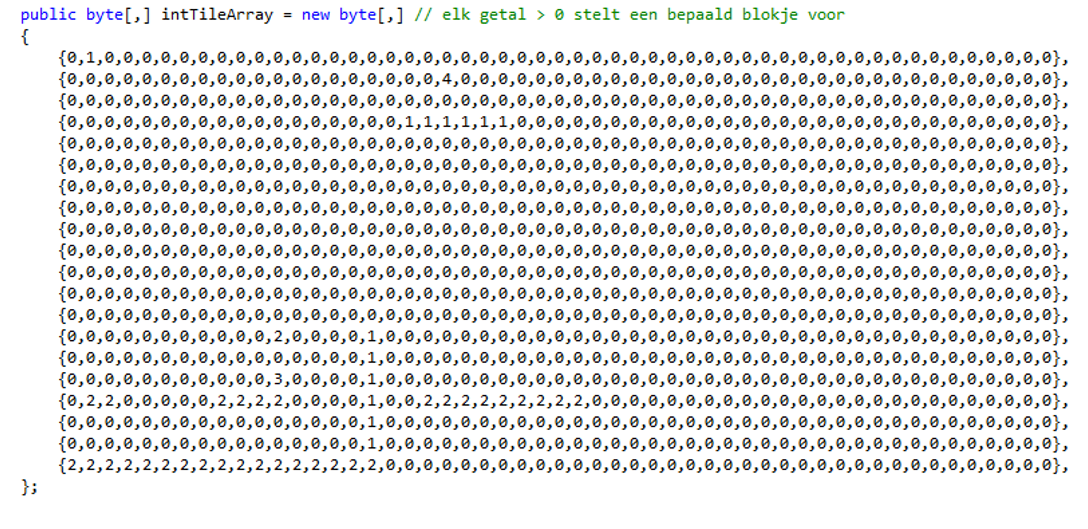
# Wat vooraf ging…

In de cursus [GameDevelopment](https://bamaflexweb.ap.be/BMFUIDetailxOLOD.aspx?a=97576&b=5&c=1) wordt volgende stuk beschreven ivm hoe je best je level opbouw bewaard mbv een 2D array van getallen. Hierbij stelt ieder getal een specifiek element voor (bijvoorbeeld 1= vernietigbaar blok, 2=vast blok, 3= monster Type 1, 4=monster Type 2, etc):

Onderstaande array bepaalt de plaats van een object: de array is 50x20 dimensies groot, en dit alles doe je maal 30, omdat elk object een veelvoud van 30 pixels is. Dus met andere woorden heb ik een level opgebouwd dat 50x30 = 1500 pixels breed en 20x30=600 pixels hoog. Onderstaande array fungeert als plattegrond van je level.

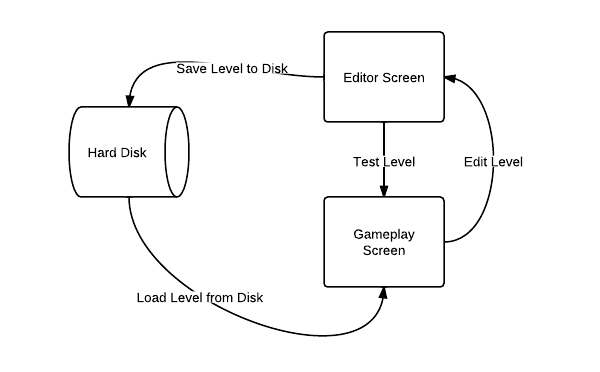


Via de volgende workshop zullen we ervoor zorgen dat de 2D-dimensionele array, die ons spel voorstelt, niet meer in code moet geschreven worden maar als aparte file kan ingeladen worden.

# Het begin van een mapeditor

## Concept

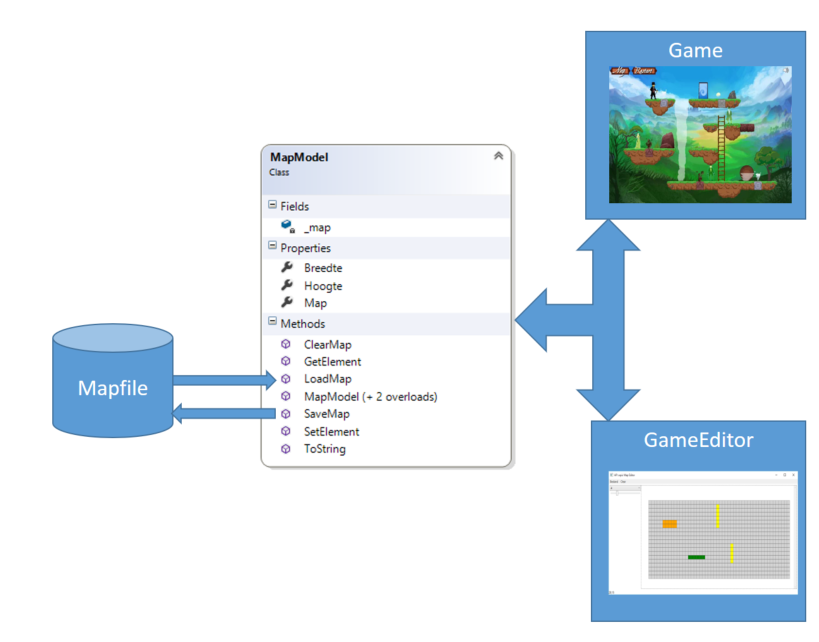
De ideale mapeditor voor een 2D-spel is eentje die ons toelaat om snel mappen te maken, zonder dat we daarvoor C# van het spel zelf moeten maken (denk bijvoorbeeld aan [Tiled](http://www.mapeditor.org/), één van de beste opensource editor). De mappen worden met andere woorden niet in de code zelf aangemaakt, maar op externe bestanden.



## Doel

We zullen een mapeditor maken in WPF. We gaan hierbij zo object georiënteerd mogelijk werken en zullen de “MapModel”klasse ontkoppelen van het grafisch gedeelte.

Door dit te bewerkstelligen kunnen we de MapModel klasse hergebruiken in het spel zelf, zonder dat hierbij nog code moet aangepast worden. MapModel is als het ware een bibliotheek.



## Deel 1: MapModel

De MapModel klasse is het hart van ons project. Het is een klasse die een interne voorstelling van de kaart zal hebben en die enerzijds zal toelaten dat kaarten kunnen bewaard en ingeladen worden, en anderzijds opdat de GameEngine en/of MapEditor de kaart kan gebruiken en aanpassen.

### Basis MapModel

We maken een nieuwe klasse MapModel die een kaart voorstelt in zijn rudimentairste vorm: een 2D- arrys van bytes (bytes zodat we minder plek nodig hebben per element. Nadeel is wel dat iedere element maar 255 mogelijke waarden kan hebben):

private byte[,] \_map;

We willen dat de buitenwereld deze kaart enkel kan uitlezen, en maken dus een read-only property:

public byte[,] Map

{

get

{

if (\_map != null)

return \_map;

throw new NullReferenceException("Map not created");

}

}

[We gooien de NullReferenceException op voor het geval de gebruiker (programmeur) van deze klasse is vergeten een kaart ook effectief aan te maken (zie verder)]

Om de buitenwereld nog te helpen voegen we nog enkele handige properties toe die de Hoogte en Breedte van een map teruggeven:

public int Hoogte

{

get { return \_map.GetLength(0); }

}

public int Breedte

{

get { return \_map.GetLength(1); }

}

Voorts maken we 2 methoden opdat de buitenwereld individuele elementen op de kaart kan uitlezen en aanpassen:

public void SetElement(int x, int y, int value)

{

\_map[y, x] = (byte)value; //Todo check if valid x, y value

}

public int GetElement(int x, int y)

{

return Convert.ToInt32(\_map[y, x]); //Todo check if valid x, y value

}

Alsook een methode om de volledige kaart leeg te maken:

public void ClearMap()

{

for (int i = 0; i < Hoogte; i++)

{

for (int j = 0; j < Breedte; j++)

{

\_map[i, j] = 0;

}

}

}

### Constructors

We voorzien 3 constructors, opdat de gebruiker op verschillende manieren een nieuwe kaart kan aanmaken:

1. Door een breedte en hoogte in te geven
2. Door een bestaande 2D bytearray in te geven
3. Door de locatie van een mapfile in te geven die dan zal ingeladen worden

public MapModel(int breedte, int hoogte)

{

\_map = new byte[hoogte, breedte];

}

public MapModel(byte[,] map)

{

\_map = map;

}

public MapModel(string path)

{

this.LoadMap(path); //Komt later

}

[De LoadMap()-methode ontbreekt nog maar dat lossen we zo meteen in het volgende hoofdstuk op.]

## Deel 2: MapModel IO

Alles staat of valt uiteraard met het correct in en uitladen van een tekstbestand waarin we de kaart zullen beschrijven in een voor normale stervelingen leesbare tekst.

### Opbouw file

We kiezen als bestandinhoud voor volgende layout:

* Breedte map = lengte van iedere lijn
* Hoogte map = aantal lijnen
* Data per lijn, gescheiden met een ko[[1]](#footnote-1)mma

Een 5 bij 3 map waarbij er in het midden een blokje (1) staat zouden we als volgt naar een bestand wegschrijven:

*5*

*3*

*0,0,0,0,0*

*0,0,1,0,0*

*0,0,0,0,0*

### Load en Save

We gaan 2 methoden toevoegen aan ons MapModel:

* LoadMap: inladen van map uit bestand
* SaveMap: bewaren van map naar bestand

We gaan het MapModel niet zelf de locatie van de file laten bewaren, dit omdat ik daar geen zin in heb ☺ Beide methoden moeten dus aangeroepen van buitenuit met als extra argument een string die het path bevat van de te gebruiken file:

public void LoadMap(string path){…}

public void SaveMap(string path){…}

Wanneer we data naar een bestand wegschrijven is het belangrijk dat je steeds de Laad en Bewaar methoden “synchroon” houdt, dit maakt het programmeren ervan een pak eenvoudiger.

Beide methoden moeten namelijk de data in het bestand op dezeldfe manier gebruiken. Als bijvoorbeeld de LoadMap methode de eerste lijn in het bestand als Breedte gebruikt, maar de SaveMap bewaard de Hoogte in de eerste lijn…dan is er een synchronisatieprobleem.

### SaveMap

Het wegschrijven van de map naar een file zal uit 4 delen bestaan:

1. Steamwriter naar file openen
2. Breedte en hoogte wegschrijven
3. 2D array wegschrijven
4. Streamwriter sluiten

Laten we dit doen:

**Streamwriter naar file openen:**

StreamWriter writer = new StreamWriter(path);

**Breedte en hoogte wegschrijven:**

writer.WriteLine(Breedte);

writer.WriteLine(Hoogte);

**2DArray wegschrijven:**

for (int i = 0; i < Hoogte; i++)

{

for (int j = 0; j < Breedte; j++)

{

writer.Write(\_map[i,j]);

if (j < Breedte - 1) //Geen komma op einde van een lijn

writer.Write(",");

}

writer.WriteLine();

}

**Streamwriter sluiten**

writer.Close();

### LoadMap

De LoadMap-methode zal dezelfde 4 delen bevatten , daar deze ‘synchroon’ moet werken zoals de SaveMap methode:

1. Steam**reader** naar file openen
2. Breedte en hoogte **uitlezen**
3. 2D array **uitlezen**
4. Stream**reader** sluiten

Uiteraard zullen we bij het uitlezen van data gebruik moeten maken van de Convert-bibliotheek zodat we de data naar de juiste vorm kunnen uitlezen[[2]](#footnote-2).

**Steamreader naar file openen:**

StreamReader reader = new StreamReader(path);

**Breedte en hoogte uitlezen:**

int breedte = Convert.ToInt32(reader.ReadLine());

int hoogte = Convert.ToInt32(reader.ReadLine());

**2D array uitlezen:**

for (int i = 0; i < hoogte; i++)

{

//lees lijn per lijn

var lijn = reader.ReadLine();

//Splits op komma’s

var gesplitst = lijn.Split(',');

for (int j = 0; j < gesplitst.Length; j++)

{

resultaat[i, j] = (byte)Convert.ToInt32(gesplitst[j]);

}

}

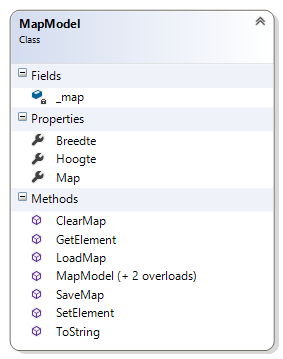
\_map = resultaat;

**Steamreader sluiten**

reader.Close();

### Conclusie

De MapModel klasse is reeds klaar en kan integraal gebruikt worden in andere projecten:



We hebben er bewust voor gekozen om zo weinig mogelijk publieke methoden en properties aan te bieden, wat de bruikbaarheid van onze klasse, hopelijk verhoogd.

Je zou deze klasse nu reeds kunnen testen in een Console applicatie. Een voorbeeld gebruik:

//Maken nieuwe kaart, 5 breed, 3 hoog

MapModel mijnMap = new MapModel(5, 3);

//Element met coordinaten 2,2 op 6 zetten

mijnMap.SetElement(2, 2, 6);

//Kaart wegschrijven

mijnMap.SaveMap("testje.map");

//Kaart opnieuw inladen

MapModel mijnandereMap = new MapModel("testje.map");

//Waarde van element 2,2 uitlezen

int waarde = mijnandereMap.GetElement(2, 2);

//Waarde op scherm zetten, hopelijk komt er 6

Console.WriteLine(waarde.ToString());

Tekst in testje.map:

5

3

0,0,0,0,0

0,0,0,0,0

0,0,6,0,0

## Deel 3: WPF Mapeditor

De WPF MAPEditor UI bestaat uit 3 delen:

1. Een **Menu** bovenaan die we gebruiken om kaarten in te laden, bewaren, leegmaken, etc
2. Een soort **Toolbox** links waar we alle zaken zetten om de kaart mee te manipuleren.
3. De **Canvas** rechts waar de eigenlijke map zal getoond worden

### Menu [XAML]

We definiëren in het hoofdgrid 2 rijen. De bovenste rij zal het Menu bevatten en zetten we dus op Auto. Alle overige ruimte is voor de Toolbox en het Canvas:

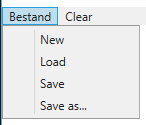
<Grid.RowDefinitions>

<RowDefinition Height="auto"></RowDefinition>

<RowDefinition></RowDefinition>

</Grid.RowDefinitions>

Het menu bestaat uit een handvol knoppen met bijhorende acties:



<Menu Grid.Row="0">

<MenuItem Header="Bestand">

<MenuItem Header="New" Name="menuNew" Click="menuNew\_Click"></MenuItem>

<MenuItem Header="Load" Name="menuLoad" Click="menuLoad\_Click"></MenuItem>

<MenuItem Header="Save" Name="menuSave" Click="menuSave\_Click"></MenuItem>

<MenuItem Header="Save as..." Name="menuSaveAs" Click="menuSaveAs\_Click"></MenuItem>

</MenuItem>

<MenuItem Header="Clear" Name="menuClear" Click="menuClear\_Click"></MenuItem>

</Menu>

### Menu acties voor IO [C#]

We willen overal in de applicatie aan de kaart (currentMap) en z’n filelocatie (currentMapPath) geraken en defenieren daarom in de MainWindow klasse volgende 2 private fields:

private MapModel currentMap;

private string currentMapPath="";

We kunnen nu achter de verschillende menuitems de nodige FileDialogs plaatsen:

**New**

Klikken op New behandelen we verderop.

**Load:**

We vragen aan de gebruiker welke file moet geopend worden en slaan de locatie vervolgens op en laden de kaart in.

OpenFileDialog dialog = new OpenFileDialog();

if (dialog.ShowDialog() == true)

{

currentMapPath = dialog.FileName;

currentMap = new MapModel(currentMapPath);

LoadMapOnView();

}

[De LoadMapOnView method zal de kaart op het canvas tekenen, wat verderop zal uitgelegd worden]

**Save:**

We kunnen enkel Save gebruiken indien er reeds een file effectief gekozen is[[3]](#footnote-3):

if (currentMapPath == "" )

{

SaveFileDialog dialog = new SaveFileDialog();

if (dialog.ShowDialog() == true)

{

currentMapPath = dialog.FileName;

currentMap.SaveMap(currentMapPath);

}

}

**Save As:**

Deze is quasi identiek aan Save, alleen moeten we niet controleren of de file reeds bestaat daar de gebruiker dit sowieso zal doen[[4]](#footnote-4):

SaveFileDialog dialog = new SaveFileDialog();

if (dialog.ShowDialog() == true)

{

currentMapPath = dialog.FileName;

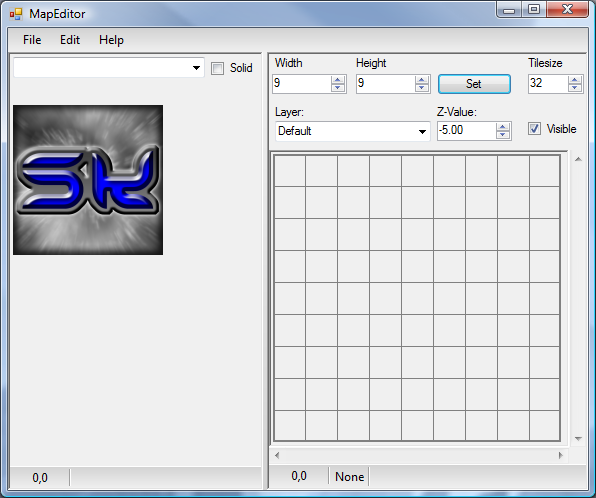
currentMap.SaveMap(currentMapPath);

}

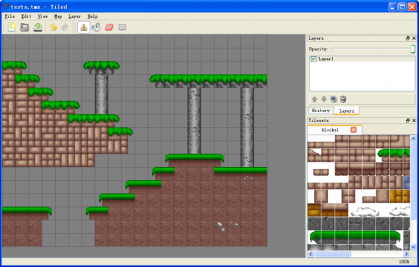
## Bestaande voorbeelden



<http://www.codeproject.com/Articles/26044/Tile-Editor-Control>



<http://www.codeproject.com/Articles/26166/2D-Map-Editor>



<http://www.mapeditor.org/>

1. We kiezen er bewust voor om ieder element met een komma te scheiden. Dit laat ons toe om getallen groter dan 9 toe te laten als waarde (daar we met bytes werken kunnen we dus eender welk getal tussen 0 en 255 gebruiken). [↑](#footnote-ref-1)
2. We laten het aan de lezer over om de nodige foutcontrole in te bouwen bij het uitlezen van foutieve informatie en deze vervolgens alsnog trachten te converteren. [↑](#footnote-ref-2)
3. Tip: je kan de UX verbeteren door deze knop pas actief te zetten wanneer hij mag gebruikt worden. [↑](#footnote-ref-3)
4. We zouden dus eigenlijk deze knop ook gewoon de Save –code kunnen laten uitvoeren, wat weer wat code uitspaart. [↑](#footnote-ref-4)