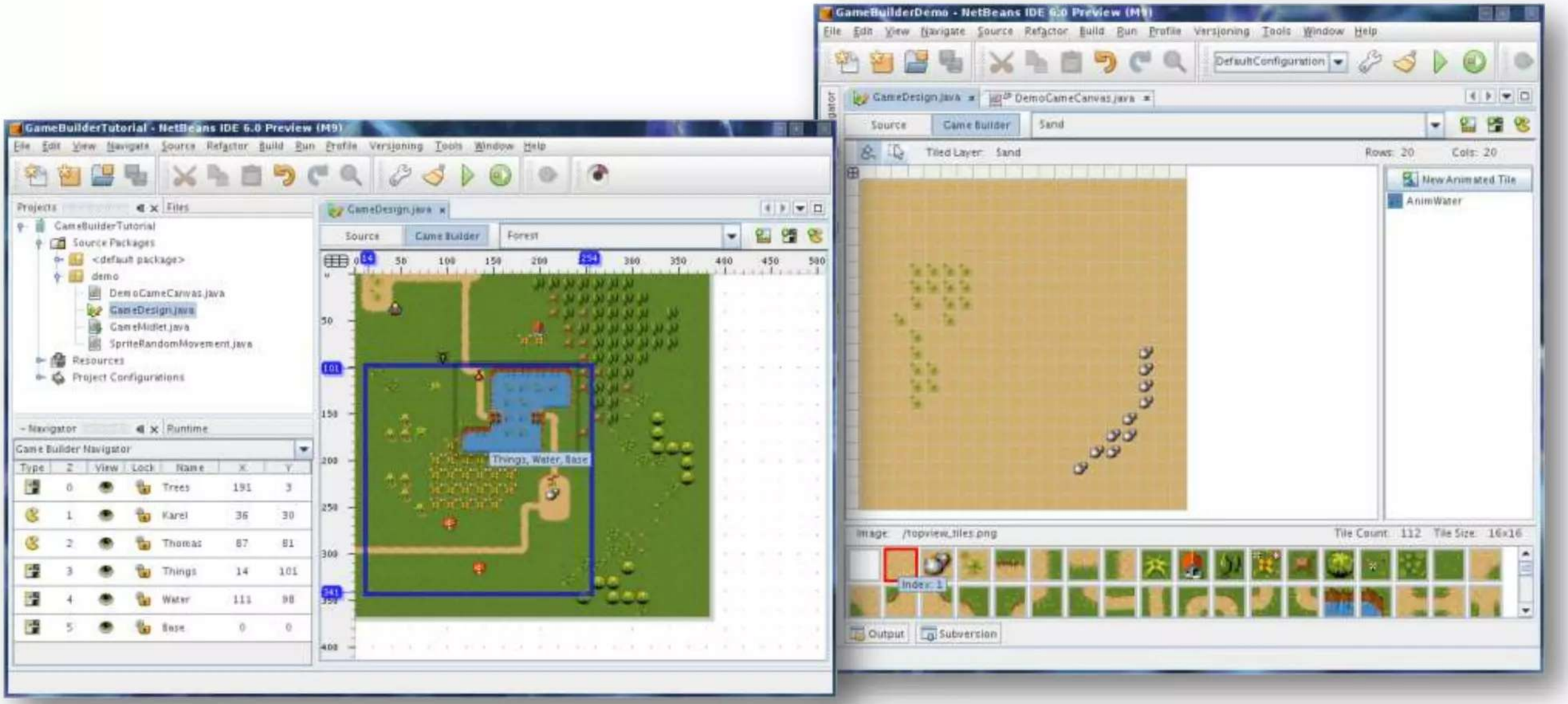
Positioning Layers




```
iLayerManager.setViewWindow (100, 100, iWidth, iHeight);
iBgLayer.setPosition (-50, 30);
iPlayerSprite.setPosition (200, 200);
iBulletSprite.setPosition (250, 200);
```

NetBeans GameBuilder

- NetBeans 6+ has got an integrated game editor



That’s it!

Thanks for your attention