

Programmier AG

Stoffsammlung

Version: 1.0

Erstellt von: Jannik Gröger

Inhalt

Projekterstellung mit Template	4
Datentypen.....	7
Variablen	8
Objekttypen	9
Objekterstellung in SFML	10
Rechenoperationen und Operanden	15
Einfache Bewegungen	16
Verzweigungen & Bedingungen.....	18
Die if-Anweisung	18
Bedingungen	19
Einfaches Objektverhalten	20
Events	21
User Input	23
User Input über Abfrage	24
User Input mit Events.....	25
Methoden	27
Arrays	29
Schleifen	30
Listen	32
Objekte hinzufügen & entfernen	34
Kollisionsabfrage mit Hitbox.....	36
Abfrage der Kollision	37
Kollision mit Shapes	38
Bewegungskontrolle	39
Externe Ressourcen.....	41
Texturen & Sprites	43

Musik & Sound	44
Spritesheets.....	45
Timer/Counter	46
Animationen	47
Klassen & Objekte	53
Klassen für Spielobjekte: Entities.....	58
Objekt Orientierung: Vererbung	61
Entities erstellen	63
Physik-Simulation	69
Beispiel.....	69
Simulation von Elastizität: Gummiband	70
Graphical User Interface(GUI)	74
Der Button.....	75
Screens	78
SubScreens.....	83
Speichern & Laden	85
Tilemaps & Tilesets	90
Netzwerkkommunikation &	95
Client-Server-Modell	95
Client-Server-Modell	96
Online Multiplayer mit SFML.....	97
Zustandsautomaten.....	103
Künstliche Intelligenz.....	104

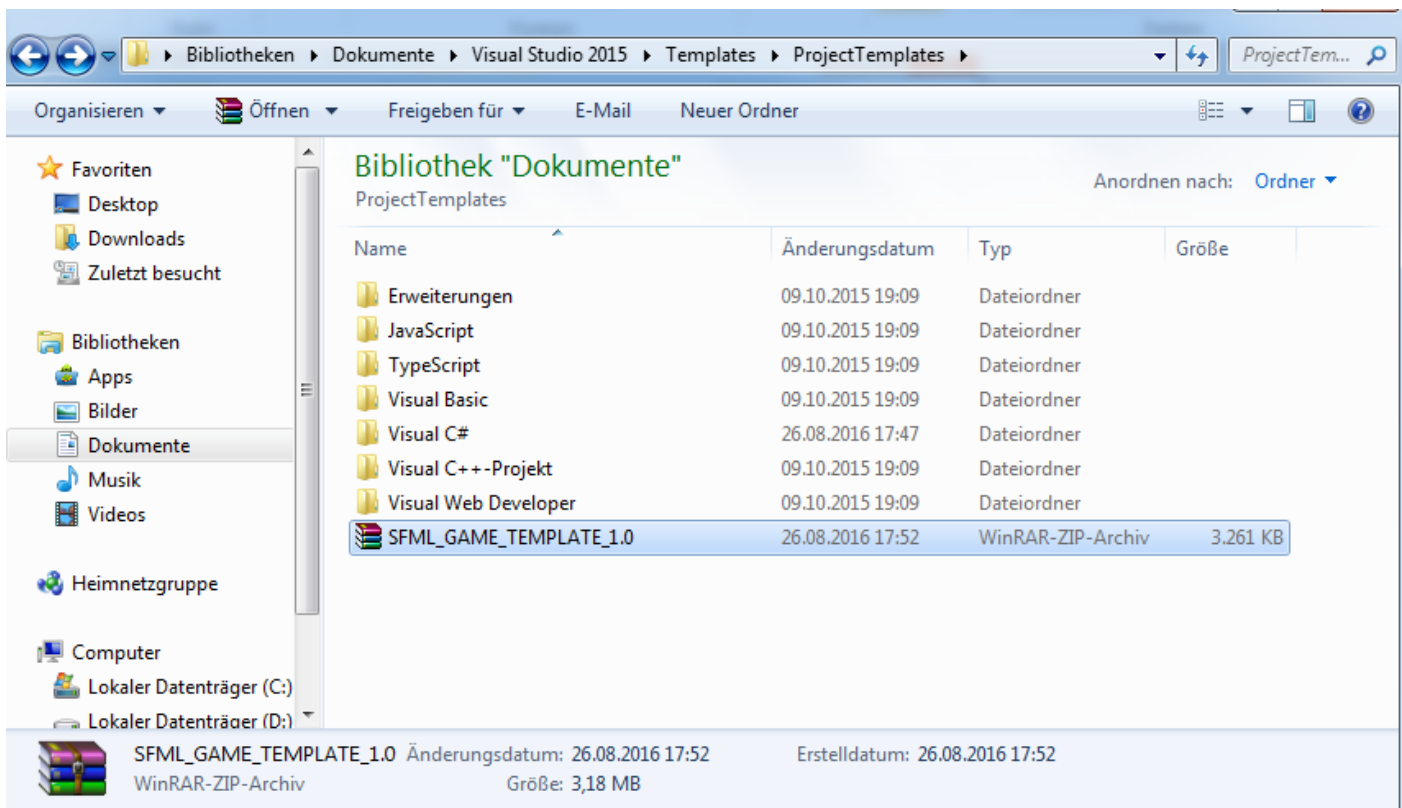
Projekterstellung mit Template

Um einen einfachen Einstieg zu ermöglichen, benutzen wir ein Template, dass bereits das grundlegende Gerüst eines SFML Projektes hat. Dazu nehmen wir das .zip-Archiv "SFML_Game_Template" und legen es unter:

"C:\Users\<User>\Documents\Visual Studio 2015\Templates\ProjectTemplates"

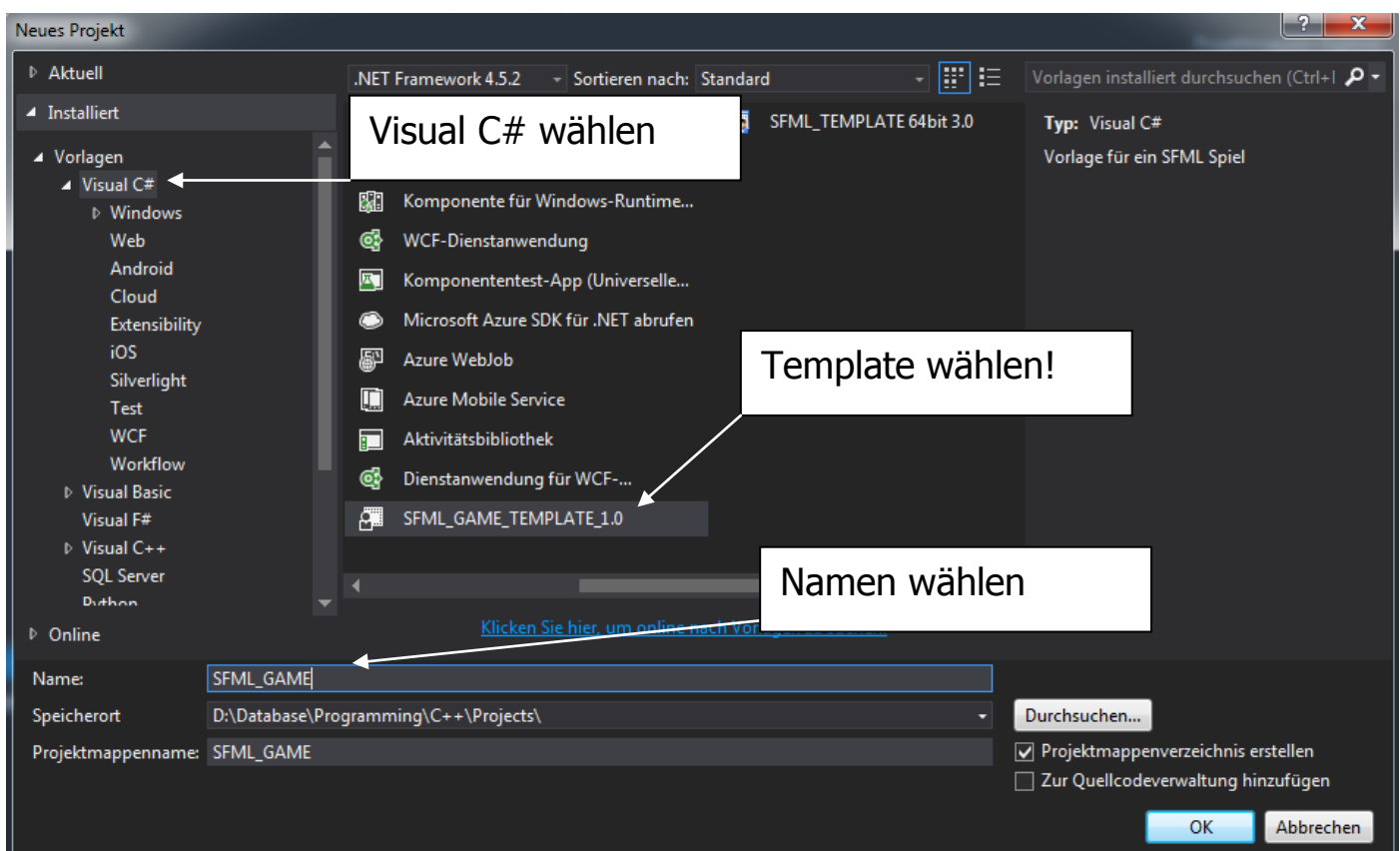
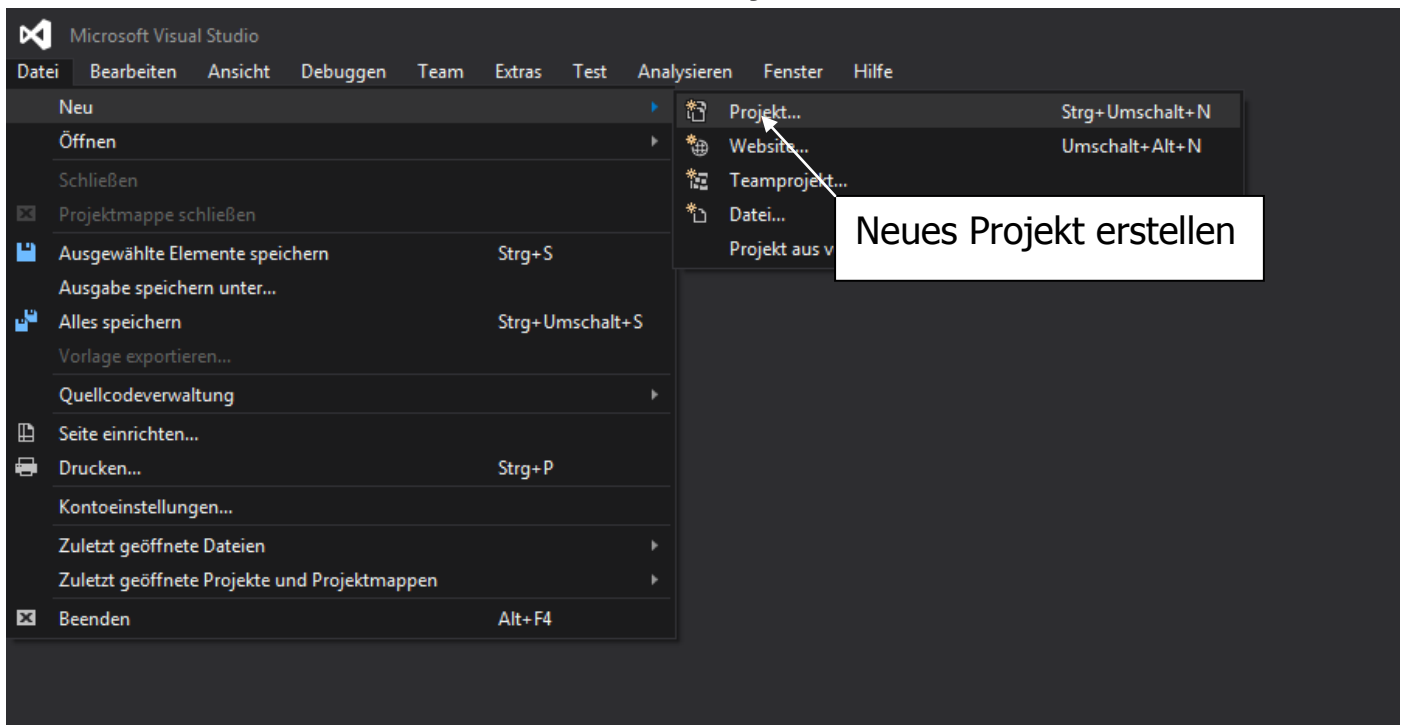
ab. (.zip Archiv nicht entpacken!)

Um den ProjectTemplates Ordner zu finden, geht es am einfachsten im Startmenü(unten links) auf Dokumente zu klicken, von dort in den Ordner Visual Studio 2015, dann Templates und schließlich ProjectTemplates.



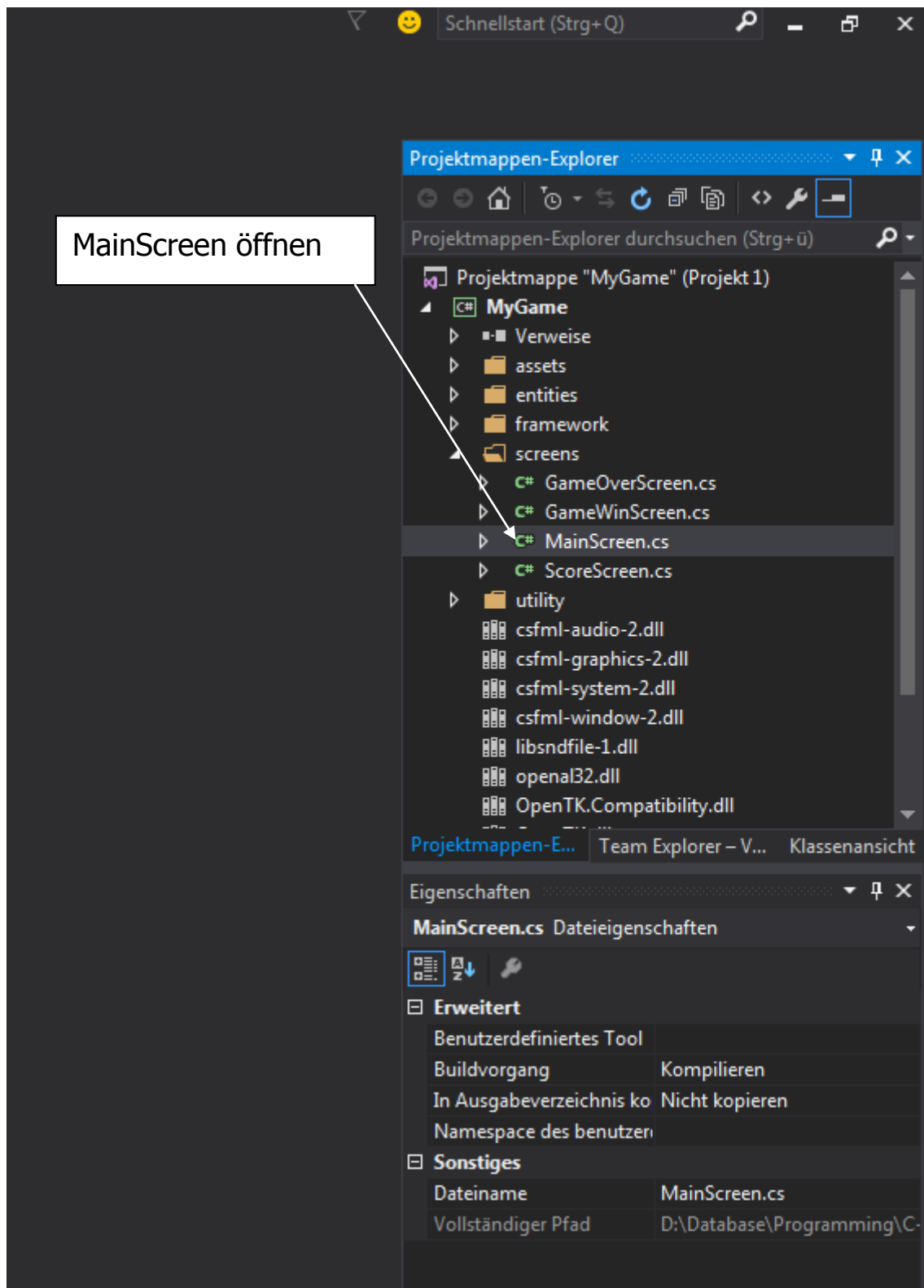
Nun starten wir Visual Studio 2015.

Oben links auf "Datei" -> "Neu" -> "Projekt..." klicken.



Wähle im neuen Fenster "SFML_Game_Template" aus. Gebe einen passenden Namen für dein neues Projekt ein und klicke auf OK.

Im Projektmappen Explorer rechts unter "screens" die mainScreen Klasse öffnen(Falls es keinen Projektmappen-Explorer gibt, in der oberen Leiste unter "Ansicht" öffnen). Jetzt ist alles bereit.



Datentypen

Ein Datentyp ist ein Format in dem Daten gespeichert werden können. In einem Programm werden alle Informationen mithilfe von Datentypen gespeichert.

Dabei gibt es verschiedene Typen, die verschiedene Arten von Daten speichern können. Es ist wichtig für die Daten die richtigen Datentypen auszuwählen.

Hier ist eine Auflistung der wichtigsten Datentypen:

Name	Inhalt	Beispiele
bool (Boolean)	true oder false	true false
int (Integer)	Positive und negative Ganzzahlen (keine Komma- Zahlen!)	1 2 0 5 -89 256 1003 -6
float (Single)	Dezimalzahlen mit "f" hinten (mit oder ohne Punkt komma)	7.23544f 9f -56f 40.8f 1.33333f 0.2f
double	Dezimalzahlen mit Punkt komma (hat mehr Nachkommastellen)	7.23544 9.0 -56.0 40.8 3.14159265359
string	Folge von Zeichen und Symbolen(Texten) Muss von doppelten Anführungszeichen umgeben sein (Manche Sonderzeichen müssen mit einem \ "escaped" werden)	"Peter" "18.56" "" " " "Ich esse gerne Äpfel." "Sie sagte:\"Hallo!\""

Variablen

Wenn man seine Daten nun speichern will, muss Variablen verwenden. Variablen sind „Kisten“ in die wir unsere Werte reinpacken. Eine Variable besteht aus 3 Dingen:

Dem **Bezeichner**, der „Name“ der Variable, um sie von anderen zu unterscheiden. Hier kann sich der programmier einen passenden Namen überlegen.

Dem **Datentyp**, der angibt, welche Arten von Daten man in ihr speichern kann und was man mit ihnen machen kann.

Dem **Inhalt**, die Daten, die letztendlich in der Variable gespeichert werden.

Während Bezeichner und Datentyp einer Variable sich nicht ändern, kann der Inhalt einer Variable geändert werden oder sogar unbestimmt sein.

Um in C# eine Variable zu erstellen, benötigt man ein **Statement**.

Quellcode ist im Grunde nichts anderes als eine lange Liste von Statements.

Um eine Variable zu erstellen, muss man sie erst **Deklarieren**.

Deklaration:

`int zahl1;`

Bezeichner

Datentyp

Zuweisung:

`zahl1 = 25;`

Inhalt

Initialisierung:

`int zahl1 = 36;`

Weitere Beispiele:

Beim Deklarieren wird der Bezeichner festgelegt und der Datentyp angegeben (**Am Ende jedes Statements muss ein Semikolon(;) stehen**). Der Inhalt wird noch nicht angegeben, und ist damit unbestimmt. Eine Variable deren Inhalt unbestimmt wird als **nicht initialisiert** bezeichnet. Um die Variable zu initialisieren, muss man ihr einen Wert zuweisen. Dies wird über eine Zuweisung getan. Dazu wird nur der Bezeichner angegeben und mit dem **Zuweisungsoperator =** und einem Wert mit dem **gleichen Datentypen** auf der rechten Seite versehen. Man kann die Variable schon beim Deklarieren initialisieren.

bool aktiv = true;

string name = "Max Mustermann";

float prozent = 45.67f;

double groesse = 176.45;

int anzahl = 6;

Objekttypen

Die Daten *bool*, *string*, *float*, *double* und *int* zählen zu den "primitiven" Datentypen. Neben den primitiven Datentypen gibt es die Objekttypen, davon gibt es beliebig viele. Diese Arten von Typen haben keinen einfachen Wert den man ihnen gibt, stattdessen werden sie durch eine Kombination von Werten von anderen Typen erstellt. Dazu verwendet man den "new" Befehl.

Hätte man zum Beispiel einen Typen "*Dreieck*", der aus 3 *double* Werten besteht die die Kantenlängen angeben, würde man ihn so erstellen:

Dreieck dach = new Dreieck(4.6 , 6.4 , 2.5);

So hätte man ein neues Dreieck mit dem Namen "dach" und den Kantenlängen 4.6, 6.4 und 2.5. Welche Werte man beim Erstellen der Variable angeben muss, ist abhängig von dem Typen. Visual Studio gibt üblicherweise Hinweise beim Schreiben des Codes, welche Typen man benötigt. Diese anderen Arten von Typen nennt man auch **Objekttypen**, bzw. ihre Werte **Objekte**. Objekte haben sogenannte **Member**, die auch wieder primitive Typen oder Objekttypen sind. Auf Member eines Objekts kann man mit dem "." Operator zugreifen. Hätte das Dreieck hier die Member kanteA, kanteB und kanteC vom Typ double, könnte man das Dreieck so verändern: *dach.kanteA = 5.0;*

Objekterstellung in SFML

Die Grafikbibliothek SFML bietet viele Hilfsmittel, um Dinge in dem Programmfenster anzuzeigen. Die drei wichtigsten Objekte dazu sind:

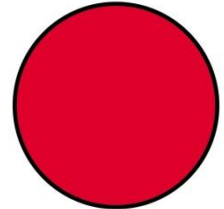
Sprite



RectangleShape



CircleShape



Das **RectangleShape** ist eine rechteckige einfarbige Form. Die Größe und Breite sowie die Farbe sind festlegbar. Ein **CircleShape** ist ähnlich zum **RectangleShape**, nur dass es rund ist.

Das **Sprite** stellt ein von der Festplatte geladenes Bild dar und kann damit alle möglichen 2D Formen darstellen. Da der Umgang mit Sprites allerdings komplex ist, liegt der Fokus zuerst bei den Shapes.

Ein Objekt, welches wir auch oft verwenden werden ist **Vector2f**.

Ein **Vector2f** beschreibt einen 2-Dimensionalen Vektor aus floats. Er besteht dementsprechend nur aus 2 float Werten. Um einen **Vector2f** zu erstellen, brauchen wir den "new" Befehl.

```
Vector2f vektor = new Vector2f(45f, 53f);
```

Hier erstellen wir einen neuen **Vector2f**, dazu müssen wir 2 float Werte angeben, die diesen definieren. Die Werte stehen hier für Pixel. Das Programmfenster ist 800 x 600 Pixel groß, ein Vektor mit 800x600 würde also von einer Ecke zur anderen gehen.

Wenn man RectangleShapes erstellen will, gibt es zwei Möglichkeiten.

```
RectangleShape shape = new RectangleShape();
```

```
Vector2f vektor = new Vector2f(100, 100);
```

```
RectangleShape shape2 = new RectangleShape(vektor);
```

Man kann das RectangleShape ohne Daten erstellen, oder mit einem Vektor2f. Erstellt man das RectangleShape ohne Daten, so hat es zuerst die Länge und Breite 0, ist also unendlich klein und damit nicht sichtbar.

Gibt man einen Vektor an, so hat das RectangleShape die Dimensionen des Vektors, hier 100x100.

Man kann das RectangleShape nicht nur bei der Erstellung definieren. Die Größe eines RectangleShapes ist durch den **Member** Size gegeben. Möchte man ein bereits erstelltes Shape ändern, so benutzt man den Punkt Operator ".":

```
shape.Size = new Vector2f(50,20);
```

So kann man das RectangleShape beliebig anpassen.

Andere Member die oft genutzt werden:

```
shape.Rotation = 40.5f;
```

Rotation gibt die Drehung des Shapes in Grad an(0° - 360°), der Winkel wird als Float übergeben. Negative Beträge und Beträge über 360 werden umgerechnet:
-30° = 330° , 380° = 20°, 730° = 10°

```
shape.FillColor = Color.Blue;
```

Die Farbe des Shapes, kann man gut über die Color Klasse angeben, kann aber auch mit einem new Operator über 3 Werte(RGB) erstellt werden.

```
shape.Origin = 0.5f * shape.Size;
```

Der Origin des Shapes ist der innerhalb des Shapes der als Mittelpunkt für Drehungen genutzt wird und Referenzpunkt für die Position. Standardmäßig ist der Origin in der oben linken Ecke, für Drehungen ist es sinnvoll ihn in die Mitte zu bewegen.

Um nun ein Objekt dem Fenster hinzuzufügen, müssen wir mehrere Dinge tun, zuerst sollten wir ein neues Objekt **deklarieren**. Das tun wir unter der Klassendefinition, aber über den Methoden!

Das Objekt das wir haben wollen heißt "RectangleShape", eine Rechteckform. Um das Objekt zu deklarieren, schreiben wir den Objekttyp hin und geben ihm einen beliebigen Namen. Am Ende jeder Anweisung sollte immer ein Semikolon (;) stehen!

```
// Deklariere hier Objekte!  
RectangleShape BoxA;
```

Jetzt wo wir unser Objekt deklariert haben, müssen wir es **initialisieren**, dazu gehen wir in die setup() Methode, und schreiben dort:

```
// Setup, wird immer einmal zu Beginn eines Screens aufgerufen  
// Hier Startwerte setzen!  
override public void setup()  
{  
    BoxA = new RectangleShape();  
}
```

Hier wird ein neues RectangleShape erzeugt, erkennbar an dem Wort "new".

Der Zuweisungsoperator: =

Das Gleichzeichen wird in der Informatik als Zuweisungsoperator benutzt. Man verwendet es, um einer Variable einen neuen Wert zuzuweisen. Dazu schreibt man erst den Namen der Variable, den Zuweisungsoperator und dann den neuen Wert. Diese Reihenfolge ist wichtig! Hier geben wir der Variable "wertA" den neuen Wert 5.

wertA = 5; <--- *Richtig!*

5 = wertA; <--- *Falsch!*

Als letztes, müssen wir noch dafür sorgen, dass unser Objekt auf dem Bildschirm erscheint, dazu schreiben wir in die loop() Methode:

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    draw(BoxA);
}
```

Die "Draw()" Methode zeichnet unser Objekt jetzt 60 mal die Sekunde auf den Bildschirm. Allerdings, sehen wir bei erneutem Drücken auf F5 noch nichts. Das liegt daran, dass es zwar ein Rechteckobjekt gibt, es aber noch eine Größe von 0 besitzt.

Jetzt müssen wir also anfangen, die Eigenschaften unseres Objektes zu verändern. Dazu schreiben wir den Objektnamen und danach den Namen der Eigenschaft, getrennt durch ein "." .

Da wir die Eigenschaften unseres Objektes nur einmal festlegen müssen, schreiben wir sie in die setup()-Methode.

Die Eigenschaft, die wir verändern wollen ist "Size", die Größe.

Also schreiben wir :

```
// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
override public void setup()
{
    BoxA = new RectangleShape();

    BoxA.Size = new Vector2f(100, 100);
}
```

Die Eigenschaft "Size" ist ein Vector2f Objekt, um dieses zu setzen müssen wir mit "new Vector2f()" ein neuen Vector erstellen. In die Klammer kommen 2 Zahlenwerte mit Komma getrennt, der erste ist der X Wert (Die Breite) und der zweite der Y Wert(Die Länge). Wir erstellen hier also ein Rechteck mit 100 Pixeln Breite und 100 Pixeln Länge.

Wenn wir F5 drücken, sollten wir das Rechteck sehen können.

Jetzt können wir neben der Größe, auch andere Eigenschaften ändern.

Farbe, Position, Drehung. Hier ein paar Beispiele:

```
// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
override public void setup()
{
    BoxA = new RectangleShape();

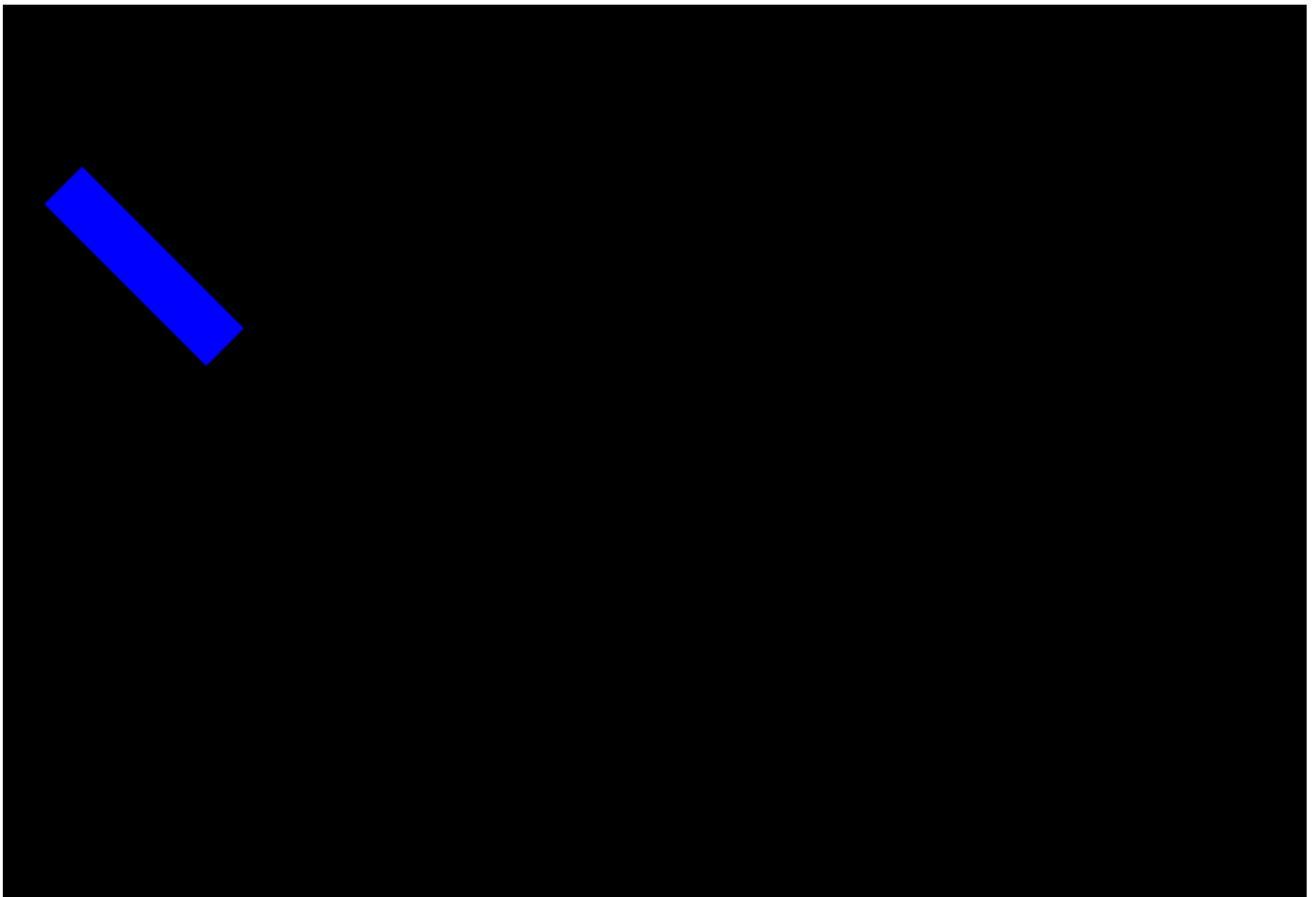
    // Setze Größe auf 140x33
    BoxA.Size = new Vector2f(140, 33);

    // Setze Farbe auf Blau
    BoxA.FillColor = Color.Blue;

    //Winkel = 45 Grad
    BoxA.Rotation = 45;

    //Platziere auf Punkt (50,150)
    BoxA.Position = new Vector2f(50, 150);
}
```

Aus dem Beispiel oben würde dann folgendes Bild entstehen:



Rechenoperationen und Operanden

Sobald Variablen initialisiert wurden, kann man mit ihnen Rechnen. Dazu braucht man immer zwei Variablen(oder direkte Werte wie z.B. 36) und einen Operator. Die Operanden Plus "+", Minus "-", Mal "*", Geteilt "/" funktionieren wie in der Mathematik. Es gibt noch Modulo("%"), der den Rest vom Teilen zurückgibt. ($10 / 3 = 3$ Rest 1 | $10 \% 3 = 1$)

Beispiele:

```
int A = 1;           // A = 1
int B = 3 + A;       // B = 4
int C = B / 2;       // C = 2
int D = C * B;       // D = 8
int E = D + C;       // E = 10
int F = E - B;       // F = 6
int G = F % 4;       // G = 2
```

Bei diesen Operanden werden Zahlenwerte benötigt(int,double,float) und das Ergebnis ist wieder ein Zahlenwert mit einem dieser Datentypen.

Man beachte, wenn man zwei **INT** durcheinander teilt, kommt auch ein **INT** raus, egal wie man es speichert. Deswegen sollte man wenn man mit Kommazahlen arbeitet immer FLOAT oder DOUBLE benutzen.

```
int A = 5;
int B = 2;
double C = A / B;
// C = 2

double D = 5 / 2; // Ganze zahlen ohne Komma gelten als INT,
// D = 2.0        deswegen keine Kommazahl als Ergebnis

double E = 5.0/2.0; //Zahlen mit Komma(ohne f) gelten als DOUBLE
// E = 2.5

float F = 5f / 2.0f; //Zahlen mit "f" am Ende, gelten als FLOAT
// F = 2.5f
```

Will man den Wert einer Variable nicht absolut setzen(z.B. = 5) sondern relativ erhöhen oder senken (z.B. 3 höher als vorher), kann man die Rechenoperation abkürzen durch die +=, -=, *=, /= Operatoren.

```
int A = 3;
A = A + 2; // A = 5
A += 5;    // A = 10
A *= 2;    // A = 20
A /= 10;   // A = 2
A -= 5;    // A = -3
```

Einfache Bewegungen

Nach der erfolgreichen Erstellung eines RectangleShape Objekt und dem Setzen der Eigenschaften, wollen wir unserem Objekten ein wenig Leben einhauchen. Und zwar in dem wir sie Bewegen.

Was ist eine Bewegung in einem Programm überhaupt?

In Programmen kann man eine Bewegung als ein ständiges Ändern von Werten, z.b. der Position, bezeichnen. Und genau das wollen wir auch tun. Deswegen gehen wir in unsere Loop() Methode, denn diese wird 60 mal die Sekunde ausgeführt, also quasi "ständig".

Hier ändern wir die Werte. Wenn wir allerdings nur versuchen die Werte zu setzen, z. b. mit:

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    BoxA.Position = new Vector2f(250, 250);

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

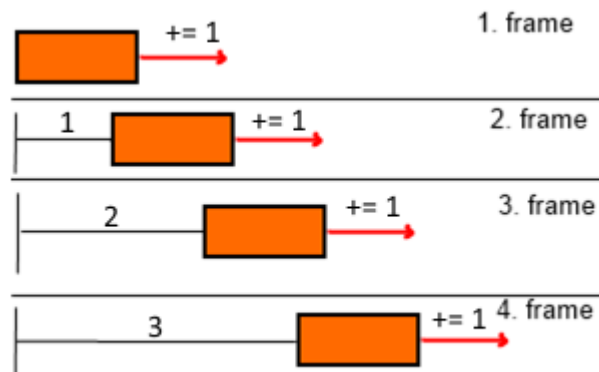
..verändert es zwar die Position beim ersten Durchlaufen, aber danach bleibt es dort. Wir wollen aber eine ständige Veränderung, also dürfen wir die Werte nicht immer auf einen absoluten Wert setzen, sondern müssen sie relativ verändern! Dazu benutzen wir die **arithmetischen Operatoren +=**. Anstatt also die Position auf einen fixen Wert zu setzen, setzen wir sie auf einen Wert der relativ zum Alten verändert ist.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    // 60 Pixel pro Sekunde nach rechts
    BoxA.Position += new Vector2f(1, 0);

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

Hier setzen wir den Wert auf die alte Position, aber mit der X Komponente um einen erhöht.

Wenn wir nun unser Programm debuggen, können wir sehen wie sich unser Objekt nach rechts bewegt. Da wir sagen, dass es sich um 1 Pixel pro Frame bewegt, und wir eine Framerate von 60 mal die Sekunde besitzen, ergibt sich eine Geschwindigkeit von 60 Pixeln pro Sekunde.



Um das Objekt nach links zu bewegen, erhöhen wir den X-Wert nicht, sondern verringern ihn.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    // 60 Pixel pro Sekunde nach links
    BoxA.Position += new Vector2f(-1, 0);

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

Gleiches gilt für eine Bewegung nach unten und oben, wenn man die Y-Komponente verändert.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    // 60 Pixel pro Sekunde nach links und oben
    BoxA.Position += new Vector2f(-1, -1);

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

Wenn man Dezimalzahlen nehmen will, muss man diese immer mit "f" beenden, da es C# für Float Werte so vorschreibt.

```
override public void loop() {
    // 30 Pixel pro Sekunde nach links
    BoxA.Position += new Vector2f(-0.5f, 0);

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

Verzweigungen & Bedingungen

Die if-Anweisung

Die if-Anweisung ist eine Anweisung, die je nachdem ob ein angegebener **Boolwert** wahr oder falsch ist, einen **Codeblock** ausführt.

```
int maxSpeed = 80;

bool regen = true;

if (regen) // Regen ist true, also wird Klammer ausgeführt
{
    maxSpeed = 50;
}
```

Wenn der Boolwert wahr ist, wird der Codeblock ausgeführt, wenn er falsch ist, dann wird der Codeblock nicht ausgeführt.

Eine Erweiterung der if-Anweisung ist die if-else-Anweisung. Bei ihr gibt es einen **zweiten Codeblock**, der nur ausgeführt wird, wenn **der erste** nicht ausgeführt wird.

```
int geld = 500;
string sitzKlasse;

if (geld < 300) // Geld ist größer als 300 also wird if block nicht ausgeführt
{
    sitzKlasse = "Standart";
}
else // if block wurde nicht ausgeführt, also wird else block ausgeführt
{
    sitzKlasse = "Premium";
}
```

Bedingungen

Die if-Klammer enthält immer eine Bedingung in Form eines Bool Wertes. Statt einer Bool Variable kann aber auch Vergleiche benutzen. Dazu gibt es die Operatoren:

<code>==</code> - ist gleich	<code>/=</code> - ist nicht gleich (ungleich)
<code>></code> - ist größer als	<code><</code> - ist kleiner als
<code>>=</code> - ist größer als oder gleich	<code><=</code> - ist kleiner als oder gleich

Beispiele:

```
bool A = 10 > 5;           // A = true
bool B = 10 - 3 > 9;       // B = false
bool C = 5 - 3 == 2;       // C = true
bool D = 20 >= 20;         // D = true
bool E = 50 <= 49;         // E = false
```

Man kann auch Vergleiche mit Bool-Werten bilden, die Operatoren dafür sind: `&&` (AND) `||` (OR) `!` (NOT)

```
bool A = true && false;
//Der AND-Operator "&&" wird nur wahr, wenn beide Operanden true sind

bool B = false || true;
//Der OR-Operator "||" wird nur wahr, wenn min. ein Operand wahr ist.

bool C = !A;
//Der NOT-Operator wird gibt das Gegenteil des Operanden aus, also !true = false und
!false = true
```

Einfaches Objektverhalten

Wir können nun Objekte erstellen und sie auf verschiedene Arten in Bewegung versetzen. Jetzt wollen wir das Verhalten noch mehr kontrollieren, indem wir Bedingungen und Verzweigungen benutzen.

Hier ein kleines Anwendungsbeispiel:

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
override public void loop()
{
    BoxA.Position += new Vector2f(3, 0);

    if (BoxA.Position.X > 500)
    {
        BoxA.Position = new Vector2f(0, 200);
    }

    // Zeichne das Rechteck auf den Bildschirm
    draw(BoxA);
}
```

Hier sehen wir, dass wir ein Objekt nach rechts bewegen.

Die If Klammer hat als Bedingung "BoxA.Position.X > 500", die Bedingung ist also "true" sobald BoxA sich so weit nach rechts bewegt hat, dass seine X-Position größer als 500 ist. Dann wird der Code in dem If-Block ausgeführt und setzt die Position zurück auf 0|200. Man nennt sowas eine Verzweigung, weil es mehrere mögliche Wege für den Code bedeutet.

Der else-block wird immer dann ausgeführt, wenn der if-block nicht ausgeführt wird. Also wird immer einer von beiden Blöcken ausgeführt, aber nie beide und nie keins.

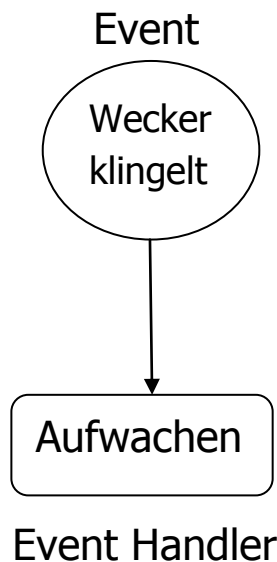
```
if (BoxA.Position.X < 500) {
    BoxA.Position += new Vector2f(3, 0);
}
else {
    BoxA.Position += new Vector2f(0, 3);
}

// Zeichne das Rechteck auf den Bildschirm
draw(BoxA);
```

Hier sehen wir, dass wenn BoxA einen X Wert unter 500 hat, dass es sich nach rechts bewegt, aber wenn BoxA 500 überschreitet, bewegt er sich nach unten.

Events

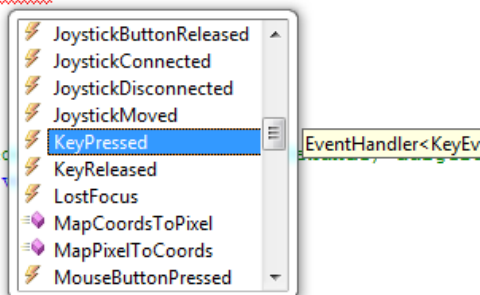
Events in C# kann man benutzen, um bestimmtes Verhalten an ein auslösendes Ereignis zu binden. Dazu bindet EventHandler Funktionen an Events. Wenn also ein bestimmtes Ereignis "Event", eintritt, werden alle darauf gebunden EventHandler Funktionen ausgeführt.



Hier ein Beispiel:

Das Fenster von SFML, mit dem wir Arbeiten ist ein Objekt vom Typ "RenderWindow". Von der Mainscreen Klasse kann man mithilfe von "game.gameWindow" darauf zugreifen.

```
// Setup, wird immer einmal zu Beginn eines Screens au  
// Hier Startwerte setzen!  
public override void setup()  
{  
  
    game.gameWindow.  
  
}  
  
// Loop, wird jede  
public override v  
{
```



Die in der Memberliste aufgeführten "Blitze" sind Events, bzw. Eventhandler. Wir können nun mit dem += Befehl, einen neuen Eventhandler einem Event hinzufügen.

```
// -----  
// Hier Startwerte setzen!  
public override void setup()  
{  
  
    game.gameWindow.MouseButtonPressed +=  
        new EventHandler<MouseButtonEventArgs>(gameWindow_MouseButtonPressed); (Press TAB to insert)  
  
}
```

Visual Studio gibt uns hier die Möglichkeit mit dem Drücken von TAB automatisch einen Eventhandler zu erstellen und die Methode dazu auch.
(Kann etwas schwierig sein, muss man ausprobieren bis es geht)

nn eines Screens aufgerufen

```
sed +=new EventHandler<MouseButtonEventArgs>(gameWindow_MouseButtonPressed);  
Press TAB to generate handler 'gameWindow_MouseButtonPressed' in this class
```

ie Sekunde) aufgerufen

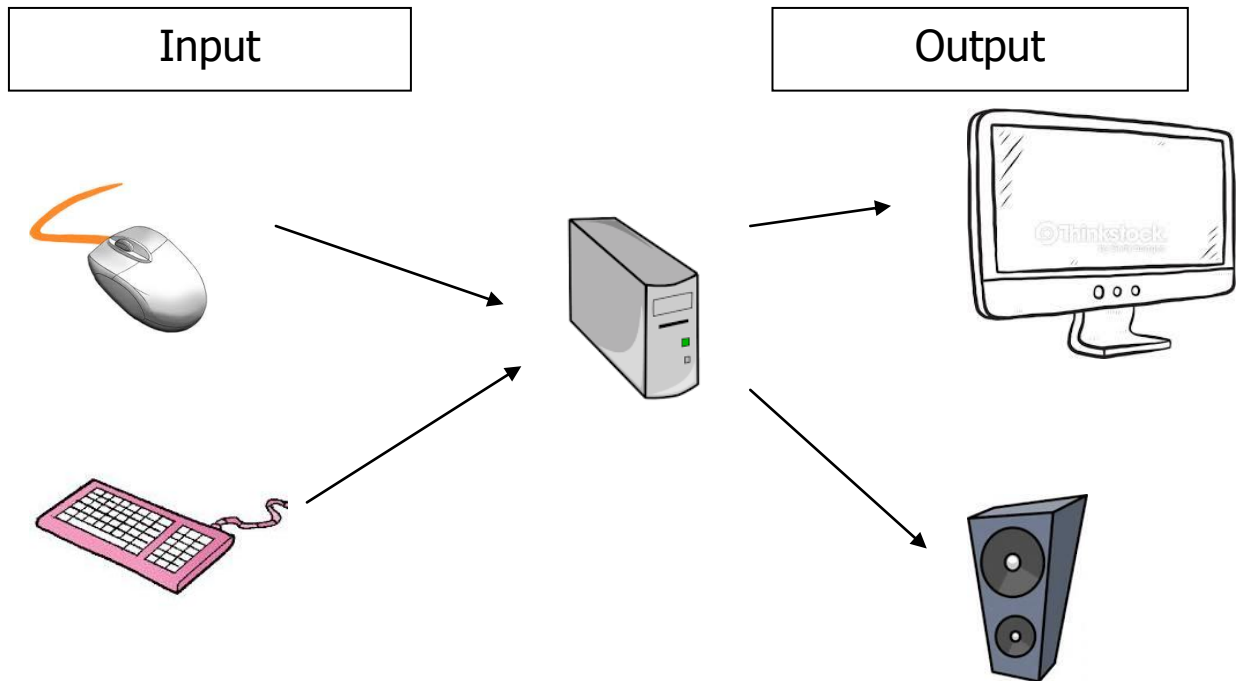
Den Namen der Eventhandler Methode kann man frei wählen.

```
public override void setup()  
{  
  
    game.gameWindow.MouseButtonPressed += new EventHandler<MouseButtonEventArgs>(OnMouseButton);  
  
}  
  
void OnMouseButton(object sender, MouseButtonEventArgs e)  
{  
    throw new NotImplementedException();  
}
```

Die so entstandene Methode besitzt ein "throw new [...]Exception", wenn diese Codezeile ausgeführt wird, dann würde unser Programm abstürzen. Deswegen entfernen wir sie. In unsere Eventhandler Methode können wir nun bestimmen, was passieren soll wenn das Event ausgelöst wird.

User Input

Bisher sind unsere Programme mehr Video als Spiel, da sie immer gleich ablaufen und man nicht mit ihnen Interagieren kann. Das wollen wir jetzt ändern. Dazu fügen wir unseren Programmen User Input hinzu. Der Benutzer/Spieler kann also Einfluss auf den Verlauf des Programmes nehmen.



Der klassische User Input sind Maus und Tastatur. Deswegen werden wir uns auf diese Beschränken. Um mit User Input zu arbeiten, gibt es 2 Möglichkeiten in SFML: Über Abfrage und über Events.

User Input über Abfrage

User Input über Abfrage ist meistens einfacher als über Events, hat aber auch seine Nachteile. Wir können in SFML über die Keyboard und Maus Klassen die Zustände der Tasten bzw. der Mausposition abfragen, und demnach handeln. Dazu müssen wir in unserer **loop()-Methode** über if-Klammern die einzelnen Zustände abfragen.

Folgende Methoden sind zur Abfrage möglich:


```
Keyboard.IsKeyPressed(Keyboard.Key.<Hier Taste Einfügen>);
```

```
Mouse.IsButtonPressed(Mouse.Button.<Hier Taste Einfügen>);
```

Die beiden Methoden geben "true" zurück wenn die entsprechende Taste gedrückt ist, und false, wenn sie nicht gedrückt ist.

```
Mouse.GetPosition(game.gameWindow);
```

Gibt einen Vektor2f zurück, mit den Mauskoordinaten



Wichtig! Da sonst die Mauskoordinaten auf dem Bildschirm und nicht des Fensters genommen werden!

Wenn wir also wollen, dass beim Drücken der Taste "G" ein Objekt sich nach rechts bewegt, würde das so aussehen:

```
if (Keyboard.IsKeyPressed(Keyboard.Key.G))  
{  
  
    Box.Position += new Vector2f(1, 0);  
  
}
```

Beachte! User Input durch Abfrage hat den Nachteil, dass es pro Frame 1 Mal abfragt. Wenn man aber eine einmalige Veränderung bei einem Tastendruck möchte, braucht man Events.

User Input mit Events

In C# kann man EventHandler an Events binden. SFML bietet dazu mehrere Events an, die Member der RenderWindow Klasse ist, auf die wir mit game.gameWindow zugreifen. Die wichtigsten Events sind hier aufgelistet:

game.gameWindow.KeyPressed

// Wird beim Drücken irgendeiner Taste der Tastatur ausgelöst.

(Achtung: Wenn man eine Taste länger gedrückt hält, kommt es zur KeyRepetition, dann wird die Taste schnell hintereinander gedrückt. Man kann dies mit "game.gameWindow.setKeyRepeatEnabled(false)" ausschalten)

game.gameWindow.KeyRelease

// Wird beim loslassen irgendeiner Taste der Tastatur ausgelöst.

game.gameWindow.MouseMoved

// Wird ausgelöst wenn die Maus bewegt wird

game.gameWindow.MouseButtonPressed

// Wird ausgelöst wenn eine Maustaste gedrückt wird

game.gameWindow.MouseButtonReleased

// Wird ausgelöst wenn eine Maustaste losgelassen wird

Wenn wir also auf einen Tastendruck reagieren wollen, machen wir das so: (Für Details zu Events und Eventhandlern, siehe vorheriges Kapitel)

```
game.gameWindow.KeyPressed += new EventHandler<KeyEventArgs>(OnKeyPress);  
  
}  
  
void OnKeyPress(object sender, KeyEventArgs e)  
{  
  
    Box.Size *= 2;  
    Box.Origin = Box.Size / 2;  
  
}
```

Jetzt wird, wann immer wir irgendeine Taste auf der Tastatur drücken, die Größe von Box verdoppelt. Wenn wir das ganze jetzt auf eine Taste beschränken wollen, müssen wir den Parameter e benutzen.

```
void OnKeyPress(object sender, KeyEventArgs e)  
{  
    if (e.Code == Keyboard.Key.G)  
    {  
        Box.Size *= 2;  
        Box.Origin = Box.Size / 2;  
    }  
  
}
```

Der Member "Code" von e, enthält den KeyCode der Taste, die gedrückt wurde. Wir müssen sie also mit einem der "Keyboard.Key" Konstansten vergleichen, um herauszufinden, welche Taste gedrückt wurde. Um genaueres über einzelne Events, deren Parameter und deren Inhalte zu erfahren, kann man sich in der Dokumentation hier alles anschauen:

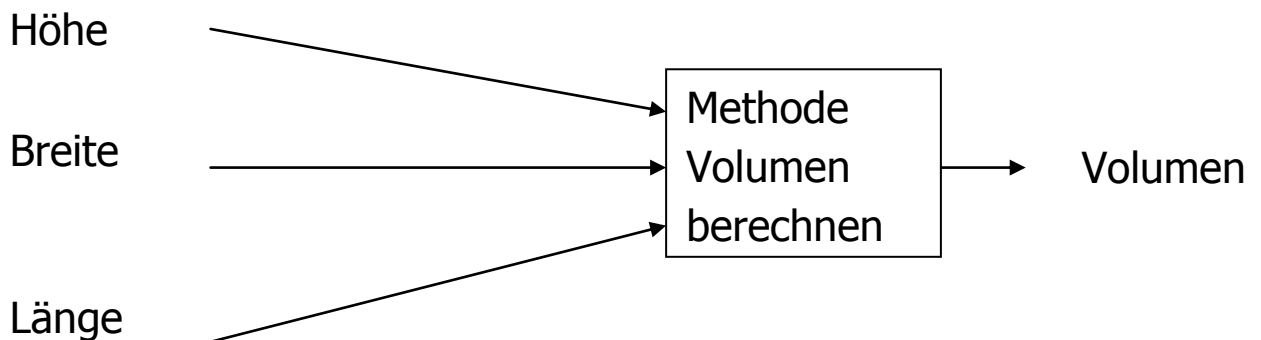
http://www.sfml-dev.org/documentation/2.3.2/classsf_1_1Event.php

Methoden

In C# schreibt man den Code in sogenannte Klassen, eine Codedatei entspricht meistens einer Klasse. Wir arbeiten in "MainScreen.cs", was die Datei zur Klasse MainScreen ist. Eine Klasse besteht aus Methoden und hat zusätzlich noch Variablen außerhalb von Methoden. Eine **Methode** ist ein Stück Quellcode, der **Daten bekommt(Parameter)** und **Daten zurückgibt(Rückgabewert)**.

Parameter

Rückgabewert



Als Code würde die Methoden VolumenBerechnen so aussehen:

```
public double volumenBerechnen(double hoehe, double breite, double laenge)
{
    double volumen = hoehe * breite * laenge;

    return volumen;
}
```

Gehen wir die Methode von vorne nach hinten durch:

"public" steht für die Sichtbarkeit, es bedeutet jeder kann die Methode benutzen(wird später genauer erklärt)

Das erste "double" ist der Typ des Rückgabewertes, hier ein Double da das Volumen ein Double ist(Gibt die Methode nichts zurück, ist der Rückgabotyp "void").

volumenBerechnen ist der Bezeichner der Methode, er kann frei gewählt werden

Danach folgt eine Klammer mit 3 double Werten höhe, breite, länge, dies sind die Informationen die die Methode bekommt, die Parameter.

In den geschweiften Klammern("{}") steht nun der Quellcode der Methode. Hier wird das Volumen berechnet mithilfe der Parameter. Das "return volumen", sagt das die Variable "volumen", die den ausgerechneten Wert enthält der Rückgabewert sein soll.

Arrays

Wenn man in C# große Datenmengen speichern möchte, z.b. 500 int-Werte, dann wäre es ein viel zu großer Aufwand dafür 500 Variablen zu schreiben und mit diesen zu arbeiten. Stattdessen gibt es sogenannte Arrays. Arrays sind spezielle Variablen, die viele Variablen vom gleichen Datentyp zusammenfassen. Beispiel:

```
// 500 Int variablen einzeln
int x0 = 32;
int x1 = 24;
int x2 = 245;
int x3 = 23;
int x4 = 423;
int x5 = 34;
int x6 = 100;
//[...]
int x499 = 1000;
```

Index	Wert
0	0
1	0
2	0
3	0
...	...
499	0

```
// 500 Int variablen in einem Array
int[] X_Werte = new int[500];
```

Da in einem Array die einzelnen Variablen keinen Namen besitzen, greift man auf sie mit dem sogenannten Index zu:

```
// Normale Variable Setzen
x1 = 5;

// Array Element mit Index 1 setzen
X_Werte[1] = 5;
```

In einem Array werden alle Elemente durchnummeriert, man startet immer bei der 0 und endet mit der **Anzahl der Elemente -1**, es ist ein häufiger Fehler, dass man auf eine Index zugreift, der außerhalb dieser Grenzen liegt, dies führt zu einer "**IndexOutOfRangeException**".

Oft will man eine bestimmte Operation mit allen Elementen eines Arrays durchführen. Dazu benutzt man **Schleifen!**

Schleifen

Schleifen sind, ähnlich wie If-Blöcke, ein Element in C#, die eine bestimmte Bedingung haben. Sie sind dazu da, bestimmten Code zu wiederholen, bis bestimmte Bedingungen eintreffen.

Die einfachste Schleife ist die **while()-schleife**. Eine while Schleife wiederholt ihren {}- Block **solange** die Bedingung in ihren ()-Klammern gilt:

```
int x = 0;
while (x < 100)
{
    x += 1;
}
```

Diese Schleife wiederholt **solange** die "x += 1" Operation, bis X nichtmehr kleiner als 100 ist, also bis X == 100 ist. Nach dieser Schleife ist X = 100.

Schleifen sind besonders nützlich, wenn man mit Arrays arbeitet. Will man alle Elemente eines Int-Arrays auf einen Wert setzen, geht das so:

```
int[] X_Werte = new int[500];

for (int i = 0; i < X_Werte.Length; i += 1)
{
    X_Werte[i] = i;
}
```

Index	Wert
0	0
1	1
2	2
3	3
...	...
499	499

Hier wird eine **for()-Schleife** benutzt, sie ist speziell für den Gebrauch mit Arrays nützlich. Sie hat in ihren ()-Klammern immer 3 Argumente mit ; getrennt. Zu erst wird eine **Iterationsvariable** und deren Startwert bestimmt, hier i. (i, j, n, m sind sehr oft dafür gebrauchte Namen).

Dann wird wie bei der While Schleife eine Bedingung festgelegt. Hier muss i immer kleiner sein als die Größe des X_Werte Arrays, also immer kleiner als 500.

Das dritte ist ein Befehl der nach jedem durchlauf der Schleife ausgeführt wird. Hier wird also mit jedem Durchlauf i um 1 erhöht.

Zusammen mit der Bedingung heißt das, dass die Schleife genau 500 mal durchläuft, und i dabei am Anfang 0 ist und bis 499 hochzählt.

Weil das auch genau die Grenzen für den Array-Index sind, können wir i als Index angeben und somit auf alle 500 Elemente des Array zugreifen. Wir setzen sie auf i, also enthält das Array jetzt alle Zahlen von 0 bis 499.

Ein weiteres Beispiel:

```
int[] X_Werte = new int[100];  
  
for (int i = 0; i < X_Werte.Length; i += 1)  
{  
    X_Werte[i] = 100 - i*2;  
}
```

Index	Wert
0	$100 - 0*2 = 100$
1	$100 - 1*2 = 98$
2	$100 - 2*2 = 96$
3	$100 - 3*2 = 94$
...	...
99	$100 - 99*2 = -98$

Listen

Listen sind spezielle Formen von Arrays. Wenn man ein Array erstellt, muss man die Größe dieses Arrays angeben. Man kann das Array danach nichtmehr vergrößern oder verkleinern. Listen funktionieren anders. Sie passen ihre Größe immer an die Anzahl der Elemente an. Listen kann man fast genauso benutzen wie Arrays, nur bei der Erstellung sieht es so aus:

```
// Int Array:  
int[] X_Werte = new int[500];  
  
// Int Liste:  
List<int> Y_Werte = new List<int>();
```

Nach der Erstellung hat die Liste immer die Größe 0. Erst durch hinzufügen mit "Add()" steigt die Größe.

Wenn man nun ein Element der Liste hinzufügen möchte, schreibt man:

```
// Füge 5 hinzu, Größe steigt von 0 auf 1  
Y_Werte.Add(5);
```

Hierbei ist zu beachten, dass die Größe natürlich steigt! Der Index des neuen Elements ist dann entsprechend der Höchste, also 0.

Beim Entfernen gibt es nun 2 Möglichkeiten.

Einmal kann man ein Objekt mithilfe des Index entfernen, mit:

```
// Entferne Element an Stelle 0, Größe sinkt auf 0.  
Y_Werte.RemoveAt(0);
```

Hierbei sinkt die Größe dementsprechend, und **alle Elemente mit höherem Index sinken um 1**. Sie "fallen" praktisch nach wenn man Elemente unter ihnen entfernt.

Die Zweite Möglichkeit ist, ein Objekt direkt zu Entfernen, indem man es angibt.

```
// Entferne die erste 5, die in der Liste vorkommt!  
Y_Werte.Remove(5);
```


Wenn in diesem Fall die 5 mehrmals vorkommt, wird die 5 mit dem kleinsten Index entfernt. Auch hier fallen die höheren Elemente einen Index nach unten.

Wenn man jetzt mit For-Schleifen die Liste verändern, müssen wir aufpassen:

Einerseits gibt es kein `Y_Werte.length`, sondern bei Listen heißt es `Count`.

Außerdem, da die Größe von der Liste sich verändern kann, ist es ganz wichtig in der For Schleife herunterzuzählen!

```
for (int i = 0; i < Y_Werte.Count; i += 1)
{
    Y_Werte.RemoveAt(i);
}
```

Man würde erwarten, dass diese Schleife alle Elemente der Liste entfernt, aber sie entfernt nur die Hälfte, weil durch das "Nachrücken" die Indexe sich verändern.

```
for (int i = Y_Werte.Count - 1; i >= 0; i -= 1)
{
    Y_Werte.RemoveAt(i);
}
```

Diese Schleife zählt runter und entfernt alle Elemente.

Objekte hinzufügen & entfernen

Wenn wir Objekte unserem Spiel hinzufügen, machen wir das bisher indem wir sie im Code direkt als Variable deklarieren. Sobald es aber nötig ist, ein Element während der "Laufzeit" unseres Programms zu hinzuzufügen oder zu entfernen, klappt das so nichtmehr. Deswegen nutzen wir an dieser Stelle eine Liste.

```
// Liste mit Rectangle Shapes
List<RectangleShape> shapes = new List<RectangleShape>();
```

Diese Liste kann nun eine beliebige Anzahl an Shapes speichern und sie während der Laufzeit hinzufügen oder entfernen.

Um also auf "knopfdruck" ein neues Rectangle hinzuzufügen, schreiben wir:

```
void OnKeyPress(object sender, KeyEventArgs e)
{
    // Wenn C gedrückt wird:
    if (e.Code == Keyboard.Key.C)
    {
        // Füge ein neues RectangleShape hinzu
        shapes.Add(new RectangleShape());
    }
}
```

(Hier wurde ein Eventhandler verwendet)

Allerdings haben wir das Problem, dass wir so nicht die Eigenschaften des RectangleShapes verändern können. Deswegen kann man eine kleine Hilfsvariable benutzen, um das Shape zwischenspeichern.

```
// Wenn C gedrückt wird:
if (e.Code == Keyboard.Key.C)
{
    // Erstelle ein neues Shape und setze die Eigenschaften
    RectangleShape nShape = new RectangleShape();
    nShape.Size = new Vector2f(50, 50);
    nShape.FillColor = Color.Green;
    nShape.Position = new Vector2f(400, 300);

    // Füge nShape hinzu
    shapes.Add(nShape);
}
```

Natürlich darf man nicht vergessen, die Shapes mit der draw() Methode zu zeichnen. Wenn wir eine Liste benutzen, kann man das über eine for-Schleife tun:

```
for (int i = shapes.Count - 1; i >= 0; i--)
{
    draw(shapes[i]);
}
```

Jetzt werden alle Elemente der Liste nacheinander gezeichnet. Wir sehen allerdings maximal 1, weil sie sich alle überdecken. Deswegen lassen wir nach unten bewegen. Um also eine Bewegung für alle Listen-Elemente zu machen, benutzen wir wieder eine For-Schleife:

```
for (int i = shapes.Count - 1; i >= 0; i--)
{
    shapes[i].Position += new Vector2f(0,5);
}
```

Jetzt können wir noch programmieren, dass man das neuste Element löscht, wenn man X drückt:

```
// Wenn X gedrückt wird:
if (e.Code == Keyboard.Key.X)
{
    if (shapes.Count > 0)
    {
        shapes.RemoveAt(shapes.Count - 1);
    }
}
```

Aber Vorsicht! Wenn man löscht, sollte man immer prüfen, ob es überhaupt ein Element gibt, sonst crasht das Programm!

Kollisionsabfrage mit Hitbox

Um in Spielen, in denen sich Objekte bewegen, festzustellen ob zwei Objekte sich berühren(kollidieren), braucht man die Kollisionsabfrage. Im 2D Bereich gibt es verschiedene Möglichkeiten, aber die simpelste und meist genutzte ist das Hitbox-Verfahren.

Das Hitbox-Verfahren umschließt ein GameObjekt mit einem oder mehreren Hitboxen. Diese Stellen die Region da, in der das GameObjekt ein anderes Treffen kann.

Umso mehr Hitboxen man verwendet, desto genauer aber auch rechenintensiver wird die Abfrage.

Wichtig ist hierbei zu beachten, dass Hitboxen oft mehr abdecken, als man möchte und man sie nicht drehen kann. Dies kann manchmal zu scheinbar unsichtbaren Kollisionen führen(siehe Abbildung 2).

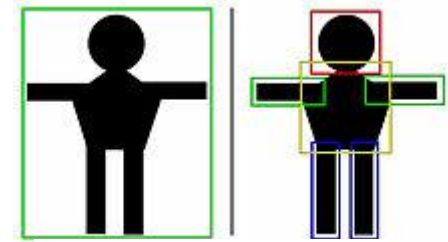


Abbildung 1 - Hitboxen

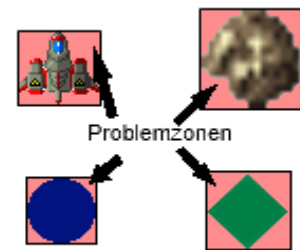


Abbildung 2 - Probleme bei Hitboxen

Abfrage der Kollision

Um zu Überprüfen ob sich zwei Hitboxen überschneiden, gibt es ein simples Verfahren. Hierbei ist es tatsächlich effizienter zu gucken, ob sie sich nicht überschneiden. Wenn sich zwei Vierecke nicht überschneiden, dann gilt eine der folgenden Aussagen:

- Der X-Wert der **rechten Seite** von A ist kleiner als der X-Wert der **linken Seite** von B.
- Der X-Wert der **linken Seite** von A ist größer als der X-Wert der **rechten Seite** von B.
- Der Y-Wert der **oberen Kante** von A ist kleiner als der Y-Wert der **unteren Kante** von B.
- Der Y-Wert der **unteren Kante** von A ist größer als der Y-Wert der **oberen Kante** von B.

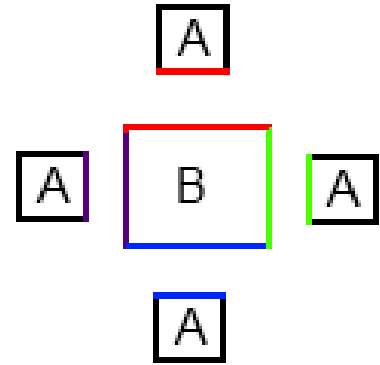


Abbildung 3 - Hitbox-Kollisionsabfrage

Als Code könnte man diese Abfrage so verwirklichen:

```
if (BoxA.Position.X + BoxA.Size.X < BoxB.Position.X ||  
    BoxA.Position.X > BoxB.Position.X + BoxB.Size.X ||  
    BoxA.Position.Y > BoxB.Position.Y + BoxB.Size.Y ||  
    BoxA.Position.Y + BoxA.Size.Y < BoxB.Position.Y)  
{  
    // KEINE KOLLISION!  
}  
else  
{  
    // KOLLISION  
}
```

Kollision mit Shapes

In SFML gibt es bereits eingebaute Hitboxen für jedes Shape. Auf diese können wir mit **Shape.getGlobalBounds()** zurückgreifen.

Um zu prüfen, ob sich 2 Hitboxen überschneiden nutzen wir die **"Intersects(shape2.getGlobalBounds())"** Methode der Hitbox.

Hier ein Beispiel:

```
if (BoxA.GetGlobalBounds().Intersects( BoxB.GetGlobalBounds() ))
{
    // Wenn die Hitboxen sich überlappen
}
else
{
    // Wenn sie sich nicht überlappen
}
```

Jetzt können wir programmieren, was passieren soll, wenn sich die beiden Hitboxen überlappen.

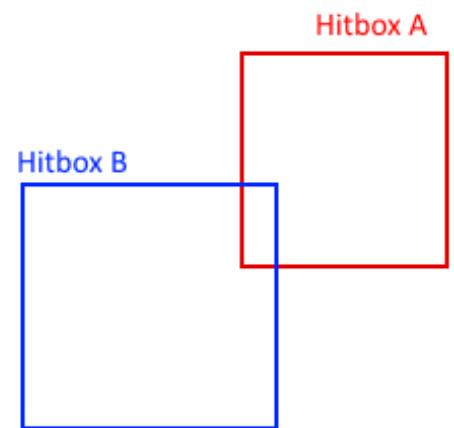


Abbildung 4: Zwei Hitboxen überlappen sich

Alternativ können wir auch prüfen, ob sich ein Punkt innerhalb einer Hitbox befindet, dazu nutzen wir die **"Contains(x,y)"** Methode.

Beispiel:

```
if (BoxA.GetGlobalBounds().Contains(200, 300))
{
    // Hitbox befindet sich über Punkt 200,300
}
else {
    // Hitbox befindet sich nicht über Punkt 200,300
}
```



Abbildung 5: Hitbox ist über Punkt(x,y)

Bewegungskontrolle

Eine weitere Möglichkeit Hitboxen zu benutzen ist, die Bewegung eines Objektes einzuschränken, z.b. durch eine Wand, die ein weiteres Objekt darstellt.

```
RectangleShape Player;
RectangleShape Wall;

Player = new RectangleShape();
Player.Size = new Vector2f(50, 50);
Player.Position = new Vector2f(200, 400);
Player.FillColor = Color.Green;
Player.OutlineColor = Color.Red;
Player.OutlineThickness = 1;

Wall = new RectangleShape();
Wall.Size = new Vector2f(10, 400);
Wall.Position = new Vector2f(400, 200);
Wall.FillColor = Color.Blue;
Wall.OutlineColor = Color.Red;
Wall.OutlineThickness = 1;
```

Wenn wir nun unseren Player wie gewohnt mit **isKeyPressed** bewegen:

```
if (Keyboard.IsKeyPressed(Keyboard.Key.D))
{
    Player.Position += new Vector2f(3, 0);
}
```

Dann kann er einfach durch unser "Wall" Objekt. Wir könnten jetzt auf Kollision prüfen:

```
if (Player.GetGlobalBounds().Intersects( Wall.GetGlobalBounds() ))
{
}
else
{
    if (Keyboard.IsKeyPressed(Keyboard.Key.D))
    {
        Player.Position += new Vector2f(3, 0);
    }
}
```

Und so nur Bewegung zulassen, wenn sie sich nicht überschneiden.

Leider führt das dazu, dass sobald eine Kollision entsteht, unser Shape am Wall "festklebt". Deswegen müssen wir prüfen ob eine Kollision nach der Bewegung entstanden ist, und wenn ja, die Bewegung rückgängig machen. Man beachte hierbei das "-=" !

```
if (Keyboard.IsKeyPressed(Keyboard.Key.D))
{
    // Bewege Player
    Player.Position += new Vector2f(3, 0);

    if (Player.GetGlobalBounds().Intersects(Wall.GetGlobalBounds()))
    {
        // Kollision, bewege Player zurück
        Player.Position -= new Vector2f(3, 0);
    }
}
```

Jetzt wird eine ungültige Bewegung einfach rückgängig gemacht.

Jetzt kann der Player den "Wall" nichtmehr passieren.

Allerdings gibt es noch Sonderfälle zu beachten. Wenn die Geschwindigkeit des Players sehr hoch wird, dann kann es sein, dass er den "Wall" einfach überspringt und keine Kollision auslöst.. Es gäbe zwei Möglichkeiten um diese Fehler zu verhindern:

- Maximale Geschwindigkeit und mindest "Wanddicke" für Objekte(begrenzt die Möglichkeiten für das Spiel)
- Die Kollisionsabfrage erweitern, so dass die Fehler nichtmehr auftreten (kostet mehr Leistung als normale Abfrage)

Externe Ressourcen

Um in unserem Programm Musik abzuspielen, Text zu schreiben oder Bilder anzuzeigen müssen wir externe Ressourcen nutzen.

Der Ordner namens "Assets" ist dafür vorgesehen, hier sollten alle Ressourcen gelagert werden. Standardmäßig ist bereits die Datei "arial.ttf" in dem Ordner. Sie ist nötig um Text auszugeben. Alle Ressourcen sollten also in den Assets Ordner.

Außerdem ist es wichtig, dass man in den Eigenschaften von jeder Ressourcendatei "Kopieren wenn neuer" einstellt. Ansonsten kann sie nie geladen werden!

Pfade

Um im Code auf diese Dateien zu verweisen, nutzen wir die Pfade. Wenn man eine Datei aus dem Projektmappenexplorer in das Codefenster zieht, wird automatisch der absolute Pfad angegeben.

Beispiel:

```
C:\Users\User\AppData\Local\Temporary Projects\Spiel\assets\arial.ttf
```

Dieser Pfad ist "**absolut**" weil er bei dem Laufwerk "C:\\" startet. Diese Pfade geben den Ort der Datei auf dem Computer genau an. Das ist aber schlecht, da wir unser Programm auch von anderen Orten ausführen wollen, z. b. auf einem anderen Computer, als da wo wir das Programm programmiert haben.

Deswegen brauchen wir einen "**relativen**" Pfad. Dieser startet immer da, wo das Programm ausgeführt wird. Da der Ordner Assets immer mit kopiert wird, und damit immer neben dem Programm liegt, können wir ganz einfach relativ darauf verweisen.

Mit

```
assets\arial.ttf
```

können wir also immer auf die TTF Datei verweisen, egal wo sich unser Programm befindet, solange der "Assets" Ordner dabei ist.

Zuletzt müssen wir noch aus jeden "\" ein "/" machen, da in C# "\" eine besondere Bedeutung haben.

Also ist es jetzt:

```
assets/arial.ttf
```

Externe Ressourcen kann man gut finden unter:
opengameart.org

Mit Lizenzen:

"CC0" bzw. "Open Domain"

Texturen & Sprites

Will man in SFML ein Bild einfügen, so muss man Sprites und Texturen dafür nutzen. Die "Sprite" Klasse lässt sich fast genauso benutzen wie ein "RectangleShape". Wir können also in Mainscreen ein Sprite erstellen.

```
public Sprite graphic;
```

Wenn wir unsere Sprite initialisieren mit "new", müssen wir allerdings eine Textur angeben. Die Textur stellt hierbei die Bilddatei dar, die wir benutzen. Hier benutzen wir die "duck.png".

Wir müssen also erst unser Bild in eine Textur laden und dann können wir das Sprite mit der Textur erstellen.

```
Texture duck = new Texture("assets/duck.png");  
graphic = new Sprite(duck);
```

Hierbei sollte man folgendes beachten:

Eine Textur belegt viel Speicher und sie zu erstellen dauert lange. Wenn möglich sollte man **nie mehrere Texturen mit demselben Bild erstellen**, da es Platzverschwendung ist! Sprites dagegen verbrauchen nicht viel Platz oder Leistung.

Jetzt sollte unsere Entity die Ente als Bild haben.

Eigenschaften wie Fillcolor oder Size sind nichtmehr vorhanden, da Sprite automatisch die Größe seiner Textur annimmt. Möchte man das Sprite trotzdem vergrößern/verkleinern, sollte man die Eigenschaft "Scale" benutzen.

Musik & Sound

Um in SFML Musik oder Sounds abzuspielen, braucht man dementsprechend ein "Music" oder "Sound" Objekt.

Äquivalent zur Texture/Sprite gibt es hier den "Soundbuffer". Wir müssen unseren Sound also erst in den Soundbuffer laden, und diesen dann unseren Sounds zuweisen. Bei dem "Musik" Objekt brauchen wir diesen allerdings nicht. Auch hier gilt, niemals mehrere Music/Soundbuffer Objekte, die dasselbe beinhalten!

Obwohl Musik auch als langer Sound angesehen werden könnte, gibt es in SFML eine Extra Klasse für Musik, sie ist besser darauf ausgelegt lange und in Schleife laufende Audiospuren auszugeben.

Als Beispiel Benutzen wir hier die "happyTune.ogg" als Musik und die "explosion.wav" als Sound.

Das Abspielen von einem Musik und Sound Objekt ist fast identisch.

```
SoundBuffer explosion = new SoundBuffer("assets/explosion.wav");  
Sound boom = new Sound(explosion);  
boom.Play();
```

```
Music happyTune;  
happyTune = new Music("assets/happyTune.ogg");  
happyTune.Play();
```

Bemerkung:

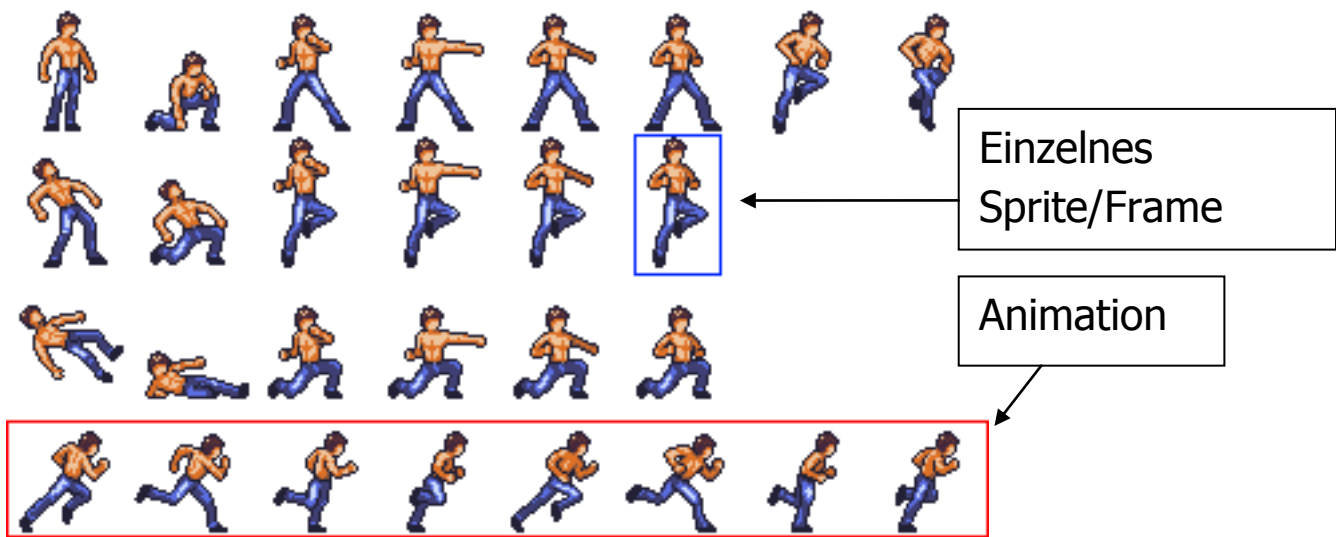
Bei Sounddateien gibt es viele Formate, die leider nicht unterstützt werden (z.b. mp3). Generell ist das "ogg" Format zu empfehlen. Online gibt es viele Converter die alle möglichen Formate umwandeln können.

Von allen Assets nimmt Audio, insbesondere lange Musik, am meisten Speicherplatz ein. Man sollte hierauf ein Auge haben. Man kann die kbps(Kilobyte per second) von Audiodateien kleiner wählen(beim konvertieren) und damit die Dateigröße verkleinern, jedoch auf Kosten der Soundqualität.

Spritesheets

Wir können mit dem Sprite Objekt jetzt Bilder anstatt von Rechtecken oder Kreisen anzeigen. Als nächstes wollen wir lernen, wie man animierte Objekte erstellt. Dazu brauch man erstmal ein **Spritesheet**. Ein Spritesheet ist eine Bilddatei, die nicht nur aus einem, sondern aus vielen aneinandergehängten Bildern besteht, die zusammen eine oder mehrere Animationen bilden.

Hier ist ein typisches Spritesheet:



Man spricht von einer Animation als mehrere Sprites, die hintereinander angezeigt werden können damit eine sinnvolle Bewegung erkennbar ist.

Da die Animationsgeschwindigkeit je nach Animation unterschiedlich sein kann, ist es meistens nötig durch Timer/Counter die Wechsel der einzelnen Sprites zeitlich anzupassen.

Timer/Counter

Um in SFML bestimmte Dinge zeitlich abzustimmen, kann man sogenannte Counter benutzen. Diese zählen einfach jeden Frame hoch und setzen sich zurück wenn sie einen maximalen Wert erreichen. Beim Zurücksetzen wird auch gleichzeitig das Ereignis ausgelöst, dass man Zeitlich abstimmen möchte.

Typischer Counter Code(in Loop):

```
int counter;

// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
public override void setup()
{
    counter = 0;
}

// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
public override void loop()
{
    counter++;
    if (counter >= 30)
    {
        // Dieser Code wird nur alle 30 Frames ausgeführt, also 2
        // mal pro Sekunde

        counter = 0;
    }
}
```

The diagram includes two text boxes with arrows pointing to the code. The first box, labeled 'Anzahl der Frames für den Counter', has an arrow pointing to the line `if (counter >= 30)`. The second box, labeled 'Hier Code einfügen', has an arrow pointing to the line `counter = 0;` inside the if-block.

Animationen

SFML bietet selber keine Objekte, um Animation zu vereinfachen. Deswegen arbeiten wir mit einer Liste. Als Spritesheet nutzen wir die "spriteSheet46x50.png".

Als erstes müssen wir wie im vorherigen Kapitel die Textur erst laden.

Wir erstellen ein Texture Objekt in Mainscreen:

```
public Texture spritesheet;

// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
public override void setup()
{
    spritesheet = new Texture("assets/spriteSheet46x50.png");
}
```

Dafür muss die "spriteSheet46x50.png" natürlich im assets Ordner liegen und die "kopieren wenn neuer" Eigenschaft eingestellt sein.

Dann können wir in unserer Entity, schon mal eine Sprite Liste erstellen, in die alle Sprites unserer Animation geladen werden.

```
public List<Sprite> animation;
public Sprite graphic;

MainScreen screen;

// Konstruktor
public ExampleEntity(MainScreen parentScreen)
{
    screen = parentScreen;
    animation = new List<Sprite>();
}
```

Jetzt müssen wir die Liste füllen, testweise erstellen wir erstmal ein Sprite und packen es in die Liste. Außerdem sagen wir, dass unsere Graphic das erste Sprite der Liste sein soll.

```
Sprite nSprite = new Sprite(screen.spritesheet);

animation.Add(nSprite);

graphic = animation[0];
```



Jetzt haben wir das gesamte SpriteSheet als Sprite, wir wollen aber nur einen Teil davon. Deswegen müssen wir beim Erstellen des Sprites neben der Textur, noch ein **IntRect** angeben.

Statt:

```
Sprite nSprite = new Sprite(screen.spritesheet);
```

jetzt:

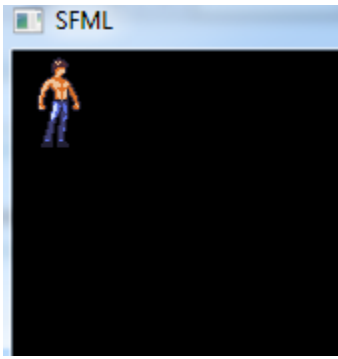
```
Sprite nSprite = new Sprite(screen.spritesheet, new IntRect(0,0,0,0));
```

Die vier Zahlen des IntRects stehen für X-Position, Y-Position, Breite, Höhe in dieser Reihenfolge. Die Breite und Höhe des Teilbildes ist im Dateinamen des SpriteSheets angegeben, 46x50.

(Wenn das nicht der Fall ist, muss man nachsehen/nachzählen)

```
Sprite nSprite = new Sprite(screen.spritesheet, new IntRect(0,0,46,50));
```

Da als X/Y 0 angegeben ist, nehmen wir den Teil der Spritesheet ganz oben links. Das Ergebnis:



Sieht schonmal gut aus. Jetzt wollen wir mehrere Sprites in die Liste laden, und zwar immer andere. Dazu ändern wir immer die X/Y Position. Am einfachsten geht dies durch eine For Schleife:

```
for (int i = 0; i < 8; i++)
{
    Sprite nSprite = new Sprite(screen.spritesheet, new IntRect(46*i,0,46,50));
    animation.Add(nSprite);
}
```

Das "46*i" sorgt dafür, dass wir jeden Durchlauf ein Bild 46 Pixel weiter rechts nehmen(die breite eines bildes).

Jetzt müssen wir nurnoch die Graphic nacheinander die verschiedenen Sprites anzeigen lassen.

Dafür brauchen wir einen typischen Counter, eine Int-Variable.

```
public List<Sprite> animation;
public Sprite graphic;

int currentFrame = 0;

MainScreen screen;
```

Diesen lassen wir in der Loop hochzählen, aber sobald er die 8 erreicht, soll er wieder auf 0 gesetzt werden.(Die Anzahl sollte immer gleich sein zu der For schleife im Konstruktor!)

```
// Loop Methode
public void loop()
{
    currentFrame++;
    if (currentFrame >= 8)
    {
        currentFrame = 0;
    }
}
```

Jetzt lassen wir die graphic in der Loop immer auf die aktuelle Animationssprite zeigen:

```
// Loop Methode
public void loop()
{
    currentFrame++;
    if (currentFrame >= 8)
    {
        currentFrame = 0;
    }

    graphic = animation[currentFrame];
}
```

Da wir mit 60 FPS arbeiten, ist die Animationsgeschwindigkeit sehr hoch. Um diese besser zu kontrollieren, brauchen wir noch einen Counter.

```
public List<Sprite> animation;
public Sprite graphic;

int currentFrame = 0;
int frameCounter = 0;

MainScreen screen;
```

Diesen lassen wir wieder genauso hochzählen in der loop, allerdings diesmal bis 5.

```
// Loop Methode
public void loop()
{
    currentFrame++;
    if (currentFrame >= 8)
    {
        currentFrame = 0;
    }

    frameCounter++;
    if (frameCounter >= 5)
    {
        frameCounter = 0;
    }

    graphic = animation[currentFrame];
}
```

Wenn wir jetzt "currentFrame++" in die if-Klammer des Frame Counters tun, können wir die Animation um den Faktor 5 verlangsamen.

```
// Loop Methode
public void loop()
{
    frameCounter++;
    if (frameCounter >= 5)
    {
        currentFrame++; // <--- hier einfügen
        frameCounter = 0;
    }

    //currentFrame++; <-- hier entfernen
    if (currentFrame >= 8)
    {
        currentFrame = 0;
    }

    graphic = animation[currentFrame];
}
```

Jetzt haben wir eine schöne Animation.

Achtung! Wenn man Position/Rotation/Scale etc. ändern möchte:

Sollte man diese Werte wie hier übernehmen, bevor man das Sprite wechselt:

```
// Loop Methode
public void loop()
{
    frameCounter++;
    if (frameCounter >= 5)
    {
        currentFrame++;
        frameCounter = 0;
    }

    if (currentFrame >= 8)
    {
        currentFrame = 0;
    }

    animation[currentFrame].Position = graphic.Position;
    animation[currentFrame].Rotation = graphic.Rotation;
    animation[currentFrame].Scale = graphic.Scale;
    graphic = animation[currentFrame];
}
```

Da sonst nur einzelne Frames geändert werden, aber nicht die ganze Animation!

Klassen & Objekte

C# ist eine Objekt-Orientierte Programmiersprache. Deswegen ist der Code eines C# Programms immer in Klassen angeordnet. Bisher haben wir nur in der mainScreen Klasse gearbeitet. Um aber kompliziertere und größere Programme zu schreiben, sollten wir anfangen Objekt-Orientiert zu arbeiten.

Im Projektmappen-Explorer kann man durch:

Rechtsklick auf einen Ordner -> Hinzufügen -> Neues Element -> Klasse eine neue C# Klassendatei in diesem Ordner erzeugen.

Jede neue Klasse hat folgenden Standardaufbau:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace NameSpaceName
{
    class KlassenName
    {
    }
}
```

Hinter "class" steht der Klassenname.

Nach "namespace" kommt immer der NameSpace, dieser ist quasi der Unterordner in der sich unsere Datei befindet. Wenn eine Datei in andere Ordner verschoben, sollte man den NameSpace anpassen.

Die "using" Anweisungen geben an, auf welche NameSpaces die Klasse zugreifen kann. Will man auf Klassen zugreifen, die nicht den selben NameSpace haben, muss man sie hier angeben. (Blöderweise ist die Anweisung "using System.Linq;" standardmäßig drin, erzeugt aber einen Error, weswegen man sie entfernen muss.)

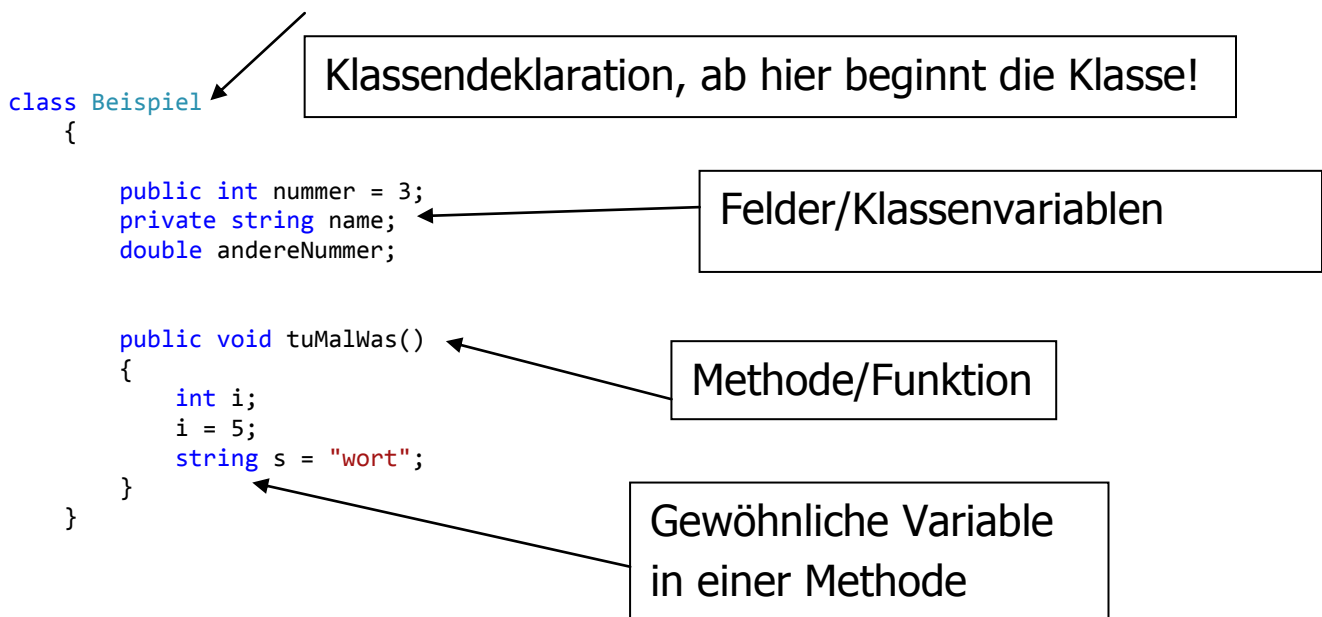
Aus **Klassen** werden **Objekte** erzeugt, mit denen man arbeiten kann. Wir kennen schon verschiedene Klassen.

RectangleShape ist zum Beispiel eine Klasse.

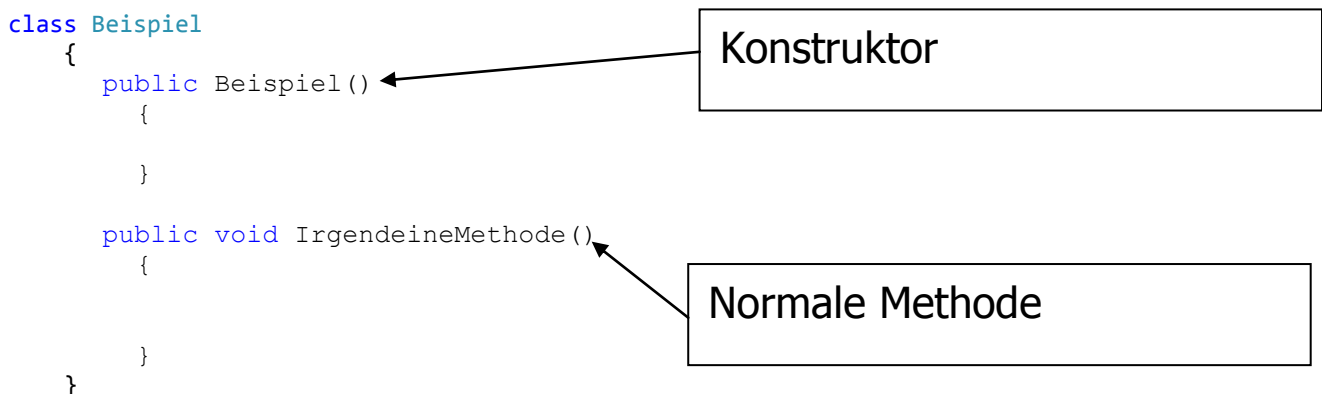
Immer wenn wir "new RectangleShape()" benutzen, erzeugen wir ein **Objekt** aus dieser **Klasse**.

Jedes Objekt besteht hauptsächlich aus 2 Dingen. Seinen **Feldern** und **Methoden**.

Felder kann man auch als "Klassenvariablen" sehen. Es sind Variablen die in dem gesamten Objekt benutzt werden können, anders als die normalen Variablen in Methoden. **Felder** erkennt man daran, dass die **außerhalb von Methoden in ihrer Klasse deklariert werden**. Hier ein kleines Beispiel für Felder.



Klassen/Objekte können außerdem eine besondere Methode enthalten. Den **Konstruktor**. Den Konstruktor erkennt man daran, dass er **genauso heißt wie die Klasse**. Außerdem hat er **keinen Rückgabebetyp**(kein "void")



Der Konstruktor ist besonders, weil er nicht wie eine normale Methode ausgeführt werden kann, sondern immer nach Erstellung des Objekts ausgeführt wird. Er ist also die erste Methode die das Objekt ausführt.

Man benutzt den Konstruktor üblicherweise um alle Startwerte zu setzen. Damit ist er Vergleichbar zur "Setup()" Methode die wir schon aus der mainScreen Klasse kennen.

Da wir jetzt wissen, wie man eigene Klassen erstellt und sie so nutzt, dass man komplexere Objekte damit erstellen kann, wollen wir die Möglichkeiten die Entities uns bieten nochmal genauer betrachten.

Eine Entity kann im Grunde jeder sichtbare Teil eines Spiels/Programms sein(und teilweise auch nicht sichtbare). Eine Spielfigur, eine Plattform, ein Projektil, ein Text, ein Hintergrund, ein grafischer Effekt, eine Lebensanzeige, ein Button, etc.

Eine Entity besteht im wesentlichen aus 2 Dingen.

Eigenschaften(auch Felder genannt) und **Methoden**.

Eigenschaften

Eigenschaften beschreiben den Zustand der Entity.

Beispiel:

Wo sie ist, wie schnell sie ist, wie viel Leben sie hat, in welcher Animationsphase sie sich gerade befindet usw.

Die **Eigenschaften** bestehen aus Variablen

Beispiel:

Vector2f position, für die Position

Vector2f speed, für die Geschwindigkeit

int life, für die Lebenspunkte

Methoden

Methoden beschreiben anders als Eigenschaften nicht den Zustand, sondern wie sich die Entity verhält. Sie sind ausführbare Teile von Code. Man gibt ihnen beim Ausführen Parameter und sie geben einen Rückgabewert zurück. Dies entspricht der Mathematischen Sichtweise von Funktionen (z.B. $f(x) = x^2$), aber in der Informatik benutzt man sie auch für Objekte, also auch unsere Entities.

Immer wenn die Entity etwas tut, muss das in einer **Methode** stehen. Man kann sich also vorstellen, die **Methoden** sind wie alle Handlungen einer Entity. Diese sind aufgrund ihrer Funktionsweise oft vom Rückgabotyp "void".

Beispiel

Man möchte eine Entity erstellen, die einem Auto ähnelt. Was kann man mit einem Auto machen? Es fahren. Also würde man in die Klasse eine "fahren" **Methode** schreiben. Die Parameter wären die Informationen die man braucht, wenn man fährt. Das wäre hier z.B. das Fahrziel.

Oder man möchte eine Wecker-Entity erstellen. Was kann man mit einem Wecker machen? Ihn stellen. Also eine "weckerStellen" **Methode**, die nötigen Parameter wäre die Uhrzeit, auf die man sie stellt. Ein Wecker kann auch klingeln. Dafür könnte man eine "klingeln" **Methode** machen. Parameter braucht man hier allerdings keine.

Klassen für Spielobjekte: Entities

Will man in SFML Spielobjekte erstellen, die zu kompliziert sind um sie als einfaches RectangleShape, CircleShape oder Sprite darzustellen, so muss man eine eigene Klasse für diese Objekte erstellen. Diese Art von Spieleobjekte nennt man "Entity".

Im Template gibt es neben der MainScreen Klasse auch andere Klassen. Unter dem Ordner entities/ gibt es die Klasse Entity. Diese sieht wie folgt aus:

```
using System;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using System.Collections.Generic;
using System.Text;
using SFML_GAME.framework;
using SFML_GAME.screens;

namespace SFML_GAME.entities
{
    class Entity
    {
        // Graphic Variable sollte in jeder Entity drin sein
        // RectangleShape, CircleShape oder Sprite
        public RectangleShape graphic;

        MainScreen screen;

        // Konstruktor
        public Entity(MainScreen parentScreen)
        {
            screen = parentScreen;
            graphic = new RectangleShape();
            graphic.Size = new Vector2f(50, 50);
            graphic.FillColor = Color.Blue;
            graphic.Position = new Vector2f(300, 300);
        }

        // Loop Methode
        public void loop()
        {

        }

        public void draw()
        {
            screen.draw(this.graphic);
        }
    }
}
```

Sie besteht aus den Methoden:

1. Entity: Der Konstruktor der Entity, wird bei der Erstellung der Entity aufgerufen. Kann wie setup() in mainScreen benutzt werden.
2. loop: soll jeden Frame aufgerufen werden. Kann wie loop() in mainScreen benutzt werden.
3. draw: In dieser Methode sollen alle Objekte die zur Entity gehören gedrawt werden.

In der Konstruktor-Methode(Entity) wird ein Screen übergeben. Dieser dient der Entity als Referenz, wo es existiert. Jede Entity sollte diese Variable speichern.

Außerdem wird schon ein RectangleShape erstellt und ihm Werte zugewiesen.

In der draw Methode wird das RectangleShape auf dem Screen gedrawt.

In unserem MainScreen können wir nun die Entity erstellen, als würden wir andere Objekte(Sprite, RectangleShape, CircleShape) erstellen. Allerdings müssen wir bei der initialisierung (new Entity()) zwischen die Runden Klammern ein "this" schreiben, damit die Entity den Parameter bekommt. In loop() sollten wir die loop() und draw() Methode von ihr aufrufen.

```
namespace SFML_GAME.screens
{
    class MainScreen : Screen
    {
        // Deklariere hier Objekte oder Klassenvariablen!
        Entity box;

        // Setup, wird immer einmal zu Beginn eines Screens aufgerufen
        // Hier Startwerte setzen!
        public override void setup()
        {
            box = new Entity(this);
        }

        // Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
        public override void loop()
        {
            box.loop();

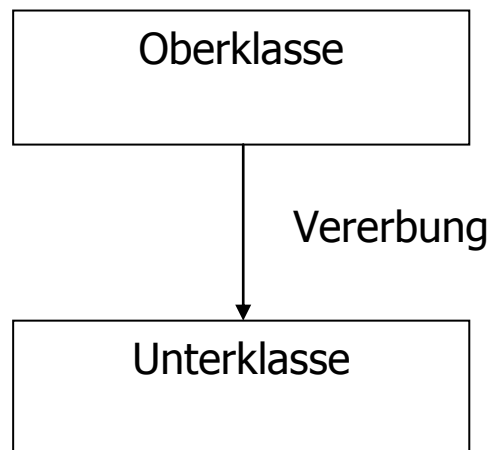
            box.draw();
        }
    }
}
```

Jetzt können wir in der Entity in ihrer loop() mit dem Shape arbeiten wie wir es von MainScreen gewohnt sind.

Objekt Orientierung: Vererbung

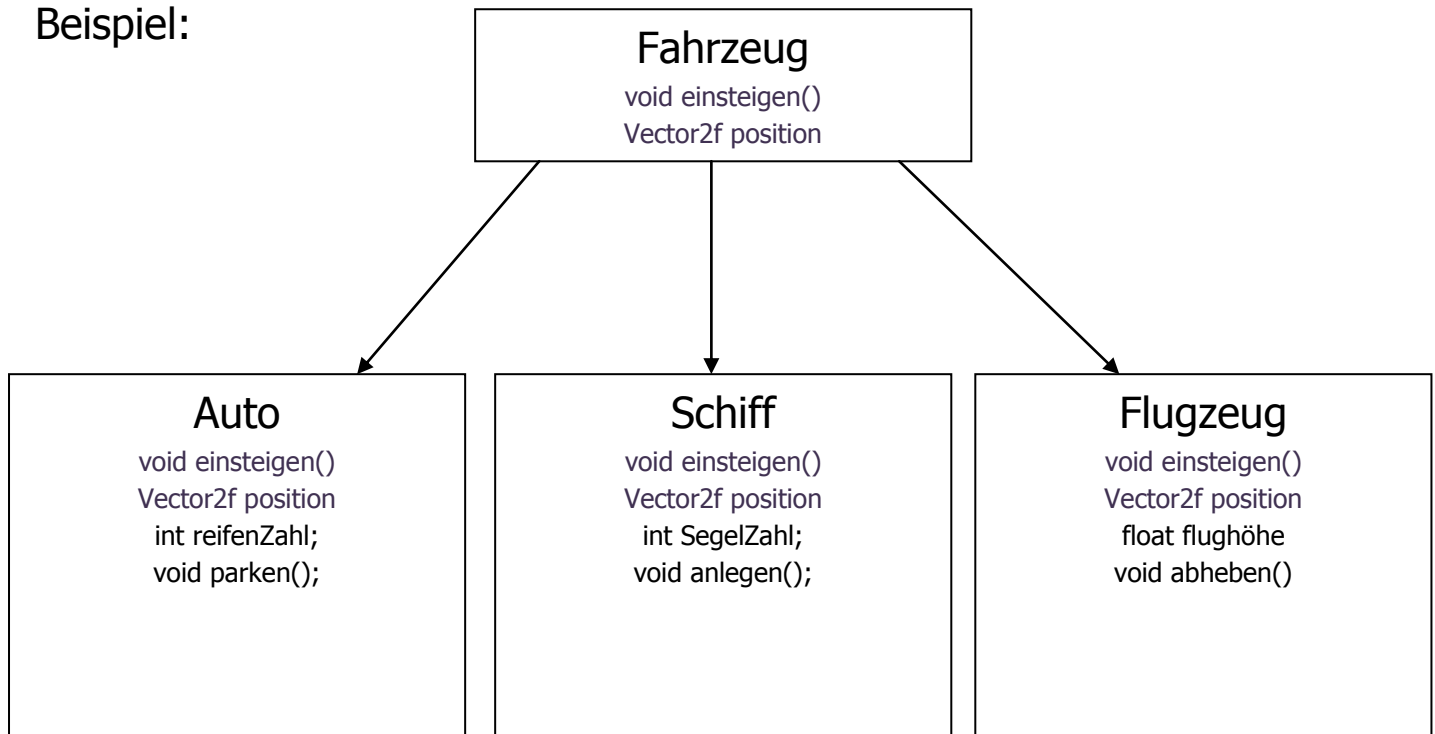
Was man damit erreichen möchte ist, dass alle Dinge, die eine Entity betreffen, also ihre Eigenschaften, wie sie sich verhält und was man mit ihr machen kann, in nur eine Code-Klasse stehen. Man versucht alles in seinem Programm in Objekte aufzuspalten, dies nennt sich Objekt Orientierung.

Ein großer Vorteil von Objekt Orientierung ist die **Vererbung**. Dabei erbt eine Klasse von einer anderen. Man hat also eine **Oberklasse** von der eine **Unterklasse** erbt. (Man spricht auch oft von Elternklasse und Kindklasse)



Wenn eine Klasse von einer anderen erbt, erhält sie automatisch alle Methode und alle Eigenschaften(also Variablen), die auch die Oberklasse hat. Eine Unterklasse kann alles, was ihre Oberklasse kann und mehr. Eine Unterklasse kann man auch als Erweiterung der Oberklasse ansehen.

Beispiel:



Hier ist die Oberklasse "Fahrzeug". Die Unterklassen sind Auto, Boot und Flugzeug.

Die Klasse Fahrzeug besitzt "einsteigen()" und "vector2f position".

Da Auto, Schiff und Flugzeug von Fahrzeug erben, haben sie auch diese Methode und Variable. Zusätzlich hat jede Unterklasse noch ihre eigenen Methoden und Eigenschaften, die nur für diese Unterklasse gelten.

Hier müsste man den Code für die Position und das Einsteigen nur einmal programmieren und es wäre automatisch in den 3 Unterklassen eingebaut.

Außerdem gibt es etwas, das sich Polymorphie nennt, es erlaubt uns, Objekte von Unterklassen in Variablen der Oberklassen zu speichern.

Also z.b. :

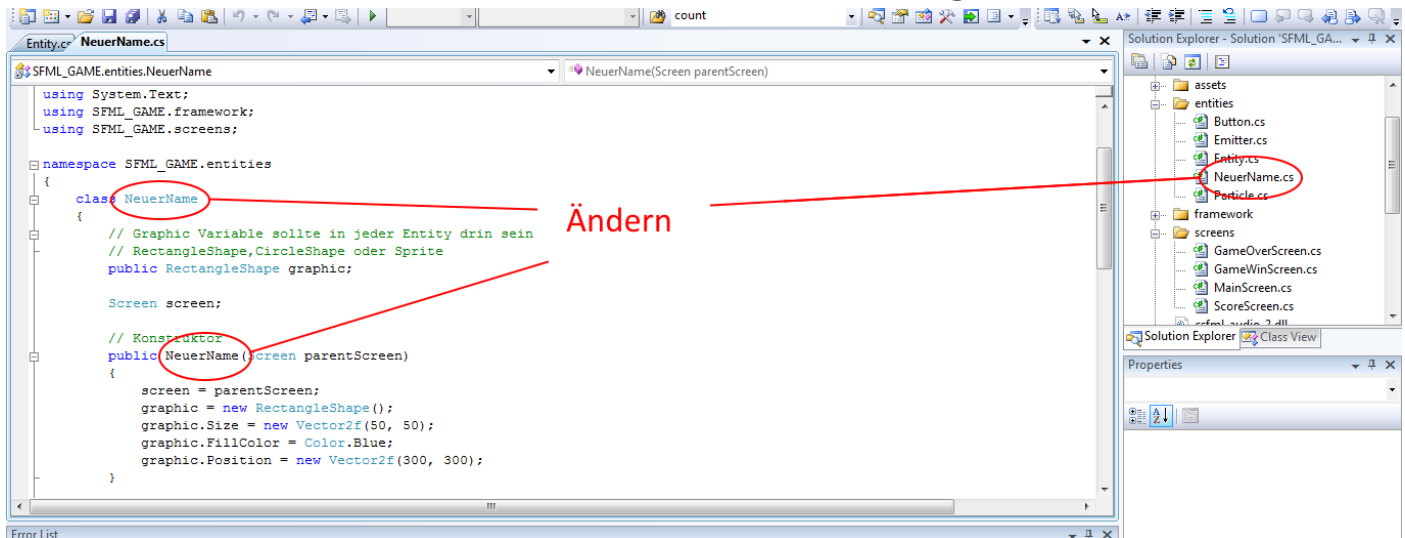
```
Fahrzeug x = new Auto();
```

Dies ermöglicht viele Verallgemeinerungen im Quellcode, die das Programmieren vereinfachen.

Entities erstellen

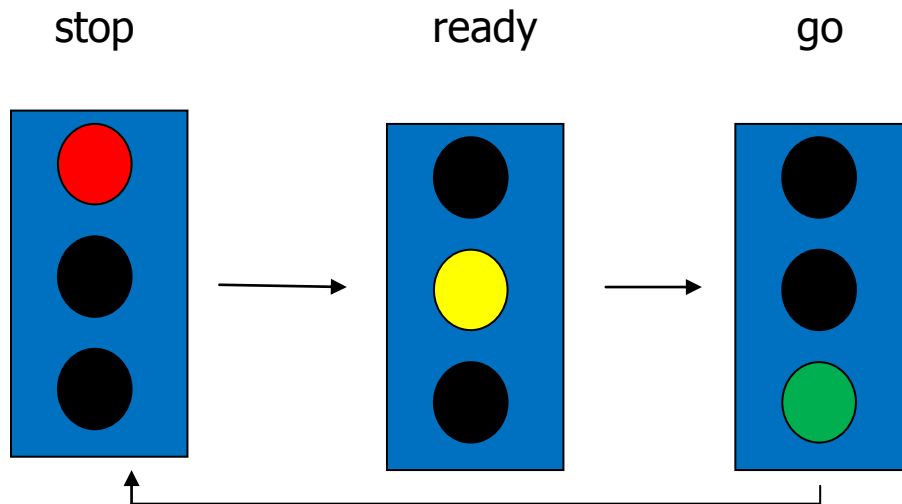
Wie schon vorher schreiben wir neue Entities nicht komplett neu, sondern kopieren uns die "Entity.cs" im Ordner "entities". Sie dient als Vorlage für alle weiteren Entities um Zeit und Nerven zu sparen.

Wir ändern die **Datei**, den Namen der **Klasse** und den Namen des **Konstruktors** alle zu dem Namen unserer neuen Entity. Außerdem sollte die Datei im **Ordner** "entities" liegen.



In der Entity-Klasse ist Beispielhaft schon ein RectangleShape als Grafik. Dieses kann mit einem oder mehreren Shapes/Sprites ersetzt werden.

Beispielhaft wollen wir hier eine Ampel machen:



Die Ampel soll immer eine der 3 Zustände haben und zwischen ihnen in der angezeigten Reihenfolge wechseln. Das Wechseln soll über Tastendrücke ausgelöst werden.

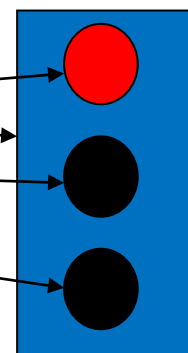
Als erstes erstellen wir eine neue Entity Klasse, also kopieren wir Entity.cs, ändern in der Kopie den Dateinamen zu "Ampel.cs", den Klassen und Konstruktornamen zu "Ampel".(siehe Oben)

Als nächstes brauche wir die Shapes. Man kann sehen, dass die Ampel aus einem RectangleShape und 3 CircleShapes besteht. Also erstellen wir diese:

```
class Ampel
{
    // Graphic Variable sollte in jeder Entity dr
    // RectangleShape, CircleShape oder Sprite
    public RectangleShape box;
    public CircleShape red;
    public CircleShape yellow;
    public CircleShape green;

    Screen screen;

    // Konstruktor
    public Ampel(Screen parentScreen)
    {
        screen = parentScreen;
    }
}
```



In unserem Konstruktor müssen wir diese natürlich erstmal erstellen.

```
public Ampel(Screen parentScreen)
{
    screen = parentScreen;

    box = new RectangleShape();
    box.Size = new Vector2f(100, 300);
    box.FillColor = Color.Blue;
    box.Position = new Vector2f(100, 100);

    red = new CircleShape();
    red.Radius = 40;
    red.FillColor = Color.Red;
    red.Position = new Vector2f(110, 110);

    yellow = new CircleShape();
    yellow.Radius = 40;
    yellow.FillColor = Color.Black;
    yellow.Position = new Vector2f(110, 210);

    green = new CircleShape();
    green.Radius = 40;
    green.FillColor = Color.Black;
    green.Position = new Vector2f(110, 310);
}
```

Damit wir unsere Shapes später auch sehen können, müssen diese in DrawAll() auch gezeichnet werden.

```
public void draw()
{
    screen.draw(box);
    screen.draw(red);
    screen.draw(yellow);
    screen.draw(green);
}
```

Jetzt können wir in der mainScreen unsere Entity einbauen.

```
class mainScreen : Screen
{
    // Deklariere hier Objekte oder Klassenva
    Ampel ampel;

    // Setup, wird immer einmal zu Beginn ein
    // Hier Startwerte setzen!
    public override void setup()
    {
        ampel = new Ampel(this);
    }

    // Loop, wird jeden Frame (60 mal die Sek
    public override void loop()
    {
        ampel.loop();

        ampel.draw();
    }
}
```

Was jetzt noch fehlt, ist das Schalten der Ampel.

Dazu schreiben wir eine "step" Methode **in die Ampel Entity**.

Diese soll den Zustand immer einen Schritt weiterwechseln.

```
.
public void step()
{
}
}
```

Bevor wir den Inhalt unserer Step-Methode schreiben, wollen wir vorher eine Variable schreiben, die den Zustand der Ampel speichert. Dazu erstellen wir ein int-Variable mit den Namen "state". Wobei state 0 ist, wenn die Ampel rot, 1 bei gelb und 2 bei grün.

```
public CircleShape red;
public CircleShape yellow;
public CircleShape green;

//0 = red, 1 = yellow, 2 = green
public int state;

Screen screen;

// Konstruktoren
```

In Step müssen wir durch If-Abfragen jetzt überprüfen, welchen Zustand die Ampel im Moment hat.

```
public void step()
{
    if (state == 0)
    {
        //Ampel is rot
    }
    else if (state == 1)
    {
        //Ampel ist gelb
    }
    else if (state == 2)
    {
        //Ampel ist grün
    }
}
```

Jetzt müssen wir den Zustandswechsel der Ampel programmieren, in dem wir den entsprechenden nächsten Zustand einstellen. State sollte auch auf den neuen Zustand gesetzt werden.

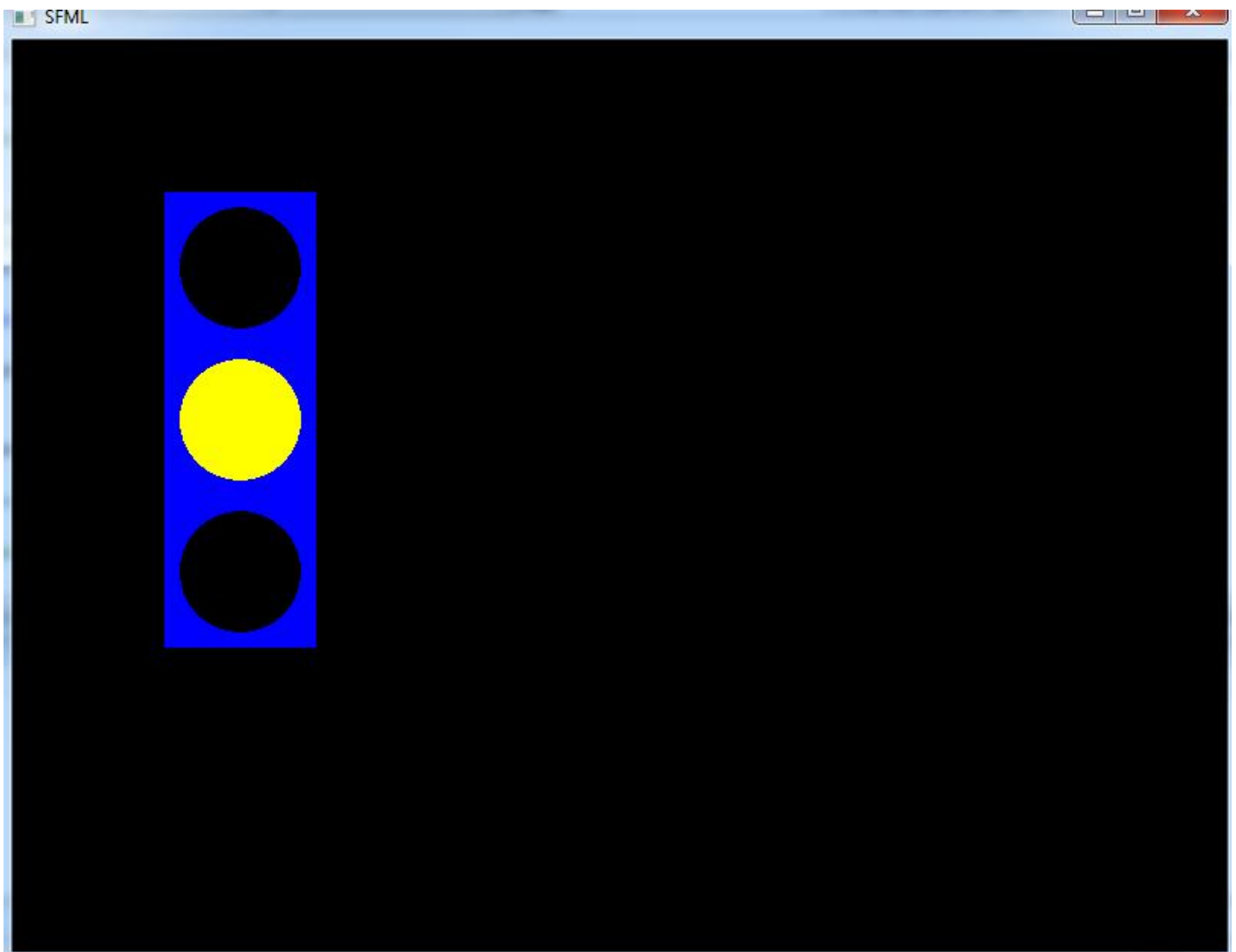
```
public void step()
{
    if (state == 0)
    {
        //Ampel is rot, wird gelb

        red.FillColor = Color.Black;
        yellow.FillColor = Color.Yellow;
        state = 1;
    }
    else if (state == 1)
    {
        //Ampel ist gelb, wird grün
        yellow.FillColor = Color.Black;
        green.FillColor = Color.Green;
        state = 2;
    }
    else if (state == 2)
    {
        //Ampel ist grün, wird rot
        green.FillColor = Color.Black;
        red.FillColor = Color.Red;
        state = 0;
    }
}
```

Somit ist unsere Ampel schon fertig.

Jetzt müssen wir in der Mainscreen nurnoch step() aufrufen, immer wenn wir Space drücken.

```
// hier startweise setzen:  
public override void setup()  
{  
    ampel = new Ampel(this);  
  
    game.gameWindow.KeyPressed += new EventHandler<KeyEventArgs>(gameWindow_KeyPressed);  
}  
  
void gameWindow_KeyPressed(object sender, KeyEventArgs e)  
{  
    if (e.Code == Keyboard.Key.Space)  
    {  
        //Spacebar wurde gedrückt  
        ampel.step();  
    }  
}
```



Physik-Simulation

In Spielen will man oft, dass Objekte sich physikalisch korrekt verhalten. Dazu muss man die Naturgesetze die man aus der Physik kennt im Programm umsetzen.

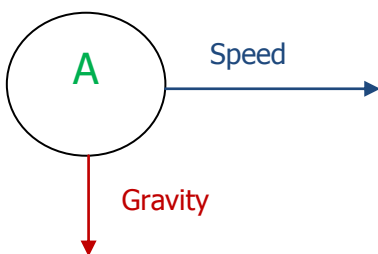
Mithilfe von Vektoren lassen sich die meisten physikalischen Verhaltensweisen gut simulieren. Da in SFML die Position bereits auf Vektoren basiert, erleichtert das viele Verfahren.

Beispiel

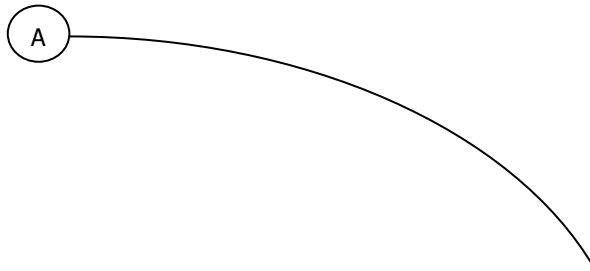
Häufig muss man Geschwindigkeit & Beschleunigung für z. b. ein Jump & Run Spiel in seinem Quellcode umsetzen. Dies wird vor allem über Geschwindigkeitsvektoren gehandhabt.

So erhalten alle "bewegbaren" Objekte einen Geschwindigkeitsvektor(hier Speed), um den die Position des Objekts jeden Frame erhöht wird. Der Geschwindigkeitsvektor wird wiederum von äußerlichen Einflüssen verändert(z. b. Gravitation).

```
A.Position += speed;  
speed += gravity;
```



Resultierende
Flugbahn



Simulation von Elastizität: Gummiband

Als Übung wollen wir ein Gummiband simulieren.

Um das Gummiband zu simulieren, gehen wir von einem Haken und einem Ball aus, der durch ein Gummiband mit dem Haken verbunden ist.

Wir erstellen also ein CircleShape und RectangleShape im Mainscreen.

```
// Deklariere hier Objekte oder Klassenvariablen!
CircleShape Ball;
RectangleShape Holder;

// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
public override void setup()
{
    Ball = new CircleShape();
    Ball.Radius = 20;
    Ball.Origin = new Vector2f(20, 20);
    Ball.FillColor = Color.Red;
    Ball.Position = new Vector2f(400, 400);

    Holder = new RectangleShape();
    Holder.Size = new Vector2f(10,10);
    Holder.Origin = new Vector2f(5, 5);
    Holder.FillColor = Color.Green;
    Holder.Position = new Vector2f(400, 300);
}

// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
public override void loop()
{
    draw(Ball);
    draw(Holder);
}
```

Da der Ball sich bewegen soll, erstellen wir einen Geschwindigkeitsvektor für ihn.

```
// Deklariere hier Objekte oder Klassenvariablen!
CircleShape Ball;
RectangleShape Holder;
Vector2f speed;

// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
public override void setup()
{
    speed = new Vector2f(0,0);
}
```

```
[...]  
}
```

Da speed ein Geschwindigkeitsvektor ist, müssen wir jeden Frame die Position von Ball um diesen Vektor ändern.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen  
public override void loop()  
{  
    Ball.Position += speed;  
  
    draw(Ball);  
    draw(Holder);  
  
}
```

Noch bewegt sich der Ball nicht, weil keine Kräfte auf ihn wirken. Das ändern wir indem wir einen Vektor gravity erstellen, der die Gravitation darstellt. Diesen stellen wir auf (0|0,2).

```
Vector2f gravity;  
  
// Setup, wird immer einmal zu Beginn eines Screens aufgerufen  
// Hier Startwerte setzen!  
public override void setup()  
{  
    speed = new Vector2f(0,0);  
    gravity = new Vector2f(0,0.2f);  
    [...]  
}
```

Diese Kraft müssen wir jetzt noch auf unseren Geschwindigkeitsvektor addieren.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen  
public override void loop()  
{  
    speed += gravity;  
    Ball.Position += speed;  
  
    draw(Ball);  
    draw(Holder);  
  
}
```

Jetzt sollte der Ball runterfallen. Als nächstes wollen wir das Gummiband einbauen. Dazu brauchen wir physikalische Formeln, die das Verhalten eines Gummibands beschreiben.

Als vereinfachung gehen wir davon aus, dass ein Gummiband eine gedämpfte harmonische Schwingung erzeugt.

Für Harmonische Schwingungen gilt, dass die Rückstellkraft proportional zur Auslenkung ist. Das heißt, umso länger wir unser Gummiband ziehen, desto stärker zieht es sich zusammen. Wir berechnen also die Länge des Gummibands und können daraus eine Kraft herleiten, die auf den Ball wirkt.

Hierfür nutzen wir die Utility.Function Library im Template.

Die Länge des Gummibands kann man errechnen, indem man die Positionen des Haken und des Balls voneinander abzieht und die Länge des resultierenden Vektors berechnet.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
public override void loop()
{
    speed += gravity;
    Ball.Position += speed;

    float strength = (float)utility.Functions.lengthOfVector(Ball.Position -
                                                             Holder.Position);

    draw(Ball);
    draw(Holder);
}
```

Jetzt haben wir unsere Kraft, diese wollen wir jetzt auf unseren Ball anwenden. Aber wir brauchen dafür einen Vektor. Deswegen bilden wir den Einheitsvektor des Differenzvektors, dieser zeigt vom Ball direkt zum Holder und hat die Länge 1. Wenn wir diesen mit der Stärke multiplizieren, ist das unsere Rückstellkraft. Diese versehen wir noch mit einem Faktor, damit die Kraft nicht zu stark ist.

```
// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
public override void loop()
{
    float strength = (float)utility.Functions.lengthOfVector(Ball.Position -
                                                             Holder.Position);
    Vector2f force = strength * utility.Functions.getUnitVector(Holder.Position
                                                                - Ball.Position) * 0.01f;

    speed += force;
    speed += gravity;
    Ball.Position += speed;

    draw(Ball);
    draw(Holder);
}
```


Jetzt schwingt unser Ball, aber noch ist es kein Gummiband.

Gummibänder üben nur Kraft aus, wenn man sie länger zieht als ihre gewöhnliche Länge. Deswegen ziehen wir 100 von der strength variable ab, so dass das Gummiband eine Länge von 100 Pixeln hat. Außerdem müssen wir jetzt negative Werte nullen.

```
public override void loop()
{
    float strength = (float)utility.Functions.lengthOfVector(Ball.Position -
                                                              Holder.Position) - 100;

    if (strength < 0) strength = 0;

    Vector2f force = strength * utility.Functions.getUnitVector(Holder.Position
                                                                - Ball.Position) * 0.01f;

    speed += force;
    speed += gravity;
    Ball.Position += speed;

    draw(Ball);
    draw(Holder);
}
```

Als letztes Simulieren wir noch Reibung, indem wir den Speed Vektor mit einem Faktor multiplizieren, der etwas kleiner als 1.0 ist.

```
public override void loop()
{
    float strength = (float)utility.Functions.lengthOfVector(Ball.Position -
                                                              Holder.Position) - 100;

    if (strength < 0) strength = 0;

    Vector2f force = strength * utility.Functions.getUnitVector(Holder.Position
                                                                - Ball.Position) * 0.01f;

    speed += force;
    speed += gravity;
    speed *= 0.95f;
    Ball.Position += speed;

    draw(Ball);
    draw(Holder);
}
```

Das Gummiband ist jetzt fertig, aber um es wirklich in Aktion zu sehen, kann man mithilfe von Mausklicks den Haken bewegen. Dazu brauchen wir nur folgende If-Klammer:

```
if (Mouse.IsButtonPressed(Mouse.Button.Left))
{
    Holder.Position = new Vector2f(Mouse.GetPosition(game.gameWindow).X,
                                   Mouse.GetPosition(game.gameWindow).Y);
}
```

Graphical User Interface(GUI)

Das **GUI** (**G**raphical **U**ser **I**nterface) ist in so gut wie jedem Spiel vorhanden. Obwohl man den Begriff auf mehrere Weisen verstehen kann, werden damit im Game Programming vor allem Steuerelemente bezeichnet, die nicht Teil der dargestellten Spielwelt sind und somit nur den Spielkomfort des Spielers da sind. Dabei geht es von einfachen Buttons und Labels über Lebensbalken bis zum Fadenkreuz.



Der Button

Wir wollen uns hier insbesondere mit einem Element befassen: dem Button.

Ein Button ist eine Fläche auf dem Bildschirm, die wenn man die Maus darüber fährt und klickt, eine bestimmte Aktion auslöst. Dabei kann es sein, dass die Klickdauer(gedrückt halten) einen Einfluss hat oder nicht(Click-Event).

Außerdem hat der Button meist optische Änderungen, wenn man mit der Maus darüber "schwebt", sowie wenn man ihn drückt.

Im SFML Template befindet sich die Entity "Button". Diese können wir nutzen um Buttons zu erzeugen. Dazu erstellen wir wie immer zuerst eine Variable:

```
// Deklariere hier Objekte oder Klassenvariablen!  
Entity example;  
Button knopf;
```

Dann müssen wir den Button erstellen. Dabei ist zu beachten, der Button benötigt eine Vielzahl von Parametern im Konstruktor:

```
new Button(MainScreen parentScreen, Vector2f Position, Vector2f Size, String Text, uint  
TextSize)
```

- parentScreen = der Verweis auf den Screen, hier einfach "this".
- Position = ein Vektor, der die Button Position angibt
- Size = Ein Vektor, der die Größe des Button angibt
- Text = Der Text, der in dem Button angezeigt wird
- TextSize = Die Textgröße, in Pixelhöhe(TextSize = 30 bedeutet ein Zeichen ist 30 Pixel hoch)

Wir setzen die Werte und erstellen uns so einen Button der "Press me" als Text besitzt.

```
// Initialisiere Beispiel (das führt automatisch den Konstruktor aus)  
example = new Entity(this);  
  
knopf = new Button(this, new Vector2f(20, 20), new Vector2f(100, 40),  
"Press me", 30);
```

Jetzt müssen wir den Button noch updaten und zeichnen, indem wir `loop()` und `draw()` entsprechend in der `loop`-Methode ausführen.

```
knopf.loop();
example.loop();

// draw(example.graphic) ist veraltet, stattdessen example.draw()
example.draw();
knopf.draw();
```

Anmerkung: Vorher wurde immer `draw(example.graphic)` benutzt. Dies wurde jetzt durch den `example.draw()` Befehl abgelöst. Die alte Methode ist aber immer noch nutzbar.

Jetzt erscheint der Button in der oberen Linke Ecke und kann mit der Maus gedrückt werden. Um den Button eine Funktion zuzuweisen, müssen wir in der `loop()`-Methode die `button.isPressed` und `button.click` Variablen abfragen:

```
public override void loop()
{
    if (knopf.isPressed)
    {
        // Knopf wird gerade gedrückt
        // Wird immer mehrmals ausgeführt!
    }

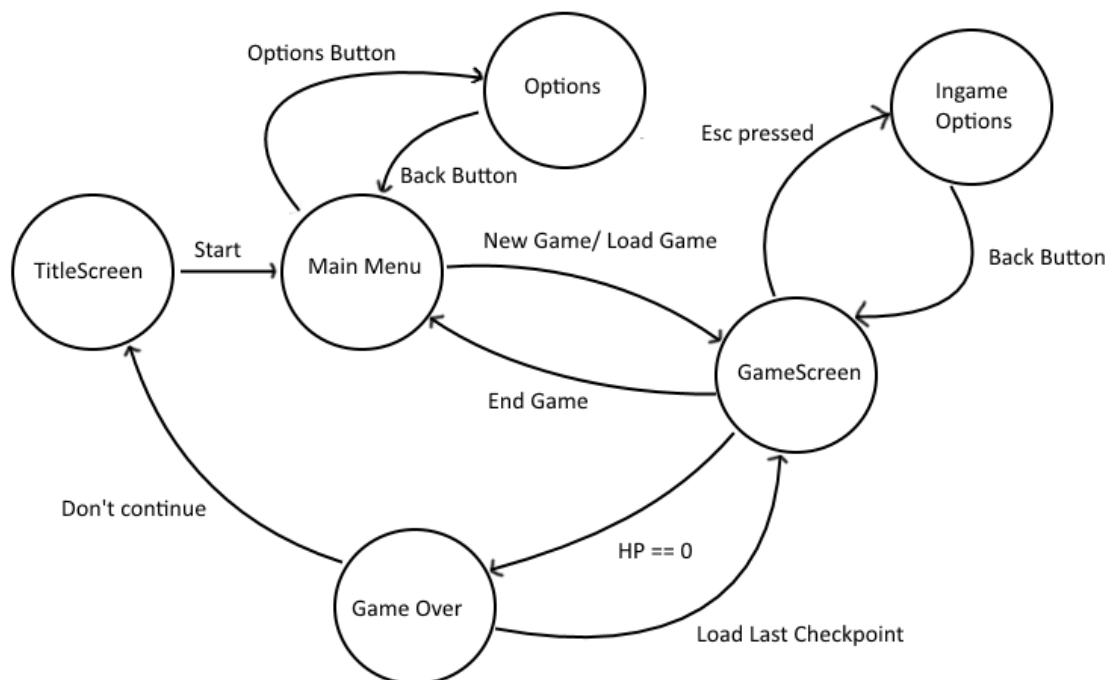
    if (knopf.click)
    {
        // Knopf wurde gerade gedrückt
        // Wird immer nur einmal pro Click ausgeführt!
    }
}
```

Screens

In Spielen spricht man oft von Screens. (Manchmal auch State oder Scene genannt) Wir haben bisher nur in der "MainScreen" Klasse gearbeitet. Aber größere Spiele brauchen oft mehrere Screens. Einen **Screen** kann grob als **Zustand** des Spiels bezeichnet werden. Bekannte Beispiele sind z. b.:

Hauptmenü, Optionen, GameScreen, GameOverScreen

Dabei sind neben den Inhalten der Screens auch die Übergänge wichtig. Welcher Screen muss wann zu welchem anderen führen? Man kann sich einen **Zustandsautomaten** erstellen, um die Übersicht zu behalten.



Neben normalen Screens gibt es noch SubScreens. Subscreens sind Screens die nicht alleine stehen, sondern nur als Teil eines anderen Screens angezeigt werden.

Typische Beispiele: Inventar, CharakterMenü, Ingame Options



In unserem Framework haben wir bereits mehrere Screens. Wir haben zwar nur in der mainScreen gearbeitet, aber es gibt auch den GameOverScreen, GameWinScreen und den ScoreScreen. Auf diese konnten wir mit vorgegebenen Methoden wechseln.

Zuerst erstellen wir uns einen neuen Screen. Um zu wissen, wie eine Screen Klasse aufgebaut sein muss, sehen wir uns mainScreen an:

```
using System;
using System.Collections.Generic;
using System.Text;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using SFML_GAME.entities;
using SFML_GAME.framework;
```

Using Anweisungen

Angabe der Parent Klasse
(haben wir noch nicht
behandelt)

```
namespace SFML_GAME.screens
```

```
{
    class mainScreen : Screen
    {
```

```
        // Deklariere hier Objekte oder Klassenvariablen!
```

```
        // Setup, wird immer einmal zu Beginn eines Screens aufgerufen
```

```
        // Hier Startwerte setzen!
```

```
        public override void setup()
        {
```

```
        }
```

```
        // Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
```

```
        public override void loop()
        {
```

```
        }
```

```
    }
}
```

setup() und loop()

Eine Screen Klasse muss immer das ": Screen" hinter der Klassendeklaration besitzen. Das macht sie erst zu einem Screen. Dies hat mit Vererbung zutun, was wir später näher behandeln.

Dazu brauchen wir immer die Methoden "setup()" und "loop()". Außerdem sollten die Using Anweisungen übernommen werden.

Hier eine frisch erstellte Klasse namens NewScreen:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SFML_GAME_2.screens
{
    class NewScreen
    {

    }
}
```

Hier können wir jetzt unsere oben genannten Elemente einfügen.

```
using System;
using System.Collections.Generic;
using System.Text;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using SFML_GAME.entities;
using SFML_GAME.framework;
```

Using Anweisungen

Angabe der Parent Klasse
(haben wir noch nicht
behandelt)

```
namespace SFML_GAME_2.screens
{
    class NewScreen : Screen
    {
```

```
        public override void setup()
        {

        }
```

setup() und loop()

```
        public override void loop()
        {

        }
    }
```

```
}
```

NewScreen ist jetzt ein neuer leerer Screen. Wir können ihn jetzt wie MainScreen mit Elementen füllen.

```
using System;
using System.Collections.Generic;
using System.Text;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using SFML_GAME.entities;
using SFML_GAME.framework;

namespace SFML_GAME_2.screens
{
    class NewScreen : Screen
    {
        Entity example;

        public override void setup()
        {
            example = new Entity(this);
        }

        public override void loop()
        {
            example.loop();
            example.draw();
        }
    }
}
```

Um jetzt zwischen Screens wechseln zu können, brauchen wir die Methode "game.switchScreens(..)" in die Klammern kommt der Screen, zu dem wir wechseln wollen. Für unseren NewScreen also:

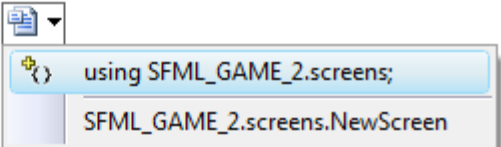
```
game.switchScreen(new NewScreen());
```

Dabei kommt es wahrscheinlich zu einem Error, dass die Klasse MainScreen unseren NewScreen nicht kennt. Dafür einfach mit Maus auf das Kontext Menü warten und die entsprechende Using Anweisung anklicken.

```
// Initialisiere Beispiel (das führt automatisch den Konst

game.switchScreen(new NewScreen());

Loop, wird jeden Frame
blic override void loop(
```



The screenshot shows a code completion menu in an IDE. The menu is open over the text 'new NewScreen()' in the code. It contains two options: 'using SFML_GAME_2.screens;' and 'SFML_GAME_2.screens.NewScreen'. The first option is highlighted in blue.

Jetzt können wir zu anderen Screens wechseln.

SubScreens

Um aus einen Screen, einen SubScreen zu machen, ist eigentlich nicht viel nötig. Man muss nur folgendes beachten:

Aus dem "Screen", sollte man ein "SubScreen" machen. Außerdem sollte "loop()" zu "subloop()" unbenannt werden.

Letztlich muss man noch folgenden Code in die Klasse einfügen:

```
public NewScreen(Screen parentScreen)
    : base(parentScreen)
{
}
```

Wobei "NewScreen" durch den entsprechenden Klassennamen ersetzt werden muss.

Unsere NewScreen SubScreen Klasse sieht jetzt so aus:

```
using System;
using System.Collections.Generic;
using System.Text;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using SFML_GAME.entities;
using SFML_GAME.framework;

namespace SFML_GAME.screens
{
    class NewScreen : SubScreen
    {
        public NewScreen(Screen parentScreen)
            : base(parentScreen)
        {

```

```

    }

    public override void setup()
    {

    }

    public override void subloop()
    {

    }
}

```

Wenn wir jetzt zu unserer NewScreen Klasse wechseln wollen mit:

```
game.switchScreen(new NewScreen());
```

Gibt es einen Error, weil ein SubScreen braucht einen HauptScreen, auf dem er angezeigt wird. Deswegen geben wir ein "this" mit, damit es auf den MainScreen projiziert wird.

```
game.switchScreen(new NewScreen(this));
```

Jetzt werden alle Inhalte von NewScreen auf MainScreen projiziert. Beide Screens sind gleichzeitig aktiv.

Wenn man jetzt wieder zurück auf den MainScreen wechseln will, gibt es 2 Möglichkeiten.

Mit

```
game.switchScreen(new MainScreen());
```

wird ein neuer MainScreen erzeugt, dabei geht der alte verloren.

Mit

```
this.goBack();
```

gelangt man wieder zum vorherigem MainScreen, mit allen Objekten so wie sie waren.

Speichern & Laden

Bisher haben wir nur Programme geschrieben, die bei jedem Neustart in einem vordefinierten Anfangszustand starten. Um aber Spiele zu programmieren, die man nicht nur in einer langen Sitzung durchspielen kann, sondern auch Zwischendurch beenden und weitermachen kann, muss man eine Speicher & Ladefunktion einbauen.

Das grundsätzliche Prinzip ist simpel:

Speichern

- Ein Event(z. b. ein Knopfdruck) löst das Speichern aus.
- Der momentane Ist-Zustand des Spiels wird in einer gut speicherbaren Datenform festgehalten.
- Diese Daten werden auf der Festplatte des Computers gespeichert.

Laden

- Ein Event löst das Laden aus.
- Das Programm sucht nach gespeicherten Daten.
- Wenn es welche findet, werden diese in das Programm geladen.
- Aus den Daten wird gelesen, wie der Zustand des Spiels im Moment des Speicherns war.
- Der Spielzustand wird so gesetzt, dass er dem Zustand beim Speichern entspricht.

Es gibt viele Wege, eine Speicher- und Ladefunktion einzubauen. Beim Programmieren dieser sollte man sich über folgende Dinge Gedanken machen:

- Wann darf gespeichert werden? (immer oder nur an bestimmten Speicherorten)
- Was wird alles gespeichert?

- Darf der Spieler mehrere Speicherstände gleichzeitig haben?

Um in SFML das Speichern umzusetzen, müssen wir als erstes Wissen, wie man Sachen auf der Festplatte speichert. Dafür nutzen wir die von C# vorgegebene System.IO Klasse. Also werden wir in MainScreen als erstes eine using Anweisung schreiben, damit wir diese verwenden können:

```
using System;
using System.Collections.Generic;
using System.Text;
using SFML.Window;
using SFML.Audio;
using SFML.Graphics;
using SFML.System;
using SFML_GAME.entities;
using SFML_GAME.framework;
using System.IO;

namespace SFML_GAME.screens
{
    class MainScreen : Screen
    {
```

Um jetzt mit der Datei zu arbeiten, benutzen wir sogenannte Streams. Diese sind Schnittstellen, um mit externen Daten zu arbeiten. In System.IO gibt es 2 Klassen die wir brauchen: StreamReader und StreamWriter.

StreamReader kann aus Dateien lesen.

StreamWriter kann neue Dateien erstellen und in diese Schreiben.

Also erstellen wir als erstes ein StreamWriter Objekt.

Beim Erstellen des StreamWriter Objekts, müssen wir den Dateinamen + eventuellen Pfad angeben.(siehe "Externe Ressourcen").

```
StreamWriter writer = new StreamWriter("savefile.txt");
```

Da die Datei "savefile.txt" noch nicht existiert, wird sie dadurch erstellt.

Um jetzt etwas in die savefile zu schreiben, nutzen wir die write-Methode von StreamWriter.

```
writer.Write("Hallo Welt!");
```

Nachdem wir alles geschrieben haben, was wir wollten, müssen wir den StreamWriter immer schließen, weil die Datei sonst "offen" bleibt.

```
writer.Close();
```

Nachdem wir das Programm ausgeführt haben, sollten wir im Debug-Ordner unseres Projektes die geschriebene Datei finden.

Achtung: Mit der hier gezeigten Weise wird der vorherigen Inhalt der savefile.txt immer gelöscht. Will man den Inhalt retten, so muss man ihn vorher lesen und abspeichern.

Um jetzt den Inhalt lesen zu können, benutzen wir den StreamReader auf die gleiche Weise. Nur statt write benutzen wir die ReadToEnd Methode, die uns den gesamten Inhalt als String zurückgibt.

```
StreamReader reader = new StreamReader("savefile.txt");  
String inhalt = reader.ReadToEnd();  
reader.Close();
```

Jetzt können wir also Strings auf die Festplatte schreiben und sie wieder lesen. Was jetzt noch fehlt ist Daten in Strings umwandeln zu können und zurück. Nehmen wir als Beispiel die Position einer Entity.

```
float x = example.graphic.Position.X;  
float y = example.graphic.Position.Y;
```

Wir speichern die Werte erstmal in passende Variablen.

```
String save = x.ToString() + "x" + y.ToString();
```

Mit der toString() Methode, können wir die Zahlen in ein String konvertieren und wir trennen X und Y wert mit einem "x".

Also wenn example an der Position 200|300 wäre, dann würde der String "200x300" sein.

Diese String schreiben wir dann mit unserem StreamWriter in die savefile.

```
float x = example.graphic.Position.X;  
float y = example.graphic.Position.Y;  
  
String save = x.ToString() + "x" + y.ToString();  
  
StreamWriter writer = new StreamWriter("savefile.txt");  
  
writer.WriteLine(save);  
writer.Close();
```

Jetzt müssen wir beim Laden den String dementsprechend aufteilen und die Werte herauslesen. Die Split() Methode ist dafür gut geeignet.

```
StreamReader reader = new StreamReader("savefile.txt");  
  
String inhalt = reader.ReadToEnd();  
  
reader.Close();  
  
String PosX = inhalt.Split('x')[0];  
String PosY = inhalt.Split('x')[1];
```

Split() gibt dir von einem String einen Array zurück, wobei das Zeichen in den Klammern ein Trennzeichen darstellt, welches den String aufteilt.

Aus "300x200" wird also {"300","200"} in einem Array. Das [0] und [1] hinter den Ausdrücken greift jeweils das erste und zweite Element des Arrays ab, also die "300" und die "200".

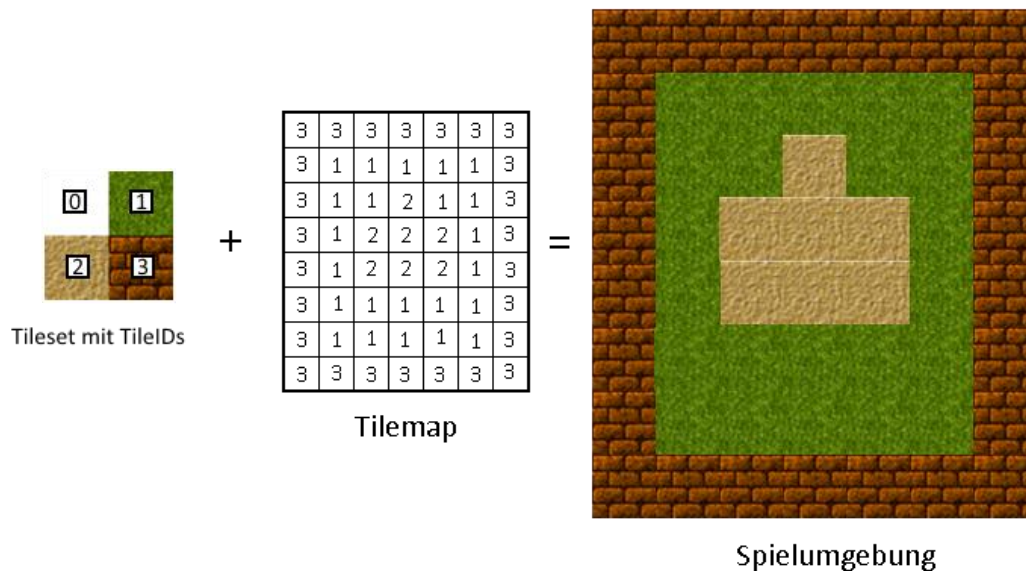
Jetzt müssen wir aus den Strings wieder float Werte machen, dazu nutzen wir `System.Convert.ToSingle()`.

```
float x = System.Convert.ToSingle(PosX);  
float y = System.Convert.ToSingle(PosY);  
  
example.graphic.Position = new Vector2f(x, y);
```

Achtung: Hier wäre eine Überprüfung Sinnvoll, ob der Inhalt der Savefile wirklich so aussieht wie wir denken und nicht anders. Sonst könnte es hier zu abstürzen kommen.

Tilemaps & Tilesets

Will man in einem Spiel eine größere Spielwelt bauen, die der Spieler erkunden können soll, dann wäre es sehr aufwendig diese komplett zu zeichnen (und würde oft auch einfach zu viel Speicher in Anspruch nehmen). Deswegen benutzt man oft **Tilemaps**. **Tilemaps** sind aus wenigen vorgefertigten Bildern zusammengesetzte Umgebungen für die Spielwelt.



Da es effizienter ist, speichert man alle **Tiles**(die Bausteine) als ein großes Bild. Diese nennt man **Tilesets**. Dann nummeriert man die **Tiles** in dem **Tileset** durch und vergibt jedem **Tile** eine **TileID**. Jetzt kann man sich aus den Bausteinen die Umgebungen zusammenbauen. Dazu nimmt man ein kariertes Muster und füllt die Felder mit den **TileIDs** der **Tiles**, die man an dieser Stelle haben möchte. Das Programm rendert jetzt die Tiles an den entsprechenden Stellen. Dadurch kann man sehr große Spielwelten erstellen, die keinen großen Speicher benötigen, da die Tilemaps nur aus Zahlen bestehen und es nur wenige tatsächliche Texturen gibt, die man speichern muss.

Um eine Tilemap in SFML umzusetzen, brauchen wir als erstes eine neue Entity, die wir Tilemap nennen.

```
class Tilemap
{
    Screen screen;

    // Konstruktor
    public Tilemap(Screen parentScreen)
    {

    }
    // Loop Methode
    public void loop()
    {

    }
    public void draw()
    {

    }
}
```

Eine Tilemap braucht auf jedenfall immer ein Tileset. Deswegen fügen wir als Parameter im Konstruktor eine Textur hinzu. Die speichern wir auch als Variable in der Klasse.

```
Texture tileset;

// Konstruktor
public Tilemap(Screen parentScreen, Texture set)
{
    tileset = set;
}
```

Außerdem müssen wir zusätzlich die Abmessungen der Tiles wissen, um das Tileset unterteilen zu können. Dazu fügen wir noch die Parameter tileWidth und tileHeight hinzu und speichern auch sie in der Klasse.

```
Texture tileset;
int tileWidth;
int tileHeight;

// Konstruktor
public Tilemap(Screen parentScreen, Texture set, int width, int height)
{
    //Speicher alle Parameter in der Klasse
    screen = parentScreen;
    tileset = set;
    tileWidth = width;
    tileHeight = height;
}
```

Jetzt müssen wir das Tileset im Konstruktor in seine Tiles aufteilen. Dazu erstellen wir uns einen Array von Sprites. Jedes Sprite soll ein Tile sein,

also brauchen wir soviele Sprites wie Tiles. Da wir die Maße eines Tiles kennen und die Höhe und Breite der Textur, können wir ausrechnen wie viele Tiles wir haben.

```
public Tilemap(Screen parentScreen, Texture set, int width, int height)
{
    //Speicher alle Parameter in der Klasse
    screen = parentScreen;
    tileset = set;
    tileWidth = width;
    tileHeight = height;

    //Anzahl der Tiles in einer Reihe
    int w = (int)(tileset.Size.X / tileWidth);

    //Anzahl der Tiles in einer Spalte
    int h = (int)(tileset.Size.Y / tileHeight);

    //Anzahl der Tiles im Tileset
    int amount = w * h;

}
```

Jetzt brauchen wir einen Sprite-Array, der die Tiles speichern kann.

```
Texture tileset;
int tileWidth;
int tileHeight;
Sprite[] tiles;

// Konstruktor
public Tilemap(Screen parentScreen, Texture set, int width, int height)
{
    //Speicher alle Parameter in der Klasse
    screen = parentScreen;
    tileset = set;
    tileWidth = width;
    tileHeight = height;

    //Anzahl der Tiles in einer Reihe
    int w = (int)(tileset.Size.X / tileWidth);

    //Anzahl der Tiles in einer Spalte
    int h = (int)(tileset.Size.Y / tileHeight);

    //Anzahl der Tiles im Tileset
    int amount = w * h;

    //Erstelle Array zum Speichern der Tiles
    tiles = new Sprite[amount];

}
```

Jetzt müssen wir mit einer Schleife das Array füllen, dazu nehmen wir uns immer ein Stück aus dem Tileset und gehen dann zum nächsten Tile.

```
// Konstruktor
public Tilemap(Screen parentScreen, Texture set, int width, int height)
{
    [...]

    //Erstelle Array zum Speichern der Tiles
    tiles = new Sprite[amount];

    //Fülle das Array mit allen Tiles
    for (int i = 0; i < amount; i++)
    {
        tiles[i] = new Sprite(tileset, new IntRect( (i/w) * tileWidth,
                                                    (i/w) * tileHeight, tileWidth, tileHeight));
    }
}
```

Jetzt haben wir unser Array befüllt. Was noch fehlt ist jetzt die eigentliche map, die wir zeichnen wollen. Die map ist ein 2D Feld aus ganzzahlen, deswegen benutzen wir einen 2-Dimensionalen-Integer Array. Ein 2 Dimensionaler Array ist einfach ein Array, der 2 Indexe hat. Diese kann man wie x/y koordinaten sehen um die Werte in einem 2D Feld anzuordnen, wie in einer Tilemap. Die Map kriegen wir auch als Parameter im Konstruktor.

```
int[][] map;

// Konstruktor
public Tilemap(Screen parentScreen, Texture set, int width, int height,
               int[][] tilemap)
{
    //Speicher alle Parameter in der Klasse
    screen = parentScreen;
    tileset = set;
    tileWidth = width;
    tileHeight = height;
    map = tilemap;;
    [...]
```

Als letztes müssen wir in der Draw-Methode unsere Tiles nurnoch wie in der Map zeichnen. Dazu laufen wir durch das 2-Dimensionale Array (dafür brauch man 2 for-schleifen),und bewegen und zeichnen das Tile, das in der Map steht.

```
public void draw()
{
    //Durchlaufe jede Reihe der Map
    for (int y = 0; y < map.Length; y++)
    {
        //Durchlaufe jedes Tile der Reihe
        for (int x = 0; x < map[y].Length; x++)
        {
            //Bewege das Tile an die richtige Stelle
            tiles[map[y][x]].Position = new Vector2f(tileWidth*x,
                                                    tileHeight*y);

            //Zeiche das Tile
            screen.draw(tiles[map[y][x]]);
        }
    }
}
```

Jetzt ist unsere TileMap Klasse fertig. Wir können sie jetzt in unserem Mainscreen benutzen. Dazu brauchen wir eine Tileset Textur, deren Tilemaße und eine Map. Das Tileset findet man im Ressourcen Ordner(die Maße stehen im Dateinamen), eine Beispiel Map ist hier gegeben.

```
Tilemap tilemap;

int[][] map = new int[][] {
    new int[] {1,1,1,1,1,1},
    new int[] {1,3,2,2,3,1},
    new int[] {1,2,2,2,2,1},
    new int[] {1,2,2,2,2,1},
    new int[] {1,2,2,2,2,1},
    new int[] {1,3,2,2,3,1},
    new int[] {1,1,1,1,1,1}
};

// Setup, wird immer einmal zu Beginn eines Screens aufgerufen
// Hier Startwerte setzen!
public override void setup()
{
    Texture tileset = new Texture("assets/tileset32x32.png");

    tilemap = new Tilemap(this, tileset, 32, 32, map);
}

// Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
public override void loop()
{
    tilemap.loop();
    tilemap.draw();
}
```

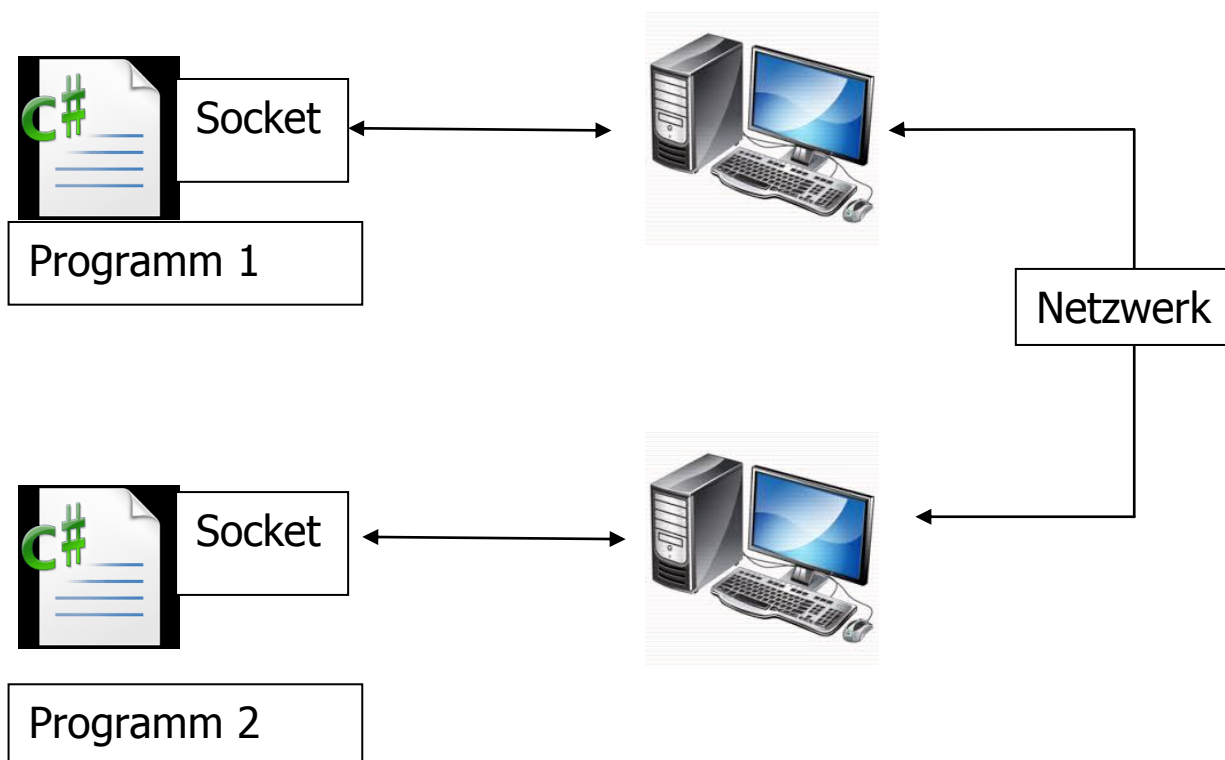
Netzwerkcommunication &

Client-Server-Modell

Will man in C# mit einem anderen Programm im selben Netzwerk kommunizieren, gibt es mehrere Möglichkeiten.

Eine davon sind sogenannte Sockets. Ein Socket kann man als Verbindung zwischen zwei Programmen betrachten. Dabei hat jedes Programm seinen eigenen Socket.

Die Programme können über ihre Sockets beide Informationen senden und empfangen.



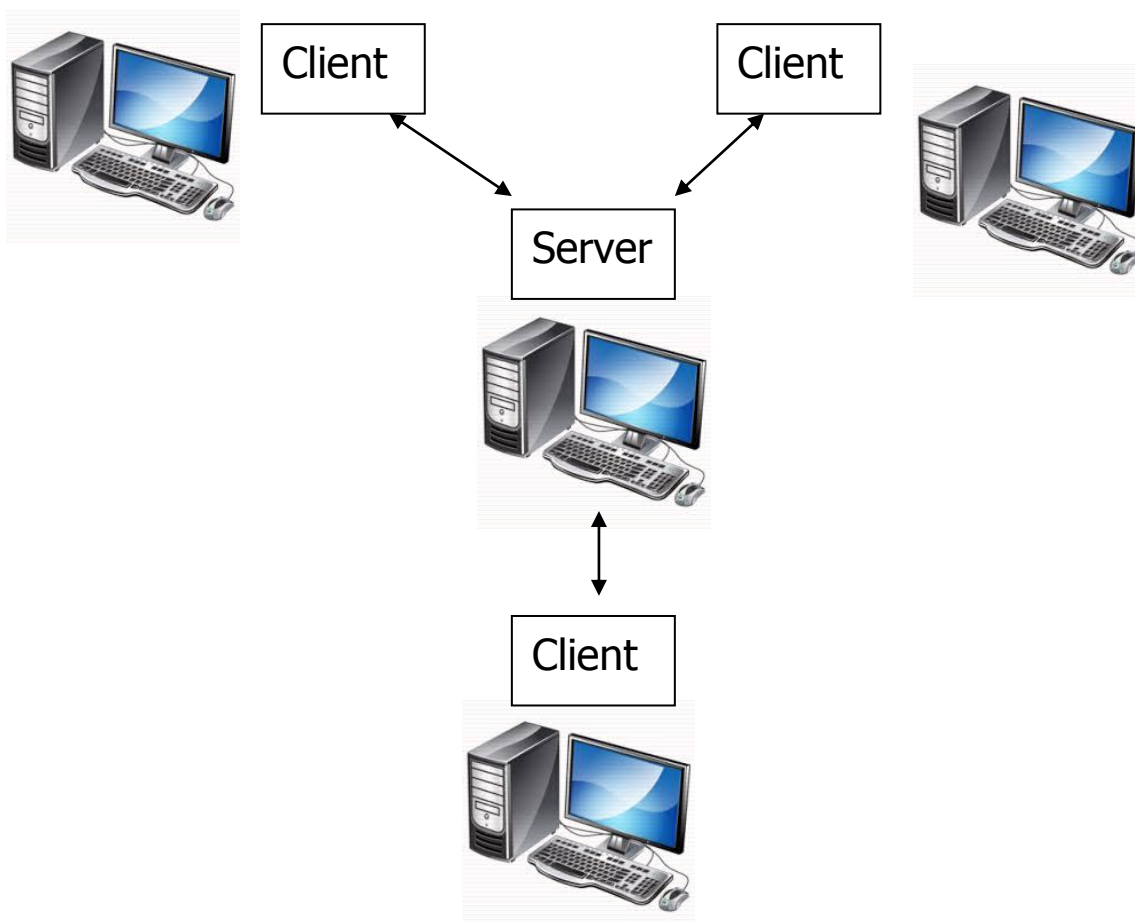
Eine einfache Verbindung zwischen 2 Programmen nennt man Peer-To-Peer Verbindung. Für moderne Spiele ist dieses Modell aber nichtmehr ausreichend. Deswegen benutzt man das sogenannte Client-Server-Modell.

Client-Server-Modell

Um bei Online-Multiplayer Spielen ein optimales Spielerlebnis zu erzeugen, muss dafür gesorgt werden, dass alle Teilnehmer in einer synchronen "Spielwelt" sind. Das heißt alle Aktionen die ein Spieler macht, müssen in den Programmen der anderen auch angezeigt werden. Und zwar möglichst gleichzeitig.

Um das zu realisieren gibt es das Client-Server-Modell. Dabei wird ein Server Computer benötigt, der je nach Programm auch Client sein kann. Der Rest der Spieler ist an sogenannten Client Computern.

Wenn ein Client eine Aktion ausführt, so soll die Information darüber vom Client zum Server geführt werden. Der Server erhält von allen Clients Informationen und errechnet dadurch den aktuellen Status des Spiels und sendet ihn anschließend allen Clients, die mit den Informationen die Spielwelt darstellen.



Online Multiplayer mit SFML

In dem Template sind zwei Klassen: NetworkClient und NetworkServer. Diese beiden Klassen enthalten eine simple Implementierung eines Client-Server-Modells.

Wir konzentrieren uns hier nur auf den NetworkClient:

Der Network-Client besitzt folgende relevante Methoden/Felder:

ConnectTo:

```
NetworkClient.connectTo("127.0.0.1", 8888);
```

Der Client versucht sich mit einem Server mit der gegebenen IP Adresse und dem gegebenen Port zu verbinden. Falls der Versuch fehlschlägt, passiert nichts.

isConnected:

```
bool isConnected = NetworkClient.isConnected
```

Ein Boolwert, der angibt ob eine funktionierende Verbindung besteht oder nicht.

read:

```
List<String> responses = NetworkClient.read();
```

Gibt eine Liste aller empfangen Nachrichten vom Server zurück.

Die Liste wird danach geleert, also wird keine Nachricht doppelt auftauchen.

send:

```
NetworkClient.send("Hello Server");
```

Sendet eine Nachricht an den Server, wenn eine Verbindung besteht.

Diese wollen wir nun nutzen, um uns mit einem Server zu verbinden und zu kommunizieren.

Zuerst, versuchen wir eine Verbindung mit dem Server aufzubauen. Die IP-Adresse eines Computers in einem Netzwerk kann über den CommandLine Befehl "ipconfig" herausgefunden werden.

Der Port sollte vom Server festgelegt sein. In der AG werden wir immer den Port 8888 benutzen.

```
public override void setup()
{
    // Initialisiere Beispiel (das führt automatisch den Konstruktor aus)
    example = new Entity(this);

    NetworkClient.connectTo("xxx.xxx.xxx.xxx", 8888);
}
```

Sobald eine Verbindung besteht, können wir über NetworkClient.read() die Nachrichten vom Server empfangen.

```
public override void loop()
{
    List<String> responses = NetworkClient.read();

    example.loop();
    example.draw();
}
```

Nun muss sich auf einen Verschlüsselungscode von Informationen geeinigt werden. Ähnlich wie bei den Savegames(siehe #13.2) müssen wir die Entity Informationen in einen String schreiben.

Wir nehmen also die Positionsdaten unserer Entity und schreiben sie in einen String:

```
String position = example.graphic.Position.X.ToString() + "x" +
example.graphic.Position.Y.ToString();
```

Das gleiche tun wir für Rotation und Size.

```
public override void loop()
{
    List<String> responses = NetworkClient.read();

    String position = example.graphic.Position.X.ToString() + "x" +
        example.graphic.Position.Y.ToString();

    String rotation = example.graphic.Rotation.ToString();

    String size = example.graphic.Size.X.ToString() + "x" +
        example.graphic.Size.Y.ToString();

    example.loop();
    example.draw();
}
```

Jetzt fügen wir die Strings mit Trennzeichen zusammen in einen langen String. Diesen senden wir dann mit NetworkClient.send() an den Server.

```
public override void loop()
{
    List<String> responses = NetworkClient.read();

    String position = example.graphic.Position.X.ToString() + "x" +
        example.graphic.Position.Y.ToString();

    String rotation = example.graphic.Rotation.ToString();

    String size = example.graphic.Size.X.ToString() + "x" +
        example.graphic.Size.Y.ToString();

    String entityData = position + "&" + rotation + "&" + size;

    NetworkClient.send(entityData);

    example.loop();
    example.draw();
}
```

Jetzt müssen wir die Daten, die wir empfangen nur noch auswerten, dafür müssen wir die Daten aus dem String lesen und daraus neue Entitäts erzeugen.

Die "responses" Liste enthält möglicherweise 0 Nachrichten, falls der Server langsam ist, oder mehr als 1 wenn der Server schneller ist. Wir müssen alle Möglichkeiten abfangen.

Als erstes Erstellen wir eine Liste von RectangleShapes. Diese enthält dann die Entitys der anderen Spieler auf dem Server.

```
// Deklariere hier Objekte oder Klassenvariablen!
Entity example;
List<RectangleShape> shapes;

public override void setup()
{
    // Initialisiere Beispiel (das führt automatisch den Konstruktor aus)
    example = new Entity(this);
    shapes = new List<RectangleShape>();

    NetworkClient.connectTo("xxx.xxx.xxx.xxx", 8888);
}
```

Wir wollen nur etwas an unserem Spiel ändern, wenn wir Nachrichten erhalten haben. Wenn responses also keine Nachricht enthält, tun wir nichts.

```
public override void loop()
{
    List<String> responses = NetworkClient.read();

    if (responses.Count > 0)
    {

    }
}
```

Falls "responses" mehr als eine Nachricht enthält, interessiert uns nur die neuste Nachricht. Also lesen wir nur diese aus:

```
if (responses.Count > 0)
{
    //Nehme nur die letzte/neuste Nachricht aus responses
    String message = responses[responses.Count - 1];
}
```

Jetzt müssen wir die einzelnen EntityDaten voneinander trennen. Dafür müssen wir das Trennzeichen kennen. Hier nehmen wir an, es wäre eine "+". Mit split('+') erzeugen wir einen Array der die EntityDaten enthält.

Diese müssen wir jetzt nurnoch alle einzeln abarbeiten mit einer for-schleife.

```
if (responses.Count > 0)
{
    //Nehme nur die letzte/neuste Nachricht aus responses
    String message = responses[responses.Count - 1];

    //Spalte die message in einzelne Entitydaten auf
    String[] data = message.Split('+');

    //Lese jede Entitydata aus
    for (int i = 0; i < data.Length; i++)
    {
    }
}
```

Jetzt spalten wir die EntityData in ihre einzelnen Komponenten auf(wir machen das Rückwärts, was wir beim senden gemacht haben).

```
//Lese jede Entitydata aus
for (int i = 0; i < data.Length; i++)
{
    String posData = data[i].Split('&')[0];
    String rotData = data[i].Split('&')[1];
    String sizeData = data[i].Split('&')[2];

    String posX = posData.Split('x')[0];
    String posY = posData.Split('x')[1];

    String sizeX = sizeData.Split('x')[0];
    String sizeY = sizeData.Split('x')[1];

}
```

Jetzt brauchen wir ein RectangleShape, das diese Informationen darstellt.

```
for (int i = 0; i < data.Length; i++)
{
    String posData = data[i].Split('&')[0];
    String rotData = data[i].Split('&')[1];
    String sizeData = data[i].Split('&')[2];

    String posX = posData.Split('x')[0];
    String posY = posData.Split('x')[1];

    String sizeX = sizeData.Split('x')[0];
    String sizeY = sizeData.Split('x')[1];

    RectangleShape shape = new RectangleShape();

    shape.Position = new Vector2f(float.Parse(posX), float.Parse(posY));

    shape.Position = new Vector2f(float.Parse(posX), float.Parse(posY));
}
```

```

        shape.Rotation = float.Parse(rotData);
        shape.Size = new Vector2f(float.Parse(sizeX), float.Parse(sizeY));
    }

```

Jetzt fügen wir dieses RectangleShape unserer Liste hinzu. Allerdings sollten wir unsere Liste immer vor der for-schleife Clearen, damit keine RectangleShapes doppelt gezeichnet werden.

```

shapes.Clear();

//Lese jede Entitydata aus
for (int i = 0; i < data.Length; i++)
{
    String posData = data[i].Split('&')[0];
    String rotData = data[i].Split('&')[1];
    String sizeData = data[i].Split('&')[2];

    String posX = posData.Split('x')[0];
    String posY = posData.Split('x')[1];

    String sizeX = sizeData.Split('x')[0];
    String sizeY = sizeData.Split('x')[1];

    RectangleShape shape = new RectangleShape();

    shape.Position = new Vector2f(float.Parse(posX), float.Parse(posY));
    shape.Rotation = float.Parse(rotData);
    shape.Size = new Vector2f(float.Parse(sizeX), float.Parse(sizeY));

    shapes.Add(shape);
}

```

Jetzt fehlt nurnoch, dass wir unsere Liste von RectangleShapes mit einer For-Schleife zeichnen.

```

for (int i = 0; i < shapes.Count; i++)
{
    draw(shapes[i]);
}

```

```

example.loop();
example.draw();

```

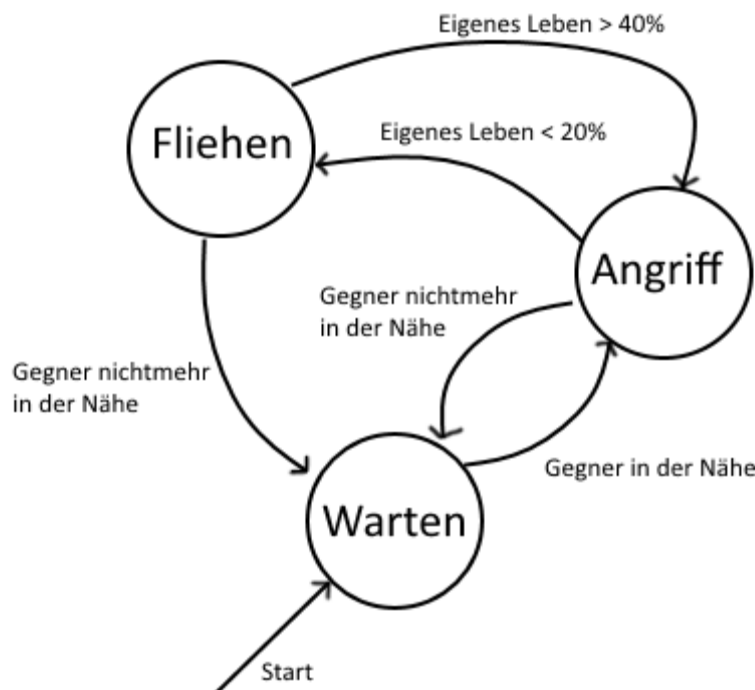
(Damit man etwas sieht, sollten wir unsere Entity bewegen können.)

Zustandsautomaten

Ein einfacher Weg um komplexes Verhalten für Entities zu programmieren ist endliche Zustandsautomaten zu verwenden.

Endliche Zustandsautomaten bestehen aus **Zuständen** und **Übergangsregeln**.

Es kann immer nur ein Zustand aktiv sein. Während ein Zustand aktiv ist, werden die Übergangsregeln überprüft. Sollte eine Regel erfüllt werden, so wird der Zustand in einen anderen gewechselt. Außerdem gibt es einen Startzustand.



Dafür muss für jeden Zustand das Verhalten programmiert werden. Diese Art von "KI", lässt sich leicht erweitern, ist simpel und verständlich. Dabei ist zu erwähnen, dass man nicht wirklich von Künstlicher Intelligenz reden kann, es aber im Sprachgebrauch so benutzt wird.

Künstliche Intelligenz

Um unsere KI jetzt in SFML zu programmieren brauchen wir natürlich eine Entity. Dazu erstelle man eine neue Entity oder nimmt eine bereits erstellte.

```
class Entity
{
    // Graphic Variable sollte in jeder Entity drin sein
    // RectangleShape, CircleShape oder Sprite
    public RectangleShape graphic;

    Screen screen;

    // Konstruktor
    public Entity(Screen parentScreen)
    {
        screen = parentScreen;
        graphic = new RectangleShape();
        graphic.Size = new Vector2f(50, 50);
        graphic.FillColor = Color.Blue;
        graphic.Position = new Vector2f(300, 300);
    }

    // Loop Methode
    public void loop()
    {

    }

    public void draw()
    {
        screen.draw(this.graphic);
    }
}
```

Jetzt müssen wir dieser Entity eine **state** Variable geben. Die State Variable **speichert den Zustand** unseres Zustandsautomaten.

```
// Graphic Variable sollte in jeder Entity drin sein
// RectangleShape, CircleShape oder Sprite
public RectangleShape graphic;

string state;

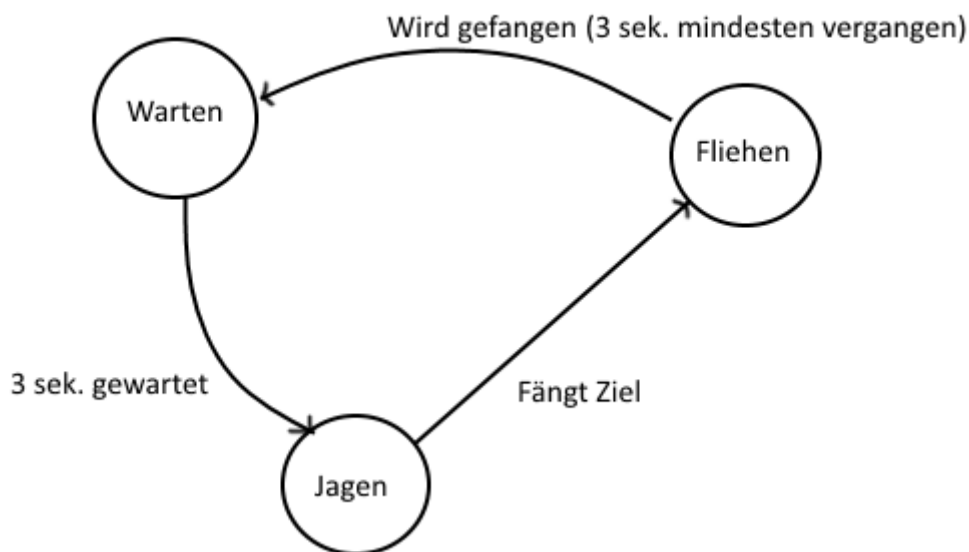
Screen screen;
```

Den state kann man als int, enum, string oder auch vielen anderen Typen speichern. Hier nehmen wir einen String.

Als nächstes machen wir uns Gedanken, wie unsere Zustände sind. Für dieses Beispiel nehmen wir folgenden Zustände:

Warten, Fliehen und Jagen

Der Sinn dieser Zustände ist: Das Objekt spielt "fangen". Es läuft weg bis man es berührt("fängt"), dann wartet es eine Zeit damit man selbst weglaufen kann und dann verfolgt es einen bis man es fängt. Danach läuft es wieder weg(wobei es ein paar Sekunden "Unfangbar" bleibt.) Der Zugehörige Zustandsautomat sieht wie folgt aus:



Der Startzustand ist hierbei das "Fliehen". Dazu setzen wir unsere State Variable entsprechend im Konstruktor auf "Fliehen". Außerdem speichern wir ein Transformable in der Klasse ab, die das Ziel darstellt.

```
public RectangleShape graphic;
Screen screen;

string state;
Transformable target;

// Konstruktor
public Entity(Screen parentScreen, Transformable target)
{
    screen = parentScreen;
    graphic = new RectangleShape();
    graphic.Size = new Vector2f(50, 50);
    graphic.FillColor = Color.Blue;
    graphic.Position = new Vector2f(300, 300);

    this.target = target;
    state = "FLIEHEN";
}
```

Jetzt wollen wir für jeden Zustand eine Methode erstellen. Also drei Methoden: fliehen, jagen und warten.

```
public void fliehen()
{

}

public void jagen()
{

}

public void warten()
{

}
```

Diese Methoden beinhalten das Verhalten in den jeweiligen Zuständen.

In der Loop überprüfen wir den Zustand und rufen immer die entsprechende Methode auf:

```
// Loop Methode
public void loop()
{
    if (state == "FLIEHEN")
    {
        fliehen();
    }
    else if (state == "JAGEN")
    {
        jagen();
    }
    else if (state == "WARTEN")
    {
        warten();
    }
}
```

In den Methoden müssen wir jetzt die Übergangsregeln definieren. Dafür brauchen wir noch zusätzlich eine Counter-Variable:

```
public RectangleShape graphic;

string state;

Screen screen;
Transformable target;

int counter;
```

Für Fliehen zählen wir den Counter jetzt hoch und sobald er über 180 ist(180 Frames = 3 sek.) überprüfen wir ob das target zu Nahe ist(hier kann man auch eine Kollisionsabfrage machen, wenn man als Target ein RectangleShape definiert), und wenn das der Fall ist, wechseln wir zu Warten und setzen den Counter zurück.

```
public void fliehen()
{
    if (counter >= 180)
    {
        // Ist die distanz kleiner als 50 pixel?
        if (utility.Functions.distance(this.graphic, target) < 50)
        {
            state = "WARTEN";
            counter = 0;
        }
    }
    else
    {
        counter++;
    }
}
```

In jagen überprüfen wir auch die Distanz und wechseln zu fliehen, sobald wir nahe genug sind.

```
public void jagen()
{
    // Ist die distanz kleiner als 50 pixel?
    if (utility.Functions.distance(this.graphic, target) < 50)
    {
        state = "FLIEHEN";
    }
}
```

In Warten lassen wir einfach den Counter bis 180 zählen.

```
public void warten()
{
    if (counter >= 180)
    {
        state = "JAGEN";
        counter = 0;
    }
    else
    {
        counter++;
    }
}
```

Jetzt haben wir die Übergangsregelungen und die States eingebaut. Jetzt fehlt noch das Verhalten.

Dazu lassen wir das Objekt in fliehen immer vom Target wegbewegen(aber nie über die Bildschirmgrenzen hinaus).

```
public void fliehen()
{
    //Der Einheitsvektor in Flieh-Richtung
    Vector2f dir = utility.Functions.getUnitVector(this.graphic.Position -
    target.Position);

    this.graphic.Position += dir * 2; //Mit Geschwindigkeit 2 weg vom Target

    //Gehe nie über Bildschirm hinaus
    if (this.graphic.Position.X > 800)
    {
        this.graphic.Position = new Vector2f(800, this.graphic.Position.Y);
    }

    if (this.graphic.Position.X < 0)
    {
        this.graphic.Position = new Vector2f(0, this.graphic.Position.Y);
    }

    if (this.graphic.Position.Y > 600)
    {
        this.graphic.Position = new Vector2f(this.graphic.Position.X, 600);
    }
    if (this.graphic.Position.Y < 0)
    {
        this.graphic.Position = new Vector2f(this.graphic.Position.X, 0);
    }

    if (counter >= 180)
    {
        // Ist die distanz kleiner als 50 pixel?
        if (utility.Functions.distance(this.graphic, target) < 50)
        {
            state = "WARTEN";
            counter = 0;
        }
    }
    else
    {
        counter++;
    }
}
```

Für das Jagen machen wir dasselbe, nur dass wir uns auf das Target zubewegen(ohne bildschirmgrenzen)

```
public void jagen()
{
    //Der Einheitsvektor in Jag-Richtung
    Vector2f dir = utility.Functions.getUnitVector(target.Position -
    this.graphic.Position);

    this.graphic.Position += dir * 2; //Mit Geschwindigkeit 2 das target jagen

    // Ist die distanz kleiner als 50 pixel?
    if (utility.Functions.distance(this.graphic, target) < 50)
    {
        state = "FLIEHEN";
    }
}
```

Die Warten Funktion kann man so lassen.

Jetzt die Entity wie immer in Mainscreen hinzufügen und als zweiten Parameter den Player angeben.

```
class MainScreen : Screen
{
    // Deklariere hier Objekte oder Klassenvariablen!

    Entity entity;
    RectangleShape player;

    // Setup, wird immer einmal zu Beginn eines Screens aufgerufen
    // Hier Startwerte setzen!
    public override void setup()
    {

        player = new RectangleShape();
        player.Size = new Vector2f(50, 50);
        player.Position = new Vector2f(400, 400);
        player.FillColor = Color.Green;
        player.Origin = player.Size * 0.5f;

        entity = new Entity(this, player);

    }

    // Loop, wird jeden Frame (60 mal die Sekunde) aufgerufen
    public override void loop()
    {

        if (Keyboard.IsKeyPressed(Keyboard.Key.W))
        {
            player.Position += new Vector2f(0, -3);
        }

        if (Keyboard.IsKeyPressed(Keyboard.Key.A))
        {
            player.Position += new Vector2f(-3, 0);
        }

        if (Keyboard.IsKeyPressed(Keyboard.Key.S))
        {
            player.Position += new Vector2f(0, 3);
        }

        if (Keyboard.IsKeyPressed(Keyboard.Key.D))
        {
            player.Position += new Vector2f(3, 0);
        }

        draw(player);

        entity.loop();
        entity.draw();

    }
}
```