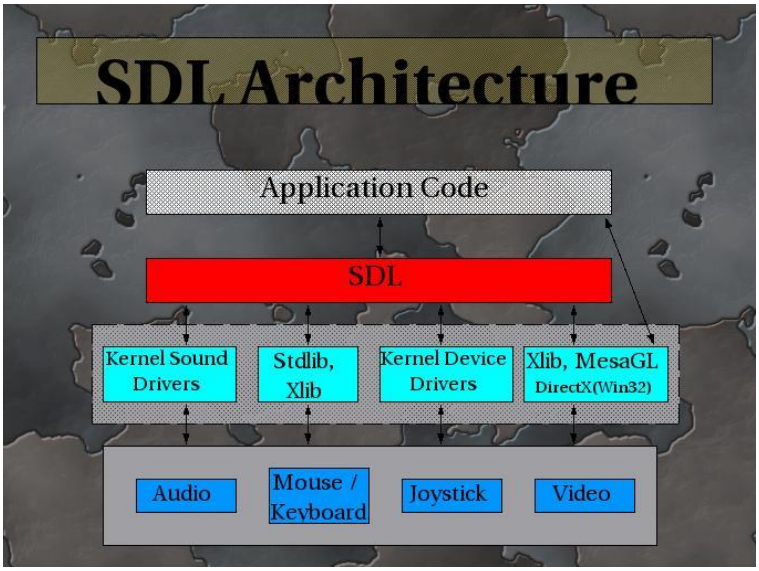
SDL LIBRARY

SESSIE 1: SDL

Er wordt gebruik gemaakt van de SDL –(Simple Direct Layer) component dewelke een cross-platform multimedia library is, ontwikkeld in C om toegang te verkrijgen naar Audio, Keyboard, Mouse, Joystick, 3D Hardware en 2D Video Framebuffer componenten. SDL.Net voorziet bindingen naar de C library. Deze layer wordt onder meer gebruikt door MPEG playback software, emulators, populare computer spelletjes zoals civilization, …

De SDL library is de ideale springplank naar geavanceerde programmeerprojecten door:

- Stabiliteit: SDL werkt bovenop multimedia systemen zoals DirectX, OpenGL
- Simpel: SDL heeft een compacte API



- Portable: Omzettingen naar andere platformen verlopen makkelijk

INLEIDING

Laten we beginnen om de multimedia API te exploreren. SDL bevat subsystemen om verschillende aspecten van een multimedia applicatie af te handelen:

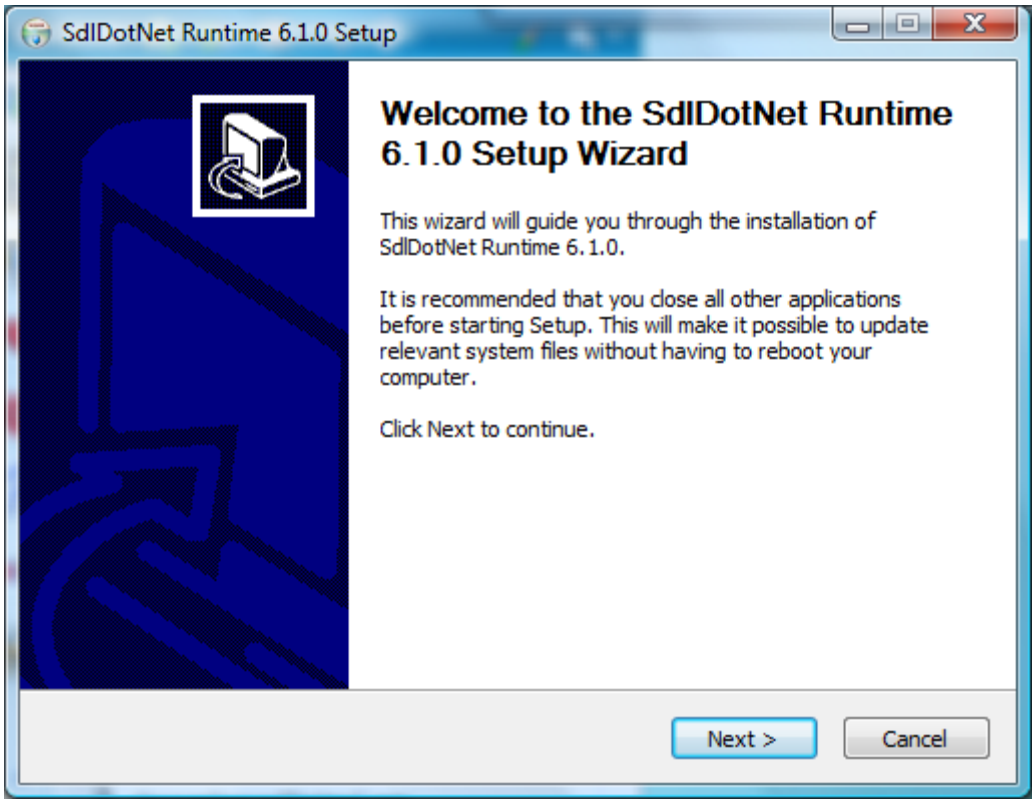
- Video
- JoyStick
- Event Handling
- Audio
- Mouse/Keyboard

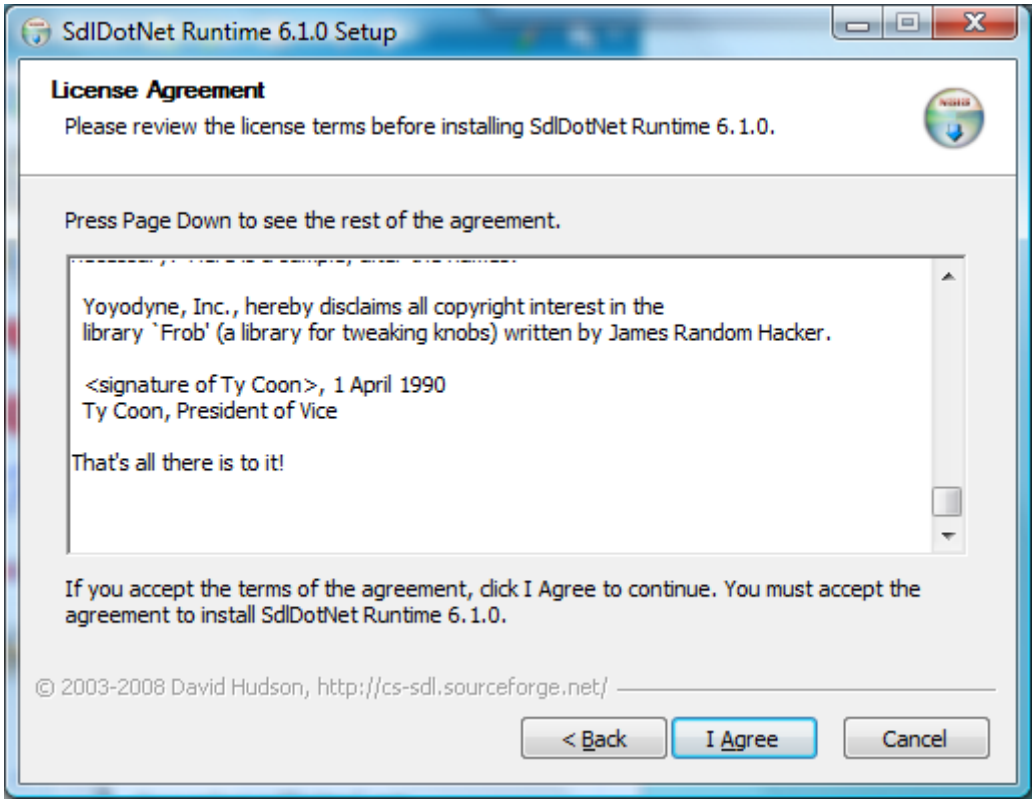
In elke applicatie zal je **video** en **eventhandling** sub-systemen minstens terugvinden, want zonder iets op het scherm of geen mogelijkheden om input te geven kunnen we moeilijk van een applicatie spreken..

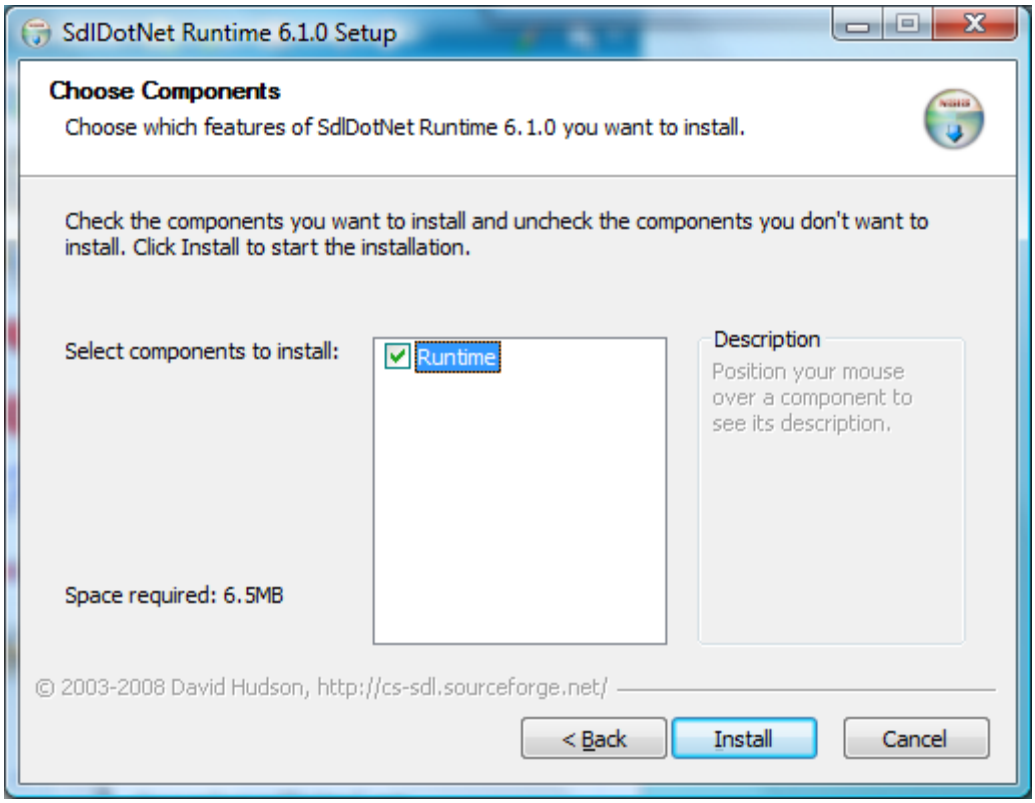
SDL Video werkt met surfaces, rechthoeken, kleuren, color keying, alpha blending…, en kan in full-screen of video mode werken. Maar vooraleer in de code te duiken, gaan we eerst SDL.Net installeren.

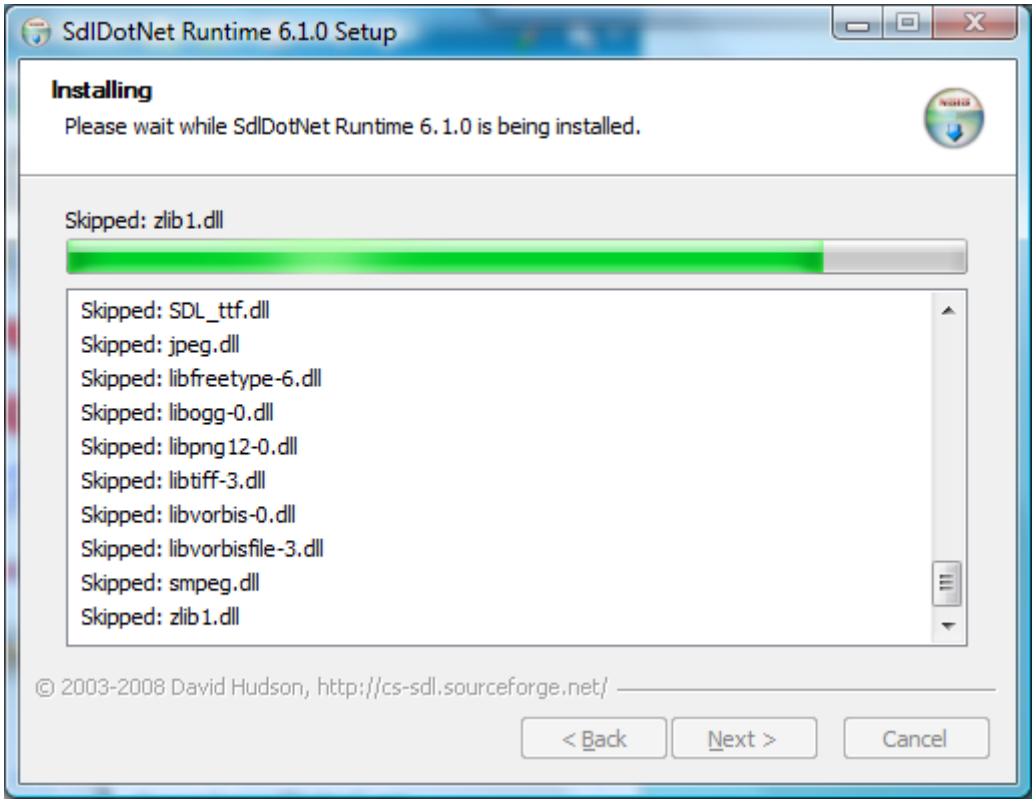
GEBRUIK

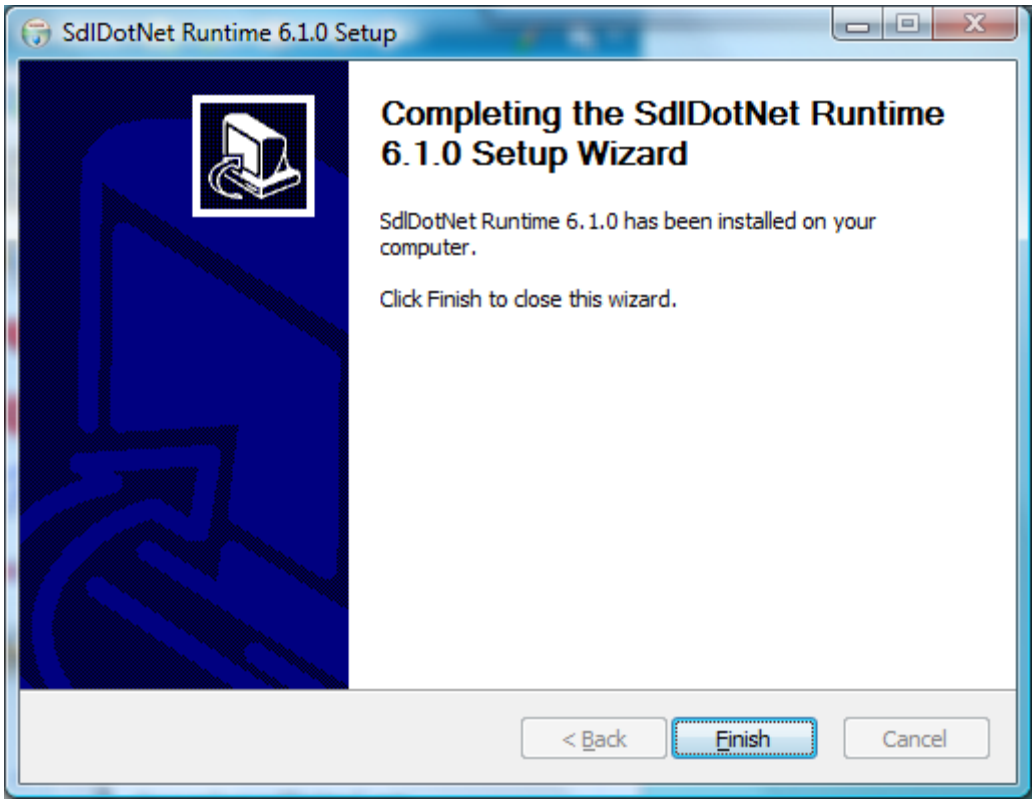
Download sdldotnet-6.1.0-runtime –setup.exe van http://sourceforge.net/projects/cs-sdl/files/ en voer uit.


SdlDotNet Runtime 6.1.0 Setup
Welcome to the SdlDotNet Runtime 6.1.0 Setup Wizard
This wizard will guide you through the installation of SdlDotNet Runtime 6.1.0.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel

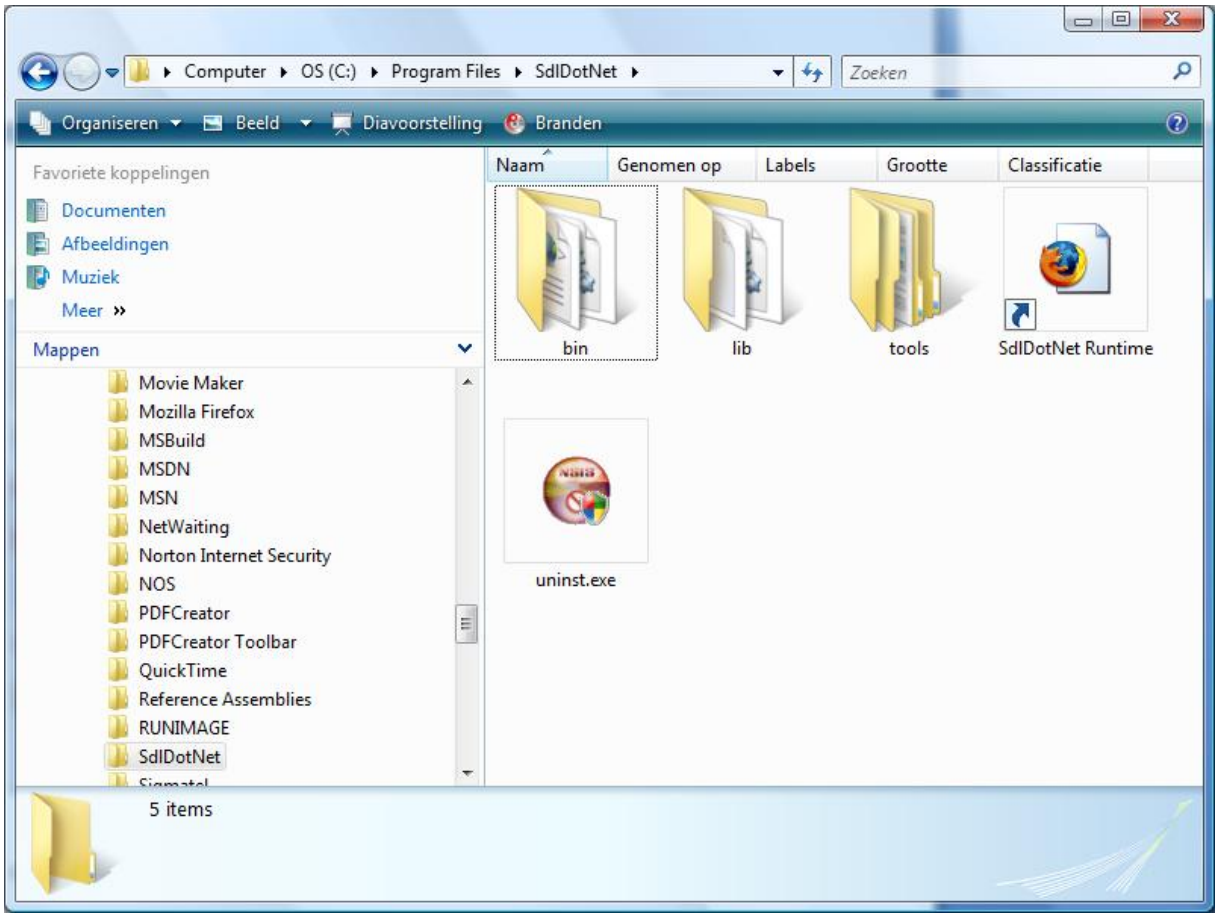

SdlDotNet Runtime 6.1.0 Setup
License Agreement
Please review the license terms before installing SdlDotNet Runtime 6.1.0.
Press Page Down to see the rest of the agreement.
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library `Frob' (a library for tweaking knobs) written by James Random Hacker.
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
That's all there is to it!
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install SdlDotNet Runtime 6.1.0.
© 2003-2008 David Hudson, http://cs-sdl.sourceforge.net/
< Back
I Agree
Cancel


SdlDotNet Runtime 6.1.0 Setup
Choose Components
Choose which features of SdlDotNet Runtime 6.1.0 you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Install to start the installation.
Select components to install:
Runtime
Description
Position your mouse over a component to see its description.
Space required: 6.5MB
© 2003-2008 David Hudson, http://cs-sdl.sourceforge.net/
< Back
Install
Cancel

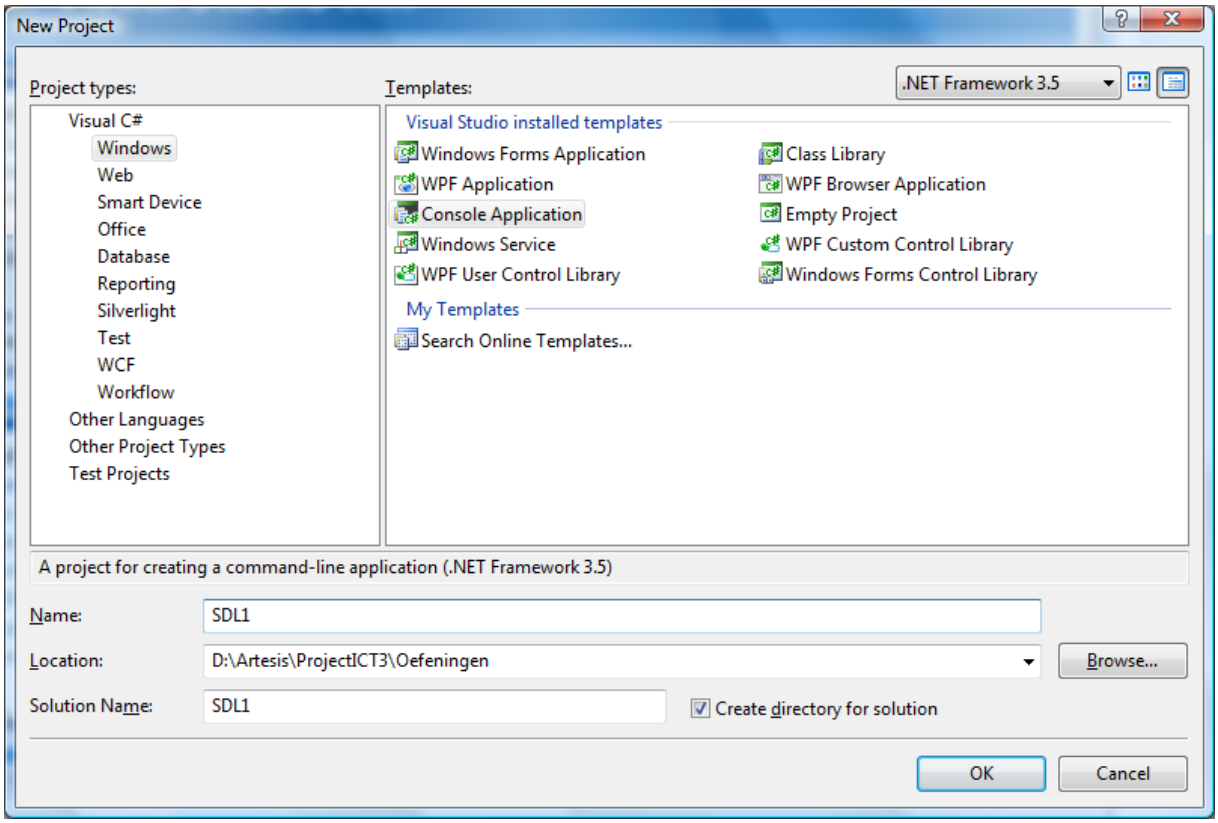

SdlDotNet Runtime 6.1.0 Setup
Installing
Please wait while SdlDotNet Runtime 6.1.0 is being installed.
Skipped: zlib1.dll
Skipped: SDL_ttf.dll
Skipped: jpeg.dll
Skipped: libfreetype-6.dll
Skipped: libogg-0.dll
Skipped: libpng12-0.dll
Skipped: libtiff-3.dll
Skipped: libvorbis-0.dll
Skipped: libvorbisfile-3.dll
Skipped: smpeg.dll
Skipped: zlib1.dll
© 2003-2008 David Hudson, http://cs-sdl.sourceforge.net/
< Back
Next >
Cancel



Geinstalleerd in c:\Program Files\SdlDotNet



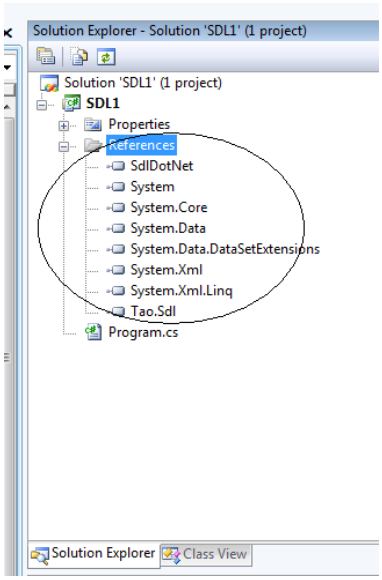
Start Visual Studio op en maak een nieuw Console Project aan:



Om SDL.Net te gebruiken moeten we 2 managed DLL's in ons project toevoegen. Deze zijn te vinden in de bin directory:

- SdlDotNet.dll
- Tao.Sdl.dll

Rechtermuisklik op je project en kies "Add Reference". Dit laat je toe om van de SDL.Net bibliotheek gebruik te maken.



Via het ontwikkelen van projecten zullen we kennis maken met de SDL library en inzicht verschaffen in praktische multimedia concepten.

EERSTE PROJECT – SDL INITIALISATIE

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SdlDotNet.Graphics;                                    (1)
using SdlDotNet.Core;                                        (2)

namespace Artesis.ProjectICT.SDL                             (3)
{
    class Program
    {
        static void Main(string[] args)
        {
            Video.SetVideoMode(640, 480);                    (4)
            Events.Quit += new EventHandler                  (5)
                        <QuitEventArgs>(Events_Quit);
```

```
            Events.Run();                                    (6)
        }
    }

      static void Events_Quit(object sender, QuitEventArgs e)
        {
            Events.QuitApplication();                        (7)
        }
}
```

(1)(2) Om klassen herbruikbaar te maken kan je werken met een klassenbibliotheek (class library).Deze kunnen we importeren in een ander programma met behulp van het using statement. Om de SDL library aan te spreken, gaan we met behulp van het “using” statement de namespace SdlDotNet.Graphics inladen voor de Video component terwijl SdlDotNet.Core dient om de Events klasse aan te spreken.

Let wel, enkel klassen gedefinieerd als public kunnen herbruikt worden. Niet publieke klassen in een assembly (dll) kunnen enkel door de klassen binnen de assembly gebruikt worden!

(3) Onze eigen namespace waarin we zullen coderen. Bedenk steeds een goede naamgeving, met herbruikbaarheid in het achterhoofd!

(4) Initialisatie van het Video sub-systeem. Deze lijn code zal een nieuwe SDL surface maken (met breedte 640 en hoogte 480 pixels). Video is een static class, dus roepen we de methods aan zonder een object te maken. Static classes worden gemaakt om om methodes te organiseren dit niet bij een object horen. Op deze manier initialiseren we het Video SubSysteem!

(6) We subscriben ons op de Quit-handler van de Events klasse

(5) Events is ook een static class en start de Event Loop om events te processen, en zo initialiseren we het Event Handling sub-systeem: dit start het hart van de applicatie!

STATIC CLASSES

Static constructors worden geladen wanneer een klasse wordt geladen, of anders gezegd wordt deze opgeroepen wanneer het programma start. (ofwel static constructors kunnen eigenlijk gewoonweg niet worden opgeroepen).

Static Methods worden niet opgeroepen door object instanties, maar door de klasse definitie zelf. Vb.

```
Class MyMath
{
    Static double DoubleNumber(double n)
    {
        Return n*n;
    }
}
```

Oproepen door: MyMath.DoubleNumber(10),

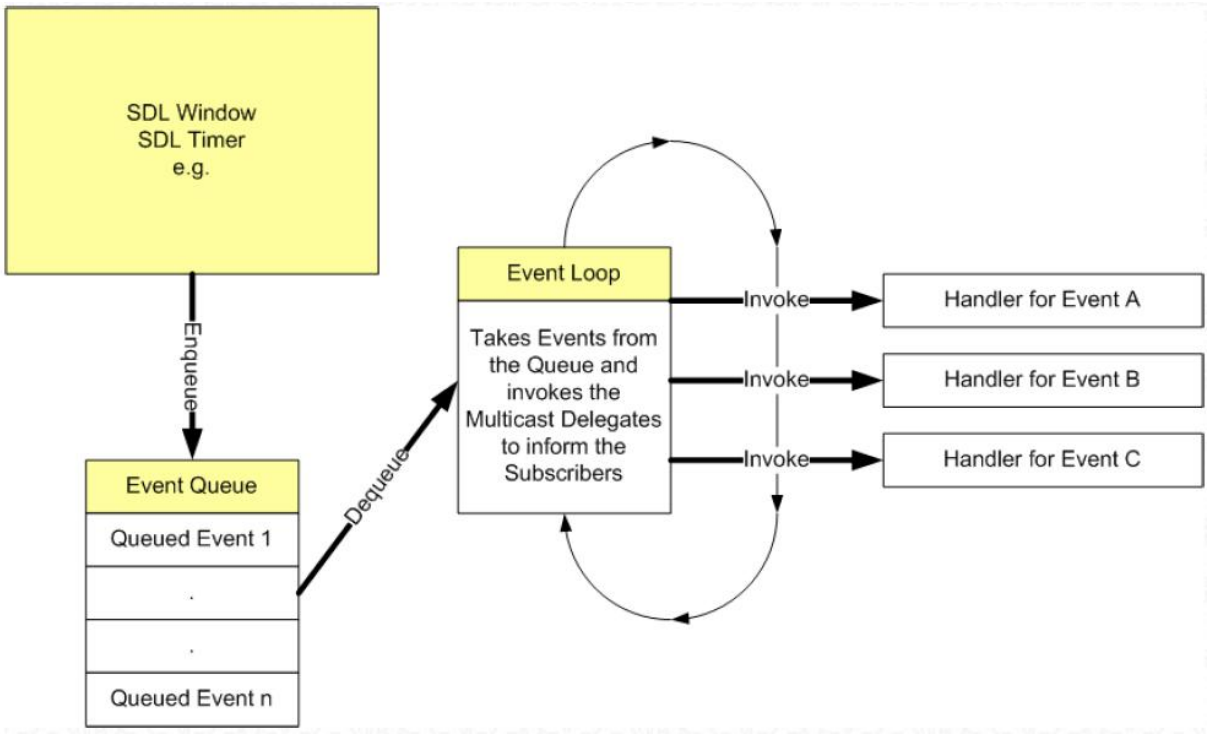
en niet door: MyMath m = new MyMath();

m.DoubleNumber(10);

Static methods zijn nuttig voor vb. wiskundige functies waar data niet wordt gebruikt als deel van de klasse. De static methodes nemen normaal gezien input waarden aan, zullen ze proberen te processen en retourneren een waarde.

EVENT LOOP

Telkens iets gebeurt in SDL, zoals indien een gebruiker een muisknop indrukt, of een Timer Tick loopt af , … wordt er een event gegenereerd. Deze events worden bewaard in de Event Queue, dewelke deze 'dequeued' , een eigen argument klasse instantieert, en roept de multicast delegate aan zodat je applicatie kan reageren op het event.



Met andere woorden telkens je gewaarschuwd wil worden indien een SDL actie (bijvoorbeeld Mouse/keyboard-input) getriggered wordt, moet je jezelf inschrijven op dat specifieke event:

(5) het inschrijven op het SDL Quit event

(7) Dit is de implementatie van de Quit-Handler, met als argument een variabele van het type QuitEventArgs: maw je eigen code wat te doen indien de gebruiker de applicatie afsluit.

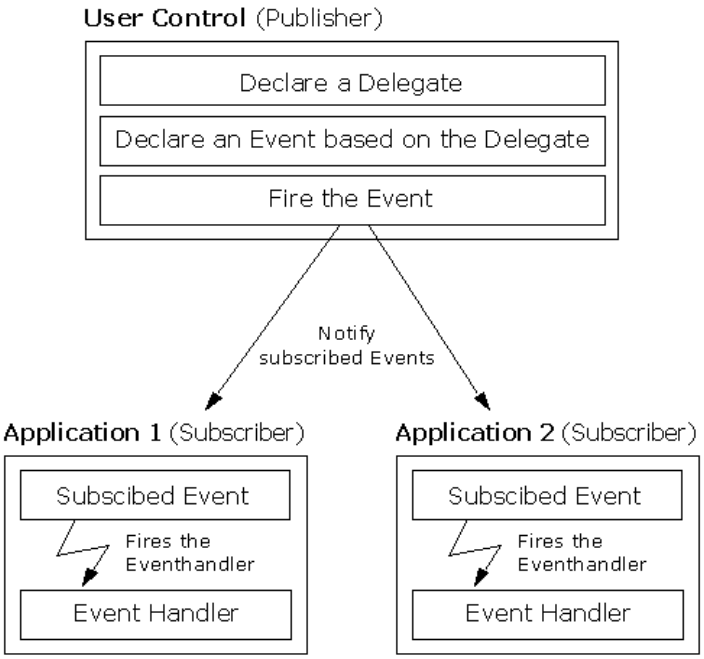
KLOK TICK

Een klok Tick is een interrupt dat periodisch gebeurt, instelbaar volgens de “Frame Rate”, of FPS. Indien je de FPS op 25 zet, zal het Tick event 40 maal per seconde optreden (1000ms / 25 fps). Beschouw dit als het systeem zijn hartslag (in het algemeen is deze tussen de 10 en 200 ms).

NOTA EVENTS

Een event is een C# element die een zekere User defined“ gebeurtenis voorstelt. Een mechanisme die een dynamische vorm van communicatie voorziet tussen andere programma elementen. Events worden gebruikt om geïnteresseerde luisteraars te triggeren, en voorziet een callback functionaliteit in programma’s.

Events voorzien in een “PUBLISH/SUBSCRIBE” patroon, wat wil zeggen dat objecten hun beschikbaarheid om events te genereren te publiceren zodat geïnteresseerde componenten zich kunnen inschrijven op het event. Indien het event getriggered wordt zal de ingeschreven component genotifieerd worden.



Voorbeeld: Stel je hebt een background taak lopen (long task run), en andere partijen zijn geïnteresseerd indien de background taak op een error/trigger loopt. Klasses die hierin geïnteresseerd zijn schrijven zich in op dit event.

Een event wordt gedeclareerd met behulp van een delegate (is een functie pointer):

Stel ik heb een klasse Printer:

```
namespace ConsoleApplication1
{

    public delegate void actieDlgt(string actie);

    public class Printer
    {

        public event actieDlgt actie;

        public Printer()
        {
        }

        public void run()
        {

            //long task run..
            for (; ; )
            {
                // Sleep 1 Second
```

```
                Thread.Sleep(1000);


                if (actie != null)
                {
                    actie("Taak van printer");
                }
            }
        }

    }
}
```

Verder heb ik een klasse UI die geïnteresseerd is in de events van Printer. Dus de UI klasse schrijft zich in op het event, en krijgt boodschappen indien nodig:

```
  public class UI
    {

        public UI()
        {
        }

        public void SubscribePrinter(Printer p)
        {
            p.actie+=new actieDlgt(p_actie);
        }

        void p_actie(string actie)
        {
            Console.WriteLine(actie);

        }
    }
```

Dit alles wordt opgestart in de Main:

```
static void Main(string[] args)
        {
            Printer p = new Printer();
            UI i = new UI();
            i.SubscribePrinter(p);
            p.run();

        }
```

TEKENEN IN SDL.NET

Coordinaten stelsel

Het coördinaten stelsel bij SDL – maar dit is bij de meeste "tekenbibliotheken" het geval – begint in de linker bovenhoek:

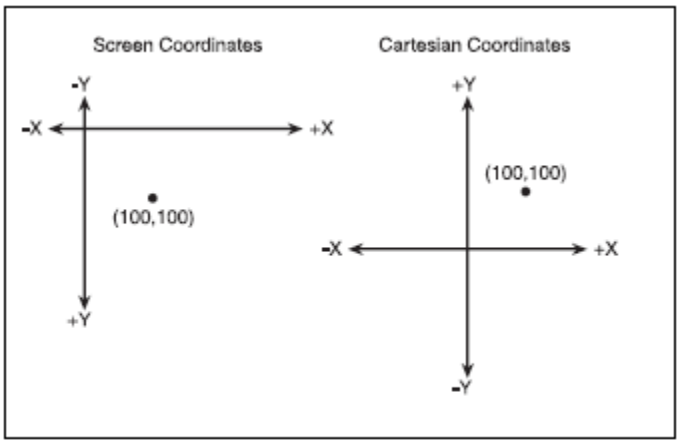


Figure 3.2 *The difference between screen coordinates and standard Cartesian coordinates*

SURFACES

SDL gebruikt surfaces om grafische data voor te stellen. Het is niets meer dan een blok geheugen om een rechthoekige regio van pixels op te slaan. Elke Surface heeft een breedte (width) en hoogte (height) en een specifiek pixel formaat. Het is te beschouwen als de belangrijkste bouwsteen van het Video sub-systeem.

Er is steeds 1 surface die de display voorstelt,(Het scherm is dus ook een surface, geinitializeert door Video.SetVideoMode() – *zie eerste project*).

```
Surface m_Video      = Video.SetVideoMode(640, 480);
```

Alle andere surfaces zijn buffers om pixel data in te bewaren en die eventueel op het hoofdscherm getoond zullen worden.

```
Surface s = new Surface(WIDTH, HEIGHT, PIXELDEPTH,
ALPHACHANNEL);
```

SDL laadt beelden rechtstreeks in een surface object. Surfaces kunnen worden gekopieerd op elkaar, wat **blitting** wordt genoemd. (**Blit = Block Image Transfer**).

Daar het scherm ook een Surface is, kunnen beelden naar het scherm worden gestuurd met 1 blit operatie. Op het Surface object kan men de "Convert"-method aanroepen, die het geladen beeld naar de pixel depth van het scherm Surface omzet.

Pixel depth refereert naar het aantal kleuren mogelijk, of hoeveel bits gebruikt worden om een individuele pixel te creeren.

(dit is op te vragen door : `Video.Screen.BitsPerPixel)`

```
m_Background = ( new Surface( @"D:\temp\DemoBackground.png"
)).Convert ( m_VideoScreen, true, false );
```

Indien de Convert method niet aangeroepen wordt, zal telkens het beeld geblit wordt een automatische conversie plaats vinden, wat tijd- en geheugen rovend is.

RECTANGLES

Het Video subsystem is 2D en heeft voornamelijk als taak om een rechthoekige blok pixels te kopieren van de ene surface naar de andere. (dit is de manier om met rasterdisplays te werken). Er bestaat dus een Rectangle object om met dergelijke data om te gaan.

Een rectangle heeft (x,y) coördinaten die de linkerbovenhoek specifieren, en tesamen met de width en height bepaalt men de rechthoek oppervlakte. Een rechthoek heeft dus enkel maar 2 punten nodig (parallel) om de deze te definieren, het rechteronderpunt is dus x+ width en y+height. Het bewaren van de width en height is dus nuttig, want we kunnen eenvoudig weg de rectangle rondbewegen door simpelweg de x en y coördinaten aan te passen.

TWEEDE PROJECT

Tweede project: We maken een nieuwe surface aan en “blitten” er een afbeelding in. Daarna zullen we er een tweede afbeelding bovenop tonen (lees “blitten”). Maak gebruik van volgende afbeeldingen:

<< backgroundImage.gif >>

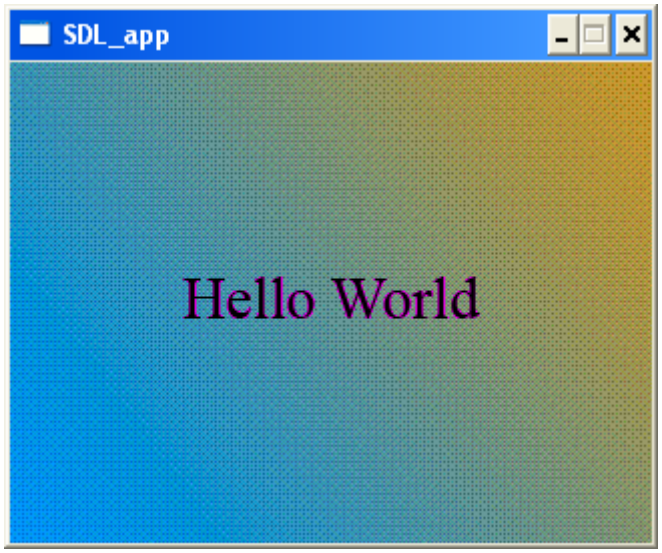
<< voorgrond.tif >>

```
public class TweedeProject
    {
        private  Surface m_VideoScreen;
        private  Surface m_Background;
        private  Surface m_Foreground;
        private  Point m_ForegroundPosition;

        public TweedeProject()
        {

        }

        public void Run()
        {
            m_VideoScreen = Video.SetVideoMode(320, 240, 32, false, false,
            false, true);
            LoadImages();
            Events.Quit += new EventHandler<QuitEventArgs>(Events_Quit);
            Events.Tick += new EventHandler<TickEventArgs>(Events_Tick);
            Events.Run();
        }

        void Events_Tick(object sender, TickEventArgs e)
        {
            m_VideoScreen.Blit(m_Background);
            m_VideoScreen.Blit(m_Foreground, m_ForegroundPosition);
            m_VideoScreen.Update();

        }

        void Events_Quit(object sender, QuitEventArgs e)
        {
            Events.QuitApplication();
        }

        private void LoadImages()
        {
            m_Background = (new
            Surface(@"backgroundImage.gif")).Convert(m_VideoScreen, true,
            false);

            m_Foreground = (new
            Surface(@"foregroundImage.gif")).Convert(m_VideoScreen, true,
            false);
```

```
            m_Foreground.Transparent = true;
            m_Foreground.TransparentColor = Color.FromArgb(255, 0, 255);

            m_ForegroundPosition = new Point(m_VideoScreen.Width / 2 -
            m_Foreground.Width / 2,m_VideoScreen.Height / 2 -
            m_Foreground.Height / 2);


        }
    }
```

WAT DOEN WE IN DEZE CODE:

1. We creeren onze Video Surface (initialiseren van het Video Sub-Systeem).
2. We creeren onze achtergrond surface met een gif bestand
3. We creeren onze tweede afbeelding (voorgrond) met een gif bestand
4. We blitten elk frame beide beelden op het video scherm, en laten het zien.

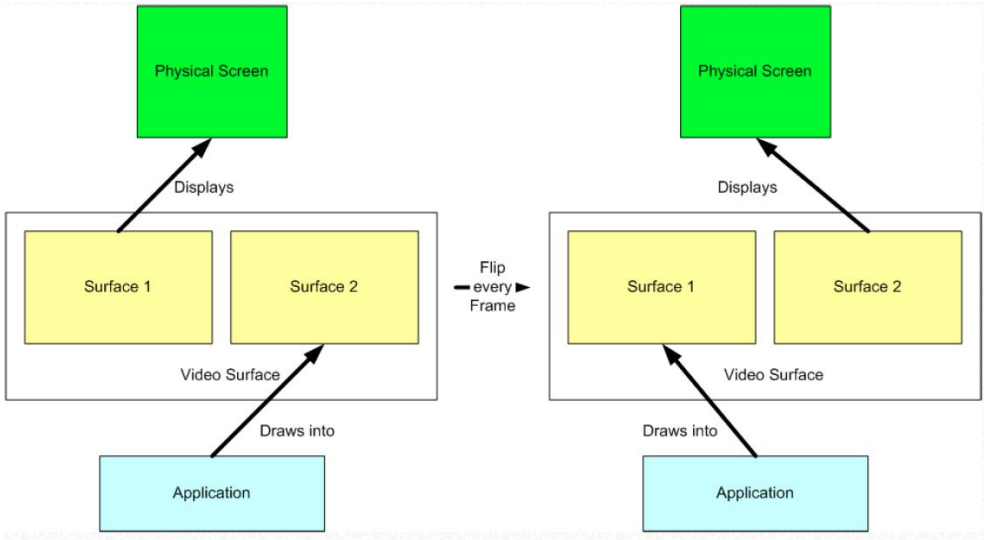


We laden een afbeelding vanuit disk , en creeren een nieuwe compatibele Surface (dit is nodig want de geladen afbeelding heeft misschien niet dezelfde pixel diepte dan onze grafische kaart . zonder creatie van een compatibele Surface zouden we bij elke blit een conversie berekening moeten uitvoeren , wat de performantie aanzienlijk zou verminderen.: daarom het gebruik van de Convert methode op het Surface object.

In elke frame wordt ons videoscherm geupdate door volgende regel:

```
_VideoScreen.Update();
```

Achterliggend maken we gebruik van “double buffering” wat betekent dat er steeds 2 Surfaces in de achtergrond geladen zijn, en 1 ervan wordt getoond op het scherm, terwijl de andere als “Back”-Buffer gebruikt wordt om te manipuleren (~tekenen). Zonder double buffering zal Flickering optreden.



PROJECT DRIE : COLOR KEYING, ALPHA, ALPHA CHANNELS

Wanneer je Surfaces blit , zal je niet altijd het gehele rechthoekige Surface willen blitten , maar slechts een gedeelte ervan (mogelijk door een rectangles mee te geven tijdens blit-operatie:

```
m_VideoSurface.Blit(m_Sprite, destinationRectange, sourceRectangel);
```
).

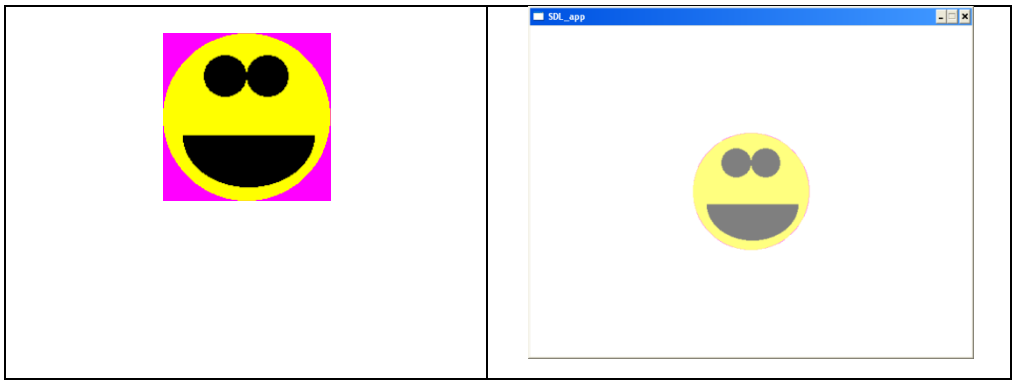
Maar ook wil je sommige delen transparant, of semi-transparant. Daarom volgende items:

- **Color Keying** : het specifieren van een kleur dewelke wordt genegeerd tijdens de Blit-operatie.

- Alpha: specifieert hoe transparant of opaque het gehele surface is
- Alpha Channel : normaal hebben kleurenafbeeldingen drie kleuren voor elke pixel: rood , groen, blauw. Sommige afbeeldingen hebben een extra Alpha-Channel dewelke een extra layer is (1 byte per pixel). Deze byte geeft aan hoe transparant de overeenkomstige pixel is (0 is opaque, 255 is transparant).

```
public class DerdeProject
    {
        private Surface m_VideoSurface;
        private Surface m_Sprite;

        public DerdeProject()
        { }

        public void Run()
        {

            m_VideoSurface = Video.SetVideoMode( 640, 480, false,
            false, false, true );
                        m_Sprite = new
            Surface(@"SmileyNoAlphaChannel.tif").Convert(m_VideoSurfa
            ce, true, true);

            m_Sprite.Alpha = 128;
            m_Sprite.AlphaBlending = true;
            m_Sprite.Transparent = true;
            m_Sprite.TransparentColor = Color.FromArgb(255, 0, 255);


            Events.Tick += new
            EventHandler<TickEventArgs>(Events_Tick);
         Events.Run();
        }

        void Events_Tick(object sender, TickEventArgs e)
        {
            m_VideoSurface.Fill(Color.White);
            m_VideoSurface.Blit(m_Sprite, new
            Point(m_VideoSurface.Width / 2 -
         m_Sprite.Width / 2,
            m_VideoSurface.Height / 2 - m_Sprite.Height / 2));
            m_VideoSurface.Update();
        }
    }
```

Bovenstaande code zou onderstaande afbeelding in het videoscherm op een witte achtergrond, en de magenta achtergrond van de afbeelding mag niet zichtbaar zijn, ok hebben we deze surface op 50% transparant gezet.




```
m_Sprite.Alpha = 128;
m_Sprite.AlphaBlending = true;
```

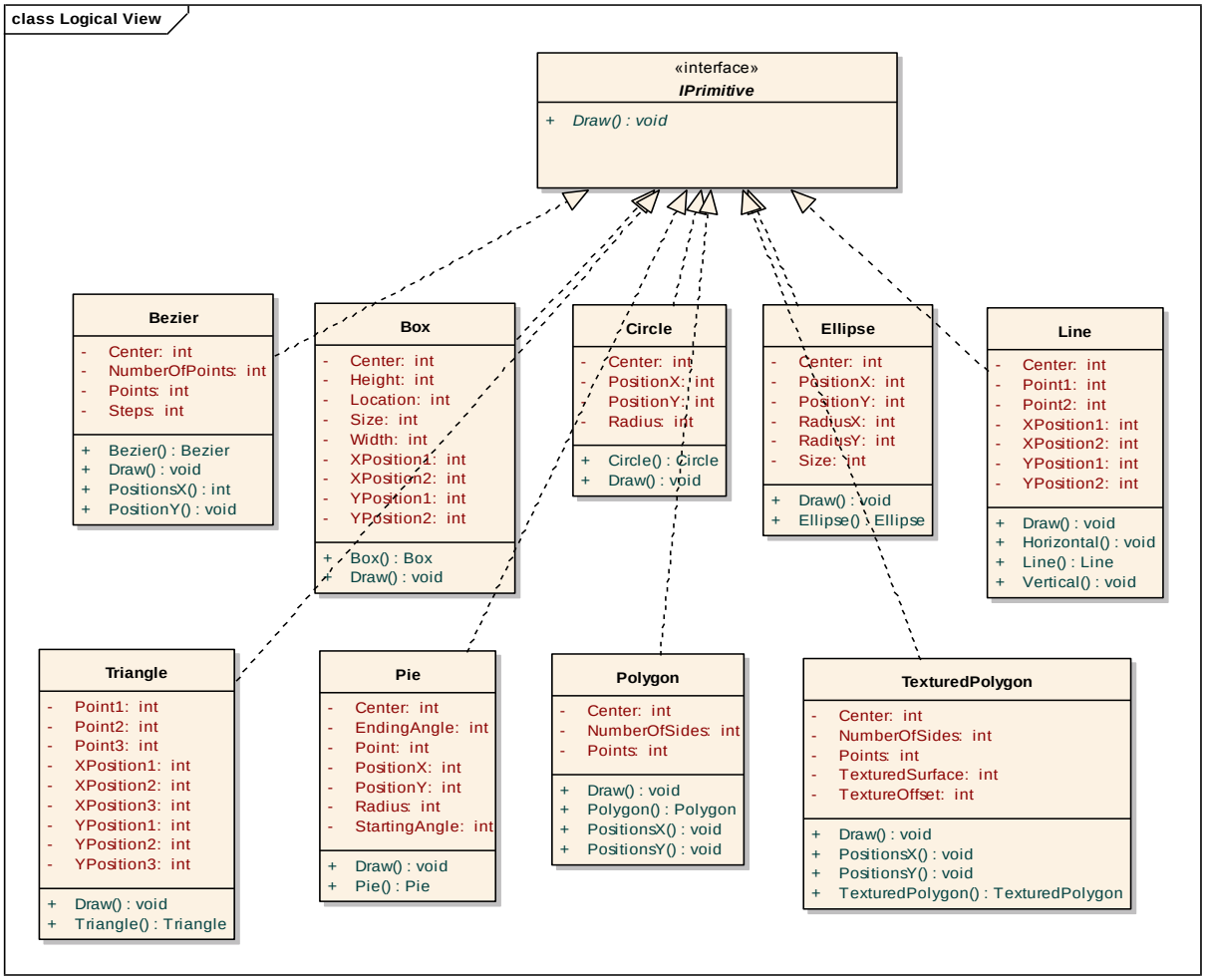
Bovenstaande lijnen specifieren dat deze Surface een Alpha heeft van 128 (dus half opaque).

We moeten dit wel activeren door AlphaBlending op True te zetten.

PRIMITIVES (SDLDOTNET.GRAPHICS.PRIMITIVES)

Primitieven in de SDL.Netl library zijn te beschouwen als basis teken-elementen, zoals een rechthoek, circel, lijn… Elk Surface object heeft een functie draw die dergelijke primitieven tekent in een Surface.

Het klasse diagramma van de primitives ziet er als volgt uit:



Alle klasses zijn gerealiseerd door de interface IPrimitives (deze implementatie verplicht elke klasse om een Draw functie te implementeren.

VIERDE PROJECT

```
video = Video.SetVideoMode(500, 500);
Circle c = new Circle(new Point(250, 250), 50);
c.Draw(video, Color.Yellow);
video.Update();
Events.Run();
```

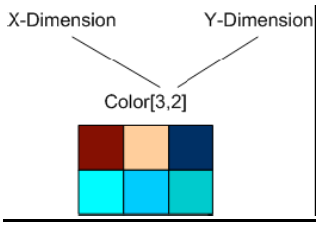
PIXEL MANIPULATIE

Met de SDL library is het ook mogelijk om individuele pixels te manipuleren. Er zijn 2 mogelijke functies beschikbaar:

Color GetPixel (Point p) // weergave van de pixelwaarde op positie x,y in de surface.

Void SetPixel(point p, Color[,] color : we zetten de kleur op positie p

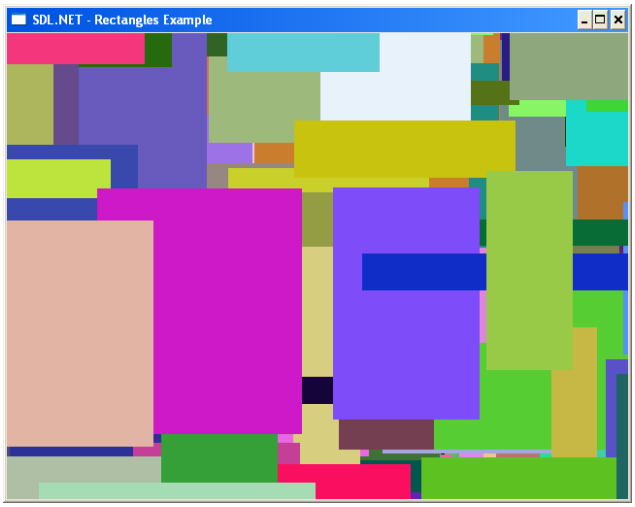
De array:



OEFENREEKS

1. HET VIDEO SUBSYSTEEM HEEFT EEN FILL METHOD OM ER KLEUREN AAN TOE TE KENNEN. IMPLEMENTEER EEN MECHANISME OM ELKE HALVE SECONDE EEN RANDOM KLEUR OP HET SCHERM TE TOVEREN.

2. EEN UITBREIDING OP VORIGE OEFENING. TEKEN OP RANDOM PLAATSEN IN JE VIDEOSCHERM RECHTHOEKEN TUSSEN 20 EN 300 PIXELS BREEDTE EN HOOGTE IN RANDOM VERSCHILLENDE KLEUREN. DEZE RECHTHOEKEN BLIJVEN EENS GEBLIT OP HET SCHERM STAAN (TENZIJ ER EEN RANDOM RECHTHOEK DEZE OVERSCHRIJFT NATUURLIJK).



3. SCENE VERANDERINGEN ZIJN FACTOREN WAARMEE EEN DEVELOPER TIJDENS IMPLEMENTATIE REKENING MEE MOET HOUDEN. TIJDENS DEZE OEFENING ZULLEN WE EEN VIEWPORT ONTWIKKELEN.

a. Programmeer een ‘cut’-scene verandering tussen background1 en background2.

b. Programmeer een ‘Wipe’ scene verandering van links naar rechts.

c. Programmeer een ‘Wipe’ scene verandering van boven naar onder.

Voor A2 en A3 : we verzetten geen frame rate, maar trachten met een mechanisme vertraging in te bouwen zodat de wipe overgang geleidelijk aan verloopt.

.WERKEN MET PRIMITIVES.

- - - - - - - -- - - - - - - -- - - - - - - -- - - - - - - -- - - - - - - -- - - - - - - -- - - - - - - -

4. IMPLEMENTEER EEN ZOOM FUNCTIE

A. Gebruik replica-techniek
B. Maak gebruik van interpolatiemethodes:
    a. Nearest neighbour
    b. Bilinear interpolatie

(Uitleg over de interpolatiemethoden: zie bijlage!)

5.MAAK EEN IMAGE ENHANCEMENT APPLICATIE MET BEHULP VAN HISTOGRAM EQUALIZING.

Werkwijze:

- 1. Bereken Histogram
- 2. Bereken Cummulatieve histogram (de waarde van het histogram op punt i is gelijk aan het aantal voorafgaande waarden van het histogram tot dat punt
- 3. Doe Transformatie

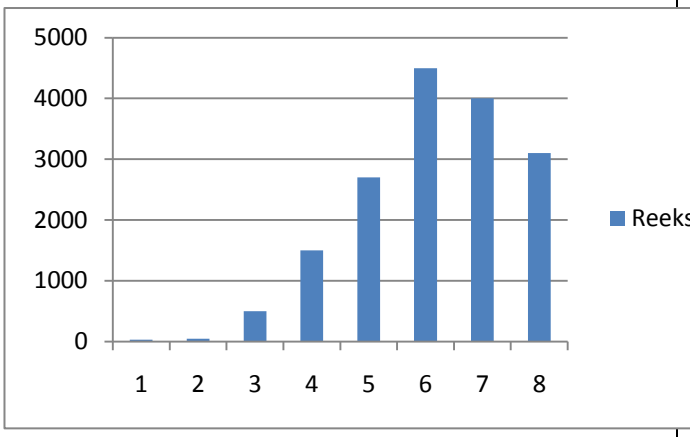
g(i) =

$$\frac{M-1}{n_t}\sum_{j=0}^{i} n_i$$

Voorbeeld:

- Veronderstel een 128x128 afbeelding met 8 mogelijke grijswaarden per pixel., het histogram ziet er als volgt uit:

Gray Level	Number of Pixels
0	34
1	50
2	500
3	1500
4	2700
5	4500
6	4000
7	3100



Het cummulatieve histogram gaat als volgt:

grijswaarden	Cumm.
0	34
1	84
2	584
3	2084
4	4784
5	9284
6	13284
7	16384

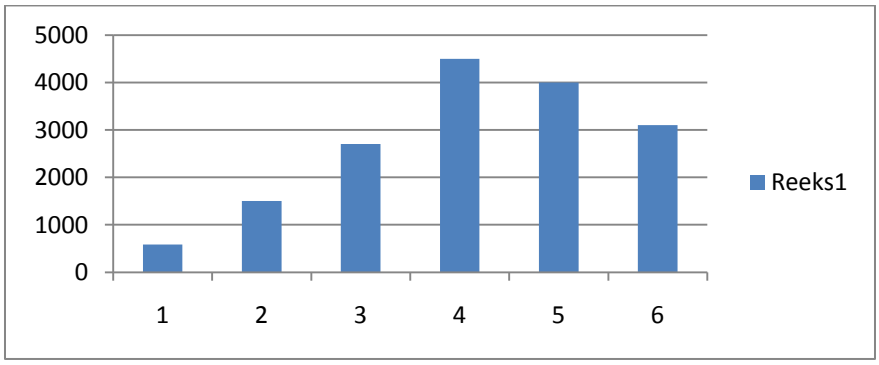
Dankzij de transformatie worden de grijswaarden als volgt gemapt:

grijswaarden gemapt naar grijswaarde

0	0.016
1	0.04
2	0.2852
3	1
4	2
5	5
6	6
7	7

Dit wil zeggen, pixelwaarden 0,1 en 2 worden naar 0 gemapt, enzovoort.

Het histogram ziet er nu als volgt uit:



BIJLAGE

It is interesting to consider what is going on behind the scene when an image processing program resamples an image. The simplest method for upsampling is ***replication***, a process of inserting pixels and giving them the color value of a neighboring pre-existing pixel. Replication works only if you are enlarging an image by an integer factor. An example of replication on a small block of pixels is shown in the Figure below. For each original pixel, a new pixel with the same color value is inserted to the right, below, and diagonally below and to the right.

220	230	240
235	242	190
118	127	135

Original pixels

220	220	230	230	240	240
220	220	230	230	240	240
235	235	242	242	190	190
235	235	242	242	190	190
118	118	127	127	135	135
118	118	127	127	135	135

Pixels after replication

Correspondingly, the simplest method of downsampling is *row-column deletion*, the inverse of replication

Think about how replication and its inverse might affect the quality of an image. Row-column deletion throws away information about the image, so you obviously lose detail. Replication, on the other hand, makes a guess about the colors that might have been sampled between existing samples—if the sampling rate had been higher. The values that are introduced in replication may not be the exact colors that would have been detected if the image had been sampled at a higher resolution to begin with. Thus, even though an upsampled image gains pixels, it doesn't get any sharper. In fact, usually the image loses quality. Since the new pixel values are copied from neighboring pixels, replication causes blockiness in the resampled image. Magnifying a view of an image in an image processing program can be done with simple replication. The image gets bigger, but the blockiness caused by upsampling becomes increasingly evident the more you zoom in. This is illustrated in the Figure below. Of course there is no harm done to the file, since the pixel values are upsampled only for display purposes. The values stored in the image file don't change

You'll notice a similar effect if you upsample a digital image in order to print it out at a larger size. Say that you scan in a 4 × 5 inch image at 200 pixels per inch but decide that you want to print it out at a size of 8 × 10 inches. Increasing the size and keeping the resolution at 200 ppi requires upsampling, since you'll end up with more pixels than you originally captured. But keep in mind that the picture you print out at 8 × 10 inches can't be any clearer or more detailed than the original 4 × 5 inch image.

The point is that the only true information you have about an image is the information you get when the image is originally created—by taking a digital photograph or scanning in a picture. Any information you generate after that—by upsampling—is only an approximation or guess about what the original image looked like. Thus, it's best to scan in digital images with sufficient pixel dimensions from the outset. For example, you could scan the 4 × 5 inch image at 400 ppi so that you could then increase its print size to 8 × 10 inches without resampling. If you deselect "resample image" in your image processing program and then change the image's size to 8 × 10 inches, the number of pixels per inch will change so that the total number of pixels does not change. That is, the resolution will automatically be cut in half as the print size is doubled. Then you'll have your 8 × 10 inches at a resolution of 200 ppi, which may be good enough for the print you want to make, and you won't have had to generate any new pixels that weren't captured in the original scan.

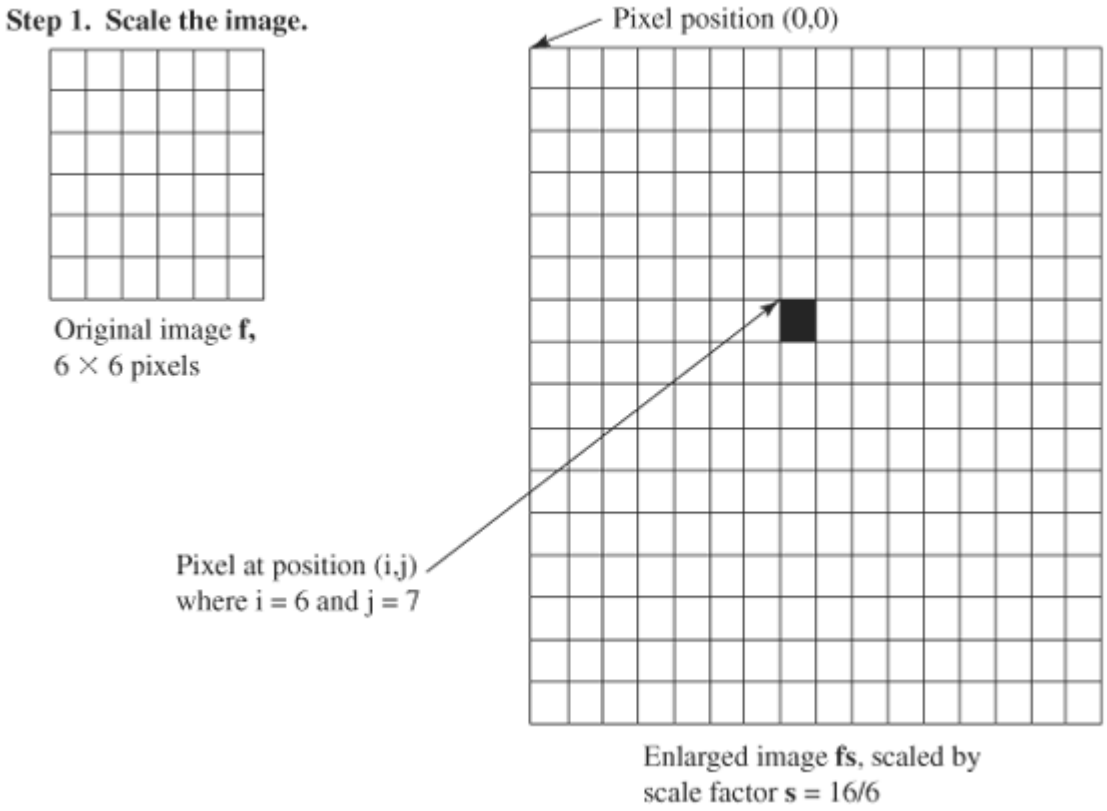
Even with your best planning, there will still be situations that call for resampling. Fortunately, there are interpolation methods for resampling that give better results than simple replication or discarding of pixels. ***Interpolation*** is a process of estimating the color of a pixel based on the colors of neighboring pixels.

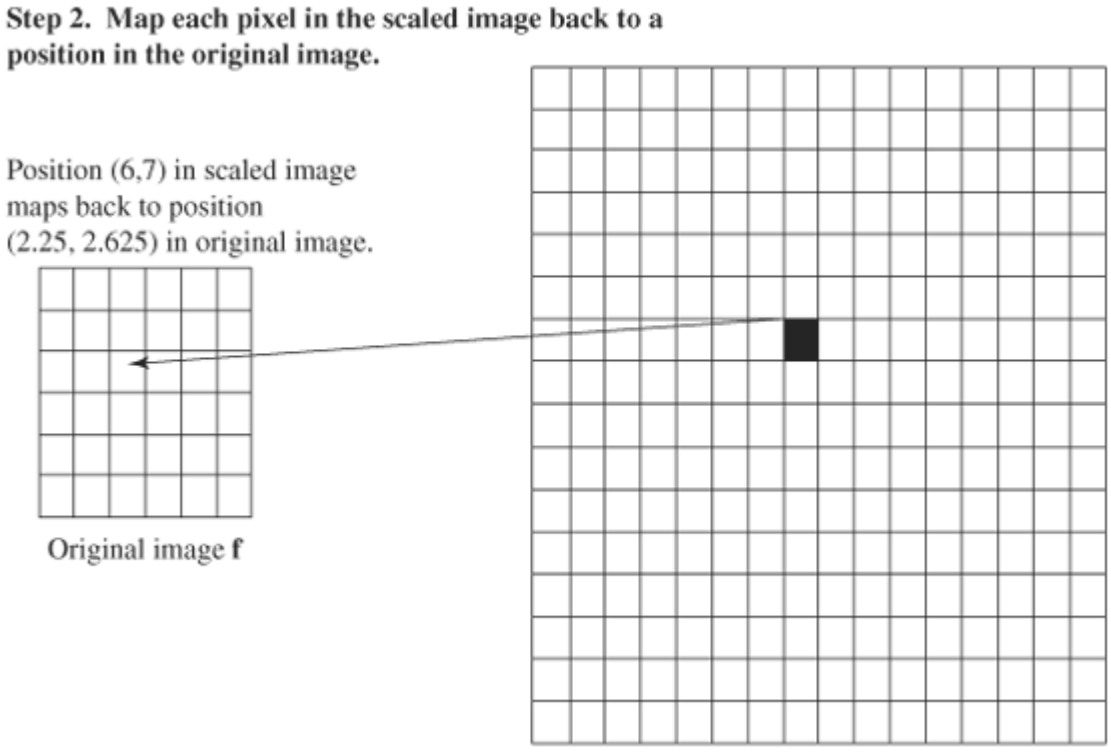
You can choose the interpolation method: nearest neighbor, bilinear, or bicubic. Nearest neighbor is essentially just replication when the scale factor is an integer greater than 1. However, it can be generalized to non-integer scale factors and described in a manner consistent with the other two methods, as we'll do below. We'll describe these algorithms as they would be applied to grayscale images. For RGB color, you could apply the procedures to each of the color channels.

```
algorithm resample
/*Input: A grayscale image f of dimensions w × h.
Scale factor s.
Output: fs, which is image f enlarged or shrunk by scale factor s.*/
{
   w' = (integer)(w * s)
   h' = (integer)(h * s)
/* create a new scaled image fs with dimensions w' and h'*/
   for i = 0 to h' * minus; 1
     for j = 0 to w' - 1 {
     fs(i, j) = interpolate(i, j, f, method)
   }
}
```

```
algorithm interpolate(i, j, f, method) {
/*i and j are pixel coordinates in the scaled image fs.*/
  a = i/s /*This is real-number division since s is real.*/
  b = j/s
/*a and b are coordinates in the original image, f. Note that a and b are not
necessarily integers. Interpolation entails finding integer-valued coordinates
that are neighbors to a and b.*/
   if method = "nearest neighbor" then
      return f(round(a), round(b))
   else if method = "bilinear" then {
/* Find the coordinates of the top left pixel in the neighborhood of (a, b) */
   x = floor(a)
   y = floor(b)
/*Each pixel's weight is based on how close it is to (a, b)*/
   value = 0
   for m = 0 to 1 {
     for n = 0 to 1 {
       t = a - (x + m)
       u = b - (y + n)
       value = value + (1 - |t|)*(1 - |u|)* f(x + m, y + n)
     }
   }
   return value
  }
```

The algorithm describes two different interpolation methods—nearest neighbor, bilinear.. All two begin with the same two steps. First, the dimensions of the scaled image are determined by multiplying the original dimensions by the scale factor. A new bitmap of the scaled dimensions is created. In the example, a 6 × 6 pixel image called ***f*** is enlarged to a 16 × 16 image called ***fs***, so the scale factor *s* is 16/6. The values of the pixels in the scaled image are to be determined by interpolation. The second step is to map each pixel position in the scaled image back to coordinates within the original image. A pixel at position (*i*, *j*) in ***fs*** maps back to position (*i/j*, *j/s*) in ***f***. Clearly, not all pixel positions in ***fs*** map back to integer positions in ***f***. For example, (6, 7) maps back to (2.25, 2.625). The idea in all three interpolation algorithms is to find one or more pixels close to position (*i/s*, *j/s*) in ***f***, and use their color values to get the color value of ***fs***(*i*, *j*).

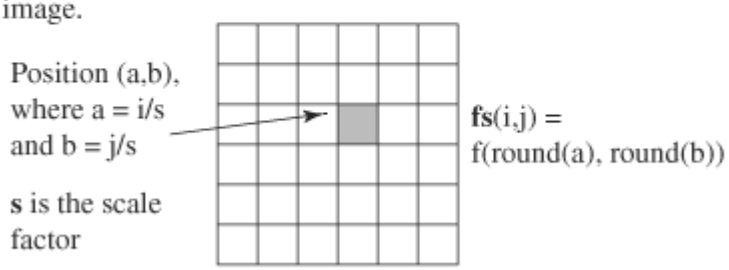




Nearest neighbor interpolation simply rounds down to find one close pixel whose value is used for ***fs***(*i*, *j*) .In our example, ***f***(2, 3) is used as the color value of ***fs***(6, 7). If you think about this, you'll realize that when *s* is an integer greater than 1, the nearest neighbor algorithm is effectively equivalent to pixel replication. However, it also works with noninteger scale factors, as shown in our example.

a and b are not necessarily integers, so they don't necessarily correspond to actual pixel coordinates in the original image.

The **nearest neighbor algorithm** assigns to **fs**(i,j) the color value f(round(a), round(b)) from the original image.



The nearest neighbor is marked in gray.

To describe the two interpolation algorithms discussed earlier, it is sufficient to specify a mask (also called convolution mask!) for each. (Explain the interpolation with a mask is easier programming work!!)

Let's do this for nearest neighbor interpolation. We have a $w \times h$ image called ***f*** that is being scaled by factor s, with the result being written to a new image bitmap ***fs*** of dimensions $w' = w * s$ and $h' = h * s$. The color value of each pixel ***fs***(i, j) is obtained by mapping (i, j) back to coordinates in ***f*** with $a = i/s$ and $b = j/s$.

A convolution mask is nothing more than an array with weighted coefficients in!

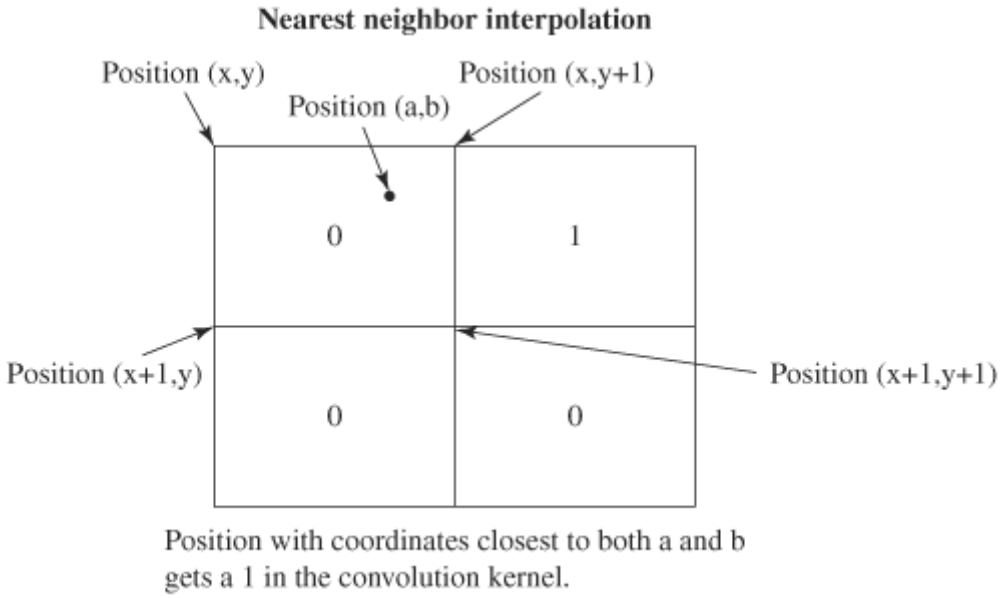
Let $x = floor(a)$ and $y = floor(b)$. Then the neighborhood of (a, b) in ***f*** consists of $f(x, y)$, $f(x + 1, y)$, $f(x, y + 1)$, and $f(x + 1, y + 1)$. For each pixel ***fs***(i, j), define the nearest neighbor convolution mask $h_{nn}(m, n)$ to be a 2×2 matrix of coefficients as follows:

for $0 \leq m \leq 1$ and $0 \leq n \leq 1$,

$h_{nn}(m, n) = 1$ if $-0.5 \leq (x + m) - a < 0.5$ and $-0.5 \leq (y + n) - b < 0.5$

otherwise $h_{nn}(m, n) = 0$

This effectively puts a 1 in the pixel position closest to (*a*, *b*) and a 0 everywhere else, so ***fs***(*i*, *j*) takes the value of its single closest neighbor, as shown in the example in the figure. Note that the position of the 1 in the mask depends on the location of (*a*, *b*). The mask is applied with its upper left element corresponding to ***f***(*x*, *y*).



Bilinear interpolation uses four neighbors and makes ***fs***(*i*, *j*) a weighted sum of their color values. The contribution of each pixel toward the color of ***fs***(*i*, *j*) is a function of how close the pixel's coordinates are to (*a*, *b*). The neighborhood is illustrated in the Figure The method is called bilinear because it uses two linear interpolation functions, one for each dimension. This is illustrated in the figure. Bilinear interpolation requires more computation time than nearest neighbor, but it results in a smoother image, with fewer jagged edges.

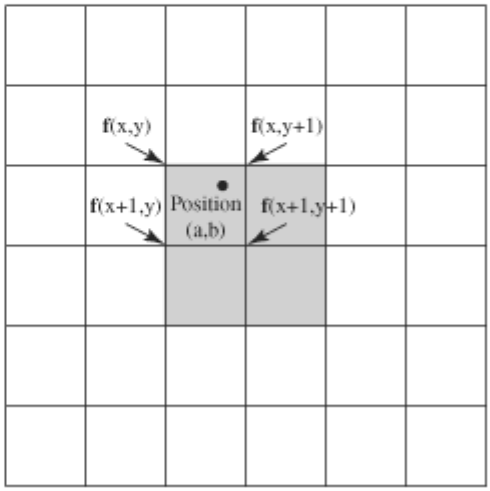
Bilinear interpolation uses an average color value of the four pixels surrounding position (a,b) in the original image. Each neighbor's contribution to the color is based on how close it is to (a,b). Let

x = floor(a)
y = floor(b)

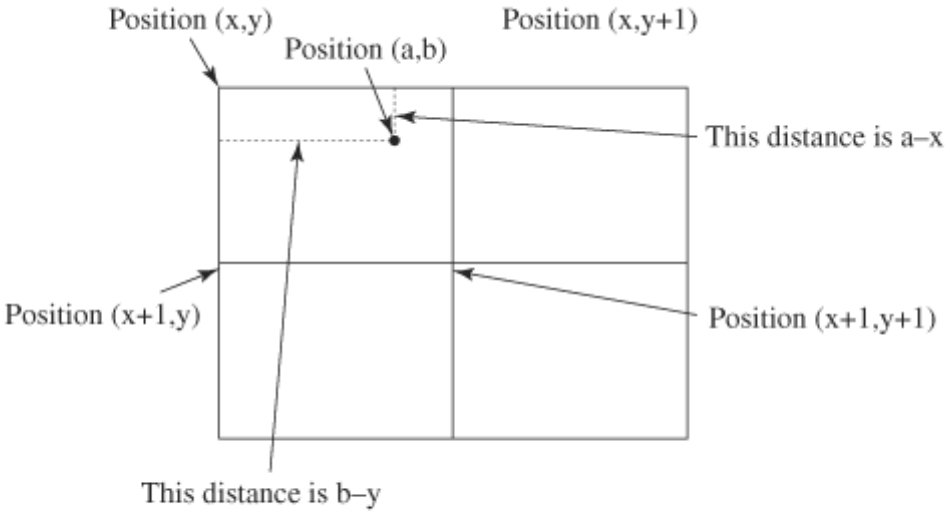
Then the pixels surrounding position (a,b) are

f(x, y)
f(x+1, y)
f(x+1, y+1)
f(x, y+1)

Neighborhood is shown in gray.



Bilinear interpolation



The color of the pixel in image **fs** is a weighted average of the four neighboring pixels. Weights come from each pixel's proximity to (a,b).

To specify the convolution mask for bilinear interpolation, let $t(m, n) = a - (x + m)$ and $u(m, n) = b - (y + n)$ for $0 \le m \le 1$ and $0 \le n \le 1$. Then the mask $h_{bl}(m, n)$ is defined as $h_{bl}(m, n) = (1 - |t|)(1 - |u|)$.