FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Dynamic Scene Graphs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Christian Folie
Matrikelnummer 0225810

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.Prof. Dipl.-Ing. Dr. Werner Purgathofer
Mitwirkung: Dipl.-Ing. Dr. Robert Tobler

Wien, 24.03.2010 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuer)

Declaration

Christian Folie
Linzerstrasse 429/2115
1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift)

Abstract

In this thesis a new scene graph system is introduced as a new concept, the dynamic scene graph. With this method it is possible to generate scene nodes on demand and use them within the application. It can be understood as a translation of the model-view-controller concept used in the desktop application development to computer graphics. The dynamic scene graph consists of two scene graphs, the first one is the semantic scene graph which holds nodes with a direct meaning to the user. Components placed in this scene graph are interpreted by rules and translated into the representation or rendering scene graph. This translation could also be interpreted as a just in time compiler for scene graphs.
As a concrete application of the scene graph procedural generated tree models are used. The models are configured by parameters stored within semantic components. These components are then interpreted by the dynamic scene graph rules to create the rendering scene graph. Interaction methods with the scene graph in form of dynamic traversals are presented.

Kurzfassung

In der Diplomarbeit wird ein neues Szenegraphen Konzept präsentiert, der sogenannte dynamische Szenegraph. Mit diesem Graphen ist es möglich Knoten in dem Szenegraph nach Bedarf zu erstellen, verwalten und auf Änderungen zu reagieren. Die vorgestellte Methode kann mit dem Model-View-Controller Ansatz, welcher bei konventioneller Anwendungsentwicklung verwendet wird, verglichen werden. Der dynamische Szenegraph besteht aus zwei Graphen. Der erste enthält semantische Daten, die in Knoten des Graphen gespeichert werden. Diese Daten haben für den Benutzer der Applikation eine direkte semantische Bedeutung. Diese Komponenten werden von Regeln, welche auf ihnen definiert sind, interpretiert und im Zuge der Interpretation wird der zweite Szenegraph erzeugt. Die Übersetzung von dem semantischen in den Darstellungsgraphen kann man auch als eine Kompilierer für Szenegraph Daten verstehen.
Als konkretes Anwendungsbeispiel für das präsentierte System werden automatisiert generierte Baummodelle verwendet. Diese Modelle werden als Komponenten gespeichert und können über verschiedene Parameter konfiguriert werden. Die Komponenten werden dann von den jeweiligen Regeln interpretiert und die Geometrie dazu wird generiert. Interaktionsmöglichkeiten mit dem System werden in Form von dynamischem durchlaufen des Graphen präsentiert.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Modern computers and graphic cards get faster and more powerful from year to year. These new possibilities are used by current computer graphics to create more astonishing pictures, full of detailed content. But to create these beautiful and feature rich images the models which are displayed in the scene are required. Today the production of these content models is often more cost expensive and time intensive than the development of the application itself.

To simplify the content generation a technique called procedural geometry can be used. This method is able to create geometry based on different parameter values. The generated models can produce a wide variety of different results by influencing the parameters with random values. The task of the artist who is designing the content is to find the best fitting parameters for a scene. Once the parameters are found, the whole model is represented just by this set of values. The representation as a parameter set is another benefit over a conventional model, because it is a very compact way of storing geometric data by using less storage space. Although storage space is getting cheaper every year, graphic applications require much more space due to the increased detail richness. This is especially a problem for applications which get their content models over the network.

The data generated by different procedural geometry methods has to be represented in the rendering system. In rendering applications this data structure is usually called scene graph. Data stored in the scene graph includes geometry information, positions of light within the scene and material properties, but also scene specific elements like the the camera position within the scene.

With conventional created model data the data itself is transparent to the scene graph for the most applications. This means that the scene graph does not have any information

about the data and treats it as a single object. The scene graph must be able to load the geometry data from a storage medium and convert it into a format which can be displayed later. The scene graph should be able to perform different operations on the data, which is difficult if we have only a single object.

1.1 Motivation

As already mentioned in the introduction computers get more powerful every year and to use this power feature rich scenes are required. But detail content is not the only motivation for the generation of geometry. One benefit of procedural geometry is that each time a scene is loaded the content presented in it can look different.
With this it is possible to show the user different scenes each time the application is used. The user can explore different worlds each time and these worlds could not even be created by an artist.

An example for the usage of procedural geometry can be found within the graphic demo scene. This community has the goal to create computer games or graphic demos which require only a small amount of storage space, typically 64 or 96 kilobyte. To create a demo within this small storage space only procedural methods can be used for the content. They use different methods to create images, scene and even audio data. Examples for demos can be found at the homepage `http://demo.org` or for a concrete demo the game .kkrieger by .theprodukkt [.th04]. These examples show how powerful procedural geometry can be if it is used within an application.

To generate these scenes we need an efficient method to produce and store the models in it. Rendering data is typically stored in scene graphs, a data structure containing positional data, geometric and environment information. In the most systems these scene graph is not explicitly modelled to contain dynamic scene data. This kind of dynamic scene data is then made to fit into the scene graph by emulating or hacking the graph in a certain way.. This emulation is either unsatisfying from the scene graph side, because the graph does not have any detail information over the data which is generated, or from the model side the models cannot interact the way they need to with the scene.
To solve this problem the dynamic scene graph is introduced. This scene graph should create a bridge between an ordinary scene graph and a procedural geometry method. It allows generating data based on rules which model the transition from a user controlled node into a presentable structure.

1.2 Outline of the document

In the second chapter existing scene graph systems are presented and their influence in the researched area is discussed. Especially their behaviour and possibilities for creating dynamic data is analysed. Afterwards different methods for creating procedural geometry are presented, with a special focus on tree generation algorithms and models.
In the first part of the third chapter the theoretical basics of the work are examined. Procedural geometry is discussed in full detail and scene graphs with the operations on them are revisited. Then, as a major part of this thesis, the concept of the dynamic scene graph is introduced in the second part of chapter 3. The different elements defining the scene graph are described and the system is compared with two analogies found within computer science.
The forth chapter presents a case study with an implementation of the dynamic scene graph, a system which is able to define and model families of trees. These trees are able to create different looking instances, and two interaction methods with the generated trees are presented as last part of the chapter.
The thesis concludes with the chapter 5 which presents the results of the implementation of the dynamic scene graphs, a discussion of the results and an outlook to possible work in the future.

CHAPTER 2

State of the Art

In this chapter an overview on different technologies and their approaches is given. They all had influence onto the work accomplished in this thesis. The first part introduces different scene graph systems with their varying approaches and the benefits of them. Their dynamic behaviour and the possibilities to use them are compared as a special topic of this work.
The second part of this chapter examines procedural geometry methods. A special field of application of these methods is discussed, the generation of realistic tree models created by different procedural geometry methods.

2.1 Scene Graphs

A scene graph is a data structure that contains the representation of a graphical scene. The structure is based on nodes which create a *directed acyclic graph* (DAG) containing objects to be displayed. A DAG is a graph without any cycles or loops and with directed edges between the nodes. Starting form a root node a DAG forms a clear defined hierarchy. A general DAG allows its nodes to have multiple parents and multiple children. If only one parent is allowed for each scene graph node then we have a special case of a graph, the so called tree, as defined in the "Dictionary of Algorithms and Data Structures" [Bla08].
On the graph several operations can be performed. One operation is the traversal of the graph, a defined way of visiting nodes in the graph. This operation is used for the enumeration of nodes, performing actions on them or displaying the graphic output on the

screen. Another type of operations change the structure of the graph. They move, insert, add or delete nodes in the tree.

The scene graph should create a layer of abstraction from the way objects are displayed to how they are represented. This means the user "should be able to specify what it is and not have to worry about how to draw it" as stated by Strauss [SC92]. The scene graph system hides the technical details, for example the render API used for displaying the scene, from the user.

Scene graphs include a rich description of a scene, with data for the position and orientation of objects in the 3D space, the surface material of objects or the color of them. The scene graph must provide a mechanism both for interaction with the user and interaction with the environment of the scene. These interacting components include a dynamic behaviour of objects, used for animation, for restriction of user interactions or for simulation purposes.

We are going to examine a few systems and discuss how their scene graph is structured. We look how these systems solved and implemented these basic operations on their graphs.

At the end the dynamic behaviour of scene graphs are compared. Under the dynamic behaviour of a scene graph we understand the possibility to create or change the graph structure and the contained data during the runtime of an application. This is not only needed for reacting on user input data, performing simulation in the scene but especially for procedural geometry and complex graph operations.

Open Inventor

The *Open Inventor* project started as *IRIS Inventor* presented by Strauss [Str93]. The design goal of Open Inventor was to provide a higher abstraction layer for the OpenGL rendering API. With Open Inventor it is simple to create and program 3D applications without caring about implementation details. Open Inventor supplies a rich set of nodes which can be used just out of the box and no deeper going knowledge about computer graphics is needed. It is easy to realize a wide band of applications with the supplied nodes. With this goal the ease of use was preferred over other, technical, issues like the rendering performance.

In Open Inventor everything from the scene is added as a node to graph. This includes geometry, transformations or reaction to user inputs. It is possible to reuse a node and all of the sub nodes in other parts of the scene graph. This can be achieved by allowing a node to have multiple parent nodes, which breaks with the definition of a classical tree. Another way to reuse node structures is to create node kits and use them as if they were

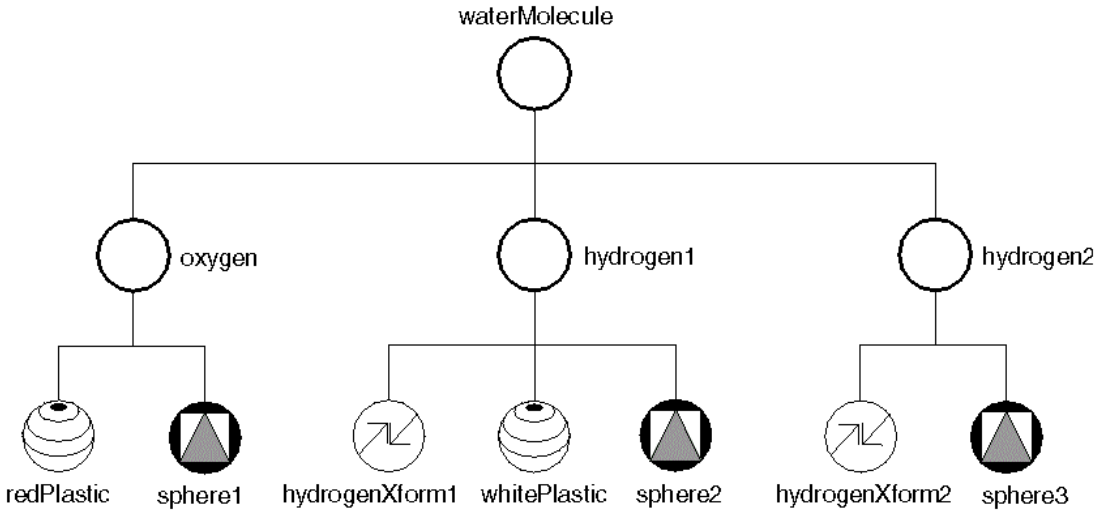

Figure 2.1: Shows a simple example graph in Open Inventor. The graph consists of three sphere objects, where one has a red plastic material and the other two have a white plastic appearance. Because of the evaluation of the graph, which is depth first and then from left to right, the *hydrogen2* sphere does not need an additional *whitePlastic* material. Open Inventor remembers the last material state set by the hydrogen1 node. The last *hydrogen2* sphere needs only a new transformation and inherits the material setting. Graphics taken from The Inventor Mentor [Wer93]

a new, single node. These kits allow the reuse of sub-trees with all properties and can be seen as a replacement for a node hierarchy. Figure 2.1 demonstrates a simple scenario with three objects in Open Inventor.

Open Inventor delivers a set of default nodes which can be used to create a scene. The supplied nodes can be divided into different kinds of nodes. The first group has only influence on the hierarchy of the scene graph and does not add content to the scene. Their purpose is to group, select or structure information and data in the scene graph, which is created by other nodes. Open Inventor defines a so called *separator node* with the purpose of separating different rendering states in the scene graph.

The second group of nodes are used for adding information to the scene. These types are called property and shape nodes. In Open Inventor a scene graph is always evaluated from top to bottom and from left to right. These type of nodes set a state in the scene graph which is kept until another node is changing this specific state, or the evaluation of the graph hits a separator node.

Geometry is rendered during the evaluation of the graph with the state set by the node properties. Custom nodes can be created by extending any existing node and customiz-

ing the behaviour of it during the evaluation.
Open Inventor has no built-in concept for a different, dynamic traversal through the scene graph. It uses a predefined pre-order, or a depth first traversal when any particular action is applied to a hierarchy or a path. A path is a list of nodes which identifies a special node in the graph.

It is possible to place interactive objects into the scene graph and react on them. This concept is called *engine* in Open Inventor. Engines are used for the animation of objects and to constraint a certain part of a scene or a user behaviour. They transform any number of input fields into output fields by applying a rule set defined for them. Engines can affect only other nodes or engines in the scene graph. This restriction removes flexibility from the scene graph because it is not possible to change the environment outside of the graph.
Engines are nodes and therefore part of the scene graph. As every objects of the graph they are only evaluated when the graph is traversed. This has the positive effect that calculations are only performed when they are actually needed, this is called lazy evaluation. The new values are pushed to all connected engines for the next evaluation cycle. One downside of this kind of evaluation is that it slows down the run through of the graph because additional calculations are performed during the traversal.
To capture input from the user action and event nodes have to be placed in the graph, like all other nodes. These nodes get the desired information from the user and provide it to other nodes in the scene graph, via callback functions similar to engine nodes. Nodes for the same event can be created multiple times in the hierarchy, each reacting locally on the raised event.

Open SG

Open SG is a scene graph system with a main focus on performance and not like Open Inventor on programmability and ease of use. Open SG was developed by Reiners [RVB02] as part of the OpenSG Plus project and the development is now continued as an open source project.
Open SG has a big focus on extensibility, both of the framework and changes to the scene graph. Open SG uses different, dynamic traversals of the scene graph as explained by Reiners et al. [Rei02], in order to execute and visit only the nodes needed for a certain action. The scene graph itself has full support for multi threading and clustering of computers and graphic cards.
The scene graph itself consists of a typical tree with each node having exactly one single parent node. These nodes are used only for structuring the information and creating a

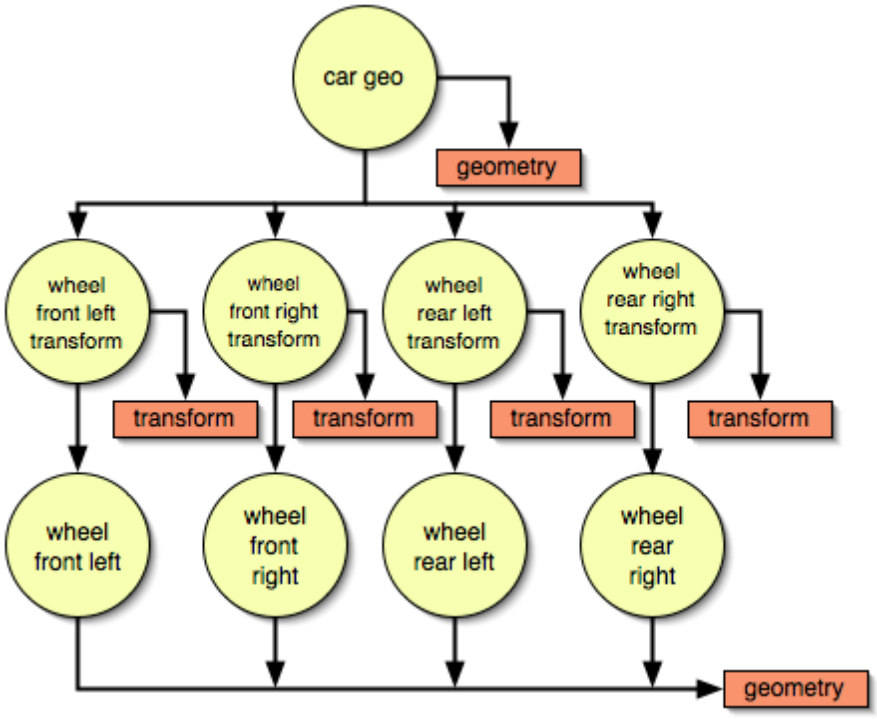

Figure 2.2: The figure shows an Open SG scene graph containing a car geometry. It consists of one root node representing the car and containing its geometry data, four nodes for the transformation of the wheels, with the transformation itself stored as core. The four transformation nodes have one sub node each, pointing to a common geometry core which represents the wheel geometry. Image taken from Neumann [Neu07].

hierarchy in the graph. Each node can have a single "*Core*" assigned, but this core can be shared with other nodes. These core objects can be transformations, geometry, materials or any other data needed in the graph. In Figure 2.2 an example shows the usage of the core-pattern in Open SG.

Open SG provides a very flexible way of traversing the scene graph. Different traversal objects are visiting the graph by moving in a defined behaviour through the graph. Each node can decide if it or its children need to be visited for the current traversal. This creates a flexible way of evaluating a scene graph because each traversal needs to visit only the nodes and sub-nodes where it can be applied in a meaningful way, for the traversal. The pattern behind this traversal approach is called *Visitor pattern* and is mentioned by Gamma [GHJV95].

The obvious example for a traversal type is the rendering traversal or, as it is called in Open SG, the *render action*. On the render traversal object itself different settings, needed

for rendering the scene, are set. These settings include flags for enabling culling, lights or the current camera position. The result of the traversal is influenced by the settings because they can enable or disable features in the scene graph, change the properties of some cores or change the way the graph is traversed.
Another important traversal on the scene graph is the intersection traversal. This traversal contains a ray which is intersected with the scene graph and therefore allows to retrieve the hit points with the hit objects.
A different kind of traverse functions in Open SG are the graph operators. These traversals move through the tree and perform operations on the graph and its cores. They can be used to merge geometry or materials together, apply subdivision on objects, prune unneeded geometry or verify if the geometry is valid to a given criteria. These graph operators are used to optimize the scene graph for a current situation, because these operations are only applied when necessary and can be repeated or revoked as needed.

The concept of separating data from the hierarchy of the graph makes Open SG a very powerful scene graph system. This design enables especially reoccurring operations to be implemented very intuitively as a scene graph traversal, which needs only to be applied in situations where it is explicitly needed.
Another bonus of this architecture is that it is easy to create multi-threaded applications or even application distributed on a cluster of different computers. The basic concept for the multi-threaded architecture is explained in detail in the paper "A multi-thread safe foundation for scene graphs and its extension to clusters" by Voss [VBRR02]. The authors state that data which can be shared, like vector data, should be shared between threads, whereas scalar data like color or transformations are replicated. Synchronization between the threads for rendering, input of haptic devices or simulation is done via a list of changed sub-trees for each thread. During a synchronization the tree needs never to be traversed completely. Cluster based applications can use the same approach as the multi-threaded application.

Open Scene Graph

Open Scene Graph (OSG) is a scene graph system which was inspired by SGI Performer and evolved from it. Its main focus lies on a high rendering performance for 3D applications. Despite its similar name to Open SG these two scene graph systems do not have anything in common. Open Scene Graph was started by Burns and Osfield [BO04] as an open source project and is still maintained and developed by contributers around the world.
The scene graph contains different group nodes which organize the geometry and the

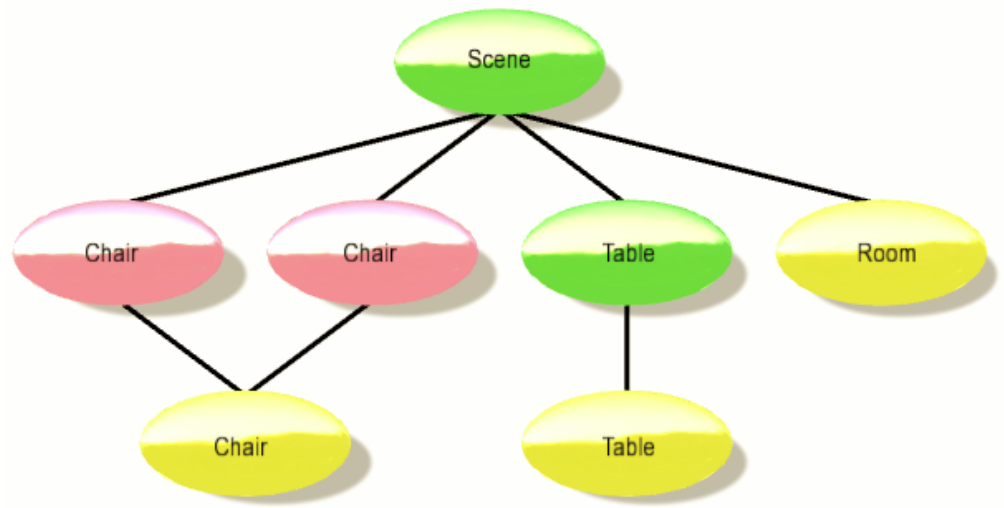

Figure 2.3: Shows a typical Open Scene Graph graph. The yellow nodes represent a Geode node which references a drawable geometry. Red nodes represent transformation nodes, and the green nodes are normal group aggregation nodes. Graphics from the Open Scene Graph Quick Start Guide [KM09a]

rendering state of the graph. In OSG only group and root nodes are allowed to have multiple children, whereas all the other nodes have a single child.
All drawable objects can only be stored at a special leaf node, the *Geode* node. Geode nodes are the only nodes in Open Scene Graph which can be referenced by different parent nodes to allow reuse of data. Beside this feature they can reference multiple drawable objects. The principle of separation from data an hierarchy is only used for geometry and rendering data. Transformations are a direct part of the scene graph. Figure 2.3 shows an example of a basic graph in Open Scene Graph.
Property nodes are applied only to children and their sub-trees below them. Properties are not handled as an explicit state like in Open Inventor, instead they are set and applied directly by the nodes. In addition to the properties, *state sets* can be attached to nodes and drawable objects. These state sets allow a finer control over the rendering state of the application by allowing parents to override the child settings or protect the settings from being changed. These state sets, together with the properties set on a node, affect the OpenGL state machine and therefore the rendering state of the application. But Open Scene Graph provides a better encapsulation of the state manipulation than Open Inventor does. Changes in the properties and state sets are optimized to reduce the changes in the OpenGL state machine and provide a better render performance [KM09a].
Graph traversal is realized with the Visitor pattern, which is implemented similar to the

pattern mentioned on page 7. This pattern is used for rendering and evaluation, but there is no comparable implementation to the graph operators from Open SG.
Dynamic content is realized with callbacks, which are user defined functions. Each node can register functions for different traversal types which are called when a node is visited during a traversal. One problem of the callback concept is that the way it is implemented is limited to a small number of callback functions. In Open Scene Graph only callback functions for update, cull and draw are allowed. To provide additional callbacks new functions for all nodes would have to be created.

SceniX - Scene Management Engine

SceniX [KM09b] is a scene graph system created by the graphic processing company nVidia. It is a multi-purpose scene graph which runs on different platforms and is developed for fast rendering with high image quality. This scene graph supports different output and rendering techniques. It has full support for shaders and in the newest version the scene graph provides an output support for interactive ray tracing, which is also based on shaders. The goal of the scene graph is that it can be used in a broad variety of applications, including CAD, gaming and cluster based applications.

The central design concept of the scene graph is the separation of the data stored in the scene graph, from the operations which are performed on them. This separation provides the flexibility which is needed to support the different applications and techniques mentioned before. All nodes in the scene graph just encapsulated data and they cannot perform any actions on their own. To execute a certain operation a "traverser" is needed, which links between the data and a specific operation.
Similar to Open Scene Graph, nodes in SceniX are also just used for hierarchy, grouping and transformations. Geometry is included into the graph by special leaf nodes, the GeoNodes. A GeoNode consists of geometry, which is in SceniX stored as a combination between a StateSet, which contains state information and attributes, and any number of Drawable objects attached to the StateSet. Each GeoNode can have multiple geometries attached to it, but each Drawable and StateSet must be unique within the node. Beside the GeoNode only one other leaf node exists, the VolumeNode which holds data for one volume and an associated shader for rendering the volume. All other node types in SceniX are group nodes and they can hold more than one node. Transformation nodes are extended group nodes too, which can have several transformations, like scaling, orientation or translation, attached to them. These operations are then applied to all children of them. Figure 2.4 shows an example scene graph with multiple nodes.

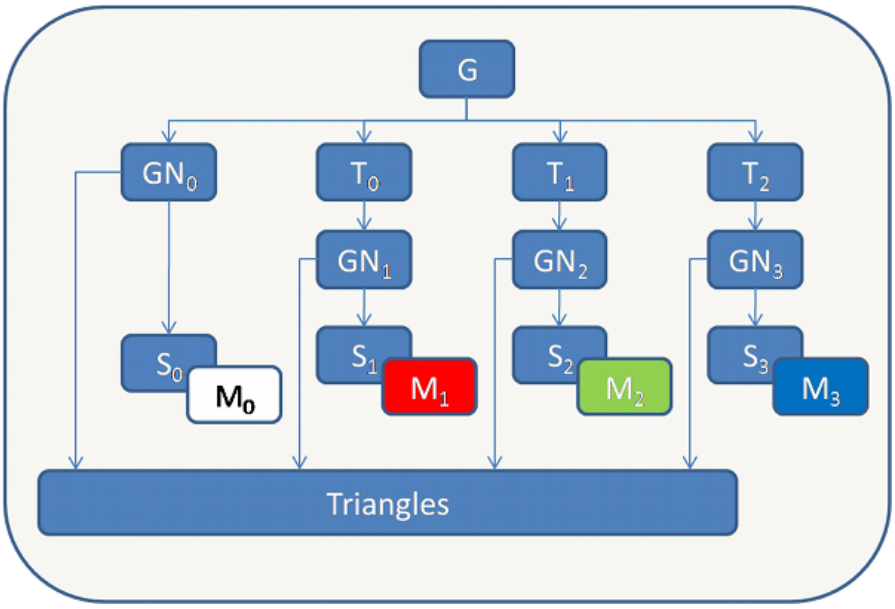

Figure 2.4: A group node G holding different transformation nodes T_i and a single GeoNode GN_0. The GeoNodes GN_i contain a StateSet S_i with materials M_i attached to them and they reference drawable triangles. Image taken from NVSG-SDK documentation [KM09b].

As explained previously the scene graph nodes can not execute any particular actions, but special Traverser objects are needed instead. Traversers iterate through the graph and perform an action on every object in the scene graph. A special feature of the traverser in the SceniX graph is that it is able to obtain a per-object locking mechanism on a node. With this locking it is possible that multiple traversals can visit the scene graph in a thread safe manner.

Starting with this locking mechanism two basic traversal types are defined. The exclusive traversal is used for read-write applications on the graph, where locking of different parts of the graph is required and denying other traversers a writing access to the locked sub graph. Operations which require a write lock are all manipulating either the data or the hierarchy of the scene graph, for example an optimization of the graph or smoothing and stripping of geometry data stored in the graph.

The other type of traversal is the shared traverser which can be used for read only operations, which are easy to parallelize because the different traversers do not interfere. With this concept it is possible to create multiple outputs of a scene graph in parallel because a render or save traversals of a graph do not modify any data. This enables a dual graphic

output which uses a common graphics API like OpenGL or DirectX in parallel to a ray tracing mechanism.

SeniX includes the basic nodes for dynamic data in the graph as all the other systems do, such as animated transformations, level of detail, switch and billboard nodes. It has no specific construct for creating scene graph data on demand or during runtime.

Dynamic Behaviour of the Scene Graphs

All presented scene graph systems support basic dynamic structures like *Level of Detail* (LOD), a *Switch* node or *Billboard Clouds*. These structures allow the selection of different sub trees in the graph depending on certain conditions defined in the node. These conditions are evaluated when the node is visited during a traversal. The node evaluates a criteria which selects the sub trees to be used by the current traversal. These kind of nodes allow a dynamic change in the evaluation structure but they are limited to predefined options specified when the graph is created.
These basic node types do not provide a concept for creating dynamic structures based on several parameters. In the following part, the possibilities of the previously presented scene graph systems are exterminated.

One challenge for dynamic scene graph rendering with Open Inventor is that moving or creating new objects in the scene graph has a big impact on already existing parts of the graph. This is caused by the common state during the evaluation and rendering process. To reduce the influence of new nodes on the state, separator nodes have to be used. They localize the influence of the new nodes which change the state to the new created sub-graph below them.
To realize a dynamic design of the scene graph besides new nodes, a custom action is required. This custom action has to be applied to the scene graph before any other action is executed. The purpose of this action is that, when it runs through the graph and hits a new node, the action creates the dynamic part of the graph as intended by the original node.
In the paper presented by Schmalstieg et al. [RS05] a dynamic extension of Open Inventor, similar to the previous mentioned technique of custom nodes with custom traversals is presented. The paper describes how the Open Inventor scene graph has to be extended to create a context aware dynamical scene graph. This is achieved by creating a new traversal which contains a context object or a context state. In addition the scene graph can be annotated with context nodes. During the traversal the context state of the traversal is combined with the annotated content in the graph. This combined data is now

used to make decisions for the rendering path in the scene graph or to modify properties of already existing nodes.

Open SG has a strong concept on dynamic traversal of the scene graph as explained in section 2.1. This offers great flexibility for evaluating the scene graph. But in the current version Open SG lacks of a concept to create scene graph structure dynamically. Like Open Inventor or Open Scene Graph, new traversals and new nodes would have to be used to create the scene graph on demand. In Open SG this could be done a lot easier than in Open Inventor because of the better separation between the structure of the graph and the data used in the graph. New nodes can be added without influencing the already existing parts. Reusing already existing data can be done by referencing the cores storing the data.
With the graph operators Open SG has a concept especially for manipulating the scene graph. But their main focus is on applying changes to already existing graph nodes and not creating new structures.

In Open Scene Graph changes to the structure of the graph can only be done during the update traversal. There is no concept for dynamic generation of graph structure like in the two other scene graph systems. A similar method for the creation of a custom node and a custom traversal has to be implemented to create dynamic behaviour in Open Scene Graph.

The concept of the separation of data stored in the nodes and the actions applied to the graph as it is implemented in SceniX provides a good extension point for dynamic generation. Because the nodes are just a container for holding data, all the logic has to be implemented in one or more traversers. They have full access to data in the graph because the execution of traversers is a main design feature of SceniX. Therefore, new traversers which generate scene graph data and hierarchies on demand, are easier to implement than in the other, previously introduced frameworks.

2.2 Tree Generation

This section deals with a special procedural geometry topic, the automatic generation of tree models. It is used as an example for different implementation approaches one can choose with procedural geometry to create results.
The first tree models were created directly by the user with the help of modelling tools. This method has the advantage that the user has full control over the result. He can be sure that the designed tree looks the way he wants it to look. But the major drawback of this approach is the huge amount of work somebody has to put into creating a single tree.

The user has to define every stem and every branch by himself. If he needs any changes on his tree a lot of the work would have to be redone. Beside the huge amount of work at the end of the modelling process, only one finished tree model is generated. Because of these issues a hand modelled tree is rarely an option and new, procedural approaches have to be taken into account.
These approaches have in common that they define always a whole family of trees. By a small variation of the parameters different trees of the same underlying type can be created. This allows the user to act as a meta designer by creating only the types of trees and not a single tree. In the following sections different approaches for creating these tree types with meta modelling are presented.

L-Systems

Lindenmayer [Lin68] introduced L-systems as a parallel rewriting system for cellular interaction. The L-systems were extended together with Prusinkiewicz [PL96] for the procedural generation of plant structures. An L-system starts with an initial seed value and a set of rules which define how single elements have to be substituted. All rules are applied during an iteration on the values of the last iteration. This is repeated until a certain stop criteria is reached, in most cases a maximum number of iterations.
Prusinkiewicy defines three basic types of L-systems. The first one is the partial or 0L-system, which is a deterministic system and produces simple results.
In the following several forms of indeterminism are introduced. This is realized either via multiple rules for a single symbol, where a rule is chosen randomly. Or it can be done with context sensitivity, which means that rules can only be applied when pre and post conditions of the rule are fulfilled.
Another type are parametric rules with an attached criteria which is evaluated for the selection of the rule. This criteria can depend on environmental parameters like the position in the 3D space, the branching path or just some random values. A more detailed explanation of L-Systems is given in section 3.1.
The most detailed type in Prusinkiewicz model is the *complete L-system*. This system includes information about the growth rate of a stem, different branching angles or environmental influences like position of the sun or wind sway. Different classes for the branching of stems or for phyllotaxis, which is the distribution of leaves around the flower head, can be used within the model.
The L-System itself describes only the topology of the branching structure. To display a L-system, a *turtle* [Pru86] based approach can be used. The turtle can be understood as a state based function which interprets the symbols of the tree generated by the L-

System as they occur. The turtle "moves" along the symbols created by the L-System and draws it. Two alternative approaches for L-system visualizations are presented in the next sections.

L-Systems are classified as fractals and with their property of self similarity they are suited best to model regular, or nearly regular structures. One problem of a L-system is to find a suitable rule set which creates the desired plant. This is called the inference problem and is one big issue in the application of L-systems. Prusinkiewicz introduced two methods to solve this issue. Edge rewriting is used to substitute figures of polygons whereas node rewriting is used to modify the polygon vertices. Both approaches try to capture the recursive structure of a plant or figure.

Recursive Level Based Method

The method presented in this section is a combination between a scene graph data structure and a tree generation model. The method follows a rule based approach and it can be used for ray tracing and rendering of graphics.

Traxler and Gervautz [GT96] introduced an approach which is able to generate on one side the data and on the other side the representation for trees and other fractal based objects. They use a modified parametric L-system to generate the tree data. The key point of their modification is that their method does not require a reinterpretation of a L-system output as it is needed by other approaches, for example the turtle based approach explained earlier. Instead it is possible to generate the graphical representation directly out of the generation algorithm.

Their method is based on constructive solid geometry (CSG). In CSG graphic objects are created out of different primitives, like a cube, a sphere or a pyramid. These shapes are combined by applying boolean operators like union, difference or intersection on them. Every scene can be described with a combination of these binary operators, primitives and correctly placed brackets. A CSG structure can be used efficiently for rendering with a ray tracing method. In CSG an intersection operation on the whole model is reduced to simple intersection checks with the primitives and boolean combinations of the results.

Hart and deFanti state in their paper [HD91] that the hit calculations for objects which are generated by iterative function systems or other contractive transformations can be reduced to an application of the inverse transformation of this particular derivation step. These transformations can be applied multiple times on the ray. Based on the statement this means that transformations which where once applied to the model, or sub parts of it, are valid for all the following sub parts in this branch. This subdivision can be mapped

to an L-system where we have multiple transformations in form of a rule for a specific symbol.
Kajiya [Kaj83] on the other side was able to reduce the memory consumption during rendering by only generating those parts of the model which were actually needed. This was done by checking the approximated bounding box of the sub part and if the ray would not hit it, this part was never evaluated.
Traxler and Gervautz combined both approaches for parametric L-systems. L-systems are not contractive per definition, but they have always a termination condition. When this condition is fulfilled all remaining symbols can be replaced with a primitive. The geometry of such a system evolves from the derivation sequence, which means that the same operations are applied multiple times on the graph by replacing parts of the model. But the changes of these operations happen always in the sub parts of the model, according to Hart and deFanti. The primitives are only created once and they can be referenced and reused for the hit calculations, which reduces the required memory for the representation.

The expressions generated for an L-system can now be modelled as a CSG expressions by using only three operators. The first one is a transformation node, which performs a geometric transformation on the following expression. A selection node is needed for deciding which rule of a particular rule set is used for the current replacement. This decision is based on parameters configured in the system, which are evaluated and calculated by calculation nodes. These nodes modify, like the geometric nodes, the environment for the following expressions.
With these three simple nodes a graph can be designed which creates on execution a fractal object. Figure 2.5 shows a simple form of a graph which creates a Sierpinsky-Tetrahedron as result.

Mesh based L-systems

Tobler, Maierhofer and Wilkie [TMW02] presented a rule based method to generate meshes from L-systems. Influenced by the work of Traxler and Gervautz [GT96] they created a mesh representation from a parametrized L-system, which can be used for rendering directly. In addition their output mesh can be processed by a generalized subdivision algorithm to create a more complex, natural and smoother geometry model.

Instead of the symbol output which is usually generated by a L-system the result of a derivation step in this method is a template mesh, where each face of the mesh represents a symbol. One advantage of this method is that the topological structure of an object generated by such a mesh based L-system is already encoded in the connectivity of the

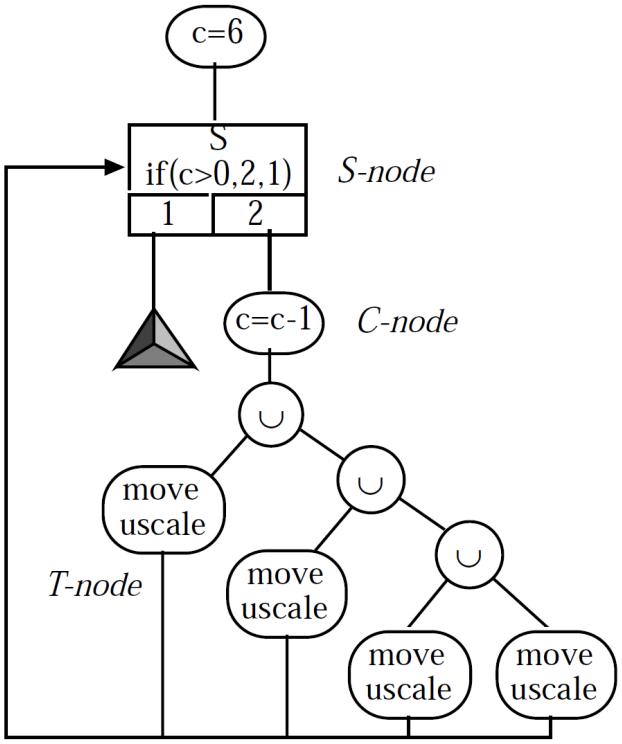

Figure 2.5: A simple object graph which creates a Sierpinsky-Tetrahedron by creating four copies of itself. Before the copies are created a calculation node reduces the parameter c by one. The objects are moved and scaled by transformation nodes and the copies are created by linking back to the initial selection node, now with a reduced parameter c. The initial value of 6 for c defines the recursion depth of the model. Image taken from Traxler and Gervautz [GT96]

mesh. This means that no additional grouping symbols are needed for creating branch structures. The meshes which are generated by this method are already prepared for the optional subdivision following, because this method has the benefit that no T-vertices or other malformed faces are generated if the template meshes of the rules are all well-formed.

This method requires, like all L-systems, an initial seed value which is in this case an initial mesh which acts as start symbol. Then we need a set of production rules which map from one symbol to a set of output symbols. This is translated into a face mapping to a template mesh, where each face of the template represents a symbol.

Parameters and calculations on these parameters can be defined in the L-system and

they can change during a derivation step. Each rule can have conditionals which selects a specific mesh, based on parameter values.

The meshes need to be connected correctly to avoid degenerated geometry, which would produce bad results in subdivision. These connections are done by an auto-attachment operator which tries to find a transformation to connect the previous end face with to the new face. The operator aligns the normals and minimizes the distances between the vertex points of the faces. This operator calculates a transformation which is applied to the new mesh, including a scale, translate and rotate operation.

An extension to ordinary parametric L-systems is the introduction of replacement rules. These replacement rules can produce loops and holes in the model, which are required to create more complex structures such as steel frameworks. As extension multiple symbol replacement is allowed, which means that instead of only one face multiple faces of a mesh can be replaced in a single step. Symbols or faces can also be tagged with a specific value which is calculated by an arbitrary computation rule. Special join replacement rules are then used to connect a set of tagged faces with a template mesh.

Voxel based Model

Green introduced a different approach for tree generation [Gre89]. Trees are created with a *Voxel* based method. A voxel is a volumetric element describing a discrete element in three dimensional space. Voxels create a subdivision of the 3D space into coordinates of elements. The author states that sensing the environment during the growth process is a lot easier with voxels.
Green creates an automaton in the voxel space. This automate applies a set of defined rules which contain geometric constrains, like a intersection avoidance or a proximity constraint. The intersection avoidance rule tries to create a branch which does not intersect any geometry already created by the plant or any existing environmental geometry. Whereas the proximity constraint for a tree is defined that all branches of it must lay within a bias. This bias is defined on an object or on a certain geometry.
Other rules can include the density within a certain region, center of mass of the tree or any other rule based on relationship of voxels.
Figure 2.6 shows the picture of a house on which vine grew. This picture is the result of the combination of different voxel space rules.

Figure 2.6: A vine yard by Green [Gre89], generated by a voxel space automate with proximity and intersection constraints.

Botanical Structures

Reffye [dREF+88] created a structural model which follows the botanical laws of plants. The model includes as much knowledge over a plant type as possible. This includes the age of the plant, different growing conditions or the physics of the branches. The author tries to remodel the growth process of a plant to simulate a individual. This is achieved by simulating the activity of buds at discrete time steps. A bud can turn into a flower or a leaf, which eventually dies and disappears in another time step. A bud can go into a sleeping or break mode for some time steps or it can turn into an internode and spawn new buds. Each plant has a different parameter set for every branch level which controls the evolution of the plant.

The decision which rule is applied for a bud depends on age, growth speed of the branch, the number of buds for the branch and the probability for the three possible

outcomes for a bud. For each branch a development trend can be chosen. These development trends include the direction a branch grows which can be either horizontal or vertical, the direction and shape of the bud itself or magnitude of the gravity influence on the branch.
One benefit of this model is that all growing stages of a plant can be simulated. A plant evolution starts from its seed and ends when the plant dies. With this simulation it is possible to create plants for a specific time of the year within in a scene.

Generic Level Based Model

Weber and Penn [WP95] introduced a parameter based model. In their model the user defines a set of parameters which shape the tree with its branches and sub branches and - as the last stage of modelling - the leaves. These parameters include a branching angle which defines the average angle a new branch takes from its old stem, a ratio for the size of a child in relation to its parent, a taper ratio for the narrowing of a stem, the exact number of branches per stem or the shape and size of leaves.
All parameters can be defined for different levels of the tree with different values. This independence allows the root stem to have completely distinct properties then the first level branches or the leaf branches. Weber et al. call this feature *branch level control* .
Branch level control allows flexibility because every stem can be designed for itself, and therefore allows the user to incorporate in small steps. The user deactivates all levels below the stem he wants to design, and forms the stem after his desires. When the user is finished he starts modelling the stem of the next level, until the tree finally gets the desired appearance. With this method a reduction of the overall complexity for creating trees is introduced. Figure 2.7 shows some of the parameters the user can change and how they are applied in the model.

Another optimization is the concept of cloning or splitting of stems. Clones are copies of a stem, which means they have the exact same properties as their parent from which they have been cloned. With this feature Weber and Penn are able to simulate dichotomous branching, a branching type where two or more branches tend to split apart into different directions.
The parent - child branch in the Weber & Penn paper simulates a monopodial branch, where one of the branches continues inline with the original stem whereas the other stems grow into a given direction. The model allows a smooth mixture of both branch types in a single tree, unlike other models which define a specific branching type for a tree, for example in Honda [Hon72].
To create a tree with a particular shape, the model provides two methods. The first

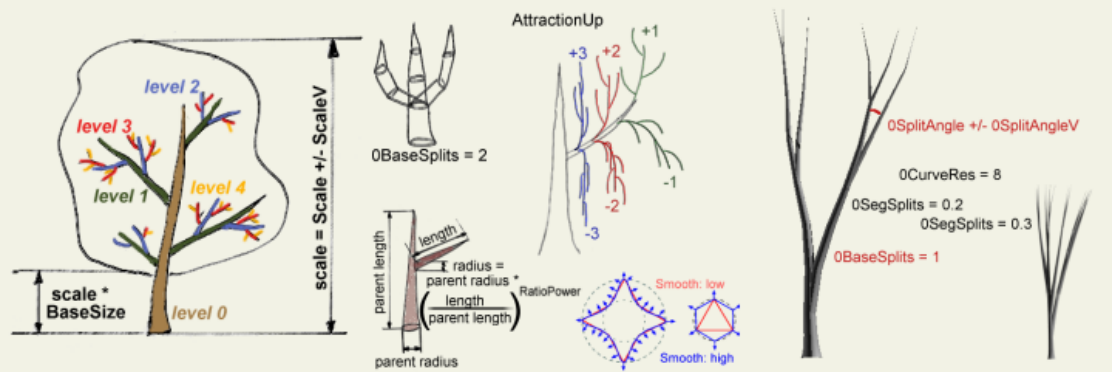

Figure 2.7: Overview of some parameters introduced in the Weber & Penn paper. Image Courtesy of Wolfram Diestel [Die07].

method uses a shape function. This function depends on the position in the current stem and returns a length ratio for new branches or clones of the stem. For example a conical based shape function can be used to create a tree with a wide basis and a narrow top. This allows a loose control over the shape of the tree with no fixed borders for the tree.
The second method to model the shape of a tree is the definition of a pruning envelope. A pruning envelope is a fixed border for the tree, which can not be crossed by any branches. This is achieved by reducing the length of the branches iteratively until the whole tree fits in the pruning envelope. The pruning envelope can be used to create a tree which grows into or along user defined structures, for example the growth of vine along a wall, similar to Greens model. For a pruning envelope any geometric object can be used. An example for trees with different shape or pruning functions can be seen in figure 2.8.
The model includes also environmental influences like wind sway, vertical attraction or leaf orientation. Wind sway is the bending of stems in the wind direction, vertical attraction defines the tendency of a tree to let its branches grow upwards or, like a weeping willow, downwards. Leaf orientation is the orientation of the leaf surface to a given point or direction. This can be used to orientate the leaves towards a bright light source like the sun. For scattered light no specific leaf orientation needs to be specified because leaves should be oriented randomly.
Weber et al. introduce a method for degradation at range to reduce the number of polygons needed to be drawn by grouping items of a type together into "masses". Depending on parameters the degradation replaces polygons with lines for rendering the picture. This creates smooth transitions between the different detail levels because geometry does not disappear from one stem to the other.

Figure 2.8: Overview over different shapes and pruning types of trees. Image Courtesy of B. Lintermann and O.Deussen [LD98]

Structure Graph based

Lintermann and Deussen [LD96] introduced a method for modelling plants based on a structure graph. The user uses a graphical metaphor to combine basic elements together into a graph which creates the desired pant, this graph is called structure graph by the authors. The graph is created via an interactive user interface, which gives the user a real time feedback on the shape and look of the designed tree. This method is a mixture between pure hand modelling and procedural generation. Based on the paper and some publications Lintermann and Deussen [LD98] created the commercial tree generating tool X-Frog. Figure 2.8 shows pruned trees generated with the mentioned program.

The model defines different types of elements which can be placed in the graph. The first type includes components for generating basic geometry data, like a cube, a horn generated by a customizable sweep or a simple leaf.
These elements can be combined with the next type of elements, the iteration and arrangement components. An example for these components is the tree - branch structure which creates a configurable branching structure used specially for trees. A hydra-component can be used to multiply the subsequent components in a uniform circle. As a special type the phi-ball-component is introduced which distributes the underlying elements in a sphere, according to the golden section.
To create global and environmental influences there are special components, which form the third type of elements. Components which allow gravity influence, phototropic arrangement of the leaves and stems or they add pruning constraints to the tree.
All the components have customized parameters and they are able to alter their values

randomly. The components can be applied recursively on the structure graph to generate the plant.

The designed tree is stored as a structure or prototype graph. For creating the geometry this graph has to be expanded to a temporary tree structure. In the expansion step the recursive structures and links to other components have to be resolved. For recursive members and multiple children randomness is applied onto the parameters of a child to vary the created shape. Based on the temporary tree structure the output is created by a traversal through the whole tree. Each component creates its geometry, triggered by the parameter, in a local coordinate system and the results are combined together.
One benefit of this approach is that with fast feedback the user has direct control over the tree he is designing. He can enable features like gravity only on stem or sub branches where it is useful for him and is not restricted to any laws like in the model presented by Reffye in section 2.2. The user can edit the generated geometry afterwards and deform it according to his wishes.

2.3 Summary

We have seen and compared different scene graph approaches used in various applications. These approaches have all an influence onto the design of the dynamic scene graph in this thesis. Then we looked at different tree model generation types and their results. The special focus on this procedural geometry topic was taken because of the major appliance of the dynamic scene graph where it is used to generate trees.

CHAPTER 3

Theory

This section describes the idea behind the dynamic scene graph and how geometry can be generated with it. At first a short overview of the basics is given. We take a look at procedural geometry and within this scope especially on L-systems. Then we have a more detailed view on scene graphs.
At the end the concept of the dynamic scene graph is introduced, the major parts of it are explained and it is demonstrated how they interact to create data.

3.1 Basics

This section describes the two basic concepts used in this thesis and how they influence the dynamic scene graph. The first chapter explains the idea behind procedural geometry and the way it works in full detail. As a special field of application, the L-System is discussed in further detail.
Then the concept of a scene graph is reviewed and some more advanced possibilities of it are introduced, which are used especially by the dynamic scene graph later.

Procedural Geometry

Graphics consist of two main elements - geometry G and the textures T attached to G. The geometry includes the object representation which is stored in the most systems as vertex points p_i and the triangle data t_i, which connects the vertex points with each other. Additional data like texture coordinates is used to attach the texture to the geometry, vertex normals for lightning calculation or the color of the vertex itself can be stored

within the geometry data.
A texture is an image stored in a specific format in a file. In some cases, like MipMaps, multiple versions of the texture are stored within an image. Both, geometry and textures, require a big amount of storage space, even for simpler models.
The main problem, beside the size of the data, is that it has to be defined in the first place. This definition is done in most cases via a modelling tool, where the user designs the geometry and applies the textures to it. This is a good method for complex, but not to detailed geometry or a very uniquely shaped structure. Structures with reoccurring, similar or regular parts on the other side are difficult to model by hand, because the user would have to use a copy, paste and modify method to replicate the elements. This would result in unnatural looking structures. To create a more pleasing model the user would have to define each of the elements from scratch which creates a lot of additional work.
With a procedural geometry method the user needs just to call an algorithm with slightly altered parameters. This algorithm will generate then a new, different looking instance of the geometry.

Procedural Geometry uses a different approach to create the displayable data. It can be written as a function F, which is defined in equation 3.1.

$$F\left(p_0, p_1, ..., p_n\right) \longmapsto \langle G, T \rangle \tag{3.1}$$

F depends on a set of input parameters p_i and generates geometry and texture data as result. This function is evaluated when the geometry and texture data is needed for further tasks. One type of task can be generation of the data and store it to disk for further processing. Another task is to create the data in memory the first time it is actually needed. This can happen for example when the data needs to be rendered. This concept of generating the data on demand is called *lazy evaluation* and is a very popular concept in programming, especially for procedural methods [Har06].
Lazy evaluation has several advantages. One of them is that the data is only created and loaded when it is actually used. This reduces the computing time to a minimum because often data is loaded which is never displayed or used in the scene. But lazy evaluation introduces the problem that generating the data requires some time. This depends on one side on the size of data, on the other side on the complexity of the algorithm which generates it. To make sure the data is available when it is actually needed, pre-fetching strategies have to be implemented. These strategies start the computation processes some time ahead before the data is actually used. This method works best with parallel computing technologies, shifting the workload for the creation process to another thread running in the background.

Another big advantage of generating the data on demand is that it can be created directly in render-aware data structures. Other geometry and texture data is loaded from a certain file format which has to be converted from its storage format into the render data structures. This creates an additional implementation and computation effort, which the direct evaluation eliminates.

Different implementation types of the function F are possible. The simplest form of an implementation of F is to create the geometry data directly in code. For example a sphere, a cube or a torus are simple objects which can be created by a parametrized algorithm. This allows the user to create different primitives by just calling a function with different configuration values and each call creates a new individual primitive. The output of this direct and hard coded implementation is not very complicated and it could also be created by a modelling tool. This direct implementation approach does not scale for bigger models because generating valid geometry data gets complicated very fast. So this method provides no real advantage compared to the plain hand modelling except that it does not use as much storage space.

Another implementation type is the cellular automaton. This automaton has a finite set of states and transitions between these states. With an initial state and initial parameter values p_i the automaton is evaluated as long as it is not in a terminal state. This means that no transitions from the current state to any other state exist and therefore nothing can be executed. An automaton can also be terminated when a criteria is fulfilled, like a maximum polygon count of the created geometry. Parameters set in the algorithm influence the transitions between the state and for this reason change the output geometry.

A cellular automaton can be created either in code which is already a more abstract approach than the direct implementation explained before. The developer creates different state- and transition modules which generate the data and configures them together to create the actual algorithm. A better approach for the cellular automaton is to connect the states together via a modelling tool. This approach is similar to the classical modelling approach where the user designs the object he wants. But instead of defining the model step by step he combines abstract components which form the algorithm in the background. An example of this approach was already presented with Xfrog in section 2.2, where different tree components are connected together and configured in a modelling tool.

Other popular implementations of procedural geometry include particle systems and the generation of terrain data by another fractal method, the random midpoint displacement.

Particle systems consist of many small elements, the particles, which interact with each

other and the environment. This interaction is often based on physical laws but this is not a requirement. Particle systems are usually used in a dynamic visualization, for example a waterfall over a cliff, whereas the random midpoint replacement is a classical pre-calculation operation. A terrain is created by dividing a plane and move the mid point of this plane by a random value up or down. If this is repeated several times this method creates a smooth looking arbitrary terrain.
Another implementation type of the function F is presented in the next section, the L-system.

L-systems

As mentioned in the section 2.2 L-systems are parallel rewriting systems for strings used to create branching structures. A L-system includes a starting word, or often called axiom, ω_0 which contains the initial state. This word is later expanded by a set of different rules. It consists of a list of symbols which are part of the alphabet V, $\omega \in V$. A set of production rules P is applied on the word in each derivation step. A rule $a \rightarrow b$ describes the transition from one symbol a to another symbol b in a single derivation step. A rule can produce any number of different symbols. In a single step all possible rules are applied on the current word. A L-system is deterministic if only one outgoing transition rule for one symbol exists. In the normal case a L-system does not terminate by itself, only if no symbol has any production rule which are applicable to it and therefore no changes are made in the word. The replacement is stopped in the most cases after a defined number of iterations.
Equation 3.2 shows a simple L-system with the axiom, three production rules on the left side and the produced output sequence of the first four iterations on the right.

$$\begin{aligned} &\omega_0 : a && a \\ &p_1 : a \rightarrow ab && ab \\ &p_2 : b \rightarrow ac && abac \\ &p_3 : c \rightarrow d && abacabd \\ & && \dots \end{aligned} \tag{3.2}$$

If multiple production rules for a single symbol exist the system has to decide which rule will be chosen for the current interpretation. One way of deciding can be done by using a stochastic method, choosing the rule by a random value. This method creates different instances each time the L-system is evaluated but of course at the cost of re-productivity. A better way to decide between multiple rules is the usage of paramet-

$$\omega_0 : \quad F \qquad (3.3)$$
$$F \rightarrow \quad F[+F]F[-F]F$$

Figure 3.1: Simple L-system which generates branching structures with the starting symbol w_0 and the only production rule (3.3) on the left and its output after 5 iterations.

ric L-systems where parameters influence the selection. These parameters can be the position in the word ω, the position of the turtle in 3D space, or the neighbours of the symbol a. Parametric L-systems are fully reproducible and allow greater flexibility than the stochastic method because modelling them is far more intuitive.

To usual approach to create a graphic representation for an L-system is to use a "turtle", which interprets the generated symbols. When the turtle creates a graphic representation it interprets the symbols of the word as they occur from left to right and translates them into commands for creating geometry. The simplest form of the turtle understands only the instruction F for move forward a step and draw a line, f for move forward without drawing a line, $-$ for turning left, and $+$ for turning right in a fix defined angle. A turtle with the mentioned commands can only be used to create 2D graphics. For a 3D representation a bigger alphabet and more rotation commands are needed.
The presented commands can create only linear and line based L-systems. For the purpose of creating branching structures the turtle alphabet needs to be extended. It must be able to save the current position and orientation of the turtle into a stack. This is usually represented by the symbols [and] for pushing and popping the values on or from the stack. With these two operators it is possible to create a new branch and, after the new one is finished, continue the previous branch with the previous position and orientation. Figure 3.1 shows an example for a L-system with branching structures and the created output after 5 iteration steps.

Node order	Description	Used for
pre-order - depth first	visits the tree node, then the left and then the right sub-tree	connected components, topological sorting
in-order - symmetric	first the left, than the current node and then the right sub-tree	output of a binary search tree
post-order	left, then right sub tree and the current node as last	identifies the outline of a tree
level order - breadth first	visits every node at the same tree level at first and moves then down in the hierarchy	shortest path between two nodes

Table 3.1: Different traversal types on a graph

The method of creating an L-system to produce a string output, which is then interpreted by another program to generate the graphic representation, is an additional effort. A better approach is to create data structures which model the L-system on one side and are able to create the geometric output on the other side. Two ways to realize this approach were already presented in section 2.2 by Traxler et al. and in section 2.2 by Tobler et al. A new possibility to implement this approach, with an dynamic scene graph system, is presented in the section 3.2

Scene Graphs

As explained in the earlier chapter 2.1 a scene graph is a directed acyclic graph, or DAG. This kind of graph consists, like all graph types, of nodes or sometimes they are called vertices. The vertices V are connected by a set of directed, ordered pairs of vertices A. These represent the edges, or arrows, of a graph. Directed means that each edge has a starting point and an end point and they can be navigated through only in this direction.
A general graph can contain loops or cycles, which means that beginning from one node one can return to this node by following the edges in a specific order. But our graph is acyclic which means that once we leave a node v over an edge e we can never return to it.
For a scene graph we allow only one node to act as root node for our graph. A root node is a node which has no incoming edges and at least one outgoing edge. Starting from the root node it is possible to reach every other node in the graph by traversing it. A traversal of a tree can be done in different orders as shown in table 3.1 with different results. The traversal type depends on the intended action which should be executed on the tree.

A scene graph is used in computer graphics for storing all the data needed for creating an output image. These nodes contain geometric data, transformations, material or surface properties. For some operations it is needed that the traversal visits the nodes in a specific order. As example: drawing objects sorted from the front to the back is more efficient than drawing them in a random order. Another efficient method is to render geometry which uses the same texture data sequentially because then the rendering environment does not have to load the textures for multiple times. These requirements can be fulfilled by reorganizing and rebuilding the scene graph or if a special traversal moves through the graph and collects the execution order of these nodes.

Data in the graph

All data in a scene graph is stored in the node with local or model coordinates. This means that geometric data is created always around the coordinate offspring. When the data is now used in the scene graph transformations have to be applied onto it. These transformations move and orientate the model always in respect to its parent node coordinate system. The root node of the scene graph does not have a parent and for that reason no parent coordinate system. Usually a fixed system, which is in most cases a Cartesian coordinate system, is chosen for the root node.
To get the absolute position and orientation of an object in the 3D-space all transformations of the hierarchy above the node have to be accumulated. Then this combined transformation is applied to the local geometry data of the node. The accumulation itself is usually achieved by matrix multiplication of the different transformations. A scene graph must be able to execute this action when the data is either rendered for display, or for example the calculation of a hit point with a ray is needed.

A normal node can have any number of sub nodes. When the node is visited by a traversal it depends on the kind of traversal if the state of the current node has an influence on the nodes below it. Some nodes, like group nodes, are used only as pure aggregation nodes. They contain only sub nodes and add no additional value to the scene graph. Other nodes may only add attributes to already existing nodes, which is a similar approach to the one from Open SG shown in section 2.1.
Attributes can have a very wide range of functionality. One type influences the traversal itself, for example an on-off attribute allowing the traversal to move in the underlying subgraph or a selector attribute which redirects the traversal into one specific sub graph. Another type applies values stored in the attribute to its child node when it is evaluated. Geometric transformations can be implemented this way for example. The third type of attributes change the context of the traversal. This means they add a value to the traversal

which visits the lower nodes and removes this attribute afterwards. An example for this can be the rendering style of geometry, if data is rendered as solid or only as wire frame.

3.2 The Dynamic Scene Graph

The dynamic scene graph is a data structure which on the one side is an ordinary scene graph as presented in the earlier sections 2.1 and 3.1. On the other side it is possible to use the scene graph for creating procedural geometry as presented in the previous section. This means that the system is able to create its scene graph nodes and the hierarchy on demand. The generation process is influenced by parameters and can be compared with the procedural equation in equation 3.1.

The user or the application itself adds only roughly shaped elements to the scene graph, which have a direct meaning. This application near structure is called the *semantic scene graph*, because of this specific content. The components and nodes stored in this scene graph have in common that they have a very high degree of abstraction and represent a specific semantic intent. For example a semantic node could contain the reference to a file on the disk, which contains data to load and display a model. Or the node could store the initial state of an L-system. Section 3.3 explains these components with more detail.
An important fact about these components, or *instances*, is that they do not contain much logic. Not containing much logic means in this context that they do not store procedural algorithms to create geometry or other elements. Instead they encapsulate all the essential data which is needed to execute algorithms later. The mentioned component, which should display a geometric model in the application, does not have any routine to load the data from the file system. It contains only the necessary information which is needed to perform the loading process. This information could be in the current example the location of the file or a reference to a data stream.

These semantic components of the scene graph are interpreted by *rules*. A rule is responsible for creating the intended behaviour of the corresponding semantic component. This is done by executing different actions, according to the parameter values which where configured on the component. The result of this interpretation is always a scene graph element, which can be yet another component. A rule is nothing more than a transformation function from one scene graph element to another graph element or even to a hierarchy of scene graph elements. Therefore a rule is a high level representation of the basic function (3.1) of procedural geometry. But in the most implementations the rule does not create geometry and textures for itself. Instead it combines other components

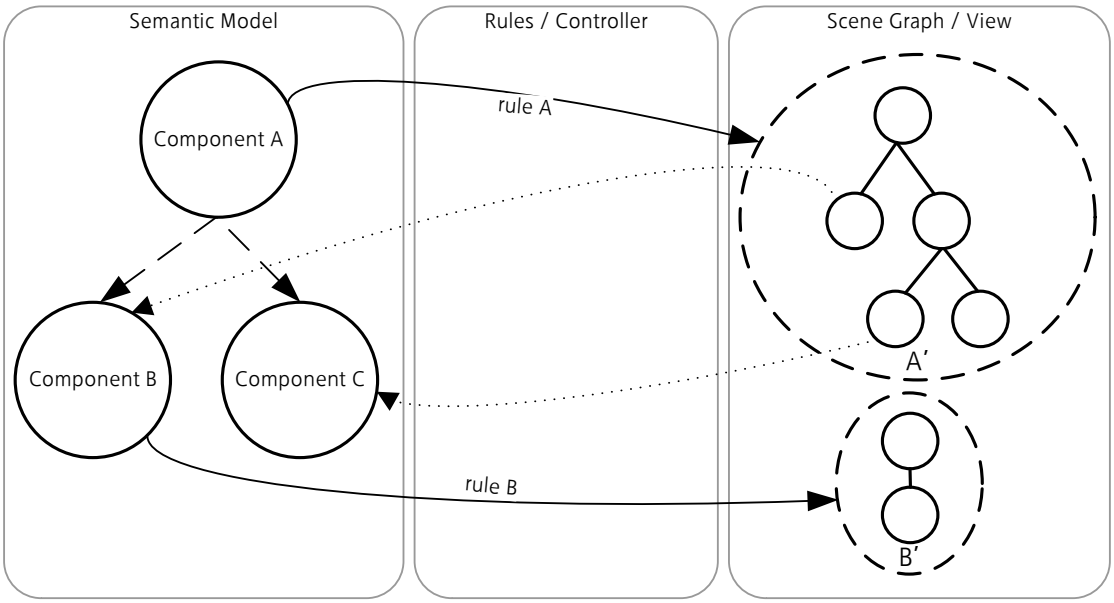

Figure 3.2: Shows an example of a semantic scene graph. At first only component A exists, which is interpreted and expanded by $RuleA$ into a set of scene graph elements. This hierarchy results in two new semantic components B and C produced by the rule. For component B another rule exists which expands it into two new sub components. The rules and their result in dashed lines are stored in a cache, where they are identified by the semantic components.

together which either create the data or even delegate the generation further to yet another component. The introduction of rules modularizes the generation process of data to a set of rules which transform simple components.

At the given point this approach is very similar to a common state machine or cellular automaton. They encapsulate the logic into different states and transitions between these states, which are influenced by the parameters stored in a state. But the difference to the state machine is, that all components or instances of a dynamic scene graph, are part of the same scene graph and therefore changes in parameters have a direct influence on the result, unlike to the state machine. A cellular automaton must be re-evaluated to react on a changed parameter set. It would generates an entire new geometry model for the parameter set. This requires a whole round trip and is not as dynamic and agile as the direct usage.

Figure 3.2 shows an example of a scene graph where the components are expanded by rules. On the left side the semantic view of the scene graph is represented, only the components which might be of interest for the user are shown. The right side contains

the actual scene graph, extended by a lot of additional scene graph nodes generated by rules.

Analysis of the Dynamic Scene Graph

Before the exact details of the scene graph system will be explained we analyse the different parts of the system in an abstract high level view. The elements used in the scene graph are compared with two other well known concepts used in various fields of computer science to give a better demonstration of the idea behind the dynamic scene graph.

Model-View-Controller pattern

The most obvious analogy is the Model-View-Controller (MVC) pattern which is used in general application development. This pattern describes the separation of the data from the way it is displayed to the user and the interaction of the user with it. The pattern was introduced at first by Reenskaug [Ree79] and was modified later by Fowler [Fow02] as the Model-View-Presenter (MVP) pattern. It consists of a model, which represents the data the user wants to work with. The view is used to display this data to the user, in the most cases via a graphical user interface. The controller or presenter is positioned between the model and the view and processes requests coming from the view and the user. It modifies the data in the model according to the request and instruments the view to display the results to the user.

The elements of the MVC/MVP pattern can be mapped to parts of the dynamic scene graph. The model is equivalent to a component or node of the semantic scene graph, it contains the data which should be displayed to the user. The model data is either part of the initial scene or it is created in the application, based on configuration values from other model data. A rule is the representation of the controller or presenter. It translates the model data into the rendering scene graph which is then displayed. The controller in the dynamic scene graph has full control over the view, it creates the representation based on values stored in the model and reacts on modifications of the data. The view is represented by a graph hierarchy which is called the rendering scene graph. It contains either other model data, expanded later by another controller, or it consists of nodes which can be interpreted by a graphics renderer to create a display output.

A big benefit of the MVC pattern is that the same model structure can have different views and controllers attached to it. This means that the same information or parts of it can be visualized differently, depending on the requirements. Applied to the dynamic

scene graph this means that different rules can interpret the semantic scene graph and produce different rendering scene graphs as output.
An example for different rules using the same model would be the generation of a textual or overview representation of the current scene. For the normal, rendering traversal of the graph a rule set which takes the semantic nodes and interprets them by generating a 3D rendering scene graph would be used. If for the same scene a textual description of the scene is needed or if we need a map based overview, a different rule set can be applied to the semantic graph. This rule set will reinterpret the nodes and generate a different output, for example a rendering scene graph which is a 2D representation of the scene.

An other example for an application of the MVC pattern is a web browser. In this application the model is represented by the markup language HTML which is interpreted by a layout or rendering engine of a browser and is presented to the user in a view. The dynamic scene graph can be compared to a web browser as the rules interpret the semantic graph and create a visualization. Different rules can act as different layout engines and display only parts of the model to the user, like the layout engine of a browser can ignore some tags, attributes or plug ins.

This section demonstrated that the dynamic scene graph can be seen as a translation of the model-view-controller concept to the fields of computer graphics and scene graphs particularly.

Just-in-time Compiler

Just in time compilation (JIT) of programs is a technique used for interpreted programs to improve their runtime performance. Interpreted programs are either interpreted line by line or they run in an intermediate byte-code which is executed in a virtual machine. Aycock [Ayc03] presents the history of JIT, according to him the first JIT implementations reach back to 1960 and the programming language LISP. Deutsch and Schiffman [DS84] presented a JIT compiler for Smalltalk-80, which is the first implementation of a JIT in a modern, object oriented language. Nowadays JIT-compilers are used in the most modern languages, for example the HotSpot compiler for the Java Virtual Machine or the JIT-compiler included in the Microsoft .NET framework.
Interpreted programs are independent from a certain operating system or a computer architecture, but as drawback the execution is usually slower than the direct execution of code at machine level. A JIT-compiler is a program which translates the byte-code to machine code during execution of the code. Usually procedures and functions are taken as smallest compilation unit. After a delay for the time needed to compile the code fraction,

this compilation can speed up the execution of the program. The result of the compilation is cached which allows the reuse on every following execution. JIT-compilers can manage their cache by them selves and remove compiled code if it is not needed any more. They can even store it into a permanent storage, which is then an optimized version of a classical ahead of time compiler.
The compiler analyses the executed byte-code and performs optimizations suitable for the current machine architecture on it. In some cases JIT-compiled programs can produce even better and faster results than programs which are compiled directly into machine code, as example a comparison between Java and C++ [LN03].

The dynamic scene graph can be compared with a program which is optimized by a JIT-compiler. The elements of the semantic graph represent the byte-code, which is created automatically or by the user. The rules which interpret the nodes are the equivalent to the JIT-compiler, they take the semantic nodes, interpret them and prepare them for the later rendering. The result of this interpretation is stored in a cache, so that this time consuming process is only executed once and the result is called the rendering scene graph. It contains optimizations and expansions done by the rules, for example an additional subdivision or, in contrary, a simplification of models. The rendering scene graph is then interpreted by a renderer, which is synonymous for the machine which executes the optimized code. Different rules simulate different compilers which perform other optimizations on the semantic scene graph.
The semantic scene graph is stored in a local cache. When an element of the semantic scene graph is needed the rule searches the cache for a compiled version. If an entry is found, this entry is then used for further processing. If no compiled version is available the interpretation is started which needs, like the real JIT-compiler, a certain execution time. The result is then cached and reused on every other iteration eliminating the interpreting time. When properties of the semantic node change the cached element is removed and it is reinterpreted later.

One can see that the dynamic scene graph behaves very similar to an interpreter and JIT-compiler used for programming languages. So we can call the dynamic scene graph system a *compiler for scene graphs*.

3.3 Elements of the Dynamic Scene Graph

After the introduction of the dynamic scene graph and the idea behind the concept in this section the elements of it are explained in full detail and it is shown how they interact with each other. At first the traversal concept is presented and its function is explained.

Traversal	Description	Result
Rendering	Walks through the graph and interprets geometric data	Image which is displayed
Bounding box	Calculates the bounding box of the objects stored in a hierarchy	A box in which encloses all geometry of the sub-graph
Intersection	Moves through the hierarchy and intersects a 3D ray with the geometry	Hit-point and hit-object of the graph
Counting	Count how often a specific type of component occurs	Number of elements
Pruning	Prune a scene graph against another object	Modified geometry

Table 3.2: Examples for different traversals

Then the component and the attribute, the two nodes of the semantic scene graph are explained. As last section the rules which interpret the nodes of the semantic scene graph and translate them into the rendering scene graph are examined.

Traversal

The most important point in the dynamic scene graph is the traversal of the graph. A traversal moves through the graph hierarchy and executes different tasks on the various nodes it visits. Depending on the type of traversal the actions on the nodes are applied in a different order. The traversal itself is represented by an object. It can store information gathered during the run through of the nodes, and any additional environmental data. The most traversals generate a certain result during their application on the graph. Table 3.2 shows some examples of traversal types.

Some traversals require storing values obtained during the traversal on a stack and remove them after the node and its sub nodes are visited. An example for this stack based storage would be the rendering traversal. It uses a stack to keep track of the relative positions and orientations of objects in the scene graph. Objects are always positioned relative to their parent object, therefore these translations have to be saved for each node. After a sub node is rendered the values are restored before the next sub node is processed. Therefore they have the same position and orientation as the other child elements.

For other traversals neither it does matter in which order they are visiting the nodes nor do they need to save any values in a special format. An example for this kind of traversal would be the counting of some nodes in the scene graph. The order in which the nodes

Component type	Additional properties
Selector node	Different scene graph nodes to select from
Rotation node	Axis, angular velocity of the rotation
Level of detail	Different detail models, distance values, decision properties
Model	Geometric data, textures, rendering mode
Billboard	Location of the Billboard, orientation

Table 3.3: A few examples of components with possible properties stored in them.

are counted is not important.Also the counted value has not to be saved in any special format in the traversal itself, instead it can simply be returned.

During the traversal of the graph components are expanded by their rules the first time they are visited by any traversal. This is the implementation of the lazy evaluation paradigm in the dynamic scene graph as presented earlier on page 26. The benefits of the rule expansion are discussed in the next sections.

Components - Nodes of the Semantic Scene Graph

A component contains the essential semantic data for a scene graph element. It only needs a unique identifier for itself which helps later to select an appropriate rule to interpret and expand it. Other properties are only used for interpretation purposes later. Table 3.3 shows some components with possible properties stored in them.

An important fact is that the components do not need to know the rules which will interpret them. This allows great flexibility for both the user to work with the components and for the developer too. He can specify the data and structure in the components and then develop and deploy the rules independently. The relationship between a component and its rule is configured during the runtime of the application and can be changed at any time resulting in an reinterpretation of the element.

When a component is used in the application only the essential parameters need to be published to the user. He is not bothered with other implementation specific details because they are not even known when the components are applied in a program. For example a model component will store geometric information, textures and rendering modes in a user friendly way, for example in a file reference. Whereas the interpreting rule will it transform into a more machine centred data structure which can be used for further processing.

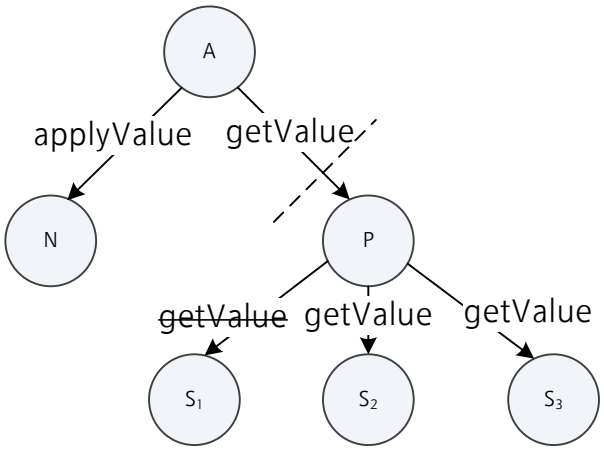

Figure 3.3: This shows an example of an Attribute A which is applied to a node N. The value which is applied to it is obtained from the property P by calling the function *getValue* on a sub-graph. The property delegates the call to underlying nodes S_1 to S_3. S_1 does not implement the given function and returns nothing or a default value. The other two nodes implement it and do return a value which is combined in in the node P and returned to the caller, the actual attribute A which performs then the intended action onto the node N.

Attributes

As already described in section 3.1 attributes can be used to apply custom information to a node. An attribute is always a node which has only a single sub node onto which the defined behaviour is executed. The parameters of the attribute can be stored either directly in the attribute which is the simpler case, or as a more sophisticated method the values can be stored in another graph hierarchy. If they are encapsulated in the hierarchy a way to retrieve the value from it has to be defined. This can be done by starting a special traversal on the hierarchy, which searches it for implemented functions. These functions return values which are applied later on the node. Multiple values from different sub graphs can be combined by some logic in the traversal, and they can be added to a state. This method allows great flexibility because different traversal types can be defined and all nodes can implement a different reaction to the traversal. A major benefit is that this method does not force the application to store values in a specific format. The programmer can decide the way he wants these values to be stored and retrieved. An example for this concept is visualized in graphic 3.3, where the hierarchy is searched for a *getValue* function.

One example of this concept could be a transformation applicator. This applicator is an attribute of any node and performs geometric transformations, for example a rota-

tion, translation of a scaling operation, on it. To get the transformations the sub-graph is searched for any nodes which are able to return this transformation data. The data in the sub-graphs is combined in a typical way for transformations, the multiplication of the single transformation matrices. To apply the value to the sub node the current transformation is pushed to the traversal state, then the node is visited with the modified transformation stack. Because the value is stored on the stack, all elements below this node are then transformed with this modified state. Afterwards, when the traversal leaves the sub node, the applied transformation is removed and the state remains unchanged.
In figure 3.3 and even more complex scenario is visualized. Here we have a custom rotor node as attribute. This rotor node calculates its transformation based on the current time of the scene. Because the time changes for each traversal this creates a different rotation each time. To get the current time value the rotor node interacts with the environment. This rotor node demonstrates one of the many possible ways this concept can be extended.

Rules - Interpreter of Nodes

A rule is a function which provides an interpretation of a specific component of the semantic scene graph, when the component is visited by any traversal. The rule takes the component together with its parameters and applies several actions to it and produces another scene graph element as output. This element is in many case the root node of a small hierarchy of elements created by the logic of the rule. In some cases, for example for a selector node, it returns one of a few predefined scene graph nodes already stored in the component. By dealing with rules two functions are executed by the scene graph system.

Initialization

The first function is used when the component is visited the first time and the interpreting rule is searched and initialized. During this initialization the original component is available and based on the parameters, the data structures are initialized. For the most rules the majority of work is executed during this step. This includes the generation of new scene graph elements and their configuration. The initialization is usually called only once and the created rule instance is saved in a cache where it is uniquely identified by the component. When the component is visited during another traversal of the scene graph this saved instance is used.
The initialization itself will produce a little lag or delay during the first traversal when the

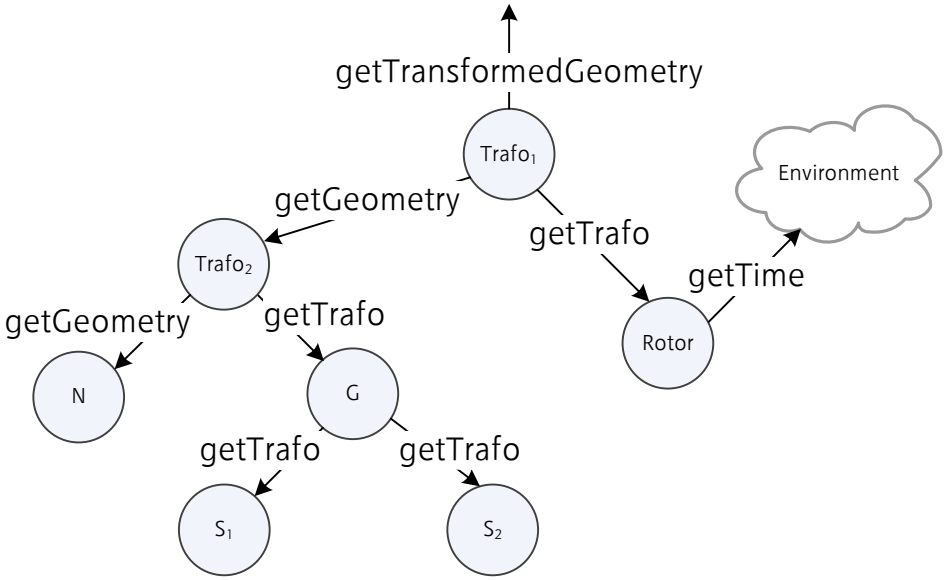

Figure 3.4: The graphic shows a scene graph with two transformation applicators, $Trafo_1$ and $Trafo_2$. Below the first attribute, $Trafo_1$, a rotor component is stored. This component calculates its transformation based on the time in the application, which creates an animation. The node on which the rotation from the rotor node is applied, is yet another transformation node. The transformation for the stored geometry node N is retrieved from a group node G, which combines the values from the two sub nodes S_1 and S_2. The transformations are applied in the trafo nodes, so the $Trafo_2$ node returns the transformed geometry to the $Trafo_1$ node, which rotates it after the rotation calculated by the rotor node.

method is executed. This lag can be avoided for time critical applications with an extra pre-calculation traversal which is applied some time ahead the actual traversal. The pre-calculation traversal does not execute any specific actions on the node but it triggers the initialization as it reaches the component the first time and therefore the rule is cached in the system.

Visitation

The second function is called on each visit of the rule and contains the actual logic of the rule beside its initialization. This method is used to react on different dynamic requirements of the scene graph on each traversal. One example of a dynamic property would be the animation of an object. The rule changes during each traversal some properties, like the position or orientation of the scene graph element it returns. This alteration cre-

ates then an animation when it is executed often enough, as already presented with the rotor applicator in the previous section. These operations for changing the transformation are then implemented in the visit function of the rule.
To perform these tasks in the rule for each visit, the current traversal with all data stored in it can be used which includes the actual transformation or some rendering settings. Additional objects which were created during the initialization of the rule are available too.
This method should be kept as simple as possible, because it must be executed fast. Therefore it is called often and will affect the overall execution speed of the scene graph. This might, for example, have an impact on the rendering performance of the whole system. The goal should be to do as much work as possible in the initialization and reuse and reconfigure the pre-calculated elements during each visit of the node.

3.4 Summary

The cascading of rules is a very modular approach. It allows the semantic components within a scene graph to be developed isolated and the data can be encapsulated within the component and its interpreting rule. It moves the focus from the pure implementation driven, "imperative", approach, as it is implemented by the most existing scene graph systems, to a new, more result orientated, functional approach. This approach is supported especially through the strict separation of the components and their interpreting rules. With enough components in the system the user of the scene graph only needs to choose different components, combine them in a new rule and with these simple parts create a new element.

The traversal of the resulting hierarchy is a major part of the whole system because it is responsible for the complete dynamic behaviour of the scene graph. It allows flexibility in the design of the components because it resolves the connection between a component and its implementing rule during the run through of the graph. As shown, this concept can be seen as a kind of run time interpreter or compiler of scene graph data. To achieve this it must be easy to create and modify existing traversal types and customize them for a special operation. If an existing component should react on a certain traversal it or its implementing rule, it needs to implement a specific interface which defines a function how to handle a visitation during a traversal. In the most applications, the rule itself is not a member of the scene graph, but it always produces at least one scene graph element as output.
To obtain good performance, caching of the interpreted rules is important. This limits

the execution of the expensive interpretation of a component to a single call and allows all further calls just to execute any dynamic behaviour on the rule. If major changes in the components parameter occur and a reinterpretation is necessary, it requires just a removal of the cached instance value and it will react in the next visit of a traversal.

One implementation of this dynamic scene graph approach is discussed in the next chapter, where different trees are generated by components and rules, with a procedural geometry method similar to a L-system.

CHAPTER 4

Dynamic Tree Generation

In this chapter an implementation is presented which is based on the presented dynamic scene graph. It uses the introduced concept of components and rules to create different tree models, which is a special implementation of a procedural geometry method. Two custom traversal types are implemented, one is used to perform a pruning operation on the generated trees which gives them a particular, custom shape. The other traversal is used to create a single, smooth connected geometry mesh from the different parts originally generated by the model.

The chapter gives a guideline how an implementation of the scene graph system could be realized. It demonstrates a way to split up the generation of geometry into different components and rules. At the end an interaction with the scene graph is shown by using custom traversals.

4.1 Basic Model and Parameters

The basic idea of the tree model was taken from the Weber and Penn paper [WP95], which was already explained with more detail in section 2.2. Weber et al. have developed a level based approach to define trees. In their model the user has a fixed set of parameters with which he can configure different detail levels of a tree. With the help of these "parameter sets" families of trees are defined. These tree model families can be evaluated later to shape different individuals.
The Weber and Penn paper was used as a reference during the implementation. The majority of the parameters were used and implemented directly as they were defined in the paper. In addition new parameters were introduced, for example for a finer control over

the distribution of values. Other features mentioned in the Weber and Penn paper were not included into the program due to different requirements on the model. For example the time depended wind sway, a feature where the stems are deformed according to the wind which blows at a certain point in time from a given direction. Or a degradation at range, which is a level of detail feature where the amount of geometry created for a tree depends on the distance to the viewer.

At first the modelling process itself and the influence of the parameters on it is discussed, then a method to alter the parameters for different tree instances is introduced. At the end a way how to store the defined data in a structure is explained.

Parameters

The typical modelling process starts by defining the overall shape of the tree. This is done with multiple steps and methods. The first action is the selection of a basic type for the tree. Then a growing attraction for leaves and stem is defined and the number of recursion levels we want to use is set. Afterwards the root stem of the tree is configured by applying various parameter values. For example a configuration for a flared base of the root stem or additional lobes around it. The curvature of the stem can be defined and the branch less area at the bottom of the trunk can be specified.
Other parameters are possible which can be applied on all levels of the tree. For example the rotation and direction of new branches, which is always defined in relation to the current stem. Or other parameters for narrowing of stems, their splitting probability and the absolute number of child branches.
As the last step the leaves are designed by giving them a shape, size and an orientation toward a certain direction, for example towards the sun.
A list with explanation of some properties can be found in table 4.1 and some parameters are visualized in graphic 2.7 on page 22.

Alteration of Parameters

The model defines a family of trees where every instance of it should look different. It should still be possible to classify a tree instance as an individual of this family. An additional requirement of the model is to create reproducible trees. Reproducibility means that when a scene is loaded multiple times, all trees defined in it should have the exact same shape, each time they are loaded. As result of these two premises a reproducible way to obtain random numbers, which influence the generation algorithm of the tree model, is required.

Part	Property	Description	Examples
Tree	Shape	defines the overall shape of a tree	conical, spherical, flame
	Levels	maximum # of recursion for a tree	usually 2 - 4 branch levels
	Ratio & reduction	ratio between width and length of a stem	its reduction along the stem
	Attraction up	tendency of the branches to grow upwards	0 for no effect, > 0 for up and < 0 for down
	Base splits	splits at the base of a tree	> 0 for trees with multiple trunks
Branch	Curvature	direction into which a stem grows from its offspring	angle between -90 and 90 degree
	Length	length as fraction of its parent	values between 0 and 1
	Segments	segments a stem is divided into	more for better control over the stem
	Splits	number of splits in a segment	any values, fractional values add up along the segments
	Branches	total number of branches along the whole stem	desired number of sub stems
Leaves	Number	count of leaves per parent stem	
	Shape	shape of a single leaf	oval, diamond, triangle, maple leaf
	Bend	magnitude of orientation of a leaf toward a point (sun)	0 for no orientation, 1 for full

Table 4.1: List of properties used in the discussed model

A way to fulfil this request is to initialize the random number generator of a tree with a fixed value calculated from the scene. This initialization value could be a hash code which is calculated from the absolute position where the tree is planted in the scene. To be more precise, this position is the position of the local coordinate offspring in which the tree is actually generated, transformed into world coordinates. This method assures that each tree, with the same parameter set, created at this position will look exactly the same. All random numbers are the same for the trees because the random number generators are initialized with the same seed value. This requires that all random decisions, which where taken to model the tree, depend only on this random number generator.
An extension to this initialization could be that each branch or stem of a tree has its own random number generator, which is initialized with the world coordinates of its offspring from the parent stem. This allows that the random number generator is used only locally within the current rule and its not needed to store the generator within the tree model.

If it is not important to create reproducible trees in the scene, the random number generator can be initialized with any other value, like the current time. This will still create a family of trees with different individuals.

The majority of the parameters are defined as a combination between their mean value and a corresponding variance. This is used to fulfil the family requirement of trees and gives the parameters a certain range. To get a random value, which is then used in an instance of the model, we need to define the parameters with a certain statistical distribution. This distribution could be any distribution which includes a mean and a variance, but in the model only the uniform or the Gaussian distribution was used.

Components

The mentioned properties of the model can be represented with only two components. The first one is the tree component which contains the common data used to model the tree. It includes leaf information, an overall shape or a shape function, and special style information for the trunk.
The other component is used for all the other stem and branch levels including the general shape of the trunk. A single component type is enough because the parameter set on each level is the same and only their values are different.
The majority of the parameters is stored with a distribution in combination of a mean value and variance as described earlier.
With these two components the tree is designed and the modelling process is finished. The branch information is added to the tree component and the tree node itself is placed into the scene graph. When the tree node is traversed, the interpreting rule is searched and executed. The rule will start to expand the tree into its trunk with sub components and sub rules. At the end of the interpretation the geometry is generated by the components and rules as explained in the following section.

4.2 Interpretation of the Model

During the interpretation of a particular tree model many parameters have to be calculated and they are reused in different parts of the model. Some of the parameters are stored in the components and they can be used unmodified in the rules or subcomponents. Other parameters, like random values or variables depending on other parameters have to be calculated regarding to their rules and their result value is stored within the rule for further processing. For the most components these calculations are pre-calculations and they are executed during the initialization of the rule, which was ex-

plained on page 42. The other rules execute these calculations each time they are visited by a traversal.

The following section explains how the implementation is split up into different components and interpreting rules. These components and rules can be divided into two basic types of elements. The first type includes logical components, which are responsible for creating the tree structure. The second type are the representation components, responsible for the generation of geometric output data.
A visualization how all of these components interact together is shown in figure 4.2.

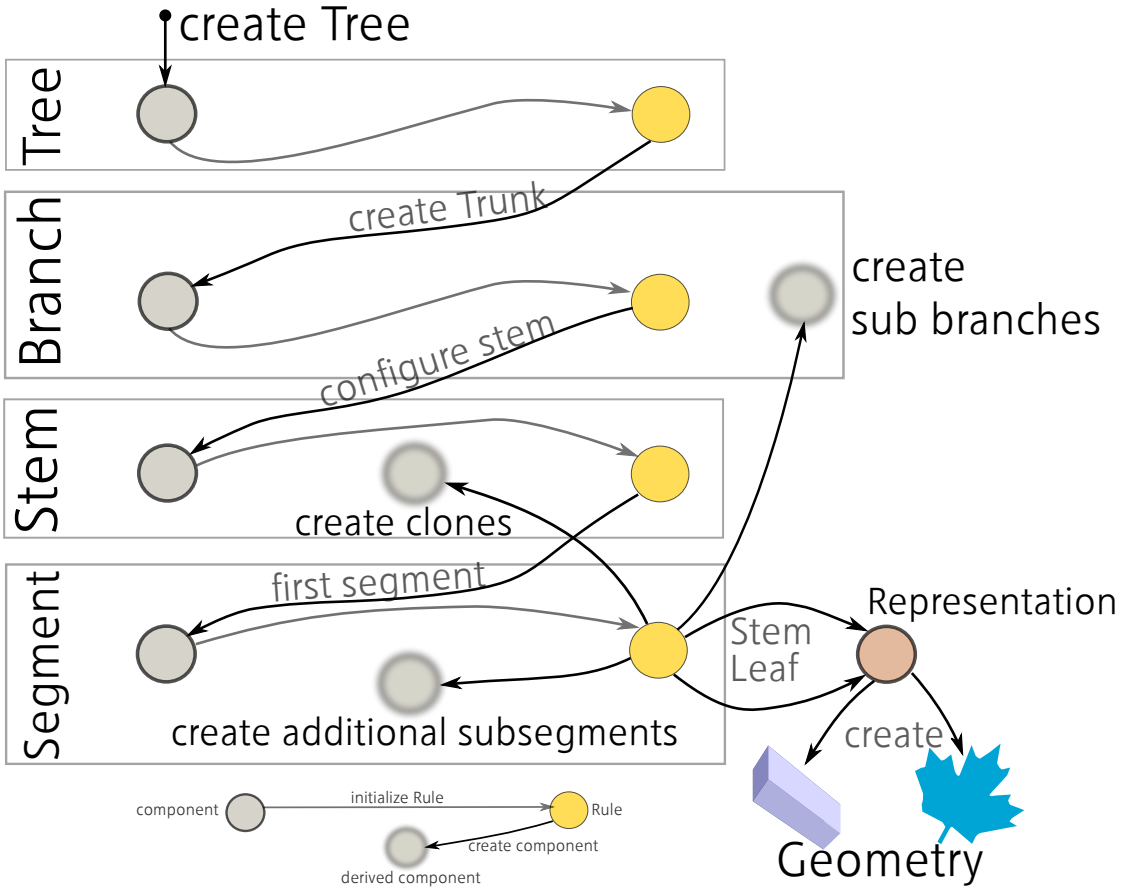

Figure 4.1: Shows the interaction of the different elements of the model. On the left side the components are created along with their interpreting rules. One can see that the segment is a very important part of the model because it creates, depended on already existing elements in the scene graph, the other components of a tree.

Logical Components and Rules

The starting point of the interpretation is the tree component. When this component is visited by any traversal it calculates the general tree properties and returns, as the result of the rule, the component which defines the trunk of the tree. The component representing the trunk and its sub stems is called branch.

Branch

The branch component can be seen as semantic representation of a main branch in a tree. This includes any sub stems which grow out of the branch and the overall shape of them.
During the interpretation of a certain branch the rule calculates the parameters as explained before, whereas it is important that the branch rule has access to the parameters defined in the corresponding tree rule. The calculation of the most parameters which are stored in the branch depends on the recursion level of the branch within the tree. For example the maximum length of a stem is calculated differently for the trunk, the first level branches and all the sequential stems.
The branch rule is responsible to create an even distribution of new sub branches, cloned stems and leaves. This is done by calculating the amount of elements needed for a branch at the initialization. This amount is then distributed along the whole length of a branch.
As the result of the rule a stem component is returned which is the main representation of the branch and its parameters are configured according to the variables calculated in the branch.

Stem

The stem component is used for the semantic encapsulation of one specific stem with all its parameters. This is important because along the stem it is able to create either pure sub stems or clones of itself.
A sub stem is one level below the current stem and modelled via an additional branch and configured on the tree component. A clone on the other side is a new stem which has the exact same parameter values than the stem component it was cloned from, and it is still part of the same branch. These two methods are used to create the branching structure of the tree. The probability to create clones along a stem depends on the parameter defined in the current branch.
Each stem which is created has a specified number of segments, and the stem rule returns the first of these segments as result of it.

Segment

The segment is the last part of the model where an actual logic, responsible for creating tree structures, is executed. But it is also one of the most important components of the model. Because many different actions are executed, which are all influenced by parameters calculated in the stem, branch and tree level above the segment. Elements below the segment are only needed for representation purposes and the do not create any tree structure. The interpreting segment rule produces as output a list of different components.

For each segment it is decided if one or several other clones have to be created. For the decision the branch node is needed because the distribution of clones should be balanced over the length of the whole branch. This can only be guaranteed by the branch itself, because a single segment or stem has no information about the number of already created clones. If one or more clones are needed they are created by copying the current stem in which the segment lies and they are added to the result list for the segment. The direction of the clones is defined by parameters stored at stem and branch level and their position within the segment is altered by a random value.

The direction of each segment is influenced on one side directly by stem parameters which are responsible for curvature. On the other side it is influenced by an attraction factor, which guides the segment towards a specific target. Another impact on the direction of the stem is the number of splits which occur in the current segment. If there are multiple splits in a segment the best visualization is to arrange them in a circle or oval shape. This changes the original orientation of the stem. The previously mentioned wind sway effect could be implemented at this point by adding a time dependent influence to the orientation of the segment. The required time could be obtained from the traversal.

When the segment is visited, new segments are added after the current one if they are needed in the present stem. Sub stems and leaves are generated on demand during the evaluation of the segment, based on the already pre calculated parameters.

For a better graphical representation geometry is not created in the segment but additional sub segments are added and returned as components in addition to the already created sub stems, leaves and clones.

Geometric Components and Rules

To separate the generation of general tree data, which contains the branching structure and algorithmic logic, from the actual procedural generation of geometry special components are defined. These components are used for the graphic representation of stems

and leaves. This separation allows a very efficient implementation of the graphical structures because when they are implemented the components can just focus on the task of generating geometry. All data which is needed to create a mesh is already calculated by a previously executed rule.
Another benefit of this separation is that geometry components can have different implementing rules. For example one rule could just create a very basic and fast skeleton graphic output which can be used for a quick preview of a tree. Another rule can generate geometry in full detail, with textures and other additional data attached to it. Again this is a very modular approach and it tries to push logic, in this case the generation of geometry output, in small, specialized components.

Segment Representation

A sub segment is a simple component which is defined by the length of the segment, a radius for its base and its top and a shape type. The type is used to create different looking segments and sub segments for different shaped trees. As example a palm would use a structure with many repeating bumps in the stem, whereas an ordinary tree might use a stem which is getting more narrow to the top and vanish in a point. Or another type of stems could stay nearly constant in the size and shape and narrow at the end to a spherical tail.
This part of the model is the first one where actual geometry is calculated and a mesh is returned as result of the rule. But the generation of geometry is reduced to a set of quite simple operations. They depend only onto the parameters length, the base and top radius and the shape type of the segment. The geometric mesh is always modelled around the local coordinate offspring and is just scaled by the parameters.
Because of these simple operations, independent from any semantic context, it is easy to optimize the algorithm for the generation of this geometry.

Leaves

Leaves are, like the sub segments, represented by a small set of parameters, namely the width and the length of the leaf, its shape and an orientation. Depending on their shape different geometric forms are generated. The simplest shapes are an oval, triangle or diamond leaf, but more complicated profiles for example a maple leaf are also possible. The generation is again very simple because the leaves are just basic forms which are modelled also around the local coordinate offspring. At the end they are scaled by the

width and length parameters and their orientation is set. This transformed geometry is returned as result of the rule.

4.3 Traversals Types in Model

To show how to interact with the model custom traversal types are defined. This demonstrates the extensibility of the framework to add new functionality on top of an existing implementation by defining traversals. Two types of traversals were implemented, the first one is used to perform pruning of the tree against different shapes and structures. The second is used for creating smooth meshes out of the different meshes generated for each sub segment.

The two traversals are implemented as described in the earlier section 3.3. An interface is defined with a function responsible for executing the desired task on the object. This interface is only implemented by the rules which should react to this specific traversal. A custom traversal object is created which stores additional properties required during the evaluation. This traversal searches the hierarchy for a class implementing the mentioned interface. If such a class is found, the method defined in the interface is executed. This method performs then an action on the component and returns a result. This result is combined in the traversal with results obtained from other hierarchies. The combination is returned at the end to the initiator of the traversal.

Pruning

Pruning is used to restrict the growth of plants to a certain shape or structure. It is, as already explained in section 2.2, a method to design the overall shape of a tree. This method can be used in addition to the other design techniques provided by the model. Pruning should be seen as an operation which is, like in the real world, executed after the growth process of a plant. It reduces and changes the look of a plant to a desired form. The actual structure which defines the boundaries of growth is called the pruning envelope. The pruning process implemented in the tree model is, like cutting hedges in the real world, an incremental task. It is executed on each branch of the tree, regardless of the size of the branch, until the tree reaches the final shape.

As already explained pruning is a local process, which is executed for each branch and stem of the tree. If pruning is needed it is started for these elements when they are evaluated the first time. During the pruning process the branches are interpreted and the pruning traversal moves through each stem of the branch and through each segment

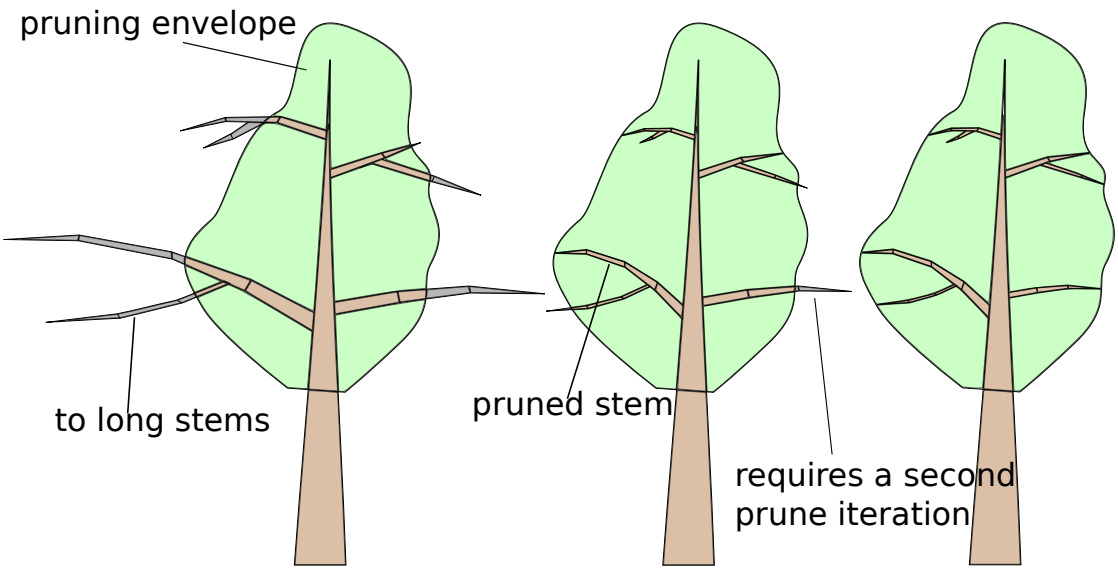

Figure 4.2: Shows a tree with a pruning envelope and miscellaneous stems with their segments. In the first illustration several stems do not fit in the surrounding envelope. Their length is reduced in step two and three until they fit completely into the envelope. Sub stems are reduced by the same size factor as their parent stem. As soon as the parent fits into the pruning envelope, the sub stems size is changed independently from their origin, until the sub stem fits into the desired shape.

of these stems. The traversal checks if any of these segments are outside of the defined pruning envelope. If a segment is found outside of the envelope, the overall length of the current branch is reduced by a certain factor calculated from the branch length and the number of segments lying outside. Then the pruning is executed again, on this now shortened branch. This iteration is continued until either a minimum branch length is reached or the whole branch lies within the envelope. In figure 4.3 a pruning situation is illustrated with two iterations.

It is important that after each pruning iteration all state parameters and the random number generator are restored to their original values they had before the pruning. If this would not be done the new generated branch might look different than the one used for the pruning calculation. Because of the changed parameters and the different random values a different branch is generated and this branch might not fit correctly into the envelope. The only effect of the pruning should be the reduction of the size of a particular branch, but it should preserve its shape and sub stems. After the pruning is executed for the current branch, and it fits into the envelope, the interpretation of the rest of the tree

model is continued, as explained previously by calculating the rest of the parameters and returning a component as result.
The pruning envelope is defined and stored on the traversal object itself and is available all the time during the traversal of the graph. This simplifies the test to check if a stem lies inside of the boundaries. The pruning interface is just implemented by the segment rule, where a simple check is performed if the segment lies still within the boundary specified on the traversal object. All the other components pass the traversal just to their children until a segment is reached. As result the segment of the stem which lies outside the boundary is returned to the branch. The different results need not be combined in any special way, because the pruning is just executed per branch. The caller, which lies always within the branch, interprets the result and performs the actual pruning operation, as described above, by reducing the length of the current branch.

Welding

The second traversal type is used for welding geometry together. Welding means in this context that geometry, created by the model, is modified so that it fits perfectly together and creates a smooth mesh. The welding process is required because each of the segments is just modelled within its own local coordinate system, with no respect to any hierarchy or orientation information. This simplifies the modelling and geometry generation process as explained in the section 4.2, but it creates small cracks and holes in the different meshes because the edges of the transformed meshes do not fit exactly on top of each other.
To solve this problem a welding traversal is created. This traversal moves through the component hierarchy and gathers information about the segments and stems and how they should be glued together to form a single, smooth mesh. The actual welding is performed on the results, retrieved from this traversal, by removing the defects in the meshes and creating smooth transitions between the different geometries. The problem is illustrated in graphic 4.3 and explained in detail below.

The interface of the traversal returns as result a list of the weldable geometry within the hierarchy. This weldable geometry data consists of two parts, a vertex geometry which is already transformed into the world coordinates, and an attached welding information. The welding information uniquely identifies every vertex stored in the geometry of the current stem. This identification is done via the segment, sub segment and the vertex position within the sub segment. Each vertex of the geometry is labelled in this way.
Two vertices which should represent the same vertex have the same semantic identifier,

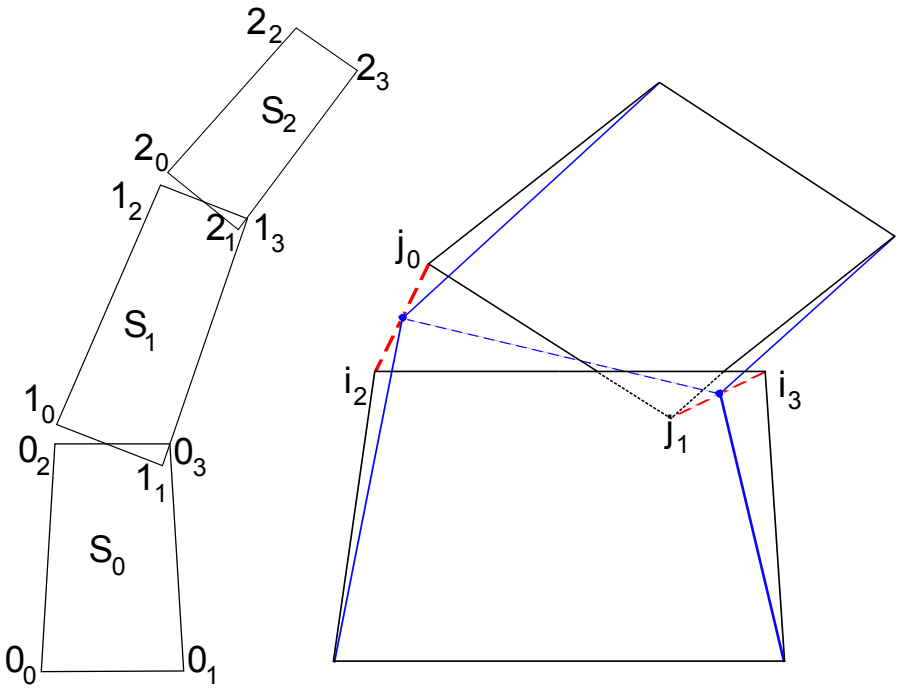

Figure 4.3: On the left side three segments of a stem are visualized with the cracks created by the rotation towards a certain direction. It is shown how the segments S_i are identified by unique numbers i_j, where i stands for the current segment and j identifies the vertex within this segment. One can see from the graphic on the left side that the vertex i_0 maps to $(i-1)_2$ and vertex i_1 maps to $(i-1)_3$ and that these vertices should have the same coordinates in order to create a smooth mesh. On the right side this mapping is visualized in detail, it is shown how the new intersection points are constructed, by averaging the points along the red line and creating new, blue coloured connection lines to continue the geometry.

except that they lie in the leading or trailing segment or sub-segment. Because of this information a new, common vertex position is calculated which represents both of the vertices. This common position can be calculated out of the average vector from the two single points. This point now replaces the two original vertex positions of the segments in the mesh. Because they have now the exact same coordinates, the crack between the vertices disappears and a smooth mesh is created. The geometry of the mesh itself is modified slightly due to the changed vertex positions, but these changes have no influence on the topology or shape of the overall geometry in the mesh. This process has to be applied every time the modelling properties of the tree are changing, because the resulting geometry is different. But the results can be cached and reused later.
The welding traversal is applied on the whole tree and starts its evaluation at the trunk.

The traversal keeps track of the transformations as it moves through the hierarchy and it stores the current stem it is visiting in the state. For each sub stem, clone or branch a new traversal is started. This is needed because otherwise the welding operations of different stems would be mixed and no clear geometry could be created. After the welding traversal is executed on all stems, the gathered vertex geometry with its additional information is processed as described above. As result the modified vertex geometries which represent the new welded geometry are returned. The major steps of the algorithm is visualized in figure 4.3.

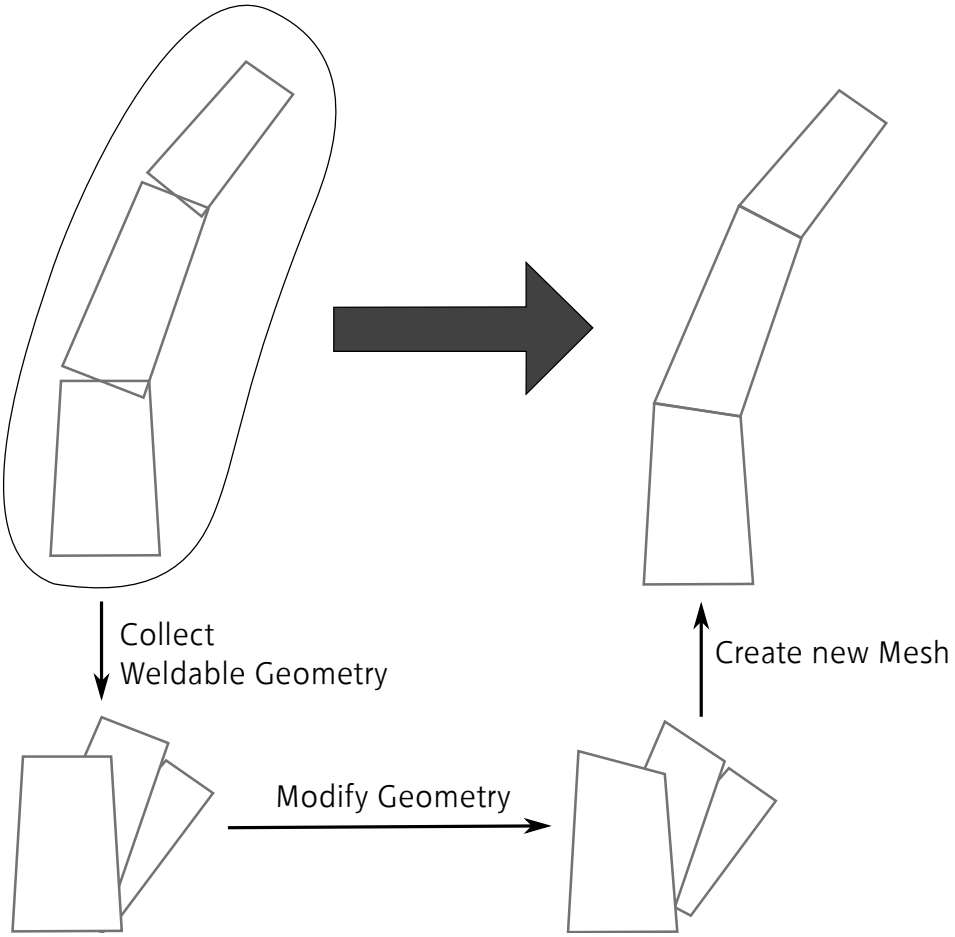

Figure 4.4: The figure shows the three steps of welding geometry.

4.4 Summary

This shows that in procedural geometry the creation of the actual geometry is a small task compared to the setup and calculation of the parameters. The exact results and a conclusion taken from the implementation and the model is presented in the next chapter.

CHAPTER 5

Results, Conclusion and Outlook

In this chapter results from the reference implementation are presented, a conclusion from the results and the implementation is taken. They are compared with other systems and an outlook to future possibilities in extending the model are given.

5.1 Results

In this section some examples with images of generated trees are presented and the influence of different parameter values onto the created models is discussed. Not every parameter which is available in the model is visualized, only a small subset of them are presented here.
The presented models use just colouring and shading for the stems and leafs. Additional textures could be attached but are dismissed for simpler representation of the models.

Overall Tree Shape

As explained in the earlier chapter the model is used to define a family of trees where each tree instance should look differently. This feature is possible because the parameters are defined with a mean and a variance value and they have a distribution which creates a random value for them. The influence of the basic shape of the tree on the family is demonstrated in figure 5.1 and 5.2. All trees within a family are created with the same parameter set, but they were initialized with a different random seed resulting in another shape. The random seed is generated by the position of the tree in the 3D-space.

Figure 5.1: Three black tupelo trees, approximately 370,000 triangles for each tree including the leaves.

The first family visualized in 5.1 represents black tupelo trees and they use a tapered, cylindrical form for their overall shape. This means that the tree is shaped like a round cylinder which narrows to the top. One can see this shape in the figures, where the basis of the tree is wider than its top.

The other tree family in 5.2 are representing quaking aspens, they are shaped like a burning flame. But this shape is not applied continuously along the tree, different sections in the tree can decide how much of the shape factor they want to apply. The tree on the left side applied the shape in the lower parts, the tree on the right side on its top. The tree in the middle used the shape through its whole length. To show the branching of stems the tree is visualized without leaves.

Figure 5.5 show two other trees as example. On the left side a wheat is shown and on the right side a palm is visualized. The palm has a bumb like stem structure as presented below in 5.1.

Branching

The direction of branches descending from a stem is influenced by various parameters. A random rotation and down angle defines the approximate direction for each branch and

Figure 5.2: Three quaking aspen trees with approximately 40,000 triangles without the leaves.

a global growing attraction for the tree influences the orientation of the stems. In figure 5.1, on the left side, three trees with different growing attractions are visualized. The first tree has no special attraction and its branch direction is only defined by the rotation and down angle. Whereas the second tree has a negative, downwards pointing orientation and the third tree a positive, upwards orientation.

Stem Forms

The shape of a stem can be changed by different parameters, as visualized in figure 5.1 on the right side. The trees, from left to right, show a stem where more and more parameters are configured. The first stem has only a certain length and curvature. The next two stems have different types of narrowing. The second stem narrows only at its top, with a spherical ending. The third stem has a special, bump like structure which can be used for a palm. On the forth stem a flare at the base is added to simulate an expanded stem. In addition it has a taper, narrowing to a point, at its top. In the last picture seven

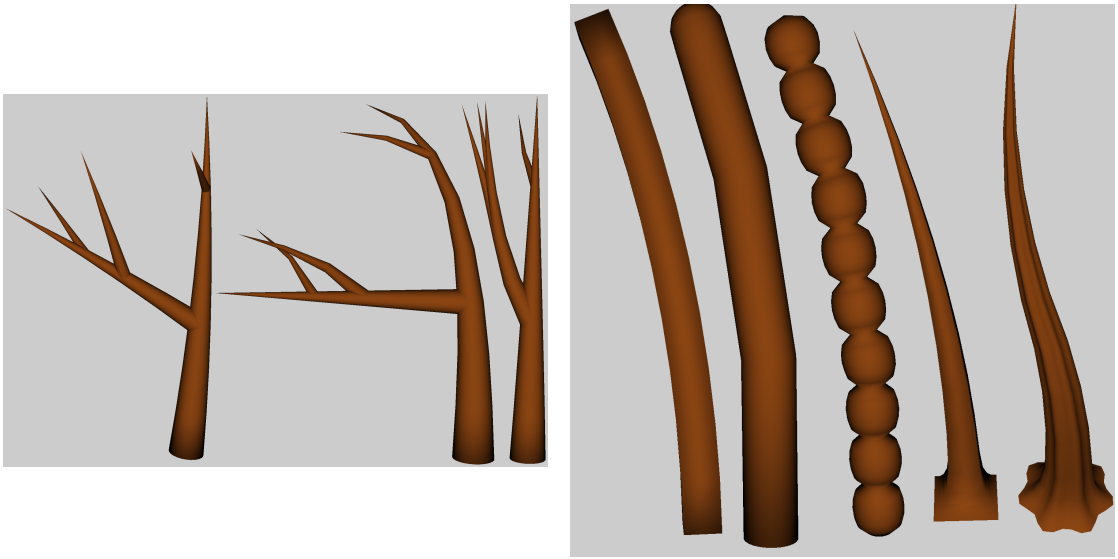

Figure 5.3: On the left side the influence of branching parameters is visualized, the right side shows different stem shapes.

lobes, with a defined depth, are added and the stem has a backwards curvature in the second half of it. With the extra curvature it is possible to create an "S"-shaped stem.

Pruning

Pruning on a tree is an additional effect to model the shape of it, by defining a boundary which limits the growth of the tree. In figure 5.4 two trees are shown with the the pruning traversal applied on them. They can be compared with the trees shown in figure 5.1 and 5.2. On the first tree, on the left, the pruning envelope was narrow on its lower part. The upper part was not changed by pruning. The tree on the right side was pruned more on the upper part than on the lower.

Welding

Welding is needed to create a smooth mesh out of the different parts generated by the algorithm, it was explained in section 4.3. To simplify the geometry generation each segment is treated for its own and creates geometry, independent from other, surrounding segments. This causes small cracks or holes in the mesh as seen in the first two trees in figure 5.6. In addition it creates irregular normals which produces a wrong shading, demonstrated on the flares of the middle tree. To correct this a welding traversal was

Figure 5.4: A pruned version of a quaking aspen and black tupelo tree.

Figure 5.5: A wheat on the left and a palm on the right.

implemented which glues the different meshes together to a single, smooth connected mesh, as seen in the tree on the right side.

5.2 Implementation

The model was implemented with the .NET Framework, in the language C#. The implementation is part of the Aardvark rendering engine which was designed at the VrVis. It was realized via the components and rules described in the implementation chapter 4.

In table 5.2 an overview on the number of triangles and leaves which are generated for a tree instance of a specific family is given. The time the execution needs on a typical computer is shown in the last column.

Figure 5.6: Visualization of a non connected mesh and the effects of a welding traversal.

Tree family	# of triangles	# of leaves	execution (seconds)
Black Tupelo	370,000	20,000	10
Quaking Aspen	115,000	10,000	2
Palm	55,000	6,000	1
Wheat	3,000	70	0.1

Table 5.1: Table displaying number of triangles and leaves for a tree and the time which is needed to generate it

5.3 Conclusion

As already presented in the implementation chapter 4, a clear separation between generating structural data and creating the geometry, to visualize the data, was achieved. The majority of the rules are used to define the tree and its branching structure. Only a few rules are needed to generate the geometry. A semantic approach was shown to define tree models in a scene graph and use them to generate geometry out of this representation. This is a new way to generate data with a procedural geometry method.

Because of the two different scene graphs, a separation between the semantic model and the rendering graph used to generate the view is introduced. These two graphs are

combined via a controller in between. The controller generates the view graph out of the model by interpreting and applying parameters, and reacting on state changes in the environment.

Unlike other models, the dynamic scene graph is able to combine a common scene graph with a procedural geometry method. Other scene graph systems have just a focus on holding data and visualize it. Their dynamic behaviour is limited to parameter changes, in order to perform either an animation, select a level of detail or an image from a billboard cloud. Neither of them uses the scene graph to create data on demand, nor is it possible to implement this feature with only small effort.
The most tree generating systems are used to generate a tree and use it later in a scene graph, as two separate steps. The work from Deussen and Lintermann presented in section 2.2 and the work from Traxler and Gervautz in section 2.2 are two exceptions. These models combine the data structure for the generation of geometry with the modelling structure. The two methods are more dynamic and flexible than the other approaches. Both models are not pre-computed and they are generated on demand within the application. But the way they are used within the scene graph is very limited because they were not designed for heavy interaction with scene graphs.
The dynamic scene graph provides a much better extension in these points and allows interactive modelling and an intuitive storage within the application.

5.4 Outlook

A few features, mentioned in the Weber and Penn paper in section 2.2, are not implemented, for example the time dependent wind sway. To animate the trees based on the wind blowing from one direction, the visitation function of the rule has to calculate the transformation and apply them to the branches. Some pre-calculations done during the initialization of the rule would have to be moved into the visitation function.
The degradation feature, a level of detail method, could be implemented as described in the paper. Two better approaches for this feature are possible. The first one uses a custom traversal which moves through the generated tree structure and generates, depending on the detail level of an element, a different output geometry. The benefit of this approach is that this traversal could be implemented only on top of the geometry nodes. The other approach could use different rules for the components which generate the geometry. This would eliminate a second traversal but the whole tree would have to be re-evaluated for every detail level.

CHAPTER 6

Acknowledgements

This work has been done at the VRVis Research Center in Vienna, Austria under supervision of Robert Tobler. Special thanks are given to him for the support (and the patience) he gave me during the two years I needed to complete this thesis. His ideas and input were the crucial contributions to this work.
I would also like to thank all the people supporting me in the last years with advice, hope and motivation. Especially my parents Kathrin and Elmar who made it possible for me to study.
A last thank-you goes to my proof readers 'Scrub' and Stefan!

Bibliography

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[Bla08] Paul E. Black. tree (data structure). in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology, 14 August 2008. (accessed 1 July 2009) Available from: http://www.itl.nist.gov/div897/sqg/dads/HTML/tree.html.

[BO04] Don Burns and Robert Osfield. Open scene graph a: Introduction, b: Examples and applications. In *VR '04: Proceedings of the IEEE Virtual Reality 2004*, page 265, Washington, DC, USA, 2004. IEEE Computer Society.

[Die07] Wolfram Diestel. Arbaro software. *Arbaro software*, 10 Jun 2007. (accessed 5 August 2009) Available from: http://sourceforge.net/projects/arbaro/.

[dREF+88] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1988. ACM.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gre89] N. Greene. Voxel space automata: modeling with stochastic growth processes in voxel space. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 175–184, New York, NY, USA, 1989. ACM.

[GT96] Michael Gervautz and Christoph Traxler. Representation and realistic rendering of natural phenomena with cyclic csg-graphs. *The Visual Computer*, 12:62–74, 1996.

[Har06] John C. Hart. On efficiently representing procedural geometry, 2006.

[HD91] John C. Hart and Thomas A. DeFanti. Efficient antialiased rendering of 3-d linear fractals. *SIGGRAPH Comput. Graph.*, 25(4):91–100, 1991.

[Hon72] Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. In *Journal of Theoretical Biology*, pages 331–338, 1972.

[Kaj83] James T. Kajiya. New techniques for ray tracing procedurally defined objects. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 91–102, New York, NY, USA, 1983. ACM.

[KM09a] Bob Kuehne and Paul Martz. *OpenSceneGraph Reference Manual v2.2*. Blue Newt Softwar, LLC, February 2009.

[KM09b] Holger Kunz and Phillip Miller. NVIDIA® SceniX™ scene management engine, 24 Nov 2009. (accessed 6 December 2009) Avaliable from: http://developer.nvidia.com/object/scenix-home.html.

[LD96] Bernd Lintermann and Oliver Deussen. Interactive modelling of branching structures. In *SIGGRAPH 96 Visual Proceedings*, pages 139–151. Press, 1996.

[LD98] Bernd Lintermann and Oliver Deussen. A modelling method and user interface for creating plants. *Computer Graphics Forum*, 17(1):73–??, 1998.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, March 1968.

[LN03] J.P. Lewis and Ulrich Neumann. Performance of java versus c++, Jan 2003. (accessed 10 December 2009) Available from http://scribblethink.org/Computer/javaCbenchmark.html.

[Neu07] Carsten Neumann. Open SG Tutorial. *Open Sg*, 15 Aug 2007. (accessed 15 August 2009) Available from: http://opensg.vrsource.org/trac/wiki/Tutorial.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[Pru86] Przemyslaw Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of Graphics Interface 86 - Vision Interface 86*, pages 247–253, 1986.

[Ree79] Trygve Reenskaug. Models - views - controllers, 19 Dec 1979. (accessed 08 December 2009) Available from: http://folk.uio.no/trygver/themes/mvc/mvc-index.html.

[Rei02] Dirk Reiners. A flexible and extensible traversal framework for scenegraph systems. In *1st OpenSG Symposium*, 2002.

[RS05] Gerhard Reitmayr and Dieter Schmalstieg. Flexible parametrization of scene graphs. *Virtual Reality Conference, IEEE*, 0:51–58, 2005.

[RVB02] Dirk Reiners, Gerrit Voss, and Johannes Behr. Opensg: Basic concepts. In *1. OpenSG Symposium*, 2002.

[SC92] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 341–349, New York, NY, USA, 1992. ACM.

[Str93] Paul S. Strauss. Iris inventor, a 3d graphics toolkit. *SIGPLAN Not.*, 28(10):192–200, 1993.

[.th04] .theprodukkt. .kkrieger: chapter 1, 2004. (accessed 29 December 2009) Available from: http://kk.kema.at/files/kkrieger-beta.zip.

[TMW02] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. Mesh-based parametrized l-systems and generalized subdivision for generating complex geometry. *International Journal of Shape Modeling*, 8(2):173–191, 2002.

[VBRR02] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *EGPGV '02: Proceedings of the*

Fourth Eurographics Workshop on Parallel Graphics and Visualization, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Wer93] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[WP95] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128, New York, NY, USA, 1995. ACM.