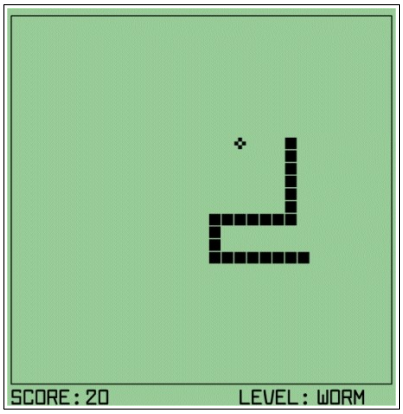
Snake – ein Klassiker

Jeder, der schon mal Computerspiele genutzt hat, hat mit hoher Wahrscheinlichkeit von diesem Spiel gehört oder es auch schon gesehen. Snake[1] ist -so wie Tetris oder PacMan- ein Klassiker unter den Computerspielen.



Das Bild[2] zeigt das Spielprinzip: Innerhalb einer Umrandung befindet sich eine anfangs kurze Schlange, die sich automatisch bewegt. Die Bewegungsrichtung kann zumeist mit Pfeiltasten geändert werden, nur nicht entgegen der gerade gewählten Bewegungsrichtung.

In (un-)regelmäßigen Zeitabständen tauchen Futterhappen auf dem Spielfeld auf, die von der Schlange gefressen werden können. Dafür erhält man Spielpunkte.

Mit jedem gefressenen Futterhappen wird die Schlange länger. Läuft die Schlange jedoch gegen den Spielfeldrand oder gegen ihren eigenen Körper, stirbt die Schlange und das Spiel endet.

Diese Spielidee lässt sich gut in einem Konsolenprogramm umsetzen, da hierfür keine aufwändige Grafik zwingend erforderlich ist.

Im Folgenden soll nun dieses Spiel in der Programmiersprache C# als Konsolenprojekt umgesetzt werden. Hierzu möchte ich die Programmentwicklung schrittweise vornehmen und das Spiel soll schrittweise modularisiert entstehen. Ich werde die notwendigen Überlegungen entsprechend ausformulieren und Hinweise zur Implementation geben. Die eigentliche Implementation liegt aber beim Lernenden. Allerdings werde ich meine Beispiel-Implementation vorstellen!

Für die Implementation sollte man folgende Befehlsstrukturen kennen:

- Entscheidungen mittels **`if`** bzw. **`if...else`**
- Fallauswahl mitels **`switch`**
- Wiederholungen mittels **`for`**, **`while`** und **`do...while`**
- Methoden zur Modularisierung
- Arrays in ein- bzw. zweidimensionaler Form

Die eine oder andere Neuerung könnte hinzukommen. Aber das ganze Projekt verbleibt in der strukturierten und prozeduralen Programmierung. Objektorientierung spielt hier noch keine Rolle! Obwohl man manches dann vielleicht eleganter lösen könnte...

```
# # # # # # # # # # # # # # # # # # # #
#                                       #
#                                       #
#       o                           o   #
#                                       #
#                           o           #
#                                       #
#                   < s s s s           #
#                                       #
#       o                               #
#                                       #
#                                       #
#             o                         #
#                                       #
# # # # # # # # # # # # # # # # # # # #
```

Vorüberlegungen

Bevor es losgehen soll, sind einige Vorüberlegungen nötig. Das Spiel soll ja als Konsolenprojekt gestaltet sein. Jedes Konsolenfenster besteht prinzipiell aus einer gewissen Anzahl von Zeilen (rows) und Spalten (columns) wie im Bild dargestellt.

Jede Zelle wird also durch eine Zeilen- und Spaltennummer adressiert. Typischerweise beginnt man beim Zählen mit der Zahl 0. Die Matrix im Bild hat 15 Zeilen und 20 Spalten.

1 Siehe auch **`https://de.wikipedia.org/wiki/Snake_(Computerspiel)`** abgerufen am 13.05.2020

2 Quelle: **`https://www.chip.de/downloads/Snake_18757550.html`** abgerufen am 13.05.2020

© Uwe Homm Version vom 28. Mai 2020

Der Zeilenindex, der eine bestimmte Zeile bestimmt, kann also die Werte von 0 bis 14 annehmen. Das Gleiche gilt für den Spaltenindex. Dieser kann also Werte von 0 bis 19 annehmen.

Eine bestimmte Zelle in dieser Matrix wird also durch das Wertepaar Zeile/Spalte adressiert. So steht beispielsweise das Zeichen „**<**“ in der Zelle mit dem Zeilenindex 7 und dem Spaltenindex 9.

Jede Zelle kann **genau ein Zeichen** aufnehmen. Eine „leere“ Zelle enthält das Leerzeichen.

Der Rand des Spielfeldes wird durch ein vom Programmierer gewähltes Zeichen gebildet. In diesem Projekt soll hierfür das Zeichen „**#**“ verwendet werden.

Im Verlauf des Spiels sollen „Futterhappen“ für die Schlange erscheinen. Diese befinden sich dann in einer bestimmten Zelle. Verwendet werden soll das Zeichen „**O**“, also ein großes O.

Die Schlange wird in diesem Projekt durch den Schlangenkopf, der einer Pfeilspitze ähnelt, und dem Schlangenkörper, der aus dem Buchstaben „**S**“ gebildet wird, dargestellt. Je nach Bewegungsrichtung wechselt die Darstellung des Schlangenkopfes.

Eine Bewegung der Schlange wird dadurch erreicht, in dem man die Position eines Zeichens in dieser Matrix verändert. Soll die Schlange, bestehend aus den Zeichen „**<SSS**“, sich waagerecht von rechts nach links bewegen, müssen also die Inhalte der betreffenden Zellen, die den Schlangenkörper enthalten, jeweils um eine Position nach links „wandern“.

Das Konsolenfenster soll folgende Abmessungen haben: Breite (Width) sind 80 Spalten und die Höhe (Height) sind 40 Zeilen, wobei für das eigentliche Spielfeld mit der Umrandung 30 Zeilen verwendet werden sollen; die verbleibenden 10 Zeilen sind für Ausgaben wie Punktezahl etc. vorgesehen.

Wie Sie bereits wissen, werden Ausgaben in der Konsole entweder mit der Methode **`Write()`** oder mit der Methode **`WriteLine()`** des Konsolenobjekts durchgeführt. Diese beiden Methoden unterscheiden sich dadurch, dass bei der Methode **`WriteLine()`** nach der Ausgabe noch ein Zeilenumbruch ausgegeben wird. Da jedoch immer nur in einzelnen Zellen eine Ausgabe benötigen, wird die Methode **`WriteLine()`** zur Ausgabe der aktuellen Spielsituation nicht benötigt.

Damit die Ausgabe eines Zeichens an einer bestimmten Stelle erfolgt, muss der Cursor an diese Stelle (Zelle) positioniert werden. Hierzu benötigen wir die Methode **`SetCursorPosition()`** des Konsolenobjekts. Diese Methode benötigt als Parameter die nullbasierte Zeilen- und Spaltennummer für eine Zelle in Form eines Ganzzahlwertes.

Um z.B. das Randzeichen „**#**“ in der unteren rechten Ecke auszugeben, muss zuerst der Cursor an diese Position gebracht werden und dann kann die Ausgabe erfolgen.

```
Console.SetCursorPosition(19, 14);
Console.Write("#");
```

Diese Position kann natürlich auch durch entsprechende Integer-Variablen angegeben werden!

Es müssen aber im Rahmen des Spieles nicht nur Ausgaben gemacht werden! Nach jeder Bewegung der Schlange muss u.a. geprüft werden, ob sie z.B. auf einen Futterhappen oder auf den Rand getroffen ist. Damit dies durchgeführt werden kann, muss man also den Inhalt einer bestimmten Zelle „abfragen“ können.

Und für diesen Fall liefert das `Console`-Objekt leider keine Methode! Es lässt sich also nicht direkt ermitteln, welches Zeichen sich an welcher Position befindet. Das ist ein grundlegendes Problem, für das es aber eine relativ einfache Lösung gibt.

Man verwendet eine geeignete Datenstruktur, in der die aktuelle Spielsituation gespeichert wird. Und nach einer Änderung der Spielsituation wird der Inhalt dieser Datenstruktur im Konsolenfenster ausgegeben.

© Uwe Homm Version vom 28. Mai 2020

Mit welcher Datenstruktur lässt sich nun der darzustellende Inhalt des Spielfeldes sinnvoll speichern? Das Spielfeld besteht, wie schon mehrfach erwähnt, im Prinzip aus einer Vielzahl von Zellen, die in Form einer Tabelle mit Zeilen und Spalten angeordnet sind.

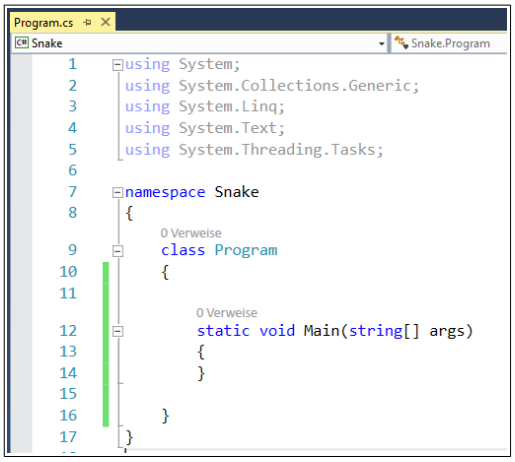
Diese Darstellungsform findet sich nun genauso als ein zweidimensionales Array, mit einer definierten Anzahl von Zeilen und Spalten! Da in jeder Zelle nun genau ein einzelnes Zeichen gespeichert werden soll, ist der sinnvollerweise zu verwendende Datentyp der „Character“ (char), der ja genau ein einzelnes Zeichen aufnehmen kann. Um das Spielfeld zu erzeugen, bietet sich also folgende Anweisung an: `char[,] gameArea = new char[rows, columns];`

Damit wird ein zweidimensionales Array mit der Bezeichnung **`gameArea`** zur Speicherung von Zeichen erstellt. Die gewünschte Anzahl der Zeilen und Spalten in diesem Array wird durch die Variablen **`rows`** und **`columns`** angegeben.

Initialisieren des Konsolenfensters

Beginnen wir mit der Erstellung des Projekts. Erstellen Sie in Visual Studio ein Konsolenprojekt und benennen Sie es der Einfachheit halber „Snake“.

Anschließend sollte das Programm-Grundgerüst etwa so wie im Bild aussehen.



Beginnen wir mit der Implementation. Aber noch eine kurze Vorbemerkung! Kommentieren Sie ihr Programm an allen Stellen, wo es Ihnen sinnvoll erscheint! Lieber einen Kommentar zu viel, als einen zu wenig. Wenn man nach geraumer Zeit solch ein Programm wieder im Quellcode öffnet, helfen die Kommentare sehr, um bestimmte Dinge zu verstehen.

Wie zuvor festgelegt, soll das Konsolenfenster eine bestimmte Größe haben. Wir werden später sehen, dass diese Angaben zur Fenstergröße bei bestimmten Ausgaben benötigt werden.

Daher ist es sinnvoll, diese Angaben im Programm als Konstante zu definieren. Da sich diese Werte nicht ändern und auch innerhalb des gesamten Programms zugreifbar sein sollen, werden die Konstanten direkt nach der Anweisung **`class Program`** und **außerhalb** der Methoden definiert und sind somit für die **gesamte Klasse `Program`** aus jedem Programmteil heraus zugreifbar.

```
0 Verweise
class Program
{
    // Voreinstellungen (Konstanten)
    // ==============================
    // Fenstergröße festlegen
    const int windowWidth = 80;
    const int windowHeight = 40;
    // Unterer Rand für Spielinfos
    const int bottomBorder = 10;
    // Zeichen für Spielfeldrand
    const char gameBorderCharacter = '#';

    0 Verweise
    static void Main(string[] args)
    {
```

Ein weiterer Vorteil bei dieser Art der Definition ist, dass man nur an dieser Stelle etwas ändern muss, um z.B. die Spielfeldgröße oder das Zeichen für den Rand zu ändern.

Wie Sie sehen, verwende ich lieber etwas längere aber dafür aussagekräftigere Bezeichner. Und bei Variablen und Konstanten verwende ich die sogenannte CamelCase-Schreibweise.

Okay...lassen Sie uns nun dafür sorgen, dass das Konsolenfenster in der gewünschten Größe initialisiert und dann das Spielfeld gezeichnet werden kann. Eigentlich ist das ja nur die Umrandung mit dem in der Konstanten **`gameBorderCharacter`** festgelegten Zeichen.

Folgendes muss man wissen: Ausgaben erfolgen bei einer Konsole in einem sogenannten Bild-

schirmpufferbereich[3]. Die Größe des Fensters kann kleiner sein. Dann würden ggf. Rollbalken angezeigt werden. Ist die momentane Fenstergröße nach Programmstart jedoch größer als der anschließend einzustellende Bildschirmpuffer, kommt es zu einem Programmabsturz!

Wenn nun also der Bildschirmpuffer und die Fenstergröße eingestellt werden sollen, kann man sich wie neben dargestellt behelfen.

```
// Konsolenfenster initialisieren
1 Verweis
static void initializeWindow()
{
    // Fenster einstellen
    Console.SetWindowSize(1, 1);
    Console.SetBufferSize(windowWidth, windowHeight);
    Console.SetWindowSize(windowWidth, windowHeight);
    Console.Title = "Snake (by Uwe Homm)";

    Console.BackgroundColor = gameBackgroundColor;
    Console.CursorVisible = false;
    Console.Clear();
}
```

Zuerst wird die aktuelle Fenstergröße auf nur eine Zeile und eine Spalte eingestellt.

Danach wird der Bildschirmpuffer mit den beiden Konstanten **`windowWidth`** (Anzahl der Spalten) und **`windowHeight`** (Anzahl der Zeilen) auf die gewünschte Größe eingestellt.

Nun kann auch die Fenstergröße auf die gleichen Werte eingestellt werden.

Zusätzlich erhält das Konsolenfenster noch einen Titel, der in der Titelzeile des Fensters angezeigt wird. Der Hintergrund des Konsolenfensters bekommt eine Farbe verpasst, die ebenfalls in einer Konstante namens **`gameBackgroundColor`** gespeichert ist. Der Datentyp dieser Konstante ist neu für Sie und lautet **`ConsoleColor`**.

```
// Hintergrundfarbe Spielfeld
const ConsoleColor gameBackgroundColor = ConsoleColor.White;
```

Diese Konstante wird, wie alle anderen auch, direkt im Bereich der Klasse **`Program`** definiert. Der Wert wird durch eine sogenannte „Enumeration“ (d.i. eine Aufzählung) zur Verfügung gestellt. Tippt man bei der Wertangabe hinter dem Wort **`ConsoleColor`** den Punkt, öffnet sich eine Liste mit allen möglichen Werten, die man angeben kann.

Anschließend wird die Ausgabe des Cursors deaktiviert, so dass später bei der Ausgabe der Cursor nicht über den Bildschirm „huscht“ und zum Abschluss wird der Bildschirminhalt noch gelöscht.

Die zuvor angegebene Initialisierung verpackt man am Besten in eine Methode, die man dann bei Bedarf aufruft. Überlegen Sie mit mir, wie diese Methode definiert werden muss.

Zuerst einmal benötigt sie einen Namen. In meinem Projekt wird **`initializeWindow()`** als Bezeichner verwendet, da ja hier die Initialisierung des Bildschirmfensters stattfindet.

Als nächstes muss man darüber nachdenken, ob diese Methode einen Ergebniswert liefern muss, oder ob der Rückgabetyp **`void`** sein kann. Da die Methode nicht wirklich etwas berechnet oder irgendeine Art von Ergebnis hat, nutze ich den Rückgabetyp **`void`**.

Benötigt die Methode irgendwelche zusätzlich Informationen, mit denen Sie arbeiten soll? Falls ja, würde man diese Informationen als Parameter übergeben. Aber diese Methode muss nur wissen, wie groß das Fenster sein soll und mit welchem Zeichen der Rand zu zeichnen ist. All dies steht in den Konstanten, die global definiert und damit für alle Methoden zugänglich sind. Auf Parameter kann also verzichtet werden.

Wie der Methodenkopf aussieht, können Sie dem Kasten oben entnehmen. Diese Methode wird dann zu Beginn der **`Main()`**-Methode aufgerufen.

3 Näheres hierzu unter **`https://docs.microsoft.com/de-de/dotnet/api/system.console?view=netcore-3.1#Buffer`** abgerufen am 13.05.2020

© Uwe Homm Version vom 28. Mai 2020

Initialisieren des Spielfeldes

Kommen wir nun dazu, das Spielfeld vorzubereiten! Wie schon weiter oben beschrieben, soll ein zweidimensionales Array mit der Bezeichnung **`gameArea`** verwendet werden, in der die jeweils aktuelle Spielsituation gespeichert wird.

Die Werte in diesem Array werden dann im Konsolenfenster ausgegeben. Dazu gleich mehr. Erst einmal muss dieses Array vorbereitet werden.

Die Initialisierung des Arrays soll ebenfalls in einer Methode erfolgen, die dann bei Bedarf aufgerufen wird. Auch hier muss man sich einen Bezeichner für diese Methode ausdenken. Gewählt wurde von mir der Bezeichner **`initializeGameArea`**.

Auch hier muss man sich überlegen, ob es Sinn macht, einen Wert zurück zu geben oder ob die Methode vom Typ **`void`** ist. Da auch hier nur eine Initialisierung stattfindet, muss man nicht wirklich einen Wert zurückgeben. Somit wird auch hier der Datentyp **`void`** verwendet.

Wie sieht es nun mit Parametern aus? Benötigt diese Methode ergänzende Informationen? Ja! Die Methode soll ja ein Array initialisieren und die Zeichen für den Spielfeldrand eintragen! Und hierfür wird nun der Name dieses Arrays benötigt. Eine Methode kann nicht direkt auf Daten zugreifen, die außerhalb ihrer Methodendefinition gespeichert werden! Um einer Methode diese Daten zugänglich zu machen, übergibt man diese Daten als Parameter innerhalb der runden Klammern hinter dem Methodenbezeichner (siehe Arbeitsblatt „Methoden“)

Bei der Definition des Methodenkopfes muss man sich eine Parameterbezeichnung ausdenken. Ich verwende hier und auch künftig der Einfachheit halber die ursprüngliche Bezeichnung. Außerdem muss man vor der Parameterbezeichnung den Datentyp angeben.

Somit lautet der Methodenkopf plus leerem Anweisungsblock wie neben dargestellt.

```
static void initializeGameArea(char[,] gameArea)
{
    // Array mit Leerzeichen vorbelegen
    for (int row = 0; row < windowHeight - bottomBorder; row++)
    {
        for (int column = 0; column < windowWidth; column++)
        {
            gameArea[row, column] = ' ';
        }
    }
    // Zeichnen der Spielfeldumrandung
    // To Do
}
```

Was muss in dieser Methode nun gemacht werden?

Zuerst einmal muss jedes Arrayelement (also quasi jede Zelle) mit einem Leerzeichen befüllt werden. Dies geschieht durch zwei ineinander geschachtelte **`for`**-Schleifen.

Warum **`for`**-Schleifen?

Ganz einfach! Für diese Aufgabenstellung ist im Voraus bekannt, wie viele Zeilen und Spalten zu durchlaufen sind! Diese Werte sind ja in den Konstanten **`windowWidth`**, **`windowHeight`** und **`bottomBorder`** bereits festgelegt. Also ein typischer Fall für eine Zählschleife. Als Schleifenvariablen habe ich die Bezeichner **`row`** (=Zeile) und **`column`** (=Spalte) gewählt.

Die Anzahl der Zeilen ergibt sich aus der Höhe des Fensters abzüglich der Anzahl der Zeilen für die Spielinformationen (`windowHeight - bottomBorder`). Die Anzahl der Spalten ergibt sich aus der Breite des Fensters (`windowWidth`).

Nach dem Durchlauf der beiden ineinander geschachtelten Schleifen, enthält jedes Arrayelement nun ein Leerzeichen. Im Anschluss daran, kann nun der Spielfeldrand eingezeichnet werden.

© Uwe Homm Version vom 28. Mai 2020

Anzeigen der aktuellen Spielsituation

Nun soll es um die vorerst letzte Methode gehen: Diese Methode soll den Inhalt des Array **`gameArea`** im Konsolenfenster ausgeben.

Als Methodenbezeichner wird **`drawScreen`** verwendet.Es soll ja auf den Bildschirm „gezeichnet“ werden. Auch diese Methode benötigt nicht wirklich einen Rückgabewert, daher ist der Datentyp **`void`** (=leer). Aber die Methode muss wissen, welche Informationen auszugeben sind! Daher muss die Methode wiederum das Array **`gameArea`** als Parameter erhalten. Der Datentyp von **`gameArea`** ist ja bekannt.

Somit lautet der Methodenkopf wie im Kasten dargestellt.

```
static void drawScreen(char[,] gameArea)
{
    // Hier wird der gesamte Inhalt des Array gameArea
    // tatsächlich in das Fenster geschrieben

    // To Do
}
```

Die Implementation übernehmen Sie als Übung! Orientieren Sie sich hierbei an der Initialisierung des Arrays **`gameArea`**!

Diese Methode wird dann ebenfalls in der **`Main()`**-Methode aufgerufen.

Momentan sieht die Main-Methode daher noch wie im unteren Kasten dargestellt aus, aber das wird sich bald ändern :-)

```
static void Main(string[] args)
{
    // Der Inhalt dieses Arrays wird zur Darstellung
    // der aktuellen Spielsituation ausgegeben
    // gameArea[rows, columns]
    char[,] gameArea = new char[windowHeight - bottomBorder, windowWidth];

    // Fenster initialisieren
    initializeWindow();

    // Spielfeld initialisiere
    initializeGameArea(gameArea);

    // Inhalt des Spielfeldes auf
    // Bildschirm ausgeben
    drawScreen(gameArea);

}
```

© Uwe Homm Version vom 28. Mai 2020

Übungen

1. Erstellen Sie ein Konsolenprojekt und benennen Sie es einfach nur Snake :-)

2. Tragen Sie an der angegebenen Stelle alle bisher verwendeten Konstanten ein.
   Siehe hierzu Seite 3.
   Ergänzen Sie die **`Main()`**-Methode wie auf der vorigen Seite dargestellt.

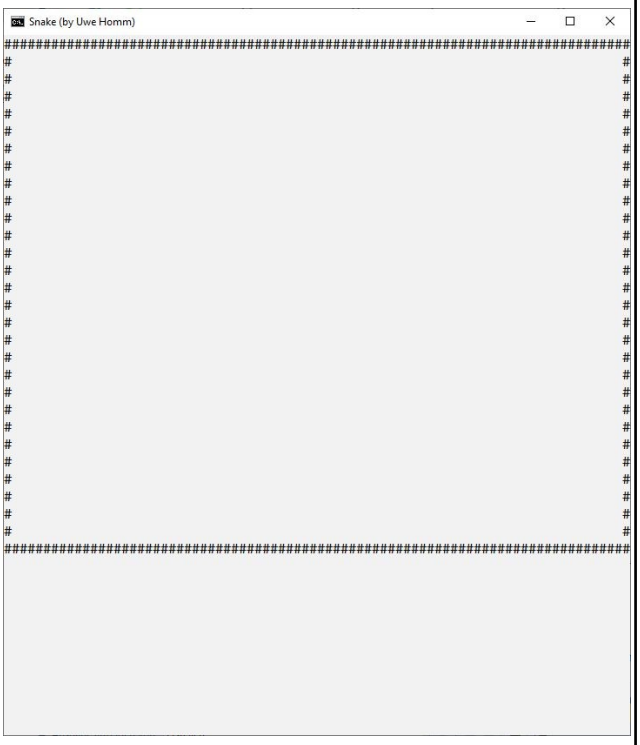
3. Implementieren Sie alle zuvor genannten Methoden außer- und unterhalb von **`Main()`**.

4. Ergänzen Sie die oben teilweise dargestellte Methode **`initializeGameArea`**.
   Versuchen Sie -möglichst elegant und mit wenigen Anweisungen mittels Wiederholungsstrukturen- den Rand des Spielfeldes in das Array **`gameArea`** an der mit **`//To Do`** gekennzeichneten Stelle einzutragen.
   Verwenden Sie hierzu die Zeichenkonstante **`gameBorderCharacter`**.

5. Definieren Sie bei den anderen Konstanten eine weitere Konstante mit der Bezeichnung **`gameBorderColor`**.
   Weisen Sie dieser Konstanten eine Wert zu; z.B. **`ConsoleColor.Black`**.
   Implementieren Sie nun die Methode **`drawScreen()`**, um die Zeichen in **`gameArea`** auf dem Bildschirm auszugeben.
   Stellen Sie vor der Ausgabe die Vordergrund- und Hintergrundfarbe ein.

6. Starten Sie das Programm mit **`STRG-F5`**.
   Das Ergebnis sollte wie dargestellt aussehen.
   Nur die Farben können u.U. abweichen :-)

7. Experimentieren Sie mit anderen Werten für die Fensterbreite und -höhe oder für die Farbwerte
   Wenn Sie **überall** die Konstanten verwendet haben, sollte dies problemlos möglich sein



© Uwe Homm Version vom 28. Mai 2020