Dynamic Animation and Re-Modelling of L-Systems

Nicholas Hampshire

August 21, 2009

Masters Thesis

MSc Computer Animation and Visual Effects

N.C.C.A Bournemouth University

Contents

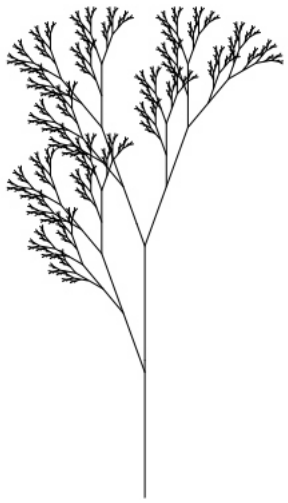

Figure 1: A simple 2D tree generated with L-System grammar [Prusinkiewicz and Lindenmayer 2004].

1 Introduction

1.1 L-Systems

Aristid Lindenmayer first proposed his mechanism of axiomatic biological development using formal grammar by studying and simulating the development of algae [Lindenmayer 1968]. The rewriting structures of growth featuring in his pivotal work soon became known as L-Systems.

In this project, a technique is presented to allow dynamic animation and re-modelling of L-Systems using elastic deformable body mechanics. The implementation is presented as an interactive program, but other uses are discussed.

L-Systems provide the platform for quickly creating not only complex, but also realistic representations of plant structures. Prusinkiewicz and Lindenmayer demonstrate the potential with a wide range of examples in their studies [Fig. 1][Prusinkiewicz and Lindenmayer 2004]. In computer graphics, generating plant structures in this way is a highly practical approach to the creative problem of having to manually do so.

1.2 Problem Statement

With such a procedural approach to creating realistic complex structures comes the unfortunate restriction that is the lack of precise regionalised control over shape. In the creative environment, artists need the capability of applying

specific unique form and character to any geometrical structure they generate, in a non-procedural manner. The task of manually adjusting the point and line geometry of a generated structure with traditional modelling methods is just as prohibitive as modelling it in the first place.

Further to this is the static property of the L-System geometry. Creating convincing motion and animation in the branches or leaves of the structure is another very difficult task. In a dynamic environment, there may even be the need to have other objects interacting with the system, and the system in turn affecting these objects.

2 Previous Work

2.1 Modelling of L-Systems

Decomposing the generation process of L-Systems has been a previous field of study. Both physical and non-physical approaches to altering the structure have been examined in order to create unique shape based on environmental factors such as forces of gravity and wind. Prusinkiewicz and Lindenmayer [Prusinkiewicz and Lindenmayer 2004] propose two examples: The affect of external forces such as gravity on the structure; The simulation of tropism for plants and trees, giving their branches additional realism with bend during growth. These affecters are augmented as an additional stage during the generation process itself.

Prusinkiewicz et al. [Prusinkiewicz et al. 2001] define methods of modelling individual plant stems and leaves in a highly controllable and detailed way. Their approach uses a combination of pre-defined L-System rules and extended syntax, where curves and surfaces used to shape regions are defined in the grammar. This in-depth approach combining procedural setup and uniquely modelled stems produces varied and precisely detailed results, focussing equally on the underlying structural shape and the geometrical surface output.

Using an inverse-kinematics system, Power et al. [Power et al. 1999] present a method of interactive manipulation of models whilst retaining a high level of botanical realism. Using flexural and torsional stiffness attributes in the plant branch joints, the branches are connected as rigid segments that can be interactively repositioned to obtain a new arrangements. Since they are rigid, each branch segment has complete rotational information allowing accurate bending and twisting. Their system also demonstrates clipping and pruning of the plants.

2.2 Procedural Animation of L-Systems

Noser et al. [Noser et al. 1992] present a technique of animating L-Systems via time integration of the construction points in velocity fields, defined within the system's generation syntax. They demonstrate how elastic deformations can be achieved on each articulation of the structure, with unique attributes such as bend specified for each giving greater realism and variation to the possibilities available. This technique is an interesting approach to obtaining realistic animation, with the potential for regionalised control. However, due to the paradigm in which the forces are applied, that is, within the production rules of the L-System, it is difficult to manipulate the exact affect of the forces on the structure other than in a procedural pre-emptive way. A force description has to be mathematically defined prior to the generation of the animated geometry sequence. This is more a problem of feedback than a lack of control afforded by the method. A syntactical approach to applying forces on a point structure for specific animated results is perhaps unmanageable for any effect other than fluid-like force interactions, for which the method was principally presented.

3 Technical Background

In the realm of visual effects, where creativity and visual result are favoured over complete physical or botanical accuracy, the techniques previously discussed have displayed many concepts which are useful to take into account when designing interesting and versatile, if less accurate, re-modelling and animation of L-Systems.

3.1 Dynamic Approaches

By using a physics-based approach, it is possible to define an L-System that could behave according to its internal physical properties. It can then also be subjected to external physical factors such as environmental forces. To obtain the natural flexibility and springy nature of branches and plants, the representation will need elastic deformation properties. A brief examination of some existing deformation models is appropriate in order to find a good basis for this project. Since L-Systems are concerned with point and mesh data, only Lagrangian-style discretised point deformation methods are considered, as opposed to Eulerian volumetric types.

Terzopolous et al. [Terzopoulos et al. 1987] first introduced deformable bodies to the computer graphics field, realising the potential of a physically based approach to simulating objects that are not intended to be perfectly rigid. Many new systems have been proposed since this introduction for different kinds of effects, some more accurate yet computationally expensive, while some cheaper but inaccurate.

Breen et al. [Breen et al. 1994] consider a field of particles connected by spring constraints to model the bending, stretching and shearing forces found in cloth. They show how their technique is able to simulate cloth with good likeness to real cloth references. Nowadays this is considered a spring and mass approach, where the particles are point masses, in this case arranged at the crossing positions of the cloth weave, and the connecting springs were arranged along the weave. For this reason, the topology of the connections will greatly affect the nature of the dynamic simulation.

A model of representing solid objects by Teschner et al. [Teschner et al. 2004] uses a similar point mass technique where potential energies derived from trying to preserve surface area and solid volume. This technique allows triangulated meshes or tetrahedral solids to be deformed elastically or plastically at interactive speeds, with the capability of inter-object relationships such as collision response forces. Since the system is designed for use with triangulated meshes, the shape of an assumed triangle input mesh should not affect the nature of the simulation. This is a sought-after property, although the initial condition of only triangulated meshes, or their tetrahedral solid equivalent, may remain a restriction.

Finally, an intriguing system introduced by Muller et al. [Muller et al. 2005], which is completely topology independent, is based on shape matching. Their approach takes the initial point cloud of an input mesh and simulates them as

free particles, requiring no connectivity information. At each simulation step, the un-deformed mesh position is moved to the locale of the point cloud via a transformation calculated using the method of least squares. The position of each point is then compared to that of its counterpart's position in the original un-deformed transformed mesh. Because the ideal position of the deformed point is known, the integration step which advances the point forward in time based on the force acting on it can be prevented from overshooting its destination. This aspect of their system's design is a superbly simple solution to enforcing simulation stability in any circumstance.

3.2 A Suitable Dynamic Approach

In this project, the focus is on the deforming nature of an L-System. As discussed, an L-System implementation can produce point and line data, before additional schemes are used to apply a visual surface to the structure. The intention is to find a dynamic approach to animating L-Systems that has the advantages of the modern dynamic systems, such as stability and flexibility of use, whilst also being able to operate on this minimal data set.

4 Solution

Unlike previously mentioned animation approaches which use additional syntax within the L-System grammar to drive the dynamic animation [Noser et al. 1992], the problem is instead approached by taking the static output geometry (the bare point and line data) from an L-System generation scheme, and using this as input to a new system specifically designed to manipulate and animate hierarchical data structures. The reason for this decision is to cleanly separate the generation and animation stages. Such an abstraction is important to allow for a more flexible working pipeline: The generative phase of an L-System should be considered the initial modelling phase of a traditional pipeline; Animation is a phase that occurs secondarily to modelling. It also means that the animation can be produced visually and interactively. Should the two processes be interconnected, the partition between these clear production phases becomes clouded. Appropriately, a modeller should be responsible for creating the initial shape of the tree, and an animator of its movement.

4.1 L-System Setup

Using the built-in L-System generation node in Houdini [SideFX 2009], botanical L-System geometry could be quickly created and used as a working example set to develop the system with. Support was developed for the Wavefront OBJ (.obj) [Wavefront 1995] and ASCII Houdini GEO (.geo) [SideFX 2009] geometry file formats. These specifications produce human readable geometry information. This was important initially so that an L-System data type could be developed which could be easily extended to interpret other L-System generator output geometries.

An advantage of using the Houdini .geo file format is that of custom attribute passing. Specific point attributes created in the L-System generation process can be included in the geometry file. These comprise details such as point generation and width, two common L-System point attributes.

The C++ programming language was chosen as a developing platform due to its object-oriented and cross-platform nature, and also because of its popularity in the visual effects industry, such as in large libraries like the Maya API [Autodesk 2009].

An L-System class was therefore created (LSystem_3i) which could load data from geometry files into a common data type, capable of performing the same general operations irrespective of the attributes that were or were not available in the geometry file. When the information is initially loaded from file, three sets of data are allocated for: States (State_3i), an array which individually holds position, velocity and force information for each of the points loaded (LSystem_3i::pSte); Nodes (LSysNode_3i), an array of exactly equal size to the array of states, holding the L-System attributes at the point, where a node array index can be interchangeably used in both the node and state arrays for the related state/node attributes (LSystem_3i::pNde); Bones (LSysBone_3i), an array of line segments which hold two indices, one to the start and one to the

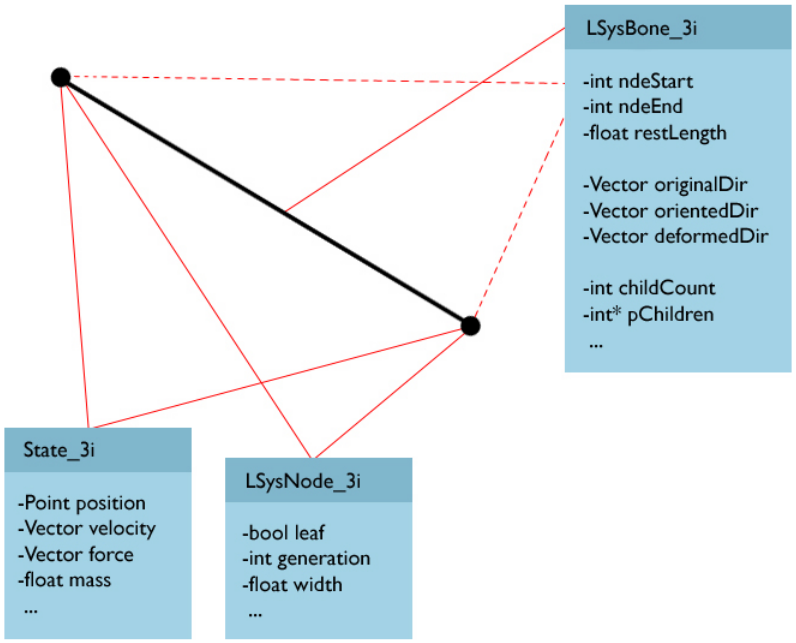

Figure 2: The State, Node and Bone classes, showing the most poignant attributes.

end state/node from which the line segment is constructed (LSystem_3i::pBne). [Fig. 2].

Nodes carry a Boolean leaf attribute that indicates whether or not it is at the very end of a branch. Bones carry a "restLength" attribute defining the original length of the line when loaded. The vector "originalDir" is a unit direction vector directing from the start state to the end state positions of the bone in the initial pose of the system. The vector "orientedDir" will be explained later [4.4], and defines the direction of the original vector after the system has been transformed to a new orientation. The vector "deformedDir" is used during the simulation step and will also be explained later [4.2.2], and is initialised to the same value as the "originalDir" vector.

If the generation attribute of the node is not available in the input geometry file, then it is assigning the number of the point to which it relates. This may seem careless, but in-fact L-Systems grow from an initial point, the root point, and after finding this the true generations are then procedurally generated by working up from the node with the lowest generation. This root point is the so called "original root node". Next, an "original root bone" is found, being the

bone whose start node index is that of the original root node.

In order to construct a hierarchy of all bones, a recursive L-System class method is used (LSystem_3i::mSetupBoneHierarchy(...)) which accepts the index to any bone in the bone array. The children of this bone are then find (The bones in the hierarchy which are directly connected to the input bone end node) by comparing the input bone end node indices with all other bone start indices. Those bones that have a start node index identical to the input bone's end node index are descendants of the input bone, and are stored as the children of the input bone. The bone attributes "childCount" and "pChildren" are for this purpose, holding a count of the number of children the bone has, and an array with the actual bone indices to the child bones in it, respectively. This method is called once from the main setup method (LSystem_3i::mSetup(...)) with the original root bone as the argument, but subsequently calls itself until all branches have been traversed to their leaf node.

Whilst this process is carried out, the generation attribute in all nodes is set so that it begins as 1 in the original root node and increases by 1 every additional node step along the hierarchy. The leaf attribute is also set true in a node when it is found to be at the very end of a branch, or false otherwise.

If the width attribute is not available in the input file, it is created by dividing 1 by the node's generation. This creates a fall-off where the original root bone will be thickest and all children will become gradually thinner, with a clamped minimum width of 0.1. At this point, all generation and width attributes are set however the data was obtained from file. The width attribute is subsequently normalised. This process was introduced after having developed the simulation somewhat, as a method if standardising initial behaviour resulting from changes in differing L-System width scales. It involves finding the greatest of all node widths, then dividing each node width by it. The resulting widths are in the range 0 to 1.

4.2 Dynamic L-System Evaluation

In order to simulate the state of the L-System over time, regular evaluation of the current attributes of the state array must occur. The time interval that is covered, referred to as the time-step, can be arbitrarily defined. However, smaller time-steps are required for greater accuracy. In order to achieve the current results, each evaluation of the L-System sees two separate integration steps occur. During the first, forces that have been applied to the state from external sources such as the environmental forces in the scene [4.3], and standard internal forces generated by the dynamic L-System [4.2.1], are concatenated and integrated. In the second, parameterised forces are generated by the dynamic L-System [4.2.2] and are integrated.

The L-System class contains an integration scheme (base class type Integrator_3i) to step the states over time after forces have been applied, and then flushing the applied forces. There are two derived integrator types, IntegEuler_3i and IntegRungeKutta4_3i, performing explicit Eulerian integration [Fig. 3] and Runge-Kutta-4 type integration, respectively. These schemes

$$X(t+dt) = X(t) + (V(t) * dt)$$
$$V(t+dt) = V(t) + (F(V(t), X(t), t) * dt)$$

Figure 3: Explicit Eulerian integration, where t is the current time and dt is the time-step to cover. X is the state attribute position, V is the state velocity and F is the force acting on the state.

$$F = -kx$$

Figure 4: Hooke's law of elasticity. F is the spring force, k is the spring constant and x is the displacement of the spring end from its equilibrium position.

were derived from examples presented by Fiedler [Fiedler 2006], and work directly with the Lagrangian discretised point architecture that was first chosen for the structure representation. The IntegEuler_3i Eulerian scheme is cheap but inaccurate, while the IntegRungeKutta4_3i scheme is expensive but more accurate. Both were implemented for comparative studies.

4.2.1 Standard Force Concatenation and Integration

Utilising spring and damper techniques outline by Breen et al. [Breen et al. 1994], an implementation was designed to generate spring forces on all states based on bone connectivity. This force is simple to calculate and apply.

Hooke's law of elasticity [Fig. 4] states that the extension of a spring is in direct proportion with the load added to it as long as this load does not exceed the elastic limit.

Assuming that each bone is a spring, with length of equilibrium as its "restLength" attribute, a spring force can be generated by finding the displacement at each step between this rest length and the current distance between the bone start and end states. The bones can be assumed to have no elastic limit like a real spring would, in order to simplify the simulation. The method LSystem_3i::mEvalForceStd(...) is used to generate and apply this force. First, the spring displacement is found, and subsequently a spring force, which is applied negatively to the bone start state, and positively to the bone end state, in the direction of the unit vector between the two current bone state positions. This results in the bone states being pulled together if the spring current length exceeds the bone rest length, or being pushed apart if they become closer than the bone rest length [Fig. 5].

$$L = (P_1 - P_0)$$
$$F = (\frac{L}{|L|}) * (restLength - |L|) * k$$

Figure 5: The standard spring force rule used. L is the vector between the current bone state positions P_0 and P_1, k is the spring constant and F is the spring force.

$V = V_1 - V_0$
$B = -V * b$

Figure 6: The spring damping term. V is the relative velocity between the two bone state velocities V_0 and V_1, b is the damping coefficient and B is the damping force.

A spring damping term is also applied to smooth the application of force [Fig. 6].

These forces shall be considered "stretch" forces, since they are concerned solely with maintaining the branches length. The application of these forces alone will not help retain the shape of the structure if it is subject to any external forces, it will only suffice to maintain the rest distance between the bone state positions. The states are subsequently integrated over half of the full time-step with a scheme such as [Fig. 3].

4.2.2 Parameterised Force Concatenation and Integration

In order to maintain the shape of the tree structure, additional forces must be applied to the states that draw bones toward their preferred direction. A preferred direction vector is known for the branches, the "originalDir" bone attribute, as calculated in the setup stage [4.1]. In combination with the "restLength" attribute, the exact original desired position for any one bone's states can be found, assuming the predecessors of the bone are completely un-deformed. Since alterations to bone directions toward the original root bone should actually affect the bone directions of each child bone, this information alone will not suffice. An initial implementation of this primitive method shows how the forces drawing every bone state toward its original direction will yield unsatisfactory results [Fig. 7]. Instead, the alteration of a root bone direction should propagate down into all child bones, giving a true representation of their orientation [Fig. 8].

The forces which are generated as a result of a difference in a bones original and deformed direction shall be considered "bend" forces, since they are concerned with the direction the bone is facing, not its length. In order to calculate bend forces, a recursive, hierarchy traversing system was used. The root call of this calculation process is the method LSystem_3i::mEvalForcePar(...). At the start of the procedure, all bones have their deformed direction vector attribute set the same as their original direction vector. A method LSystem_3i::mCalcForceParChildren(...) is then called with the original root bone as the initial argument. This method recurses over all bones in the hierarchy, where the input bone is considered the root bone for the current operation. It then call itself for all child bones of the root bone passed in.

The forces, which are applied to the input root bone end state, are generated in the following steps:

1. The exact current direction vector C between the input bone constructing states P_0 and P_1 is found: $C = (P_1 - P_0)$

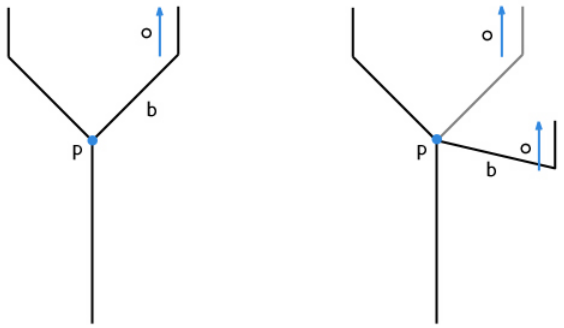

Figure 7: The direction vector of a root bone b is changed from its original direction. The original direction vectors o of each child bone is wrongly sought by them, displaying a lack of transform inheritance from the parent bone.

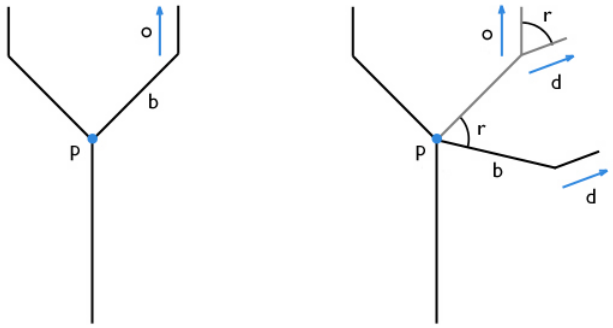

Figure 8: The direction vector of a root bone b is changed from its original direction. All child bone's original direction vectors o are transformed by rotation angle r about the deformed parent bone b start node p to obtain the new deformed direction vector d, which they now seek instead of original vector o.

2. The ideal directed length vector I is found by scaling the input bone deformed direction vector D by the rest length of the bone: $I = D * restLength$

3. For the current input bone, the goal difference vector G between its ideal end state position and its current end state position is found by taking the current direction vector from the ideal directed length vector: $G = C - I$

4. The goal force S is then calculated with using the spring equation where the direction of the spring force is G magnitude of the spring is $|G|$, which is then scaled by the spring constant k and the bone $width$ attribute, and applied as force to the bone end state: $S = G * k * width$

An important reason for generating the bend forces in this way is that, much like the goaling scheme illustrated by Muller et al. [Muller et al. 2005], the ideal destination of the state being integrated is known. This means that there can be a limit applied to the size of the position step (the magnitude of the goal difference vector, which is also stored in the state), preventing it from overshooting its ideal location. This has a major affect on the stability of the entire system. The spring constant of the forces involved can be considerably tighter with a far smaller adverse impact on stability using this method, allowing much more realistic, rigid plants to be simulated.

After the force has been applied to the current root bone end state, a quaternion rotation to map the current bone direction unit vector onto the bone deformed direction vector is found (the rotation discussed in [Fig. 8]) and applied to the deformed direction vectors of all children of the current bone with the recursive LSystem_3i::mCalcForceParRoteHierarchy(...) method. The quaternion rotation, which is applied with centre of rotation at the position of the current root bone start state, is found in the following steps:

1. Normalise the current bone direction vector C: $C = \frac{C}{|C|}$

2. Find the angle r between the current unit direction vector C and the deformed unit direction vector D: $r = \cos^{-1}(C \odot D)$

3. Find the normal axis A of rotation by finding the vector perpendicular to both C and D: $A = \frac{C \otimes D}{|C \otimes D|}$

After the entire hierarchy has been traversed, the states are integrated for the second time. On this occasion, a modified integration step occurs, which limits the maximum position step of the state by the stored magnitude of the state's goal difference vector. The modified schemes are present in all overloaded versions of the pure abstract integrator method Integrator_3i::mIntegratePar(...), which enforce the step size condition.

4.2.3 Resolution

An additional attribute associated with the L-System itself is stretch resistance. This attribute gives the user control over how much stretching is allowed to

occur in the bones of the structure. Since the bone stretch concerns length only, a simple resolution procedure can be used to enforce the degree of stretching allowed. This is performed in the recursive LSystem_3i::mResolve(...) method. This is called once with the original root bone as the initial root. Once again, the current bone length is calculated and compared with the rest length. Assuming the start state of the bone is fixed, the displacement of the end node is the difference between the bone rest length and current length. This displacement is scaled by the stretch resistance parameter and applied as a negative translation to the end state position along the current direction vector of the bone. This operation will naturally also prevent compression of the bone.

4.3 Forces

The ability for the user to interact, move and animate the L-System structure was the principal aim of the project. This functionality is incorporated entirely through user-malleable forces.

To accommodate for these operations, a scene object was created (Scene_i) which manages selection of items such as the different forces and L-System bones. It also takes charge of instigating the regular L-System evaluation process, and can be used as an entry point for all scene object operations.

Three force types were implemented, each allowing a specific different kind of manipulation of the L-System. Pulling forces (ForcePuller_3i) allow a force to be applied to a specific location on an L-System bone. The magnitude of the force applied is based on the distance it is moved from the bone once attached. Uniform vector forces (ForceUniVec_3i) allow a specific directed force to be applied uniformly to all states in the L-System. This is useful for applying regular force such as gravity. The user can direct and alter the magnitude of this force. Finally, velocity field forces (ForceVField_3i) create a voxel field of velocities which are advected and diffused as a fluid container using the stable interactive fluid solving technique proposed by Stam [Stam 2003], which is also regularly evaluated along with the L-System by the scene. This force type is useful for re-creating effects such as that of wind blowing or water moving through the structure.

In order for the scene to cope with multiple forces of each type, a container object was created for each force type, each derived from the base class Force-Bin_3i. These act as managers for all forces of a particular type.

4.3.1 Force Manipulation and Application

For the user to have proper control of the L-System, precise selection and manipulation of the forces was required. This problem was approached via ray selection based on techniques by Comninos [Comninos 2006] and Ericson [Ericson 2005], where mouse clicks to the interface viewport are translated into rays and cast into the scene object. Using ray selection enabled very specific parts of the scene objects to be selected, such as points along an L-System bone, or a specific part of the force manipulator tools.

After creating a pulling force at a point along a bone, a perpetual spring force is proportionally applied to both states of this bone, weighted according to the position of the pulling origin along the bone line segment. Since the force objects and L-System objects are highly individual entities, a standard method of applying the generated forces between the types was required. This caused the architectural decision to separate the L-System point types into the now separate State_3i and LSysNode_3i classes. The State_3i class is not directly associated with the LSystem_3i class, instead stands alone as a physical state associated with the integration classes. With this design, both the force objects and the L-System objects operate on the states as though they were a central resource, although they are still belong inside the L-System object. The L-System can return its State_3i array via the LSystem_3i::mGetStateSet(...) method, and also the size of the state array with another. Each force type is then responsible for the application of its force onto the point set no matter how it has been designed.

At each regular evaluation step, the scene collects the L-System state set, and passes it to each of the different force manager types for their force contribution to be applied to it. Architecturally, this allows the forces to store little or no information about the state set it shall operate on, but have complete information about them when applying the forces.

This design was in-fact essential to the operation of the pulling force specifically, which acts like a dynamic spring constraint between the puller manipulator position and the arbitrary states to which it is connected. For the connection to be made between a pulling force and an L-System bone, a selection on the L-System is first required. The selection can then be obtained by the scene in the form of the two state indices which define the bone in the L-System, and the parameter along the line defined by the two state positions where the exact selection occurred. A pulling force can then be created which stores these parameters. When the force object applies its force contribution to the state set it receives, it can assert that the state indices it holds are in the valid state array range before application. This is a robust mechanism, far more so than perhaps the dangerous alternative of allowing the pulling force object to store pointers directly to the state addresses within the L-System, although this would be simpler to implement.

The uniform vector force object applies force to all states in the set it receives according to the direction of the arrow manipulator tool.

The velocity field force operates in a slightly different nature than other types. The force manipulators, which are similar in appearance to those of the uniform vector, source velocity into single a voxel field maintained by the ForceBinVField_3i class, which also contains all of the individual velocity field forces. The voxel field is poised at the position of the L-System original root node by the scene in order that it always encompasses the structure. The state positions are each embedded into the voxel field to find the voxel cell they occur in. The linearly interpolated velocities of the 8 surrounding cells are applied to the states as a drag force [Fig. 9].

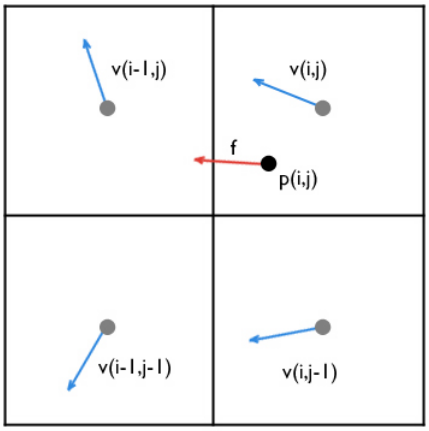

Figure 9: A 2D visualisation of the 4 cells of the voxel field surrounding the state p. The cell indices i and j for the cell containing the state p are first found; the velocities of the 4 nearest cells are then interpolated to find the velocity component f at p. In 3D, the 8 surrounding cells are considered.

The force application techniques could not be nearly so diverse without the centralisation of the L-System state resource via the intervention of the scene. Each force container applies their forces via an overloaded version of the pure abstract ForceBin_3i::mApply(...) method in the base class.

4.4 Positioning, Orienting and Attaching the L-System

In order for the user to position the L-System how they desire inside the scene, additional functionality was added to the L-System itself as well as the scene. The L-System was augmented with a source position point, source normal and source tangent vectors that allow arbitrary root bone orientation. The normal vector defines the direction of a surface that the L-System is assumed to be upon. With no attachment, this is a vertical direction, assuming a flat ground surface. This vector is independent of the actual original root bone direction, whose pitch is relative to this normal vector. The tangential vector can be used to specify the heading of the structure. The scene can set these attributes directly, allowing the user to alter the root bone yaw angle. Alternatively, a mesh can be loaded, any face of which can be selected and act as a source location for the L-System to be attached.

The Mesh_3i class can load a triangulated polygonal mesh from either the Wavefront OBJ (.obj) [Wavefront 1995] or the ASCII Houdini Geometry (.geo) [SideFX 2009] files, much like the LSystem_3i class. The mesh can also be

selected using ray-triangle intersection testing [Ericson 2005] to find any point on a polygon surface. In the same manner as L-System selection parameters can be obtained by the scene, a mesh selection can also be obtained in the form of the three points constructing the selected triangle and the barycentric coordinates of the selection within the face. The scene can also collect the point and normal set of the mesh, enabling the exact selection position and surface normal at this position to be found. A tangent vector is then calculated by normalising a line vector between the selection point to any consistent point in the selection set. These attributes are passed directly to the L-System, which orients its position, normal and tangent direction vectors accordingly.

To allow for a richer set of possibilities, a sequence of meshes can be loaded. These can be scrolled through one at a time, or played through at the scene evaluation rate. For this reason, the attachment process is conducted also at each scene evaluation. As long as the source mesh point set numbering does not vary, a selection will stay consistent throughout playback since the position is parameterised through the selected face point indices and their barycentric coordinates.

When the L-System receives the new sourcing information in the form of the point position, normal and tangential vectors, it must orient the un-deformed bone vectors. This is the purpose of the "orientedDir" vector attribute, which acts as a pre-computed, oriented version of the original bone direction. For this reason, the L-System evaluation process no longer begins by setting the deformed direction of the bones as the original direction vector, but instead the oriented direction vector.

The orientation operation, which occurs in LSystem_3i::mSetRootSource(...), finds the angular difference between the input normal vector and the default up vector {0,1,0}. Finding the angle between two arbitrary vectors is prone to rounding errors should they be tending towards parallelism or anti-parallelism. To avoid these potential problems, and abolish any restrictions that could be implied by maintaining and using one static reference vector, a dynamic solution has been implemented. Initially the scalar product of the two vectors is taken. This can be used to de-mystify the relationship between them, and classify the operations to follow based on the four different approximate cases:

1. A scalar product less than -0.7 indicates that the vectors are tending towards anti-parallelism. Find a new vector in the plane defined by the origin and the inverted up vector {0,-1,0} as a normal. The angle can safely be calculated using the identity [Fig. 10] with the input normal vector and new plane vector, and adding 270 degrees. The axis of rotation can also now be safely found using the cross product of these two vectors.

2. A scalar product between -0.7 and 0.0 indicates the vectors are tending towards being perpendicular, where their cross product will generate a vector in a left handed coordinate system. Use the inverted up vector to find rotation angle with [Fig. 10] and adding 180 degrees. Find the rotation axis with the cross product of the inverted up vector and the input normal vector.

$a = \cos^{-1}(V_0 \odot V_1)$

Figure 10: The angle a between two arbitrary normalised vectors V_0 and V_1.

3. A scalar product greater than 0.7 indicates the vectors are tending towards parallelism. Find a new vector in the plane defined by the origin and the default up vector as the normal. The angle can safely be calculated using [Fig. 10] with the input normal vector and the new plane vector, and adding 90 degrees. The axis of rotation can now be safely found using the cross product of these two vectors

4. A scalar product between 0.7 and 0.0 indicates the vectors are tending towards being perpendicular. The angle can be safely calculated using [Fig. 10] with the input normal vector and the default up vector, and the axis of rotation can be found using the cross product of these two vectors.

A similar process occurs when mapping the input tangential vector onto the default tangential vector {0,0,1}, except the problem is flattened into the plane defined by the origin and the input normal vector. Both of the calculations generate a quaternion rotation that can be combined and used to rotate every original bone direction vector to find their new oriented direction vector.

4.5 Modelling the L-System

Unlike a traditional modelling environment where point or line elements could be transformed individually or in a group, this mechanism simply allows the currently deformed shape of the tree to become its new un-deformed rest pose. This allows the tree bones to be shaped by any of the same forces that would be used during animation. Unlike a traditional single or group transformation of point/line data, alterations to any part of the structure propagate naturally along the bone hierarchy that they belong.

After the new shape has been created, a user can set it to be the rest shape. This calls part of the original L-System setup procedure LSystem_3i::mSetupBoneAttrib(...), which will set the original bone direction vector as the current direction vector between the constructing states, and the rest length as the current distance between them. This operation is identical to what occurs after the state set has been loaded from the geometry file, after the hierarchy has been established. At this point, the scene removes all existing forces so that they do not have a double-transforming effect on the newly shaped structure. The simplicity of this freezing operation is due to the simplicity of the L-System data, merely point and line information.

4.6 Capturing and Using L-System Animation

In order for the resulting motion of the L-System to be useful in other environments other than just this demonstration program, the class itself can write

the current structure to a geometry file in the same format that it was loaded. The data is written in the form of point positions and line segments, like that of the input data. This capturing process can occur whilst the user is interacting with the dynamic L-System without affecting performance too noticeably. If the ASCII Houdini Geometry file (.geo) [SideFX 2009] is used, additional point attributes are written, such as velocity, mass, width and node up-righting vectors.

These geometry files can then subsequently be imported into any 3D software package that can accept line segment meshes in the format used. For demonstration purposes, two small Houdini [SideFX 2009] digital assets were created which can usefully interpret the data passed in an ASCII .geo file:

1. The "nhLSGenRoughSurface" asset uses built in nodes and some additional Houdini expressions to form a polygonal surface in place of the line segment mesh, using the width attribute to define the radius of the surface around each segment. This can subsequently be scaled to alter the overall thickness of the structure.

2. The "nhLSGenLeafSurface" asset uses built in nodes to copy user defined leaf geometry onto the leaf nodes of the structure, allowing varied scale and orientation to be added to each.

4.7 Architectural Overview

To allow easy interaction between all elements of the scene, the lower classes were designed to allow another object to control them. A clear hierarchy was established, keeping classes at good access levels in relation to the information they needed to operate on. The scene containing the forces and L-System controls dynamic connections, not the objects themselves. A diagrammatic overview is shown in Figure 11.

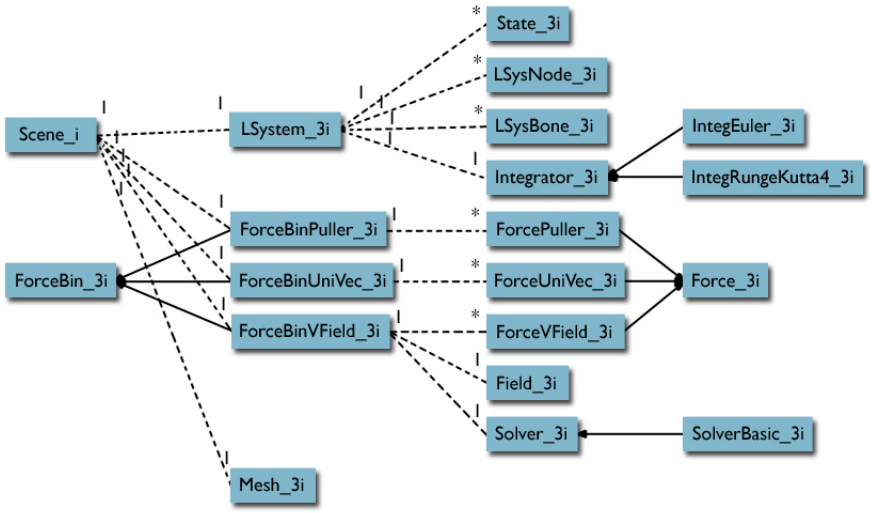

Figure 11: L-System state data can be obtained by the scene and connected to the mesh or forces. The force bins are responsible for all individual forces.

5 Results

5.1 Visual Result

The visual effectiveness of the system can be evaluated from a number of different perspectives, since animation has a number of styles. Cartoon-style animation seeks to amplify movements by stretching and exaggerating proportions. This is done by manually altering the scale and length of, for example, character limbs. Animation aimed at capturing reality seeks to avoid any unnatural proportional distortion.

By comparing photographs with output from the simulation, some comparisons can be made between the effectiveness of the pulling forces [Fig. 12], the uniform vector force [Fig. 13], and the velocity field force [Fig. 14]. Unfortunately, the comparable photographs show only small differences in shape of the captured trees.

5.2 User Interaction

The ray selection method proves to be effective for allowing precise position and selection of items. The addition of a Qt interface displaying program controls is also helpful. Functional controls are split into different tabs, such as L-System parameter options, force creation, mesh loading and playback, and animation capturing. The navigation of the scene is familiar to any 3D software package user, allowing tumbling, tracking and dollying of the camera. The manipulation of the forces is easy and transparent, although the lack of precise control over position and rotation of vector based forces can become inconvenient since they are moved only in the plane of the camera for simplicity.

5.3 Dynamic Simulation

Some aspects of the simulation are effective while others are unstable and erroneous. These are briefly discussed and solutions proposed.

5.3.1 Simulation Accuracy

Briefly re-examining the L-System internal force application processes in section 4.2.1 and 4.2.2, it is noticeable that there is in-fact a duplication of certain force components applied. Namely, the forces applied along the bone direction vector to prevent stretching or compression [4.2.1] are in the same direction as a component of the force applied to goal the bone end states back to their ideal position [4.2.2]. Since the goaling forces of section 4.2.2 are capable of reforming the shape of the tree by themselves, it would seem appropriate to drop the addition of those force in section 4.2.1 altogether. As we have seen, the goaling forces are unconditionally stable, no matter the time-step size. By adding the stretch forces of section 4.2.1, the system is in-fact made less stable by the unstable explicit integration step which is taken. Unfortunately, by removing these pure stretching forces, certain attractive behaviours are lost

Figure 12: Pulling forces are used to mimic deformation on a tall thin tree which is moved by the wind. Because the pulling forces can be positioned at very precise positions along the branch, and moved to any position around the scene, tactile and intuitive reshaping of the structure is easy to achieve. By moving a bone lower in the hierarchy, all child bones are affected by change in direction. The effect can be created using as many puller forces as desired, adding and removing wherever necessary. If the structure contains a very deep hierarchy, i.e. there are many bones sequentially connected constructing one branch route, such the L-System depicted, the tree can become more difficult to control. Small alterations to pulling forces attached nearer the root bone affect the behaviour of the bones towards the leaf nodes more erratically. Reducing the bone stretch resistance parameter can lessen this.

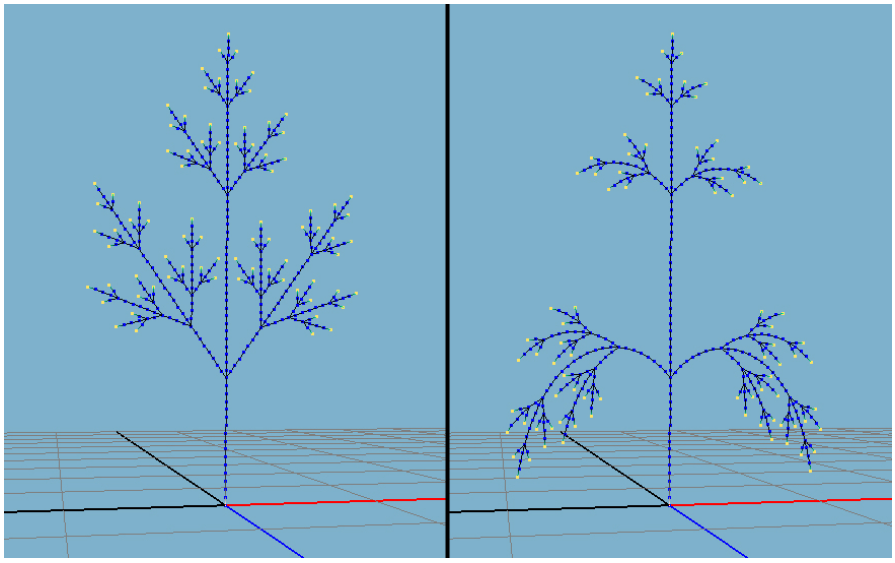

Figure 13: A uniform vector force is applied to a previously static L-System (left) to imitate the effect of gravity (right). It is clear how the bones each bend slightly in the direction of the downward force, resisting only part of the force applied to them. Each successive generation, the sagging of the branches is propagated down the hierarchy. For simple uses, the uniform vector force is very effective, such as the gravity in this situation. For other uses such as imitating a sideways wind force, the result can be very regular and not at all convincing.

Figure 14: A velocity field force is applied to the higher bones of a voluminous tree. The movement of the leaf nodes and the bones is interesting and varied, as well as localised to the injection stream of the force manipulator. The velocity voxel field is quite coarse to enable interactive use, which results in less interesting wind vortices to be created. This can adversely affect the detail of the forces applied to the different states. The system is less concerned with this aspect however, since should the L-System deformation technique be implemented in an existing animation package with fluid support [Autodesk 2009, SideFX 2009], then these built-in fluid fields could be used to apply the force.

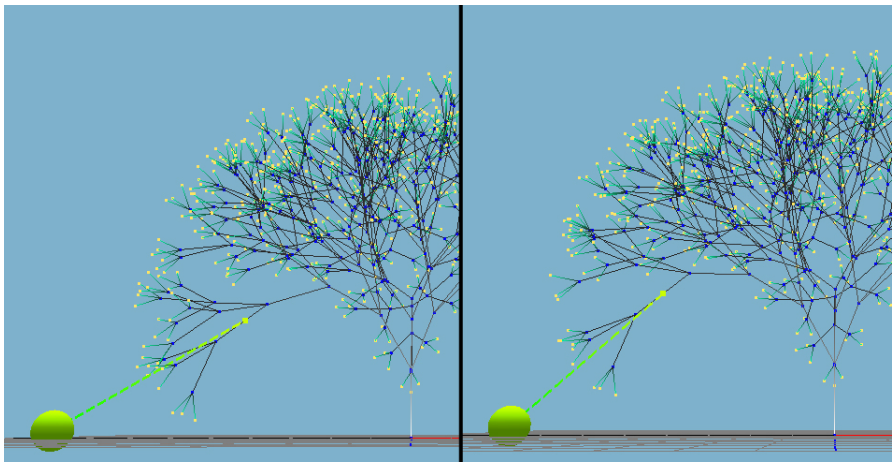

Figure 15: On the left, the stretch forces are being applied. On the right, there are no stretch resisting forces. In both cases, a pulling force of the same magnitude is applied to an identical L-System. On the left, the pulling force attracts not only the bone to which it is directly attached, but also the ancestors and children of this bone, in a natural manner. On the right, the pulling force only affects the children of the bone and the bone itself. This leads to extremely unnatural behaviour where the plant seems locked at all states other than those children of the pulled bone.

from the simulation [Fig. 15]. For this reason, the force was retained, despite the instabilities it introduces.

By keeping the goal force applied, stability which is lost due to the stretch force is regained, although not completely. Hard coded restrictions have been applied to the maximum resistances that can be applied for this reason. This is an unfortunate way to have to resolve the stability problems. Exactly the same behaviour is lost when the stretch resistance parameter is set to 1.0, which will eliminate all stretching of the bones at the resolution step [4.2.3] for the same reasons [Fig. 15].

A subtle disadvantage of using goal-based forces as is seen in the bend resistance is that the branches do not display any springiness/bounciness. This occurs where a state does overstep its ideal location during integration, but in a manner that does not lead to instability. Since using a goaling scheme prevents any overstepping, all springiness is lost. This can be seen as advantageous, allowing the animator more precise control over adding the bounce manually where they like, or as a disadvantage because natural secondary animation is lost.

5.3.2 Orientation Difficulties

Since all forces are generated purely through the use of oriented vectors, there is degeneracy along the axis of the vector where the actual bone yaw is impossible to deduce. The yaw of the root bone is enforced solely by maintaining a fixed offset between a default heading vector {0,0,1} and the current source tangential vector [4.4]. Unlike a skeletal animation system where bones have full orientation information, there is no use for the L-System bones to carry this information because it becomes invalid after one simulation step: No full orientation information can be deduced from the arbitrary vector between the new state positions. This simplifies the simulation, but makes certain effects impossible to achieve. For example, true twisting of a bone, where the tangent of the bone vector differs along it.

5.3.3 Collision Detection

As an aside from the internal L-System simulation, collision detection and response is an important aspect of a dynamic scene. In this situation, collision detection between the L-System and meshes could be trivially performed via line segment-polygon intersection or nearest point testing [Ericson 2005]. Collision response could be implemented in any familiar fashion such as penalty forces rejecting states from any intersections occurring.

5.3.4 Future Considerations

In order to combat the conflict of interests experienced between stability and behaviour [5.3.1], a purely goal based approach is suggested for future consideration and experimentation:

1. Concatenation of forces resulting from bone stretch, all external forces from the environment and a direct goal force to the current deformed configuration of the tree, with maximum step size enforced on integration.

2. Reconfiguration of the tree ideal bone/state goal positions based on true deformed state positions.

This would avoid the unstable goal-free step, whilst allowing the states to move freely as though connected in a loose spring system [Breen et al. 1994] and maintaining the L-System shape.

A foundation for introducing a bone-twisting scheme could involve using tubes instead of line segments for the bones. The tubes could be simulated as free rigid bodies which are connected through joints at the current state positions. True orientation information can be obtained from a properly simulated rigid body, indicating that twist may be derivable by some means.

5.4 Surface Generation

The digital assets created in Houdini [SideFX 2009], which create a rough tree surface to visualise the plant allow for primitive representation of the L-System.

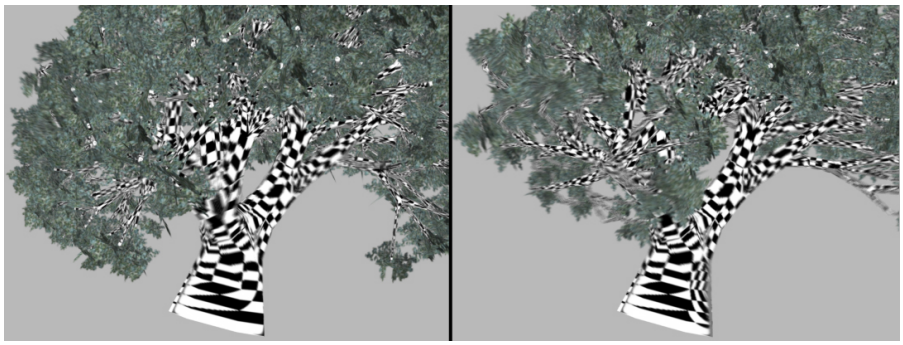

Figure 16: The checker pattern reveals the UV coordinates applied to the model. Along the right side of the main tree trunk, on two successive frames, the UV coordinates have jumped erratically due to the alteration in mesh topology.

They demonstrate a few key points that should be considered however. Currently, a built-in "polywire" node is used to generate a polygonal surface around the wire-frame structure. This operation is responsible for also creating the texture coordinates for each vertex of this polygonal surface. As the wire-frame L-System changes over time, the topology of the geometry that is generated also varies uncontrollably [Fig. 16]. This is a problem likely to plague any surface generation scheme of this type which, as recently discussed, has only degenerative vectors with which to establish an oriented surface from. Instead, the generation of one initial rest mesh is proposed. This mesh will then be deformed based on the deformation occurring on the wire-frame L-System. This would ensure that the topology is consistent. The static mesh could be deformed via some skinning or lattice scheme. This is perhaps more fitting, since a highly detailed surface could be generated once and deformed, instead of at each different wire L-System frame.

6 Conclusion

This project demonstrates a technique for dynamic L-System animation and remodelling in the form of an interactive environment, while the core mechanism behind the L-System dynamics are suitable for any kind of implementation. The mechanism could be implemented in a number of situations such as: 3D animation software, in the form of a plug-in where the forces could be properly keyframed for repeatable animation; Interactive computer games as fully interactive foreground items or more realistic backing elements; L-System generation software where the generated plants can be further manipulated or animated.

The project proves that it can produce acceptable results even when physical evaluation step sizes are large and infrequent, a restriction most evident in interactive programs.

7 Source Code Guide

Following is a breakdown of the source files included in the project, identifying the aspects that were written for this project and the aspects that were used or based on other sources.

C++ header and source code files, all classes originally were written for the Animation Software Development unit and directly re-used with only minor modifications:

- programming/includeOther/Point.hpp
- programming/includeOther/Vector.hpp
- programming/includeOther/Window.hpp
- programming/sourceOther/Point.cpp
- programming/sourceOther/Vector.cpp
- programming/sourceOther/Window.cpp

C++ header and source code files, all classes were originally written for the Animation Software Development unit. The Quaternion_i class is based on quaternion example code [Bourg 2009]. A new Quaternion_i::rotate(...) method has been added which allows a point to be directly rotated by the quaternion, which combines what can also be achieved by combining the operations: "Quaternion_i * Quaternion_i(Point) * Quaternion_i.inverse()". The new method reduces obsolete calculations when performing the rotation:

- programming/includeOther/Quaternion.hpp
- programming/sourceOther/Quaternion.cpp

C++ header and source code files, all classes researched and outlined during the Personal Inquiry unit, and subsequently re-factored. All classes based on example code by Fiedler [Fiedler 2006]:

- programming/include/Integrator.hpp
- programming/include/Integrator.cpp

C++ header and source code files, all classes developed for this project. Aspects of the code have been generated by the Designer-qt4 interface-building scheme [TrollTech 2008]. The original interface designer project file is: programming/Qt/lsAnimator.ui; the deprecated output file from the interface code generation process is: programming/Qt/QtTurboHeader.hpp

- programming/include/Interface.hpp
- programming/source/Interface.cpp

C++ header and source files, all code written for this project, except Scene_i:: mInitialiseScene(...) which was originally written for the Animation Software Design unit. This method was formerly responsible for creating an OpenGL display list used to visualise the scene origin grid shape, but has been subsequently re-factored:

- programming/include/Scene.hpp
- programming/include/Scene.cpp

C++ header and source code files, the Solver_3i class was originally written for the CGI Tools unit. The class structure has been architecturally revised. The Field_3i class has been added, taking attributes and behaviours originally inside Solver_3i on. The Solver_3i::mBoundarySink(...) method was added for use applying a sink boundary condition to the fluid simulation field. The SolverBasic_3i class has been added for a basic fluid solving specialisation of the base Solver_3i abstract solver class:

- programming/include/Solver.hpp
- programming/source/Solver.cpp

C++ header and source code files, all code written for this project. Code in the "mCreateSelection(...)" family of methods found in the ForceBin_3i class and descendants, and the Mesh_3i class is based on the mathematical principles outlined by Comninos [Comninos 2006] and Ericson [Ericson 2005] for efficient ray intersection tests. In the context of these methods, they are presented as line segment to line segment nearest point tests in 3D:

- programming/include/ForceBin.hpp
- programming/include/Mesh.hpp
- programming/source/ForceBin.cpp
- programming/source/Mesh.cpp

C++ header and source code files, all code written for this project. Code in the "mClosestPtLineLine(...)", "mIntersectRayPlane(...)", "mIntersectRaySphere(...)", "mCollidePointPlane(...)", "mCollidePointPlane(...)" family of methods of the Utility_i class based on the examples presented by Comninos [Comninos 2006] and Ericson [Ericson 2005] for ray to primitive intersection tests and distance of point from plane geometric tests in 3D:

- programming/include/ Utility.hpp
- programming/source/ Utility.cpp

C++ header and source code files, all code written for this project:

- programming/include/Camera.hpp

- programming/include/LSystem.hpp
- programming/include/mainGui.hpp
- programming/include/mainNoGui.hpp
- programming/include/Ray.hpp
- programming/source/Camera.cpp
- programming/source/LSystem.cpp
- programming/source/mainGui.cpp
- programming/source/mainNoGui.cpp
- programming/source/Ray.cpp

Shell Script, written for this project to build the lsAnimator and lsAnimatorNoGui versions of the simulation program:

- programming/build.sh

References

[Autodesk 2009] 2009 *Maya.* (2008 Extension 2) [Computer Program]

[Breen et al. 1994] Breen, D.E., House, D.H., Wozny, M.J., 1994. Predicting the Drape of Woven Cloth Using Interacting Particles. *In: Proceedings of the ACM 1994 SIGGRAPH.* July 24-29.

[Bourg 2009] Bourg, D., 2009. *Physics for Game Developers.* O'Reilly Media.

[Comninos 2006] Comninos, P., 2006. *Mathematical and Computer Programming Techniques for Computer Graphics.* London: Springer-Verlag.

[Ericson 2005] Ericson, C., 2005. *Real-Time Collision Detection.* Morgan Kauffman.

[Fiedler 2006] Fiedler, G., 2006. *Integration Basics.* Available from: http://gafferongames.com/game-physics.integration-basics/ [Accessed 20.08.2009]

[Lindenmayer 1968] Lindenmayer, A., 1968. Mathematical models for cellular interactions in development I. Filaments and one-sided inputs. *Journal of Theoretical Biology.* 18 p.280-289.

[Muller et al. 2002] Muller, M., Dorsey, J., McMillan, L., Jagnow, R., Cutelr, B., 2002. Stable Real-Time Deformations. *In: Proceedings of the 2002 ACM SIGGRAPH / Eurographics Symposium on Computer Animation.* p.49-54.

[Muller et al. 2005] Muller, M., Heidelberger, B., Teschner, M., Gross, M., 2005. Meshless Deformations based on Shape Matching. *In: ACM Transactions on Computer Graphics, ACM SIGGRAPH 2005.*

[Noser et al. 1992] Noser, H., Thalmann, D., Turner, R., 1992. Animation based on the Interaction of L-Systems with Vector Force Fields. *In: Proceedings of Computer Graphics International 1992.*

[Power et al. 1999] Power, J.L., Brush, A.J.B., Prusinkiewica, P., Salesin, D.H., 1999. Interactive Arrangement of Botanical L-System Models. *In: Proceedings of the 1999 Symposium on Interactive 3D Graphics.* p.175-182 and 234.

[Prusinkiewicz and Lindenmayer 2004] Prusinkiewicz, P., Lindenmayer, A., 2004. *The Algorithmic Beauty of Plants.* Electronic Edition. New York: Springer-Verlag.

[Prusinkiewicz et al. 2001] Prusinkiewicz, P., Muendermann, L., Karwowski, R., Lane, B., 2001. The Use of Positional Information in the Modeling of Plants. *In: Proceedings of the 2001 ACM SIGGRAPH.* August 12-17. Los Angeles, California. p.289-300.

[TrollTech 2008] TrollTech ASA, 2008. Qt Interface Library. [Software Library]. Nokia.

[SideFX 2009] SideFX Software, 2009. Houdini Master. (9.5.379). [Computer Program].

[Stam 2003] Stam, J., 2003. Real-Time Fluid Dynamics for Games. *In: Proceedings of the Game Developers Conference.* March 2003.

[Terzopoulos et al. 1987] Terzopoulos, D., Platt, J., Barr, A., Fleischer, K., 1987. Elastically Deformable Models. *In: Proceedings of the 1987 ACM SIGGRAPH.* p.205-214.

[Teschner et al. 2004] Teschner, M., Heidelberger, B., Muller, M., Gross, M., 2004. A Versatile and Robust Model for Geometrically Complex Deformable Solids. *In: Proceedings of Computer Graphics International (CGI).* June 2004. p.312-319.

[Wavefront 1995] Wavefront, 1995. The OBJ Specification. [File Format Specification] Available from: http://netghost.narod.ru/gff/vendspec/waveobj/obj_spec.txt [Accessed 16.08.2009]