Übungen - Bildgenierung

Übung 09.

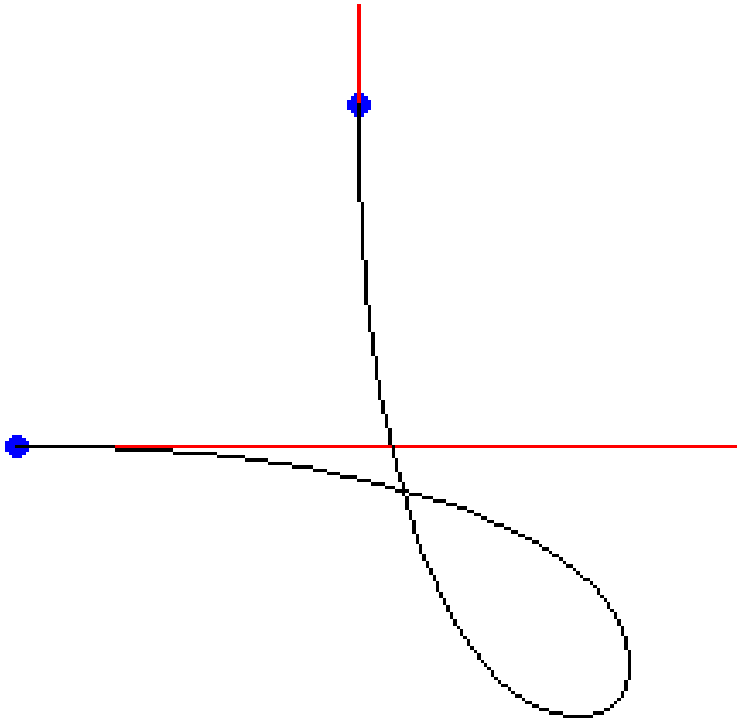
Jose Jimenez

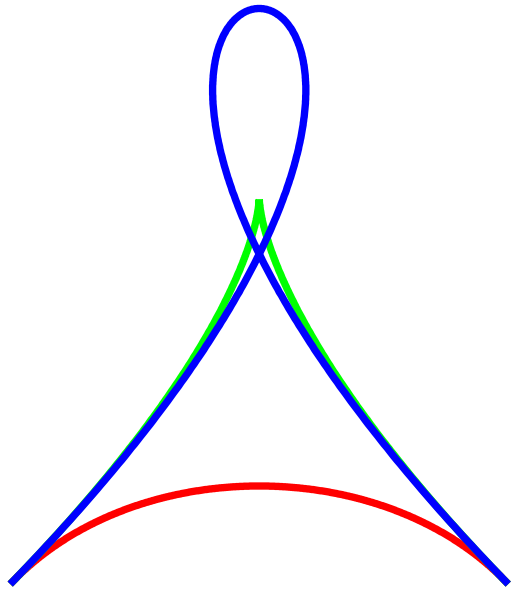
Angewandte Informatik
Bergische Universität Wuppertal

January 9, 2024

Table of Contents

Hermite-Kurven Schleife





Hermite-Kurven Animation

1. Legen Sie die beiden Endpunkte auf die Gerade $y = x$,
2. verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten).
3. Variieren Sie die Längen der Tangentenvektoren.

Hermite-Kurven Animation

1. Legen Sie die beiden Endpunkte auf die Gerade $y = x$,
2. verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten).
3. Variieren Sie die Längen der Tangentenvektoren.

```
int maindraw(){
  Drawing pic(600, 600);
  DPoint2D p1, p4, r1, r4;
  int b = 250;
  int c = 100;
  pic.show();
  //Nummer 1:
  p1 = DPoint2D(b, b);
  p4 = DPoint2D(b+c, b+c);
  ...
```

Hermite-Kurven Animation

1. Legen Sie die beiden Endpunkte auf die Gerade $y = x$,
2. verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten).
3. Variieren Sie die Längen der Tangentenvektoren.

Hermite-Kurven Animation

1. Legen Sie die beiden Endpunkte auf die Gerade $y = x$,
2. verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten).
3. Variieren Sie die Längen der Tangentenvektoren.

```
int maindraw(){
  Drawing pic(600, 600);
  DPoint2D p1, p4, r1, r4;
  pic.show();
  p1 = DPoint2D(b, b);
  p4 = DPoint2D(b+c, b+c);

  //Nummer 2 und 3:
  for (int rho = -600; rho <= 1000; rho += 10) {
      r1 = DPoint2D( rho, 0 );
      r4 = DPoint2D( 0, rho );
      ...
```

Hermite-Kurven Animation

1. Legen Sie die beiden Endpunkte auf die Gerade $y = x$,
2. verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten).
3. Variieren Sie die Längen der Tangentenvektoren.

```
int maindraw(){
  Drawing pic(600, 600);
  DPoint2D p1, p4, r1, r4;
  pic.show();
  p1 = DPoint2D(b, b);
  p4 = DPoint2D(b+c, b+c);

  for (int rho = -600; rho <= 1000; rho += 10){
      r1 = DPoint2D( rho, 0 );
      r4 = DPoint2D( 0, rho );
      //Mahlen!!!
      pic =255;
      maleHermiteKurve(pic, p1, p4, r1, r4, 50);
```

Bézier-Kurven und B-Splines

Für die **Hermite-Kurven**:

$$Q(t) = t^3 \underbrace{(2P_1 - 2P_4 + R_1 + R_4)}_{c_0} + t^2 \underbrace{(-3P_1 + 3P_4 - 2R_1 - R_4)}_{c_1} + t \underbrace{(R_1)}_{c_2} + \underbrace{P}_{c_3}$$

$$Q(t) = c_0 t^3 + c_1 t^2 + c_2 t + c_3 t$$

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

c_i sind Vektoren, d.h., $c_i(x, y)$.

Bézier-Kurven und B-Splines

Für die **Hermite-Kurven**:

$$Q(t) = c_0 t^3 + c_1 t^2 + c_2 t + c_3 t$$
$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

c_i sind Vektoren, d.h., $c_i(x, y)$.

Bézier-Kurven und B-Splines sind auch Polynome, d.h. man kann sie so beschreiben

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Bézier-Kurven und B-Splines

dann sieht der Code genauso aus wie vorher.

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Bézier-Kurven und B-Splines

dann sieht der Code genauso aus wie vorher.

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

```
for (i = 1; i <= n; ++i){
  t = ninv * i;
  anf = end;
  end.x = ((cx[0] * t + cx[1]) * t + cx[2]) * t + cx[3];
  end.y = ((cy[0] * t + cy[1]) * t + cy[2]) * t + cy[3];
  pic.drawLine(round(anf), round(end));
  }
```

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Die antwort ist: Hermite-Form

- **Bézier-Kurven** (9-6)

$$Q(t) = G_B M_B T = C_{Be} T$$

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Die antwort ist: Hermite-Form

- **Bézier-Kurven** (9-6)

$$Q(t) = G_B M_B T = C_{Be} T$$

- **B-Splines** (9-16)

$$Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$$

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Die antwort ist: Hermite-Form

- **Bézier-Kurven** (9-6)

$$Q(t) = G_B M_B T = C_{Be} T$$

- **B-Splines** (9-16)

$$Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T$$

und M sind die Basismatrizen.

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Die antwort ist: Hermite-Form

- **Bézier-Kurven** (9-6)

$$Q(t) = G_B M_B T = C_{Be} T$$

- **B-Splines** (9-16)

$$Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T$$

und M sind die Basismatrizen.

Bézier-Kurven und B-Splines

Die Frage ist denn natürlich: was sind die Koeffizienten c_i?

$$Q(t) = ((c_0 t + c_1)t + c_2)t + c_3$$

Die antwort ist: Hermite-Form

- **Bézier-Kurven** (9-6)

$$Q(t) = G_B M_B T = C_{Be} T$$

- **B-Splines** (9-16)

$$Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T$$

und M sind die Basismatrizen.

Bézier-Kurven und B-Splines

- **Bézier-Kurven** $Q(t) = G_B M_B T = C_{Be} T$
- **B-Splines** $Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T \tag{2}$$

und M sind die Basismatrizen.

$$M_B = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 4 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad M_{BS} = \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

```
for (k = 3; k <= m; k += 3) {
    // Die Kurve besteht dann aus m/3 einzelnen Kurvenstücken.
    }
for (k = 3; k <= m; ++k){
    /* "die Funktion die den zu p_0,...,p_m gehörenden
        B-Spline malt."*/
    }
```

Bézier-Kurven und B-Splines

- **Bézier-Kurven** $Q(t) = G_B M_B T = C_{Be} T$
- **B-Splines** $Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T \tag{3}$$

und M sind die Basismatrizen.

$$M_B = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 4 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad M_{BS} = \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

```
for (k = 3; k <= m; k += 3) { //Bézier
    cx[0] = -p[k - 3].x + 3 * p[k - 2].x - 3 * p[k - 1].x + p[k].x;
    cx[1] = 3 * p[k - 3].x - 6 * p[k - 2].x + 3 * p[k - 1].x;
    cx[2] = -3 * p[k - 3].x + 3 * p[k - 2].x;
    cx[3] = p[k - 3].x;
    }
```

Bézier-Kurven und B-Splines

- **Bézier-Kurven** $Q(t) = G_B M_B T = C_{Be} T$
- **B-Splines** $Q_i(t) = G_{BS_i} M_{BS} T_i = C_B T_i$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T \tag{4}$$

und M sind die Basismatrizen.

$$M_B = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 4 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad M_{BS} = \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

```
for (k = 3; k <= m; ++k) {
  cx[0] = (-p[k - 3].x + 3 * p[k - 2].x - 3 * p[k - 1].x + p[k].x) /
↪  6.0;
  cx[1] = (3 * p[k - 3].x - 6 * p[k - 2].x + 3 * p[k - 1].x) / 6.0;
  cx[2] = (-3 * p[k - 3].x + 3 * p[k - 1].x) / 6.0;
  cx[3] = (p[k - 3].x + 4 * p[k - 2].x + p[k - 1].x) / 6.0;
    }
```

Catmull-Rom-Splines (ganz ähnlich)

- **Catmull-Rom-Splines** $Q(t) = G_B M_{CR} T = C_{CR} T$

Hier:

$$G = (P_{i-3}, P_{i-2}, P_{i-1}, P_i) \qquad \text{und} \quad T = (t^3, t^2, t, 1)^T \tag{5}$$

und M sind die Basismatrizen.

$$\mathrm{M}_{CR} = \frac{1}{2} \begin{pmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix}$$

```
for (k = 3; k <= m; ++k){
  cx[0] = (-p[k - 3].x + 3 * p[k - 2].x - 3 * p[k - 1].x + p[k].x) /
↪  2.0;
  cx[1] = (2 * p[k - 3].x - 5 * p[k - 2].x + 4 * p[k - 1].x -
↪  p[k].x) / 2.0;
  cx[2] = (-p[k - 3].x + p[k - 1].x) / 2.0;
  cx[3] = p[k - 2].x;
```

Objekterzeugung mit Grammatiken, Blumenwiese

Ok, wir brauchen wieder eine Animation. Und wir brauchen auch einige Pflanzen.

Objekterzeugung mit Grammatiken, Blumenwiese

Ok, wir brauchen wieder eine Animation. Und wir brauchen auch einige Pflanzen.Wie üblich können wir Vektoren verwenden.

```
const int anzahlBilder = 30;
int maindraw()
{
  vector<Drawing> pics(anzahlBilder);
  vector<Blume> blumen{  .... } ?


  Drawing pic(1400, 800, 0);
  pic.show();
}
```

Blumen Class: darum kümmern wir uns später

Objekterzeugung mit Grammatiken, Blumenwiese

dann wollen wir so etwas in einer for-Schleife machen...

```
const int anzahlBilder = 30;
int maindraw()
{
  ...
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background malen

      // 2. Blumen weiterentwickeln lassen

      // 3. Blume malen

      // 4. Bild im Vektor speichern
  }
  Drawing::makeanim(pics, "blume", "gif", 30);

}
```

Objekterzeugung mit Grammatiken, Blumenwiese

1 und **4** sind einfach und wir haben die schon gemacht.

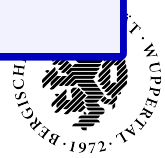
Objekterzeugung mit Grammatiken, Blumenwiese

1 und **4** sind einfach und wir haben die schon gemacht.

```
int maindraw()
{
  ...
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background malen
      pic = DrawColour(227, 247, 227); // hellgrün

      // 2. Blumen weiterentwickeln lassen
      // 3. Blume malen

      // 4. Bild im Vektor speichern
       pics[i] = pic;
  }
  Drawing::makeanim(pics, "blume", "gif", 30);
}
```



Objekterzeugung mit Grammatiken, Blumenwiese

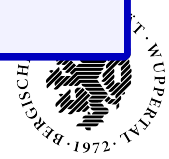
1 und **4** sind einfach und wir haben die schon gemacht.

```
int maindraw()
{
  ...
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background malen
      pic = DrawColour(227, 247, 227); // hellgrün

      // 2. Blumen weiterentwickeln lassen
      // 3. Blume malen

      // 4. Bild im Vektor speichern
       pics[i] = pic;
  }
  Drawing::makeanim(pics, "blume", "gif", 30);
}
```

2 und **3**... sie könnten Methoden der Klasse Blume sein.



Objekterzeugung mit Grammatiken, Blumenwiese

Eine Blume sollte eine Position, eine Ableitung und 2 Methoden haben....

Objekterzeugung mit Grammatiken, Blumenwiese

Erinnerung:
Ein Zweig **z** mit einer Knospe **K** an der Spitze kann sich auf verschiedene Arten entwickeln:

1. Die Knospe kann absterben.
2. Die Knospe kann eine Blüte **B** erzeugen und dann absterben.
3. Die Knospe kann ein Blatt **l** erzeugen.
4. Die Knospe kann austreiben:
   - Neben der Knospe entstehen evtl. eine oder zwei weitere Knospen.
   - Die ursprüngliche Knospe wächst weiter zu einem Zweig.
   - An der Spitze dieses Zweigs ist wieder eine Knospe.

Objekterzeugung mit Grammatiken, Blumenwiese

Klase Blume: Eine Blume sollte eine Position, eine Ableitung und 2 Methoden haben....

Objekterzeugung mit Grammatiken, Blumenwiese

Klase Blume: Eine Blume sollte eine Position, eine Ableitung und 2 Methoden haben....

```
const int anzahlBilder = 30;
class Blume{
private:
  DPoint2D position;
  string ableitung;
public:
  Blume(DPoint2D pos){
    position = pos;
    ableitung = "zK";  // Warum?
  }
  void weiter();  // Blumen weiterentwickeln lassen
  void maleMich(Drawing& pic);   // Blume malen

}
```

Lass uns zuerst "**weiter**" implementieren.

Objekterzeugung mit Grammatiken, Blumenwiese

```
void weiter(){

}
```

"Wählen Sie bei mehreren Ableitungsmöglichkeiten jeweils passende Wahrscheinlichkeiten, so dass sich insbesondere ein sinnvoller Anteil an Knospen weiterentwickelt".

Objekterzeugung mit Grammatiken, Blumenwiese

Startwort und erinnerung

Nichtterminale:	**K**, **B**
Terminale:	**z**, **I**, **(**, **)**, **[**, **]**
Produktionen:	$\mathbf{K} \rightarrow \mathbf{K} \mid \epsilon \qquad \epsilon =:$ "nichts“
	$\mathbf{K} \rightarrow \mathbf{B}$
	$\mathbf{B} \rightarrow \mathbf{B} \mid \epsilon$
	$\mathbf{K} \rightarrow \mathbf{I}$
	$\mathbf{K} \rightarrow \mathbf{zK} \mid \mathbf{(K)zK} \mid \mathbf{[K]zK} \mid \mathbf{(K)[K]zK}$
Startwort:	**zK**

Z = Zweig, K= Knospe, B= Blute, I=blatt.

Objekterzeugung mit Grammatiken, Blumenwiese

Startwort und erinnerung

Nichtterminale:	**K**, **B**
Terminale:	**z**, **I**, **(**, **)**, **[**, **]**
Produktionen:	$\mathbf{K} \rightarrow \mathbf{K} \mid \epsilon \qquad \epsilon =:$ "nichts“
	$\mathbf{K} \rightarrow \mathbf{B}$
	$\mathbf{B} \rightarrow \mathbf{B} \mid \epsilon$
	$\mathbf{K} \rightarrow \mathbf{I}$
	$\mathbf{K} \rightarrow \mathbf{zK} \mid \mathbf{(K)zK} \mid \mathbf{[K]zK} \mid \mathbf{(K)[K]zK}$
Startwort:	**zK**

Z = Zweig, K= Knospe, B= Blute, I=blatt.

```
Blume(DPoint2D pos)  {
    position = pos;
    ableitung = "zK";   //Warum?
  }
```

Objekterzeugung mit Grammatiken, Blumenwiese

Es gibt 2 Fälle, in denen wir Produktionen haben: **K** und **B**.

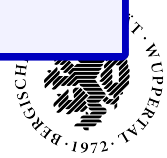
Nichtterminale:	**K**, **B**
Terminale:	**z**, **I**, **(**, **)**, **[**, **]**
Produktionen:	$\mathbf{K} \rightarrow \mathbf{K} \mid \epsilon \qquad \epsilon =:$ "nichts"
	$\mathbf{K} \rightarrow \mathbf{B}$
	$\mathbf{B} \rightarrow \mathbf{B} \mid \epsilon$
	$\mathbf{K} \rightarrow \mathbf{I}$
	$\mathbf{K} \rightarrow \mathbf{zK} \mid \mathbf{(K)zK} \mid \mathbf{[K]zK} \mid \mathbf{(K)[K]zK}$
Startwort:	**zK**

Z = Zweig, K= Knospe, B= Blute, I= blatt.

Objekterzeugung mit Grammatiken, Blumenwiese

Es gibt 2 Fälle, in denen wir Produktionen haben: **K** und **B**.

```
void weiter(){
    string neu = "";
    double r;
    for (auto z : ableitung)
      switch (z){
          case 'K':
            ...
          case 'B':
            ...
          default:
            neu += z;
        }
    ableitung = neu;
}
```



Objekterzeugung mit Grammatiken, Blumenwiese

"Wählen Sie bei mehreren **Ableitungsmöglichkeiten** jeweils passende Wahrscheinlichkeiten, so dass sich insbesondere ein sinnvoller Anteil an Knospen weiterentwickelt".

Nichtterminale:	$\mathbf{K}$, $\mathbf{B}$
Terminale:	$\mathbf{z}$, $\mathbf{I}$, $\mathbf{(}$, $\mathbf{)}$, $\mathbf{[}$, $\mathbf{]}$
Produktionen:	$\mathbf{K} \rightarrow \mathbf{K} \mid \epsilon \qquad \epsilon =:$ "nichts"
	$\mathbf{K} \rightarrow \mathbf{B}$
	$\mathbf{B} \rightarrow \mathbf{B} \mid \epsilon$
	$\mathbf{K} \rightarrow \mathbf{I}$
	$\mathbf{K} \rightarrow \mathbf{zK} \mid \mathbf{(K)zK} \mid \mathbf{[K]zK} \mid \mathbf{(K)[K]zK}$
Startwort:	$\mathbf{zK}$

Objekterzeugung mit Grammatiken, Blumenwiese

"Wählen Sie bei mehreren **Ableitungsmöglichkeiten** jeweils passende Wahrscheinlichkeiten, so dass sich insbesondere ein sinnvoller Anteil an Knospen weiterentwickelt".

Nichtterminale:	**K**, **B**
Terminale:	**z**, **I**, **(**, **)**, **[**, **]**
Produktionen:	$\mathbf{K} \rightarrow \mathbf{K} \mid \epsilon \qquad \epsilon =:$ "nichts"
	$\mathbf{K} \rightarrow \mathbf{B}$
	$\mathbf{B} \rightarrow \mathbf{B} \mid \epsilon$
	$\mathbf{K} \rightarrow \mathbf{I}$
	$\mathbf{K} \rightarrow \mathbf{zK} \mid \mathbf{(K)zK} \mid \mathbf{[K]zK} \mid \mathbf{(K)[K]zK}$
Startwort:	**zK**

Wir müssen also eine Zufallszahl zwischen 0 und 1 erzeugen, um zwischen diesen Optionen zu wählen.

Objekterzeugung mit Grammatiken, Blumenwiese

switch(z) ...

```
...
case 'K':
  r = rnd01();
  if (r < 0.2) // 20%
  neu += 'K';
  else if (r < 0.25) // 5%
  break; // ϵ
  else if (r < 0.375) // 12,5%
  neu += 'B';
  else if (r < 0.5)  // 12,5%
  neu += 'l';
 ...
```

```
...
 else if (r < 0.625)  // 12,5%
  neu += "zK";
  else if (r < 0.75)  // 12,5%
  neu += "(K)zK";
  else if (r < 0.875)  // 12,5%
  neu += "[K]zK";
  else  // 12,5%
  neu += "(K)[K]zK";
  break;
case 'B':
...
```

Objekterzeugung mit Grammatiken, Blumenwiese

```
void weiter(){
string neu = "";
    double r;
    for (auto z : ableitung)
      switch (z)
        {
          case 'K':
            ...
          case 'B':
            r = rnd01();
            if (r < 0.5)
              neu += 'B';  //  und ϵ für r ≥ 0.5
            break;
          default:
            neu += z;
        }
    ableitung = neu;
}
```

Objekterzeugung mit Grammatiken, Blumenwiese

Klase Blume: Eine Blume sollte eine Position, eine Ableitung und 2 Methoden haben....

```
const int anzahlBilder = 30;
class Blume{
private:
  DPoint2D position;
  string ableitung;
public:
  Blume(DPoint2D pos){
    position = pos;
    ableitung = "zK";  // Warum?
  }
  void weiter();  // Blumen weiterentwickeln lassen
  void maleMich(Drawing& pic);   // Blume malen

}
```

Jetz, male Funktion:

Objekterzeugung mit Grammatiken, Blumenwiese

Jetz, male Funktion:

```
void maleMich(Drawing& pic) const{
    // ??
 }
```

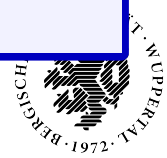
Objekterzeugung mit Grammatiken, Blumenwiese

```
void maleMich(Drawing& pic) const{
    for (auto z : ableitung)
      switch (z) {
          case 'K': // Knospe: kleiner blauer Kreis EINFACH
          case 'B': // Blüte: 3 rote Kreise, 2 klein, einer größer
          case 'z': // Zweig: braune Linie
          case 'l': // Blatt: größerer grüner Kreis
          case '(': // Neigung nach Links
          case '[': // Neigung nach Rechts
          case ')':
          case ']':
    }
  }
```

Objekterzeugung mit Grammatiken, Blumenwiese

Knospe, Blüte und Blatt sind endlich und einfach.

```
void maleMich(Drawing& pic) const{
    DPoint2D pos = position;
    for (auto z : ableitung)
      switch (z)
        {
          case 'K': // Knospe: kleiner blauer Kreis
            pic.drawCircle(pos, 3, true, DrawColour(95, 95, 255),
                           DrawColour(197, 197, 255));
            break;
            case 'B': // Blüte: 3 rote Kreise, 2 klein, einer größer
            ...
            case 'l': // Blatt: größerer grüner Kreis  EINFACH
            ...
    }
  }
```



Objekterzeugung mit Grammatiken, Blumenwiese

Bis hier:

```
void maleMich(Drawing& pic) const{
    for (auto z : ableitung)
      switch (z) {
          case 'K':           // pic.drawCircle
          case 'B':           // 3 mal pic.drawCircle
          case 'z': // Zweig: braune Linie
          case 'l':           // pic.drawCircle
          case '(': // Neigung nach Links
          case '[': // Neigung nach Rechts
          case ')':
          case ']':
    }
  }
```

wir werden uns jetzt um den Fall "Zweig" kümmern

Objekterzeugung mit Grammatiken, Blumenwiese

```
void maleMich(Drawing& pic) const{
    DPoint2D pos = position;
    DPoint2D posneu;
    double winkel(M_PI / 2);  int level = 1; //Rekursionsebenen

    for (auto z : ableitung)
      switch (z) {
        case 'z': //braune Linie, kürzer für tiefere
↪   Rekursionsebenen
            posneu
              = pos + 14 * exp(1 / level) * DPoint2D(cos(winkel),
↪   sin(winkel));
            pic.drawLine(pos, posneu, DrawColour(164, 70, 8));
            pos = posneu;
            break;
    }
  }
```

Objekterzeugung mit Grammatiken, Blumenwiese

Bis hier:

```
void maleMich(Drawing& pic) const{
    for (auto z : ableitung)
      switch (z) {
          case 'K':          // pic.drawCircle
          case 'B':          // 3 mal pic.drawCircle
          case 'z':          // pic.drawLine(pos, posneu)
          case 'l':          // pic.drawCircle
          case '(': // Neigung nach Links
          case '[': // Neigung nach Rechts
          case ')':
          case ']':
    }
  }
```

wir werden uns jetzt um den Fall "Neigung nach Links" kümmern

Objekterzeugung mit Grammatiken, Blumenwiese

```
void maleMich(Drawing& pic) const{
    stack<DPoint2D> posstack;   //<------ NEU
    stack<double> winkelstack;  //<------ NEU
    int level = 1; //Rekursionsebenen

    for (auto z : ableitung)
      switch (z) {
        ...
        case '(': // Neigung nach Links
            posstack.push(pos);
            winkelstack.push(winkel);
            winkel += M_PI / 8 * pow(0.9, level - 1);   //kleiner
            ++level;
            break;
        ...
    }
  }
```

Objekterzeugung mit Grammatiken, Blumenwiese

```
void maleMich(Drawing& pic) const{
    ...
    for (auto z : ableitung)
      switch (z) {
        ...
        case '(': // Neigung nach Links
            posstack.push(pos);
            winkelstack.push(winkel);
            winkel += M_PI / 8 * pow(0.9, level - 1);   //kleiner
            ++level;
            break;
         case ')':
            pos = posstack.top(); //letzte gespeicherte Position
            posstack.pop();  //Entfernen
            winkel = winkelstack.top();
            winkelstack.pop();
            --level;
    }
  }
```

Objekterzeugung mit Grammatiken, Blumenwiese

Bis hier:

```
void maleMich(Drawing& pic) const{
    for (auto z : ableitung)
      switch (z) {
          case 'K':          // pic.drawCircle
          case 'B':          // 3 mal pic.drawCircle
          case 'z':          // pic.drawLine(pos, posneu)
          case 'l':          // pic.drawCircle
          case '(':          // Neigung nach Links --> Stack
          case '[': // Neigung nach Rechts
          case ')':
          case ']':
    }
  }
```

wir werden uns jetzt um den Fall "Neigung nach Rechts" kümmern.
Analog zu Neigung nach Links

Objekterzeugung mit Grammatiken, Blumenwiese

```
void maleMich(Drawing& pic) const{
    ...
    for (auto z : ableitung)
      switch (z) {
        ...
        case '[': // Neigung nach Rechts
            posstack.push(pos);
            winkelstack.push(winkel);
            winkel -= M_PI / 8 * pow(0.9, level - 1);
            ++level;
            break;
         case ')':
         case ']':
            pos = posstack.top(); //letzte gespeicherte Position
            posstack.pop();  //Entfernen
            winkel = winkelstack.top();
            winkelstack.pop();
            --level;
    }
```

Objekterzeugung mit Grammatiken, Blumenwiese

Back to main:

Objekterzeugung mit Grammatiken, Blumenwiese

Back to main:

```
const int anzahlBilder = 30;
int maindraw()
{
  ...
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background malen

      // 2. Blumen weiterentwickeln lassen

      // 3. Blume malen

      // 4. Bild im Vektor speichern
  }
  Drawing::makeanim(pics, "blume", "gif", 30);

}
```

```
int maindraw(){
  vector<Drawing> pics(anzahlBilder);
  vector<Blume> blumen{ DPoint2D(100, 10), DPoint2D(300, 10),
  DPoint2D(500, 10), DPoint2D(700, 10),
  DPoint2D(900, 10), DPoint2D(1100, 10) } ;

  Drawing pic(1400, 800, 0);
  pic.show();
  for (int i = 0; i < anzahlBilder; ++i){
      pic = DrawColour(227, 247, 227); // hellgrün
      for (auto& b : blumen)
        b.weiter();   //Blumen weiterentwickeln
      for (auto& b : blumen)
        b.maleMich(pic); Blume malen

      pics[i] = pic; //Bild im Vektor speichern
    }
  Drawing::makeanim(pics, "blume", "gif", 30);
  return 0;
}
```

Partikelsysteme, Feuer

1. Ok, wir brauchen wieder eine Animation.

Partikelsysteme, Feuer

1. Ok, wir brauchen wieder eine Animation.
2. Und wir brauchen auch **viele** Partikeln.

Partikelsysteme, Feuer

1. Ok, wir brauchen wieder eine Animation.
2. Und wir brauchen auch **viele** Partikeln.

Wie üblich können wir Vektoren verwenden.

```
int maindraw()
{
  vector<Drawing> pics(anzahlBilder);
  vector<Partikel> p;  //(???)

  Drawing pic(600, 600, 0);

}
```

Partikelsysteme, Feuer

Dann wollen wir in einer for-Schleife so etwas tun...

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background

      // 2. neue Partikel erzeugen

      // 3. Partikel bewegen

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern

    }
}
```

Hilfreiche Konstanten

Hilfreiche Konstanten

```
const int anzahlBilder = 500;
const int neuProBild = 1000;
const double g = 0.003;          // Gravitation
const DrawColour farbverlauf[] =
                { DrawColour(255, 255, 255), // weiß
                  DrawColour(255, 255, 0),   // gelb
                  DrawColour(255, 05, 0),    // rot
                  DrawColour(0, 0, 0) };     // schwarz
int maindraw(){
  ...
}
```

Ok, zurück zu mandraw()

Partikelsysteme, Feuer

was können wir schon tun?

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background

      // 2. neue Partikel erzeugen

      // 3. Partikel bewegen

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern

    }
}
```

Partikelsysteme, Feuer

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background
        pic = DrawColour(90, 180, 90); // grün
      // 2. neue Partikel erzeugen
        for (int j = 0; j < neuProBild; ++j)
        p.push_back(Partikel(.....));

      // 3. Partikel bewegen

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern
       pics[i] = pic;
    }
}
```

Partikelsysteme, Feuer

Also, wir kümmern uns darum:

3. Partikel bewegen
4. verschwundene Partikel löschen
5. Partikel malen

Partikelsysteme, Feuer

Also, wir kümmern uns darum:

3. Partikel bewegen
4. verschwundene Partikel löschen
5. Partikel malen

Noch einmal: Wir brauchen so etwas wie eine "**Partikel**"-**Klasse**.

Klassenattribute

```
class Partikel
{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;

  // Alter des Partikels: 0 bis 4, Start bei 0.8 (weiße Phase soll
↪  kürzer sein)
  double gluehzustand;
};
```

Partikelsysteme, Feuer

Methoden:

Partikelsysteme, Feuer

Methoden:

- Konstruktor

Partikelsysteme, Feuer

Methoden:

- Konstruktor
- Bewegung

Partikelsysteme, Feuer

Methoden:

- Konstruktor
- Bewegung
- Farbe (gluezustand)

Partikelsysteme, Feuer

Methoden:

- Konstruktor
- Bewegung
- Farbe (gluezustand)
- mahlen

Partikelsysteme, Feuer

Methoden:

- Konstruktor
- Bewegung
- Farbe (gluezustand)
- mahlen
- und Verschwinden der Partikel.

Partikelsysteme, Feuer

Methoden: Konstruktor, Bewegung, Farbe, mahlen und Verschwinden der Partikel.

```
class Partikel{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;
  double gluehzustand;
public:
  Partikel(DPoint2D startpos); // Konstruktor

};
```

Partikelsysteme, Feuer

Methoden: Konstruktor, Bewegung, Farbe, mahlen und Verschwinden der Partikel.

```
class Partikel{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;
  double gluehzustand;
public:
  Partikel(DPoint2D startpos);
  void weiter(); //bewegen
};
```

Partikelsysteme, Feuer

Methoden: Konstruktor, Bewegung, Farbe, mahlen und Verschwinden der Partikel.

```
class Partikel{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;
  double gluehzustand;
public:
  Partikel(DPoint2D startpos);
  void weiter();
  bool verschwunden() const { return gluehzustand >= 4; }
};
```

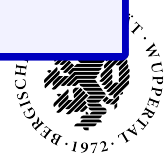
Partikelsysteme, Feuer

Methoden: Konstruktor, Bewegung, Farbe, mahlen und Verschwinden der Partikel.

```
class Partikel{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;
  double gluehzustand;
public:
  Partikel(DPoint2D startpos);
  void weiter();
  bool verschwunden() const { return gluehzustand >= 4; }
  DrawColour farbe() const
  { return farbverlauf[static_cast<int>(gluehzustand)]; }
};
```

Methoden

```
class Partikel{
private:
  DPoint2D position;
  DPoint2D geschwindigkeit;
  double gluehzustand;
public:
  Partikel(DPoint2D startpos);
  void weiter();
  bool verschwunden() const { return gluehzustand >= 4; }
  DrawColour farbe() const
  { return farbverlauf[static_cast<int>(gluehzustand)]; }
  // Partikel als einfache Punkte
  void maleMich(Drawing& pic) const { pic.drawPoint(position,
↪   farbe()); }
};
```



Konstructor

```
Partikel::Partikel(DPoint2D startpos){


}
```

Konstructor

```
Partikel::Partikel(DPoint2D startpos){
  // x variiert in [-20,20]
  position = startpos + DPoint2D(40 * rnd01() - 20, 0);
  // x variiert in [-1,1], y variiert in [1.5,3.5]
  geschwindigkeit
    = DPoint2D(0, 2.5) + DPoint2D(2 * rnd01() - 1, 2 * rnd01() - 1);
  gluehzustand = 0.8;

}
```

Bewegung

```
void Partikel::weiter()
{

}
```

Bewegung

```
void Partikel::weiter()
{
  // dx, dy variieren in [-0.3,0.3]
  geschwindigkeit
    = geschwindigkeit + 0.6 * DPoint2D(rnd01() - 0.5, rnd01() -
↪  0.5);
  // Gravitation: y reduzieren um 0.0015
  position = position + geschwindigkeit - 0.5 * DPoint2D(0, g);
  // Gravitation: y reduzieren um 0.003
  geschwindigkeit = geschwindigkeit - DPoint2D(0, g);
  // Leuchtkraft reduzieren
  gluehzustand += 0.05 * rnd01();
}
```

Bewegung

```
void Partikel::weiter()
{
  // dx, dy variieren in [-0.3,0.3]
  geschwindigkeit
    = geschwindigkeit + 0.6 * DPoint2D(rnd01() - 0.5, rnd01() -
↪  0.5);
  // Gravitation: y reduzieren um 0.0015
  position = position + geschwindigkeit - 0.5 * DPoint2D(0, g);
  // Gravitation: y reduzieren um 0.003
  geschwindigkeit = geschwindigkeit - DPoint2D(0, g);
  // Leuchtkraft reduzieren
  gluehzustand += 0.05 * rnd01();
}
```

Fertig! Back to maindraw().

Partikelsysteme, Feuer

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background
        pic = DrawColour(90, 180, 90); // grün
      // 2. neue Partikel erzeugen
        for (int j = 0; j < neuProBild; ++j)
        p.push_back(Partikel(DPoint2D(300, 10)));

      // 3. Partikel bewegen

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern
       pics[i] = pic;
    }
}
```

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background
        pic = DrawColour(90, 180, 90); // grün
      // 2. neue Partikel erzeugen
        for (int j = 0; j < neuProBild; ++j)
        p.push_back(Partikel(DPoint2D(300, 10)));

      // 3. Partikel bewegen
        for (auto& pp : p)
          pp.weiter();

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern
       pics[i] = pic;
    }
}
```

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background
      // 2. neue Partikel erzeugen
      // 3. Partikel bewegen
        for (auto& pp : p)
          pp.weiter();

      // 4. verschwundene Partikel löschen
         for (auto it = p.begin(); it != p.end();) {
           if (it->verschwunden()) {
             it = p.erase(it); // Lösche das Element und  ++
           }else {
             ++it; // Gehe zum nächsten Element
            }
         }
      // 5. Partikel malen

      // 6. Bild Speichern
       pics[i] = pic;
    }
}
```

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background
      // 2. neue Partikel erzeugen
      // 3. Partikel bewegen
        for (auto& pp : p)
          pp.weiter();

      // 4. verschwundene Partikel löschen
         for (auto it = p.begin(); it != p.end();) {
           if (it->verschwunden()) {
             it = p.erase(it); // Lösche das Element und  ++
           }else {
             ++it; // Gehe zum nächsten Element
            }
         }
      // 5. Partikel malen
        for (auto pit = p.rbegin(); pit != p.rend(); ++pit)
        pit->maleMich(pic);
      // 6. Bild Speichern
       pics[i] = pic;
    }
```

```
int maindraw(){
  for (int i = 0; i < anzahlBilder; ++i){
      // 1. Background

      // 2. neue Partikel erzeugen

      // 3. Partikel bewegen

      // 4. verschwundene Partikel löschen

      // 5. Partikel malen

      // 6. Bild Speichern

    }

    Drawing::makeanim(pics, "feuer", "gif", 4);
}
```