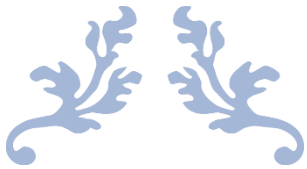


PROCEDURAL TREES GENERATOR

SISTEMI INTELLIGENTI

Gioacchino Luca Paduano
Matr. - 886472
gioacchinoluca.paduano@studenti.unimi.it

Contents

Descrizione del problema affrontato

I videogiochi sono complessi, lunghi e costosi da realizzare. La mole di materiale da realizzare o la quantità di persone necessarie per produrre contenuti può diventare un serio problema. Spesso, dunque, si ricorre a tecniche di *Procedural Content Generation* (*PCG*) per ridurre tali problemi. La PCG è un metodo per creare dati algoritmicamente piuttosto che manualmente. L'utilizzo di tale tecnica permette, ad esempio, di creare mondi complessi in poco tempo o di generare questi ultimi automaticamente, spesso permettendo di scegliere in modo selettivo cosa generare. L'**obiettivo** in questo progetto è di generare alberi utilizzando il sistema di *Lindenmayer*, comunemente noto anche come **L-System**.

L-Systems

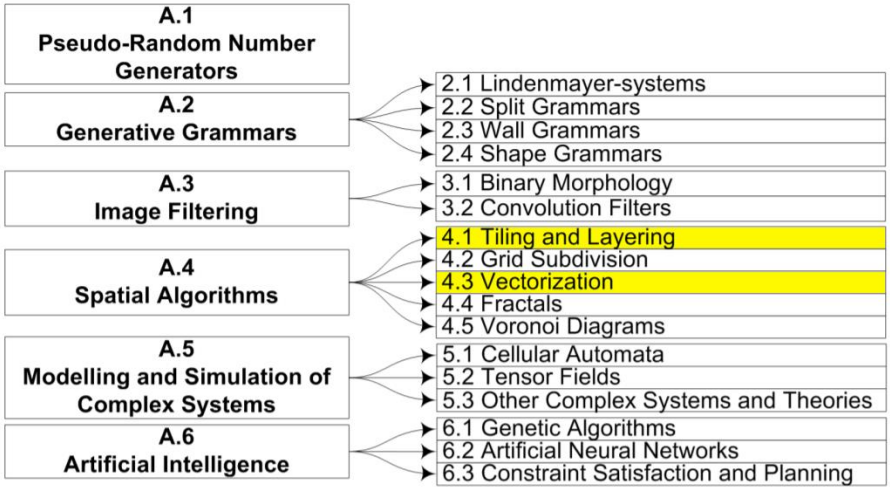
Dove ne abbiamo parlato? Come è stato approfondito l'argomento?

Gli L-Systems sono stati trattati in uno specifico argomento di corso, ossia quello riguardante i neuroni, in particolare della loro morfologia e funzionamento. Abbiamo visto infatti che i loro parametri e struttura possono essere ottenuti attraverso l'utilizzo di L-Systems.

Abbiamo poi approfondito questi ultimi analizzando un possibile uso, ossia quello di generare modelli geometrici di piante, utili proprio nel campo della generazione procedurale di vegetazione.

Lo scopo di questo progetto è proprio quello di esplorare più nel dettaglio questo tipo di contenuti.

Nel testo "*Procedural Content Generation for Games: A Survey*" si parla di tante tipologie di *PCG*, applicate in una varietà di contesti ed utili a scopi distinti. Nelle sezioni *4.1* e *4.3* in particolare vengono menzionati gli L-Systems e il loro utilizzo.



Come descritto nelle sezioni *4.1.3*, gli L-Systems vengono utilizzati per creare della vegetazione non auto-simile: i *simboli* possono rappresentare parti di piante come tronchi o foglie mentre le *regole di produzione* determinano la "crescita" della pianta partendo dal tronco iniziale. Questo sarà l'argomento trattato in questo progetto.

Nelle successive sezioni *4.1.4* e *4.1.5* invece viene descritto che tali algoritmi possono essere utilizzati per generare *costruzioni* (come avviene nel software "*CityEngine*") e usati per determinare *comportamenti* variabili in base al contesto.

Un ulteriore utilizzo, descritto nella sezione *4.3.2* descrive anche la possibilità di poter generare *reti stradali* cittadine con parametri di controllo come densità di popolazione, patterns stradali, vincoli di elevazione e vincoli locali come vicinanza ad acqua o ad altre strade.

Cosa è un L-System?

Un *L-System* o *Lindenmayer System* è un sistema di riscrittura parallelo e un tipo di grammatica formale.

Un L-System consiste in:

alfabeto di simboli	usato per creare stringhe, composto da elementi che possono essere sostituiti (*variabili*) e da quelli che non posso esserlo (*costanti* o *terminali*).
regole di produzione	convertono ciascun simbolo in una stringa di simboli.
assioma	una *stringa iniziale* da cui inizia la costruzione.
meccanismo	usato per tradurre le stringhe generate in strutture geometriche.

Perché usare gli L-System?

Gli L-Systems rappresentano una *tecnica efficiente* per costruire strutture organiche come piante ed arbusti, ma anche strutture con una certa ripetitività, come labirinti o strade cittadine. Una volta impostate le regole di costruzione è possibile iterare in maniera ricorsiva sostituendo la simbologia per ottenere delle strutture sempre più grandi e complesse.

Come funziona?

Con gli L-Systems piante e forme organiche sono facili da definire. Essi sono algoritmi *ricorsivi*. In particolare, le piante più complesse possono essere ottenute incrementando il *livello di ricorsione*.

Sono definiti da una tupla

$$\boldsymbol{G} = (V, \omega, P)$$

dove:

$\boldsymbol{V}$	(*alfabeto*) è un insieme di simboli contenente sia quelli che possono essere sostituiti (*variabili*) che quello che non possono essere sostituiti (*costanti* o *terminali*).
$\boldsymbol{\omega}$	(*start*, *assioma* o *iniziatore*) è una stringa di simboli di *V* che definisce lo stato iniziale del sistema.
$\boldsymbol{P}$	(*insieme di regole di produzione*) definisce il modo in cui le variabili possono essere sostituite con combinazioni di costanti ed altre variabili.

Regole di produzione

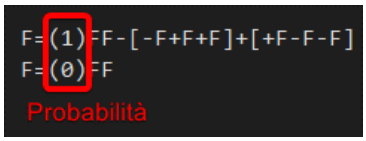
In particolare, una ***produzione*** o ***regola di produzione*** è una regola di riscrittura che specifica come un simbolo può essere sostituito ricorsivamente per generare nuove sequenze di simboli $u \rightarrow v$. La parte sinistra (che non può mai essere "vuota") viene sostituita dalla parte destra. Per ciascun simbolo *A* (che fa parte

dell'insieme *V*, ossia i simboli che compongono la regola di produzione) che non compare nella parte sinistra della produzione *P*, si assume automaticamente la produzione di identità $A \rightarrow A$. Quindi il simbolo resta invariato e non viene sostituito. Tali simboli vengono chiamati *costanti* o *terminali*.

Una *produzione* consiste di due stringhe, un *predecessore* e un *successore*.

Probabilità e Contesto

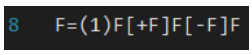
Le *regole di produzione* vengono applicate iterativamente partendo dallo stato iniziale. Ad ogni iterazione si cerca di eseguire quante più regole possibile. Questa caratteristica rende gli L-Systems una "*grammatica formale*" e non un "*linguaggio formale*", dove viene applicata solo una regola ad ogni iterazione. È possibile infatti definire diverse regole, ciascuna con certo gradi di probabilità di essere scelta in ciascuna iterazione.



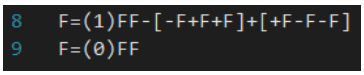
In particolare, un L-System è definito *libero da contesto* (*context-free*) se una *regola di produzione* si riferisce solo ad uno specifico simbolo e non anche ai suoi vicini. In quest'ultimo caso si parla di "*Context-sensitive L-Systems*" e si usa una "*grammatica libera da contesto*".

Deterministicità e Stocasticità

Se c'è esattamente una produzione per ciascun simbolo, allora l'L-System è definito *deterministico* (un *context-free L-System deterministico* è definito *DOL*). Se invece ci sono più produzioni per ciascun simbolo è definito *stocastico*.



Deterministico



Stocastico

Interpretazione a Tartaruga

Cos'è la Turtle Interpretation?

Negli anni sono state proposte diverse metodologie per interpretare gli L-Systems. L'*Interpretazione a Tartaruga* (***Turtle Interpretation***) delinea un modello di rappresentazione particolarmente flessibile, utile soprattutto per gestire rappresentazioni più complesse.

Come viene utilizzata?

Viene utilizzata per interpretare *graficamente* una stringa, ossia una sequenza di simboli, adoperando delle specifiche regole. Da notare che la stringa a cui ci riferiamo è quella generata applicando ricorsivamente l'algoritmo di Lindenmayer.

Come funziona?

L'idea di base dell'*Interpretazione a Tartaruga* è la seguente.

Uno *stato* della tartaruga è definito come una tripletta (x,y,α) dove:

$(\boldsymbol{x}, \boldsymbol{y})$	Sono coordinate cartesiane che rappresentano la *posizione* della tartaruga.
$\boldsymbol{\alpha}$	È l'angolo, chiamato *heading*, che descrivere la direzione verso cui è rivolta la tartaruga.

Data una certa *dimensione del passo* $\boldsymbol{d}$ (*step size*) e l'*angolo di incremento* $\boldsymbol{\delta}$ (*angle increment*), la tartaruga risponde ai comandi rappresentati dai seguenti simboli:

F	Si muove avanti di un passo di lunghezza d. Lo stato della tartaruga cambia da (x,y,α) a (x',y',α), dove $x'=x+d\cos\alpha$ e $y'=y+d\sin\alpha$. Viene disegnato un segmento di linea tra due punti (x,y) e (x',y').
f	Si muove avanti di un passo di lunghezza d senza disegnare la linea.
+	Gira a sinistra di un angolo δ. Il prossimo stato della tartaruga sarà $(x,y,\alpha+\delta)$. L'orientamento positivo dell'angolo è antiorario.
−	Gira a destra di un angolo δ. Il prossimo stato della tartaruga sarà $(x,y,\alpha-\delta)$.

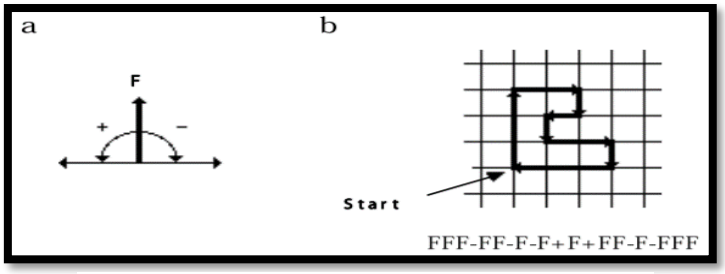


Figura 1: Esempio di applicazione. La figura a destra (b) rappresenta il risultato dell'esecuzione usando la regola della figura a sinistra (a)

Modellazione a tre dimensioni

L'interpretazione a tartaruga degli L-Systems appena descritta è utile per rappresentazioni 2D, ma può essere estesa al 3D nel seguente modo.

Il concetto chiave è nel rappresentare l'*orientamento* attuale della tartaruga in uno spazio a tre vettori $\vec{H}, \vec{L}, \vec{U}$ che indicano, rispettivamente, la testa della tartaruga (*heading*), la direzione a sinistra (*left*) e la direzione in alto (*up*).

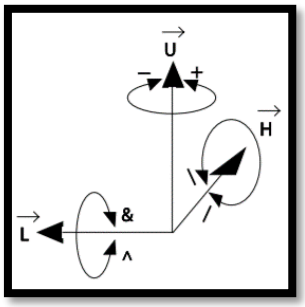


Figura 2: Controllare la tartaruga a tre dimensioni

Tali vettori hanno una lunghezza unitaria e sono perpendicolari tra loro.

Le *rotazioni* della tartaruga sono espresse attraverso l'equazione:

$$[\vec{H}, \vec{L}, \vec{U}] = [\vec{H}, \vec{L}, \vec{U}]\,\boldsymbol{R}$$

La rotazione di un certo angolo α dei tre vettori è ottenuta nel seguente modo:

$$\mathbf{R_U}(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_L}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$\mathbf{R_H}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

dove $\boldsymbol{R}$ è una matrice di rotazione 3x3.

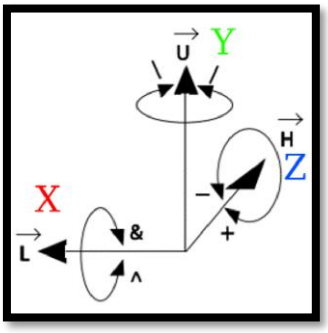
La simbologia che segue invece serve a controllare l'*orientamento* della tartaruga:

+	Gira a sinistra di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_U(\delta)}$.
−	Gira a destra di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_U(-\delta)}$.
&	Abbassa di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_L(\delta)}$.
^	Alza di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_L(-\delta)}$.
\	Gira a sinistra di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_H(\delta)}$.
/	Gira a destra di un angolo δ, usando la matrice di rotazione $\boldsymbol{R_H(-\delta)}$.
|	Gira in direzione opposta, usando la matrice di rotazione $\boldsymbol{R_U(180°)}$.

Nel progetto la *tartaruga* è rappresentata attraverso una *struttura* d'appoggio, definita nel seguente modo:

```
public struct Turtle
{
    public Quaternion direction;
    public Vector3 position;
    public Vector3 step;

    public Turtle(Turtle other)
    {
        this.direction = other.direction;
        this.position = other.position;
        this.step = other.step;
    }

    public Turtle(Quaternion direction, Vector3 position, Vector3 step)
    {
        this.direction = direction;
        this.position = position;
        this.step = step;
    }

    public void Forward()
    {
        position += direction * step;
    }

    public void RotateX(float angle)
    {
        direction *= Quaternion.Euler(angle, 0, 0);
    }

    public void RotateY(float angle)
    {
        direction *= Quaternion.Euler(0, angle, 0);
    }

    public void RotateZ(float angle)
    {
        direction *= Quaternion.Euler(0, 0, angle);
    }

}
```

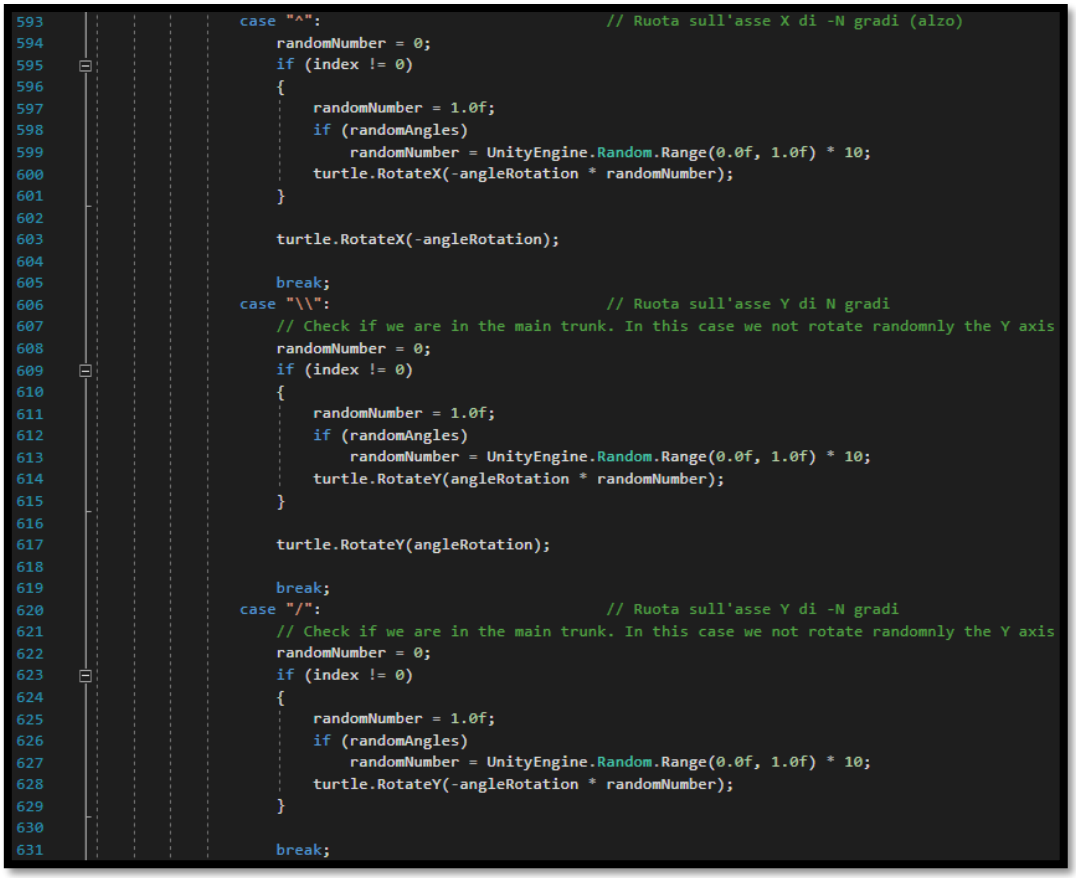
L'*orientamento* della tartaruga si ottiene così:



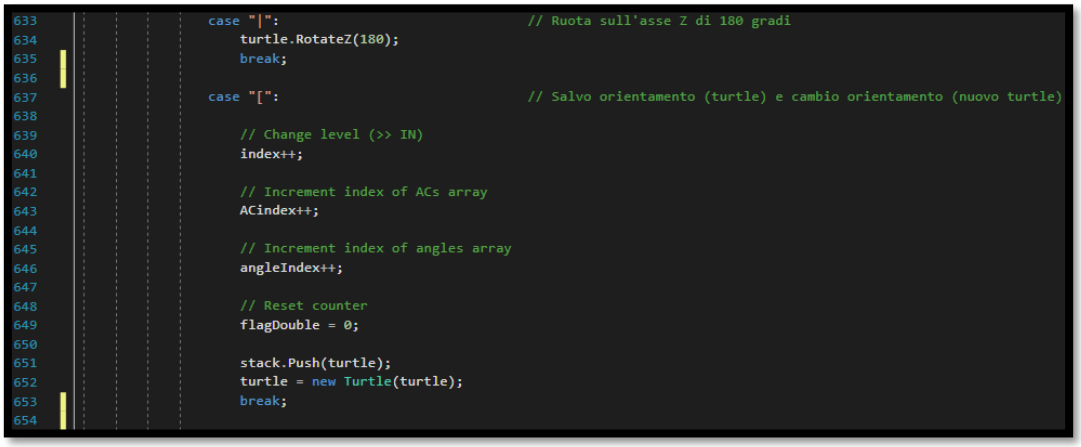
Ottenuta la regola finale, dopo una serie di iterazioni della regola di produzione, si sfrutta la simbologia della tartaruga per gestire l'orientamento.

```
for (int i = 0; i < moduleString.Length; i++)
{
    string module = moduleString[i] + "";

    // Check level to manage

    switch (module)
    {

        case "F":                           // Cylinder (piece of branch)
        case "G":

            CreateSegment(
                segmentAxisSamples,         // Campioni situati sull'asse del segmento
                segmentRadialSamples,       // Campioni radiali del segmento
                segmentWidth,               // Larghezza segmento
                segmentHeight,              // Altezza segmento
                narrowBranches,             // Tronchi piccoli
                trunkMaterial,              // Materiala del tronco
                turtle,                     // Turtle
                stack.Count,                // Stack usato per gestire i livelli
                ref currentMesh,            // Mesh (riferimento)
                ref chunkCount,             // Numero di pezzi (riferimento)
                trunk,                      // Tronco
                segmentsCache,              // Struttura con tutti i segmenti
                ref actualLevel,            // Livello attuale dell'albero
                index,                      // Indice di "actualLevel"
                ref prevAngle,              // Angolo salvato (usato per calcolare il pezzo di ramo precedente)
                posNegAngle,                // Salvo la positività/negatività dell'angolo
                ref prevPosNegAngle,        // Salvo array che contiene la positività/negatività dell'angolo dell'iterazione precedente
                angleRotation,              // Di quanto abbiamo ruotato prima di entrare qui dentro
                ref AC,                     // Salvo array AC utile per fare i calcoli dei punti delle curve di Bézier
                ACindex,                    // Indice dell'array AC
                ref angles,                 // Salvo array "angles" utile per fare i calcoli dei punti delle curve di Bézier
                angleIndex);                // Indice dell'array "angles"
            turtle.Forward();

            posNegAngle = 0;
            //Debug.Log("posNegAngle = RESET | posNegAngle > " + posNegAngle);

            break;
```

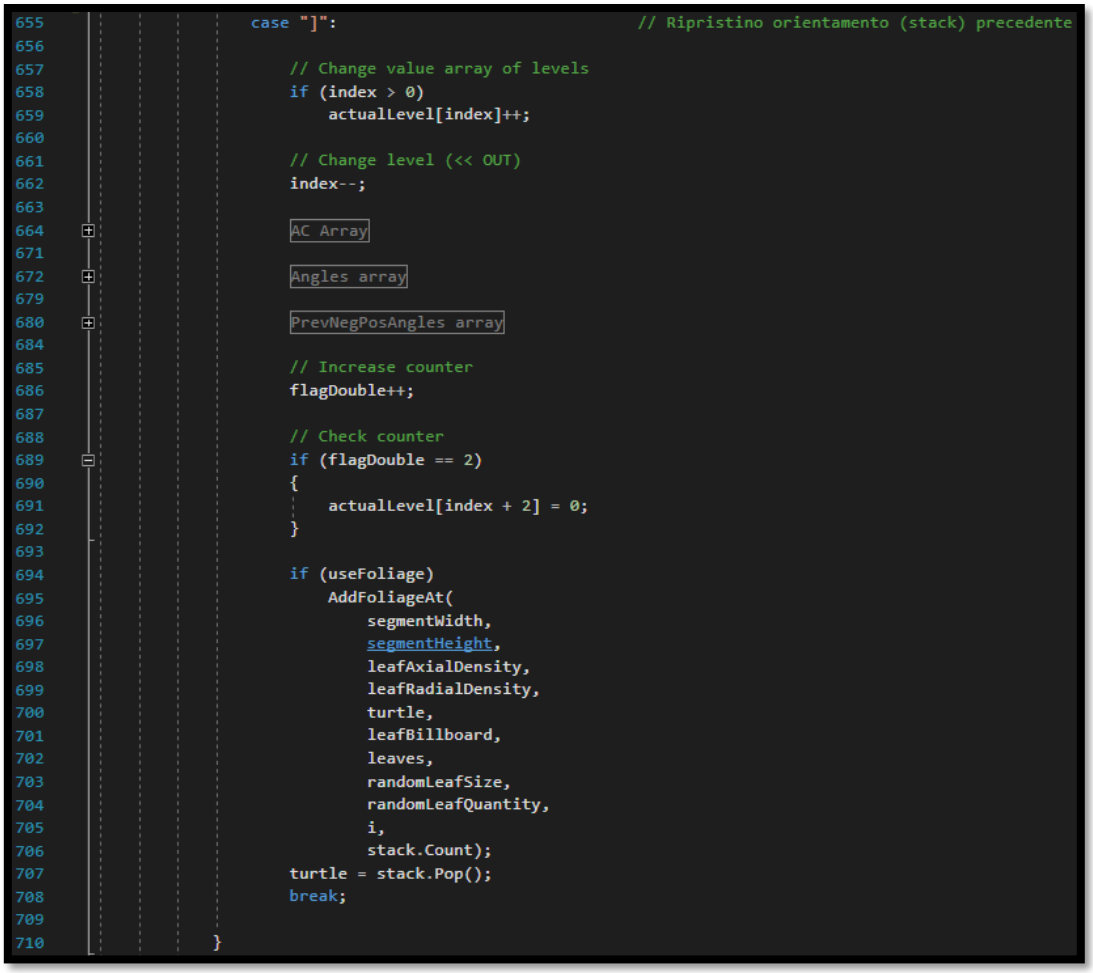
```
            case "+":                               // Ruota sull'asse Z di N gradi (sinistra)

                //DEBUG
                DEBUG

                turtle.RotateZ(angleRotation);
                posNegAngle = 1;

                //DEBUG
                DEBUG

                break;

            case "-":                               // Ruota sull'asse Z di -N gradi (destra)

                //DEBUG
                DEBUG

                turtle.RotateZ(-angleRotation);
                posNegAngle = -1;

                //DEBUG
                DEBUG
                break;

            case "&":                               // Ruota sull'asse X di N gradi (abbasso)
                float randomNumber = 0;
                if (index != 0)
                {
                    randomNumber = 1.0f;
                    if (randomAngles)
                        randomNumber = UnityEngine.Random.Range(0.0f, 1.0f) * 10;
                    turtle.RotateX(angleRotation * randomNumber);
                }

                turtle.RotateX(angleRotation);

                break;
```



```
593             case "^":                                   // Ruota sull'asse X di -N gradi (alzo)
594                 randomNumber = 0;
595                 if (index != 0)
596                 {
597                     randomNumber = 1.0f;
598                     if (randomAngles)
599                         randomNumber = UnityEngine.Random.Range(0.0f, 1.0f) * 10;
600                     turtle.RotateX(-angleRotation * randomNumber);
601                 }
602 
603                 turtle.RotateX(-angleRotation);
604 
605                 break;
606             case "\\":                                  // Ruota sull'asse Y di N gradi
607                 // Check if we are in the main trunk. In this case we not rotate randomnly the Y axis
608                 randomNumber = 0;
609                 if (index != 0)
610                 {
611                     randomNumber = 1.0f;
612                     if (randomAngles)
613                         randomNumber = UnityEngine.Random.Range(0.0f, 1.0f) * 10;
614                     turtle.RotateY(angleRotation * randomNumber);
615                 }
616 
617                 turtle.RotateY(angleRotation);
618 
619                 break;
620             case "/":                                   // Ruota sull'asse Y di -N gradi
621                 // Check if we are in the main trunk. In this case we not rotate randomnly the Y axis
622                 randomNumber = 0;
623                 if (index != 0)
624                 {
625                     randomNumber = 1.0f;
626                     if (randomAngles)
627                         randomNumber = UnityEngine.Random.Range(0.0f, 1.0f) * 10;
628                     turtle.RotateY(-angleRotation * randomNumber);
629                 }
630 
631                 break;
```



```
633             case "|":                                   // Ruota sull'asse Z di 180 gradi
634                 turtle.RotateZ(180);
635                 break;
636 
637             case "[":                                   // Salvo orientamento (turtle) e cambio orientamento (nuovo turtle)
638 
639                 // Change level (>> IN)
640                 index++;
641 
642                 // Increment index of ACs array
643                 ACindex++;
644 
645                 // Increment index of angles array
646                 angleIndex++;
647 
648                 // Reset counter
649                 flagDouble = 0;
650 
651                 stack.Push(turtle);
652                 turtle = new Turtle(turtle);
653                 break;
654 
```



```
655             case "]":                                   // Ripristino orientamento (stack) precedente
656
657                 // Change value array of levels
658                 if (index > 0)
659                     actualLevel[index]++;
660
661                 // Change level (<< OUT)
662                 index--;
663
664                 AC Array
671
672                 Angles array
679
680                 PrevNegPosAngles array
684
685                 // Increase counter
686                 flagDouble++;
687
688                 // Check counter
689                 if (flagDouble == 2)
690                 {
691                     actualLevel[index + 2] = 0;
692                 }
693
694                 if (useFoliage)
695                     AddFoliageAt(
696                         segmentWidth,
697                         segmentHeight,
698                         leafAxialDensity,
699                         leafRadialDensity,
700                         turtle,
701                         leafBillboard,
702                         leaves,
703                         randomLeafSize,
704                         randomLeafQuantity,
705                         i,
706                         stack.Count);
707                 turtle = stack.Pop();
708                 break;
709
710             }
```

Strutture ramificate

La rappresentazione a tartaruga, a due e tre dimensioni, è un buon metodo di rappresentazione che ci permette di gestire un certo numero di casi. In natura però quasi tutte le piante presentano delle *strutture ramificate* e per questo motivo sono necessarie strutture matematiche per generare alberi (grafi).

Albero assiale

Un *albero assiale* (*axial tree*) è il complementare (nella nozione della *teoria dei grafi*) di un *albero con radice* (*rooted tree*) unito alla nozione (botanica) dell'*asse del ramo*.

Un *albero con radice* possiede degli archi etichettati ed orientati. La sequenza di archi passa da un nodo primario, detto ***root node*** (●Tree root), ad un nodo finale, detto ***terminal node*** (○Terminal node).

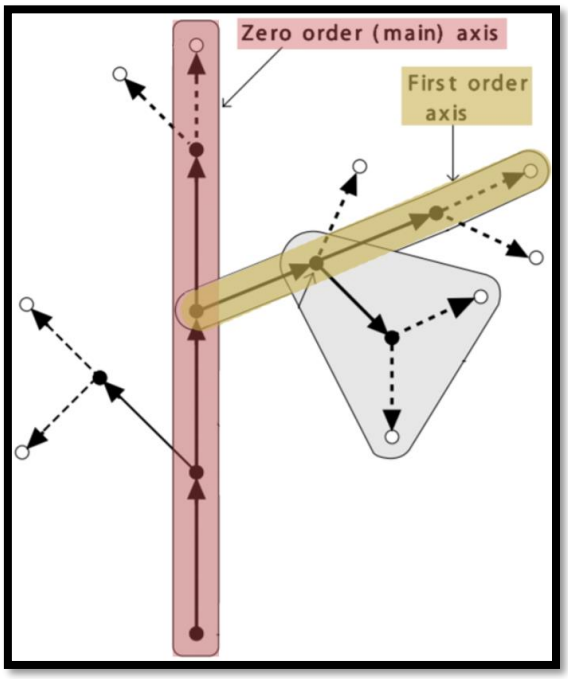
In biologia questi archi vengono indicati come *segmenti di diramazione* (*branch segments*). Un segmento seguito da almeno un altro segmento è chiamato *internodo* (***internode***) (⟶Internode). Un segmento terminale (cioè non seguito da altri archi) è chiamato apice (***apex***) (⇢Apex).

Come già accennato, un *albero assiale* è uno speciale tipo di *albero con radice*. In ciascuno dei suoi nodi, si distingue al massimo *un segmento diritto* (***straight***) in uscita.

Tutti gli altri archi sono chiamati *segmenti laterali* (***lateral segments***).

Una sequenza di segmenti è chiamata *asse* (***axis***) se:

1) il primo segmento nella sequenza ha origine nella radice (*root*) dell'albero o come un segmento laterale (*lateral segment*) in qualche nodo,
2) ciascun segmento seguente è un segmento dritto (*straight segment)*, e
3) l'ultimo segmento non è seguito da nessun segmento dritto nell'albero



Considerando tutti i suoi discendenti, un *asse* costituisce un *ramo* (***branch***), che è esso stesso un *sottoalbero assiale* (***axial (sub)tree***).

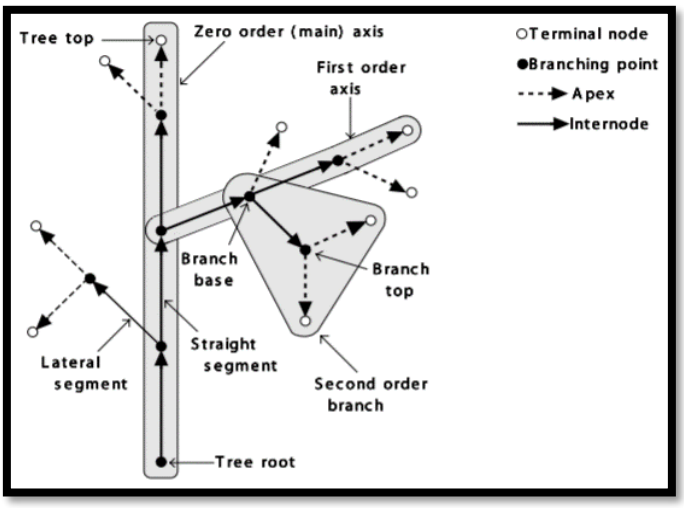


Figura 3: Esempio di albero generato tramite struttura assiale

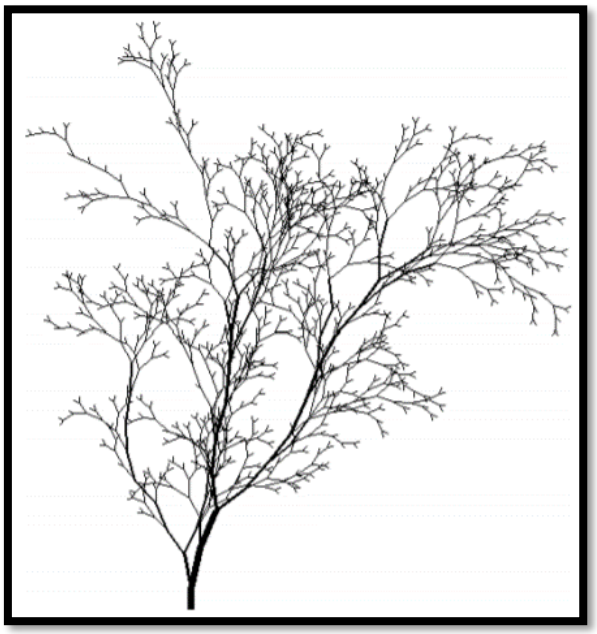
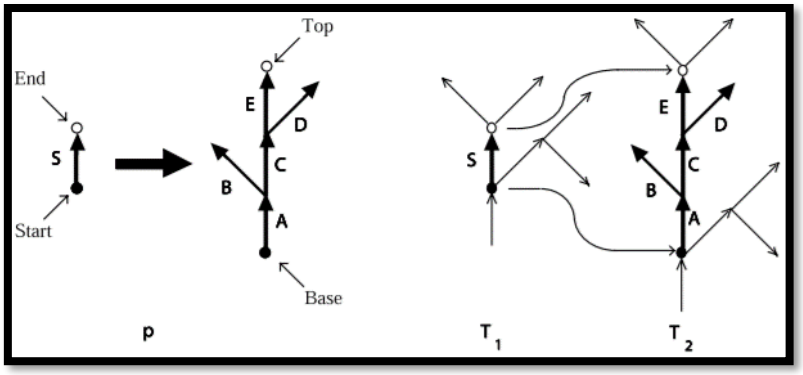


Figura 4: Esempio di albero generato tramite struttura assiale

Tree OL-systems

Per modellare lo sviluppo di strutture ramificate è possibile utilizzare un meccanismo di riscrittura che opera direttamente su *alberi assiali*.

Una *regola di riscrittura*, o *produzione di alberi*, sostituisce un arco predecessore con un albero assiale successore in modo tale che il nodo iniziale del predecessore sia identificato con la base del successore e il nodo finale sia identificato con la parte superiore del successore.



Bracketed OL-systems

Dato che gli L-Systems non posseggono una struttura dati per rappresentare *alberi assiali*. Un modo per farlo è quello di usare le *stringhe con parentesi quadre*, che vengono aggiunte alla rappresentazione a tartaruga con la relativa simbologia.

Tali simboli vengono interpretati dalla tartaruga nel seguente modo:

[*Inserisce* lo stato corrente della tartaruga in uno *stack*. Le informazioni salvate nella pila contengono la posizione e l'orientamento della tartaruga, e possibilmente altri attributi come il colore e la larghezza delle linee che vengono disegnate.

] *Estrae* uno stato dallo stack e lo rende lo stato corrente della tartaruga. Non viene tracciata alcuna linea, sebbene in generale la posizione della tartaruga cambi.

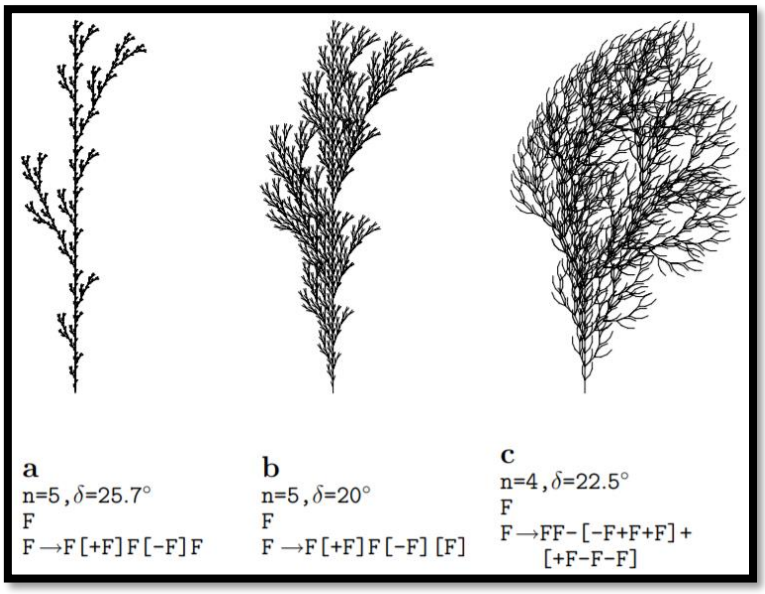


Figura 5: Esempi di L-Systems 2D edge-rewriting

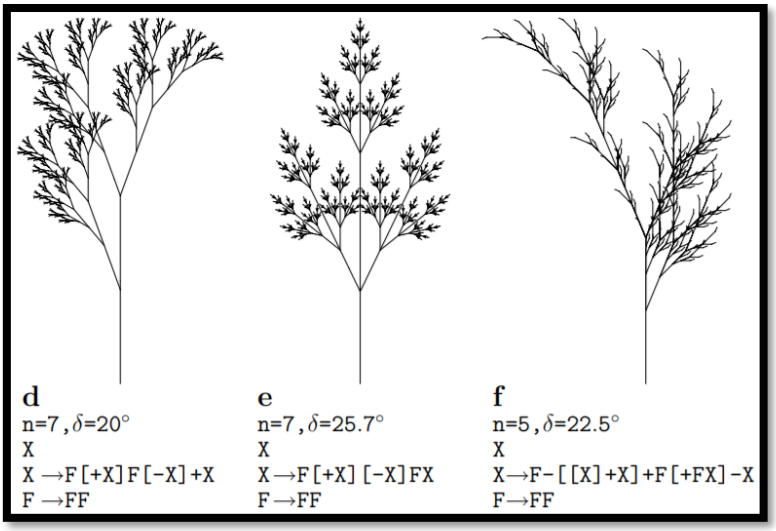


Figura 6: Esempi di L-Systems 2D node-rewriting

Gli esempi appena proposti sono stati implementati.

Di seguito possiamo vedere il codice utilizzato per ciascuno di essi:

```
// tree A (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=25.7
number of derivations=3

F=(1)F[+F]F[-F]F
```

```
// tree B (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=20
number of derivations=3

F=(1)F[+F]F[-F][F]
```

```
// tree C (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=22.5
number of derivations=2

F=(1)FF-[-F+F+F]+[+F-F-F]
F=(0)FF
```

```
// tree D (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=20
number of derivations=3

F = (1)G[+F]G[-F]+F
G = (1)GG
```

```
// tree E (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=25.7
number of derivations=3

F = (1)G[+F][-F]GF
G = (1)GG
```

```
// tree F (pag. 25) => http://algorithmicbotany.org/papers/abop/abop.pdf
// number of derivations => MIN VALUE = 1

axiom=F
angle=22.5
number of derivations=3

F = (1)G-[[F]+F]+G[+GF]-F
G = (1)GG
```

Descrizione del codice:

axiom	Assioma contenente i simboli che generano i rami.
angle	Angolo di rotazione della tartaruga.
number of derivations	Numero di iterazioni dell'algoritmo.
F	Variabile dell'alfabeto.
$\boldsymbol{FF-[-F+F+F]}$	Regola di produzione.
$(\mathbf{1})$	Probabilità di usare questa regola di produzione.

Quelli che seguono sono i risultati ottenuti:

Come possiamo notare gli alberi in questione presentano una struttura bidimensionale.

L'albero che segue presenta invece una struttura *tridimensionale*.

g
n=3, δ=22.5°
F
F → F[-&^F][^++&F]||F[--&^F][+&F]

Figura 7: Modello 3D

La *tridimensionalità* è raggiunta modificando la rotazione della tartaruga non solo sull'asse Z ma anche sugli altri due assi X ed Y.

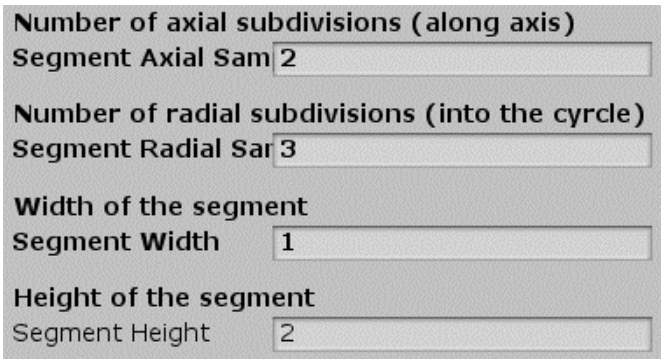
Le fasi che compongono la realizzazione dell'albero sono:

- Estrazione della tupla: *alfabeto*, *assioma*, *regole* (da un **file di testo**).
- Applicazione delle regole in modo iterativo (per ottenere l'intera **stringa**).
- Analisi della stringa e conversione in mesh (albero composta da tronco, rami, foglie).

Implementazione

Segmenti dell'asse

La realizzazione di ciascuna *segmento* prevede l'impostazione di diversi parametri.



Il loro ruolo è il seguente:

Number of axial subdivisions	numero di suddivisioni lungo l'asse che compone il ramo (l'intero segmento).
Number of radial subdivisions	numero di suddivisioni che compongono la circonferenza del ramo.
Width of the segment	larghezza del ramo.
Height of the segment	lunghezza del ramo.

Due parametri fondamentali sono la *suddivisione assiale* e la *suddivisione radiale*. Il primo suddivide l'*asse* in N parti; il secondo suddivide il *raggio* del segmento in M parti.

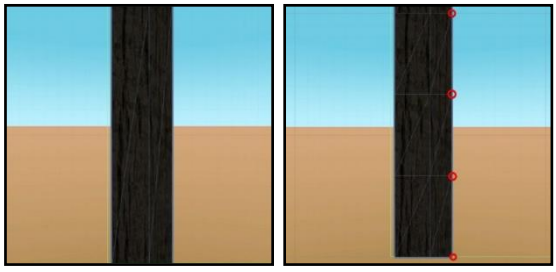


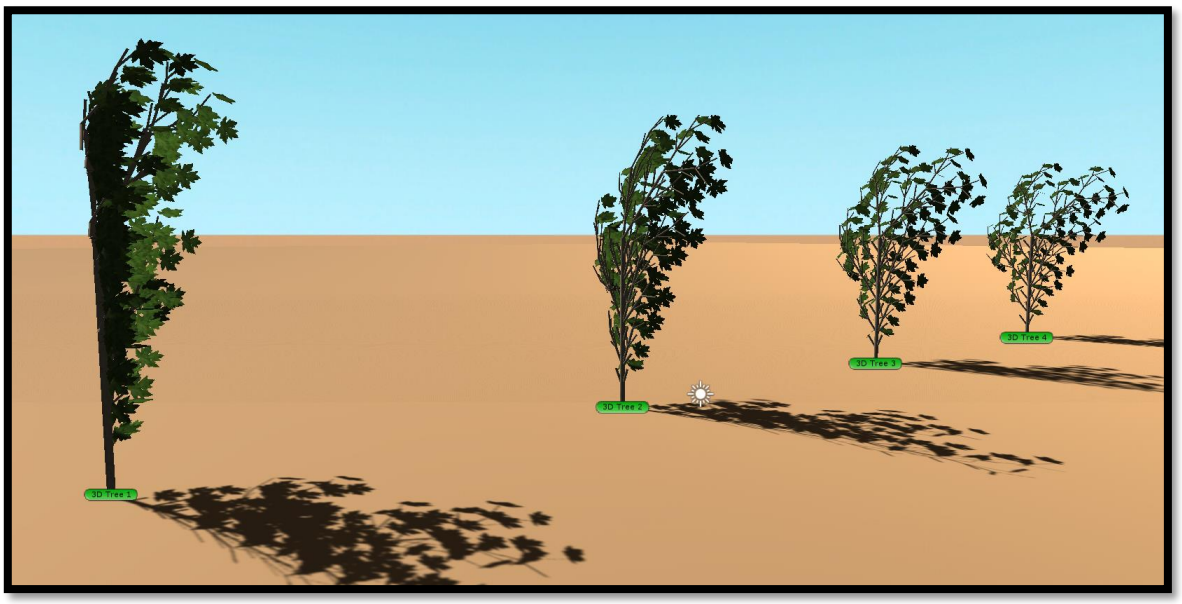
Figura 8: Sinistra: suddivisione assiale a 2 punti; Destra: suddivisione assiale a 4 punti

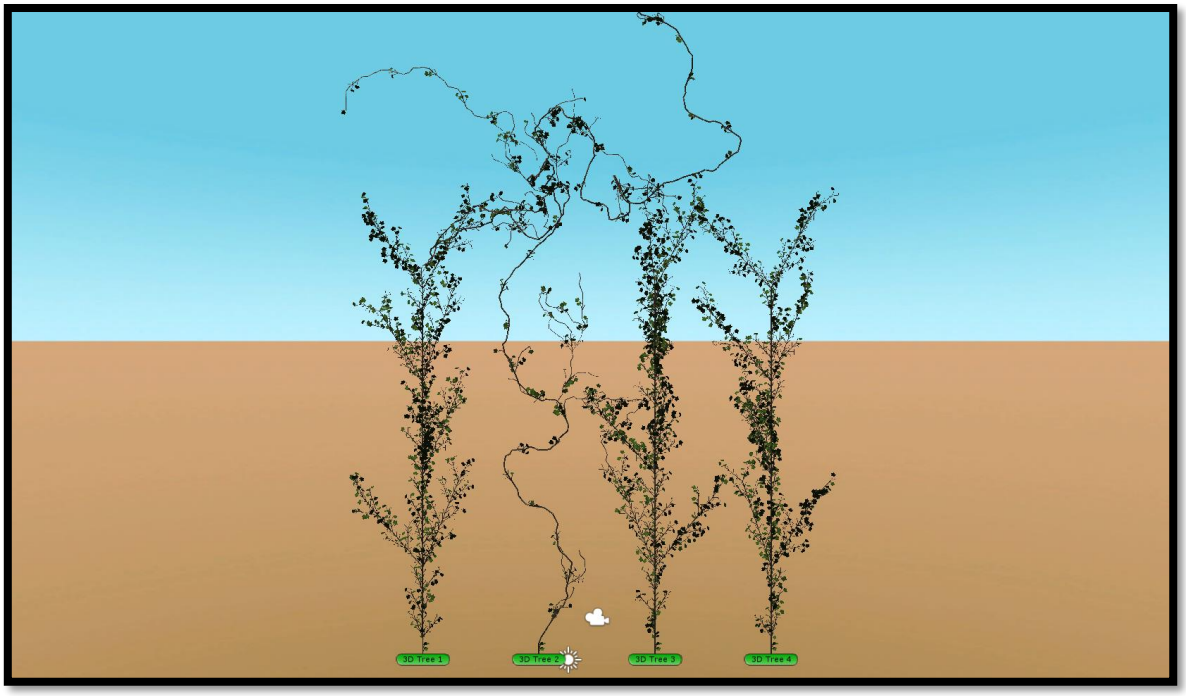
Figura 9: Sinistra: suddivisione radiale a 5 punti; Destra: suddivisione rradiale a 10 punti

L'aumento/diminuzione di tali parametri determina, in maniera proporzionale, una conseguente segmentazione del pezzo di ramo/tronco (come possiamo notare nelle due figure sopra). L'aumentare di essi aumenta dunque anche la segmentazione della "*mesh"*.

Casualità arbusti (struttura e tipologia)

Per creare strutture più realistiche e soprattutto per renderle 3D, è stato implementata una certa casualità nel sistema di generazione procedurale degli alberi. Questa casualità è strettamente correlata dalla variazione di angoli del vettore direzione della tartaruga. L'idea è quella di aggiungere in specifici punti della *stringa* ottenuta alcuni caratteri speciali (+, -, &, ^, \, /, |), che determinano la rotazione della tartaruga su uno degli assi (X,Y,Z).

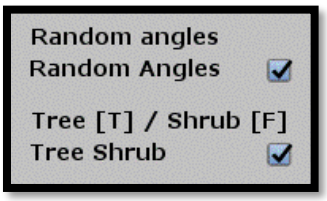




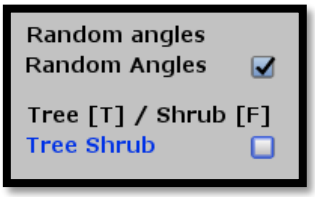
Come anticipato, la casualità permette di "trasformare" in 3D anche gli alberi 2D. Nella figura soprastante possiamo vedere come lo stesso albero subisca un processo di casualità e cambi di conseguenza la sua struttura.

NOTA: il primo, terzo e quarto elemento rappresentano lo stesso albero randomizzato; il secondo elemento nella figura è tale albero trasformato in arbusto.

Nell'*editor* basta spuntare queste due opzioni:



Albero

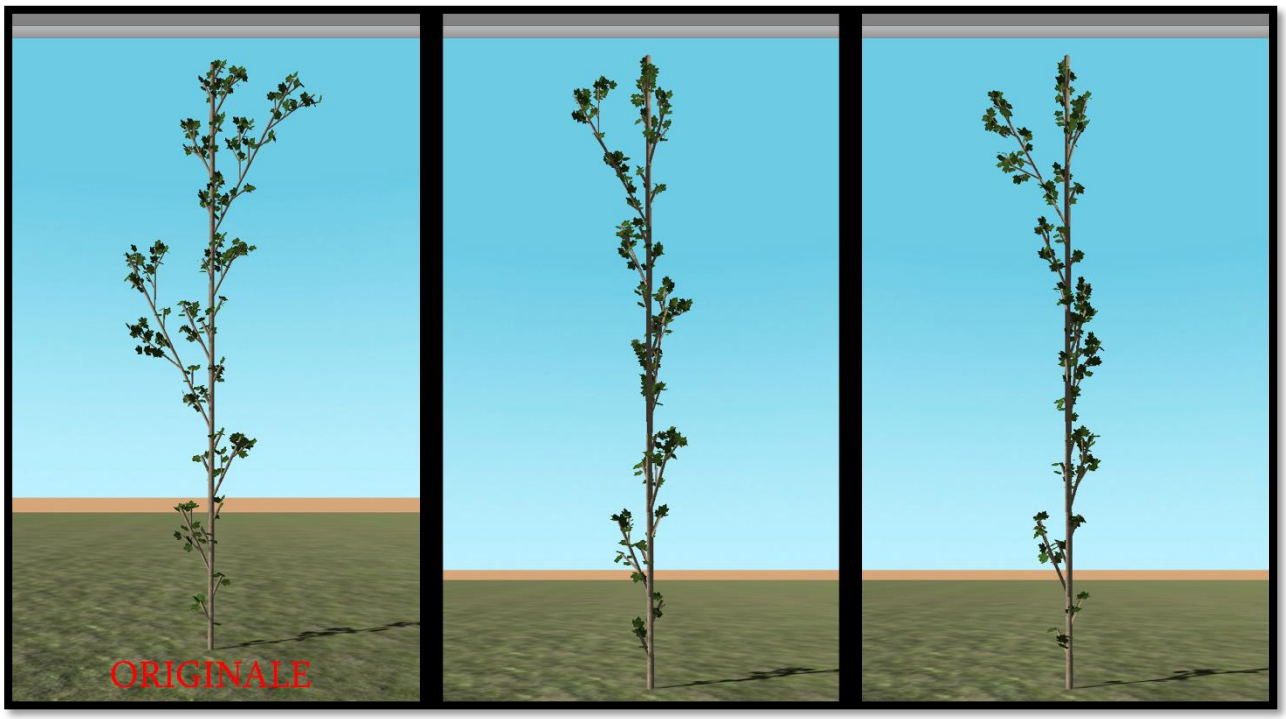


Arbusto

I parametri di casualità toccati sono i seguenti:

- Variazione degli *angoli* di generazione dei rami. ☑
- Scelta tra generazione di albero o arbusto. ☑
- Variazione delle *regole di produzione*.
- Presenza di più *variabili*.
- Associazione di *regole di produzione distinte* a variabili distinte o alla stessa variabile (*stocasticità*).

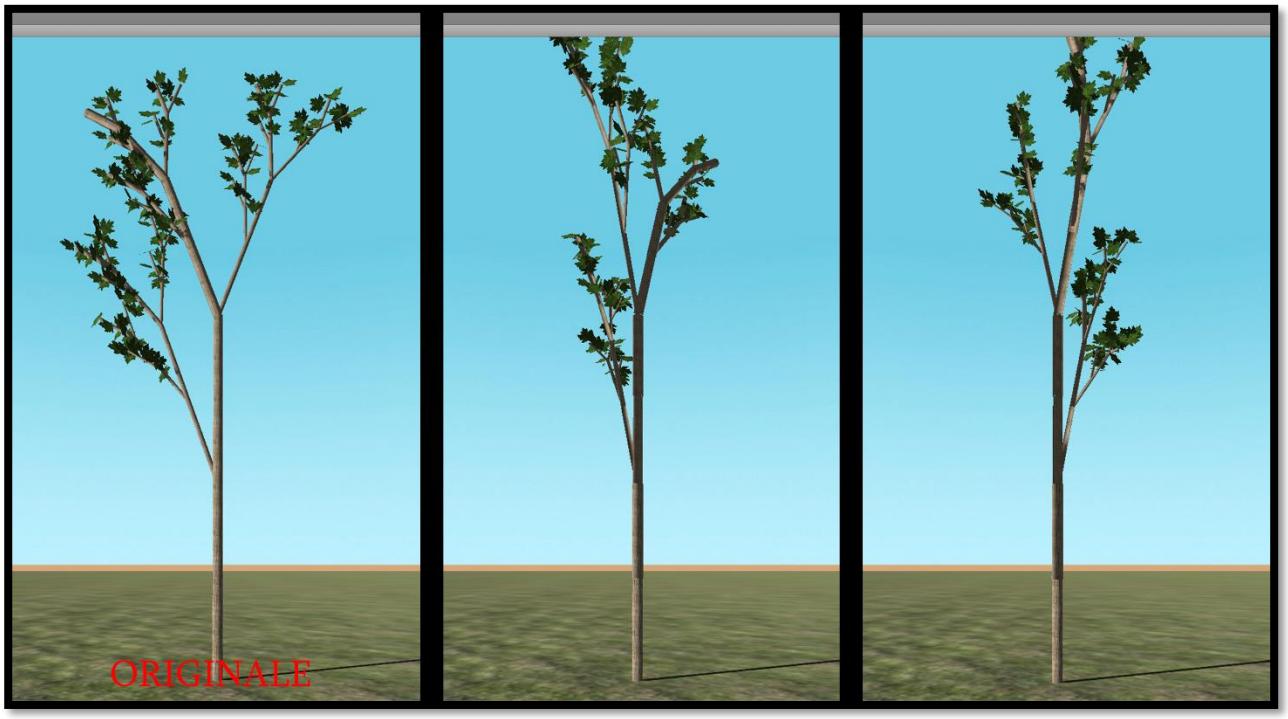
Altri esempi



Albero A

Albero B

Albero D



Albero E

Arbusti (extra)

È stata inserita inoltre la possibilità di creare arbusti adoperando sempre la casualità. Tale struttura è possibile poiché viene generato l'intera struttura semplicemente non considerando i diversi alberi assiali ma un unico principale. Gli angoli determinano la variazione della direzione della tartaruga.

Figura 8: Arbusto

Casualità foglie

La casualità riguarda anche le *foglie*. Impostando le diverse *opzioni di casualità* per le foglie è possibile far cambiare di molto la resa finale dell'albero:

Per quanto riguarda i punti in cui vengono generate le foglie, di base questi sono posizionati proprio ai vertici della base superiore del segmento:

I punti di *spawn* sono su tali vertici ed esattamente al centro di essi.

Per aggiungere la *casualità* è stata inserita una certa *distanza casuale*. Partendo dal punto originario di *spawn* si fa scendere/salire la foglia di una certa quantità. Da notare che la foglia sale solo nel momento in cui c'è un altro pezzo di tronco successivo, altrimenti scende solo.

Altri parametri aggiunti per rendere il tutto più casuale sono:

- Diverse *tipologie* di foglie (textures). ☑
- *Scelta casuale* della tipologia di foglia (texture). ☑
- Variabilità della *dimensione* delle foglie. ☑
- Variabilità del *numero di foglie* su ciascuna parte di ramo (dovuta al fatto che una foglia venga generata o meno su una specifica sezione del ramo). ☑
- *Rotazione angolo* delle foglie casuale.
- *Posizione* delle foglie sulla superficie del cilindro e non sull'asse (al centro).

Le diverse tipologie di foglie:

L-System stocastici

Gli alberi sono suddivisi in: *deterministici* (prefissati) e *stocastici* (hanno un certo livello di probabilità di variare).

Tutte le piante impostate finora sono *deterministiche*, il che significa che la loro struttura risulta essere pressocché identica, poiché segue una crescita artificiosa e regolare. È possibile introdurre delle *variazioni* che permettono di raggiungere un certo livello di casualità nell'interpretazione a tartaruga, nell'L-System o in entrambi.

Tutto ciò è ottenuto con degli *L-System stocastici*. Essi sono composti da una quadrupla:

$$G_\pi = \langle V, \omega, P, \pi \rangle$$

dove:

- $\boldsymbol{V}$ (*alfabeto*) è un insieme di simboli contenente sia quelli che possono essere sostituiti (*variabili*) che quello che non possono essere sostituiti (*costanti* o *terminali*).
- $\boldsymbol{\omega}$ (*start*, *assioma* o *iniziatore*) è una stringa di simboli di V che definisce lo stato iniziale del sistema.
- $\boldsymbol{P}$ (*insieme di regole di produzione*) definisce il modo in cui le variabili possono essere sostituite con combinazioni di costanti ed altre variabili.
- $\boldsymbol{\pi}$ (*distribuzione di probabilità*) $\pi : P \to (0,1]$, è una funzione che mappa l'insieme di produzioni nell'insieme di *probabilità di produzione*. Una assunzione importante è che, che ciascun simbolo $a \in V$ (alfabeto dei simboli), la somma delle probabilità di tutte le regole di produzione associate ad esso è 1.

Chiameremo la derivazione $\mu \Rightarrow \nu$ una *derivazione stocastica* in G_π se, per ogni occorrenza della lettera a nella parola μ, la probabilità di applicare la regola di produzione p con il predecessore a è uguale a $\pi(p)$. Pertanto, produzioni diverse con lo stesso predecessore (*simbolo* dell'alfabeto) possono essere applicate a varie occorrenze della stessa lettera in una fase di derivazione.

Supponiamo ad esempio di prendere un albero che abbia queste due regole di produzione:

```
4  axiom=F
5  angle=22.5
6  number of derivations=3
7
8  F=(0.5)FF-[-F+F+F]+[+F-F-F]
9  F=(0.5)FF
```

Entrambe le regole sono associate allo stesso simbolo F (sarebbe la lettera a di cui parlavamo) ed hanno entrambe la stessa probabilità ($0{,}5$) di essere scelte.

NOTA: come già detto in precedenza, la somma delle probabilità delle regole deve dare sempre 1.

La *derivazione stocastica* ci permetterà di scegliere, ad ogni derivazione e per ciascun simbolo incontrato, una delle due regole in questione. Nell'esempio corrente, essendo entrambe probabilità del 0.5%, avremo un 50% di probabilità di scegliere l'una o l'altra.

Questo ci permetterà di ottenere ogni volta un albero sempre diverso.

NOTA: I gradi di *casualità* aggiunti sono indipendenti dalla "*stocasticità*" e rappresentano un fattore *extra*.

La parte di codice che si occupa di derivare correttamente l'intera stringa di produzione dalle regole è la seguente:

```
22    public static void Derive(string axiom, float angle, int derivations, Dictionary<string, List<Production>> productions, out string moduleString)
23    {
24        moduleString = axiom;
25        for (int i = 0; i < Math.Max(1, derivations); i++)
26        {
27            string newModuleString = "";
28            for (int j = 0; j < moduleString.Length; j++)
29            {
30                string module = moduleString[j] + "";
31                if (!productions.ContainsKey(module))
32                {
33                    newModuleString += module;
34                    continue;
35                }
36                var production = ProductionMatcher.Match(module, productions);
37                newModuleString += production.successor;
38            }
39            moduleString = newModuleString;
40        }
41    }
```

In particolare, chiamando il metodo "*ProductionMatche.Match*", otteniamo ogni volta la regola giusta da applicare, probabilisticamente parlando. Tale metodo infatti ci permette di scegliere una regola di produzione tra quelle presenti nella lista delle regole di produzione (*matches*) associate uno specifico predecessore (*productions[module]*). Per far valere la probabilità, viene generato un valore casuale (*chance*) e viene confrontato ogni volta con la probabilità associata a ciascuna regola di produzione. Se il valore di probabilità raggiunto (*accProbability*) è maggiore o uguale al valore casuale (*chance*), allora viene restituita tale regola di produzione; altrimenti si aggiorna il valore di probabilità (*accProbability*), sommandolo alla probabilità della successiva regola, e si esegue nuovamente il controllo.

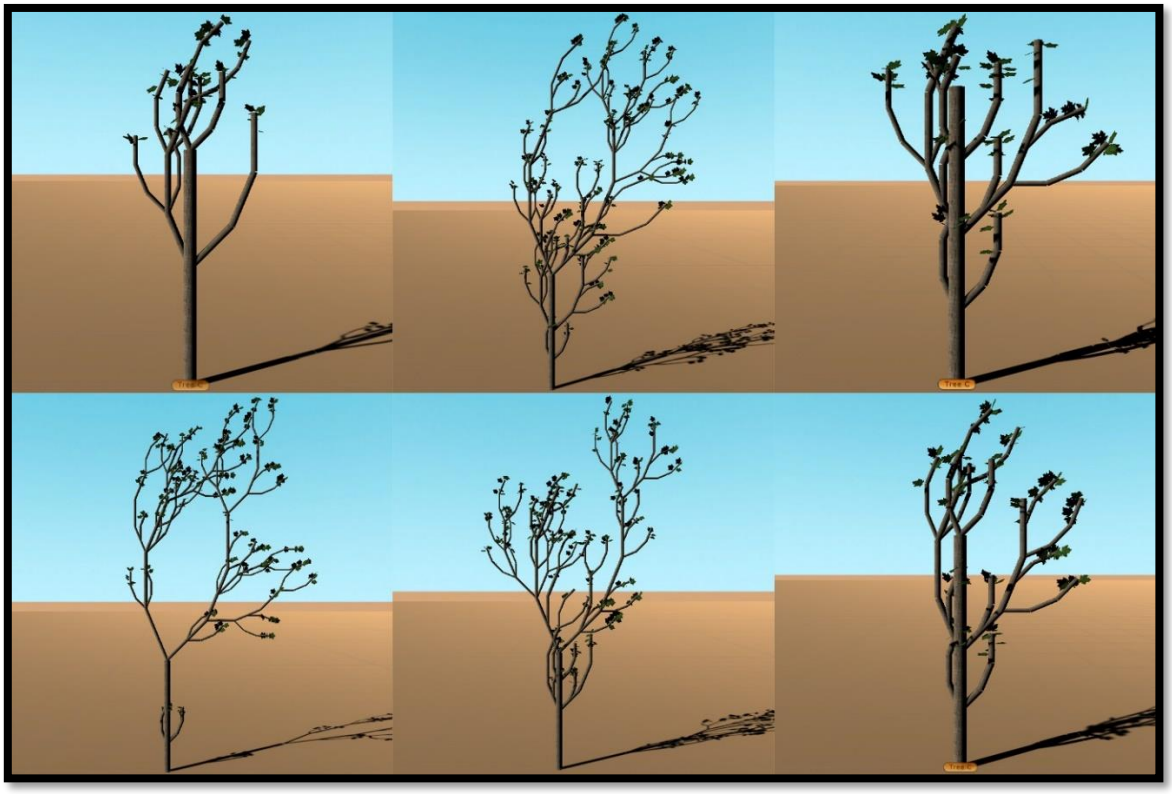
```
19    public static Production Match(string module, Dictionary<string, List<Production>> productions)
20    {
21        if (!productions.ContainsKey(module))
22            return null;
23        List<Production> matches = productions[module];
24        if (matches.Count == 1)
25            return matches[0];
26
27        float chance = UnityEngine.Random.value, accProbability = 0;
28        Debug.Log("chance: " + chance);
29        foreach (var match in matches)
30        {
31            accProbability += match.probability;
32            if (accProbability >= chance)
33                return match;
34        }
35        // TODO: throw an assertion!
36        throw new Exception("Should never happen!");
37    }
```

Alla fine, viene sempre effettuato un controllo per vedere se la somma delle probabilità è pari ad 1:

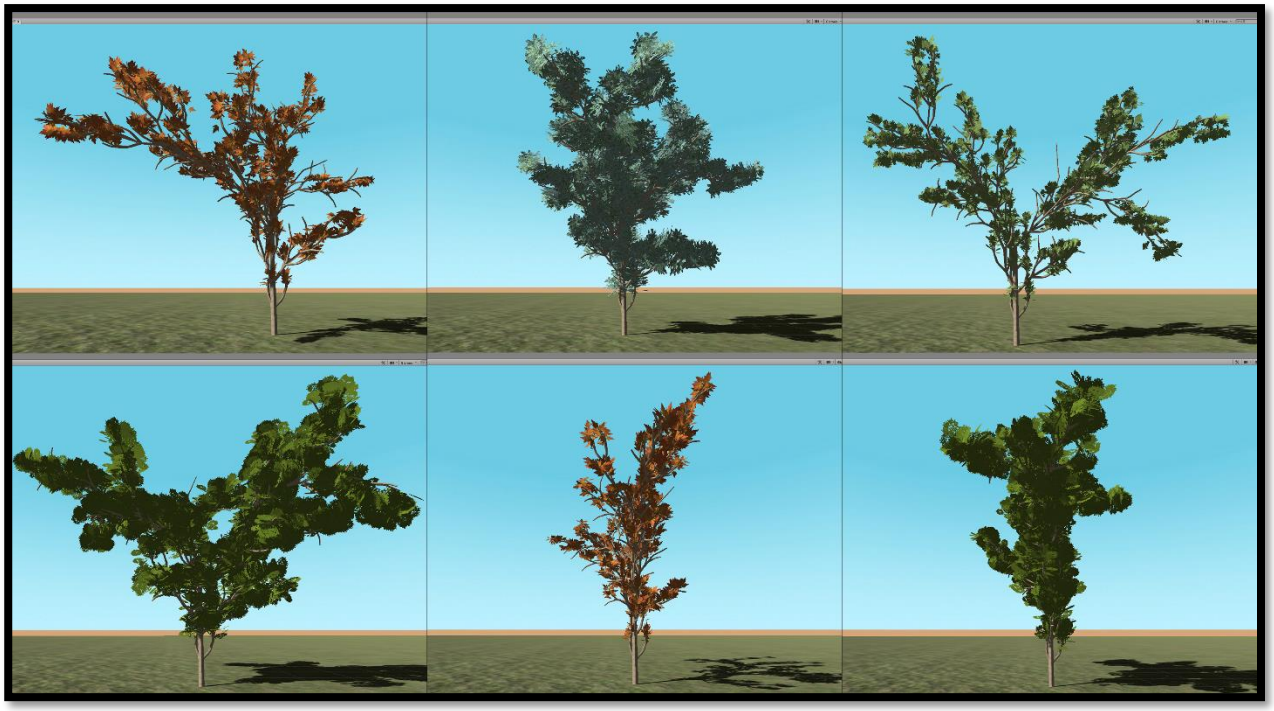
```
121           if (!ProductionMatcher.CheckProbabilities(productions))
122               throw new Exception("There's one of more production rules with probability < 1");
```

Vediamo alcuni esempi impostando tutti i diversi parametri di *casualità* (sia foglie che tronco/rami) e la *stocasticità*.

Quelli che seguono sono degli esempi 2D (*albero C*):

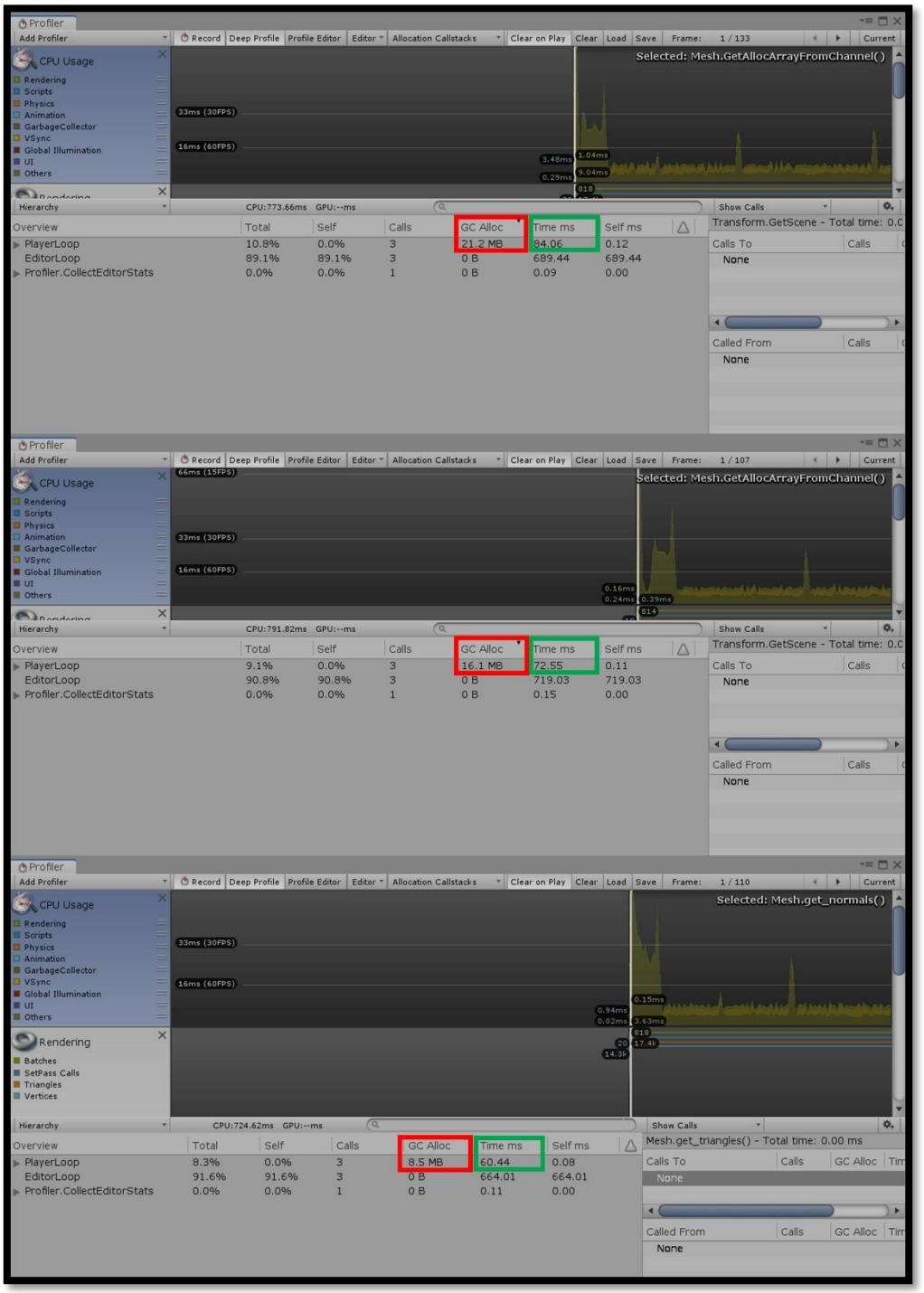


Aggiungendo ulteriori gradi di casualità e ottenendo così anche delle strutture 3D, avremo risultati di questo genere (*albero C*):



Performance

Dato che la generazione delle stringhe degli arbusti si basa di iterazioni, non si può dare per scontato il fattore performance. Si è lavorato anche su questo aspetto mirando ad ottimizzare i tempi di generazione per i casi con un numero considerevole di iterazioni (**es.** ≥ 4).



Version 1.1

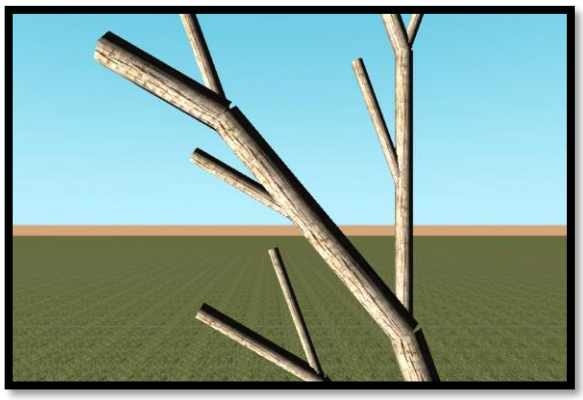
Sistema di generazione della Mesh

Nella versione 1.1 è stato creato un nuovo sistema di generazione della Mesh dell'albero. Questo sistema si estende su diversi aspetti quali:

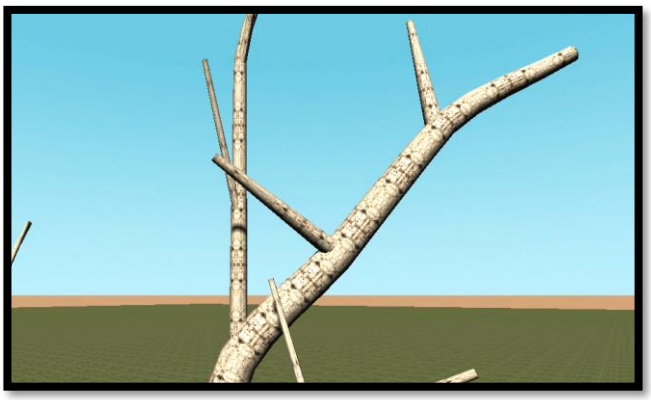
- Organicità della **forma** e delle **curve** della Mesh.
- **Casualità** dell'andamento del **tronco** principale.
- **Variazione** delle **dimensioni** dei rami e del tronco più marcata.

Forma e curve

Al fine di ottenere delle forme più sinuose e fedeli, si è fatto uso di *Spline* e *curve di Bézier cubiche*. Questo ha consentito di passare da dei semplici segmenti di ramo connessi tra loro ad un'unica *mesh*, corrispondente ad una *spline* composta da diverse curve di Bézier cubiche.



Prima



Dopo

Casualità tronco principale

Anche il tronco principale è stato reso più realistico. Per farlo sono state sfruttate ancora una volta le *curve di Bézier* e le *Spline*. In particolare, per ciascun punto di controllo delle curve di Bézier, è stata resa casuale la sua posizione. Questo ha reso la forma del tronco più "dinamica" e non rettilinea come nella versione precedente.

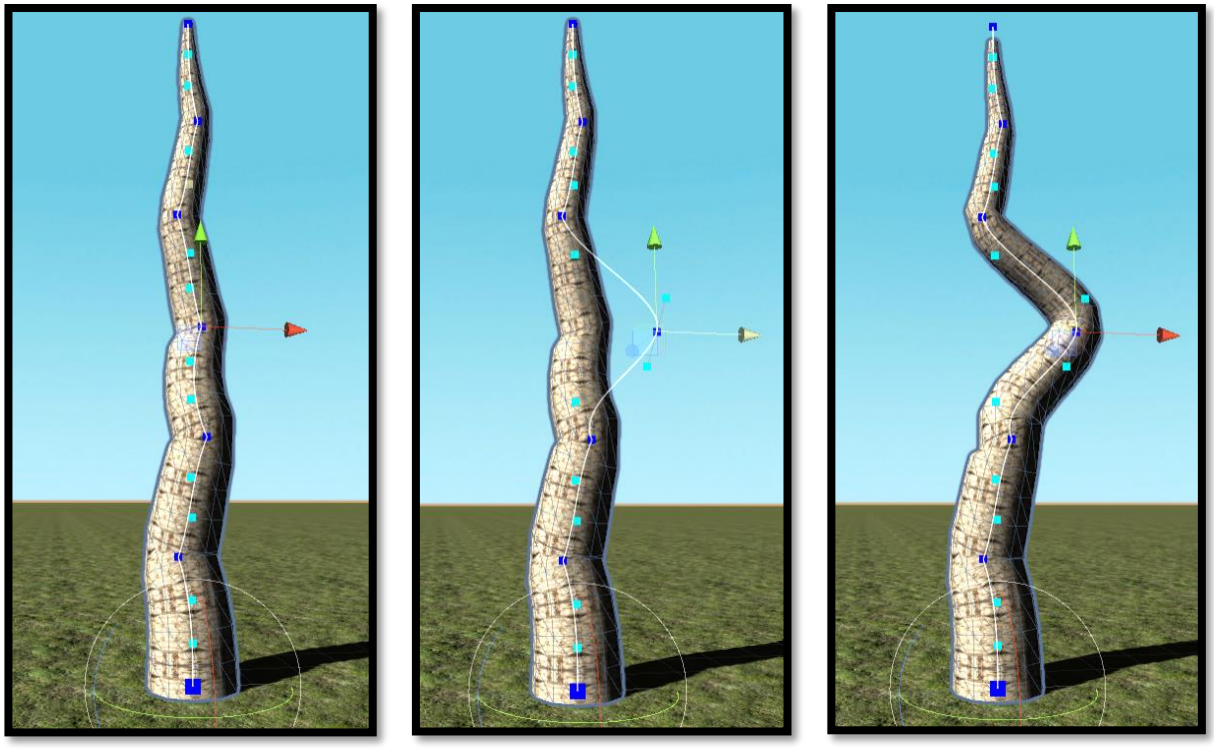
Dimensione rami e tronco

Nella versione precedente i rami cambiavano semplicemente diametro. Essendo la struttura dell'albero gerarchica, si passava dal tronco principale, che costituisce la parte d'albero col diametro più grande in assoluto, ai rami figli che partono da esso, caratterizzati da un diametro pari alla metà del diametro del livello precedente. Se ad esempio il tronco partiva da una dimensione di *0.50*, il primo ramo figlio avrebbe avuto *0.25* di diametro. Così via per i figli dei figli.

Tale struttura è stata preservata nella nuova versione ma sono state apportate ulteriori aggiunte:

- Il tronco principale ha un diametro di partenza variabile tra *0.5* e *0.9*. Il diametro finale invece è variabile tra 0.1 e 0.3.
- Sia il tronco che i rami partono da un certo diametro e iniziale che diminuisce man mano fino alla fine.

Inoltre, sia tronco che rami sono attraversati da una *spline* modificabile. Questo significa che, una volta generato l'intero albero, è possibile andare ad agire singolarmente sulle parti di cui è composto e modificarne la forma e dimensioni.

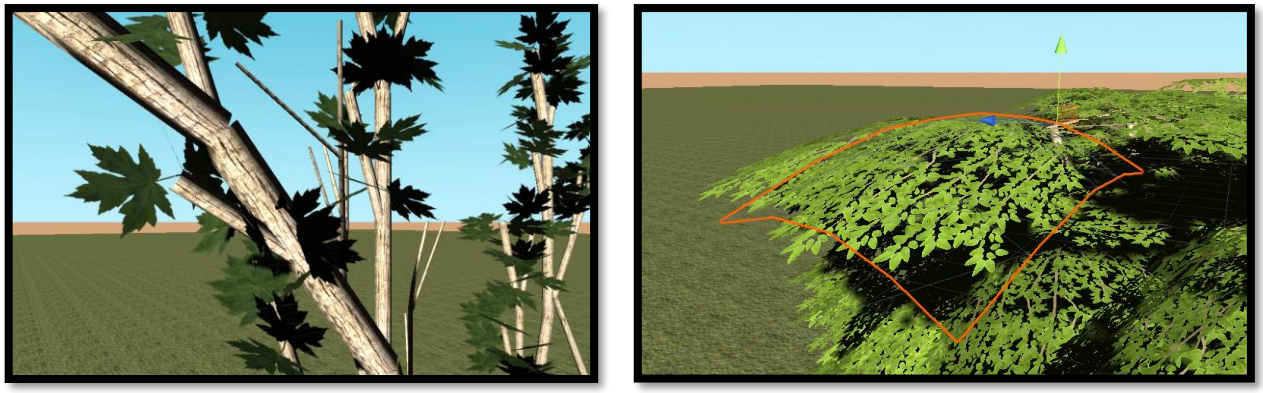


Sistema di generazione delle foglie

Per ottenere delle foglie più realistiche è stato necessario “cestinare” il precedente sistema per fare largo ad uno completamente nuovo.

Il *precedente sistema* prevedeva delle semplici texture applicate su dei “*plane*” 2D, a doppia faccia (normali ambo i lati con possibilità di visualizzare dunque la texture su entrambi).

Il *nuovo sistema* invece si basa sull’utilizzo di una mesh curva (quindi un piano curvato), generata con *Blender*, con le stesse caratteristiche del predecessore. La curvatura però consente di poter visualizzare meglio le foglie, anche in angolazioni che ne privavano la vista nella precedente versione.



Prima *Dopo*

Nel *nuovo sistema* inoltre si fa uso di una texture che è già di per se un ramo composto da foglie. Attaccarlo alle estremità dei rami generati proceduralmente simula a tutti gli effetti un rametto composto da tante foglie.

Un’altra differenza rispetto al sistema precedente è il *sistema di orientamento delle foglie* sui rami. Mentre prima l’orientamento era generato casualmente e quindi ogni singola foglia poteva essere orientata in

qualsiasi direzione, ora le foglie sono posizionate sempre perpendicolarmente al terreno. A partire da tale posizione viene poi effettuata un'ulteriore (leggera) modifica della rotazione per ottenere più casualità.

NOTA: il vecchio sistema di generazione delle foglie permane tuttora.

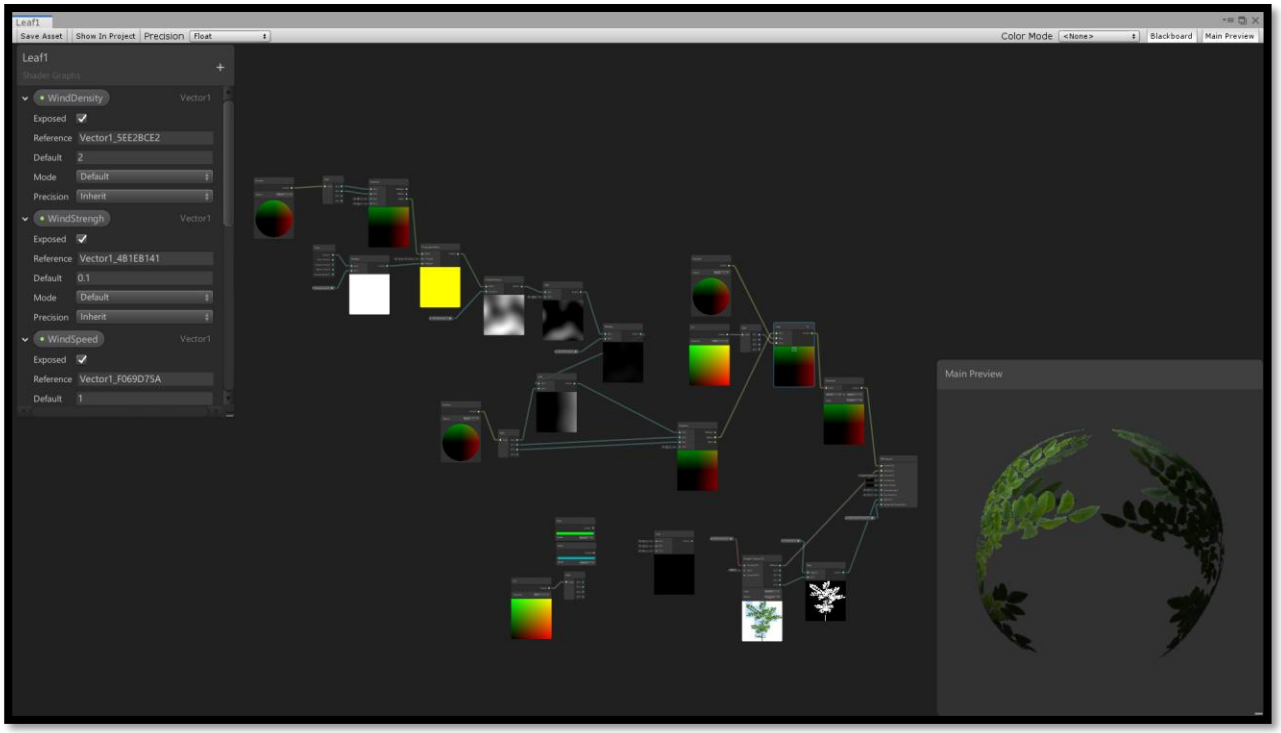
Vecchio

Nuovo

NOTA: sia il vecchio che il nuovo sistema condividono i parametri per generare in modo casuale *diversi tipi di foglie (e colori), dimensione delle foglie variabile* e *quantità delle foglie variabile*.

Simulatore di vento per le voglie

Per ottenere l'effetto del vento si è fatto uso di uno *shader* applicato sulle singole foglie, tramite l'ausilio dello *Shader Graph* di Unity3D. Sono stati creati 6 shader per 6 foglie diverse. Lo shader permette di creare un effetto che dà l'impressione della presenza di vento. Questo è dovuto allo spostamento sinusoidale della mesh che ottiene un effetto di onda, dando appunto l'idea di movimento ondulatorio, come quello generato dal vento.


Leaf1
Save Asset
Show In Project
Precision
Float
Color Mode
<None>
Blackboard
Main Preview
Leaf1
Shader Graphs
WindDensity
Vector1
Exposed
Reference
Vector1_5EE2BCE2
Default
2
Mode
Default
Precision
Inherit
WindStrengh
Vector1
Exposed
Reference
Vector1_4B1E8141
Default
0.1
Mode
Default
Precision
Inherit
WindSpeed
Vector1
Exposed
Reference
Vector1_F069D75A
Default
1
Main Preview

Riferimenti

L-Systems
Wikipedia - L-Systems
Wikipedia - Production (computer science)
Wikipedia - Turtle
PDF - The Algorithmic Beauty

Papers/PDF
L-Systems presentation
Procedural Content Generation for Games: A Survey

Videos (YouTube)
Interactive Tree Creator
Trees in the Wind - with Unity Shader Graph