Table of Contents

Introduction 1.1

Game Development

Voor de cursus game development maken we gebruik van het monogame framework.

Installatie

> http://www.monogame.net/downloads/

Als je Visual Studio 2017 hebt:

> http://bit.ly/monogamevsdevsetup

Introductie

MonoGame is een open source implementatie van het Microsoft XNA 4 framework.

Wil je een Android Game maken heb je Android nodig:

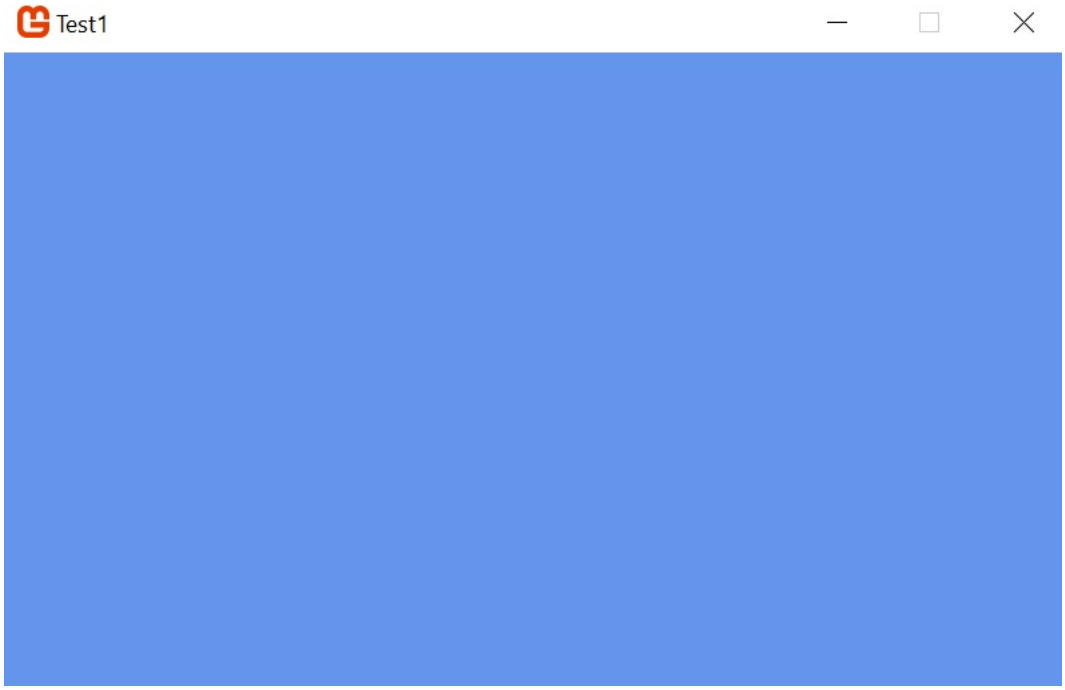
- Xamarin.Android: http://android.xamarin.com/ : (kan gebruikt worden met Visual Studio of Xamarin Studio)

Het doel van MonoGame is om een framework aan te bieden om xbox, Windows games te maken, maar deze ook te kunnen porten naar Max OS X , linux, iOS, Android platformen. Het motto is: "Write Once, Play everywhere".

De technologie die MonoGame cross-platform maakt zijn:

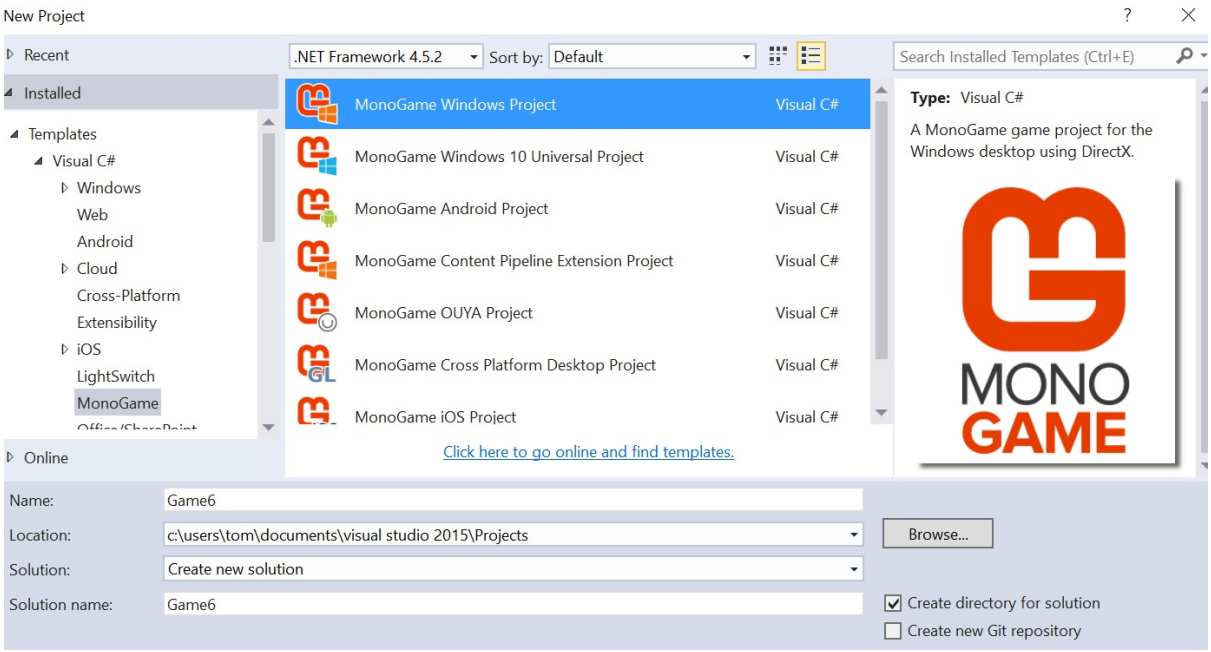
- OpenTK - een low-level C# library die OpenGL, OpenCL en OpenAL voor 3D graphics wrapt.
- SharpDX - een open-source implementation van de volledige DirectX API voor .NET
- Lidgren.Network - een networking library voor .NET framework die gebruik maakt van UDP sockets om clients aan een server te binden.

Project 1



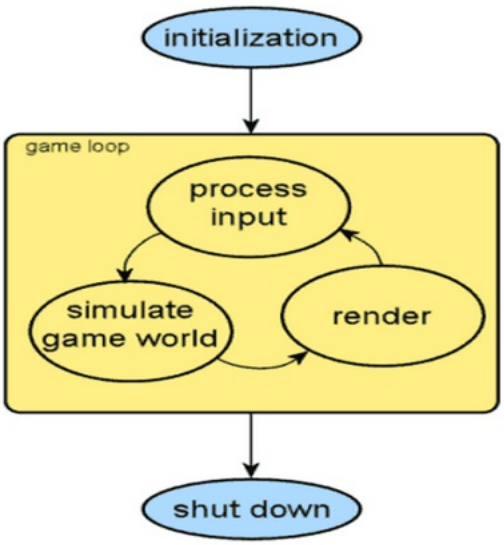
Start Visual studio en selecteer:

> MonoGame Windows Project
>
> TODO: Verander de achtergrond kleur



Project 1 - Benodigdheden

De gameloop is je 'driving force' van je spel:



Als we games maken, moeten we beweging simuleren, en deze simulatie implementeren we door gebruik te maken van volgende methoden:

Initialiseren van je elementen:

```
/// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }
```

Update van je elementen:

```
 /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
```

```
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }
```

Tekenen van je elementen:

```
        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
```

De game loop is een simpel architecturaal patroon om je game code te organiseren.

Het hoofddoel van de gameloop is een cyclus te genereren om de status van je game up-te-daten, te tekenen en een zekere tijd te wachten totdat het tijd is om dit over te doen.

We beschouwen 5 fasen:

1. startup
2. update
3. draw
4. wait
5. cleanup.

Startup fase

In de startup fase gaan we onze game klaar maken, door bijvoorbeeld alle content in te laden, input devices te initiëren,... Deze fase wordt slechts éénmaal gedaan.

Update fase

Dit is het hart van de gameloop, en wordt continu doorlopen. (totdat het spel wordt afgesloten). Normaal wordt de gameloop tussen de 24 en 60 keer per seconde doorlopen.

Het doel is bijvoorbeeld als een object in beweging is zijn coordinaten te updaten zodat bij de volgende loop wat verder bewogen is.

Taken die typisch doorlopen worden in deze fase:

- Polling input devices
- Update van je objecten (update van coordinaten)
- Controle van collisions en eventueel gevolg aan geven
- Uitvoeren van AI voor je vijanden

Draw Stage

De rendering van de beelden op het gescherm, afgehandeld door de grafische kaart.

Wait Stage

De game loop wordt met regelmatig interval terug doorlopen. De regelmatigheid wordt dus bekomen door een wait stage. Anders krijgen snelle computers meer updates dan tragere, en zal je spel zichtbaar sneller draaien. Daarom wachten we totdat de interval tijd afgelopen is en een volgende update beginnen. (Denk nu niet dat je dit wachten zal zien, want alles gaat zo snel.. herinner: tussen 24 en 60 keer per seconde!).

Game Loop Lag

Bijvoorbeeld je game loop wordt 30 keer per seconde doorlopen. Dit wil zeggen dat elke loop 33.3 milliseconden duurt. Maar wat gebeurt er als je code 40 milliseconden nodig heeft? Je zal terug opnieuw in de loop gaan, maar het zou kunnen dat je deze delays zal opmerken. Het managen van je game performantie is niet simpel! Een typische delay is het instantiëren van nieuwe objecten in de loop: tijdsintensief! Vermijd dit!

Cleanup Stage

Bij het afsluiten van de game wordt de gameloop beëindigd, en komen we in de cleanup fase. Hier kunnen we alles wat we niet meer nodig hebben opkuisen. Bijvoorbeeld het disposen van art assets, logging cleanen. Alhoewel je er niet te veel van moet aantrekken, want tussen het .NET framework en XNA wordt dit voor u gedaan.

XNA's Game Loop

Wat gebeurt er wanneer de loop start:

1. Startup Stage

- constructor van je game
- Initialize method
- LoadContent method
- BeginRun method (not overridden in the template)

2. Update Stage

- Update method

3. Draw Stage

- BeginDraw method (not overridden in the template)
- Draw method
- EndDraw method (not overridden in the template)

4. Wait Stage

Alles gebeurt achter de scene. Geen mogelijkheid om methodes te overriden.

5. Cleanup Stage

- OnExiting method (not overridden in the template)
- EndRun method (not overridden in the template)
- Dispose method (not overridden in the template)
- UnloadContent method

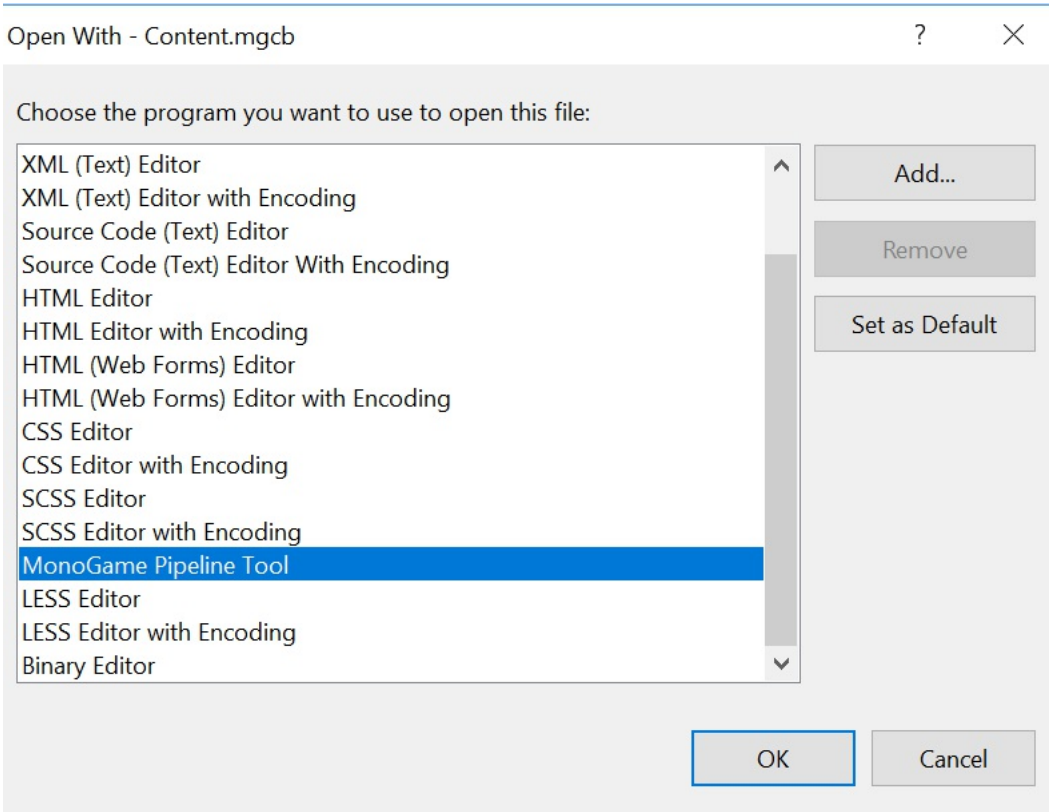
Project 1 - Tekenen

Inladen van content

Content zijn de assets die je gebruikt in je game, zoals afbeeldingen, fonts, 3D modellen, muziek, sound effecten..

Wanneer je naar je MonoGame project in de solution explorer kijkt zie je een Content folder. Hierin vind je het Content.mgcb bestand terug, wat alle content van je game definieert.

Dubbel klikken opent de MonoGame Pipeline applicatie (mee geinstalleerd met MonoGame). Wanneer het niet opent, rechtermuisknop > Open With



The MonoGame Pipeline tool wordt gebruikt om je content te managen. De content bestanden worden geprocessed door de pipeline tool en geconverteerd naar .xnb bestanden om te gebruiken in je MonoGame applicatie.

Wat is een Content Pipeline?

Een content pipeline betekent het proces om een bestand van het ene formaat naar het andere te conveteren. De pipeline tool output het bestand zodat het onmiddellijk kan ingeladen worden door het game project. Deze output bestanden zijn geoptimaliseerd voor "fast loading" en "reduced disk size".

De pipeline tool verwijdert bijvoorbeeld informatie van het bestand dat nuttig is voor de auteur, maar niet nodig tijdens de runtime. Zodat de xnb file het stripped-down versie van het bestand is, wat resulteert in minder disk size.

XNB File Extension

De .xnb file extensie is de extensie voor alle bestanden die geconverteerd zijn door de pipeline tool.

Dus afbeeldingen (.png), audio files (.wav),of anderen zullen allen ge-output worden als .xnb files.

Extra info: https://developer.xamarin.com/guides/cross-platform/game_development/cocossharp/content_pipeline/walkthrough/

Tekenen

In de klasse game1.cs maken we een variabele van het type Texture2D aan:

```
private Texture2D myTexture;
```

Daarna maken we de 'myTexture' aan in de LoadContent methode:

```
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
            myTexture = Content.Load<Texture2D>("boarMovingRight");
        }
```

Je ziet dat er ook een spriteBatch instantie wordt aangemaakt. Dit kunnen we nu gebruiken om te renderen in de Draw methode:

```
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here
            spriteBatch.Begin();
            spriteBatch.Draw(myTexture, new Vector2(0,0),Color.Aqua);
            spriteBatch.End();
            base.Draw(gameTime);
        }
```

We maken gebruik van onderstaande klassen:

- SpriteBatch: wordt gebruikt om 2D graphics op het scherm te tekenen. Sprites zijn 2D visuele elementen, en de spritebatch object kan een enkele sprite of verschillende tussen de begin en einde methode tekenen.
- Texture2D: stelt een image object voor. Texture2D instanties worden gebruikt voor de rendering met de spritebatch instantie.
- Vector2: stelt de positie in een 2D coordinaten stelsel voor.
- Rectangle: een rechthoek met een positie, breedte en hoogte. Dit wordt gebruikt om te bepalen welk deel van de Texture2D te renderen.

Oefeningen

1. 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 scherm rechthoeken tussen 20 en 300 pixels breedte en hoogte in random verschillende kleuren. (deze rechthoeken mogen op het scherm blijven staan).

> Teken van Rectangles: http://blog.dreasgrech.com/2010/08/drawing-lines-and-rectangles-in-xna.html

Animatie

Ons voorbeeld toont 3 afbeeldingen van ons karakter. Deze verschillende afbeeldingen in 1 bestand noemen we een "spritesheet". Dus van deze spritesheet wordt slechts een bepaald deel getoond op het scherm.

```
spriteBatch.Draw(myTexture, new Vector2(0,0),Color.Aqua);
```

We kunnen de draw methode overriden met nog een vierde argument, namelijk een Rectangle argument waarmee we het deel van de afbeelding selecteren dat we willen laten zien. Onze afbeelding is bijvoorbeeld 252 pixels breed, en 60 pixels hoog, en bestaat uit 3 deelafbeeldingen: Elk deelafbeelding stelt een beweging voor. Als we deze 3 deelafbeeldingen snel genoeg achter elkaar laten zien (zeker minstens 15 keer per seconden), simuleren we echte beweging. Dus het deel van de afbeelding dat we willen laten zien is 252 / 3 = 84 pixels breed. Het volgende frame moeten we dus onze rechthoek opnieuw 84 pixels opschuiven om de volgende deelafbeelding te tonen.

In code betekent dit dus:

```
Rectangle deelAfbeelding = new Rectangle(0,0,84,60);
spriteBatch.Draw(myTexture, new Vector2(0,0),deelAfbeelding,Color.Aqua);
```

Het volgend frame gaan we onze rechthoek 84 pixels opschuiven:

```
int schuifOp = 0;
    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        // TODO: Add your drawing code here
        spriteBatch.Begin();
        Rectangle deelAfbeelding = new Rectangle(schuifOp,0,84,60);
        spriteBatch.Draw(myTexture, new Vector2(0,0),deelAfbeelding,Color.Aqua);
        spriteBatch.End();

        schuifOp += 84;
        if (schuifOp > 168)
            schuifOp = 0;
        base.Draw(gameTime);
    }
```

> Nota: de Rectangle's nul positie is links boven

Oefening

1. Scene veranderingen zijn factoren waarmee een developer tijdens de implementatie rekening mee moet houden. Tijdens deze oefening zullen we een viewport ontwikkelen:
2. Programmeer een "cut"-scene: afbeelding 1 verschijnt op het scherm, en enkele momenten later verschijnt afbeelding 2.
3. Programmeer een "wipe"-scene: afbeelding 1 verschijnt op het scherm, later komt afbeelding 2, maar komt van links naar rechts op het scherm, terwijl afbeelding 1 geleidelijk aan verdwijnt: afbeelding 2 duwt afbeelding 1 van het scherm.
4. Teken een rechthoek op het scherm, en laat deze roteren.

Keyboard Events

Bijna alle games maken gebruik van keyboard events, om bijvoorbeeld een personage rond te bewegen. Als je Keyboard input wil gebruiken importeer je eerst de juiste library van Xna:

```
using Microsoft.Xna.Framework.Input;
```

Daarna ga je in je update methode checken voor inputs, zoals bijvoorbeeld onderstaande code:

```
public void Update()
        {
            //Check Keyboard state

            KeyboardState stateKey = Keyboard.GetState();

            if (stateKey.IsKeyDown(Keys.Left))
            {
                left = true;
            }

            if (stateKey.IsKeyDown(Keys.Right))
            {
                right = true;
            }

            if (stateKey.IsKeyUp(Keys.Left))
            {
                left = false;
            }

            if (stateKey.IsKeyUp(Keys.Right))
            {
                right = false;
            }

            if (stateKey.IsKeyDown(Keys.Space))
            {
                ChangeState(jumping);
                jump = true;
                startJump = true;
            }

            state.Update();
        }
```

De texture kan gestuurd worden door de pijltjes toetsen. Wanneer de gebruiker een toets indrukt, wordt er een vlag gezet om de status te bewaren (true = pressed, false= not pressed of released).

Verder wordt in de UPDATE methode gecontroleerd welke toets men ingedrukt, en wordt de positie van de rechthoek verzet: de X en Y offset voor de achtergrond afbeelding wordt maw bepaald.

Animatie

Onze afbeelding is een spritesheet: onze geanimeerde afbeelding bestaat uit deelafbeeldingen die in 1 bestand zitten. Dus als we op regelmatige tijdstippen een deel van de afbeelding laten zien en deze afwisselen, simuleren we animatie.

De afbeelding is 252px breed, en 60 pixels hoog. De afbeelding bestaat uit 3 deelafbeeldingen, dus elke afbeelding is 252/3 px breed.

De update methode ziet er als volgt uit:

```
    private Rectangle _showRectangle;
    public Hero(Texture2D texture)
    {
       _texture = texture;
       _showRectangle = new Rectangle(0,0, 84,60);
    }
    _showRectangle.X += 84;
    if (_showRectangle.X > 252)
                _showRectangle.X = 0;

    spriteBatch.Draw(_texture, new Vector2(0, 0),_showRectangle, Color.Aqua);
```

Het probleem dat we tegenkomen is dat de animatie veel te snel gaat. Dus dit moeten we proberen onder controle te houde (zie later).

Frame-independent movement

- Stel ik wil 100 px in 100 frames met een snelheid van 60 FPS bewegen, dan doe ik daar 100frames / 60FPS = 1,66 seconde over.
- stel, ik heb maar 30 FPS beschikbaar op mijn device, dus ga il elke seconde 30 pixels verder, en dus om op positie (100,0) te geraken heb ik dus 100frames/30 FPS = 3,33 seconde voor nodig.

> Conclusie: op device 1 ga ik 2x zo snel.

Dit gedrag heeft invloed op de gebruikservaring. De oplossing om deze ervaring tegen te gaan is frame independent movement. In plaats van onze figuur (vb. Mario) elk frame een zeker aantal pixels te laten bewegen, gaan we de bewegingssnelheid specifiëren.

Bijvoorbeeld we willen 50 px per seconde bewegen. Nu moeten we ook weten hoeveel tijd verstreken is sinds de laatste beweging. Dit wordt al eens de deltatijd genoemd.

> x += 50 * deltatijd if(x>100) x = 0

Als ons spel op 60 FPS draait zal de delta altijd 1/60 of 16,6 milliseconde zijn. Dus in elk frame gaan we 50_0,016 seconde = 0,83 pixels vooruit. Dus po 60FPS = 60_0,83 = 50 pixels!

Als je device dan 30 fps draait, krijg je 50*1/30 = 1,66 pixels vooruit. en dus 1,66* 30 = 50 pixels.

In mijn praktisch voorbeeld heb ik een spritesheet van 252 pixels, met daarin 3 animaties. Dus elke animatieafbeelding is 84 pixels breed. Om dus 84 pixels per frame op te schuiven:

```
double x = 0;
       int offset = 0;

       public void Update(GameTime gameTime)
       {


           double temp = 84 * ((double)gameTime.ElapsedGameTime.Milliseconds / 1000);

           x += temp;
           //Als x groter of gelijk aan 84 is laat ik het volgende //deel van de spritesheet zien
           if (x >= 84)
           {
               Console.WriteLine(x);
               x = 0;
               offset += 84;
           }
           if (offset >= 252)
               offset = 0;


       }
```

Om 3x sneller te zijn:

```
double x = 0;
       int offset = 0;

       public void Update(GameTime gameTime)
       {


           double temp = 84 * ((double)gameTime.ElapsedGameTime.Milliseconds / 1000);

           x += temp;
           //Als x groter of gelijk aan 84 is laat ik het volgende //deel van de spritesheet zien
           if (x >= 84/3)  //=> 3x sneller
           {
               Console.WriteLine(x);
               x = 0;
               offset += 84;
           }
           if (offset >= 252)
               offset = 0;
       }
```

We gaan bovenstaande code veralgemen, zodat we frame independent movement in elke spriteklasse kunnen gebruiken

Generaliseren van frame independent movement

Ik maak een nieuwe klasse AnimationFrame aan:

```
public class AnimationFrame
    {
            public Rectangle SourceRectangle { get; set; }

    }
```

Ik maak een klasse Animation aan (deze klasse bevat 1 of meerdere animationframes):

```


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Xna.Framework;

namespace AnimationClass
{
    public class Animation
    {
        private List<AnimationFrame> frames;
        public AnimationFrame CurrentFrame { get; set; }
        public int AantalBewegingenPerSeconde { get; set; }

        private int _totalWidth=0;

        public Animation()
        {
            frames = new List<AnimationFrame>();
```

```
            AantalBewegingenPerSeconde = 1;
        }
        public void AddFrame(Rectangle rectangle)
        {
            AnimationFrame newFrame = new AnimationFrame()
            {
                SourceRectangle = rectangle,
                //Duration = duration
            };

            frames.Add(newFrame);
            CurrentFrame = frames[0];
            offset = CurrentFrame.SourceRectangle.Width;
            foreach (AnimationFrame f in frames)
                _totalWidth += f.SourceRectangle.Width;
        }

        private int counter = 0;
        private double x=0;
        public double offset { get; set; }
        public void Update(GameTime gameTime)
        {
            double temp = CurrentFrame.SourceRectangle.Width * ((double)gameTime.ElapsedGameTime.Milliseconds / 1000);

            x += temp;
            if (x >= CurrentFrame.SourceRectangle.Width / AantalBewegingenPerSeconde)
            {
                Console.WriteLine(x);
                x = 0;
                counter++;
                if (counter >= frames.Count)
                    counter = 0;
                CurrentFrame = frames[counter];
                offset += CurrentFrame.SourceRectangle.Width;
            }
            if (offset >= _totalWidth)
                offset = 0;


        }

    }
}
```

Onze Hero klasse heeft nu (compositie) een animation

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace AnimationClass
{
    public class Hero
    {
        private Texture2D _texture;

        private Rectangle _showRectangle;

        private Animation _animation;

        public Hero(Texture2D texture)
        {
            _animation = new Animation();
            _animation.AddFrame(new Rectangle(0,0,84,60));
            _animation.AddFrame(new Rectangle(84, 0, 84, 60));
            _animation.AddFrame(new Rectangle(168, 0, 84, 60));
            _animation.AantalBewegingenPerSeconde = 5;
            _texture = texture;
            _showRectangle = new Rectangle(0,0, 84,60);


        }

        double x = 0;
        int offset = 0;

        public void Update(GameTime gameTime)
        {
            _animation.Update(gameTime);
        }


        public TimeSpan Duration { get; set; }
        public void Draw(SpriteBatch spriteBatch)
        {

            _showRectangle = _animation.CurrentFrame.SourceRectangle;
            spriteBatch.Draw(_texture, new Vector2(0, 0),_showRectangle, Color.Aqua);


        }
    }
```

```
}
```

Oefeningen Reeks 2

1. Teken een texture op het scherm en laat deze elke tick 1 pixel in X en Y as opschuiven. Controleer wanneer de cirkel tegen de randen van het scherm botst. Na collision beweegt de texture naar de andere kant.
2. Zoek een sprite en laadt hem als texture in. Toon de geanimeerde sprite op het scherm, maar laat je animatie enkel werken wanneer je pijltje links of rechts indrukt.
3. Zoek een sprite voor een linkse en rechtse animatie, en toon op de juiste moment de animatie.
4. Experimenteer zelf met frame independent movement

Een beetje wiskunde: Vectoren

Getallen kennen we allemaal. Maar met getallen alleen kunnen we geen bewegingen beschrijven, want ze kunnen geen richting weergeven. In de klassieke mechanica worden krachten en bewegingen weergegeven met vectoren. In de laatste decenia kwam er voor vectorrekenen steeds meer belangstelling door de robotica, visualisering...

Wat zijn vectoren

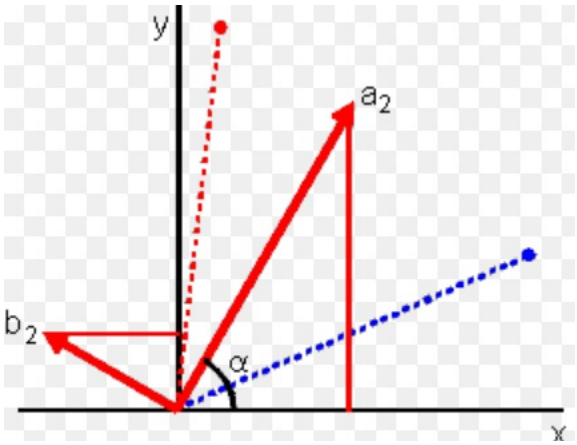
Sommige grootheden zijn volledig bepaald als de waarde ervan gegeven is. We noemen dit scalaire grootheden. Een scalair of een getal is een grootheid die beschreven wordt door een grootte eventueel met een eenheid. Een scalair heeft dus een waarde, maar geen richting. Een aantal voorbeelden van scalaire grootheden zijn de temperatuur, vb. 20°C, of de lengte van een persoon, vb. 180 cm. Het begrip vector komen we vaak tegen bij de beschrijving van grootheden als kracht, snelheid, versnelling,... Deze grootheden hebben behalve een grootte ook een richting en een zin. Vectoren kunnen we dus voorstellen als pijlen in de ruimte. We kunnen een vector voorstellen met behulp van een pijl waarvan de lengte in verhouding staat tot de grootte.

Coördinaatvoorstelling

We voorzien de ruimte met een coördinatensysteem, waarmee punten en dus ook vectoren coördinaten krijgen.

In onderstaande figuur is een vlak met oorsprong 0 en 2 punten a en b getekend. Het punt a heeft hierin coördinaten (Xa,Ya) en punt b (Xb,Yb).

Deze coördinaten kennen we ook toe aan de bijbehorende vectoren en het is daarbij gebruikelijk om ze onder elkaar in een kolom te zetten: (Xa) (Ya)



Een vector a is een georienteerd lijnstuk, dat éénduidig bepaald wordt door zijn lengte (ook norm of grootte genoemd), richting en zin. De getallen van de vector noemen we de coördinaten van de vector. De lengte wordt genoteerd als ||a||.

De lengte van de vector kunnen we gemakkelijk uitrekenen met behulp van de stelling van Pythagoras: ||a|| = sqrt(Ax^2 + Ay^2)

Met de monogame library kom je de lengte van een vector te weten door:

```
var lengteHero = Vector2.Distance(new Vector2(0,0),hero.Position);
```

Optellen van vectoren

De som van 2 vectoren is betekenisvol. Grafisch kunnen we bepalen hoe de som van 2 vectoren verloopt:

> https://nl.wikipedia.org/wiki/Vector_(wiskunde)#Optellen_van_vectoren#Optellen_van_vectoren)

De som van vectoren is nuttig in ons gamedesign om de beweging van onze objecten te simuleren.

Bijvoorbeeld ons object staat op positie (10,10), en de snelheid waarmee ons object beweegt is (1,0). Dus we bewegen enkel in de X-Richting.

Om onze volgende positie te weten te komen tellen we deze 2 vectoren op. In monogame mogelijk door vb.

```
Vector2 up = new Vector2(1,0);
```

```
Vector2 Position = new vector2(10,10)

Position = Vector2.Add(Position, up);
```

of ook mogelijk door:

```
Position += up;
```

Vectorcomponenten van een 2D vector

In de tweedimensionale ruimte kunnen we van een vector altijd de x en de y coordinaat bepalen als we de lengte ||a||, en de richting en de zin (alfa, met name de hoek tussen de richtingn en de zin en de positieve as) van de vector kennen. Uit de rechthoekige driehoek volgt dat:

```
cos alfa = a1/||a|| => a1 = ||a||*cos alfa
sin alfa = a2 / ||a|| => a2 = ||a||*sin alfa
```

we kunnen dus een 2D vector schrijven als: a = (||a|| cos alfa) (||a|| sin alfa)

Het inwendig product (dot product)

Het inwendig product, kortweg inproduct, is ook gekend als het dotproduct, omdat we deze vermenigvulding noteren met een punt (a dot). Het is het product van 2 vectoren dat als resultaat een getal oplevert.

> Definitie: het product van 2 vectoren, dat als resultaat een getal oplevert, noemen we het inwendig product, of kortweg inproduct.

a.b = a1.b1 + a2.b2+ ... + an.bn

van a = (2,3) en b=(6,-1) dan is a.b = 2.6 + 3*-1 = 9

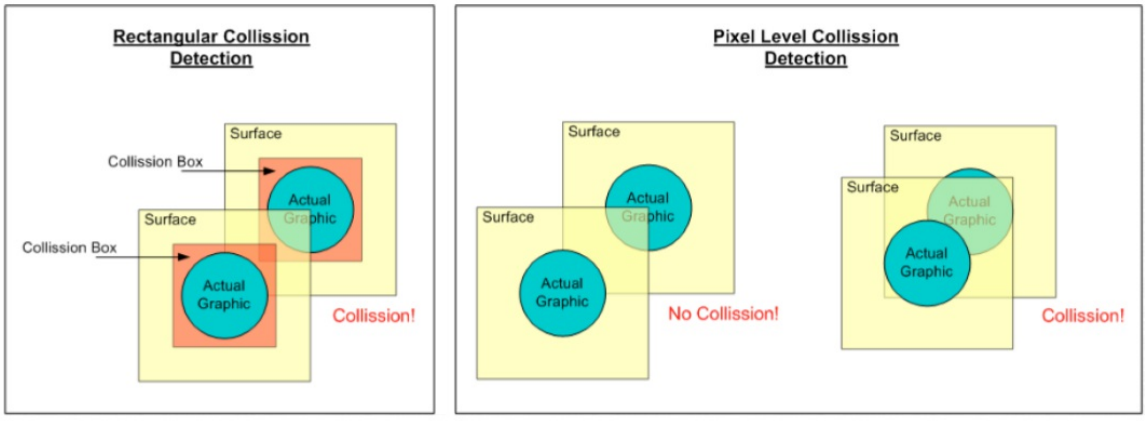
Bij het programmeren van animaties moeten we soms de hoek tussen de kijklijn naar 2 objecten kunnen berekenen, bijvoorbeeld de openingshoek van een cameradiafragma om te weten of een persoon in beeld is. Bij vectoren speelt de hoek tussen 2 vectoren vaak een belangrijke rol. De methode om deze grootheid uit te drukken in de coordinaten van de vectoren is het inwendig product.

> https://nl.wikipedia.org/wiki/Inwendig_product

Collision Detection

In de meeste games is collision detection een essentieel onderdeel. Collision detection wil zeggen dat er gecontroleerd wordt of 2 objecten elkaar raken. Er zijn 2 methode om dergelijks te checken:

- Rectangle Collision Detection
- Per Pixel Collision Detection



Rectangle Collision Detection