Spill og grafikk

Mildrid Ljosland, Institutt for informatikk og e-læring ved NTNU
Lærestoffet er utviklet for faget IFUD1042 Applikasjonsutvikling for Android

Resymé: Denne leksjonen tar for seg teknikker for å tegne på skjermen. Spesielt i forbindelse med spill er det viktig at dette kan gå fort, og gjerne i en egen tråd.

8.1 Om denne leksjonen

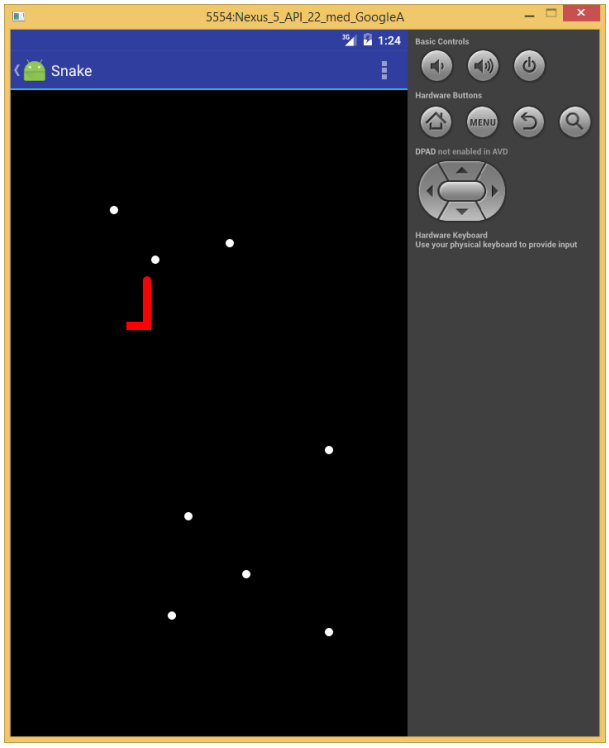
I denne leksjonen skal vi se nærmere på hvordan man tegner på skjermen og teknikker for å lage spill som krever rask oppdatering av skjermen. Enkel tegning behandles i kapttel 12 i læreboka. Dette kapitlet kan du lese før du starter på denne leksjonen.

Vi skal illustrere en del teknikker ved å se på to eksempler på spillet Snake, først en selvlaget utgave, deretter en versjon som følger med Androidpakken. Det følger ikke med noen øving til denne leksjonen, i stedet skal dere begynne å jobbe med prosjekt-øvingen.

I den selvlageted versonen av spillet bruker vi klassene SurfaceView og SurfaceHolder. Du finner litt om disse i lærebokas kapittel 19, men da i forbindelse med bruk av kamera. I tillegg kan det være lurt å lese en del på nettet.

8.2 SurfaceView og SurfaceHolder

Spillet Snake, som er gammelt, velkjent dataspill, går ut på å få en orm til å vokse ved å spise mat som dukker opp rundt omkring på skjermen. For hver matbit ormen sluker, vokser den med ett ledd. Det gjelder å få den så lang som mulig uten at den kolliderer verken med sidekantene av skjermen eller med seg selv.



I det første eksemplet brukes tre klasser (pluss enkle indre klasser). Disse tre klassene er subklasser av henholdsvis Activity, SurfaceView og Thread.

Det første vi skal se på er komponenten [SurfaceView] som er en subklasse av View. Dette er en klasse som er spesiallaget for å brukes til å tegne på. La oss se på hvordan dette kan brukes. Vi lager en ny klasse SnakeView som en subklasse av SurfaceView:

```
class SnakeView extends SurfaceView implements SurfaceHolder.Callback {
    public SnakeView(Context context, AttributeSet attrs) {
        super(context, attrs);
        // register our interest in hearing about changes to our surface
        SurfaceHolder holder = getHolder();
        holder.addCallback(this);
    }

}
```

getHolder() er en metode som gir oss tilgang til den bakenforliggende overflaten som vi skal tegne på. [SurfaceHolder] beskrives i dokumentasjonen slik:

"Abstract interface to someone holding a display surface. Allows you to control the surface size and format, edit the pixels in the surface, and monitor changes to the surface. This interface is typically available through the SurfaceView class.

When using this interface from a thread other than the one running its SurfaceView, you will want to carefully read the methods lockCanvas() and Callback.surfaceCreated()."

Vi får altså tak i en slik SurfaceHolder ved å kalle getHolder(), og vi skal senere se at vi sender det videre til den tråden vi skal opprette.

Videre lar vi vår SurfaceView implementere grensesnittet SurfaceHolder.Callback. Dette grensesnittet har tre metoder:

- abstract void surfaceChanged(SurfaceHolder holder, int format, int width, int height)
- abstract void surfaceCreated(SurfaceHolder holder)
- abstract void surfaceDestroyed(SurfaceHolder holder)

og gir oss altså anledning til å bestemme hva som skal skje når flata opprettes, endres og slettes igjen. La oss først bare skrive ut når de kalles ved å implementere dem på følgende måte:

```
public void surfaceChanged(SurfaceHolder holder, int format, int width,
        int height) {
   Log.i(SnakeActivity.TAG,"surfaceChanged");
}
public void surfaceCreated(SurfaceHolder holder) {
   Log.i(SnakeActivity.TAG,"surfaceCreated");
}
public void surfaceDestroyed(SurfaceHolder holder) {
   Log.i(SnakeActivity.TAG,"surfaceDestroyed");

}
```

Så legger vi inn en slik SnakeView i activity_main.xml ved å skrive

```
<no.hist.itfag.snake.SnakeView
    android:id="@+id/snakeView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

I vår Activity-klasse skriver vi

```
private SnakeView mSnakeView;
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       getActionBar().setDisplayHomeAsUpEnabled(true);
       mSnakeView = (SnakeView)findViewById(R.id.snakeView);
   }
```

Da kan vi kjøre programmet, og får skrevet ut de to første log-setningene straks, mens den tredje kommer når vi avslutter igjen.

Det neste vi skal gjøre, er å ta vare på den bredden og høyden som vi får fra surfaceChanged.

```
public void surfaceChanged(SurfaceHolder holder, int format,
       int width, int height) {
       mWidth=width;
       mHeight=height;
}
```

der mWidth og mHeight er objektvariabler i SnakeView. Disse skal vi senere bruke til å kontrollere om ormen vår kolliderer med skjermgrensene.

Vi skal bruke en tråd til å styre spillingen. Vi lager den i SnakeView-konstruktøren (se punkt 8.3), men starter den i surfaceCreated og avslutter den i surfaceDestroyed:

```
public void surfaceCreated(SurfaceHolder holder) {
    mThread.setRunning(true);
    mThread.start();
}
public void surfaceDestroyed(SurfaceHolder holder) {
    boolean retry = true;
    mThread.setRunning(false);
    while (retry) {
        try {
            mThread.join();
            retry = false;
        } catch (InterruptedException e) {
        }
    }
}
```

SurfaceView (som andre View) har blant annet metodene onDraw(Canvas canvas), onKeyDown(int keyCode, KeyEvent event), onKeyUp(int keyCode, KeyEvent event), og dispatchTouchEvent (MotionEvent event) som det ofte er aktuelt å omdefinere (override). I tillegg har den metoden draw(Canvas canvas), som er et alternativ til onDraw().

Med metoden dispachTouchEvent() kan vi finne ut hvor på skjermen brukeren trykker. Vi implementerer den slik:

```
 @Override
public boolean dispatchTouchEvent(MotionEvent event) {
  if (event.getAction()==MotionEvent.ACTION_DOWN) {
        Posision touch = new Posision((int)event.getX(0),(int)event.getY(0));
        if (isLeftOf(touch)) turn(LEFT);
        else turn(RIGHT);
  }
  return true;
}
```

Vi sjekker altså om berøringsskjemen er trykket (andre aksjoner er blant annet ACTION_UP som blir avfyrt når man løfter fingeren/pennen opp igjen) og leser av x- og y-verdien for punktet.

Klassen Posision (som er en indre klasse til SnakeView, slik at SnakeView-objektet kan bruke de private objektvariablene til Posision-objektene) har objektvariablene xPos og yPos som forteller hvor på skjermen man befinner seg.

Avhengig av hvor man klikker i forhold til hvor ormen befinner seg, skal ormen snu seg 90 grader til høyre eller venstre. Hvis ormen beveger seg horisontalt, sjekker vi om ormens hode (mSnake.get(0)) er over eller under berøringspunktet, mens hvis den beveger seg vertikalt,

sjekker vi om hodet er til høyre eller venstre for berøringspunktet. I alle tilfeller får vi ormen til å snu seg i retning av berøringspunktet.

```
      private boolean isLeftOf(Posision pos) {
            if (mHeading == LEFT) return mSnake.get(0).yPos < pos.yPos;
            else if (mHeading==RIGHT) return mSnake.get(0).yPos > pos.yPos;
            else if (mHeading==UP) return mSnake.get(0).xPos > pos.xPos;
            else return mSnake.get(0).xPos < pos.xPos;
      }
```

Selve snuingen av ormen gjøres ved å endre på objektvariabelen mHeading. mHeading kan ha en av de fire verdiene 0, 1, 2 eller 3. Hvis vi ønsker å snu til venstre, minker vi mHeading med 1, hvis vi ønsker å snu til høyre, øker vi den med 1. Men siden vi bare ønsker tall i området 0 til 4, tar vi % 4. Og -1 blir da det samme som +3.

```
   public void turn(int direction) {
               if (direction==LEFT) mHeading = (mHeading+3) % 4;
               else mHeading = (mHeading+1)%4;
   }
```

Så er vi klar til å begynne å tegne ormen og maten. Til det bruker vi draw(). Da får vi bruk for en del metoder som finnes i Canvas-klassen. Den har mange forskjellige draw-metoder og det er lurt å ta en titt på dokumentasjonen, slik at du ser hvilke muligheter du har.

```
@Override
public void draw(Canvas canvas) {
    super.draw(canvas);
    updateFood();
    updateSnake();
    drawFood(canvas);
    drawSnake(canvas);
}
```

Her foretar vi de nødvendige oppdateringer på ormen og maten, før vi tegner dem.

Både ormen og maten er lagret i lister:

```
      private ArrayList<Posision> mSnake = new ArrayList<Posision>();
      private ArrayList<Posision> mFood = new ArrayList<Posision>();
```

Å tegne maten er ikke så vanskelig:

```
public void drawFood(Canvas canvas) {
    for (Position m : mFood) {
        canvas.drawCircle(m.xPos+INCREMENT/2, m.yPos+INCREMENT/2,
                INCREMENT/2, mFoodPaint);
    }
}
```

For hver matbit tegner vi en hvit sirkel som plasseres innenfor firkanten xPos,xPos+INCREMENT, yPos, yPos+INCREMENT. drawCircle sine parametre er henholdsvis x-posisjon og y-posisjon til sirkelens sentrum, radius og til slutt hvilken maling som skal brukes. INCREMENT er en konstant som kan betraktes som størrelsen på et (tenkt) rutenett på skjermen. Jeg har valgt å la maten samt ormens ledd være en slik rute stor, og la ormen flytte seg en rute mellom hver gang den tegnes opp.

Paint er den malingen vi skal bruke. For å unngå at den lages mange ganger, har jeg den som en objektvariabel:

```
private Paint mFoodPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

ANTI_ALIAS_FLAG er konstant som forteller at figuren skal tegnes opp på en slik måte at den ser mest mulig naturlig ut, uten synlige hakk i konturen. I SnakeView-konstruktøren har jeg sagt at den skal være hvit:

```
mFoodPaint.setColor(Color.WHITE);
```

Ormen ønsker jeg å tegne som en firkant for hvert ledd, bortsett fra hodet, som skal være en halvsirkel. Metoden drawRect(int xStart,int yStart, int xEnd, int yEnd, Paint paint) tegner rektangler, så vi kan skrive

```
// Paint body
for (int i=1; i <mSnake.size();i++) {
       Posision body = mSnake.get(i);
       canvas.drawRect(body.xPos, body.yPos,
       body.xPos+INCREMENT, body.yPos+INCREMENT, mSnakePaint);
}
```

Ormens hode skal være en halvsirkel. Den tegner vi ved hjelp av metoden drawArc(RectF oval, float startAngle, float sweepAngle, boolean useCenter, Paint paint). Første parameter er området som buen skal plasseres i, andre parameter forteller hvor vi skal starte buebevegelsen, tredje parameter forteller hvor mange grader buen skal omfatte, fjerde parameter forteller om sentrum skal tegnes opp (vi gjør det, men det spiller egentlig ingen rolle siden sentrum her sammenfaller med begynnelsen av neste ledd), og til slutt malingen som skal brukes.

Bevegelsesretningen har betydning når vi skal tegne hodet. Objektvariabelen mHeading kan ha de fire verdiene DOWN (0), UP (2), LEFT(1) og RIGHT(3), og avhengig av dette, skal den buen vi tegner enten vende oppover, nedover, til høyre eller venstre. Det får vi til ved å la startAngle enten være 0, 90, 180 eller 270 grader. I alle tilfeller er det en halvsirkel vi skal tegne, så vi må angi at sweepAngle skal være 180 grader.

Vi må også beregne hvor i ruta buen skal plasseres, siden den bare fyller opp halvparten av ruta i den retningen den beveger seg. Dette gjøres ved å flytte enten x eller y en halv rute forover eller bakover. Så definerer vi det rektangelet som halvsirkelen skal ligge innenfor og lagrer dette i objektet rect og er klar til å tegne halvsirkelen:

```
// Paint head
Posision head = mSnake.get(0);
int x=head.xPos;
int y=head.yPos;
if (mHeading==LEFT) x+=INCREMENT/2;
else if (mHeading==RIGHT) x-=INCREMENT/2;
else if (mHeading==UP) y+=INCREMENT/2;
else y-=INCREMENT/2;
RectF rect=new RectF(x, y, x+INCREMENT,y+INCREMENT);
int startAng=90*mHeading;
canvas.drawArc(rect, startAng, 180, true, mSnakePaint);
```

Så går vi over til å se på hvordan ormen og maten oppdateres. Først maten (for den er enklest):

```
public void updateFood() {
       eatFood(mSnake.get(0));
       newFood();
}
```

Maten skal dukke opp på tilfeldige steder (men ikke oppå ormen) til tilfeldige tider. Derfor lager vi et Random-objekt og lar det være en objektvariabel. Det er ikke lurt å la det generers hver gang vi har bruk for det, for da starter den "tilfeldige" tallrekken på nytt hver gang, og da blir det ikke særlig tilfeldig.

```
public void newFood() {
    if (mRand.nextDouble() < FOODPROBABILITY) {
        int xPos = mRand.nextInt(mWidth/INCREMENT) * INCREMENT;
        int yPos = mRand.nextInt(mHeight/INCREMENT) * INCREMENT;
        Position newPos = new Position(xPos, yPos);
        boolean found = false;
        int i=0;
        while (!found && i < mSnake.size()) {// same position as snake?
            if (newPos.equals(mSnake.get(i))) found = true;
            i++;
        }
        i=0;
        while (!found && i < mFood.size()) {// same position as other food?
            if (newPos.equals(mFood.get(i))) found = true;
            i++;
        }
        if (!found) {
            mFood.add(newPos);
        }
    }
}
```

Jeg har satt FOODPROBABILITY til 0.05, slik at det er 5% sjanse for ny mat hver gang. nextInt(int x) gir et heltall mellom 0 og x, så x=mWidth/INCREMENT gir en lovlig «rute». Og når jeg da etterpå multipliserer med INCREMENT, får jeg en lovlig posisjon. Så må jeg sjekke om den nye maten haver oppå ormen eller oppå annen mat. Hvis den ikke gjør det, aksepteres den og legges inn i mat-lista.

Hvis ormens hode er i samme posisjon som en matbit, spiser ormen den og vokser med ett ledd:

```
public void eatFood(Posision head) {
      int i = 0;
      boolean found = false;
      while (!found && i < mFood.size()) {
            if (head.equals(mFood.get(i))) {
                  found = true;
            }
            else i++;
      }
      if (found) {
            mFood.remove(i); // eat the food
            mInc=true; //grow
      }
}
```

Så var det ormens bevegelse. Den skal bevege seg en rute videre i den retning som dens mHeading forteller. Istedenfor å oppdatere alle leddene, lager vi et nytt ledd forrerst og fjerner det bakerste hvis ormen ikke skal vokse, og lar være å fjerne det bakerste hvis den skal vokse.

For å beregne hvilken vei ormet skal bevege seg, bruker vi metoden advance:

```
public void advance(Posision head) {
      switch (mHeading) {
      case UP:
            head.yPos -= INCREMENT;
            break;
      case RIGHT:
            head.xPos += INCREMENT;
            break;
      case DOWN:
            head.yPos += INCREMENT;
            break;
```

```
        case LEFT:
                head.xPos -= INCREMENT;
                break;
        }
}

private void updateSnake() {
        Posision newHead = new Posision(mSnake.get(0));
        advance(newHead);
        mSnake.add(0,newHead);
        if (!mInc) mSnake.remove(mSnake.size()-1);
        else mInc=false;
        boolean OK=true;
        if (newHead.xPos < 0 || newHead.xPos > mWidth // Outside screen
                        || newHead.yPos <0 || newHead.yPos > mHeight) OK=false;
        for (int i=1; i <mSnake.size();i++){
                if (newHead.equals(mSnake.get(i))) OK=false; //Collision with itself
        }
        if (!OK){
                Log.i(SnakeActivity.TAG,"Collision!");
                Log.i(SnakeActivity.TAG,"X="+newHead.xPos+" Y="+newHead.yPos);
                mThread.setRunning(false); // game finished()
        }
}
```

8.3 Thread

Nå skal vi gå over til å se på spill-tråden. Først legger jeg nn en tråd i konstruktøren til SnakeView:

```
public SnakeView(Context context, AttributeSet attrs) {
    super(context, attrs);
    // register our interest in hearing about changes to our surface
    SurfaceHolder holder = getHolder();
    holder.addCallback(this);
    mThread = new SnakeThread(holder, this);
    mSnake.add(new Position(INCREMENT, INCREMENT)); // Snake with just head
    mFoodPaint.setColor(Color.WHITE);
    mSnakePaint.setColor(Color.RED);
}
```

Konstruktøren for klassen SnakeThread lagrer de verdiene som sendes med i objektvariabler:

```
public SnakeThread(SurfaceHolder holder, SnakeView snakeView){
    mSurfaceHolder = holder;
    mSnakeView = snakeView;
}
```

I tillegg har vi andre objektvariabler, blant annet mSleepInterval som regulerer hastigheten til ormen.

Run-metoden ser slik ut:

```
@Override
public void run() {
    Canvas c;
    while (isRunning()) {
        if (!isPaused()) {
            c = null;
            try {
```

```
                c = mSurfaceHolder.lockCanvas(null);
                synchronized (mSurfaceHolder) {
                    mSnakeView.draw(c);
                 }
            } finally {
                if (c != null) {
                    mSurfaceHolder.unlockCanvasAndPost(c);
                }
            }
        }
        mySleep(mSleepInterval);
    }
}
```

Her må vi ta i bruk den SurfaceHolder'en som ble sendt med, og låse SurfaceView'en for å forhindre at den blir endret eller slettet mens tegningen pågår. Hvis låsingen går bra, får vi returnert en canvas som vi kan tegne på. Når vi er ferdig med å tegne, sender vi canvasen til SurfaceView'en og låser den opp igjen. I tillegg til å låse canvasen, synkroniserer vi også, slik at vi er sikker på at SurfaceHolder'en ikke blir brukt av andre mens vi holder på å tegne.

Vi har null som parameter til lockCanvas. Da får vi oppdatert hele skjermen. Alternativt kunne vi ha sendt med et rektangel, da ville bare området innenfor rektangelet blitt oppdatert. Dette er aktuelt hvis det er bare et lite område av skjermen som skal endre seg – det går fortere å oppdatere bare et lite område enn hele skjermen.

mySleep() er en enkel metode som bare sørger for å passivere tråden et bestemt antall millisekunder:

```
public void mySleep(int length) {
   try {
          sleep(length);
   }
   catch (InterruptedException e) {
   }
}
```

Nå gjenstår bare å lage noen få enkle metoder. Og så kan vi lage en option-meny i Activity-klassen for å avslutte spillet (og eventuelt sette på pause, start på nytt og liknende). Så er vi klare til å spille! En fullstendig versjon ligger vedlagt i ItsLearning.

8.4 Handler

La oss nå gå over til å se på Snake-versjonen som følger med Android-systemet. Hvis du lastet ned eksemplene sammen med systemet, finner du koden under sdk/samples/android-xx/legacy/Snake. Denne applikasjonen består av fire filer, Snake som er en Activity, TileView som er en subklasse av View, SnakeView som er en subklasse av TileView samt BackgrounView som er en subklasse av View.

TileView er en klasse som er laget til denne applikasjonen, men som er så generell at du godt kan ta den i bruk i andre sammenhenger. Den består av en tabell av Bitmaps (tiles, fliser), og en todimensjonal tabell av indekser som forteller hvilken flis som skal tegnes på de ulike posisjonene på skjermen. Først laster man inn flisene, så fordeles de utover skjermen (samme flis kan brukes mange ganger!) og så tegner man dem.

For å tegne flisene, brukes onDraw som ser slik ut:

```
    @Override
```

```
    public void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        for (int x = 0; x < mXTileCount; x += 1) {
            for (int y = 0; y < mYTileCount; y += 1) {
                if (mTileGrid[x][y] > 0) {
                    canvas.drawBitmap(mTileArray[mTileGrid[x][y]],
                                mXOffset + x * mTileSize,
                                mYOffset + y * mTileSize,
                                mPaint);
                }
            }
        }
    }
```

For hver gang skjermen tegnes opp, tegnes alle flisene ved hjelp av metoden canvas.drawBitmap. Der det ikke er fliser, beholdes bakgrunnen. I drawBitmap oppgir man først hvilket bilde som skal tegnes, deretter posisjonen til øverste venstre hjørne av bildet og til slutt hvilken maling som skal brukes.

Selve spillet lages i SnakeView. Ormen og maten lages på mye av den samme måten som i det første eksemplet, bare at man bruker fliser til å tegne dem. Det som er forskjellig, er måten man håndterer trådprogrammeringen på. Her brukes klassen Handler. Det er en klasse for å håntere meldinger. Ifølge [Handler] er det to hovedgrunner til å bruke den:

"There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own."

I SnakeView brukes den til det første formålet.

```
      class RefreshHandler extends Handler {

        @Override
        public void handleMessage(Message msg) {
            SnakeView.this.update();
            SnakeView.this.invalidate();
        }

        public void sleep(long delayMillis) {
            this.removeMessages(0);
            sendMessageDelayed(obtainMessage(0), delayMillis);
        }
    }
```

Den første metoden henter og behandler en melding. Selve meldingen er uinteressant her, poenget er den tiden det tar før den kommer fram. Det bestemmes i den andre metoden. Der sendes meldingen, men ikke før det har gått delayMillis millisekunder. Resultatet er at update() og invalidate() kalles med så mange millisekunders mellomrom. På denne måten slipper vi å håntere ekstra tråder på egen hånd. invalidate gir beskjed om at skjermen må tegnes opp på nytt. update-metoden oppdaterer alt som skal oppdateres, og kaller så sleep-metoden:

```
public void update() {
        if (mMode == RUNNING) {
            long now = System.currentTimeMillis();

            if (now - mLastMove > mMoveDelay) {
                clearTiles();
                updateWalls();
                updateSnake();
                updateApples();
                mLastMove = now;
```

```
            }
            mRedrawHandler.sleep(mMoveDelay);
        }
}
```

8.5 Oppsummering og videre lesing

Grafikk kan gjøres på mange måter, fra det helt enkle statiske via mer avansert 2D-grafikk til 3D-grafikk med animasjon. 3D-grafikk behandler vi ikke i dette kurset. Når det gjelder implementasjon av 2D-grafikk, har vi flere mulige valg:

1. Tegne rett inn i en View
2. Tegne med en Canvas i en (subklasse av) View
3. Tegne med en Canvas i en SurfaceView

Alternativ 1 egner seg best når du skal ha enkel grafikk som ikke skal endre seg (vesentlig). Det er det vi har gjort tidligere, når vi f.eks. har laget ImageView av Drawable.

Alternativ 2 egner seg for for grafikk som skal endre seg, men ikke veldig raskt. Snake versjon 2 bruker denne teknikken.

Alternativ 3 egner seg når vi trenger å oppdatere så ofte at vi ikke har tid til å vente på View-hierarkiets normale tegne-prosedyrer. Da lager vi en egen tråd der vi tegner med en egen Canvas i eget tempo, og så oppdateres skjermen så ofte den klarer. Snake versjon 1 viser denne metoden.

Du kan lese mer om dette på [Grafikk]. Her henvises det til LunarLander, som også finnes i eksempelsamlinga til Android. Studer gjerne den også. Der kan du også se hvordan en Handler kan brukes til å formidle beskjeder fra en tråd til en annen

[Getting started with Android] gir en grei oversikt over hvordan spill-programmer bør organiseres. Der introduseres "the main loop"

```
public void run() {
    while (isRunning) {
        while (isPaused && isRunning) {
            sleep(100);
        }
        update();
    }
}
```

Metoden update oppdaterer alt som trengs å oppdateres, inkludert opptegning av skjermen:

```
private void update() {
    updateState();
    updateInput();
    updateAI();
    updatePhysics();
    updateAnimations();
    updateSound();
    updateVideo();
}
```

Rekkefølgen er ikke tilfeldig. Først bør man oppdatere spillets tilstand (for eksempel om spillet er ferdig), deretter leser man input fra bruker og "input" fra spillets intelligens. Så oppdateres "fysikken" – som kan være alt fra beregning av hvor ting skal flytte seg (for eksempel ormen) til om svaret på et spørsmål var riktig eller beregning av poengsum. Til slutt

oppdateres animasjonen (for eksempel flytter til neste bilde), spiller av riktig lyd og viser det som skal vises på skjermen.

8.6 Referanser

[SurfaceView] http://developer.android.com/reference/android/view/SurfaceView.html

[SurfaceHolder] http://developer.android.com/reference/android/view/SurfaceHolder.html

[Handler] http://developer.android.com/reference/android/os/Handler.html

[Grafikk] http://developer.android.com/guide/topics/graphics/index.html

[Getting started with Android] http://www.rbgrn.net/content/54-getting-started-android-game-development