C# spil (del 2) - Tilstandmaskiner i spil

Denne artikkel omhandler brugen af tilstandmaskiner i spil. Tilstandsmaskiner kan bruges til at styre logikken i et spil. Artiklen indeholder et eksempel til demonstation. I næste artikel (del 3) implementeres space invaders

Skrevet den **11. Feb 2009** af **sovsekoder** I kategorien **Programmering / C#** | ★★★☆☆

Introduktion til tilstandmaskiner
Kilde koden tilknyttet denne artikel kan findes sidst i artiklen i afsnitet kaldet: "Komplet Kilde Kode Her". De skarpe læsere vil se at BlockSprite og SpriteCollision klasserne er opdateret i forhold til artiklen C# spil (del 1). BlockSprite har fået en property kaldet, visible. Denne property angiver om spriten er synlig eller ej. Hvis spriten ikke er synlig kan den ikke kollidere med andre sprites, og den vises heller ikke. Listener klassen i SpriteCollision.cs har også fået en ny property, remove. Denne property sættes til true, hvis man ønsker at fjerne sin listener (som hører efter om 2 sprites er kollideret). Funktionaliteten bruges ikke i denne artikel, men bliver taget i brug i næste artikel C# spil (del 3), hvor space invaderes bliver implementeret.

Denne artikel vil være rar at forstå, da space invaders spillet gør brug af tilstandsmaskiner til at styre spillet med. Desuden er det en god viden at sidde inde med, når du skal igang med dit eget spil!

Før jeg plastrer kode ud over det hele vil jeg give et indblik i hvad tilstandsmaskiner er. For at forstå hvad tilstandsmaskiner er, er det første spørgsmål, der skal besvares nok endnu mere simpelt: Hvad er en tilstand egentlig? ...En tilstand defineres af programmøren selv (eller af designeren af tilstandmaskinen :D) - følgende overskrifter kan f.eks dække over tilstande, der kunne forekomme i et spil:

[Game Over] : spillet er slut og du er død!
[In Game] : du er inde i spillet og kan f.eks styre et objekt
[Wait For New Game] : venter til at brugeren sætter et spil igang
[Lost a Life] : du har tabt et liv, men er ikke game over endnu

Ovenstående tilstande dækker over "overordnede" tilstande i et spil - som vi alle kender. Man kan også have tilstande for mere specifikke områder i spillet, som f.eks for bevægelsen af invaders i "space invaders":

[Move Left] : flytter invaders til venstre
[Move Down in Left Side] : flytter invaders ned i venstre side
[Move Right] : flytter invaders til højre
[Move Down in Right Side] : invaders ned i højre side

Jeg vil nu prøve at holde fast i det sidste eksempel, med bevægelsen af space invaders,et øjeblik. For nemhedens skyld bruger jeg navnene MoveLeft, MoveLeftDown, MoveRight, MoveRightDown for de enkelte tilstande. Vi ønsker at der skal ske nget bestemt i hver enkelt tilstand. I MoveLeft skal alle invaders flytte sig til venstre. Når de kommer til den venstre kant af skærmen, skal de gå ned. Tilstanden MoveLeftDown beskriver denne situation. Bemærk at det er ikke nok med bare een tilstand der hedder MoveDown, da det i en sådan model ikke er muligt at finde ud af om vores invaders er i højre eller venstre side af skærmen.

Vi definerer altså en tilstand, som dækker over en overordnet tilstand/begivenhed i spillet, og en arbejdsproces (som siger noget om hvad der skal ske i den pågældene tilstand - f.eks invaders flyttes til venstre), samt en betingelse for hvad der gør at man kommer til en ny tilstand (f.eks at invaders stopper

med at gå til venstre, men går ned i stedet).

I tilfældet med space invaders bevægelse, vil vi gerne have at de går fra venstre mod højre, og derefter ned - så fra højre mod venstre, og ned igen. Herefter gentages processen med at gå fra venstre mod højre osvosv. Det er lige netop dette mønster der kan genskabes ved brug af en tilstandsmaskine. Lad os sige at invaders starter med at gå til højre. Vi starter derfor i tilstand: MoveRight. Vi siger nu at der er en *overgang* til tilstanden MoveRightDown. Dette betyder at når visse betingelser er opfyldt, så hopper vi over i tilstanden MoveRightDown. Dette kan vi skrive:

[MoveRight] -> [MoveRightDown]
Betingelsen for overgangen er, at invaders er nået ud til kanten i højre side af skærmen. Når denne betingelse er opfyldt går vi til tilstanden, MoveRightDown.

På denne måde kan vi give et overblik over vores tilstande, og hvordan de hænger sammen:

[MoveRightDown] -> [MoveLeft], hvor betingelsen er at vi har gået en hvis længe ned i pixels (sådan er spillet bare). Næste overgang er:

[MoveLeft] -> [MoveLeftDown], hvor betingelsen er at invaders er nået ud i venstre skærmkant. Sidste overgang er givet ved:

[MoveLeftDown] -> [MoveRight], hvor betingelsen er at invaders har gået langt nok ned.

Vi er nu endt i tilstanden MoveRight, hvor vi jo startede. Tilstandmaskinen går nu i ring, hvilket også hedder en infinite state machine (den er uendelig). Tilstandmskinen behøver ikke gå i ring. Vi kunne vælge at lade vores invaders stoppe i MoveLeftDown, så invaders bare forsatte ned. I sådan et tilælde er det så en endelig tilstands maskine (finite state machine), hvor slut tilstanden er: MoveLeftDown.

Implementering af tilstandsmaskiner
Vi stopper nu med eksemplet ang. space invaders, og går over til demonstrations projektet. Vi skal prøve at implmenetere en tilstandsmaskine der styrer en sprite, sprite1. Sprite'en skal bevæge sig rundt langs kanten på et raktangel. Vi ønsker følgende tilstande og overgange:

Tilstande:
[Right] sprite1 går til højre
[Down] sprite1 går ned
[Left] sprite1 går til venstre
[Up] sprite1 går op

Overgange:
[Right] -> [Down]
[Down] -> [Left]
[Left] -> [Up]
[Up] -> [Right]

Betingelsen for overgangene er at vi er nået kanten af retanglet. Disse betingelser implementeres senere. Resultatet af tilstandsmaskine er således at sprite1 vandrer rundt og rundt langs kanten på et rektangel.

Vi skal nu definere de 4 tilstande: Left, Right, Down og Up. Disse tilstande defineres i en enumeration, og vi erklærer en variabel, der indeholder den aktuelle tilstand:

```
// Tilstande for sprite1
    enum Sprite1States
    {
```

```
      Left,
      Right,
      Down,
      Up
    };

    // Denne variable indeholder den aktuelle tilstand
    Sprite1States sprite1State = Sprite1States.Right;
```

Vores variabel sprite1State indeholder nu start tilstanden Right. Det kan godt blive aktuelt at have flere tilstands-variable, dette vil jeg dog ikke kommer nærmere ind på. I denne serie af artikeler bruges kun tilstandsmaskiner med en variabel der beskriver tilstanden. Hvis vi ønsker at hoppe over i en anden tilstand, skal vi blot tildele vores variabel værdien for den nye tilstand. Hvis vi skal over i f.eks. Up skriver vi blot:

```
sprite1State = Sprite1States.Up;
```

Vi har nu defineret de nødvendige tilstande for at beskrive spritens bevægelse, samt oprettet en variabel til at holde styr på den tilstand vi er i nu.

Vi mangler nu at implementere betingelser for de enkelte overgange samt den arbejdsopgave der skal udføres i hver enkelt tilstand (såsom at flytte sprite'en en tak til højre, hvis vi er i tilstand: Right). Funktionaliteten og betingelserne af tilstandsmaskinen placeres i gameloopet (=timer1_tick, som nævnes i artiklen C# spil del 1, hvor step1 viser hvorledes denne metode sættes op). Gameloopet bliver kaldt et vist antal gange i sekundet. Denne metode bliver altså kørt igen og igen. Følgende kode viser implementeringen af tilstandsmaskinen (koden placeres før Invalidate(true), som er sidste linie):

```
switch(sprite1State)
{
  case Sprite1States.Down:
    // Tjek om spriten holder sig inden for y-grænsen
    if(sprite1.Y >= LowerBorder)
      // Down -> Left
      sprite1State = Sprite1States.Left;
    else
    {
      sprite1.Y++;
    }
    break;
  case Sprite1States.Left:
    // Tjek om spriten holder sig inden for x-grænsen
    if(sprite1.X <= LeftBorder)
      // Left -> Up
      sprite1State = Sprite1States.Up;
    else
      sprite1.X--;
    break;
  case Sprite1States.Right:
    // Tjek om spriten holder sig inden for x-grænsen
    if(sprite1.X >= RightBorder)
      // Right -> Down
      sprite1State = Sprite1States.Down;
```

```
    else
      sprite1.X++;
    break;
  case Sprite1States.Up:
    // Tjek om spriten holder sig inden for y-grænsen
    if(sprite1.Y <= TopBorder)
      // Up -> Right
      sprite1State = Sprite1States.Right;
    else
      sprite1.Y--;
    break;
}
```

Overnstående kode kræver at følgende konstanter er erklæret:

```
// Statiske konstanter der bruges til at definere rammen
  public static int TopBorder = 10;
  public static int LeftBorder = 10;
  public static int RightBorder = 150;
  public static int LowerBorder = 150;
```

Som man kan se er tilstandsmaskinen i bund og grund bare en switch-case. Så hvordan virker den?

Koden bliver placeret inde i gameloopet, som tidligere nævnt bliver kaldt flere gange i sekundet. Hver gang gameloopet kaldes gennemgås switch sætningen altså. Vi startede med at sætte sprite1State (vores tilstands variabel) til at være Right. Dette betyder altså at vi ryger ind i Right-casen, hver gang gameloop-metoden kaldes. I right-casen har vi både en aktion (jobbet) og en betingelse. Betingelse og job er, i tilstanden Right, givet ved:

```
if(sprite1.X >= RightBorder)
  // Right -> Down
  sprite1State = Sprite1States.Down;
else
  sprite1.X++;
```

Det der står er altså populært sagt: "hvis sprite1 kommer udover kanten i højre side, så skal vi gå til tilstand Down (det var betingelsen!). Hvis ikke dette er tilfældet så flytter vi spriten en tak til højre, ved at lægge 1 til spritens nuværende x-koordinat (og det var så arbejdsopgaven!). På denne måde vil spriten flytte sig en tak til højre lige indtil den når kanten. Når kanten er nået ændres tilstanden til Down, og så er vi altså over i Down-tilstanden, hvor der så vil være en ny betingelse og en ny arbejdsopgave for tilstanden.

De andre tilstande implementeres i samme stil som Right. Det er ikke en nødvendighed at indsætte tilstandsmaskinen i gameloopet. Hvor tilstandsmaskinen placeres i koden, afhænger af hvad den skal gøre/udføre. I dette tilfælde er det smart at placere tilstandsmaskinen i gameloopet, da spriten skal flytte sig hver gang gameloopet kaldes (og derved holde et konstant tempo). Sprite'ens position opdateres hver gang gameloop'et kaldes.

Tilstands maskinen kan også være delt. Dette har jeg valgt at vise ved at putte en ny sprite på (sprite2). Denne sprite's bevægelse er afhængig af om den er kollideret med sprite1 eller ej. Sprite2 starter med at vente i venstre side. Når den kolliderer med sprite1 (som hele tiden kører rundt vha. tilstandmaskinen der

lige er beskrevet), så flytter sprite2 sig over i højre side og venter. Når de to sprites kolliderer igen (hvilket så vil være i højre side når sprite1 er nået derover), så flytter sprite2 sig tilbage i venstre side igen. - Når de to sprites kolliderer herefter, så flytter sprite2 sig i en cirkelbevægelse tilbage til startpunktet og bevægelsen starter forfra (så sprite2's tilstandsmaskine er, som sprite1's, uendelig).

Det nye er at vi har en speciel betingelse iform af en kollision. Sprite2 skal have en overgang der er betinget af kollisionen - når de to sprites kolliderer så skal der ske "noget nyt". Dette betyder at noget af implementeringen af tilstandsmaskinen for sprite2, flyttes over i collisionhandleren. Følgende skridt skal tages for at sætte sprite 2 op samt den tilhørende tilstandsmaskine.

0) tilføj tilstandene for sprite2, samt en tilstandsvariabel
1) tilføj (sprite 1) og sprite 2
2) opret en collision handler for sprite 1 og 2.
3) tilføj en collision listener for sprite 1 og 2, der kaldet collision handleren
4) implementer noget af tilstandmaskinen i gameloopet (de dele der ikke afhænger af kollisionen)
5) implementer resten af tilstandsmaskinen i collision handleren (de tilstande som afhænger af kollisionen)

Vi går igang, kig på overskriften og så på koden - måske kan du genkende principperne fra sprite1...

0) tilføj tilstandene for sprite2 og tilstands variabelen

```
// Tilstande for sprite2
enum Sprite2States
{
  Left,
  Circle,
  WaitCollisionLeft,
  WaitCollisionLeft2,
  Right,
  WaitCollisionRight
};
// Tilstandsvariablen for sprite2
Sprite2States sprite2State = Sprite2States.Right;
```

1) tilføj sprite 1 og sprite 2

```
// Opret sprite1 (sort) og sprite2 (rød)
sprite1 = new BlockSprite(LeftBorder, TopBorder, spriteSize,spriteSize);
sprite2 = new BlockSprite(LeftBorder, (LowerBorder-TopBorder)/2+TopBorder,
spriteSize,spriteSize);
sprite2.Color = Color.Red;
```

2) Opret en collision handler for sprite 1 og 2.

```
void CollisionHandler(object sender)
    {
      // Her indsættes noget af sprite2's tilstands-maskine senere..
    }
```

Kollisions handleren vist ovenfor skal senere fyldes ud med en del af sprite2's tilstandsmaskine. Kollisions handleren kaldes så længe at sprite1 og sprite2 overlapper (de er kollideret).

3) tilføj en collision listener for sprite 1 og 2, der kalder collision handleren

```
collisions.AddListener(sprite1, sprite2, new
CollisionDelegate(CollisionHandler));
```

hvor collisions variablen er en instans af SpriteCollision klassen. Denne variabel bliver initialiseret i konstruktøren, som SpriteCollision collisions = new SpriteCollision(); - det er denne klasse der administrerer, hvordan der tjekkes efter kollisioner. Dette gøre ved at kalde collisions.Check(); hvilket typisk kan skal ske, som det første i gameloopet (se evt. kilde koden for Form1).

4) implementer noget af tilstandmaskinen i gameloopet (de tilstande der ikke afhænger af kollisionen)

```
      switch(sprite2State)
      {
        case Sprite2States.Left:
          if(sprite2.X <= LeftBorder)
            // Left -> WaitCollisionLeft
            sprite2State = Sprite2States.WaitCollisionLeft;
          else
            sprite2.X--;
          break;
        case Sprite2States.Circle:
          double amp = (LowerBorder-TopBorder)/3;
          sprite2.X = tempX + (int)(amp*(1-Math.Cos(t)));
          sprite2.Y = tempY + (int)(-amp*(Math.Sin(t)));

          if(t >= 2*Math.PI)
          {
            // Circle -> WaitCollisionLeft2
            sprite2State = Sprite2States.WaitCollisionLeft2;
            sprite2.X = tempX;
            sprite2.Y = tempY;
          }

          t+= Math.PI/100d;
          break;
        case Sprite2States.Right:
          if(sprite2.X >= RightBorder)
            // Right -> WaitCollisionRight
            sprite2State = Sprite2States.WaitCollisionRight;
          else
            sprite2.X++;
          break;
      }
```

Ovenstående tilstandsmaksine sørger altså for at flytte sprite 2 til siderne og i den omtalte cirkel bevægelse. Som det ses behandles følgende tilstande ikke: WaitCollisionLeft, WaitCollisionLeft2, WaitCollisionRight. Disse tilstande bliver behandlet i vores collision handler som vi tilføjede i punkt 2.

Implementeringen af denne del af tilstandsmaskinen kan ses i det følgende:

5) Implementer resten af tilstandsmaskinen i collision handleren (de tilstande som afhænger af kollisionen)

```
// Tilstandsmaskine for sprite2
      switch(sprite2State)
      {
        case Sprite2States.WaitCollisionLeft:
          // Initialiser circel variable og overgå til ny tilstand:
          // WaitCollisionLeft -> Circle
          t=0;
          tempX = sprite2.X;
          tempY = sprite2.Y;
          sprite2State = Sprite2States.Circle;
          break;
        case Sprite2States.WaitCollisionLeft2:
          // WaitCollisionLeft2 -> Right
          sprite2State = Sprite2States.Right;
          break;
        case Sprite2States.WaitCollisionRight:
          // WaitCollisionRight -> Left
          sprite2State = Sprite2States.Left;
          break;
      }
```

Som det ses behandler vi tilstandene: WaitCollisionLeft, WaitCollisionLeft2, WaitCollisionRight. Denne del af tilstandsmaskinen besøges kun når sprite1 og sprite2 er kollideret. De tilstande der håndteres her håndteres ikke i gameloopet, hvilket betyder at der ikke foretages noget arbejdsproces før der sker en kollision. Når kollisionen er sket, kan vi se på overgangene at vi gå over til: Circle, Right eller Left. Disse tilstande behandles så i gameloopet, og kollisionen mellem sprite1 og sprite2 vil ignoreres da disse tilstande ikke håndteres her (i collisionHandleren)! - det var kringlet, men det hjælper at se demonstrationen.

Vi har nu implementeret tilstandsmaskinen og den fornødne funktionalitet bag sprite2. Det samme gælder for sprite1. Jeg håber jeg har givet et godt indblik i tilstandsmaskiner, og hvordan de virker. Demo projektet viser, hvordan man kan implementere regler for bevægelsen af sprites. Disse regler kan dog også overføres på andre ting i spillet, såsom spillets "gang", som indikerer hvorvidt spilleren er død, gameover eller måske nået til næste level. Det er fantasien der sætter grændserne. Man kan måske sige lidt generelt, at tilstandsmaskinerne kan bruges til at udføre et reglsæt. Om reglsættet gælder for spritens bevægelse eller spillets gang eller noget helt tredie - det bestemmer programmøren.

Den næste artikel C# spil (del 3), vil ikke forklare teknikken bag tilstandsmaskiner (det skulle gerne være dækket af denne artikkel) - men vil derimod vise hvordan man bruger dem til at lave et rigtigt spil, nemlig den gamle klassiker: **"Space Invaders"**.

Når man vender sig til den - nok lidt - specielle måde at tænke på (ved brug af tilstandsmaskiner), så synes jeg selv at det letter arbejdet, og gør spil-logikken mere overskuelig. Der findes mange gode måde at afbilde tilstandsmaksiner på grafisk. Det kan jeg desværre ikke vise i denne artikkel, da jeg ikke har mulighed for at indsætte billeder. Det gode ved en grafisk afbildning er ellers, at den er relativt simpel, men illustrerer reglsættet rigtigt godt! - Når man er kommet godt ind i brugen af tilstandsmaskiner, vil en grafisk afbildning af tilstandsmaskinen også være en god/let skabelon at implementere spillet efter...

Komplet Kilde Kode Her
Sådan får du koden op at køre:

Opret et windows form projekt med namespace: StatemachineDemo. Filen med formen hedder form1.cs. Herudover skal der være to filer: spritecollsion.cs og blocksprite.cs. Disse filer opretter du ved at højre klikke på projektet og vælge add New Item, og så vælge Class..., navnet på klassen skal være det samme som filnavnet - dvs: SpriteCollision og BlockSprite. Når filerne er oprettet copy/pastes koden ind i hver enkelt fil, og så skulle det kunne køre!

Demo programmet viser udover spritenes bevægelse også de tilstand de to tilstandsmaskiner bevægersig i.

Kilde koden til filerne kan findes herunder (overskriften markerer filens navn):

Form1.cs

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace StatemachineDemo
{
  public class Form1 : System.Windows.Forms.Form
  {
    // Statiske konstanter der bruges til at definere rammen
    public static int TopBorder = 10;
    public static int LeftBorder = 10;
    public static int RightBorder = 150;
    public static int LowerBorder = 150;

    // Tilstande for sprite1
    enum Sprite1States
    {
      Left,
      Right,
      Down,
      Up
    };

    // Tilstande for sprite2
    enum Sprite2States
    {
      Left,
      Circle,
      WaitCollisionLeft,
      WaitCollisionLeft2,
      Right,
      WaitCollisionRight

    };
```

```
    // Erklæring af sprites og deres tilstande
    BlockSprite sprite1;
    BlockSprite sprite2;
    int spriteSize = 20;
    Sprite1States sprite1State = Sprite1States.Right;
    Sprite2States sprite2State = Sprite2States.Right;
    // Variabler der bruges af sprite2 til circel bevægelse
    double t;
    int tempX, tempY;
    // Håndtering af Kollisioner
    SpriteCollision collisions = new SpriteCollision();

    // Komponenter på formen
    private System.Windows.Forms.Timer timer1;
    private System.ComponentModel.IContainer components;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.Label label4;
    private System.Windows.Forms.Label label5;
    private System.Windows.Forms.Label label6;
    private System.Windows.Forms.Label label7;
    private System.Windows.Forms.Label label8;
    private System.Windows.Forms.Label label9;
    private System.Windows.Forms.Label label10;

    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      // Tegn med double-buffering
      SetStyle(ControlStyles.UserPaint, true);
      SetStyle(ControlStyles.AllPaintingInWmPaint, true);
      SetStyle(ControlStyles.DoubleBuffer, true);

      // Opret sprite1 (sort) og sprite2 (rød)
      sprite1 = new BlockSprite(LeftBorder, TopBorder, spriteSize,spriteSize);
      sprite2 = new BlockSprite(LeftBorder, (LowerBorder-
TopBorder)/2+TopBorder, spriteSize,spriteSize);
      sprite2.Color = Color.Red;

      // Opret Collision listener for sprite1 og sprite2
      // - når sprite1 og sprite2 støder sammen kaldes CollisionHandler
      collisions.AddListener(sprite1, sprite2, new
CollisionDelegate(CollisionHandler));
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
```

```
      {
        components.Dispose();
      }
    }
    base.Dispose( disposing );
  }

  #region Windows Form Designer generated code
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent()
  {
    this.components = new System.ComponentModel.Container();
    this.timer1 = new System.Windows.Forms.Timer(this.components);
    this.label1 = new System.Windows.Forms.Label();
    this.label2 = new System.Windows.Forms.Label();
    this.label3 = new System.Windows.Forms.Label();
    this.label4 = new System.Windows.Forms.Label();
    this.label5 = new System.Windows.Forms.Label();
    this.label6 = new System.Windows.Forms.Label();
    this.label7 = new System.Windows.Forms.Label();
    this.label8 = new System.Windows.Forms.Label();
    this.label9 = new System.Windows.Forms.Label();
    this.label10 = new System.Windows.Forms.Label();
    this.SuspendLayout();
    //
    // timer1
    //
    this.timer1.Enabled = true;
    this.timer1.Interval = 10;
    this.timer1.Tick += new System.EventHandler(this.gameLoop);
    //
    // label1
    //
    this.label1.Location = new System.Drawing.Point(216, 16);
    this.label1.Name = "label1";
    this.label1.TabIndex = 0;
    this.label1.Text = "Up";
    //
    // label2
    //
    this.label2.Location = new System.Drawing.Point(216, 40);
    this.label2.Name = "label2";
    this.label2.TabIndex = 1;
    this.label2.Text = "Down";
    //
    // label3
    //
    this.label3.Location = new System.Drawing.Point(216, 64);
    this.label3.Name = "label3";
    this.label3.TabIndex = 2;
    this.label3.Text = "Left";
    //
```

```
        // label4
        //
        this.label4.Location = new System.Drawing.Point(216, 88);
        this.label4.Name = "label4";
        this.label4.TabIndex = 3;
        this.label4.Text = "Right";
        //
        // label5
        //
        this.label5.Location = new System.Drawing.Point(216, 152);
        this.label5.Name = "label5";
        this.label5.TabIndex = 4;
        this.label5.Text = "Left";
        //
        // label6
        //
        this.label6.Location = new System.Drawing.Point(216, 176);
        this.label6.Name = "label6";
        this.label6.TabIndex = 5;
        this.label6.Text = "Wait(Left)";
        //
        // label7
        //
        this.label7.Location = new System.Drawing.Point(216, 200);
        this.label7.Name = "label7";
        this.label7.TabIndex = 6;
        this.label7.Text = "Right";
        //
        // label8
        //
        this.label8.Location = new System.Drawing.Point(216, 224);
        this.label8.Name = "label8";
        this.label8.TabIndex = 7;
        this.label8.Text = "Wait(Right)";
        //
        // label9
        //
        this.label9.Location = new System.Drawing.Point(216, 248);
        this.label9.Name = "label9";
        this.label9.TabIndex = 8;
        this.label9.Text = "Circle";
        //
        // label10
        //
        this.label10.Location = new System.Drawing.Point(216, 272);
        this.label10.Name = "label10";
        this.label10.TabIndex = 9;
        this.label10.Text = "Wait(Left2)";
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(520, 454);
        this.Controls.Add(this.label10);
        this.Controls.Add(this.label9);
```

```
      this.Controls.Add(this.label8);
      this.Controls.Add(this.label7);
      this.Controls.Add(this.label6);
      this.Controls.Add(this.label5);
      this.Controls.Add(this.label4);
      this.Controls.Add(this.label3);
      this.Controls.Add(this.label2);
      this.Controls.Add(this.label1);
      this.Name = "Form1";
      this.Text = "Form1";
      this.Paint += new
System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
      this.ResumeLayout(false);

    }
    #endregion

    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

    // CollisionHandler: Håndterer hvad der sker når sprite1 og sprite2
    // - kolliderer
    void CollisionHandler(object sender)
    {
      // Tilstandsmaskine for sprite2
      switch(sprite2State)
      {
        case Sprite2States.WaitCollisionLeft:
          // Initialiser circel variable og overgå til ny tilstand:
          // WaitCollisionLeft -> Circle
          t=0;
          tempX = sprite2.X;
          tempY = sprite2.Y;
          sprite2State = Sprite2States.Circle;
          break;
        case Sprite2States.WaitCollisionLeft2:
          // WaitCollisionLeft2 -> Right
          sprite2State = Sprite2States.Right;
          break;
        case Sprite2States.WaitCollisionRight:
          // WaitCollisionRight -> Left
          sprite2State = Sprite2States.Left;
          break;
      }
    }

    private void gameLoop(object sender, System.EventArgs e)
    {
      collisions.Check();
      switch(sprite1State)
      {
        case Sprite1States.Down:
```

```
        // Tjek om spriten holder sig inden for y-grænsen
        if(sprite1.Y >= LowerBorder)
          // Down -> Left
          sprite1State = Sprite1States.Left;
        else
        {
          sprite1.Y++;
        }
        break;
      case Sprite1States.Left:
        // Tjek om spriten holder sig inden for x-grænsen
        if(sprite1.X <= LeftBorder)
          // Left -> Up
          sprite1State = Sprite1States.Up;
        else
          sprite1.X--;
        break;
      case Sprite1States.Right:
        // Tjek om spriten holder sig inden for x-grænsen
        if(sprite1.X >= RightBorder)
          // Right -> Down
          sprite1State = Sprite1States.Down;
        else
          sprite1.X++;
        break;
      case Sprite1States.Up:
        // Tjek om spriten holder sig inden for y-grænsen
        if(sprite1.Y <= TopBorder)
          // Up -> Right
          sprite1State = Sprite1States.Right;
        else
          sprite1.Y--;
        break;
    }

    switch(sprite2State)
    {
      case Sprite2States.Left:
        if(sprite2.X <= LeftBorder)
          // Left -> WaitCollisionLeft
          sprite2State = Sprite2States.WaitCollisionLeft;
        else
          sprite2.X--;
        break;
      case Sprite2States.Circle:
        double amp = (LowerBorder-TopBorder)/3;
        sprite2.X = tempX + (int)(amp*(1-Math.Cos(t)));
        sprite2.Y = tempY + (int)(-amp*(Math.Sin(t)));

        if(t >= 2*Math.PI)
        {
          // Circle -> WaitCollisionLeft2
          sprite2State = Sprite2States.WaitCollisionLeft2;
          sprite2.X = tempX;
          sprite2.Y = tempY;
```

```
          }

          t+= Math.PI/100d;
          break;
        case Sprite2States.Right:
          if(sprite2.X >= RightBorder)
            // Right -> WaitCollisionRight
            sprite2State = Sprite2States.WaitCollisionRight;
          else
            sprite2.X++;
          break;
      }
      // Gentegn vinduet
      Invalidate(true);
    }

    // Tegn grafik
    private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
    {
      Graphics g = e.Graphics;
      // Tegn de to sprites
      sprite1.Draw(g);
      sprite2.Draw(g);
      // Tegn et rektangel der hvor spritene holder sig indenfor
      g.DrawRectangle(new Pen(Color.Black),LeftBorder, TopBorder,
        RightBorder-LeftBorder+spriteSize,
        LowerBorder-TopBorder+spriteSize);

      // Følgende kode markerer hvilken tilstand der er aktiv for sprite1
      // - og sprite2
      Label[] labels = new Label[]{
                       label1,
                       label2,
                       label3,
                       label4,
                       label5,
                       label6,
                       label7,
                       label8,
                       label9,
                       label10
                     };
      int index1=0, index2=0;
      switch(sprite1State)
      {
        case Sprite1States.Down:
          index1=1;
          break;
        case Sprite1States.Left:
          index1=2;
          break;
        case Sprite1States.Right:
          index1=3;
          break;
```

```
        case Sprite1States.Up:
          index1=0;
          break;
      }

      switch(sprite2State)
      {
        case Sprite2States.Left:
          index2=4;
          break;
        case Sprite2States.WaitCollisionLeft:
          index2=5;
          break;
        case Sprite2States.Right:
          index2=6;
          break;
        case Sprite2States.WaitCollisionRight:
          index2=7;
          break;
        case Sprite2States.Circle:
          index2=8;
          break;
        case Sprite2States.WaitCollisionLeft2:
          index2=9;
          break;
      }

      // Marker de aktuelle tilstande ved at tegne med gul baggrund
      for(int i=0; i<labels.Length; i++)
      {
        if(i==index1 || i==index2)
          labels[i].BackColor = Color.Yellow;
        else
                    labels[i].BackColor = Color.Transparent;
      }
    }
  }
}
```

BlockSprite.cs

```
using System;
using System.Drawing;

namespace StatemachineDemo
{
  public class BlockSprite
  {
    protected int x=0;
    protected int y=0;
    protected int width=0;
    protected int height = 0;
```

```
    protected Color color = Color.Black;
    protected bool visible = true;

    // Konstruktør, der giver mulighed for at sætte spritens properties
    public BlockSprite(int x, int y, int width, int height)
    {
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
    }

    // Tegning af spriten
    public void Draw(Graphics g)
    {
      if(visible == false) return;
      g.FillRectangle(new SolidBrush(color), x,y, width, height);
    }

    // Sprite properties herunder
    public int X
    {
      get { return this.x; }
      set { this.x = value; }
    }

    public int Y
    {
      get { return this.y; }
      set { this.y = value; }
    }

    public int Height
    {
      get { return this.height; }
      set { this.height = value; }
    }

    public int Width
    {
      get { return this.width; }
      set { this.width = value; }
    }

    public System.Drawing.Color Color
    {
      get { return this.color; }
      set { this.color = value; }
    }

    public bool Visible
    {
      get { return this.visible; }
      set { this.visible = value; }
    }
```

```
  }
}
```

SpriteCollision.cs

```
using System;
using System.Collections;
using System.Drawing;

namespace StatemachineDemo
{
  public delegate void CollisionDelegate(object sender);

  public class SpriteCollision
  {
    ArrayList listeners = new ArrayList();

    // Tilføj en listener, der lytter efter kollision mellem sprite1 og 2
    public void AddListener(BlockSprite s1, BlockSprite s2, CollisionDelegate
d )
    {
      Listener l = new Listener(s1, s2, d);
      listeners.Add(l);
    }

    // Tjek for kollisioner, ved at gennemløbe alle listeners
    public void Check()
    {
      for(int i=0; i<listeners.Count; i++)
      {
        Listener listener = (Listener) listeners[i];
        // Fjern listener hvis remove property'en er true
        if(listener.Remove)
        {
          listeners.Remove(listener);
          i--;
          continue;
        }
        // Tjek om listenerens sprites er kollideret
        if(listener.Collide())
        {
          listener.FireCollisionEvent();
        }
      }
    }
  }

  public class Listener
  {
    BlockSprite sprite1;
    BlockSprite sprite2;
    bool remove = false;
```

```
    event CollisionDelegate CollisionEvent;

    public Listener(BlockSprite s1, BlockSprite s2, CollisionDelegate d)
    {
      sprite1 = s1;
      sprite2 = s2;
      CollisionEvent += d;
    }

    // Tjek for kollision mellem sprite1 og sprite2
    public bool Collide()
    {
      if(sprite1.Visible == false || sprite2.Visible == false) return false;

      // Tjek for kollision her
      if(sprite1.X < (sprite2.X+sprite2.Width))
      {
        if((sprite1.X+sprite1.Width) > sprite2.X)
        {
          if(sprite1.Y < (sprite2.Y+sprite2.Height))
          {
            if((sprite1.Y+sprite1.Height) > sprite2.Y)
            {
              return true;
            }

          }
        }
      }
      return false;
    }

    // Fyr kollisions eventet
    public void FireCollisionEvent()
    {
      CollisionEvent(this);
    }

    public BlockSprite Sprite1
    {
      get { return this.sprite1; }
      set { this.sprite1 = value; }
    }

    public BlockSprite Sprite2
    {
      get { return this.sprite2; }
      set { this.sprite2 = value; }
    }

    // Afgører hvorvist denne listener skal fjernes fra listener kollektionen
    public bool Remove
    {
      get { return this.remove; }
      set { this.remove = value; }
```

```
    }
  }
}
```

Kommentar af skwat d. 07. Nov 2004 | 1

Det er ganske simpelt knald godt, jeg tror at sovsekoder ved hvad de unge vil ha' - skide godt.

Kommentar af cronck d. 13. May 2005 | 2

Er sq'da rart at se at der virkelig findes nogen som gider at bruge tid på at skrive deres artikler ordentligt! Super flot!

Kommentar af zeroaim d. 27. Jan 2005 | 3

Det kunne simpelthen ikke forklares bedre ! (er du i virkeligheden pædagog?):-D
SUPER

Kommentar af sorenbs d. 31. Oct 2004 | 4

Lang og kompleks :)

Kommentar af andr3as d. 02. Nov 2004 | 5

virkelig godt :o) ser frem til den næste i rækken :)

Kommentar af tobiasahlmo d. 24. Jan 2008 | 6

Meget god artikel. Men forstår den ikke helt :)

Kommentar af sovsekoder d. 23. Feb 2011 | 7

kig evt. på mit codeplex projekt ang. XNA og platform spil:

http://mrdev.codeplex.com/