L-systemy. Tworzenie form roślinnych z wykorzystaniem System.Drawing

v1.1

Miłosz Orzeł

Celem artykułu jest zapoznanie Czytelnika z ciekawym zagadnieniem algorytmu wzrostu. Uczynię to na przykładzie tworzenia realistycznie wyglądających form roślinnych, które narysujemy w oparciu o elementy przestrzeni nazw System.Drawing.

06.2007 (poprawki 07.2007)
Pierwsza wersja tego artykułu została nagrodzona w konkursie CodeGuru.pl.

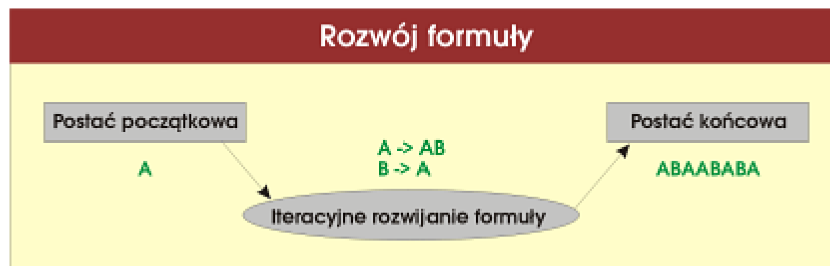
Proszę o informacje jeśli zauważysz jakiś błąd w tekście artykułu lub dołączonym do niego kodzie. ommail@wp.pl morzel.net

Kopiowanie jest dopuszczalne wyłącznie w celach niekomercyjnych i przy zachowaniu niezmienionej treści.

Trochę teorii.

L-system (system Lindenmayera) to zbiór symboli i reguł ich przetwarzania, który znajduje szerokie zastosowanie w grafice komputerowej do tworzenia fraktali i realistycznie wyglądających form roślinnych. Gramatyka ta została stworzona w 1968 roku przez węgierskiego biologa i botanika Aristida Lindenmayera dla potrzeb opisu budowy i rozwoju grzybów oraz glonów (później także bardziej złożonych form żywych).

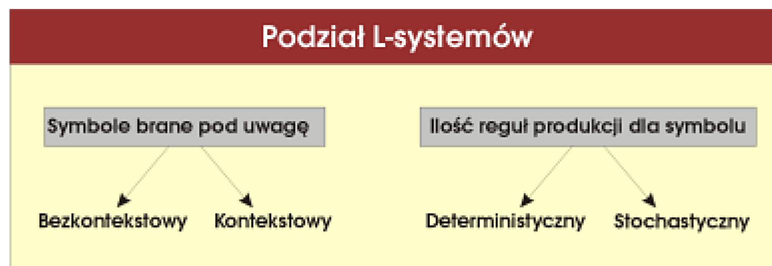
W celu stworzenia formuły (którą poddamy później interpretacji, tak by dała obraz rośliny) należy iteracyjnie rozwijać ciąg znaków poprzez zastosowanie *reguł produkcji*. Reguła produkcji określa symbol, który będzie zastępowany przez inny symbol (lub grupę symboli). Np. każde wystąpienie litery *A* w formule, może w kolejnych przebiegach pętli być zastępowane przez ciąg *AB*, zaś litera *B* może być zastępowana przez literę *A*. Dzięki wielokrotnemu powtarzaniu takiej operacji uzyskuje się coraz dłuższy i bardziej skomplikowany opis obiektu.



Rozmiar i złożoność formuły zależą od:

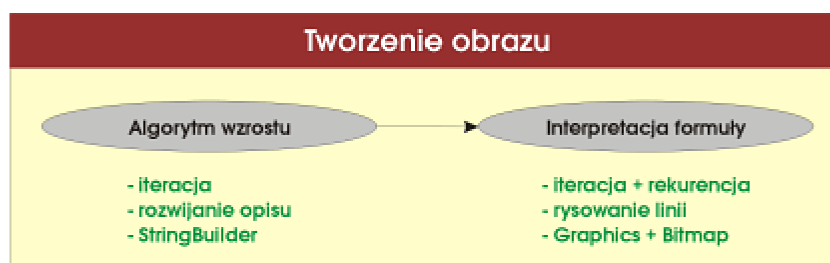
- pierwotnej postaci formuły,
- reguł produkcji,
- ilości obrotów pętli dokonującej przyrostu formuły.

Do stworzenia opisu rośliny wykorzystamy D0L-system (czyli *deterministyczny* i *bezkontekstowy* L-system). Słowo „deterministyczny” oznacza, że dla każdego symbolu istnieje wyłącznie jedna reguła produkcji. Określenie „bezkontekstowy” informuje nas, iż stosowane reguły produkcji odnoszą się wyłącznie do pojedynczych symboli (symbole sąsiednie nie są brane pod uwagę). Jeśli reguła produkcji zależy również on elementów sąsiednich, to mamy do czynienia z L-systemem *kontekstowym*. System, w którym dla danego symbolu istnieje więcej niż jedna reguła produkcji określamy jako *stochastyczny*.



Etapy tworzenia obrazu.

Program tworzący realistyczną formę roślinną przy użyciu L-systemów można podzielić na dwie części. Pierwsza z nich wykorzystuje algorytm wzrostu do stworzenia ciągu symboli, które stanowią opis rośliny. Druga część programu zajmuje się interpretacją tego opisu. W naszym przypadku interpretacja formuły będzie polegała na zamianie liter na niewielkie linie proste, które złożą się na całkiem ciekawy obraz.



Dlaczego w ogóle warto stosować L-systemy do rysowania form roślinnych? Wystarczy spojrzeć za okno i zobaczyć jak złożoną strukturę ma np. drzewo. Czy warto płacić grafikowi za malowanie każdego listka z osobna (bo przecież proste powielanie nie daje realistycznego efektu)? Na pewno nie! O wiele lepiej stworzyć obraz rośliny na drodze programowej.

Algorytm wzrostu.

Choć nazwa deterministyczny i bezkontekstowy L-system brzmi groźnie, implementacja algorytmu tworzącego opis rośliny jest naprawdę bardzo prosta. Chodzi o stworzenie jednej pętli, w której dany symbol zastąpimy innym symbolem (lub ciągiem symboli).

Uwaga! Pamiętaj, że obiekt klasy `String` reprezentuje niezmienny ciąg znaków. Co oznacza, że zupełnie nie nadaje się do operacji polegających na wielokrotnym przekształcaniu łańcucha tekstowego. Świetnie widać to na przykładzie poniższego kodu. Profiler wykazał, że podczas działania aplikacji zawierającej ten kod Garbage Collector zadziałał aż 439 razy. Gdy zamiast typu `String` użyjemy typu `StringBuilder` GC uruchomi się 3 razy!

```
string s = "";
for (int i = 0; i < 10000; i++)
{
    s += "x";
}
```

Poniżej znajduje się implementacja algorytmu wymyślonego przez Lindenmayera na potrzeby opisu rozwoju glonów:

```
private void RozwinFormule()
{
    // Tymczasowa zmienna przechowująca wartość rozwijanej formuły.
    StringBuilder nowaFormula = new StringBuilder();

    // Badamy kolejne symbole w formule i rozwijamy ją przy użyciu
    // określonych reguł produkcji.
    for (int i = 0; i < formula.Length; i++)
    {
        switch (formula[i])
        {
            case 'A':
                nowaFormula.Append("AB");
                break;
            case 'B':
                nowaFormula.Append("A");
                break;
        }
    }

    formula = nowaFormula;
}
```

Algorytm ten stosuje dwie proste reguły produkcji. Litera *A* jest zamieniana na ciąg *AB*, zaś litera *B* zamieniana jest na literę *A*. Zmienna `formula`, która przechowuje opis rośliny, jest zadeklarowana jako pole klasy o typie `StringBuilder`. Jeśli przyjmiemy, że początkową wartością zmiennej `formula` jest litera *A* to kolejne wywołania metody `RozwinFormule()` dadzą taki rezultat:

```
1 : AB
2 : ABA
3 : ABAAB
4 : ABAABABA
5 : ABAABABAABAAB
6 : ABAABABAABAABABAABABA
```

Nam jednak nie chodzi o modelowanie wzrostu alg, tylko o stworzenie opisu, który będzie się dał z łatwością zamienić na grafikę przedstawiającą roślinę. W tym celu należy wykorzystać ten nieco bardziej skomplikowany kod:

```
private void RozwinFormule()
{
    StringBuilder nowaFormula = new StringBuilder();

    for (int i = 0; i < formula.Length; i++)
    {
        switch (formula[i])
        {
            case 'L':
                nowaFormula.Append(regulaLiscTextBox.Text);
                break;
            case 'P':
                nowaFormula.Append(regulaPienTextBox.Text);
                break;
            case '[':
                nowaFormula.Append(regulaPoczatekGaleziTextBox.Text);
                break;
            case ']':
                nowaFormula.Append(regulaKoniecGaleziTextBox.Text);
                break ;
        }
    }

    formula = nowaFormula;
}
```

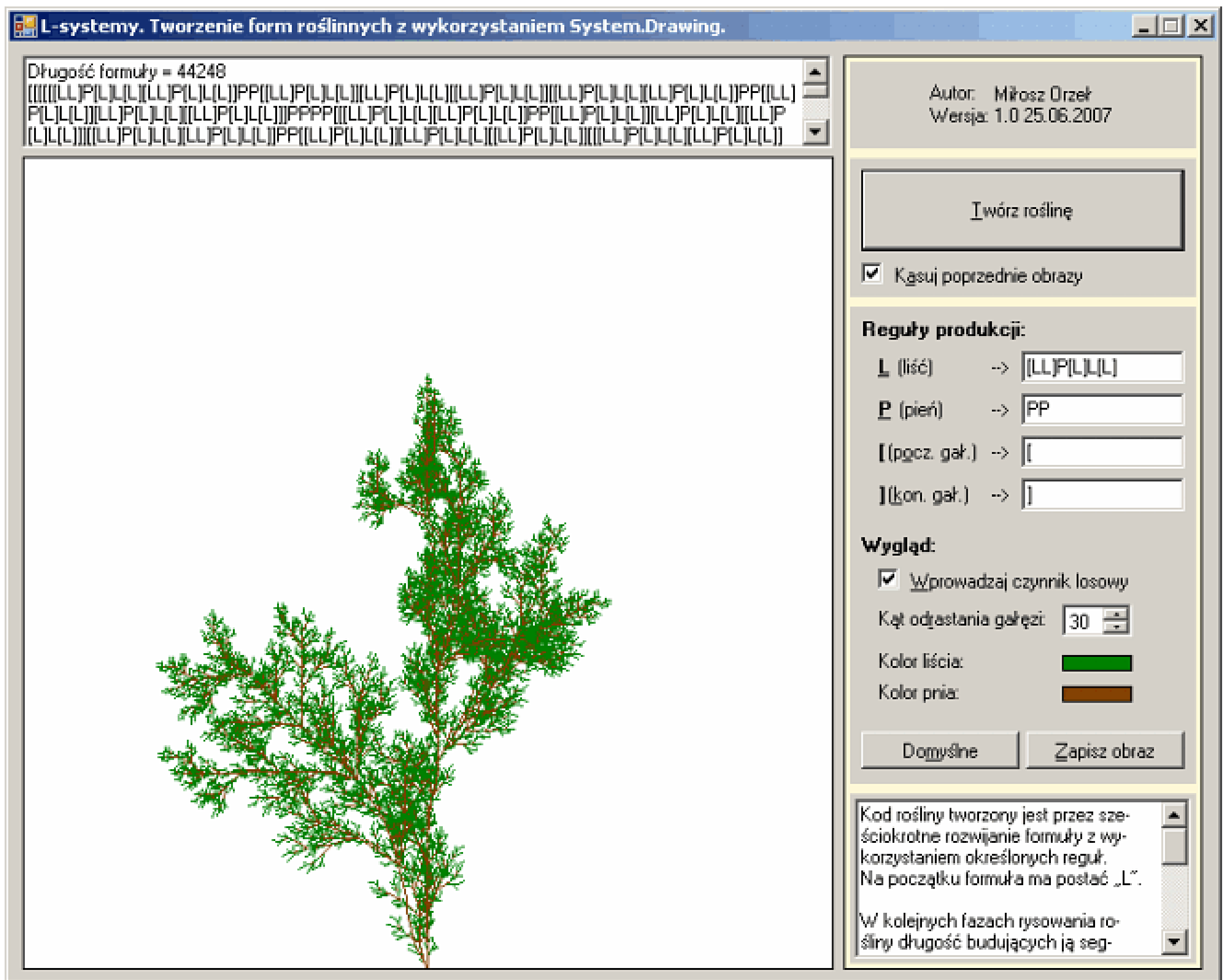
Jakie zmiany możemy dostrzec w tym kodzie? Po pierwsze reguły produkcji nie są zapisane w programie na sztywno (użytkownik może je ustalić w TextBoxach). Po drugie nie używamy liter *A* i *B* tylko *L* oraz *P* (aby łatwiej skojarzyć: *L* jak liść, *P* jak pień). Po trzecie - to najważniejsza zmiana - dodaliśmy dwie nowe reguły produkcji dla nawiasów prostokątnych. Te nawiasy oznaczają odpowiednio początek i koniec odgałęzienia. Jest to bardzo istotny element opisu rośliny ponieważ pozwoli nam na narysowanie odrostów (przy zastosowaniu rekurencji).

Ponieważ dajemy użytkownikowi możliwość wprowadzenia własnych reguł produkcji, przed przystąpieniem do interpretacji opisu rośliny należy sprawdzić jego poprawność. Formuła, w której liczba nawiasów otwierających jest różna od liczby nawiasów zamykających, może w niektórych przypadkach doprowadzić do przepełnienia stosu.

Interpretacja formuły.

Skoro znamy już sposób na stworzenie opisu (genomu rośliny), czas zabrać się za jego zamianę na obraz. Grafikę zbudujemy w prosty sposób - dzięki rysowaniu kolorowych linii. Ta łatwa technika da całkiem przyzwoity rezultat. Oczywiście o pełnym realizmie możemy mówić wtedy gdy do budowy obrazu wykorzystamy grafikę trójwymiarową z zaawansowanym liczeniem oświetlenia, teksturowaniem, antyaliasingiem itp. itd. Nam jednak zupełnie wystarczą możliwości oferowane przez elementy przestrzeni nazw `System.Drawing` takie jak klasy `Bitmap`, `Graphics` i `Pen` oraz struktura `Color`.

Przyjmijmy, że naszym celem jest zbudowanie aplikacji o następującym wyglądzie:



Krzaczek widoczny na ekranie został stworzony przez sześciokrotne rozwinięcie formuły, która pierwotnie miała postać *L*. Jak można zauważyć na samej górze okna, już po sześciu iteracjach opis rośliny przybrał znaczny rozmiar (44248 symboli). W algorytmie wzrostu zastosowane zostały następujące reguły produkcji:

```
L -> [LL]P[L]L[L]
P -> PP
[ -> [
] -> ]
```

Dlaczego akurat takie? Po prostu w połączeniu ze sposobem interpretacji formuły dały one ciekawy rezultat. Powyższy program jest dołączony do artykuły więc będziesz miał możliwość zastosowania własnych zasad rozwoju genomu rośliny - przekonasz się jak drobna zmiana w regule produkcji może całkowicie zmienić wygląd rośliny!

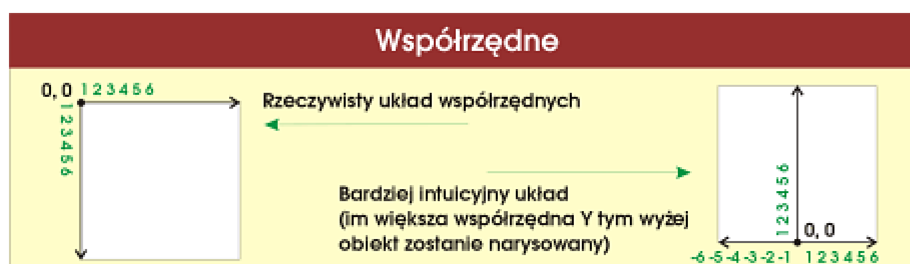
W jaki sposób można zamienić ciąg liter na obraz? Należy określić co mają znaczyć poszczególne symbole. W moim programie każda litera *L* będzie zamieniana na kreskę o kolorze zielonym, a każda litera *P* będzie zamieniana na kreskę o kolorze brązowym (barwy te można oczywiście zmienić). Gdy metoda odczytująca formułę napotka na symbol *[* oznaczać to będzie, że należy rozpocząć rysowanie bocznego odgałęzienia. Tworzenie gałęzi należy zrealizować przez rekurencyjne wywołanie metody rysującej. Warunkiem bazowym pozwalającym na wywołanie instrukcji *return* (czyli zakończenia działania metody wywołanej rekurencyjnie) jest napotkanie symbolu *]*. *]* oznacza po prostu koniec odgałęzienia i powrót do pnia, z którego zaczęło się to odgałęzienie. Kąt odrastania gałęzi może być losowo modyfikowany podczas rysowania rośliny - mutacja taka daje bardziej realistyczny efekt.

Poniżej znajduje się kod odpowiedzialny za stworzenie obrazu rośliny:

```
private void RysujRosline(int x0, int y0, double kat, double dlugosc)
{
    while (pozycja < formula.Length)
    {
        dlugosc = dlugosc * 0.995; // Im młodsze segmenty rośliny tym mniejsze.

        char symbol = formula[pozycja];
        pozycja++;

        if (symbol == 'L' || symbol == 'P') // Liść lub pień
        {
            if (czynikLosowyCheckBox.Checked)
            {
                // Pozbywamy się nienaturalnej regularności roślinki:
                kat += rng.Next(-5, 6);
            }

             // Obliczamy współrzędne linii:
            int x1 = (int)(x0 + dlugosc * Math.Cos(kat * Math.PI / 180.0));
            int y1 = (int)(y0 + dlugosc * Math.Sin(kat * Math.PI / 180.0));

            // Rysujemy pień lub listek:
            if (symbol == 'L')
            {
                rysunek.DrawLine(liscPedzel, poczatekUkladu.X + x0, poczatekUkladu.Y
                    - y0, poczatekUkladu.X + x1, poczatekUkladu.Y - y1); // Liść
            }
            else
            {
                 rysunek.DrawLine(pienPedzel, poczatekUkladu.X + x0, poczatekUkladu.Y
                    - y0, poczatekUkladu.X + x1, poczatekUkladu.Y - y1); // Pień
            }

            x0 = x1;
            y0 = y1;
        }

        if (symbol == '[') // Początek odgałęzienia.
        {
            katGalezi = -katGalezi; // Raz w prawo raz w lewo.
            RysujRosline(x0, y0, kat + katGalezi, dlugosc);
        }

        if (symbol == ']') // Koniec odgałęzienia.
        {
            return;
        }
    }

    // Wymusza odrysowanie powierzchni panelu (zdarzenie Paint).
    obrazPanel.Invalidate();
}
```

Zmienna `dlugosc` jest wprowadzona po to by ograniczyć rozmiar rośliny i nadać jej bardziej realistyczny wygląd poprzez stopniowe zmniejszanie długości budujących ją segmentów. Zmienna `pozycja` istnieje po to by móc śledzić aktualnie analizowany symbol w formule pomimo zastosowania rekurencji.

Jeśli uważnie przeanalizowałeś powyższy kod pewnie zastanawiasz się po co wprowadzony jest element `poczatekUkladu` typu `Point`. Przyjrzyj się poniższemu schematowi:



Rysunek jest tworzony przy użyciu obiektu `Graphics` na powierzchni bitmapy. Układ współrzędnych ekranowych ustawiony jest w taki sposób, że punkt *(0, 0)* znajduje się w lewym górnym rogu ekranu. W tym przypadku im większa współrzędna *Y* rysowanego punktu tym niżej znajduje się on na rysunku. Dla nas naturalna jest sytuacja odwrotna (im większy *Y* tym wyżej rysujemy punkt, tak jak w kartezjańskim układzie znanym z lekcji matematyki). W naszym konkretnym przypadku chcemy by roślina wyrastała z punktu znajdującego się na dole „kartki" tzn. by punkt *(0, 0)* był umiejscowiony tak jak to jest widoczne w prawej części rysunku. Jak to osiągnąć? Po pierwsze trzeba określić położenie początku okładu względem bitmapy:

```
poczatekUkladu = new Point(bufor.Width / 2, bufor.Height);
```

Potem należy uwzględniać ten punkt przy rysowaniu linii z użyciem metody `DrawLinie()`:

```
rysunek.DrawLine(liscPedzel, poczatekUkladu.X + x0, poczatekUkladu.Y - y0,
    poczatekUkladu.X + x1, poczatekUkladu.Y - y1); // Liść
```

Pierwszym elementem powyższego wywołania metody `DrawLine()` jest obiekt klasy `Pen`, który zawiera informacje takie jak kolor i grubość linii. Pędzle potrzebne do narysowania liści i pnia tworzone są w ten sposób:

```
liscPedzel = new Pen(kolorLisciaPanel.BackColor, 1);
pienPedzel = new Pen(kolorPniaPanel.BackColor, 1);
```

Kolejnym elementem metody `RysujRosline()`, który pewnie wzbudza wątpliwości jest ta linia:

```
obrazPanel.Invalidate();
```

Do czego ona służy? Otóż powoduje ona zaistnienie zdarzenia `Paint` w obiekcje klasy `Panel`. Żeby zrozumieć sens istnienia tego fragmentu kodu należy przyjrzeć się temu jak wyświetlany jest obraz rośliny...

Wyświetlanie obrazu.

Obraz przechowywany jest w obiekcie `bufor` klasy `Bitmap`. Operacje związane z tworzeniem samego obrazu (tzn. rysowaniem linii) wykonywane są przy użyciu obiektu `rysunek` klasy `Graphics`, który jest powiązany z obiektem `bufor`.

```
bufor = new Bitmap(obrazPanel.Width, obrazPanel.Height);
rysunek = Graphics.FromImage(bufor);
```

Po wywołaniu metody `RysujRosline()` obiekt `bufor` zawiera obraz rośliny, który może zostać pokazany na ekranie lub zapisany do pliku. W moim programie obraz jest ostatecznie wyświetlany na powierzchni obiektu `obrazPanel` klasy P`anel`, który znajduje się w po lewej stronie okna aplikacji.

Grafika jest wyświetlana w chwili zajścia zdarzenia `Paint` obiektu `obrazPanel`:

```
private void obrazPanel_Paint(object sender, PaintEventArgs e)
{
    if (bufor != null)
    {
        e.Graphics.DrawImage(bufor, 0, 0);
    }
}
```

Kiedy dochodzi do tego zdarzenia? Np. wtedy gdy cześć obiektu `obrazPanel` przestaje być przysłonięta przez okno innej aplikacji. Zdarzenie to ma również miejsce zaraz po zakończeniu rysowania linii składających się na obraz rośliny. Dzieje się tak dzięki wywołaniu metody `Invalidate()` obiektu `obrazPanel`, która wymusza odrysowanie tego obiektu (czyli wywołuje zdarzenie `Paint`).

Dlaczego grafikę warto przechowywać w obiekcie klasy `Bitmap`? Z co najmniej trzech powodów:

- zwiększona efektywność,
- możliwość odrysowania w zdarzeniu `Paint`,
- łatwy zapis do pliku.

Cały kod tworzący obraz rośliny moglibyśmy oczywiście umieścić w metodzie `obrazPanel_Paint()` ale miało by to bardzo negatywny wpływ na szybkość pracy programu. Gdybyśmy natomiast realizowali wyświetlenie grafiki w metodzie `RysujRosline()` wówczas tracilibyśmy fragment obrazu za każdym razem gdy cześć `obrazPanel` zostałaby przysłonięta przez okno innej aplikacji.

Zapisanie obrazu.

Gdy dysponujemy obiektem `bufor` klasy `Bitmap`, zachowanie obrazu rośliny do pliku typu *bmp* jest trywialne. Całą sprawę załatwia poniższy kod:

```
if (bufor != null)
{
    if (saveFileDialog.ShowDialog() == DialogResult.OK)
    {
        bufor.Save(saveFileDialog.FileName);
    }
}
```

Połączenie w całość.

Po dokładnym omówieniu najważniejszych elementów składających się na program, pozostaje tylko zestawienie całego kodu, istotnego dla tworzenia obrazu rośliny, na jednym listingu. Aby sztucznie nie zwiększać objętości artykułu, nie przedstawiam ponownie wyżej omówionego kodu metod `RozwinFormule()` oraz `RysujRosline()`.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MOrzel.Edu.LSystemy
{
    public partial class Form1 : Form
    {
        private StringBuilder formula = null; // Przechowuje opis rośliny.
        private Bitmap bufor = null; // Przechowuje obraz rośliny.
        private Graphics rysunek = null; // Umożliwia użycie metod rysujących.

        private Pen liscPedzel = null; // Określa wygląd liścia.
        private Pen pienPedzel = null; // Określa wygląd pnia.

        private Random rng = null; // Przydatne do mutacji kształtu rośliny.

        private int pozycja; // Określa element analizowany w trakcie rysowania.
        private int dlugosc; // Określa długość linii podczas rysowania.
        private int katGalezi; // Nachylenie bocznych odgałęzień.
        private Point poczatekUkladu; // Określa miejsce, które przyjmiemy za punkt
                                      // (0, 0).


        // *** Rozwija opis rośliny (algorytm wzrostu). ***
        private void RozwinFormule()
        {
            // ...
        }


        // *** Interpretuje formułę i zamienia ją na linie tworzące obraz. ***
        private void RysujRosline(int x0, int y0, double kat, double dlugosc)
        {
            // ...
        }
```

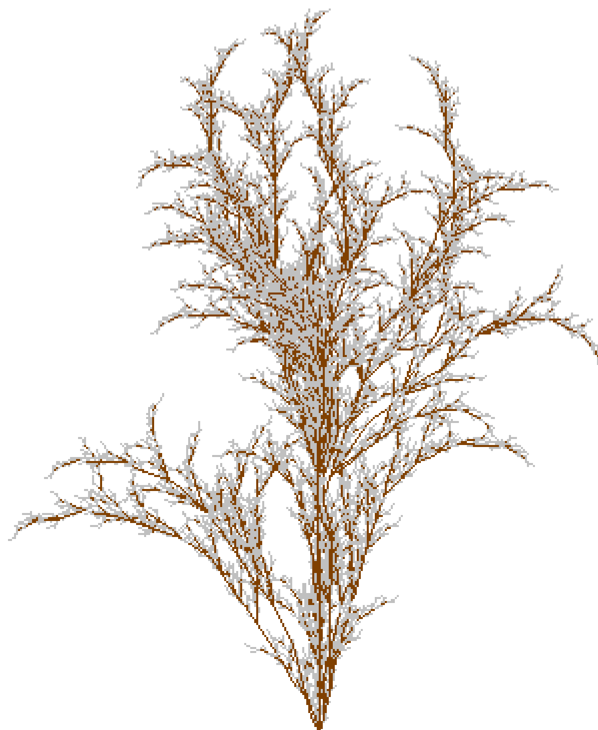
```
// *** Wywołuje metody tworzące obraz rośliny. ***
private void tworzRoslineButton_Click(object sender, EventArgs e)
{
    formula = new StringBuilder("L"); // Ustalenie formuły pierwotnej.

    liscPedzel.Color = kolorLisciaPanel.BackColor;
    pienPedzel.Color = kolorPniaPanel.BackColor;

    pozycja = 0;
    dlugosc = 5;
    katGalezi = (int)katGaleziNumericUpDown.Value;

    if (czynikLosowyCheckBox.Checked && rng.Next(2) == 0)
    {
        katGalezi = -katGalezi; // Dzięki temu roślina będzie czasem nachylona
                                // w lewo a czasem w prawo.
    }

    if (kasujPoprzednieCheckBox.Checked)
    {
        rysunek.Clear(Color.White);
    }


    try
    {
        for (int i = 0; i < 6; i++) // Wywołujemy algorytm wzrostu.
        {
            RozwinFormule();
        }

        int liczbaNawiasowOtwierajacych = 0;
        int liczbaNawiasowZamykajacych = 0;
        for (int i = 0; i < formula.Length; i++)
        {
            if (formula[i] == '[')
            {
                liczbaNawiasowOtwierajacych++;
            }

            if (formula[i] == ']')
            {
                liczbaNawiasowZamykajacych++;
            }
        }

        // Kontrolujemy poprawność formuły:
        if (liczbaNawiasowOtwierajacych != liczbaNawiasowZamykajacych)
        {
            MessageBox.Show("Niepoprawna formuła - liczba nawiasów " +
                "otwierających i zamykających nie jest równa!",
                "Błąd!", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }

        formulaRichTexBox.Text = "Długość formuły = " +
            formula.Length.ToString() + "\n" + formula.ToString();

        RysujRosline(0, 0, 90, dlugosc); // Rysujemy krzaczek.

        zapiszObrazButton.Enabled = true;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString(), "Błąd!", MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    }
}


// *** Wyświetla obraz z bitmapy na powierzchni Panelu ***
```

```
        private void obrazPanel_Paint(object sender, PaintEventArgs e)
        {
            if (bufor != null)
            {
                e.Graphics.DrawImage(bufor, 0, 0);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Tworzymy niezbędne obiekty:

            bufor = new Bitmap(obrazPanel.Width, obrazPanel.Height);
            rysunek = Graphics.FromImage(bufor);

            poczatekUkladu = new Point(bufor.Width / 2, bufor.Height);

            liscPedzel = new Pen(kolorLisciaPanel.BackColor, 1);
            pienPedzel = new Pen(kolorPniaPanel.BackColor, 1);

            rng = new Random();
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e)
        {
            // Zwalniamy nieużywane zasoby:

            bufor.Dispose();
            rysunek.Dispose();
            liscPedzel.Dispose();
            pienPedzel.Dispose();
        }

        private void zapiszObrazButton_Click(object sender, EventArgs e)
        {
            // Zapisujemy obraz:
            if (bufor != null)
            {
                if (saveFileDialog.ShowDialog() == DialogResult.OK)
                {
                    bufor.Save(saveFileDialog.FileName);
                }
            }
        }

        private void kolorLisciaPanel_Click(object sender, EventArgs e)
        {
            // Wybieramy kolor liścia:
            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                kolorLisciaPanel.BackColor = colorDialog.Color;
            }
        }

        private void kolorPniaPanel_Click(object sender, EventArgs e)
        {
            // Wybieramy kolor pnia:
            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                kolorPniaPanel.BackColor = colorDialog.Color;
            }
        }

        // ...
    }
}
```

Łączenie obrazów.

Poniższy krzew został stworzony poprzez kilkukrotne narysowanie wąskiej rośliny „na tej samej kartce”. Tworzenie obrazu bez usuwania poprzednich obrazów (tak by nakładały się na siebie) jest możliwe dzięki istnieniu tego warunku w metodzie `tworzRoslineButton_Click()` wywołującej generowanie roślin:

```
if (kasujPoprzednieCheckBox.Checked)
{
    rysunek.Clear(Color.White);
}
```



Podsumowanie.

To prawda, że wielu programistów (zwłaszcza .NET) może przez całą karierę zawodową ani razu nie skorzystać z L-systemów. Na pewno jednak warto na chwilę oderwać się od świata formatek, zapytań czy serializacji by zapoznać się ze sposobem modelowania rozwoju i budowy organizmów żywych stworzonym 40 lat temu przez węgierskiego biologa Aristida Lindenmayera.

Kod (C#, VS 2005) dołączony do artykułu można pobrać z tego adresu:
http://morzel.net/edu/lsystemy/lsystemy_src.zip

Linki.

Jeśli chcesz dowiedzieć się czegoś więcej o L-systemach polecam Ci stronę Wikipedii i znajdującą się na niej bazę odnośników do witryn poświeconych tematyce algorytmu wzrostu.
http://en.wikipedia.org/wiki/L-system

Galeria.

Poniżej znajduje się kilka roślin, które udało mi się wygenerować w programie załączonym do artykułu: