Grimaldo

A Scriptable Framework for Developing Visual Applications

Master's Thesis

at

Graz University of Technology

submitted by

Martin Hecher

Institute for Computer Graphics and Knowledge Visualization (CGV),
Graz University of Technology
A-8010 Graz, Austria

Supervisor:	Dipl.-Inform. Dr.-Ing. Sven Havemann
Advisor:	Dipl.-Inform. Volker Settgast

Abstract

This thesis describes a novel way to integrate the Generative Modeling Language (GML) into custom applications with the presented Grimaldo framework. The framework extends the GML software stack with a new entity, the *GML context*. A context encapsulates a GML model into a nodal unit. The node based approach makes it possible to manage multiple GML models within a virtual 3D scene. To efficiently organize the scene Grimaldo utilizes a node-tree system that allows to store contexts in a hierarchical structure. Changes of the structure, like the addition, removal, etc. of a node are propagated by an event system. The application subscribes to these events to react on the changes. The node management system is completely exposed to the GML's scripting facility. It is therefore possible to dynamically create models and to organize them within the scene at runtime. The runtime modification possibility allows a fast creation of the desired functionality.

The Grimaldo framework provides a data management system to attach arbitrary data blocks to models. Data blocks integrate a synchronization mechanism to establish channels between the model and the custom application for exchanging data and to react on data changes on either side.

The framework also exposes the material system of the GML. The interface allows to use an existing material system to override the built-in shading of models.

The usage of the Grimaldo framework is demonstrated with the OpenSG system. The presented techniques are used to connect the node-tree with OpenSG's scene graph and to display and interact with GML models.

Kurzfassung

Diese Arbeit beschreibt einen neuen Weg, um die Generative Modeling Language (GML) in andere Applikationen zu integrieren. Dazu wird das vorgestellte Grimaldo Framework benutzt. Das Framework erweitert die bestehende Software-Basis der GML um ein neues Objekt, den *GML Context*. Das Objekt wird verwendet, um ein GML-Modell in einen Knoten abzubilden. Der knoten-basierte Ansatz erlaubt es, mehrere Modelle in einer virtuellen 3D-Szene darzustellen. Um die Szene effizient zu verwalten, verwendet Grimaldo eine Baumstruktur, die Knoten hierarchisch organisiert. Änderungen der Baumstruktur, wie das Hinzufügen eines Knotens oder das Entfernen, werden über ein Event-System propagiert. Die Applikation hat die Möglichkeit, bestimmte Events zu registrieren, um bei Änderungen enstprechend reagieren zu können. Das Management der Knoten wird mit der Scripting-Funktionalität der GML gesteuert. Dadurch können 3D-Szenen zur Laufzeit erstellt und bearbeitet werden. Die Möglichkeit, alle Aspekte der Szene zur Laufzeit zu verändern, ermöglicht eine schnelle Entwicklung der gewünschten Funktionalität.

Für einen einfachen Austausch von Daten zwischen einem Modell und der Applikation bietet Grimaldo ein eigenes System, das es erlaubt, beliebige Datenblöcke an einen Knoten anzuheften. Die Datenblöcke sind mit einem Synchronisations-Mechanismus ausgestattet, um einen Kommunikationskanal zwischen dem Modell und der Applikation aufzubauen. Über diesen Kanal können Daten ausgetauscht und abgeglichen werden.

Existierende Applikationen bieten in den meisten Fällen ihr eigenes Materialverwaltungssystem für den Renderingprozess. Um ein solches System gemeinsam mit der GML benutzen zu können, bietet Grimaldo die Möglichkeit, die eingebauten Materialeinstellungen der GML zu steuern und gegebenenfalls durch das eigene System zu ersetzen.

Als Anwendungsbeispiel für den Einsatz des Frameworks demonstriert diese Arbeit die Integration der GML in das OpenSG System. Die vorgestellten Techniken werden verwendet, um die interne Knotenstruktur auf den OpenSG Szenengraphen abzubilden. Es wird gezeigt, wie GML Modelle gerendert werden und wie die Interaktion mit den Modellen funktioniert.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Preface

I would like to thank Sven Havemann and Volker Settgast for their guidance and technical discussions during the writing of this thesis. It was fun to work in this inspirational environment.

My special thanks goes to my parents. Their unconditional support during my students' journey gave me the unique opportunity to develop my interests without any pressure. A lovely "Thank You" to both of you.

I would like to thank my girlfriend, Almina Bešić, who had to live with me and this thesis the last months. Without your ongoing support and daily encouragement this document would not have been finished.

I also like to thank my cousin and long-term flatmate Christian Koplenig. Our discussions and your interpretation of my ideas where very inspiring.

Special thanks goes to my sister, Maria Hecher. Without you, there would be no story behind the name of this thesis.

Contents

1 Introduction

The *Generative Modeling Language (GML)* forms a novel method for describing 3D models. It reflects a paradigm shift in representing models not as a static set of low-level shape primitives but as a result of modeling operations [Hav05b, p. 53]. The GML is a scripting language that provides the functionality to create complex 3D models and to add interactive behavior to them. The advantage to use a scripting language for this task is that users can dynamically develop 3D models without the need to compile the written code. To give instant feed-back on the creation process the GML comes with a complete runtime engine. The engine integrates the shape modeling functionality with a realtime viewer for the model. This combination even allows to interactively change the shape in realtime. 3D objects constructed with the GML are not simply geometric objects, they can be seen as a *tool*. Parameters describing the degrees of freedom of a model's shape can be defined and exposed to the user. The GML is best used with models defined by a narrow set of parameters. Additional data needed for the model creation is calculated from the given parameter set. Customizable visual helper objects - so called *gizmos* - are allowing a user to interactively change the parameters and adjust a model to its needs.

To use the power of the GML a developer is provided with the *GMLApplet* object. It forms a programming environment that encapsulates the internals of the framework and provides an interface to work with GML models. The current design of the GMLApplet is targeted to represent a single model. To use multiple models within an application each is represented by a single GMLApplet instance. Therefore, each model has its own set of resources, like materials and textures. There is no direct way to share such resources between models, which is inefficient. A second drawback is that the different models are isolated from each other, parameters can not be shared between them. For instance, when changing the height of a table model it is not possible to change the height of an appertaining chair accordingly. The lacking connection between the models prevents interesting interactive behavior. By pushing a button it is possible to change the style of one model, but not the style of other models of the same type in the scene.

The challenge is to remedy the limitation of the GMLApplet to represent a single model. This project presents a novel way to support the management of multiple models within one controller instance. For this purpose, a new entity is introduced, the *GML context*. One context relates to one model, where the GMLApplet manages a set of contexts. The single GMLApplet links together the different contexts and allows to exchange data between them. Thus, the GMLApplet is transformed from a single model controller to a scene controller.

The infrastructure for integrating GML contexts is provided by the *Grimaldo framework*, introduced in this project. The core of the framework is a nodal system that is organized in a hierarchical tree structure and is capable of managing arbitrary types of nodes. In this project a special node type is developed to encapsulate a GML context into a Grimaldo node, the *GMLContextNode*. The framework is directly integrated into the GMLApplet and is responsible for managing GML models via their GMLContextNode representation. The GMLApplet is forming the frontend for managing contexts with the help of the Grimaldo infrastructure.

The design focus of the Grimaldo framework is to provide developers the possibility and a guideline to integrate the Generative Modeling Language into their own applications. It focuses on the integration into existing systems, which already contain their own rendering, event handling and shading mechanisms. Grimaldo is targeted to take these circumstance into account and provides reasonable interfaces to integrate the GML into existing structures. The framework focus on three areas:

- data exchange between a GML context and the host-application
- dynamic mapping of the internal node structure onto a target system
- control over the the GML shading system.

Grimaldo provides a flexible data exchange method between a GML context and the host-application. A GML context can define parameters controlling the shape or other aspects of the model. For instance, the model of a table defines its height and width as parameter. Changing a parameter changes the geometric shape of the table accordingly. To be able to control the model parameters from within the host-application the outcome of this project provides the possibility to easily connect them to an element of the application, for instance to a visual slider of the graphical user interface (GUI), which then controls the value of the parameter. The communication is bidirectional. Changing the slider from the application changes the value of the GML model, changing the parameter with a GML commando updates the slider of the GUI. The feature is described in section 5.3.2.

The node management functionality of the Grimaldo framework is fully exposed to the GML scripting facility. This way a scene of models can be scripted and dynamically changed at runtime. In many cases an existing application provides its own node management system, for instance some type of scene graph. Grimaldo provides a technique to connect its internal node system to an external system. The external system gets notified whenever the Grimaldo node topology changes and can react to these events. It is therefore possible to script an external node management system via this connection. The details of the mechanism are described in section 5.3.1.

The GML has an integrated shading system to manage materials used for rendering models. Grimaldo provides control over the shading system and the possibility to override material settings on demand. The mechanism allows to use the material library of the host-application for the shading of GML models. The system is described in section 5.3.3.

1.1 Document Outline

The thesis is structured into different chapters to introduce the general idea of the project and to explain the different components. Chapter 2 discusses a related project that formed the first attempt to combine the GML with a nodal system. The chapter explains the goal of the project and the used design. Chapter 3 introduces the Generative Modeling Language and the software framework behind it. The description of the GML context idea is given in chapter 4. The parts of the GML framework that have to be adapted to the context design are identified and their working process is explained. Chapter 5 introduces the Grimaldo framework. The different core components are described and their usage is explained. A *Features* section shows the possibilities of the framework to interconnect with existing applications. As a showcase chapter 6 describes how to integrate the GML into a custom host-application. In this case the *OpenSG* system is used as a render engine for GML contexts. It shows how to use the Grimaldo framework to interconnect the GML with the OpenSG scene graph structure. The last chapter discusses the strengths and weaknesses of the presented work and gives an outlook on future work.

2 Related Work

The first approach to combine the Generative Modeling Language with a node-based system was made by Björn Gerth within the scope of the *EPOCH* project [Hav05a]. The approach is concentrating on the integration of the *combined B-rep* mesh datastructure [Hav03a] into the scene graph system OpenSG [Opea]. The combined B-rep is the primary mesh representation used by the GML to display geometric models. It joins polygonal and free-form shapes into a single surface representation. The representation supports view-dependent adaptive tessellation, as well as a *macro* system to save modeling operations that can be reapplied or undone [Ger05, p. 2].

Although the GML framework provides the possibility to create multiple meshes within one scene and to position these models arbitrarily in 3D space there is one major problem. The GML does not have a scene graph of its own, so all models exist within the same coordinate system. It is not possible to organize meshes in a hierarchy. A hierarchy is needed to reasonable aggregate a model from multiple meshes. For instance, the model of a car consists of the car body and four wheels. When moving the car the wheels should automatically move with the body. The wheels therefore have to be parented to the body to inherit its transformation matrix. With a scene graph this can be accomplished. The graph consists of nodes used to represent parts of a model. In this case the car body is the parent node with a given transformation matrix. The four wheels are child nodes with local transformations. During the rendering process the transformation matrices of a node is accumulated with the matrix of its parent, which allows to reposition a model with all its subparts. Another drawback of the current GML framework is that multiple instances of one model can not be accomplished, preventing an efficient reuse of existing geometry. [GBHF05]. Hence, a scene graph provides the data structure to enable multiple instancing.

The work of Björn Gerth utilizes the OpenSG library to overcome these limitations. OpenSG is a sophisticated scene graph system for realtime rendering. It is optimized for visualizing 3D scenes on multi-host systems, like *DAVE* [FHH03] environments, but is not limited to this field of application. The OpenSG scene graph functionality manages models in a hierarchical node structure. Nodes are organized in tree-form and consist of *inner* nodes, carrying information like transformations, and *outer* nodes or *leafs*, representing geometry. The structure allows to place transformation information before a sub-tree of nodes, which is then inherited by all nodes in the sub-tree. This way parts of a model can be relocated, keeping the spatial relationship of sub-models intact. In extending the scene graph to a directed acyclic graph (DAG) parts of the node tree can be efficiently reused in the scene [Ben03].

OpenSG uses a combination of *Nodes* to manage the scene graph structure and *NodeCores* to encapsulate the functionality of a Node, for instance the display of geometry. To integrate GML models into the OpenSG scene graph a custom NodeCore was implemented that maps the combined B-rep mesh into an OpenSG compatible form. The NodeCore is capable of displaying the mesh and supports the adaptive tessellation technique. The advantage of mapping a combined B-rep to a NodeCore is that the scene graph capabilities of OpenSG can be applied to a GML model. It is possible to relatively position different models to each other, as well as to use the directed acyclic graph to enable multiple instancing for GML models.

The combined B-rep mesh data structure is represented as an *Applet* within the the GML framework. The GML comes with a set of built-in applets, each of them forming an independent small program for a certain field of application. Besides the combined B-rep, which is integrated directly with the scene graph system, Björn Gerth's work allows to display arbitrary GML applets in OpenSG. For this two converter objects are used, the *AppletContainer* and the *AbstractAppletContainer*. Both objects wrap Applets that are then computed and displayed. The AbstractAppletContainer additionally allows to synchronize the applet to OpenSG cluster servers, allowing to render it on different hosts on the network. The technique to synchronize the rendering data between different cluster server is also used to save the data to a binary file. Therefore, the AbstractAppletContainer is a powerful object to enable Applets to be displayed in cluster environments and to be serialized to and loaded from a file.

Besides the integration of GML models into the scene graph system and the possibility to bring applets into OpenSG the GML was extended to access the scene graph with the scripting language. This allows to control the OpenSG scene graph structure with GML commands, giving dynamic control for creating nodes and cores and changing their state at runtime.

The work of Björn Gerth is a sound basis for rendering GML models and integrating GML functionality into the OpenSG system. However, it is directly bound to the OpenSG scene graph library. If an application needs to integrate the GML with the scene graph functionality it is forced to integrate the OpenSG library into the project. Especially in existing software projects this is not always feasible, as the software already implements its own rendering or node management system that can not be replaced. Moreover, OpenSG is a system with a big code base. For light-weight applications that would profit from the small footprint of the GML framework the integration of OpenSG would cause a disproportional overhead, especially if sophisticated features like cluster-rendering or stereo-view supported by OpenSG are not used.

This thesis takes a more general way to combine the GML with a node based system. The Grimaldo framework directly implements a node management system in form of a *node-tree* into the GML framework, with the advantage that no third party dependency is needed. Moreover, as Grimaldo was developed directly for the use with GML, the footprint of the resulting system is compact. The features of combining a node based system with the GML achieved by Björn Gerth's work are preserved, like dynamic creation of nodes or hierarchical systems of GML objects. However, applications that already use the GMLApplet for displaying GML models directly benefit of the new features when upgrading to the Grimaldo extended version and developers are provided a flexible toolset to integrate GML functionality into their custom applications.

3 The GML Framework

The Generative Modeling Language (GML) itself is simple, stack based programming language, similar to Adobe's PostScript [Ado99]. Although it is not intuitive to believe that such a low-level language is suited for creating complex 3D meshes it actually performs well for the construction of parametric models. However, the GML is not only a formal language. It forms a complete runtime engine for generating 3D models, render them and equip them with interesting interactive behavior [Hav05b]. The term *GML* is used for either the language itself or the complete GML framework henceforth, depending on the context.

The following sections introduce the language and describe its type system. The mechanism to extend the language and the runtime system is described to show how to integrate custom functionality.

3.1 Language Introduction

This section gives a short overview of the GML language. It is intended to introduce GML related terminology, as these terms are used throughout the thesis. Additional information on the topic can be found in [Hav03b]. There an introduction of the language is given, as well as a tutorial of the usage. An extended description can be found in [Hav05b, Chapter 5]. This section is based on the information given in this document. For self-containedness an excerpt of the content related to the GML language is given here.

The GML is a simple stack based programming language. Objects are represented by *token* on an *operand stack*. The token handling is done by the *GML interpreter*. It is responsible for processing a stream of individual token one after another. A token either contains data or a functional unit, an *operator*. A data token is simply put on the stack, if an operator token is encountered it is executed. An operator can access the operand stack and takes token from it which serve as input parameter. Inside an operator - in a way as a side-effect - work is performed on the objects represented by the input token. The result of the operation can be put back on the stack and serve as input for the next operator. As a result of the work within the operator, for instance, a three-dimensional shape is created, but the field of area is not limited to that. [Hav10]

The complete GML language can be described in a set of a dozen rules, listed in [Hav05b, p. 217]. These rules are sufficient to process GML programs. To execute a program the interpreter first has to convert the input character string into a token stream. The conversion task is accomplished by a simple *tokenizer*. Token within the character stream are separated by whitespaces. The tokenizer either identifies a token or converts the input into a *name* token. Therefore, almost any input can be tokenized. If a syntax error occurs the GML interpreter sets an error.

The following list shortly describes the token types natively supported by the language:

Atomic data types The GML comes with a set of built-in atomic types. These types - each with an example in parentheses - are: *integer* (123), *float* (123.45), *vec2f* (1.2,3.4), *vec3f* (1.2,3.4,5.6), *marker* ([,{,}), *string* ("abc") and *literal name* (/myname).

Dictionary A dictionary acts as a map and relates a unique key name to a token.

Array An array stores a list of token. The token can be of any type. An array can be either *literal* or *executable*. In case the array is literal the token is put on the stack, if it is executable the contained token are executed by the interpreter. An executable array therefore acts as a *function*.

Operator Operators pop token from the operand stack and process them internally. The result can be pushed back on the stack. Operators are described in more detail in section 3.2.

```
 1 class GMLResource : public GMLOp
 2 {
 3 public:
 4     // type(): just check for type; is(): also check for validity
 5     virtual bool   isType  (const Token& ticket);
 6     virtual bool   isValid (const Token& ticket);
 7
 8     // The ticket serves as "fake dictionary", the key can have any type.
 9     virtual Token* getToken(const Token& ticket, const Token& keyname);
10     virtual bool   putToken(const Token& ticket, const Token& keyname, const Token& value);
11
12     // code serialization / de-serialization
13     virtual bool   parse   (Token& t, const char* txt);
14     virtual bool   toText  (const Token& ticket, char* buff, int len);
15 };
```

Figure 3.1: The GMLResource interface (truncated)

The GML interpreter has no built-in functionality, it is added by operator libraries [Hav05b, p. 218]. The GML has a core operator set that provides basic functionality for working with the built-in types, flow-control or comparison actions. Custom operators can be added to the language. In most cases new operators are registered by a *resource*. A resource encapsulates one or more objects and provides corresponding token types. To work with the custom types the resource defines operators. These operators are registered to the GML system during the initialization of the resource. The mechanism is described in section 3.2.1.

The GML language introduces two new features that are not present in PostScript, the *DictPath* token and *named registers*. A DictPath allows to retrieve a value from a dictionary in using a *dot notation* like so: `MyDict.myValue`, where `MyDict` is a dictionary token and `myValue` the key name of a value. As a result the token denoted by the key name is put on the stack. Registers ease the handling of values and avoid *stack acrobatics*. A register provides a convenient way to store a value within a named slot and access it with this name. Registers are as fast as accessing a value from the stack.

3.2 Extending the GML with Resources

To add custom functionality to the GML *resources* are used. Their management is accomplished by the GML interpreter. A resource encapsulates one or more object types and makes them known in form of token *types*. New types provide a fast mechanism to access the objects on the operand stack so that they can be used as input parameters for operators or be pushed onto the stack as the result of an operator calculation.

Besides the ability to represent arbitrary objects as a token the resource provides the *fake dictionary* functionality. Each resource can be treated as a dictionary, meaning that key/value pairs can be inserted into a resource and a value can be retrieved by a given key. In contradiction to a normal dictionary the keys or values can be restricted to be of a certain type [Ger05, p. 64].

Listing 3.1 shows the truncated interface of a `GMLResource`. Line 5 and 6 check whether a token is of the type represented by the resource and if the referred object is valid, respectively. Line 9 and 10 implement the fake dictionary functionality. Line 13 is responsible for converting a given character array into a token. Finally, line 14 does the opposite in outputting a token into a character buffer, for instance to serialize it to a file or to display the textual representation of the token in a graphical user interface. A GMLResource derives from the *GMLOp* class. This class is also used to represent a GML operator. Its `name()` method has to be implemented to give the resource an unique name. This name is registered within the interpreter together with the resource. The same name is used by an entity to retrieve the resource from the interpreter, if needed.

3.2.1 The Initializer System

As stated in section 3.2 the GML itself does not provide built-in functionality. The functionality is added by resources. The GML provides a flexible system to add resources to a GMLApplet, the *Initializer* mechanism. When developing a

```
 1 class GMLAppletInitializer
 2 {
 3 public:
 4    GMLAppletInitializer() ;
 5
 6    bool                setStartupPathFromEnvironment();
 7    void                setStartupPath              (const std::string& path);
 8    const std::string& getStartupPath              () const;
 9
10    virtual bool initResources   (GMLInterpreter& inter) = 0;
11    virtual bool initOperators   (GMLInterpreter& inter) { return true; }
12    virtual bool deleteResources () = 0;
13
14 protected:
15    std::string v_startupPath;
16 };
```

Figure 3.2: The GMLAppletInitializer interface

new resource a corresponding initializer is defined that takes care of

1. initializing the resource
2. connecting the resource to the GML interpreter in registering operators and token types.

An initializer is defined in deriving from the *GMLAppletInitializer* base object. Listing 3.2 shows its interface. The `initResource` method in line 11 is responsible for instantiating the resource and adding it to the resource management of the interpreter. If the resource is an *Applet* (see 3.3) it is also added to the GML event handler, which takes care of rendering the Applet and integrating it into the event handling process. Line 12 defines the `initOperators()` method. It registers the operators defined by the resource. Finally, `deleteResources()` cleans up the resource at the end of the GMLApplet's lifetime. Additionally, line 7 to 9 allow to declare a system path to load additional data from, if needed. The material resource of the GML uses this to load the material definitions form an external file.

After defining a resource and its initializer the GMLApplet has to be informed about the new functionality. For this purpose it provides the `addInitializer()` method that takes a pointer to the initializer as argument. Multiple initializers can be added. The actual registering is triggered by a call to the `initialize()` method of the GMLApplet. From then on the resources are known to the interpreter and can be required from it, for instance, in an operator.

The GML provides a default initializer that summarizes the registration of all core resources. This simplifies the process and takes care of internal dependencies. The default initializer is implemented in the *GMLAppletInitStandard* object.

3.3 Applets

The majority of the resources in the repository of the GML are made up by so called *Applets*. An Applet can be seen as a complete OpenGL application which renders content onto the screen and provides selection facilities. The GML uses a distinct interface for adding custom Applets to the system, the *Applet* interface. It is shown in listing 3.3. Lines 4 and 5 are responsible for the bounding box handling. Lines 7 to 9 form the rendering interface. The rendering cycle is separated into a `framePrepare()`, `frameDraw()` and `frameFinish()` method. The intention is to support stereoscopic rendering, in which case the `frameDraw()` method is called twice, once for the left stereo buffer, once for the right. Finally, lines 10 to 12 are used to enable the Applet to be picked. For this purpose the *SelectionEventRecord* object contains information about the pick ray and the distance to the last picked object. In case the Applet's geometry is hit the SelectionEventRecord object is updated with information, like the exact hit point.

The following list shows two representatives for the GML's Applet collection. They are used in several sections of this thesis and are described in short here:

```
class Applet
{
public:
   virtual void   updateBBox ();
   virtual Box3f boundingBox();

   virtual void   framePrepare();
   virtual void   frameDraw  () = 0;
   virtual void   frameFinish () ;

   virtual int    renderSelect  ( int startName,
                                  SelectionEventRecord& ser);
   virtual int    querySelection(SelectionEventRecord& ser);
};
```

Figure 3.3: The Applet interface (truncated)

BRepProgressive The *BRepProgressive* Applet encapsulates a combined B-Rep mesh. It is the default mesh representation of the GML for models. It allows to use free-form and polygonal faces to be combined within one mesh. The control mesh can be changed on-line, which is necessary for interactive mesh modeling as well as semantic level-of-detail [Hav05b, p. 179].

BallAndStick The *Ball and Stick* Applet can be used to equip a model with visual markers. The Applet provides spheres and arrows that can be clicked and dragged, allowing interactive behavior with a model. The Applet can also be used to create simple models consisting of spherical and cylindrical primitives.

3.3.1 Wrapping Applets into Resources

As stated before, the GML interpreter extends its capabilities with resources that (a) define their own token types and (b) provide operators to work with resources. Now, to use the functionality of an Applet it has to be wrapped into a resource. For this task the GML framework provides the *GMLResourceWrapper*. The wrapper's design makes it possible to take an existing Applet and equip it with an arbitrary interface. The advantage is that the Applet does not have to be derived from the GMLResource interface to integrate it into the GML. The GMLResourceWrapper takes care of encapsulating the Applet and provides GMLResource interface methods to the interpreter. Hence, the GMLResourceWrapper acts as "gluing" component between the actual Applet and the desired interface. This approach is very flexible. It can be used to wrap arbitrary objects into resources, not only Applets.

The GMLResourceWrapper is shown in listing 3.5. In line 2 the desired resource interface is inherited, denoted by the *Resource* template parameter. The `setResource()` method in line 8 sets the Applet (or an arbitrary object) into the wrapper. In case of Applets an extended version of the GMLResource interface is used as `Resource` template parameter. The reason being is that Applets have to be rendered and can be picked, a task the *GML event handler* is responsible for. To make Applets within the GMLResourceWrapper accessible by the event handler the interface of the basic GMLResource is extended. This is done in deriving a new resource type from the GMLResource base, the *GMLResourceApplet*. It defines additional access methods shown in listing 3.4. The methods simply forward to the corresponding methods of the contained Applet, which is accessed by the `applet()` method. As the GMLResourceWrapper is used to encapsulate the Applet it is also responsible to correctly implement this method to deliver a pointer to the encapsulated Applet.

The *GML Event Handler* is responsible for the integration of Applets into the runtime engine. It takes care of rendering and picking the Applets and allows to attach *callbacks* to an Applet. This functionality is described in section 3.4. An Applet, or to be more precise, an GMLResourceApplet is added to the event handler in its initializer (see section 3.2.1).

```
class GMLResourceApplet : public GMLResource
{
 public:  ...

  virtual  Applet∗ applet          ()  = 0;

  virtual  void    updateBBox      ()          {          applet()→updateBBox();  }
  virtual  Box3f   boundingBox     ()          { return   applet()→boundingBox(); }
  virtual  void    framePrepare    ()          {          applet()→framePrepare(); }
  virtual  void    frameDraw       ()          {          applet()→frameDraw();    }
  virtual  void    frameFinish     ()          {          applet()→frameFinish();  }

  virtual  int     renderSelect    ( int  sn,  SelectionEventRecord& ser) { return applet()→renderSelect(sn,ser); }
  virtual  int     querySelection (SelectionEventRecord& s)            {  return  applet()→querySelection(s);     }

 private :  ...
}
```

Figure 3.4: GMLResourceApplet interface (truncated)

```
template<class T, class Resource>
class GMLResourceWrapper : public Resource
{
public:
    typedef GMLResourceWrapper<T,Resource> ThisType;

    GMLResourceWrapper () : v_object(NULL) {}
    void   setResource   (T∗ object)   {  v_object = object;  }
    T∗     get              ()  const      {  return  v_object;     }

    static   ThisType∗ getResource (GMLResource∗ res) {
        if (res==NULL) { return NULL; }
        return  dynamic_cast<ThisType∗>(res);
    }
    static   T∗          get              (GMLResource∗ res) {
        ThisType∗ t = getResource(res);
        return  (t==NULL) ? NULL : t→get();
    }

private:
    T∗    v_object;
    bool v_ownObject;
};
```

Figure 3.5: GMLResourceWrapper interface

3.4 Callbacks and Temporary Applets

3.4.1 Object Picking and Show-Ticket Callbacks

Each token type that refers to an Applet of the event handler is a so called *show-ticket*. The callback management system allows to attach a callback to an arbitrary show-ticket within a GML script. A callback consists of a GML program that is executed if a user picks the corresponding Applet, referred to by the show-ticket.

Registration of a callback is done with the `io-callback` operator. It pops a show-ticket from the stack, as well as a data dictionary and an arbitrary client data token. The data dictionary usually contains model specific data and its update function. The client data token is application specific and can also be a dummy value. The last parameter is the callback function itself. The `io-callback` operator internally checks if the show-ticket belongs to a registered resource in cycling through the registered GMLResourceApplets and utilizing their `isType()` method, which validates the ticket. If a corresponding resource is found the callback and its data is saved into the internal callback list. The entries of the list are *CallbackItem* objects, which are storing the actual show-ticket, the client dictionary and data, the type of the responsible source as integer ID and the callback token itself.

If the user clicks into the viewport of the GUI application the GMLApplet calls the `findSelect()` method of the event handler. This method takes the input event data in form of a `GMLEventIO` object that stores event specific data, like the pushed mouse button or the 2D-coordinates of the mouse pointer. The `findSelect()` method cycles through the registered resources and determines if an Applet was hit. In this case it enters hit data like the token representation of the hit object and the coordinates of the hit point. If the hit token has an callback attached the callback ticket is also entered into the `GMLEventIO` object.

If a callback was attached to the picked object it gets executed after `findSelect()` finishes. Before the GML program is called the relevant data of the `GMLEventIO` object is converted into a dictionary that is put onto the dictionary stack, so that the contained data can be used within the program. Also the dictionary and the client data registered with the `io-callback` operator are put onto the stack to be visible within the callback program. Finally, the program is executed and can use the provided data to react to the mouse pick.

3.4.2 Temporary Applets

The GML framework provides pre-defined *temporary Applets* that are similar to show-ticket callbacks in that they also can contain GML commands as callbacks. However, they additionally are allowed to draw content onto the screen and can be picked. For instance, the operator `io-tripod-gizmo` registers a temporary Applet that draws a visual tripod representing a coordinate system onto the screen. The operator also registers a callback function that is called if the tripod is moved and rotated.

Temporary Applets are also used for two operators that are explicitly mentioned here as they are very useful for a variety of applications: the *capture mouse* and the *get key* applet. The first one is registered with the `iocapturemouse` operator. It registers a client dictionary and a callback function that is called whenever a mouse event [1] is issued. An interesting application for this Applet type can be found in [BHSF09], where it is used to allow a user to visually sketch a form that gets converted into a subdivision surface model.

The second temporary Applet is registered with the `iogetkey` operator. It takes a client dictionary and a character denoting a keyboard event. It registers a callback function, too, that gets executed if the registered keyboard event is fired. This callback can be used to completely script the keyboard interaction with a GML model.

3.5 What is a GML model?

The runtime frontend of the framework is the `GMLApplet`. It allows to control the GML as well as to embed it into custom applications. The GMLApplet forms a unified interface to the components making up the GML runtime. The runtime itself consists of two main components: The interpreter and a set of resources. The interpreter is responsible for managing the resources and provides access to them within the framework. With the help of the interpreter all resources internally can access each other and use the provided functionality. A user accesses the resources with the Generative Modeling

[1]The mouse *move* event is excepted for performance reasons

Language. As stated above, the language is similar to PostScript and utilizes operators to connect the interpreted language with a resources. The GML itself does not provide any functionality on its own and not even has its own keywords. All the operators are provided directly by the resources. The GML framework provides a set of existing resources that can be combined to create the model.

How is a GML model created? For this task the GMLApplet provides all necessary components. At first the resources that will make up the model have to be added to the GMLApplet and have to be initialized. The initialization mechanism is described in section 3.2.1. In short, within the initialization phase the resources add their types and operators to the interpreter. Moreover, Applets are registered to the GML event handler to be integrated into the rendering and the event handling process. After the initialization the resource types and its operators are known to the system and can be used. The model is configured in utilizing the GML language. For this purpose the GMLApplet provides access to the GML interpreter. Two methods are responsible for loading GML scripts containing a command stream, the `loadUserlib()` and the `call()` method. The first loads a GML code library, whereas the latter one directly inputs GML commands. It is also possible to load libraries directly with the `call()` method. The registered operators can now be used by the user within a GML script to build up the model. For instance, the BRepProgressive resource provides modeling operators to create a geometric shape.

To allow a user to interact with the model callbacks or temporary Applets can be registered. Both are described in section 3.4. With callbacks small GML programs can be attached to visual Applets of the model. The GML program gets executed if the user clicks or drags the element the callback was added to. For instance, the GML model of a simple table can be parametrized with two parameters, a width and a height. In changing these parameters the visual appearance of the table is changed, too. To not have to change the parameters manually in the script code the programming logic defines *gizmos* with the help of the BNS resource. For both parameters of the table one gizmo object is created. A callback is attached that converts the position of the gizmo into a numerical value which serves as value for the parameter. So, dragging a gizmo with the mouse immediately changes the parameter and updates the visual representation of the model in realtime.

After all token of the GML script are processed the model is created and the interaction with the user is defined with the registered callbacks or temporary Applets. After the initial model is created the GML runtime takes care of rendering, input processing and callback handling. The runtime is encapsulated in the GML event handler. The rendering procedure of the event handler is executed once in every frame, where all registered Applets are rendered. The input event handling takes care of offering the input data to the Applets. The runtime also takes care of object queries. If a mouse input is processed the system determines if a show-ticket object is hit and an eventually registered callback is executed.

4 The GML Context Extension

4.1 Concept

4.1.1 GML Contexts as Resource Sets

The *GML context* idea describes a novel way to work with GML models. It changes the paradigm of using one model controlled by a GMLApplet to the more flexible concept of controlling a whole scene of models with a single GMLApplet instance. This possibility equips the GML shape modeling engine with an additional scripting facility for the procedural creation of 3D scenes. The combination of both gives artists and developers a powerful tool for visualizing graphical environments.

The current implementation of the framework represents one model with one instance of a GMLApplet. The problem with this design is that three different versions of the same model, for instance, three chairs with different heights, required three distinct GMLApplets, each with an independent interpreter holding its own set of resources. The drawback of this approach is that (a) no resources can be shared between different models, and (b) no parameters can be transferred from one object to the other. The goal of this thesis is to remedy the need to instantiate multiple GMLApplets for creating multiple models. To achieve this goal the current GML framework implementation has to be extended to support the management of multiple models, instead of a single one. To achieve this a new entity is introduced, the *GML context*.

A GML context is designed to represent a fully functional GML model. The basic idea is that the set of resources making up the model is summarized into one context. A context is then treated as a node entity managed by the GMLApplet. This approach has several advantages over the representation of one model by one GMLApplet instance. The main advantage is that all contexts can share the same GML interpreter. This *unified interpreter space* forms a link between all GML contexts, allowing to share data between them. To come back to the example of the introductory part, the height of a table model is defined by a parameter. Changing this parameter now allows to also change the height parameter of a chair besides the table, as both models are represented by a GML context that is managed by the common interpreter. The interpreter has access to both parameter sets, making it possible to react on changes and adjust dependent models. The unified interpreter space also is key to enable the script based creation of models in utilizing the GML scripting facility. This enables an interesting new possibility for script developers: scripted generation and control of not only one model but multiple models forming a complete 3D scene. The advantage of separating models into contexts becomes clear in this example: To create a scene with different models without the context it is first necessary to position the starting point of a combined B-rep mesh to the desired point in space. Then one part of the model is created in utilizing the modeling operators. To add a second part of the model the starting point is repositioned and the part is created. To reposition the model parts the same GML commands have to be executed with different starting points. To create an animation, for instance, in each frame the complete model has to be recalculated to make it move. With GML contexts both model parts are created in an independent context. As such their rendering process can be influenced, which makes it possible to render each parts on different positions. The mesh data structure does not have to change, therefore it is only necessary to compute the geometry once. Section 4.2.2.2 describes how a context can store spatial data that can be used during the rendering process.

A further advantage of the context design is the possibility to share resources like materials, textures, camera settings, etc. between different contexts, enabling an efficient reuse of existing resources.

4.1.2 Context-Aware Resource System

The main challenge of integrating GML contexts is to find a way to allow the GML to work on different sets of resources. The integration has to preserve the compatibility of existing GML scripts, so that they can be reused without having to perform code changes on them. Moreover, the interface of the GML frontend, the GMLApplet, should not have to be changed, so that existing applications with GML support directly benefit from the extension after an upgrade. In the optimal case no or at least only small code changes to existing applications should be needed to directly use the extended features of the GMLApplet with GML context support.

To achieve these goals the internal GML resource management system has to be made *context-aware*. The current implementation of the interpreter and the event handler are working on the resource set that is present after initializing the GMLApplet. With the context extension it has to be ensured that there is always exactly one active context at a time that provides a valid resource set. For this reason a so called *focus context* is introduced. This context denotes the active resource set the interpreter and the event handler are working on. The focus concept is the key mechanism to enable the GML interpreter to work on multiple contexts. When setting a context as focus context all actions of the GML framework that access resources are working on the resource set provided by the focus context. This includes GML operators executed by the interpreter as well as the runtime system that is working with Applets. This way multiple contexts can exist within the same interpreter space. If the model of a context has to be accessed it is activated as focus context. With this approach multiple contexts can be worked on by the interpreter in sequence, allowing to change the models in the scene. To actually change the focus context a new GML operator pair - `node-begin` and `node-end` - is provided with the operators introduced in section 4.2.1.

The context-awareness of the resource system is controlled with the focus context. Within the GML framework not only the interpreter accesses resource. They can also be required by a host-application directly from the interpreter. This case has to be handled the same way as the resource access from within the GML language, the access to the resource does always have to be delegated to the resource set of the active focus context.

The focus context also ensures that the compatibility to existing GML scripts is kept by. In the initialization phase of the GMLApplet an initial context is created, the *root context*, and is activated as focus context. The root context is equipped with the initial set of resources. These resources are used to initialize the GML type system. Afterwards, the root context holds a set of resources and provides a GML environment compatible with existing GML scripts. This environment can be used in a traditional way, meaning that scripts executed in this environment do not have to be aware of the context extension.

A big challenge in changing the resource management system is to stay compatible with the GML operator system. Operators are the ”heart” of the GML language and are accessing and working on registered resources. Therefore, extending the GML with contexts has to ensure that the operators are compatible with the changes.

The idea to make the resource management system *context-aware* is to adapt the *GMLResourceWrapper* object. As described in section 3.3.1 this object forms the main mechanism to integrate new functionality into the GML. To discuss this idea an example is examined to see if the approach is suited. The example code in Listing 4.1 shows a typical implementation of a GML operator that accesses a resource. The fictional resource object *PrimitiveFactory* and the GMLResourceWrapper *GMLResource_PrimitiveFactory* are outlined shortly, only showing the implementation necessary for this illustrating example. Lines 1 to 8 are defining the PrimitiveFactory class with the method `addSphere()`, taking a point in space as midpoint of a sphere and a radius. The method adds a sphere with the given parameters to the render routine of the Applet. Lines 10 to 14 then wrap the resource object into a GMLResourceApplet that can be handled by the interpreter. Beginning with line 16 a new operator `add-sphere` is defined. In its `init()` method it requires the Primitive resource from the interpreter and stores it into a member variable. The code uses the `GMLGetResourceClass` macro that simply utilizes the `getResource()` method of the GML interpreter to get the resource and automatically casts it to the correct type. If the operator is called the `apply()` method checks the input parameters for validity. In line 34 the actual resource object - the PrimitiveFactory - is extracted out of the GMLResourceWrapper and its `addSphere()` method is used to add a new sphere with the given parameters to the render routine.

The first intention in the design phase for making the resource system context-aware was to change the central registration unit for resources, the interpreter. It provides an distinct interface for managing resources, which is presented by the two methods:

```
bool         addResource(GMLResource∗ resource);
GMLResource∗ getResource(const char∗ name);
```

In a first approach these two where extended for enabling a context-sensitive registration and delivery of resources. Unfortunately, it turned out that the operator system, especially the way operators are registered, was not compatible with this approach. The problem is that an operator requests the resources it intends to work on in its `init()` method with the `GMLGetResourceClass` macro. With a context-aware interpreter the `init()` method returns a pointer to the resource of the currently active context. The problem is that when the operator is actually called it accesses this pointer in its `apply()` method. If another context was created before the call to `apply()` this context is active now and the

```
class PrimitiveFactory : public Applet {

public:
  void addSphere(const Pnt3f& midpoint, float radius) {
    v_spheres.push_back(new Sphere(midpoint,radius));
  }

  // Applet implementation omitted for simplicity

private:
  std::vector<Sphere*> v_spheres;
}

class GMLResource_PrimitiveFactory
    : public GMLResourceWrapper<PrimitiveFactory,GMLResourceApplet> {

  // Implementation is omitted for simplicity
}

class GMLOpPrimitiveCircle : public GMLOp
{
public:
   GMLOpPrelude(PrimitiveSphere,prim-sphere);
   GMLOpComment("m:P3 r:Float prim-sphere ->");

   virtual bool init (GMLInterpreter& inter) {
      r_prim = GMLGetResourceClass(Primitives);
      return (r_prim!=NULL);
   }

   virtual bool apply (GMLInterpreter& inter) {
      Token* v1 = inter.pop_stack();
      Token* v0 = inter.pop_stack();
      if (inter.error() || !v1->isFloat() || !v0->isP3()) {
         return inter.setError(GMLInterpreter::TypeError);
      }

      r_prim->get()->addSphere(v0->toP3(), v1->toFloat());

      inter.commit_pop();
      return true;
   }

   GMLResource_PrimitiveFactory* r_prim;
};
```

Figure 4.1: Example operator for demonstrating resource access via a GMLResourceWrapper

operator is intended to work on the active context's operator set. However, the pointer to the resource still points to the previously active context, leading to wrong behavior. For instance, in case of an modeling operator that is working on a mesh the wrong mesh would be changed. The only way to deliver a valid resource that points to the correct context is to reinitialize all operators on a focus context change, which would assure that the correct pointer is accessed within the operator's `apply()` method. However, this method is inefficient and does not scale well.

This problem can be solved in adapting the GMLResourceWrapper to handle contexts correctly, and leaving the main resource management of the interpreter intact. In this case the resource registration to the interpreter is working as usual. If an operator requests a resource in the `init()` method the pointer to the registered resource is delivered. This same pointer is then used in the `apply()` method. However, the access to the actual resource object is done in using the `get()` method of the casted GMLResourceWrapper. This method can be adapted to determine the currently active context, request the desired resource object from it and return the object. The result is that the operator is working on the resource object that corresponds to the current context, which yields the desired behavior.

The described approach is based on two assumptions:

1. The resources used in operators are GMLResourceWrapper objects
2. All data specific to the GML model, like the created `Sphere` object in the example above, is always stored within the resource object itself (in the above example the PrimitiveFactory), and not in the GMLResourceWrapper.

The majority of the available GML resources do meet these two assumptions, as this implementation style is the suggested way of extending the GML. There are exceptions to that rule, the GML event handler resource is not implemented via the GMLResourceWrapper, but directly implements the GMLResourceApplet interface. The integration of this resource to correctly interact with contexts is described separately in section 4.2.3.

The GML framework does have other resources that do not meet the above assumptions. However, the following description on how to adapt the GMLResourceWrapper relaxes the strictness of the assumptions in introducing *global resources*. With global resources it is possible to get around the need for using a GMLResourceWrapper to be compatible with the context extension.

To demonstrate the working process of an adapted GMLResourceWrapper it is first necessary to (a) generally initialize the resource system of the interpreter and (b) register resource-sets to a context.

4.1.3 Context Management

With the introduction of GML contexts the internal working process of the framework has to be changed. The current implementation is a *single-node* system. One GMLApplet represents exactly one model. This thesis changes this paradigm in allowing a single GMLApplet instance to represent multiple models in form of GML context nodes. To achieve this a new node management system is introduced that maps a GML context to a nodal unit. The *Related Work* discussion in section 2 describes a prior implementation to combine a node-based management system with the GML. In this case the OpenSG scene graph system was extended to integrate the GML, where an OpenSG NodeCore encapsulated an instance of the combined B-rep resource. The disadvantage of this approach is that the scene graph functionality depends on the OpenSG system. The GML environment is available on various platforms, like

- Microsoft Windows
- Linux
- MacOS
- Apple iPhone
- Nokia N900
- etc.

Although OpenSG supports a variety of platforms, there is always the possibility that the GML is ported onto a new platform with no OpenSG support. OpenSG is distributed under an open source license [LGP], that allows to modify and adapt the source code, but time and a fairly amount of knowledge is needed to port the library to a new platform. Especially for unexperienced OpenSG programmers this would be a time consuming task.

To overcome the dependency and portability issue, a compact framework was created that directly extends the GMLApplet with a node-based management structure. The *Grimaldo framework* introduced in this thesis provides a node-tree structure for storing GML contexts as well as various techniques to work with the nodes. Additionally, the framework acts as flexible connection layer between the GML and a host-application. It provides general interfaces targeted to make the connection between the GML and arbitrary software packages a straightforward task. The Grimaldo framework provides

- a hierarchical node structure for managing multiple objects
- a runtime system that controls the rendering and the handling of input-device data
- a flexible data management system with notification capabilities for data changes
- an *event system* for informing host-applications about internal state changes
- control over the GML shading system.

To make use of the provided functionality it is necessary to integrate the GML context structure into a node the Grimaldo framework can work with. For this reason a node interface is provided that allows to integrate custom nodes. The interface is described in section 5.2.2.2. The system is comparable to the Applet system of the GML, where a standalone OpenGL application can be wrapped into an Applet. This Applet is then made available to the GML system in making use of a resource that connects the Applet to the GML interpreter and the event handler. Via the node interface the Grimaldo framework provides a similar way to integrate custom functionality. In section 4.2.2 the *GMLContextNode* object is described that implements a Grimaldo compatible node type abstracting a GML context.

The introduction of GML contexts not only affects the GML language, but also the runtime system. The runtime system - represented by the GML event handler - is responsible for

- rendering
- input event handling
- picking of objects and
- callback management.

With contexts the GMLApplet is extended to manages multiple nodes. This is directly affecting the runtime system in all major parts. To make the runtime system context-aware its event handler is exchanged with the equivalent component of the Grimaldo system, the *SceneViewer*. A scene viewer fulfills the same task as the GML event handler, however, it is directly working with GMLContextNodes. The scene viewer does not completely replace the GML event handler. It acts as an adapter layer that uses the GML event handler as backend to perform tasks like rendering and the dispatching of input events. However, the scene viewer organizes the process of the GML event handler and takes into account all existing nodes. The integration of the Grimaldo scene viewer and its combination with the GML event handler is described in section 4.2.3.

To connect the Grimaldo framework with the GML a new resource, the *GMLResource_Context*, is developed. The resource encapsulates the Grimaldo framework and provides access to its components. The resource also implements operators that allow to control Grimaldo via the GML scripting facility. The Grimaldo resource is described in section 4.2.1.

4.1.4 Summary

The concept of the GML context system combines the GML with a nodal management system, the Grimaldo framework. The connection is happening on two levels:

Node management extension The Grimaldo framework is used to extend the GML with a nodal system that takes care of the node management and the 3D runtime handling, like rendering of nodes. With the outcome of this thesis a single GML model is represented by a GML context. The Grimaldo system is used to manage these contexts. To allow dynamic control over the Grimaldo system GML operators are provided that connect the functionality with the GML scripting language. A user can then create and manage multiple GML models from within a GML script.

GML language interconnection The interpreter generally provides operators to perform actions on a set of resources that are making up a model. With the GML context extension it has to be ensured that the interpreter is always working on the correct resource set. For this reason the Grimaldo framework provides an operator that sets a specific context as focus context. The changes of the GML resource management described in the next sections ensure that operators are always working on the resource set of the focus context. In this way it is possible to switch between the contexts of the scene and execute GML commands on them.

GML operator	Description
node-controller	Puts the Grimaldo Controller onto the stack.
node-create	Creates a node of the given type.
node-clear	Clears all nodes under the root node and resets the root node.
node-begin	Activates the given node as focus context.
node-end	Deactivates the current focus context and activates the last active one.
node-addchild	Adds a node as child of a parent node.
node-removechild	Removes a child node referenced by an index or the node itself from a parent node.
node-removechildren	Removes all children from a node.
node-getparent	Returns the parent of a node.
node-find	Returns the node with the given name.
node-setname	Sets a name to the node.
node-getname	Returns the name of the node.
node-current	Returns the currently active focus context node.
node-getroot	Returns the root node of the Controller.
node-translate	Translates the model by a given vector.
node-scale	Scales the model by a given vector.
node-rotate	Rotates the model. A vector is used to describe the axis angles.
node-gettranslate	Returns the node's translation vector.
node-getscale	Returns the node's scale factor as vector.
node-getrotate	Returns the node's rotation. A vector is used to describe the axis angles.

Table 4.1: Node operators

4.2 GMLApplet Integration

The following sections first describe the integration of the Grimaldo framework into the GMLApplet. Next the GML context is implemented as a node compatible with the Grimaldo system. The final section describes the changes that are necessary to adapt the GML runtime system to work together with context nodes.

A complete description of the Grimaldo framework is given in chapter 5. The following sections use its base components to connect the node management functionality and the 3D runtime system with the GML framework. The components are shortly introduced at the time of their first usage. For more in depth information the reader is recommended to read chapter 5.

4.2.1 Grimaldo Resource

The management unit of the Grimaldo framework is the *Controller*. It is responsible for managing different node types and controls the life-cycle of node instances. In this thesis the Controller is used to manage multiple GML contexts. The integration of a GML context as node type into the Grimaldo system is described in detail in section 4.2.2. The interface and a usage description is given in section 5.2.4. Here the connection with the GML is described. The GML integration provides the possibility to create, modify or removed GML contexts from within a GML script.

To make the Grimaldo Controller usable within the GML framework it is wrapped into a GML resource. The resource is named *GMLResource_Grimaldo* and is accessible via the default `getResource()` method of the interpreter. With the resource the node management functionality is exposed to the GML language and can be used in GML scripts to create instances of GML contexts. Host-applications can also directly use the functionality via the C++-interface, if they have to.

The resource exposes two new types to the GML type system: the *Context* and the *Controller* type. The Controller type is used to register data with a host-application, a feature described in section 5.3.2.3. Via the Context type contexts and therefore GML models can be accessed on the operand stack and used by the Grimaldo resource itself, as well as by custom resources that work on contexts. To create and manage context token the Controller interface responsible for managing nodes is exposed in the *node* operator namespace. Table 4.1 describes the provided operators and explains their usage.

Three of the operators are explicitly explained:

`node-getroot` The Grimaldo node system is organized in a node-tree, as explained in section 5.2.2. The Grimaldo runtime system responsible for rendering the nodes is starting the rendering process from a root node, which is created when the Grimaldo Controller is instantiated. This root node is requested in a call to `node-getroot`. It puts the root node onto the stack. To add a node created with `node-create` to the system it has to be added to the root node or to an arbitrary node under the root with `node-addchild`. Examples of GML scripts using the operator are described in section 4.3.

`node-begin` and `node-end` These two operators are responsible for activating and deactivation a node. A call to `node-begin` sets the given node as active focus context. To issue GML commands within a context the desired context node has to be activated.

The node operator namespace is the foundation to dynamically create multiple GML models within a single GMLApplet instance. It is also possible to position the created models in space. For this reason each context has a data block attached that allows to set the translation, rotation and the scale factor of a context. This data is taken into account when the model is rendered by the runtime engine. The spatial data extension is described in section 4.2.2.2.

The dynamic control of model creation and management with the node operator namespace and the ability to add spatial data to a node forms a sound base for managing dynamic scenes of GML models. Examples that shows how to create multiple models with a GML script and how to change the spatial information are shown in section 4.3.

The second role of the Grimaldo resource is the integration of the node-tree structure into the runtime system of the GMLApplet. Grimaldo provides its own runtime system in form of *SceneViewer*. SceneViewer are responsible for rendering the node-tree as well as to delegate the input device data from the host-application to the nodes to enable interaction with the user. The Controller forms the frontend to the Grimaldo system, it is therefore also responsible for the runtime management. For this reason its interface provides four methods that are the entry points to the runtime:

```
void framePrepare();
void frameDraw();
void frameFinish();
void handle(InputEvent* event);
```

The interface is directly related to the corresponding methods of the GMLApplet to enable a straightforward connection between the Grimaldo Controller and the GMLApplet. To actually integrate the Controller the four corresponding methods of the GMLApplet are simply delegating their calls to the Controller pendants. However, the input parameter of `handle()` has to be converted to the Grimaldo input event format, described in section 5.2.3. The Controller - on the other hand - is not handling the calls itself. The Grimaldo framework is designed to handle all runtime specific functionality with SceneViewer. Therefore the Controller utilizes a SceneViewer for this purpose. The advantage of this approach is that multiple SceneViewer can be registered within one Controller and they can be switched at runtime. For the GMLApplet integration a custom SceneViewer is provided, the *GMLSceneViewer*. Its design and the integration into the GML framework are described in section 4.2.3.2.

4.2.2 The GMLContextNode

4.2.2.1 Interface

To integrate the GML context idea into the GML framework a concrete implementation of the GML context design given in section 4.1 into an actual node is needed. This implementation is realized with the *GMLContextNode*. The node is compatible with the Grimaldo framework and can be used with its node management as well as with the 3D runtime system of Grimaldo. To be compatible the node is derived from a Grimaldo base node. It is optimized for displaying 3D content and provides methods for rendering, determine a bounding box for culling or picking operations as well as an identification system that gives the node a name and an ID to be able to search for it in the Grimaldo node-tree.

The interface of the GMLContextNode is shown in listing 4.2. The `push()` and `pop()` methods are responsible for the focus context management. `push()` activates the GMLContextNode as focus context within the Grimaldo Controller, `pop()`, respectively, deactivates the node. Internally the Grimaldo Controller organizes the focus context node in a stack structure. The top element denotes the currently active context. The stack structure allows work with nested context structures. The most bottom element of the stack is always the root context. The root context is the first node that is

```
class GMLContextNode : public Node
{
public: ...

  virtual void update(ActionBase* action) {};
  virtual void redraw(ActionBase* action);
  virtual void finish (ActionBase* action) {};
  virtual bool handle(const GHOST::GHOST_Event &ev);

  virtual bool intersect (const Ray& ray, int near, far);

  void push();
  void pop();

  float getQuality() const;
  void setQuality(float value);

  bool call (const std::string& cmd);
}
```

Figure 4.2: GMLContextNode interface (truncated)

pushed onto the stack during the initialization phase of the GMLApplet. It is not possible to pop this context from the stack. This ensures that there is always one active context providing a valid resource set.

To efficiently pick a model from the node-tree the node interface provides an `intersect()` method. It performs an ray intersection test with the bounding box of the node. The input parameters are a ray and two distance values. The ray is defined by a starting point and a direction, the distance values define the length of the ray where the intersection test is evaluated. The used algorithm for the intersection test is described in [WBKS05].

The runtime interface is formed by the `redraw()` method, used to draw the model onto the screen and the `handle()` method that is responsible for integrating the encapsulated model into the event handling process. These two methods are tightly integrated with the GML event handler. Their implementation depends on changes that where made to the event handler to support contexts. Therefore the `redraw()` method is described in section 4.2.3.1, where the changes to the rendering part of the runtime system are summarized. There also the methods `getQuality()` and `setQuality()` are explained. The `handle()` method is also tightly integrated with the runtime system. For a better understanding of its functionality the implementation is explained on page 27. There the connection between the Grimaldo runtime system with the GML runtime is discussed.

4.2.2.2 Spatial Data

To make use of the GML context extension for representing and scripting complete 3D scenes it is necessary to add spatial data to a GMLContextNode. With this data a context can be positioned in space, which is the foundation to reasonable work with models in a 3D scene. The apparently simple extension of a GML context to carry spatial information impacts the working process of the GML runtime engine. The picking mechanism has to be adapted, which is described in section 4.2.3. A second effect that has to be considered is the view dependent rendering capability of the combined B-rep mesh data structure. These considerations are described in this section.

Transform Object To equip the context with a data structure for spatial data the *Transform* object is used. Its interface is shown in listing 4.3. This simple structure provides vectors for the translation, rotation and scale of the GML model. It practically forms the *model matrix* in terms of the OpenGL notation. To allow an easy handling of the 3D position the *getMatrix()* method returns a matrix that accumulates the three vectors into a single matrix. This matrix can then be used to setup the OpenGL model matrix. The `Mat4x4f` object returned by getMatrix() provides a `toOpenGL()` method that directly multiplies the position of the Transform object to the currently active OpenGL matrix.

```
class Transform
{
  public: ...

    void setTranslate(const Vec3f& vec);
    void setRotation (const Vec3f& vec);
    void setScale    (const Vec3f& vec);

    const Vec3f& getTranslate() const;
    const Vec3f& getRotation () const;
    const Vec3f& getScale    () const;

    /*! Updates transform matrix from translation, rotation and scale data. */
    void update();

  private:
    Mat4x4f m_matrix;
    Vec3f   m_translate;
    Vec3f   m_rotation;
    Vec3f   m_scale;
};
```

Figure 4.3: Transformation object of a GMLContextNode

The transform object is added to the GMLContextNode as a Property with the name "transform" (see section sec:Properties for an explanation of a Property). Via the synchronization feature of the Property it is possible for applications to subscribe to the Property and get notified when the transformation is changed. A transformation change can either be done in directly changing the object and emitting the "onObjectChanged" signal afterwards or via GML operators. The GMLResource_Grimaldo introduced in section 4.2.1 provides the following operators:

- node-transform
- node-rotate
- node-scale
- node-gettransform
- node-getrotate
- node-getscale

All operators take a GML context token as input. The first three operators additionally have a three dimensional vector as input. The rotation vector encodes the rotation of each axis in the corresponding vector component in degrees. So, for instance, the GML program

```
:node (2,0,0)  node−translate
:node (0,45,0) node−rotate
```

translates the GML model stored in the `node` token 2 units in the direction of the x-axis an rotates it 45° around the y-axis.

Combined B-rep Considerations The main data structure for visualizing GML models is the *combined B-rep* mesh. The mesh can be composed of both free-form and polygonal parts. The free-form parts - or *smooth* faces - are displayed as *Catmull/Clark* subdivision surfaces For efficient rendering the mesh utilizes an *adaptive level-of-detail (LOD)* algorithm to change the resolution of free-form parts dependent on the viewing angle the model is looked at. In each frame the level-of-detail is computed to balance the display quality and the rendering performance. Additionally a culling of non-visible faces is performed. [Hav05b]

The LOD computation takes into account a *view cone* that relates to the viewing frustum and a *quality* parameter that is obtained based on the render time of the last frame. The view cone and the quality parameter are examined in the `determineDepth()` method of the combined B-rep Applet. With the possibility to influence the spatial position of a GML model introduced in section 4.2.2.2 the modelview matrix of the model has to be set up before the render process. The computation in `determineDepth()` is relying on the model correctly positioned within the viewing frustum. The GMLApplet is taken care of this setup automatically. However, when rendering a model manually this step has to be included for a correct adaptive tessellation of the mesh.

4.2.2.3 Adding Resources

From the GML side-of-view the GMLContextNode is a container for a set of resources that are making up a GML model. At every time the Grimaldo Controller has one active GMLContextNode that forms the focus context described on page 13. Every operator that is executed works on the resource set provided by the focus context node. To equip a GMLContextNode with resources the node has to be integrated into the resource management system of the GML. The theoretical background on this integration is given in 4.1.2, the actual implementation is content of this section.

The Grimaldo Controller is responsible for adding new contexts in form of GMLContextNode objects to the system and provides a management interface with the necessary methods for this purpose. The responsible method for creating a new node is

```
Controller :: createNode(const std::string& type)
```

The method can be either called directly from the C++-application, however, the more common way is to call it indirectly in using the `node-create` operator in a GML script:

```
/gmlcontext node−create
```

The Controller supports the creation of various node types, which is denoted by the *type* argument. In this case the type of the node is “gmlcontext”, which creates a new GMLContextNode. The operator expects the type of the node as name token.

The `createNode()` method uses the *Initializer*-system of the GMLApplet (see 3.2.1) to equip a context with an independent resource set. After a GMLApplet is created the desired initializer set is added to it and stored in a list. The idea now is to use the initializer system as *factory* for resource sets. If a context node is created the stored initializer objects are executed and the resulting resource set is attached to the context node.

Local and Global Resources To optimize the memory usage of a context node the resources are separated into two different types: *local* and *global* resources. A global resource is not directly associated with a GML model but encapsulates functionality that affects the global state of the system. For instance the *Camera* resource that holds the viewing parameters or the *Navigator* resource responsible for the navigation in the scene are global resources. A local resource, on the other hand, stores data that is directly associated with the visual appearance and the functionality of a GML model. In contrast to a global resource this data can not be shared with other models. The most obvious local resource is the BRepProgressive resource. It forms the mesh representation of the specific model. To allow multiple contexts to have a different geometric shape it is not possible to share this resource between contexts, it has to be a local resource directly associated with the context corresponding to the model.

The separation into two resource types has two advantages:

Memory Efficiency When a new context node is created it only has to attach local resources to it. As a global resource is not model-specific it can be stored in a central pool that is accessible by all contexts.

Relaxation of the “All resources have to be of type GMLResourceWrapper” statement As described on page 15 the central resource management system of the interpreter is not going to be changed, the GMLResourceWrapper is doing the context-sensitive work. If the GML is extended in directly implementing the GMLResource or the GMLResourceApplet interface and not in using the GMLResourceWrapper mechanism there is no possibility to decide in which context a resource resides, as this functionality is a special feature of the GMLResourceWrapper. However, if a resource is a global resource it is not necessary to assign it to a context if it is requested, as all contexts share the same instance of this resource. This fact allows to relax the requirement that all resources used with the

```
 1 bool GMLAppletInitStandard::initResources(GMLInterpreter& inter)
 2 {
 3   // ...
 4
 5   r_camera = new GMLResource_Camera;
 6   r_handler = new GMLResource_EventHandler;
 7   r_pcbrep  = new GMLResource_BRepProgressive;
 8
 9   r_pcbrep→makeLocal(inter);
10
11   inter .addResource(r_camera);
12   inter .addResource(r_handler);
13   inter .addResource(r_pcbrep);
14
15   // ...
16
17   r_handler→addResourceApplet(r_camera);
18   r_handler→addResourceApplet(r_pcbrep);
19
20   // ...
21 }
```

Figure 4.4: Excerpt from the GMLAppletInitStandard object

GML context system have to be derived from a GMLResourceWrapper to "All *local* resources have to be derived from a GMLResourceWrapper".

The central pool for storing global resources is brought "for free" by the GML interpreter. Global resources are simply using the existing infrastructure of the interpreter in utilizing the default resource management interface with the `addResource()` and `getResource()` method.

The GML context extension is using the Initializer-system to equip a new context with a set of resources, or, to be precise, *local* resources, as the global resources are simply accessed with the default interpreter system. Hence, if multiple contexts are created the initializer objects are executed repeatedly. To allow an repeated execution without side-effects the interfaces involved into the initializer process do have to be adapted. Moreover, the resource system has to be extended with a possibility to mark a GMLResourceWrapper as a local resource.

Context-Aware Initializer System As described before, a new context node is created with the `createNode()` method of the Grimaldo controller. The following three lines show the relevant code that creates a new node with a set of resources:

```
1 GMLContextNode *node = new GMLContextNode();
2 pushNode(node);
3 getGMLApplet()→createResourceSet();
4 popNode();
```

In line 1 the node is created and then set active in line 2 in pushing it onto the focus context stack. In line 3 the actual work is done. The `createResourceSet()` method instructs the GMLApplet to execute all the initializer objects in its internal list. Listing 4.4 shows an excerpt of the default initializer delivered with the GML system, the GMLAppletInitStandard. It is truncated for demonstration purposes, only the relevant sections are shown.

In this demonstration case three resources are created in the lines 5 to 7. In line 9 the new `makeLocal()` method of the GMLResourceWrapper interface is introduced. The method has the task to mark a resource as local and to attach it to the currently active context. To attach the resource the *Property* system of the Grimaldo framework is used. A Property allows to encapsulate an arbitrary C++-object and attach it to the node. The Property is added a name and an integer ID so that it can be requested later on. An in depth description of the Property system is given in section 5.2.1.1. In this case the BRepProgressive resource is added as Property to the node. The Property gets the same name as the resource.

Lines 11 to 13 add the resources to the GML interpreter. Here two cases have to be distinguished. The first case is when the GMLApplet and the root context are created. The resources processes the default initialization within the interpreter. The resources are initialized (regardlessly if they are local or global) to give them the chance to register custom GML types with the interpreter. Moreover, their GML operators are initialized and added to the system. After the initialization the GML interpreter's type system and the operators are fully functional. Additionally, the root context is equipped with a set of Properties holding all local resources.

The second case is the creation of additional contexts after the root context. The `makeLocal()` method has the same behavior, it simply adds the local resource as Property to the node. The `addResource()` method of the interpreter, however, is acting differently. The method checks if a resource with the same name is already present. If it is, the resource gets added to an internal pool and the method returns. If the resource was not created before it is initialized. However, if no GML initializer object was added since the creation of the root node this case is not happening. After the initializer is finished the aforementioned internal pool is examined and all newly added global resources are deleted, as they are not needed anymore. They were already initialized and added to the interpreter when creating the root node. The local resources are not deleted, as they were added as Property to the new node with the `makeLocal()` method.

Lines 17 and 18 are adding the Applet resources to the event handler. Again, the two cases of creating the root context and creating an additional context have to be considered. In the first case the `addResourceApplet()` method is adapted to add the Applet resource as Property to the currently active context node. In this case not each resource is added as single Property, but the node has a single Property named *ResourceApplets* that holds a list the resource pointer is added to. This data representation is better suited for the event handling of show-ticket callbacks shown in section 4.2.4. In case of the root node both, local and global Applet resources, are added to the Property.

In the second case, when an additional context is created, the `addResourceApplet()` method checks if the resource is local. In this case it is simply added to the *ResourceApplets* property. If the resource is global it is skipped and deleted afterwards, it already is present in the root node's *ResourceApplets* Property.

After all initializers are finished within the `createResourceSet()` method of the GMLApplet the context is ready to use. It is equipped with one Property for each local resource as well as with a *ResourceApplets* Property that is holding the pointers to all Applets, which are needed later on for the event handling. Additionally the root context does also contain the global Applets. The GML interpreter is holding all global resources. Its type system is initialized and the operators are registered.

Context-Aware Resource Delivery The GMLResourceWrapper is responsible for delivering the objects encapsulated within a resource. Compared to the registration of resources this is a very straightforward task. Only a minor change is necessary to make the GMLResourceWrapper context-aware. This change takes place in its `get()` method. If it is called the method first checks if the resource is a global or a local resource. In case of a local resource the currently active context node is retrieved from the Grimaldo Controller. The node is then searched for a Property with the same name. If the Property is found the actual resource is extracted and the object it encapsulates is returned by the `get()` method. This straightforward process is sufficient to make the GMLResourceWrapper context-aware.

In case the resource issuing the `get()` call is a system resource simply the encapsulated object is returned directly, which yields the same behavior as before the context extension.

4.2.3 Runtime Changes

The GML framework has an integrated 3D runtime engine that takes care of rendering the registered Applets as well as of object queries. The next paragraph shortly outlines how the runtime engine is working. The description is followed by an explanation which parts of the runtime engine do have to be adapted to the GML context extension and how this is done.

The runtime engine is formed by the GML event handler. The runtime process is called once in every frame and consists of the triplet methods `framePrepare()`, `frameDraw()` and `frameFinish()`. The combination of the three allows for a flexible handling of pre-render calculations, the actual drawing and a step for processing post-render tasks. The methods are directly related to the Applet interface. An Applet contains equally named methods which are called when the event handler pendants are executed. For instance, the `framePrepare()` method of the combined B-rep Applet is taking care of computing the view-dependent level-of-detail settings for the mesh data structure as well as the culling of invisible faces. `frameDraw()` renders the faces of the mesh onto the screen. Finally, `frameFinish()` takes the frame duration and uses this value as base for the level-of-detail recalculation for the next frame.

Besides the preparation functionality for render specifics the `framePrepare()` method is responsible for the handling of *pending events*. The pending event mechanism is used to delegate input events originating from a mouse drag to an interested callback (for a description of the callback system see 3.4). When the user clicks onto the 3D viewport the event handler's `handleEventMouse()` is called. Within this method `findSelect()` determines if the click hits an Applet. If this is the case and the Applet has a callback attached the token of the callback is stored as a *focus ticket*. If the mouse click is followed by an immediate mouse drag event, this event is delivered to the callback stored in the focus ticket and can be processed.

As a third task the runtime cycle is managing temporary Applets. As described before they can be dynamically registered and unregistered. If a temporary Applet is registered it is rendered, can be picked an receives input events.

A summary of the GML event handler shows that the runtime is responsible for

- rendering the registered Applets
- processing of pending events
- processing of temporary Applets.

The schematic runtime sequence looks as follows:

framePrepare() At first a pending event is processed, if one exists. Afterwards the `framePrepare()` method of the Applets registered with the event handler as well as of the registered temporary Applets is called.

frameDraw() Here the `frameDraw()` method of registered Applets and temporary Applets is called.

frameFinish() This method is used to perform level-of-detail calculation for the combined B-rep mesh. Based on the time the frame took to render a quality parameter is calculated that is taken into account for the next frame.

The above list only gives a schematic description of the functionality of the event handler. Internally the event handler is also responsible for the handling of different render features like *anti-aliasing*, *ambient occlusion* or the creation of screen shots. These features are not described here. However, it is important that they are also supported when using contexts. Moreover, future extensions to the event handler should also be supported. To achieve this goal it is necessary to directly integrate the GML context extension into the workflow of the event handler. The integration is performed in two steps, which are described in the following sections:

1. Adaption of the current GML event handler to support the GML context extension.
2. Introduction of a new entity - the *GMLSceneViewer* - to coordinate the processing of GML contexts in combination with the existing event handler.

4.2.3.1 Adjusting the Rendering Interface

The event handler is holding a list of registered Applets on which it is performing its work. The list is stored in the vector `v_applets` holding pointers to GMLResourceApplets which contain the actual Applets. The vector is filled in the initialization phase of the GMLApplet when the initializer system is used to register the Applets with the event handler in calling its `addResourceApplets()` method. Section 4.2.2.3 describes how the Applets are added to a GML context. Each context has a Property that contains a list of its Applets. This Property is now used to make the event handler context-aware. The technique is the same as used before to adapt the resource management system to the context extension. The `v_applets` vector is replaced with the dynamic method `getResourceApplets()`. Within these method the currently active focus context is asked for its *ResourceApplet* Property and the contained list is returned. Each request to the `v_applet` variable is replaced with the `getResourceApplets()` method. With this modification it is now possible to “plug in” a specific GMLContextNode into the GML event handler in activating it as as focus context. The event handler then works on the Applets provided by the focus context.

To render all nodes onto the screen each node is activated once and the render methods of the event handler are called. This way the content of the node-tree can be rendered. However, the current structure of the render methods is not suited to call them more than once within a single frame. For each GMLContextNode in the node tree the event handler would execute a pending event and process the registered temporary Applets, as the functionality is directly integrated into the event handler's rendering methods.

```
class GMLSceneViewer : public SceneViewer
{
  virtual void framePrepare();
  virtual void frameDraw();
  virtual void frameFinish();

  virtual bool handle(GHOST::GHOST_Event& ev);

  virtual GHOST::Node::GMLNode* findSelect(int x, int y,
                                           Meshlib::GMLEventIO& data);
}
```

Figure 4.5: GMLSceneViewer interface (truncated)

To remedy the effect of multiple processing pending events and temporary Applets within a single frame the corresponding runtime methods are changed. The pending events functionality as well as the handling of temporary Applets is disabled. To make this functionality compatible with GML contexts a new entity is introduced, the *GMLSceneViewer*.

4.2.3.2 GMLSceneViewer

Rendering To render the nodes onto the screen and to ensure a correct event handling it is necessary to adapt the runtime system of the GML to take into account the existence of multiple nodes. For this reason the infrastructure of the Grimaldo runtime engine is used. The runtime is provided by the *SceneViewer*. It is responsible for rendering the content of the node-tree and to deliver user input events to the nodes. The viewer is directly integrated into the Grimaldo Controller via the *Strategy* pattern. The technique allows to change the viewer type at runtime. More information on the intention of the SceneViewer concept and its base interface is given in section 5.2.3.

The implementation of the SceneViewer is highly application specific. Therefore, to integrate it into the GMLApplet a new viewer type is created, the GMLSceneViewer. Its definition is shown in listing 4.5. The GMLSceneViewer is responsible for combining the handling of pending events and temporary Applets with the rendering of the GMLContextNodes. In the `framePrepare()` method the GML event handler is checked for an registered pending event. In this case the callback item stored as focus ticket is executed. The input event data of the last frame is made available to the callback and can be used during the execution. After the handling of the focus ticket the global resources of the root context node are prepared. Then all registered temporary Applets are prepared, too.

In the `frameDraw()` method the rendering is done. At first the view matrix of the GML camera is pushed onto the OpenGL modelview stack to setup the camera view. The system is now prepared to perform the actual rendering of the node-tree. For this the *action* technique provided by the Grimaldo framework is used. An action is a functional unit that performs a special task on all nodes of the node-tree. The system is described in more detail in section 5.2.2.1. To render the content of the node-tree the *GMLDrawAction* is used. The action is specialized to render GMLContextNodes and takes into account the spatial data information of the nodes. Listing 4.7 shows the implementation of the actions's `apply()` method. Each node of the node-tree is sequentially handed over to this method and is then processed. At first the given node has to be casted to the GMLContextNode type. The action interface is solely working with the Grimaldo base classes to provide a general processing mechanism for all kind of nodes. With the actual GMLContextNode the current quality parameter of the node is set into the event handler. Afterwards the spatial data is requested and multiplied onto the OpenGL modelview stack. With the correctly set up OpenGL environment the actual rendering method of the node - `redraw()` - is called. Listing 4.6 shows the implementation. At first the node is activated as focus context to “plug” it into the GML event handler for further processing. Then the event handlers `framePrepare()` method is called. As the quality parameter of the event handler and the OpenGL modelview matrix were set up before view-dependent calculations for the combined B-rep mesh data structure are working correctly. In line 7 the `saveModelViewMatrix()` method stores the current OpenGL modelview matrix within the GMLContextNode. This matrix is used later on for the GML picking mechanism, which is described in section 4.2.3.2. The mechanism uses the OpenGL selection buffer method to determine if a visual object was hit. For this method to work the model has to be rendered to a selection buffer. Therefore, the modelview matrix of the node has to be correctly set up to perform the picking. It is of course possible to set up the modelview matrix with the combination of the spatial data of the GMLContextNode and the view settings from the GML camera. However, if the GML camera is not synchronized with the actual camera of the rendering environment, like it is

```
void GMLContextNode::redraw(Action∗ action)
{
  push();

  m_gmlApplet→r_handler→framePrepare();
  saveModelViewMatrix();
  m_gmlApplet→r_handler→frameDraw();

  pop();
}
```

Figure 4.6: The `redraw()` method of the GMLContextNode

```
void GMLDrawAction::apply(Node∗ base)
{
  GMLContextNode ∗node = dynamic_cast<GMLContextNode∗>(base);

  // prepare the quality setting of the event handler:
  r_handler→quality = node→getQuality();

  // render the node:
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();

  node→getTransform().getMatrix().multOpenGL();
  node→redraw(this);

  glPopMatrix();
}
```

Figure 4.7: The GMLDrawAction's `apply()` method for rendering a GMLContextNode

the case when using GML models with the OpenSG scene graph described in chapter 6, the GML camera data is not valid. The OpenSG camera data has to be taken instead. The approach to save the modelview matrix directly on an OpenGL level is a universal and independent method that makes the picking method available in different viewing environments. To finally draw the model the event handler's frameDraw() method is called. This procedure is repeated for all nodes in the node-tree.

To finish the rendering process the the `frameFinish()` method is called. Within this method a second action - the *GMLFinishAction* - is send through the node-tree. Listing 4.8 shows its `apply()` method. The current quality parameter of the node is retrieved as well as the duration of the last frame time, which is accessible via the Grimaldo Controller. These two values are the input parameters for the `calculateQuality()` method. The method is increasing or decreasing the quality parameter based on the current value of rendered frames per second (FPS), which is calculated from the render time of the last frame. The `calculateQuality()` method defines a set of FPS values determining the quality value. The quality parameter is used for the next rendering cycle to adapt the level-of-detail setting of the combined B-rep mesh data structure.

Via the GMLFinishAction it is possible to dynamically adjust this value set based on the applications need. However, currently the values are statically integrated.

Object Picking The presence of multiple context nodes within a GMLApplet does change the way the object picking is working. In this section the OpenGL picking mechanism used by the GML is discussed. The mechanism is described in [WNDD99]. It is used to identify if an Applet denoted by a show-ticket is hit by a mouse click or drag.

The responsible method for object picking is the `findSelect()` method of the event handler. It is triggered by a mouse event and receives data of this event in form of the `GMLEventIO` object. The method is using the OpenGL selection

```
void GMLFinishAction::apply(GHOST::Node::Base::I_GHOST_NodeBase* base)
{
  GMLNode *node = dynamic_cast<GMLNode*>(base);

   float   quality    = node->getQuality();
  double frametime = getController()->getLastFrameTime();
  node->setQuality(calculateQuality(quality, frametime));
}
```

Figure 4.8: The GMLFinishAction's `apply()` method for recalculating the quality parameter based on the render time of the last frame

buffer to determine if a visual object was picked. Basically, a small viewport is set up around the position of the mouse click (typically a 5x5 point rectangular) and renders all Applets supporting the picking algorithm to this buffer. If objects are rendered onto the picked position the nearest object is calculated. This object then becomes the picked object.

With multiple models this mechanism becomes inefficient, especially with larger scenes. Each context has to be rendered whenever a mouse button is clicked or dragged. To enable an efficient picking mechanism the `findSelect()` method of the event handler is replaced by a corresponding method of the GMLSceneViewer. This method is pre-selecting a context before the actual OpenGL selection buffer algorithm is used to determine if an objects was hit. The selection is done with the bounding box intersection test provided by the GMLContextNode. When a mouse click or drag occurs a *GMLIntersectAction* is send through the node-tree that calls the `intersect()` method of the GMLContextNode to check if it is hit by the given picking ray. In case of a hit the action stores the node as hit candidate.

After the traversal of the tree the action contains a list of all nodes which bounding boxes were hit. The list is sorted by the distance of the hit points, so that the hit node nearest to the viewer is the first entry. The hit candidates are then sequentially processed with the picking mechanism of the GML, which is implemented within the `handle()` method of the GMLContextNode. The method utilizes the original `findSelect()` method of the GML event handler. As described before, `findSelect()` renders all Applets to determine a hit. Therefore, each node of the hit list is activated as focus context to plug the context into the event handler. Then the OpenGL modelview matrix saved during the rendering of the node is restored to correctly position the model before it is rendered to the selection buffer. The event handler's `findSelect()` method is called and determines if an Applet with an attached callback is hit. In a positive case the callback is set as the focus ticket and the GML event handling is processed as usual. If a show-ticket callback was hit the processing of the hit candidate list is stopped. If no object is hit the next node is processed until the list is finished.

The pre-selection of possible hit candidates by means of their bounding boxes is a fast way to accelerate the picking process when multiple contexts are involved. The call to the `findSelect()` method on the selected contexts ensures that the default event handling of the GML is working correctly.

4.2.4 Show-Ticket Callbacks

The show-ticket callback system of the GML event handler is described in section 3.4. With the introduction of GML contexts the system to register and execute the callbacks has to be adapted. The same consideration also apply to the type of temporary Applets which are registering callbacks. They are not explicitly covered here, but the principle is the same.

Section 3.4.1 describes the `io-callback` operator and how it is working. Before a callback can be attached to a token the event handler ensures that its type is a valid show-ticket in the `registerCallback()` method. The method cycles through the registered Applets and checks if the show-ticket is recognized by one of them. With the introduction of GML contexts the check for a valid show-ticket resource is done against the GMLResourceApplets registered with the currently active focus context. The *ResourceApplets* Property contains a list of these resources. The list is cycled through to check if the show-ticket is a known type. This process also explains why the GMLResourceApplets where stored into a list instead of attaching them directly as property to the node. It eases the handling of this special purpose. If the show-ticket is found valid the actual callback can be registered.

However, a problem arises when a user is picking the show-ticket object to trigger the execution of the callback. The execution is done during the runtime loop of the application. In this phase there is no possibility to ensure that the context the callback was registered in is active at the time the callback gets executed. If the callback directly refers to context data it is vital that the correct data is active, which means that the correct context has to be set as focus context. To be able to reactivate the context the callback was registered in it is necessary to adapt the data structure used for storing contexts, the *CallbackItem* object. It stores relevant data like the show-ticket, the callback itself and additional data the callback needs when it is executed. This object is extended with a new member variable, the `v_focusContext`, which is of type GMLContextNode. The `registerCallback()` method is adapted to take into account the new member. Whenever a callback is registered via this method the currently active focus context is saved within the CallbackItem.

The last step to finish the adaption of the callback handling is to change the places in the event handler where the actual execution of to the callback takes place in case a show-ticket object is picked. For show-ticket callbacks the method responsible for executing a callback is the `executeItem()` method. It is extended to activate the `v_focusContext` before the actual callback is executed. Afterwards the node is popped from the focus stack again. This method ensures that a callback is executed within the correct context.

4.3 Examples

The advantage of the context extension is that multiple models can be created within one GML script. They have access to each other, but their nodal representation makes it possible to treat one model as standalone unit. As such, data can simply be added to a specific unit, for instance, spatial data. Together with the hierarchical topology of the node-tree it is possible to render nodes within their own coordinate system. This allows to build up a model from subparts. The local coordinate system eases the positioning of the subparts relative to a given parent.

The GML operators to work with contexts are provided by the Grimaldo resource. They provide general access to the node management facilities of the framework. As a GML context is implemented as a node type of the framework it can be controlled via these operators. On page 17 the available operators are listed and described. This section shows examples on how to use the operators to create a virtual scene. The important code lines are explained. The summarized code is stored in the file GRIMALDOEXAMPLES.XGML.

4.3.1 First Example

The most basic operator to start with is `node-create`. It takes a name token as input argument. `node-create` internally calls the node creation method of the Grimaldo Controller, which needs to know which type of node the caller wants to create. To create a GML context the name “gmlcontext” is given as argument to the controller.

```
 1 node-clear usereg
 2
 3 /gmlcontext node-create !node
 4 :node node-begin
 5   deleteallmacros newmacro
 6    (0,0,0)  (0,0,1)  1 8  circle
 7    5 poly2doubleface
 8    [  (0.2,0.5,0)  (0.4,1,0)  ] extrude
 9 node-end
10
11 node-getroot :node node-addchild
```

The first line resets the scene in deleting all existing nodes with a call to `node-clear`. Then a new context node is created in line 3. To work with the newly created node it has to be activated as focus context. In line 4 `node-begin` performs the activation. From this point on the GML interpreter is working on the resource set provided by this node. Therefore, line 5 to 8 add an extruded circular geometry to the node. After the work on the node is finished `node-end` deactivates it and reactivates the last focus context, in this case the root context node. In line 11 the node is finally added to the root node to get displayed. The root node is accessible via the NODE-GETROOT operator. The newly created node is added with `node-addchild`.

To keep things organized it is good practice to structure parts of a scene into a dictionary. This dictionary represents the root node of a subgraph and is used to attached data to the graph the programming logic can work with. The same technique is also used by existing GML scripts to define a model and the corresponding data. This dictionary is commonly called *modeldict*. Analogue to this nomenclature the dictionary defining the root node of a scene subgraph is called *scenedict*. A context node can be used as a fake dictionary (see 5.3.2), which makes it possible to directly use the context node for this purpose.

```
 1 /gmlcontext node-create dup !node begin
 2   /scene    :node   def
 3 
 4   /radius           3 def
 5   /segments        8 def
 6   /origin     (0,0,0)  def
 7   /normal    (0,0,1)  def
 8 
 9   /scene-create {
10     scene node-begin
11       deleteallmacros newmacro
12       origin  normal radius segments circle
13       5 poly2doubleface
14       [  (0.2,0.5,0)   (0.4,1,0)  ]  extrude
15     node-end
16   }  def
17 
18   scene-create
19 end
20 
21 node-getroot :node node-addchild
```

The lines above are a minimal template to define a scenedict. In line 2 the context node itself is stored inside the dictionary. This allows to access the node within functions defined by the dictionary. Lines 4 to 7 define data responsible for creating the node's geometry. The creation is done in the `scene-create` function. Line 10 accesses the node via the `model` key and activates it. The `circle` operator now does not take hard-coded parameters to define its shape, it is using dictionary data. This approach makes it easy to change the nodes appearance in changing the dictionary data and call `scene-create` to update the geometry. The initial creation of the geometry is triggered in line 18. Finally the node has to be added to the root node.

To create a scene consisting of multiple nodes the scenedict template is extended to allow the definition of so called *gizmos*. Gizmos are a high-level GML library around the BallsAndSticks resource. They are clickable and draggable objects that allow to attach callbacks to a click or drag event. Their definition takes place in the `gizmos` function.

```
 1 /gmlcontext node-create dup !scene begin
 2   /model    :scene   def
 3   /active    :scene   def
 4   /phase           0    def
 5 
 6   /radius           3 def
 7   /segments        8 def
 8   /origin     (0,0,0)  def
 9   /normal    (0,0,1)  def
10 
11   /sliders   [  ]  def
12   /gizmos   {
13     model sliders  2
14     {  2  0  0 vector3  0  }
15     {  }  {  add-object }
16     GIZMOS.gizmo-trigger
17 
18     model node-begin
19       sliders  {
20         begin gizmo-create gizmo-activate end
21       }   forall
22     node-end
23  }  def
```

```
24
25  /add−object {
26     /gmlcontext node−create dup !node node−begin
27        deleteallmacros newmacro
28        origin  normal radius segments circle
29        5 poly2doubleface
30        [  (0.2,0.5,0)  (0.4,1,0)  ]  extrude
31     node−end
32
33    phase sin phase cos 1 vector3 !trans
34    :node :trans node−translate
35    active  :node node−addchild
36
37    /active  :node def
38    /phase phase 25 add def
39  }  def
40
41    /scene−create {
42       add−object
43    }  def
44
45    scene−create gizmos
46  end
47
48  node−getroot :scene node−addchild
```

Before the creation of the interaction gizmo is described the base scene is constructed. The creation of the geometry is now encapsulated into the function `add-object`. It creates the context node with the geometry. Then the newly created node is transformed depending on the `phase` dictionary entry, which is initially zero. The transformation is calculated in line 34 and set with the `node-translate` operator. The node is then added to the `active` node. The `active` dictionary entry denotes the last inserted node and is initially set to the scene node. Adding the new node to the active node shows the advantage of the node-tree structure. The transformation is relative to the parent node, which makes it easy to correctly position the geometry. In this case the node is simply added on a circular path depending on the `phase` with a z-coordinate of 1. This results in a spiral formation when multiple objects are added. The last step in the `add-object` function is to store the newly created node as the `active` entry and to increase the phase value, which takes place in line 35 and 36.

Now the interaction component can be added. It is used to add a new node to the scene when a user clicks onto the gizmo. To activate the gizmo (or gizmos in general) the scenedict template structure is extended with the `gizmos` function. The template now contains the `scene-create` function to generate the subgraph of the context node and the `gizmos` function to allow interaction with the user. The activation of the gizmo takes place in the lines 14 to 23. The important line is number 16. Here the callback for the click event is registered, which is the `add-object` function. Each time the gizmo is clicked a new object is created on top of the last one. An important thing to remember is that when registering gizmos within a node this node has to be the focus context. Therefore the scene node is activated in line 18 before the activation takes place.

Figure 4.9 shows the result of the code example after clicking on the gizmo various times.

4.3.2 Manipulator Tool

To be able to move nodes within a scene a *Manipulator* tool is provided within the GRIMALDODEMOS.XGML file. The manipulator is based on gizmos and allows to move an object around in all three dimensions by dragging the gizmo arrows. It can be created with a call to `GrimaldoDemos.Tools.create-manipulator`. The function takes three input arguments in this order: an offset vector, an constraint vector and a size. The offset is used to displace the manipulator from the attached object. This is necessary because in many cases the object would overlap the manipulator. The constraint vector defines the length in each direction the object can be dragged with a single drag. The values of the vector are used for both, positive and negative directions. The last parameter allows to adjust the size of the manipulator.

The following lines make a node movable by the manipulator tool:

```
/gmlcontext node−create dup !node node−begin
  deleteallmacros newmacro
   (0,0,0)  (0,1,0)  4 8  circle
  2 poly2doubleface
  [  (0.2,0.5,0)  (0.4,1,0)  ]  extrude
node−end

GrimaldoDemos.Tools.create−manipulator !manip
:manip /trans get  :node node−addchild

node−getroot :manip node−addchild
```

The node has to be added as child of the `trans` dictionary entry of the manipulator. This entry itself is a node that simply acts as transformation unit. Unfortunately, the internal structure of the manipulator does not allow to directly use `node-translate` and `node-gettranslate` directly. Using them results in unexpected behaviour. Therefore two functions are provided that take the task of setting and retrieving a translation from a manipulator: `GrimaldoDemos.Tools.manip-translate` and `GrimaldoDemos.Tools.manip-gettranslate`. These functions take into account the internal manipulator structure and correctly apply or get the translation. Currently the manipulator tool only supports the translational component, however, it can easily be extended.

Figure 4.9 shows the result of the above manipulator example. The manipulator is positioned below the object to have all arrows clickable.

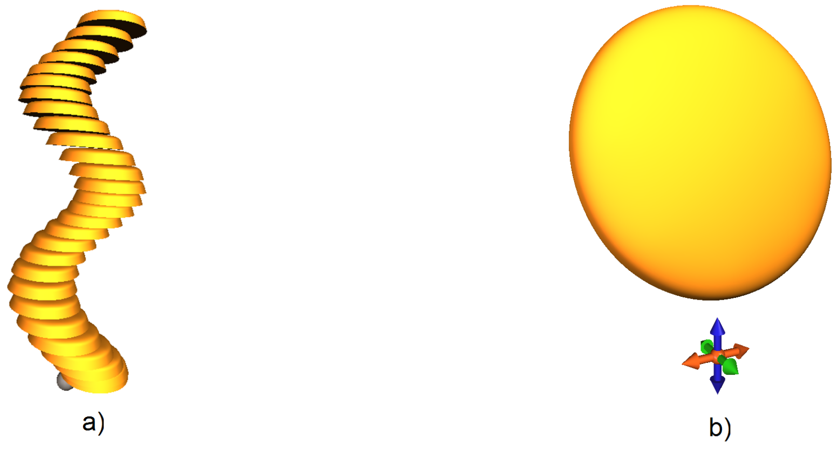


Figure 4.9: a) A spiral form generated by multiple nodes, b) the manipulator tool attached to a simple object

4.3.3 Gothic Wallbuilder

The *Gothic Wallbuilder* is a tool that allows to arrange gothic windows into a wall. The example is very basic. It combines the techniques and tools introduced in the last section in showing how to use the manipulator and a gizmo to interact with the scene.

The code structure of this example is again using the scenedict template from the previous example, the `create-scene` and the `gizmos` functions are reimplemented.

```
 1 /create-scene {
 2     GrimaldoDemos.Tools begin
 3
 4     /gmlcontext node-create !node
 5     (4,2,-2) (12,10,16) 10 create-manipulator !manip
 6     model :manip node-addchild
 7
 8     create-window !window
 9     :manip /trans get :window node-addchild
10
11     model node-begin
12       deleteallmacros newmacro
13
14       /std9hi setcurrentmaterial
15
16       (0,0,0) (0,0,1) 90 4 circle
17       1 poly2doubleface
18       [ (0.2,1,0)
19         (0.2,0.5,0) ] extrude
20       pop pop
21     node-end
22
23     end
24 } def
```

At first a manipulator is created and configured in line 5. It is then added to the model, which is the tool itself. The manipulator is used to move a window that is created with the `GrimaldoDemos.Tools.create-window` function:

```
1 usereg
2
3 /gmlcontext node-create dup !node node-begin
4   Gothic-Window.Tools
5   .demo-gothic-window-slider pop
6 node-end
```

Within the function a new context node is created and activated. Then the actual generation of the window takes place in using an existing GML program for creating gothic windows. As the node is currently the active focus context the geometry is created within the node. In line 9 the manipulator takes control over the window and makes it moveable.

Lines 11 to 21 create a ground plane the window is positioned on.

```
 1 /gizmos {
 2     model sliders 5 {
 3     (0,-6,0) 0
 4     } { } { create-scene }
 5     GIZMOS.gizmo-trigger
 6
 7     model node-begin
 8       sliders {
 9       begin
10         gizmo-create
11         gizmo-activate
12       end
13       } forall
14     node-end
15 } def
```

To create new windows a gizmo is placed within the scene. Its only purpose is to call the `create-scene` function in case of a click. In this case the function can simply be called to create a new window, no other actions have to be taken.

The constraint parameter of the manipulator takes into account the dimensions of the window. If an arrow is dragged to its end in either direction the window aligns with the next created windows. This allows to build a complete wall in a fast way. Figure 4.10 shows the creation process. The starting point is a single window. In adding more windows and dragging them to the desired location the wall is created. It is possible to change the appearance of a single window. The style can be changed with the trigger gizmo, the size with the arrow sliders.

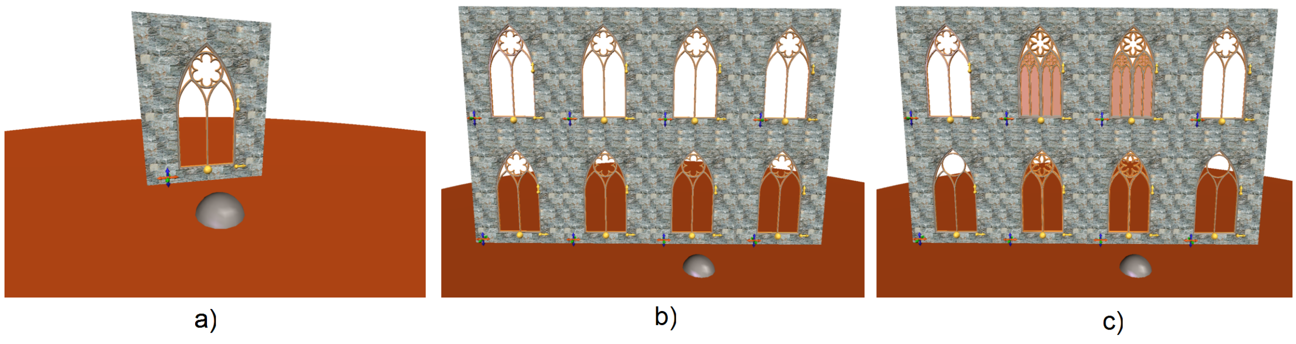


Figure 4.10: a) Initial scene with one window, b) a completed wall, c) windows are configured with different styles

4.3.4 Parametric Creation of Models

The GML is perfectly suited to create a large number of similar models starting from a parametrized base model. The example combines this advantage with the new scene creation capabilities of the framework. The discussed tool consists of two parts. The first part is the *shelf* where a base model is placed. The shape of this model is parametrized. Each free parameter can be adjusted in using gizmos to create the desired shape. The second part is the *desk* where the adjusted model can be placed once it is finished. The desk provides place for a number of models which can then be presented and compared. Figure 4.11 summarizes the tool. The left picture shows the two main parts, the initially empty table on the left, the shelf with the base model on the right. The middle picture shows the editing of the base model. The parameters can be adjusted until the model fits the desired shape. With the trigger gizmo above the model it can be put onto the desk. The right picture finally shows the filled desk with a number of similar parametrized models. With another trigger gizmo the desk is cleared to start the procedure again.

In this example only the relevant code parts are described, the additional code can be found in the GRIMALDODEMOS.XGML file. Again, the scene is encapsulated within a scenedict to keep things structured. The following lines show the code to construct the different parts of the scene.

```
 1 /create−scene {
 2     GrimaldoDemos.Tools begin
 3 
 4     8 create−desk !desk
 5     model :desk node−addchild
 6 
 7     create−shelf !shelf
 8     :shelf (−5,0,4) set−translate
 9     model :shelf node−addchild
10 
11     :shelf create−basemodel
12 
13     model /shelf :shelf put
14     :shelf /desk :desk put
15 
16     end
17 } def
```

In line 3 the `GrimaldoDemos.Tools.create-desk` function creates the desk. It takes a number as input parameter that specifies how many objects can be placed on the desk. The function internally creates a new context node with the geometry shown in figure 4.11. It also computes the number of points given to the function equally distributed on a circle. The context node provides this data in a `positions` dictionary entry. These positions are used later on to place a model on the desk. Line 5 adds the desk node to the scene. The next step is to create the shelf where the base model is presented. This is done with `GrimaldoDemos.Tools.create-shelf`. The shelf is formed by a single context node with the circular shape. The function returns the node which is then positioned and added as child of the desk.

After the shelf the base model is created. The responsible function is `GrimaldoDemos.Tools.create-basemodel`, where a new scenedict is created. As input parameter the desk and the shelf are taken. The function is self contained, which means that it does not return the created base model context node but adds it to the desk by itself. At first the model is created:

```
 1 /create-scene {
 2     model node-begin
 3       (0.2,-0.04,0)
 4       (0.27,0.44,0)
 5       (0.16,0,0.43)
 6       (0.23,0.4,0.45)
 7       (0.2,-0.1,1.04)
 8       dict
 9       Stuhl.Tools.create-Chair-2 /chairdict edef
10       chairdict  Stuhl.Tools.Chair-1-makesliders
11     node-end
12   } def
```

The model is taken from the existing GML script library. It takes a set of parameters and a dictionary in the lines 3 to 8 and creates a chair base on the given values with the `Stuhl.Tools.create-Chair-2` function. The return value is a modeldict that contains the model and the parameter set. This model is stored within the scenedict with the key `chairdict`. This dictionary will be used later on to retrieve the parameter set of the base model to put an instance of the model onto the desk. In line 10 the gizmos of the chair are created to make the model alterable by the user.

The base model is now prepared and can be changed by the user. When a parameter is changed with a gizmo the corresponding value of the modeldict is updated. To put an altered model onto the desk a trigger gizmo is placed above the model. On a click a new model is created based on the the changed parameter set and is put onto the desk.

```
usereg

/gmlcontext node-create dup !node node-begin
  chairdict  /fhR get
  chairdict  /fvR get
  chairdict  /shR get
  chairdict  /svR get
  chairdict  /lR   get
  dict
  Stuhl.Tools.create-Chair-2 pop
node-end

:node desk GrimaldoDemos.Tools.put-on-desk
```

The above listing shows only the click callback of the trigger gizmo. A new context node is created and the current parameter set is read from the `chairdict`. These values are taken as input for the chair creation function. This time no sliders are appended to the created chair model. The new context node is then put onto the table with the `GrimaldoDemos.Tools.put-on-desk` function, which takes the context node of the model and the desk as input parameter.

```
usereg !desk !node

:desk :node node-addchild

:desk /curindex get  !idx
:desk /positions  get  :idx  get  !pos

```

```
:node :pos node—translate

:desk /curindex get 1 add !idx
:desk /curindex :idx put
```

As described before the desk provides an `positions` entry. This entry contains an array of locations on the desk where models can be put. It would of course be possible to let the `put-on-desk` function decide, where to actually put the model, however, this design makes it easy to adjust the appearance of the models on the desk, if necessary. At first the node is added as child of the desk. Then the desk is asked for the next free position slot, which is available via its `curindex` dictionary entry. `curindex` is the index of the next free position in the `positions` array. This position is then used to translate the node accordingly within the local coordinate system of the desk. Afterwards `curindex` is incremented to point to the next slot. For simplicity sake there is no error check if the index is greater then the size of the positions array. If more models are added to the desk then free slots are available the model overlaps the model that is already placed there.

To give the user the possibility to clear the desk, the desk's scenedict positions a trigger gizmo above the desk with the following click callback:

```
model node—removechildren
```

`model` points to the desk context node and serves as input parameter for the `node-removechildren` operator. The operator simply deletes all the children of the given node. In this case all models on the desk are removed and the slots are free to be filled again.

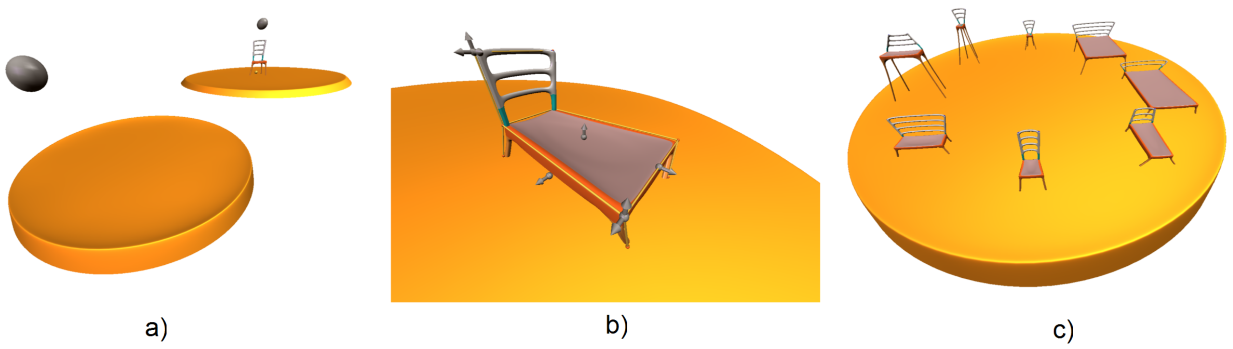


Figure 4.11: a) Scene setup of the tool, b) altering the base model, c) an arrangement of similar parametrized models

4.3.5 Arrangements

In this example the tool in the last section for creating a variety of similar models is modified to concentrate on the generative aspect for creating arrangements of models. Figure 4.12 shows the setup of the tool, which is similar to the previous one. The main parts are the desk with a number of manipulators on it. The second part is the shelf with three different parameterized models. The blue planes on the right side of the scene are *storage shelves*, which are explained in a second.

The code to create the desk and the shelf is taken from the previous example. It is slightly modified in two areas. The shelf now contains three models, which are created by an existing GML script of the GML script library (see CASSIUS-LAMP.XGML). The `create-desk` function is extended to put manipulators onto the desk. As described before the desk computes locations where objects can be put on, these are now used as origin points for the manipulators. The manipulator is restricted to only allow movement within the XY-plane.

The idea of the tool is to arrange the manipulators on the desk. A click on the gizmo above one model on the shelf creates one new context with the model's geometry for each manipulator and places it at the manipulators location. On the desk the models can be fine-tuned in either adjusting their positions or in changing the model itself with the provided gizmos. When the arrangement is finished a click on the gizmo next to the desk moves the models onto the next free storage shelf.

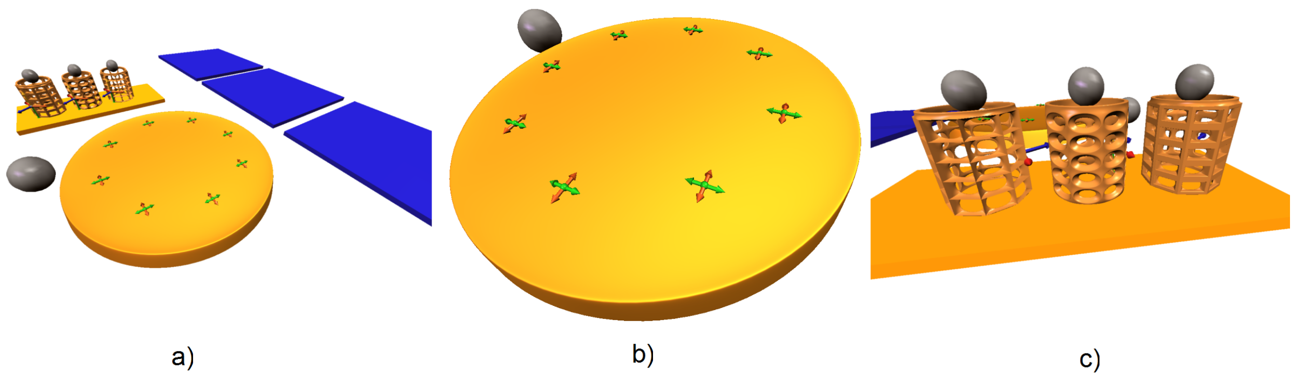


Figure 4.12: Overview of the Arrangements tool

This way multiple arrangements can be made and be compared. Figure 4.13 shows the necessary steps to fill three storage shelves with models.

For the implementation the previous desk scenedict is extended with a `place-manipulators` function, which is called at the creation of the desk.

```
/place−manipulators {
    positions
    {
    !pos

    (0,0,0)  (10,10,10) 4 create−manip−XY !manip
    :manip :pos 1 putZ manip−set−translate
    model :manip node−addchild

    manipulators :manip append

    }  forall
} def
```

The function iterates over the `positions` array of the desk and places a manipulator on each entry. The manipulator is an adapted version that only allows movement in the X- and Y-direction. It is created by the `GrimaldoDemos.Tools.create-manip-XY`. The implementation only differs from the original `create-manipulator` function in removing the gizmos for the Z-direction. Each created manipulator is stored in the `manipulators` array of the scenedict. A manipulator is then used to append the desired model from the shelf.

The second added function is `store`. It is called when the trigger gizmo next to the desk is clicked.

```
/store {
    GrimaldoDemos.Tools begin
    manipulators
    {
    !manip

    :manip /trans get 0 node−getchild !node
    :node :manip manip−get−translate 0.2 putZ set−translate

    model /storage get storeidx  get  :node node−addchild
    }  forall

    model /storeidx  storeidx  1 add put
    end
} def
```

The function iterates over the manipulators array and extracts the model context node that is located as child of the trans dictionary entry of the manipulator. To retrieve the child the node-getchild operator is used in line 8. It takes a parent node and an index as input parameter. The index specifies the location of the desired node within the children list of the parent. In this example only one child is added to the trans node, therefore index 0 references the model context node. In line 9 the current position of the manipulator is read with a call to manip-get-translate. This position is then set to the model context node (with a manual adjustment to the Z coordinate to avoid an intrusion of the model with the storage shelf). The desk scenedict now also contains a storage and a storeidx dictionary entry, which are filled during the creation of the storage shelves (the necessary code is not explicitly explained here). Storage shelves are simple context nodes with a blue plane geometry as shown in figure 4.12. The current free storage shelf denoted by the storeidx entry is retrieved in line 11 and the model is added as child of the shelf. In this case node-addchild takes care of removing the model context node from its previous parent node. This procedure is repeated for all manipulators on the desk. Finally, in line 14, the storeidx index is incremented to point to the next free shelf.

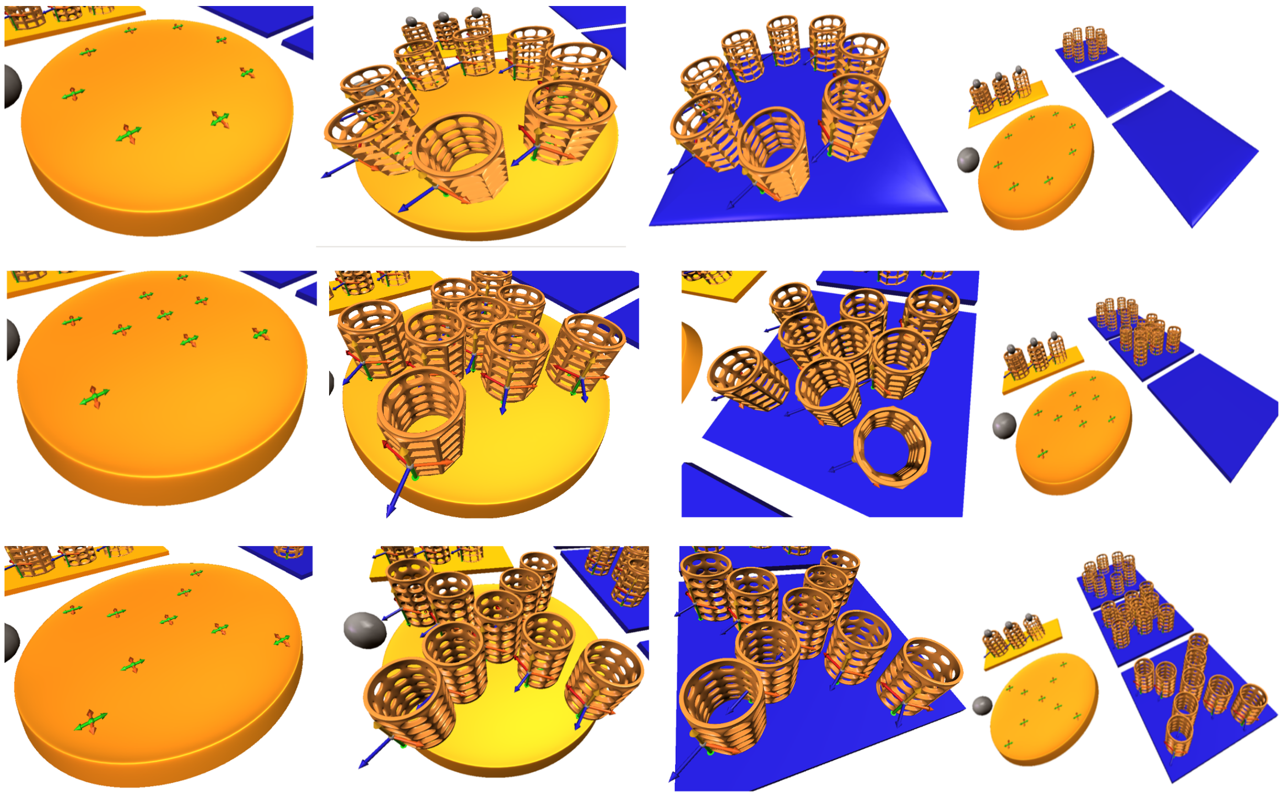


Figure 4.13: Individual steps for creating three different arrangements

5 The Grimaldo Framework

5.1 Overview

The Grimaldo framework [1] provides a software development kit (SDK) that is targeted to support the development of general visual applications. It provides components and features that originally came into existence as necessity to implement the GML context idea described in section 4. The bits and pieces where put into a consistent unit with the goal to provide (a) an efficient toolset to integrate the contexts into the GML and (b) a reusable SDK for visual applications.

The design offers a compact but flexible set of standalone components that provide:

- a hierarchical node-tree structure for managing multiple objects
- a runtime system that controls the rendering and the handling of input-device data
- a flexible data management system with notification capabilities for data changes
- an *event system* for informing host-applications about internal state changes
- control over the GML shading system.

Grimaldo is tightly coupled with the Generative Modeling Language (GML). Their connection yields an enrichment for both libraries:

- The GML is extended via the Grimaldo framework to support the management of multiple GML models within the same interpreter space.
- The Grimaldo framework on the other hand uses the powerful scripting facility of the GML to offer dynamic control over its components from within a GML script.

The combination of Grimaldo and the GML provides a flexible infrastructure to integrate the GML into a custom host-application. The host-application profits from:

- the strength of the GML as a generative modeling and geometry creation engine
- the powerful scripting environment of the GML
- the features of the Grimaldo SDK for data synchronization between the GML and the host-application.

In using the framework the GML features are delivered "for free", as they are directly integrated into Grimaldo. However, the SDK can also be used without the GML integration as standalone package.

The integration of Grimaldo into the GML was targeted to be as "minimal invasive" as possible. The current code base of the GML should be changed as few as possible to not "disturb" the traditional GML infrastructure but to provide an additional feature in a homogeneous way. In combination with the GML the Grimaldo framework provides a flexible software development kit to support the development of visual applications with dynamic scripting facilities and a powerful shape generation engine, provided by the GML.

[1]The Grimaldo framework is also called "Grimaldo SDK", "framework", "SDK" or simply "Grimaldo" throughout the document. The terms are interchangeable.

5.1.1 Framework Outline - The Big Picture

The composition of the different Grimaldo components is targeted to control major parts of a visual application, like

- (parametric) geometry creation,
- 3D scene layout,
- render engine control,
- input device handling,
- data exchange with host-applications.

The components are integrated into the GML scripting environment in providing GML operators for components, allowing to control the application with a dynamic scripting language. The ability to use a scripting language for developing visual applications allows fast application development cycles, as the result of a code change is visible in realtime without the need for the time-consuming *edit-compile-test cycle* of compiled programming languages. The current development status of the framework provides a solid foundation to start developing dynamic visual applications.

The entry point to the framework is the built-in *Controller* object. It implements a complete runtime engine for visual applications and provides access to the following interfaces:

- a node-tree system for managing multiple nodes in a hierarchical structure
- a rendering and input device handling system
- an *event system* to keep track of internal state changes.

The node-tree stores nodes in a hierarchical structure (see section 5.2.2). Custom data can be attached to the nodes via the data management component of the SDK (see section 5.2.1). Through Grimaldo's GML bindings a user is provided an interface for controlling the node management process via a GML script. The node-tree's scripting facility is the foundation of a new feature for the GML: the possibility to create hierarchical 3D scenes of GML models without the need for external library dependencies, like the OpenSG scene graph system [Opea]. This feature is achieved in mapping a GML model into an entity called *GML context*. This GML context is wrapped into a Grimaldo node object. With this approach it is possible to manage the life-time cycle of multiple GML models within a single GML interpreter instance. The details of the GML context idea are described in section 4.2.

Control over the rendering process and the handling of input devices like mouse, keyboard and others are provided as a single SDK component, the *SceneViewer*. Its idea is inspired by the concept the *Blender* suite [2] [Ble] tackles the problem of providing different views on 3D data in combination with different methods for handling input device data, depending on the context the 3D data is displayed in. A simple example is to imagine a 3D object that is created by a user within a 3D modeling program. The object's mesh is editable, e.g. faces of the model can be extruded to give it the desired shape. Furthermore, the object as a whole can be selected and positioned within the virtual 3D scene. Editing and positioning are two separated views. In *edit mode* the object's mesh is editable, the position (well, at least the objects center point) can not be changed in this mode. To reposition the object it has to be put into *object mode*. In this mode clicking the object does not change the mesh but allows to arbitrarily position it in the 3D scene.

On the technical side a SceneViewer is a general 3D runtime engine for the Grimaldo framework. Its implementation and usage is highly application specific. Multiple SceneViewer can be registered with the Controller component. This *Strategy*-like pattern [GHJV94] offers a client the possibility to implement different *views* on the 3D scene. The SceneViewer interface provides methods for preparing an output environment, respectively cleanup the environment. The Controller automatically calls this methods when a SceneViewer is switched. This makes it possible to combine different output system within a single Controller instance. For instance, Grimaldo currently supports two SceneViewer, one implements a pure OpenGL viewer for GML models within the GMLApplet, the second one displays the content of the node-tree with OpenSG as rendering system. The implementation of the OpenSG SceneViewer for the GMLApplet is described in 6.5. The scripting support for Grimaldo makes it possible to dynamically switch between different SceneViewer at runtime.

The third major component of the Grimaldo SDK is its data management system. The smallest data unit is a *Property*. This is a template based class that allows to store arbitrary C++-objects as content. A Property is equipped with an *notification interface*. An interested entity can subscribe to the Property and gets informed whenever the content changes. This technique provides a flexible method to synchronize data defined within the Grimaldo framework with a host-application.

[2]The concept used by Blender is a general technique used in other applications, too. As a personal fan of this open source software it was taken as representative example for the technique.

Properties are the foundation of Grimaldo's *event system*. The system is primarily targeted to expose changes of the node-tree topology to the host-application. Whenever a node is created, deleted, etc., an event is emitted. This is achieved in defining one property for each GML operator dealing with Grimaldo's node management. The host-application can subscribe to the desired Property in registering a function callback that receives a pointer to the involved node when the operator is called. With this information the application can react to the internal state change. It is therefore possible to synchronize the topology of the node-tree with an external system. Best suited are hierarchical, node based system, as the internal node-tree is structured this way. In chapter 6 this technique is used to map the node-tree to the scene graph of OpenSG.

The event system is not limited to topology changes of the node-tree. Another example is a Property that contains the currently selected node. When a user selects a model in the 3D viewport the Property emits an event with the selected node attached. The host-application's GUI can now display relevant information to the node or trigger arbitrary actions based on the selected node. Each time a new node is selected the GUI is automatically updated via the callback mechanism.

The advantage of Properties and the event system is that it is very easy for a host-application to hook into the internal Grimaldo or GML system in simply registering a function callback. However, the mechanism goes even further. Via the *fake dictionary* functionality of a GML resource data can be attached to a GMLContextNode within a GML script, like so:

```
usereg

/gmlcontext node-create !node

:node begin
  /myIdentifier /myName def
end

:node /myFloat 4.2 def
```

The values can be retrieved within the script via the default access methods for dictionaries. However, the GMLContextNode is storing the entries as *FakeDictProperty* objects. This special Property type can be used to observe changes that are made to a fake dictionary entry of the node in registering a callback to the Property. When the data is changed within a GML program the callback is executed, containing the changed data as an argument. Again, in registering a callback by the host-application with the Property of interest it is easily possible to synchronize data defined in GML scripts with the application and react to changes. The communication is bidirectional, as the host-application can change the received data. The change is reflected within the fake dictionary entry. The usage of FakeDictProperty is described in section 5.3.2.

5.2 Core Components

5.2.1 Data Management

The design decisions for the data management were targeted to provide a storage system usable for GML contexts. A GML context stores resources that are making up a GML model. To keep the data structure flexible and extensible for future extensions a general storage system was developed. It is capable of storing arbitrary C++-objects. Its interface provides generic access to this data and does not dependent on the actually stored data. The generic access is enabled in introducing an *envelope* formed by the *Property* object. A Property is a generic data container. It is not specialized in any form, its only functionality is to store an arbitrary C++-object and provide identification information for it.

In view of the ability to exchange data between the GML and a host-application via Grimaldo interfaces a possibility had to be found to establish a data channel between the two worlds. The foundation of this functionality is directly integrated into a Property. A Property has a *notification interface* that allows to add signals to it. A signal is emitted whenever the data stored within the Property is changed. External entities like a host-application can subscribe to a signal and register a callback function. If the subscribed signal is emitted this callback gets executed. The Property emitting the signal is delivered as argument to the callback function. This simple system forms an flexible foundation to establish data channels between the GML and a host-application.

The GML context design demands the possibility to group together different properties that are forming the functional basis of a context. The management unit providing this ability is a *PropertyGroup*. Properties can be added to a group and required from it. This functionality can be used to logically group together data or to generally provide a data storage point. PropertyGroups can be used on arbitrary places in an application, either as standalone unit or as an interface a component derives from. Most of the Grimaldo components derive from the PropertyGroup interface, enabling a flexible way to extend them in adding functional units as Properties.

5.2.1.1 Properties

The data management system of the Grimaldo framework is organized as a container system. A container allows to store an arbitrary C++-object inside. Hence, the container is an envelope that contains arbitrary content. The management system knows how to handle these envelopes without the need to know anything about the data itself. The management system on the other hand does not have any requirements to the stored data (except some minor requirements discussed later in this section). The data management system has the task of a “post-office staff member” that has to:

1. hand over an envelope on a client’s request
2. take envelopes from clients and store them in a way they can be accessed quickly later on.

The Grimaldo framework uses a so called Property object to form the envelope. A Property can store C++-objects, thus allowing to pack arbitrary data or functional units.

Listing 5.1 introduces the usage of a Property. The theoretical foundations and the implementation details are described afterwards in this section. The early code-example should give the reader a better understanding of the usage. The code shows how to create two different Property types. The type of the object is denoted via the template parameter, an `int` and a `std::string` in this case. After constructing the Property itself the content is created and stored . Note that the Property expects a pointer to the content object. The constructor of the Property allows the developer to choose whether the Property takes the ownership of the data it points to, or not. In case the ownership is delegated to the Property the memory management is automatically handled. If the Property is destroyed, the stored object is destroyed, too.

Generic Programming: Template-Based Design The design goal of a `Property` to store arbitrary data objects is perfectly suited for the generic programming technique of *templates*, provided by the C++ programming language.

The Property object consists of a combination of two classes:

Non-template base class The base class contains the identification information of a Property like the name, an ID and a type-identifier that allows to inspect the type of the Property from a base class pointer. The base class inherits functionality from a notification interface that allows to add signals to a Property. The signals are used to notify subscribers of a change of the object a Property holds. The notification interface is described in more detail in section B.

```
enum PropertyType { P_INT, P_STRING };

typedef Property<std::string> StringProperty;
typedef Property<int>          IntProperty;

// 1. This Property takes ownership of the content object:
StringProperty *p_string = new StringProperty("String Property", id, true);
p_string->setObject(new std::string (" My String Property"));
p_string->setType(P_STRING);
delete(prop0); // the memory of the content is deleted automatically

// 2. The memory of the content of this Property is managed externally:
IntProperty *p_int = new IntProperty("Unmanaged Int Property", id++, false);
 int* raw_int = new int(3);
p_int->setObject(raw_int);
p_int->setType(P_INT);

delete(p_int); // the memory of the content is not deleted. it has to be freed manually.
delete(raw_int);
```

Figure 5.1: Creation of Properties with and without delegation of the ownership to the Property object

Template class The template class derives from the base and extends the interface to allow the storage of an arbitrary C++-object. The type of the object is defined by the template parameter.

The template technique offers the possibility to theoretically store all available C++-objects within a Property. However, the current implementation has some minor limitations to the stored object type described in 5.2.1.1.

The template design allows a very compact implementation of the Property object, showing the advantage of the generic programming paradigm: the ability to let the compiler generate code at compile-time saves the developer to write a lot of boiler-plate code and enables compact code implementations. The public interface for both classes can be found in listing 5.2.

Ownership A Property automatically takes the ownership of the object it stores. The ownership takes care of deleting the stored object if the Property gets destroyed.

In some cases an automatic memory management is not desired. For this case the constructor of a Property allows to disable the behavior in not taking the ownership of the content. The developer has to manually take care of the memory management of the content object. Moreover, the object must not be destroyed before the Property itself, which would result in accessing a non existent object, which - in the lucky case - results in a program crash.

Notification Interface The Property provides a notification system that uses signals to communicate with external entities. An external entity can subscribe a callback function to an specific signal of interest. If the signal is emitted it is propagated to the registered subscribers and the callback function is executed. This mechanism decouples the invoker of the event from the receiver, a design that is known as the *Command* design pattern [GHJV94].

The decoupled notification technique is the foundation for the data synchronizing feature between the GML and a host-application via the Grimaldo framework. It offers a simple possibility for a host-application to establish a communication channel to data defined in a GML script. Whenever the data is changed within the script a notification allows the host-application to update its internal data-structures accordingly.

Currently a Property defines one default signal: *onObjectChange*. It is fired in two cases:

1. If the Property method `setObject(T* object)` is called to store a new content object the signal is automatically fired
2. A developer can fire the signal manually in using the method `emitSignal(const std::string& signalname)` with the desired signal name as argument.

```
class PropertyBase : public NotificationInterface
{
public:
  PropertyBase(const std::string& name = "", int id = −1);
  PrpertyBase(const PropertyBase& source);
   virtual ~PropertyBase();

  void  setId(int id);
  const int getId() const;

  void  setName(const std::string& name);
  const std::string& getName() const;

  void setType(int type);
   int   getType() const;
};

class Property : public PropertyBase
{
public:
  I_GHOST_Property(const std::string& name, int id, bool take_ownership)
     : PropertyBase(name, id), m_object(NULL), m_is_owner(take_ownership) {}
   virtual ~Property() { destroy(); }

   virtual void setObject(T∗ object) { m_object = object; }
  T∗ getObject() const                { return(m_object); }

  void destroy();

  /∗! Convenience function to retrieve the correctly casted object from
        its base. ∗/
   static T∗ getObject(PropertyBase∗ base)
  {
     Property<T>∗ prop = dynamic_cast<Property<T>∗>(base);
     return( (prop == NULL) ? NULL : prop→getObject() );
  }

private:
  T∗   m_object;
  bool m_is_owner;
};
```

Figure 5.2: The Property interface (truncated)

```
// typedef for better readability :
typedef Property<std::string> StringProperty;

// Function callback:
void onObjectChange(PropertyBase* prop)
{
  std :: string* content = StringProperty::getObject(prop);
  // The line above is equivalent to:
  // std :: string* content = dynamic_cast<StringProperty*>(prop)→getObject()
  // The getObject(prop) form is only a convenient form to safe some typing.

  std::cout << "The property contains the string ''' << content << "''' << std::endl;
}

// Creation of the string−property. The property takes the ownership of the string:
StringProperty p_string (" string'', 0, true);

// Creation of the string object:
std:: string *myString = new std::string (" Init value.'');

// Subscribtion of the callback function to the onObjectChange
// event:
p_string.subscribe(boost::bind(&onObjectChange, _1), "onObjectChange");

// Setting the string−object into the Property. Here the 'onObjectChange'
// event is automatically fired:
p_string.setObject(myString);

// Change the string−object and manually fire the event:
*myString = "Changed value."
p_string→emitSignal("onObjectChange");
```

Figure 5.3: Subscription of a callback function to a event of a Property object

To subscribe a function callback to a signal two methods are provided:

- `bool subscribe(boost::function<void (PropertyBase*)> cb), const std::string& signalname = "onObjectChanged")`
- `bool subscribe(EventSlot* cb, const std::string& signalname = "onObjectChanged")`

The two methods do not differ in their functionality, the difference is that the first method uses the *Boost* library [Booa], whereas the second method utilizes a callback base class a subscriber has to derive from. The base class contains a method `onNotify(Property* prop)` that has to be implemented [3]. The Boost approach offers a flexible mechanism to easily connect various function types (pointers to simple functions, pointers to member functions and functors) to an event.

When the callback function gets executed the Property that emitted the signal is delivered to the function as a parameter argument. The argument is provided in its base class form and has to be casted to the correct type. In most cases this type is known as the subscription is made to a concrete Property type. If not, the `getType()` method can be used to determine the type.

Listing 5.3 shows how to create a Property with a `std::string` content object and how to subscribe an callback function to it. The signal gets emitted two times: first when the string object is set as content of the Property and second in a manual call to the `emitSignal()` method.

[3]The dependency on Boost of the Grimaldo framework can be disabled via omitting the preprocessor switch "WITH_BOOST" for the library at compile time.

The signal-interface implementation that is used to add signals, subscribe callbacks and emit signals is described appendix B.

Limitations The template implementation of a Property allows the stored data to be of an arbitrary type. However, the ability that the memory management is handled by the Property introduces one limitation. An object needs a public destructor for destroying it when the Property is the owner of the object. If an object does not have a public destructor the compiler will complain when compiling the Property template code. Thus, there is currently no possibility to store objects with non-public constructors inside a Property (i.e. many reference counted objects do hide the destructor to prevent manual deletion).

5.2.1.2 Property Groups

A PropertyGroup is an organizational unit that acts as storage container to group together different Properties. The interface of a Property container is not very complicated as it simply offers the possibility to store and retrieve Properties.

PropertyGroups are widely used within the SDK. The Grimaldo components

- Node
- Controller
- SceneViewer

are inheriting the interface and thereby are able to store Properties. This functionality is used to attach data and functional units directly to the components.

The PropertyGroup feature of a node is forming the data backend for a GML context. As described in section 4.2 a GML context consists of *context-sensitive* resource data-blocks. The single data-blocks are packed into Properties and attached to the node. This allows an elegant way of mapping GML context to a Grimaldo node. Another example demonstrating the flexibility of a PropertyGroup can be found in section 5.2.3. In this case the support for new input devices is demonstrated in adding a Property with the corresponding functionality.

5.2.2 Node System

To extend the GML with a node-based context system a data-structure for storing and accessing nodes is necessary. For this reason the Grimaldo node-tree system was designed. The system implements a basic node-tree structure that is capable of storing nodes in a hierarchical manner. The hierarchy supports a simple *one parent - many children* relationship.

The node system is managed by a the Grimaldo Controller. The Controller initially stores one root node, further nodes can be added to it. With this structure it is possible to build up

To dispatch operations to the nodes in the tree a traversal method is provided. So called *Actions* form functional units that are recursively send through the tree. Different built-in Actions are provided and form a reasonable set of functionality to work with. For instance, an *IntersectAction* sends a ray through the scene and returns the nodes where the bounding box is hit.

The node-tree system is a very basic scene graph system. It is not intended to act as a scene graph with a rich feature-set. Many thoughts were put into scene graph systems that are available and free to use, so there is no reasonable cause to “reimplement” the wheel and construct another one. Moreover, the GML already connects to the OpenSG library [Opea], providing a powerful scene graph implementation with a sophisticated feature-set. The node-tree implementation of the Grimaldo framework is targeted to provide a flexible storage system for nodes that can easily be integrated into the node-management system of external applications or frameworks. The strength of the node-tree implementation lies in the integration of the data managements system described in the previous section. Each node is derived from a PropertyGroup object. Thus, it can have arbitrary data attached to it, allowing a flexible system for extending the capabilities of a node implementation. For instance, this feature is used to provide spatial information to each node to position it within the 3D scene. A demonstration of using the Grimaldo node-tree connect to an external scene graph system can be found in chapter 6. In this example the node-tree storage structure makes it possible to “naturally” map a node to the OpenSG scene graph system, connecting the advantages of both worlds.

5.2.2.1 Node Traversal with Actions

An Action is a functional unit that traverses through the node-tree and performs a specific task on the visited nodes. An Action starts at a given node and recursively traverses through all the nodes below the start node. The traversal follows the direction top-to-bottom and left-to-right.

The Grimaldo framework predefines a set of Actions to perform standard tasks on the node-tree. The actions are using methods from the node base interface, which is described in the next section. The following built-in actions are available:

Update The *UpdateAction* prepares a node for rendering.

Draw The *DrawAction* triggers the actual rendering of the node.

Finish The *FinishAction* traverses the node-tree after all the rendering is done.

InputEvent The *InputEventAction* is send to the node-tree whenever new input data is available, for instance when a mouse click occurs.

Intersect A request to check if a node is hit by a ray is triggered by an *IntersectAction*.

5.2.2.2 Node Interface

The node interface defines a standard set of methods that are common to every node type. To create a new type this interface has to be implemented. Listing 5.4 shows the interface of the node base class. The node is derived from the PropertyGroup object. It is thereby possible to add properties to each node type.

The interface provides methods for

- identifying a node
- rendering
- input device data handling
- bounding box intersection
- adding and removing other nodes and retrieving the parent node.

Rendering The rendering of a node is three-fold. The interface methods responsible are:

```
virtual void update(Action* action)
virtual void redraw(Action* action)
virtual void finish (Action* action)
```

At first the `update()` method of each node in the node-tree is called. It is responsible for performing pre-render calculations or other task that are needed to be performed before the actual rendering takes place. The rendering then is triggered by the `redraw()` method. After all nodes are drawn the `finish()` method is called to allow a node to perform finishing tasks like calculating the render time or similar computations. The three render methods currently do not use the given action. For further extensions to the system this action can be used to develop more sophisticated render features like OpenGL state sorting before rendering or similar techniques.

Input Handling If a node wants to handle events from input-devices it has to implement the *handle()* method. It is called whenever the host-application has new event data. The data is delivered by the InputEventAction. It contains access to the actual input events with the *InputEvent* object. The structure of an InputEvent is described in section 5.2.3. The object contains the aggregated events from all supported input devices. With this data the node can interact with mouse, keyboard or similar events. If the processing of the event makes necessary a redraw of the node the request can be set with a flag of the InputEvent.

```
class Node : public PropertyGroup
{
public:
  Node* getParent() const;
  void   addChild(Node* node);
  bool   subChild(Node* node);
  const std::vector<Node*>& getChildren() const;

   virtual void update(Action* action)        = 0;
   virtual void redraw(Action* action)        = 0;
   virtual void finish (Action* action)       = 0;

   virtual bool handle(Action* action)        = 0;

   virtual bool intersect (Action* action)    = 0;

   virtual void updateBBox()                       = 0;
   virtual Box3f getBoundingBox() const            = 0;

  void setName(const std::string& name);
  const std::string& getName() const;

  void setId(int id);
  int   getId() const;

  private: ...
}
```

Figure 5.4: Node interface (truncated)

Picking If the node should be pickable the `intersect()` method has to be implemented. The method for detecting an intersection is up to the node implementation. The IntersectAction provided to the method contains the following information to execute a pick:

- A *Ray* object with a start point and a direction
- A distance of interest given by a start and end point.

5.2.3 3D Runtime Engine

The Grimaldo framework provides a 3D runtime engine to render the managed nodes and to allow user interaction with them. The runtime engine is provided by the *SceneViewer* object. The Grimaldo framework is intended as a template framework that provides the basic infrastructure to develop visual applications. Especially the runtime engine of an application is highly specific to the field of application. Therefore the viewer only provides a general interface to control three aspects of an application which can be used as a base system. The three aspects are:

- The initialization of an output system and the possibility to switch between different systems
- Interface methods for the rendering process
- A standardized and extensible input event system to delegate input events to nodes of the node-tree.

The SceneViewer concept is very flexible and allows different *views* on a 3D scene represented by the node-tree. It is integrated into the *Controller* (the Controller is the main component of the framework and is described in section 5.2.4) with the strategy design pattern method. The realization of the strategy pattern allows to replace a viewer at runtime. In combination with the GML scripting facility this allows interesting possibilities for the development of visual applications. For example, consider a node type that is integrated into a physics engine. The position of a node is computed according to its geometrical shape and physical attributes. The implementation of the default SceneViewer is used to display the nodes of the scene in rendering the geometry. A second SceneViewer is rendering a debug version of the scene, which

```
class SceneViewer
{
public: ...
  void setName(const std::string& name);
  std :: string getName() const;

  void setId( int id);
  int  getId() const;

  virtual void show();
  virtual void hide();

  virtual bool handle(InputEvent* event);

  virtual void framePrepare();
  virtual void frameDraw();
  virtual void frameFinish();

private: ...
}
```

Figure 5.5: SceneViewer interface (truncated)

visualizes the physics forces, contact points, etc. (most physics engines available provide an integrated debug viewer). Via a GML operator it is possible to dynamically switch between two views, allowing to quickly identify errors in the physics setup. This example shows that different SceneViewers and their ability to be scripted enable an efficient deployment of 3D scenes.

SceneViewer are not restricted to display the content of the node-tree in an realtime viewport. It is also imaginable to implement special viewer that directly render the visualization of a node-tree into a image or video format. With the scripting capabilities of the Grimaldo framework it would be possible to animate a scene and directly render it out as a movie, still image, etc.

The interface of the SceneViewer is shown in listing 5.5. It provides an identification interface for setting a name or an integer ID for the object. This is necessary to identify a SceneViewer if multiple are registered with the Controller. The `show()` and `hide()` methods are used when the viewer is activated or deactivated, respectively. `show()` can be used to setup the rendering environment or to prepare internal data structures when the viewer is activated. `hide()` - on the other hand - is responsible for cleanup tasks, like restoring the original environment before the viewer was activated. The three methods `framePrepare()`, `frameDraw()` and `frameFinish()` are used to control the per-frame rendering process. It is possible to perform pre-rendering and post-rendering tasks, as well as the actual rendering in `frameDraw()`. The `handle()` method is responsible for providing the input device events from the host-application to the node-tree. Such events include mouse, keyboard and similar events. As one of the major goals of this thesis is to integrate the GML into other applications it is important to consider a standard format of bringing input device data from different applications into the framework. For this reason the *InputEvent* object is provided.

The InputEvent object The InputEvent object is responsible for connecting input devices of the host-application to the Grimaldo framework in a form that is recognized by the internal framework components. It allows a customizable abstraction of input device data that can easily be extended to support special input devices.

The need for an object that brings input device data into a standardized form is simple: The Grimaldo SDK is intended to be used in different host-applications which do have different ways and formats to store input device events. For instance, both, OpenSG and the GLUT toolkit [GLU] are providing access to input device data to mouse and keyboard. Both use a similar system to represent the data with different integer values for mouse buttons, a key or a button state. Unfortunately, the formats are not directly compatible. As the Grimaldo framework is used with both systems it was necessary to develop an own format for input device data. The format is accumulated into the InputEvent object.

Current the following mouse and keyboard events are supported:

- ButtonLeft
- ButtonRight
- ButtonMiddle
- ButtonPress
- ButtonRelease
- ButtonDrag
- ButtonNone
- KeyNone
- the value of a keyboard event

On the application "border", where the input data from is forwarded from the host-application to the framework, the data has to be converted to the Grimaldo format. For GLUT and OpenSG host applications convenience functions are provided that automatically perform this task. All internal Grimaldo components are solely using this internal format to provide a consistent input device event handling.

There are two possibilities to extend the InputEvent object. The first one is to simply derive from it and extend the needed events. The second on is to take advantage of the PropertyGroup interface the InputEvent inherits from. For example, an application is supporting a multi-touch input device and the application has to use the multi-touch events to control a GML object. For this purpose the SceneViewer integrated into the application has to be extended to support the multi-touch events. The traditional way is to derive from the object and extend the support. However, the fact that the InputEvent is a PropertyGroup allows to simply add the new functionality as a Property. For this a class with the multi-touch data entries is created and stored within a Property. This Property is than added to the InputEvent object. From then on the SceneViewer can retrieve the additional multi-touch data and work with it. The viewer can check for the existence of the multi-touch property at runtime. If the property is present the multi-touch functionality is used, otherwise only the conventional input events provided by the original InputEvent object are processed.

5.2.4 Controller

The *Controller* is the frontend of the Grimaldo framework. It is responsible for initializing the main data structures and to provide access to the components of the framework. The Controller has two main tasks:

- node management
- runtime control

These two tasks are subject of the following sections.

5.2.4.1 Node Management

The Controller is responsible for creating and holding the *node-tree*. The node-tree is the main data structure of the Grimaldo framework to manage its nodes. It provides the possibility to organize multiple nodes in a hierarchical structure. Each node can have many children but only one parent. In the initialization phase of the Controller the node-tree is constructed in creating the *root* node. With the current implementation the root node is a GMLContextNode. As described in section 4.2.2.3 it is used to provide a compatibility layer for GML scripts that are not aware of the GML context extension. After the creation the node-tree only consists of this single node, which is accessible with the `getRoot()` method.

The node-tree has exactly one node that is active at every time. This functionality is used for the resource management system of the GML to define a *focus context* (see 4.1.2). To manage active nodes a stack structure is used, the *activity stack*. It is possible to push and pop nodes from the stack. The topmost element is the active node. After the creation of the root node it is set as the first active node. This element can not be poped from the stack, as per definition there is always one active node (this design decision was made to directly support the focus context mechanism). The responsible methods for manipulating and examining the activity stack are `pushNode()`, `popNode()` and `getActiveNode()`.

```
Node* getRoot() const;
Node* createNode(const std::string& type);

Node* findNode(const std::string& name) const;
Node* findNode(int id) const;

void  pushNode(Node* node);
void  popNode();
Node* getActiveNode() const;
```

Figure 5.6: Node management interface of the Controller

```
void framePrepare();
void frameDraw();
void frameFinish();

bool handle(InputEvent* event);
```

Figure 5.7: Runtime control methods of the Controller

The Grimaldo node system supports the management of multiple types of nodes. For this reason a possibility is needed to instantiate nodes of a desired type. The Controller provides the `createNode()` method for this task. It takes a string as input parameter that is used to determine the type of node that has to be created. When a node is created a pointer to the Controller is set within the node. This is necessary to allow the node to be pushed and popped from the activity stack. The node interface provides a convenience method for these two tasks, `push()` and `pop()`. A developer can simply call these methods and does not need to utilize the stored Controller pointer directly.

Currently the framework only implements a single type, the GMLContextNode. It is created with a call to `createNode("gmlcontext")`. Nodes can be added and removed from the tree in using the `addChild()`, respectively `subChild()` methods of the node's interface. To find a specific node the Controller provides the `findNode()` method. It takes a name or an integer ID as input parameter and searches the node-tree for the desired node.

The node management interface is summarized in listing 5.6.

5.2.4.2 Runtime Control

The Controller can be used by host-applications to control the runtime engine of the Grimaldo framework. The engine is formed by a SceneViewer object that has to be implemented by the host-application to render the content of the node-tree and to delegate input event data to the nodes. Predefined Actions described in section 5.2.2.1 ease this task in providing default functionality. The methods for the runtime control of the Controller are listed in section 5.7.

The methods directly call the correspondent methods of the active SceneViewer. As with nodes there is always one active SceneViewer (although there is no activity stack this time). Multiple viewer can be registered that implement different *views* on the node-tree. To register an instance `registerViewer()` is used. A SceneViewer can be activated in calling `activateViewer()`. In this case the currently active viewer is shut down in a call to its `hide()` method before the new viewer is activated by calling `show()`. The show/hide mechanism is described in section 5.2.3.

In case of the integration of the Grimaldo framework into the GMLApplet the methods `framePrepare()`, `frameDraw()`, `frameFinish()` and `handle()` are delegated to the corresponding methods of the Grimaldo Controller. The Controller in turn utilizes the functionality of the currently active SceneViewer to handle the calls. The default SceneViewer implementation for the GMLApplet is described in section 4.2.3.2.

Event Name	Description
Node.create	Emitted whenever a node is created with the `node-create` operator
Node.addChild	Emitted whenever a node is added to another node with `node-addChild` operator
Node.removeChild	Emitted whenever a node is removed from another node with the `node-removeChild` operator
Node.removeChildren	Emitted whenever all child nodes of a parent are removed with the `node-removeChildren` operator
Node.begin	Emitted whenever a node is set as active node with the `node-begin` operator
Node.end	Emitted whenever the active node is deactivated with `node-end` operator
Node.selected	Emitted whenever a node is selected by an picking operation

Table 5.1: Events exposed by the Grimaldo Controller

5.3 Features

5.3.1 Event Subscription System

The subscription system allows an application to react on internal state changes of the GML or the Grimaldo framework. The main goal of the system is to connect the Grimaldo node-tree with the node system of an external application. It should enable the possibility to map the node-tree topology to another system *instantly*, coupled to the changes made to the node-tree structure within a GML script. For instance, an application subscribed to the *Node.create* event is notified whenever a node is created inside a script in utilizing the `node-create` operator. The application can react to this signal in creating a corresponding node in its own node-structure. Via the notification system the external application also receives a reference to the node that is created by the `node-create` operator. The OpenSG integration for GML contexts in chapter 6 uses the event system and the delivered node to create a corresponding OpenSG *Node* and adapts the delivered node to its system. The node is then displayed with the OpenSG rendering system. This example shows that the notification system can be used to directly script an external application.

Currently events regarding the node management system exported, but the system is not limited to this. In fact, each component that has access to the Grimaldo Controller can add events. The framework supports the events listed in table 5.1.

5.3.1.1 Event Subscription

The central entity to subscribe to an event is the Grimaldo controller. Internally each event is a Property attached to the controller. Therefore the subscription is handled with the PropertyGroup interface described in section 5.3.1. Listing 5.8 shows how to subscribe a callback that is called whenever a new node is created with the GML command `node-create`.

5.3.2 Data Registration from GML Scripts

5.3.2.1 Requirements

For building visual applications it is beneficial to have a way of defining data in a GML script that can be accessed within a host-application. If the data is changed in the GML environment the host-application should be informed to react on the change.

To use data defined within a GML script from a host-application it is necessary to register this data with the host. The traditional way of registering a variable-set is outlined here to get a better understanding of the general workflow. This section then shows an alternative way of data registration in using the Property system of the Grimaldo framework.

The illustrated example shows a typical development scenario for connecting variables of a GML script to a host-application. It is taken from a real-world project of a master's thesis worked on by Manfred Krieger [Kried]. The goal of his thesis is to integrate the GML into the Maya 3D suite [4] [May].

[4]Maya is an industry standard 3D suite for modeling, animating, rendering and compositing 3D objects and scenes.

```
void onNewNode(Property* prop)
{
  std::cout << "New node created" << std::endl;
}

void main(int argc, char** argc)
{
GMLApplet applet;
// ... applet initialization goes here ...

Controller& c = applet.getController();
controller.getProperty("Node.create")→subscribe(boost::bind(onNewNode, _1));

// ... functionality goes here ...
}
```

Figure 5.8: Subscription of a callback to the *Node.create* event

The demonstrated example simplified to focus on the general idea of variable registration. The idea is to use GML scripts to generate geometry that can be modified and post-processed in Maya. The GML script is a parametrized model-generator that defines a parameter-set for controlling the model's shape. The 3D artist now loads this script into Maya and gets a live-preview of the model in the 3D viewport. To change the parameter-set it is necessary to change the desired parameter within the script, reload it and evaluate the outcome in the viewport. As it is the job of the artist to generate visually appealing geometry and not to mess with editing a GML script the parameters defined in the script have to be exposed into the Maya GUI to make them editable in a comfortable way. To not have to reload the script every time a parameter is changed this task is automatically triggered when the value of a parameter alters. Changing a parameter from the GUI now recomputes the shape of the model in real-time. The direct visual feedback allows to efficiently adjust the parameters to achieve the desired model. After finishing the adjustments the shape can be exported as a Maya mesh, allowing to use the full feature-set of the 3D suite to further process the appearance of the model.

On a programming level the illustrated example has two requirements:

- A way to hand over the parameters from the script to Maya
- A callback that calls the update routine of the model when a parameter is changed.

5.3.2.2 Traditional Operator Approach

The traditional way to register the parameters and the update-callback is to implement a GML operator responsible for this task. In this case the operator is called `maya-register-dict`. Listing 5.9 shows a section of a GML script to illustrate the above example. The script creates a model of a car engine and has the two parameters

- *cylinder*, for the number of cylinders and
- *angle*, for the angle of one cylinder.

The listing does not show the full script, it takes out the section the operator is used to register the parameter and the update-callback.

It is good practice to create a dictionary that contains all the data corresponding to the model. This dictionary is created and filled with data in lines 8-13. The data consists of the parameters dictionary that is created in lines 3-6 and the update-callback `Engine.compute`. In line 12 the update-callback is called to actually generate the model. The generation code `Engine.compute` is defined elsewhere in the script and not shown here. The important thing to know is that the code is using the values from the parameter dictionary to create the shape of the car engine.

In line 15 the actual registration-magic happens. The *maya-register-dict* operator takes two token from the stack: the dictionary that defines the parameters and the update-callback of the model. Inside the operator these two are stored and connected to the Maya application. Maya extracts the actual data from the parameter token and creates a graphical

```
usereg

dict  dup !parameters begin
  /cylinder    8 def
  /angle      45 def
end

dict  dup !model begin
  /parameters       :parameters         def
  /model−update  {  Engine.compute } def

  model−update
end

:parameters :model /model−update get maya−register−dict
```

Figure 5.9: Registering data to a host application with a registration operator (here: `maya-register-dict`)

user interface for them. When a parameter is changed in the GUI Maya calls the registered update-callback to trigger a recreation of the model.

To register the data the `maya-register-dict` operator first has to be implemented and is then added to the GML script. The operator approach has two issues:

The register operator limits the compatibility of the GML script When using the register operator in the script the GML has to support it. If the operator is not know a runtime error is thrown when executing the script. Thus, the car engine script in listing 5.9 can not be used with a version of the GML that does not support the specific registration operator, although the rest of the script works perfectly fine as the operator does not influence the functionality of the script itself but only connects it to Maya. Another application has to copy the script and remove the operator to be able to use it. If the car engine script eventually gets an update from its developer all dependent scripts have to be updated, too, which does not allow a convenient update cycle.

There is no default GML operator for registering data to host applications Future applications that have to exchange data with the GML will write their own version of a registration operator to connect their application, introducing the dependency problem described above. A default registration operator provided by the GML would circumvent the problem. However, it is difficult to foresee the needs of such an operator and although it is quite flexible to register a dictionary token that can have arbitrary elements and an update function, a more general solution would be beneficial.

The Grimaldo SDK offers an alternative way to register script data that is than accessible to the host-application. It bypasses the necessity to develop a custom operator to establish a connection. Moreover, the dependency of the GML script on a specific operator is eliminated.

5.3.2.3 Alternative Technique for Data-Registration

The key to the alternative registration of data from within a script is the *fake dictionary* functionality of GML resources. It provides the possibility to access data members of a C++ class represented by a token type. In [Ger05, page 54] a fake dictionary is defined as: "*A fake dictionary is a token which behaves if it were a dictionary, i.e. it allows the insertion of key/value pairs and the retrieval of a value by a given key, with the help of a resource. However, insertion may be retricted to certain keys as well as certain key and value types.*"

The Grimaldo resource can be used as a fake dictionary token. It therefore has to implement the fake dictionary methods of the `GMLResource` interface shown in listing 5.10.

Both methods receive a *ticket* as their first parameter. This ticket denotes the resource type that is used as fake dictionary token. Inside the two methods it is used to identify if the resource is responsible for the ticket, or not. If the resource wants to handle the ticket the second parameter - the *keyname* - is used to identify the entry of interest. In case of the `getToken()` method, which provides read access, the token corresponding to the keyname is returned. In case of a

```
virtual Token* getToken(const Token& ticket, const Token& keyname);
virtual bool   putToken(const Token& ticket, const Token& keyname, const Token& value);
```

Figure 5.10: Fake dictionary interface of a GMLResource

```
1 usereg
2 dict dup !myDict begin
3   /myData 4.2 def
4 end
5 node-controller /myCustomDict :myDict put
```

Figure 5.11: Usage scenario for the fake dictionary functionality of the Grimaldo Controller

write access handled by `putMethod()` the *value* parameter is examined and used to update the entry corresponding to the keyname.

Now, how can this functionality be used to register data from within a GML script with a host-application? The Grimaldo resource handles two token types, *Context* and *Controller*. The idea is to make it possible to append data in form of a token to these two entities. The host-application should then be provided a convenient method to access this data. Listing 5.11 shows a usage scenario of the desired functionality. Lines 2-4 define a custom dictionary named *myDict* with a *myData* entry with the float value *4.2*. Line 5 adds *myDict* with the keyname *myCustomDict* to the Grimaldo Controller. The *myDict* dictionary token should now be accessible within a host-application in using the keyname *myCustomDict* to access the token.

To achieve this the Property system of Grimaldo is used. Its ability to attach arbitrary data to entities derived from the PropertyGroup interface - like the Grimaldo Controller and nodes - makes it perfectly suitable for this task. The string representation of the keyname token common to the two fake dictionary methods `getToken()` and `putToken()` is used to relate a fake dictionary entry to a Property with the same name. The host-application can then retrieve the token stored within the GML script in retrieving the corresponding Property of the Controller or the node the dictionary entry was made on.

The advantage of using Properties to store a token is the ability to register callback functions that are executed in case the content of the token is altered via a GML command. A host-application therefore can register a notification callback to any dictionary entry of the Controller or a node. It is automatically notified if the value of the subsribed entry changes. The functionality is provided by the *FakeDictProperty*.

FakeDictProperty To implement the functionality described in the previous section a new data structure is introduced, the *FakeDictProperty*. The FakeDictProperty is a specialization of the Property template class holding a token as content object. The header definition is shown in listing 5.12.

The property is directly integrated into the fake dictionary interface of the Grimaldo resource. The `putToken()` method at first checks if the resource is responsible for handling the ticket with the `isType()` method. If this is the case the token is converted into a concrete Controller or GMLContextNode object, depending on the token type. The object is than queried for the existence of a property with the keyname given to the `putToken()` method. If no property can be found a new FakeDictProperty is created. For this the keyname token is converted into its string representation forming the name of the FakeDictProperty. The constructor is called with this name, the integer ID retrieved in calling `keyname.i0()`

```
class FakeDictProperty : public Property<Token>
{
public:
  FakeDictProperty(const std::string& name, int keyname, GMLInterpreter* inter)
   virtual void setObject(Token* object);
}
```

Figure 5.12: FakeDictProperty interface

```
usereg

dict dup !parameters begin
  /cylinder     8 def
  /angle      45.0 def
end

dict dup !model begin
  /parameters     :parameters      def
  /model−update { Engine.compute } def

  model−update
end

node−controller /parameters  :parameters                put
node−controller /model−update :model /model−update get put
```

Figure 5.13: Registering data to a host application with FakeDictProperties

```
// the v_controller variable provides access to the Grimaldo Controller
Token∗ parameters  = dynamic_cast<FakeDictProperty∗>(v_controller→getProperty("parameters")>;
Token∗ model_update = dynamic_cast<FakeDictProperty∗>(v_controller→getProperty("model−update")>;
```

Figure 5.14: Accessing registered data via the Grimaldo Controller

and a pointer to the GML interpreter. Afterwards a new token is created and set as content object to the FakeDictProperty with the `setObject()` method. The method takes care of protecting the token against the garbage collection of the GML interpreter, which is the reason the FakeDictProperty needs a pointer to the interpreter. The last step is to copy the content of the *value* token to the token within the FakeDictProperty. After these steps the new property is added to the Controller or the GMLContextNode, respectively.

If the Property with the keyname ID is found it is casted to a FakeDictProperty. The contained token is retrieved with the `getObject()` method and is updated with the content of the *value* token.

To read a FakeDictProperty `getToken()` is used. Again, the given ticket is checked and in case it denotes a Controller or a GMLContextNode the actual object is retrieved. The object is searched for the Property with the ID of the keyname token. If it is found, the Property is casted to a FakeDictProperty and the token delivered by `getObject()` is returned. In case no corresponding Property is found an interpreter error is set.

Example In the Grimaldo SDK the Controller acts as a registration central data can be registered to. The host-application then uses the Controller to access the data in requiring the desired Property from its PropertyGroup interface. Listing 5.13 rewrites the former listing 5.9 to use the controller as registrar.

The fake dictionary functionality of the controller is used to register the two token of interest. The data set is split into a *parameters* and a *model-update* entry. The splitting shows the flexibility of this method: Compared to a register operator there is no rigid signature that has to be met. The number of registered entries as well as their naming is up to the script developer.

Within the host application the code in listing 5.14 shows how to access the registered token to process them further.

5.3.3 Shading System Control

The Grimaldo framework provides the possibility to use an existing shading system of a host-application. The GML comes with its own shading system that uses a set of predefined materials and materials that can be dynamically scripted at runtime to define the shading of GML models. When integrating the GML into an existing application it is beneficial to control the shading of a model with the existing material system. For instance, when using GML models within a 3D

```
class GMLMaterialHandler
{
public: ...
  virtual int  getMaterialID(const std::string& name) = 0;
  virtual bool switchMaterial(int id)                 = 0;
}
```

Figure 5.15: GMLMaterialHandler interface (truncated)

modeling environment with a material editor the different materials of the GML model should be controllable with this editor. To achieve this Grimaldo provides the *GMLMaterialHandler*.

The GMLMaterialHandler is the frontend of a hooking mechanism that is integrated into the internal rendering system of the GML. To render a model the GML uses the *MaterialIndices* object. It provides mapping structures that relates the faces of a model to the corresponding material. During the rendering process the list of active materials of a model is iterated. The material is activated in setting up the correct OpenGL states like color or textures. Then, the faces corresponding to the material are drawn. This step is repeated for all active materials.

The idea of the GMLMaterialHandler now is to hook into the activation process of the material. The MaterialIndices object contains the actual library of materials within an own structure, the *MaterialLib*. It provides a list of available materials. Each material can be activated in the `activateMaterial()` method. As input parameter the ID of a material is given. The GMLMaterialHandler is integrated into this method to allow an external application to override the default GML material activation with custom code.

There is one problem, though. An external application has no information that relates the ID given to `activateMaterial()` to the actual used material. For instance, within a GML script the material can be changed with the following code:

```
/gold setcurrentmaterial
```

All faces that are created after this call have the material *gold*, until the material is changed again. The gold material internally relates to a specific ID, which is used in `activateMaterial()`. The external application needs the information that the given ID relates to the gold material to be able to use its own representation of the material to finally activate it. For this reason the external application is provided with two bits of information, the ID of the material within the GML system and the name string of the material. To not have to process a string comparison each time a material is activated the GMLMaterialHandler implements a caching mechanism that is able to related the GML material ID to the corresponding ID of the external material system.

The described process results in the interface of the GMLMaterialHandler shown in listing 5.15. An external application has to implement the two abstract methods for integrating the own shading system into the GML. The process to activate a material is started when the GML makes a call to its `activateMaterial()`. The GMLMaterialHandler is comparing the given ID with its internal caching system. If the ID is not present the `getMaterialID()` method is called, where the external application returns its own material ID of the given material name. The caching system then adds a new entry that relates the GML ID to the external material ID. Afterwards the `switchMaterial()` method is called with the ID of the external material, which is then activated by the external application. If the mapping between the material IDs is already known `getMaterialID()` is not called and no string comparison is needed to find the related ID. The related ID is directly used to call `switchMaterial()`.

The narrow interface of the GMLMaterialHandler allows to easily use existing material systems to control the shading of GML models. In section 6.4 an example is given to use OpenSG materials for the rendering of GML models.

6 Showcase: OpenSG Integration

6.1 The OpenSG scene graph system

In computer graphics one popular tool for organizing 3D scenes is a *scene graph*. The definition in [Han03] describes it the following way: "*A scene graph is a directed, acyclic graph which holds the complete description of a virtual universe, i.e. of a complex, graphical scene. The nodes of a scene graph define the geometric data (objects and transformations, the attributes (color, material, etc.) and the camera parameters required to view the scene from a certain point in space.*"

A scene graph implementation is highly application specific. Its organizational structure varies, depending on the purpose it is used for. What is common to all scene graphs is that they form an abstraction layer of the underlying graphics hardware. Many scene graph systems are build on top of the OpenGL graphics library, which is a low-level controller for graphics hardware. OpenGL is a *state machine*, where all states can be accessed and altered by an application [Opeb]. The states includes colors, textures, transformations and other attributes important for the rendering process. As a low-level library OpenGL simply renders *vertices* in form of graphical primitives like points, lines, polygons, etc. Before a primitive is rendered the states have to be configured to set up the correct environment, including material settings or lighting. Then the graphics hardware is advised to render the desired vertex data. To use modern graphics hardware efficiently it is important to provide the hardware with large chunks of homogeneous configured vertices. State changes are time consuming, therefore it is beneficial for the performance of the rendering process to sort together primitives with the same state configuration and provide them to the hardware in one rush. This is where scene graphs come into play. In contrary to the graphics hardware a scene graph knows the whole scene and is able to optimize the contained data before it is delivered to the low-level graphics library. A scene graph provides a data structure that organizes the logical data of a 3D scene. The structure consists of *nodes*, organized in a tree or a graph structure. So, for instance, before the scene gets rendered the scene graph can get traversed to collect all nodes containing geometry data that has to be rendered. It is possible to only collect the nodes that are visible, others are skipped. Then the nodes are sorted by their state and rendered with a minimum of overhead [OSGa]. The possibility to optimize a scene before the actual rendering can dramatically increase the rendering performance.

A scene graph typically distinguishes between *inner* nodes and *leaf* nodes. Inner nodes represent attributes that influence their whole subgraph. For instance, a transformation node relocates all nodes in its subgraph or a light node lights only the nodes of the subgraph. Leaf nodes are forming geometry objects.

6.1.1 Design Goals

OpenSG is a scene graph system that is build on top of the OpenGL graphics engine. The goal of OpenSG is to be a portable scene graph system for realtime graphics programs like virtual reality applications [Opea]. The following paragraph introduces design goals of the system and describes them.

Multi-thread safe data Multi-threading is an important technique especially for realtime graphics where applications consists of several different threads. It is crucial that threads accessing the same part of the graph are working on valid data sets and that changes to the data are synchronized correctly. OpenSG introduces the *Field* data structure to store data in a way that makes it possible to consistently replicate the data between threads. A field either consists of a single value - a *SingleField* - or of multiple values of the same type organized in a dynamic vector - a *MultiField*. A SingleField contains scalar data, MultiFields are holding references, the actual data is stored outside. Fields provide an *serialization* interface. If it is implemented by the developer the data of a field can be converted to and written from a buffer, for instance a file.

To encapsulate the data replication OpenSG uses *FieldContainers*. Within this structure fields can be grouped together. A FieldContainer forms the base class for most entities that are stored within the scene graph, like *Nodes* or *NodeCores*, which are described later on. If a SingleField is replicated its scalar data is simply copied. With MultiFields only the references are replicated, unless a thread is changing the data. In this case a private copy is made and the data is changed locally [VBRR02]. To keep track of the changes a separate *change list* is utilized. When the data of a FieldContainer is changed, the FieldContainer and a bit mask indicating the changed fields are stored within this thread-specific list. On a synchronization only the data indicated by this list is copied [RVB02]. The described synchronization mechanism is also the base for the clustering functionality of OpenSG, which makes it possible to render a scene distributed on multiple hosts.

Reflectivity To allow generic access to FieldContainers an interface to provide information about the container itself is needed, as well as access to the contained data. In OpenSG this functionality is called *Reflectivity*. A FieldContainer knows which fields it contains, the fields on the other hand can be accessed by generic methods. The information needed for reflectivity is kept in *Types*. Types are holding a description of the container or the field, respectively. With this information it is possible to identify the content of a FieldContainer and to access its fields. The reflectivity interface enables to develop *generic loader and writer* applications for the scene graph. Its content can be serialized to and read from a buffer, which allows to save the content of the graph into a file. The same mechanism is used for clustering to synchronize data onto multiple cluster servers.

Extensibility OpenSG is designed to allow great flexibility in application development. It is possible to develop new FieldContainers for special purposes. Features like cluster synchronization and serialization can easily be implemented with the provided interfaces. The added FieldContainers are automatically integrated into existing functionality, for instance, an file loader/writer can serialize the new FieldContainer type without changes to its own code.

OpenSG goes even further. Via reflectivity it is possible to replace FieldContainer types at runtime with an instance of another class that features the same interface. This is possible via the *Protoype* pattern. A prototype is an instance of a FieldContainer that is accessible via its type information. If a FieldContainer is created the prototype gets cloned. To adapt an application to a special environment it is possible to replace the prototype of a FieldContainer. All instances of this type are then created as copy of the replaced prototype. In changing the prototype it is also possible to set default values for newly created FieldContainers. The mechanism allows to replace arbitrary parts of OpenSG's system, without touching the library code, making adaptations to special purposes very flexible. [RVB02]

Separation of hierarchical and content data The scene graph is divided into two entities: *Nodes*[1] and *NodeCores*. Both of them are FieldContainers, therefore the aforementioned features apply to them as well. The topology of the scene graph is formed by the Nodes. They are holding information about their children in a list, a pointer to their parent and a bounding box to allow rendering optimizations. Each node can have multiple children, but only one parent is allowed. Therefore a node can only occur once in the graph. The functionality or content of a node is given by a NodeCore. Each node references one core. For instance, a core can hold geometry data to render an object. It can be referenced by multiple Nodes, allowing to efficiently render multiple objects with the geometry data stored only once in memory. [Ger05]

6.1.2 Rendering

To render the content of the scene graph OpenSG uses a *RenderAction*. Actions are a general way to traverse the graph and perform certain tasks on the visited Nodes. For the rendering process the RenderAction collects all Nodes that contain renderable content. During the traversal the action performs optimization task like the skipping of Nodes which are not visible to the viewer. The remaining Nodes are collected within a *DrawTree*. Its purpose is to sort the Nodes by their materials, which consist of OpenGL states. The sorting allows to group together geometry with the same (or similar) states and therefore reduce the time-consuming need for state changes during the rendering. [Ger05]

6.2 Integration of GML Contexts into OpenSG

To integrate the GML context system into OpenSG the FieldContainer *GMLGeometry* was developed. The FieldContainer is a NodeCore that can be included into the scene graph. The GMLGeometry class forms a thin layer around the GML-ContextNode. It integrates a GML context into the rendering system and is used to connect the OpenSG picking with the GML picking system. Via the wrapped GMLContextNode the GMLApplet is accessible and can be used to deliver GML commands to the encapsulated context.

[1]An OpenSG *Node* is declared with a capital "N" to be able to distinguish it from a GMLContextNode, which is simply called *node* in parts of this thesis

The OpenSG system provides the possibility to render a NodeCore multiple times within the scene. It can be referenced by multiple Nodes, where the position of the Node in the scene graph defines the transformation of the core. The geometry data, however, is only stored once within the NodeCore, enabling a memory efficient reuse of the data.

This multiple instancing feature has to be considered in two areas:

1. the view-dependent level-of-detail calculations of the combined B-rep
2. the picking of GML models.

If one GMLGeometry NodeCore is referenced by multiple nodes the view-dependent calculations have to be made at the time where the OpenGL matrix setup is equal to the time the drawing of the model is taking place. The GMLContextNode already deals with this issue. The rendering procedure used for the integration into the GMLApplet describes which steps are necessary to correctly calculate view-dependent render settings (see section 4.2.3.2).

However, multiple instancing raises a new issue concerning the picking mechanism used by the GML. Section 4.2.3.2 describes how the picking mechanism is working with the GML context extension. For each context the active OpenGL modelview matrix is stored during the render process of the model. The matrix is then used to setup the correct OpenGL environment for the picking mechanism of the GML. With the multiple instancing feature of OpenSG it is not sufficient anymore to store one matrix with a GMLContextNode, as this node can be rendered multiple times with different matrices. The Grimaldo node system currently has no native support for multiple instancing and is not capable of storing multiple matrices with a single GMLContextNode. A way has to be found to store multiple modelview matrices within one GMLGeometry, each related to one Node referencing the GMLGeometry core.

The solution to solve the picking issue with multiple instancing is the introduction a new data structure, the *GMLInstanceRenderer*. A GMLInstanceRenderer is capable of rendering exactly one GML model in the scene. It is holding a pointer to the GMLContextNode and has an additional data member to store a 4x4 matrix. This matrix is used to save the OpenGL modelview matrix at the time the model is rendered. Therefore, the GMLGeometry NodeCore is not directly responsible for the rendering of a model. It is a management unit containing a number of GMLInstanceRenderer. The number depends on the amount of Nodes referencing the NodeCore. Each time a new Node is referencing the NodeCore it is adding a new entry to an internal list. To detect a new Node the `changed()` method of a FieldContainer is utilized. Whenever a data member of a FieldContainer is changed its `changed()` method is called with a *BitMask* as parameter that can be used to identify the changed data member. In this case the mask is used to determine if the list of *parent* nodes changed. If a new parent is added a GMLInstanceRenderer is created and the GMLContextNode pointer hold by the GMLGeometry is set as "render object". The GMLGeometry internally contains a list that maps a Node to a corresponding GMLInstanceRenderer. A new mapping pair is created and added to that list. So, if one GMLGeometry is referenced three times in the scene, three GMLInstanceRenderer are present, establishing a direct mapping of the Nodes referencing the NodeCore with one instance of a GMLInstanceRenderer. Figure 6.1 shows the mapping of Nodes to GMLInstanceRenderer objects.

6.2.1 Rendering

The GMLGeometry NodeCore is derived from the *MaterialDrawable* FieldContainer. This special FieldContainer is intended to represent drawable geometry. It provides a *Material* pointer and a `drawPrimitives()` *functor* that is responsible for drawing the geometry onto the screen. To integrate the NodeCore into the rendering process another functor, the *rendering functor*, has to be registered with the RenderAction. When a MaterialDrawable instance is entered during the pre-draw traversal of the RenderAction this rendering functor is called. The default implementation of MaterialDrawable provides the `renderActionEnterHandler()` method as rendering functor. In the default implementation of MaterialDrawable it registers its material pointer and the `drawPrimitives()` functor with the RenderAction. When the drawing of the geometry is done the given material is activated and the `drawPrimitives()` functor is executed to draw the geometry onto the screen. [Ger05, page 32].

To coordinate the rendering of the GMLGeometry NodeCore a custom implementation of the `renderActionEnter-Handler()` functor is registered with the RenderAction. The registration takes place in the static `initMethod()` of GMLGeometry. Listing 6.2 shows the necessary code. There the `renderActionEnterHandler()` of the GMLGeometry NodeCore method is registered as *functor* in the RenderAction. The `renderActionEnterHandler()` method receives the RenderAction as input parameter. The action is used later on to identify the Node that references the GMLGeometry.

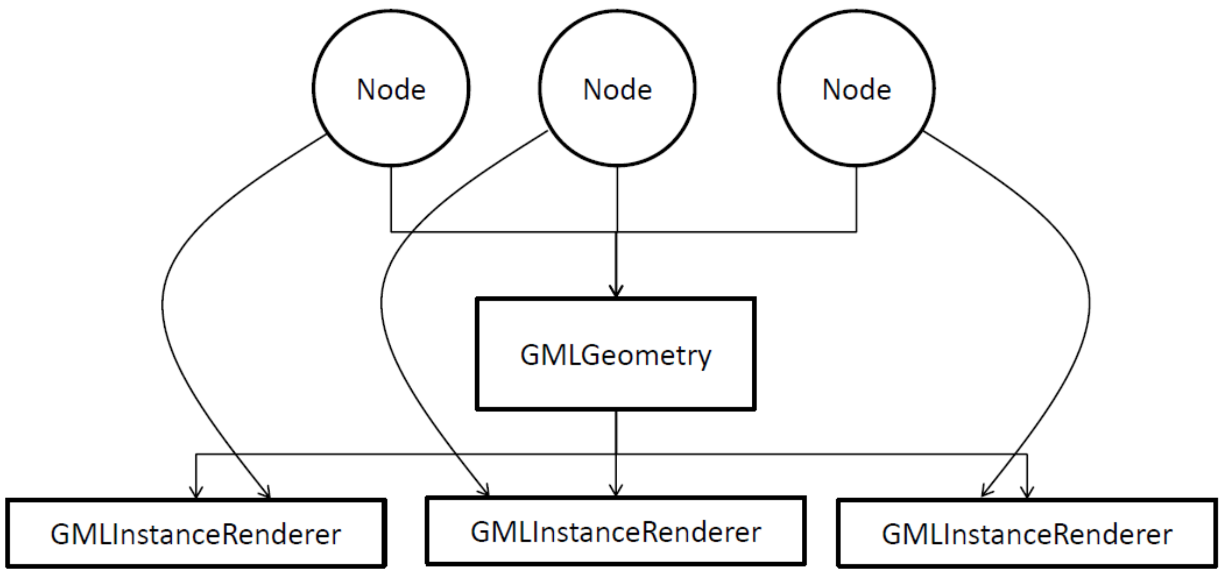


Figure 6.1: Mapping of Nodes to GMLInstanceRenderer objects within a GMLGeometry core

```
void GMLGeometry::initMethod(InitPhase ePhase)
{
   ...

  RenderAction::registerEnterDefault(
    getClassType(),
    reinterpret_cast<Action::Callback>(&GMLGeometry::renderActionEnterHandler));

   ...
}
```

Figure 6.2: Registration of a custom rendering functor

```
class GMLInstanceRenderer
{
public: ...
  void setContext(GMLContextNode *context);

  Action::ResultE draw(DrawEnv *pEnv);

  void saveMatrixStack();
  void pushMatrixStack();
  void popMatrixStack();
}
```

Figure 6.3: GMLInstanceRenderer interface (truncated)

Now, how is the rendering set up within the functor? The RenderAction given to the functor as parameter provides access to the Node that is currently traversed with the `getActNode()` method. The Node pointer is used to search for the GMLInstanceRenderer corresponding to the Node in the node pair list. As the GMLGeometry core automatically adds a node pair whenever a new Node is added as parent such a pair has to exist. The GMLInstanceRenderer is then used to set up rendering for the GMLContext node.

Listing 6.3 shows the interface of the renderer. It contains a pointer to the GMLContextNode representing the GML model, which is set with the `setContext()` method. The rendering of the context takes place in `draw()`. This method is added to the RenderAction to draw the GML model when the DrawTree is rendered. The `draw()` method is very simple. It issues a call to `GMLContextNode::redraw()`. Within this method the the view-dependent calculations for the combined B-rep are performed and the GML model is rendered. The second task - and that is the task the GMLInstanceRenderer was introduced for - is a call to `saveMatrixStack()`. There the current OpenGL modelview matrix is saved within the renderer. As stated before, the modelview matrix is then used to enable the GML picking of objects with displaced GML models.

6.2.2 Event Handling and Object Picking

The integration of GML contexts into the runtime of the GML yields a *GMLSceneViewer* object described in section 4.2.3.2. The GMLSceneViewer is integrated into the rendering, event handling and into the object picking process. Its `framePrepare()` method handles pending events, the `findSelect()` method pre-selects a node with an bounding box intersection test and applies the default GML event handling procedure to a hit GMLContextNode.

To make the GML event handling and object picking available within OpenSG a customized scene viewer is derived from the existing GMLSceneViewer for two reasons:

- The `findSelect()` method of the GMLSceneViewer is already integrated into the GML picking process and is executed when a mouse click or drag occurs. However, the implementation depends on the spatial position of the GMLContextNodes. For the OpenSG integration this information is given by the scene graph. Therefore, the implementation has to be overridden to work perform its task directly on the OpenSG scene graph.
- The two methods `framePrepare()` and `frameFinish()` are independent from the structure the GMLContextNodes are organized. Therefore they can be used without modifications.

The custom scene viewer has a further task. It is used to delegate the input events from the OpenSG system to the GML. Two methods are used: `mouse()` and `motion()`. They have the same signature as their OpenSG pendants. Internally the input events are converted into Grimaldo event codes (see section 5.2.3) the GML framework can work with. Then the `handle()` method of the GMLApplet is called. In turn, the scene viewers `findSelect()` method is executed. It has to be adapted to pick the correct Node based on the given pick ray. If the hit node is referencing a GMLGeometry NodeCore it is processed further. The implementation of the adapted SceneViewer is described in the next section.

```
class OSGSceneViewer : public GMLSceneViewer
{
public: ...
   bool mouse(int button, int state, int x, int y);
   bool motion(int x, int y);
   bool keyboard(unsigned char k, int x, int y);

private: ...
   virtual GMLContextNode* findSelect(int x, int y,
                                      GMLEventIO& data);
}
```

Figure 6.4: OSGSceneViewer interface (truncated)

6.2.2.1 An OpenSG SceneViewer

The integration of the GML picking mechanism is implemented in the *OSGSceneViewer* object. Because of the aforementioned reasons it is derived from the GMLSceneViewer. Its interface is shown in listing 6.4.

The methods `mouse()` and `motion()` directly convert the input data into GML compatible form and call the `handle()` method of the GMLApplet, which internally calls the `findSelect()` method. The method simply adapts the functionality from the *GMLSceneViewer::findSelect()* method (see 4.2.3.2 on page 26) into OpenSG structures. Therefore the OSGSceneViewer uses a custom *GMLPickingAction* that is applied to the root node of the scene and collects all hit nodes that reference a GMLGeometry NodeCore. After the traversal the hit nodes are sorted from the nearest hit point to the farest. The sorted list is then processed sequentially. From each node the the referenced GMLGeometry NodeCore is retrieved. Within its list of node pairs the GMLInstanceRenderer for the currently processed Node is searched. The found GMLInstanceRenderer contains the modelview matrix that was stored during the rendering of the model. The matrix is pushed onto the OpenGL modelview stack to prepare the environment for following picking mechanism, that is executed in calling the `handle()` method of the GMLContextNode, which is also stored with the GMLInstanceRenderer. In case an Applet was hit and contains a callback it is set as focus ticket and the processing of the hit nodes is stopped. Afterwards the OpenGL state is restored to not break the setup expected by OpenSG.

6.2.3 Summary

The procedure described in this section to integrate GML contexts into OpenSG can be taken as a general guideline to integrate contexts into similar node-based system. The first step is to map a GMLContextNode to the target node system. Then a custom SceneViewer is derived from the existing GMLSceneViewer and registered with the Grimaldo Controller, which in turn registers it with the GML event handling system. The custom SceneViewer's `mouse()` and `motion()` methods (or similar, depending on the application) are implemented to delegate the host input into GML compatible form. The last step is to implement the `findSelect()` method to adapt the functionality described in the section before. The important thing to remember is to ensure that the OpenGL modelview matrix is set up correctly before the `handle()` method of the GMLContextNode is called.

6.3 GMLController

The GMLController provides the user with an easy to use frontend to connect an OpenSG host-application with the GML environment. It is used to initialize the GML environment and set up the connection to OpenSG. After a successful initialization the GMLController can be used to create GMLGeomtry NodeCores with two different methods described below. It is also responsible for delegating the input devices events, e.g. from a mouse pointer, to the GML system. During the rendering process the GMLController takes care of pre- and post-rendering tasks. Finally, the GML interpreter is accessible via the frontend and can be used to control the scene or manipulate the created GML contexts.

The interface of the GMLController frontend is shown in listing 6.5. The following sections explain the usage of the different parts.

```
class GMLController
{
public: ...
  void gmlInit(const std::string& startup_path,
               OSG::Node* root,
               bool autoAddChild = true,
               OSG::SimpleSceneManager* mgr = NULL,);
  void gmlExit();

  void setTarget(OSG::Node* target);
  OSG::Node* getTarget() const;

  OSG::GMLGeometryTransitPtr createGMLGeometry(const std::string& command, const std::string& name = "");

  bool loadUserlib(const std::string& lib, bool is_string = false);
  bool call(const std::string& cmd, Node* targetN = NULL, bool add_transform = false);

  bool mouse(int button, int state, int x, int y);
  bool motion(int x, int y);
  bool keyboard(unsigned char k, int x, int y);

  void update();
  void finish();

private: ...
  void onCreateNodeCallback(Property* prop);
  void onAddChildCallback(Property* prop);
  void onRemoveChildCallback(Property* prop);
  void onRemoveChildrenCallback(Property* prop);
};
```

Figure 6.5: GMLController interface (truncated)

6.3.1 Initialization

To use GML contexts within OpenSG the GML environment has to be initialized first. This is done in `gmlInit()`. There a GMLApplet is created and added with a list of default Initializer to fill the GML environment with functionality. The creation of the GMLApplet also triggers the creation of the Grimaldo Controller. The OSGSceneViewer is instantiated and registered with the Controller. The viewer is activated so that the GML event handling system is delegating calls to `findSelect()` to this viewer. The method takes 4 parameters:

`const std::string&` ***startup_path*** The initialization of the GMLApplet needs a startup path that points to the location of the file MODELS/TUBS.MTL. This special file is a material database that contains the default material available with the GML. This startup path also denotes the default location that is used when relative paths are used for the `loadUserlib()` method explained in this section.

`bool` ***autoAddChild*** The `autoAddChild` option enables a feature that automatically adds GML contexts created within a script to the scene graph as a Node with a GMLContext NodeCore referencing the created context. The feature is described in section 6.3.2.

`OSG::Node*` ***root*** This parameter denotes the root Node of the scene where Nodes with GMLGeometry NodeCores reside. This Node and its subgraph are taken into account to perform post-render tasks like the calculation for the adaptive tessellation feature of the combined B-Rep.

`OSG::SimpleSceneManager*` ***mgr*** The SimpleSceneManager is needed to calculate a picking ray for the `findSelect()` method of the OSGSceneViewer.

The `gmlExit()` method is cleaning up the data structures created by the `gmlInit()` method. It should be called before `OSG::osgExit()`.

6.3.2 Creating GML Contexts

6.3.2.1 Script-based Creation

Script based creation of GML contexts is available if the automatic addition of GML contexts was enabled with the `gmlInit()` method. In this case the GMLController uses the *Event Subscription System* of Grimaldo to get notified whenever a change to the Grimaldo node-tree is made. The GMLController is connected to the node management operators via the event system, which allows to add functor callbacks to events to react on topology changes of the node-tree. In this section the two functors for creating and adding nodes are described, the functors for removing nodes are similar. The following script demonstrates a simple creation of two nodes:

```
1 usereg
2
3 /gmlcontext node-create !parent
4 /gmlcontext node-create !child
5
6 :parent :child node-addchild
7 node-getroot :parent node-addchild
```

Now, to map the creation process of the nodes to the OpenSG scene graph the two functors `onNodeCreateCallback()` and `onAddChildCallback()` are implemented by the GMLController and registered with the events *Node.create* and *Node.addChild*, respectively. In line 3 the parent node is created. The `node-create` operator sends the *Node.create* event, which triggers the `onNodeCreateCallback()` with a Property encapsulating the newly created GMLContextNode as input argument. Within the `onNodeCreateCallback()` a corresponding GMLGeometry NodeCore is created and added to an OpenSG Node. The next step is to connect the *transformation* Property of the GMLContextNode with the OpenSG system to be able to control the transformation of the model with the operators described in section 4.2.2.2. The GMLController implements the `SyncronizeTransformationCallback()` for this purpose. It is registered as functor with the transformation Property of the GMLContextNode and gets executed whenever the transformation is changed with GML operators. In this case the spatial information is read from the Property and converted to a `Transform` NodeCore, provided by OpenSG. The resulting OpenSG Node structure created within the `onNodeCreateCallback()` functor is shown in figure 6.6. The next step is to store the top Node of the structure into a temporary pool, the `_looseNodes` map. The map relates the GMLContextNode to the top Node. It is necessary to save both nodes to have the information needed to add the Node to the scene graph when the `node-addchild` operator is called. Additionally the top Node is added as a Property to the GMLContextNode. With this information the removal of a node can be handled later on. The complete procedure is repeated in line 4 to create the child node.

In line 6 the child node is added to the parent with the `node-addchild` operator. The `onAddChildCallback()` is triggered with child node as property argument. Within the `node-addchild` operator the parent is already set within the child node, therefore this information is available in the `onAddChildCallback()` functor and can be retrieved with the GMLContextNode's `getParent()` method. If the parent is the Grimaldo root node the OpenSG Node corresponding to the child is directly added to the `_rootN` of the GMLController, which is set with the `setTarget()` method of the GMLController. Otherwise the `_looseNodes` map is searched for the parent and the child is added. The `_looseNodes` map is cleared after the GMLInterpreter finishes the execution of the script.

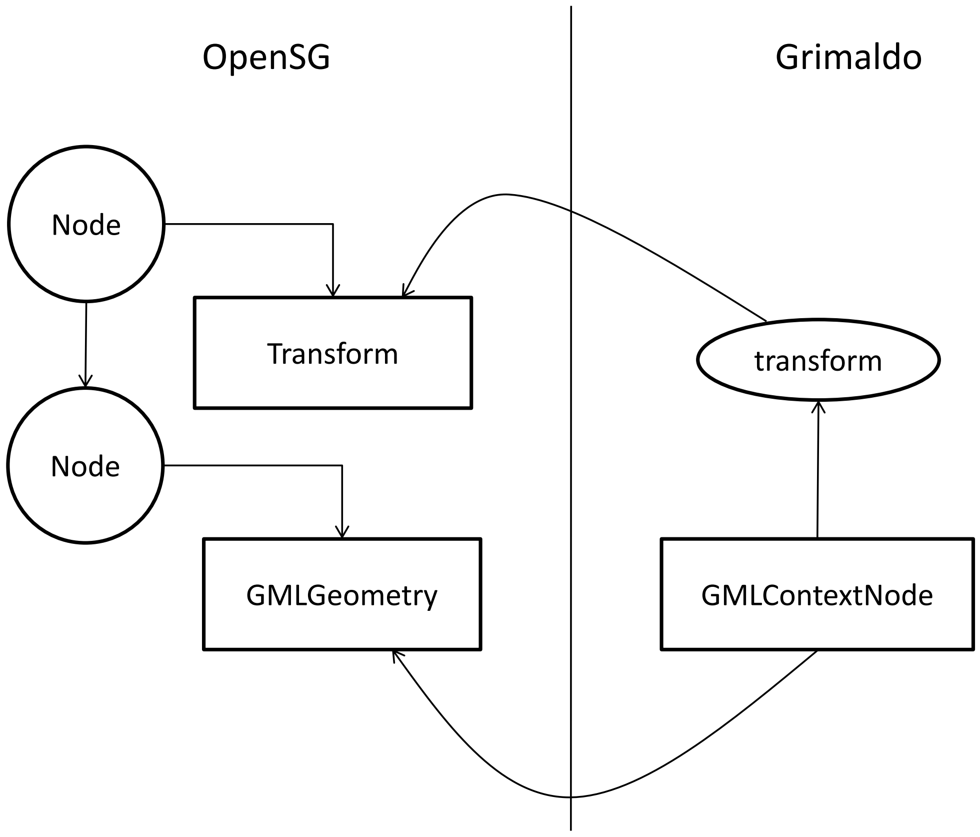


Figure 6.6: OpenSG structure representing a newly created GMLContextNode

There is a second possibility to add a GMLContextNode to the OpenSG scene graph. The work of Björn Gert [Ger05] allows to directly manipulate the OpenSG scene graph. The functionality is available via the osg-namespace GML operators. A new operator was developed that directly takes a *Context* token as input and converts it to a *FieldContainer* token the osg-operators can work with. The operator is called `osg-ctxconvert`. The following code sample show how to convert a GMLContextNode into a FieldContainer and add it to the root node:

```
usereg

/gmlcontext node−create osg−ctxconvert !fc

osg−getroot :fc osg−addchild
```

Grimaldo operators and the osg-operators can be combined, too.

6.3.2.2 Manual Creation

To manually create a GMLGeometry NodeCore the `createGMLGeometry()` method shown in line 13 of listing 6.5 can be used. It creates a new GMLContextNode in using the `createNode()` method of the Grimaldo Controller. The new context is initialized and the GML command provided with the `command` parameter is executed with the new context as active focus context. Additionally a name can be set to identify the GMLContextNode.

6.3.3 Rendering and Event Handling

The GMLController is also responsible for connecting the input event handling of OpenSG with the GML. This is done in using the three methods `mouse()`, `motion()` and `keyboard()` of the GMLController. They have the same signature as their OpenSG pendants, which makes it easy to integrate them. Internally the call to any of these methods is simply forwarded to the OSGSceneViewer, which is described in section 6.2.2.1. The input events are converted into GML compatible events and trigger the internal GML event handling. To handle pending events in each frame a call to the `update()` method has to be made. It internally calls the corresponding method of the OSGSceneViewer to handle pending events. The last step is to enable the view-dependent render optimizations of the combined B-Rep mesh data structure with OpenSG. This is done in calling the `finish()` method of the GMLController. Again, the call is delegated to the OSGSceneViewer which implements the functionality to calculate a quality parameter for the mesh depending on the frame time of the last render step.

6.4 Material Library

The Grimaldo framework provides the possibility to integrate existing material systems with the material system of the GML. This feature is described in section 5.3.3. This section demonstrates the usage of the feature in developing a simple FieldContainer, the *GMLMaterialLibrary*. It maps parts of the default GML materials to OpenSG materials. The intention is to control the shading of different materials of GML models with the OpenSG material system. Materials can be changed at runtime and new materials can be added. The added materials can then be used within a GML script, too.

The GMLMaterialLibrary is created in the initialization phase of the GMLController. It searches for the material definition file `TUBS.MTL` in the `Materials` subfolder of the `startup_path` parameter. The file is loaded with the `loadMTL()` method. Internally a temporary `MaterialIndices` object is created that is responsible for the GML material management. The material definition file is loaded and creates a list with GML material objects. These materials are cycled through and are converted to an OpenSG equivalent. Currently only the *ambient*, *diffuse*, *specular* and *shininess* information is evaluated, resulting in a *SimpleMaterial* object on the OpenSG side. The OpenSG integration described in [Ger05] shows how to convert more sophisticated material types provided by the GML like textures into a corresponding OpenSG material. This work can be integrated in future. Each converted SimpleMaterial is added the name found in the material definition file and a unique ID identifying the material within the GMLMaterialLibrary. This information is the stored within the FieldContainer.

After the materials are converted and stored they can be accessed via the GMLMaterialLibrary. Methods are provided to change existing materials or completely replace them, as well as to add new ones. Although the materials converted from the GML system are of type *SimpleMaterial*, new or replaced materials are not limited to this. Each material type provided by OpenSG can be used.

The next step is to integrate the GMLMaterialLibrary into the shading process of the GML. For this the *GMLMaterial-Handler* object described in section 5.3.3 is used. It provides a hook into the GML rendering process. The hook delegates the control over the material activation before the actual rendering take place to a custom method. In this case the method uses the GMLMaterialLibrary to activate the desired material, afterwards the GML renders the corresponding faces of the model. To achieve this a new object is derived from the GMLMaterialHandler. The *OSGMaterialHandler* implements two mandatory methods of the base class, `getMaterialID()` and `switchMaterial()`. The integration of both methods into the GML material system is described in section 5.3.3. The important part happens in `switchMaterial()`: The method gets an ID as input parameter that denotes the material the GML is going to render in the next step. The ID is used to request the correct material from the GMLMaterialLibrary which is then activated. The following render calls of the GML rendering process are drawn onto the screen with this active material.

6.5 GML Development Environment Support

A GML script consist of plain ASCII text and can be written in any text editor of choice. However, as a script is intended to create a geometric model the development is much easier when changes to the script are visible in realtime. For this purpose the GML comes with an *integrated development environment (IDE)*, that eases the development of GML scripts. [Hav03b]

The IDE contains

- a GML code editor for writing GML code and execute it
- a *Prompt* for direct execution of GML commands
- a library browser for navigation through the loaded GML libraries
- a debug window to examine script errors
- a 3D window that displays the visual output of the script.

With the GML IDE it is possible to quickly develop new scripts. To use the application for developing OpenSG scenes it is necessary to integrate the OpenSG system into the IDE. In combination with the operators for controlling the scene graph described in [Ger05, p. 52] a powerful tool is available for developing OpenSG scenes with GML geometry.

The integration of the OpenSG rendering environment with the technique described in the following section does not depend on the GML IDE, it is only used as an example. In fact, the technique is usable for every application that utilizes the GMLApplet.

6.5.1 A Custom SceneViewer Implementation

The Grimaldo framework allows to use multiple SceneViewer within one Controller. The Controller holds a list of registered SceneViewers. One viewer can be active at a time and is then responsible for the rendering and the input event handling of the scene. This feature can be used to integrate the OpenSG system into the existing GML development environment. The advantage is that the existing IDE can be used to deploy OpenSG scenes with GML models. It is not necessary to develop a separated development environment that supports the OpenSG system (unless the provided functionality is not sufficient, of course).

The idea is that the IDE's 3D window is controlled by the OpenSG system. This involves the rendering and the input device handling, which are exactly the tasks the SceneViewer concept is targeted for. Therefore, a custom SceneViewer has to be implemented that takes care of these two tasks. The viewer is then registered with the Controller and activated to take control over the runtime.

For the new SceneViewer - lets name it *PassiveOSGSceneViewer* - the OSGSceneViewer described in section 6.2.2.1 is taken as base class. The OSGSceneViewer is already integrated into the GML's event handling process and this functionality can be reused here. The PassiveOSGSceneViewer is lacking three things necessary to accomplish the OpenSG integration into the development environment:

- The OpenSG system has to be set up
- The rendering process has to be delegated to OpenSG
- The input device events do have to be consumed by OpenSG to enable interaction with the scene.

In fact, the PassiveOSGSceneViewer is *emulating* a complete OpenSG application. The term “Passive” in its name comes from the fact that the runtime loop is not controlled by the application itself, but by the GML IDE via the SceneViewer interface.

The first thing to do is to setup the OpenSG environment when the viewer is activated. This is done in its `show()` method. Here the necessary data structures for OpenSG are created. The system is initialized in a call to the `OSG::osgInit()` method. For issuing render calls OpenSG uses a *Window*. It wraps the OpenGL context and the viewports of an application. OpenSG provides a variety of different window types. For instance, a *GLUTWindow* is used to create a rendering environment via the GLUT library [GLU]. In this case the GML development environment creates an OpenGL context that can be used for rendering on its own. OpenSG provides a *PassiveWindow* that does not create its own OpenGL context but expects a valid context to be active at the time OpenSG renders the scene. Therefore, a PassiveWindow is created, as well as a SimpleSceneManager responsible for interaction with the scene. Then the root node of the scene is created an added to the SimpleSceneManager. At this time the OpenSG system is set up and the connection to the GML is established in creating an instance of the GMLController described in section 6.3. The instance is initialized in adding the created root node and the SimpleSceneManager to the GMLController. After these steps the environment is prepared to render the OpenSG scene graph.

The Grimaldo Controller allows to switch between registered SceneViewers, which leads to a repeated call of the `show()` method if the viewer is activated more than once. To prevent double initialization of the OpenSG system the method takes care of only performing the described steps once.

To trigger the rendering of the OpenSG the PassiveOSGSceneViewer’s `frameDraw()` method has to be implemented. Following the instructions in section 6.3.3 it first calls the `update()` method of the GMLController to handle pending events. The actual rendering of the scene is initiated by the SimpleSceneManager’s `redraw()` method. To allow the GML to process finishing actions for the frame the `finish()` method of the GMLController is called afterwards.

The next step is to delegate the input device events from the GML development environment to OpenSG, which takes place in `handle()`. The method receives an InputEvent object that contains the data for the current event. Depending on the event type the `mouse()`, `motion()` or `keyboard()` handler of the GMLController are called, which are delegating the event to the GML system. In case of a mouse or motion event internally the SceneViewers `findSelect()` method is called by the GML event handling process, which is the reason for deriving the PassiveOSGSceneViewer from the OSGSceneViewer. In case the event is not consumed by the GML it is delegated once more to the corresponding methods of the SimpleSceneManager, which allows to rotate, pan and zoom the scene.

6.5.2 Dynamic Switching Between SceneViewers

To make use of the PassiveOSGSceneViewer within the GML development environment it has to be registered with the Grimaldo Controller first. The registration is done with the `registerViewer()` method. It takes the pointer to the SceneViewer as well as a name for the viewer as argument. With this name the viewer can then be activated in using `activateViewer()`, which takes a string argument denoting the desired viewer. In this case the name “PassiveOSGSceneViewer” is chosen.

To take advantage of the GML’s dynamic nature the Grimaldo resource described in section 4.2.1 is extended with a new operator: `viewer-activate`. Its input parameter is simply a name token. The operator uses the Controller’s `activateViewer()` method to activate the given viewer. The default viewer for the GML development environment is the built-in SceneViewer or the GMLApplet (see section 4.2.3.2) with the name “GMLSceneViewer”, which is active when the application starts. A call to

```
/PassiveOSGSceneViewer viewer−activate
```

now activates the PassiveOSGSceneViewer in calling its `show()` method. If this is the first time of activation the OpenSG system is set up and connected to the GML. After the activation the 3D window of the development environment shows the OpenSG scene. To switch back to the default viewer the command

```
/GMLSceneViewer viewer−activate
```

reactivates it and displays the content of the Grimaldo node-tree again. This scenario shows the flexibility of the SceneViewer concept. It provides the possibility to establish different *views* onto a scene and to dynamically switch between them. The feature can be used in various ways, one example is outlined on page 47.

7 Discussion and Future Work

7.1 Discussion

This section discusses strengths and weaknesses of the implementation described in this thesis. The section is subdivided into two parts, first points related to the Grimaldo framework and its GMLContextNode implementation are listed, followed by a discussion of the points related to the OpenSG integration.

7.1.1 Grimaldo Framework

This thesis presents the combination of a general framework for visual applications with the power of the GML environment. The framework uses the shape modeling engine of the GML in providing a GMLContextNode that encapsulates a GML model and its scripting facility to enable dynamic control of Grimaldo components.

One strength of the presented framework is its data management system. The ability to store arbitrary C++-objects with components allows a flexible way to extend the functionality and adapt it to the needs of the host-application. The callback mechanism of Properties is used to inform interested entities in case the content of the Property is changed. The mechanism allows to ease data synchronization and to react on data changes in general. A custom Property implementation - the *FakeDictProperty* - provides the possibility to directly register data from within a GML script with a host-application. The callback mechanism is not limited to the notification of a data change alone. Arbitrary signals can be defined on one Property.

The usage of Properties to implement the Grimaldo *event system* is another strength of the framework. The system is used to export internal events of the GML environment. Currently events related to the node management process are exported and can be used to synchronize the node-tree topology with an external system.

With the implementation of the GMLContextNode as node type of the framework external applications can integrate GML models and use them to enrich their functionality. The management of nodes is structured in a node-tree. It can be uses as a standalone storage for nodes, as well as a base to map nodes to other structures, like the OpenSG scene graph system.

The framework provides a sound integration into the GML scripting environment. A GML resource is used to expose parts of the framework as operators to make its functionality available within the dynamic scripting environment of the GML. The resource can be used as base for future developments to expose more features of Grimaldo into the scripting environment. New operators can easily be added, as well as new types.

However, there is a lot of work and enhancements to be done in some areas. One of these areas is the current rendering of GMLContextNodes, especially the combined B-Rep Applet. The GML system uses the *MaterialIndices* data structure to keep track of the materials used by the models. Each of the materials holds a list of faces that are assigned to that material. To minimize the OpenGL state changes during a single render call the material list is iterated. Each material is activated and the corresponding faces are rendered. With the integration of GMLContextNodes into the GMLApplet the rendering process is changed. Each GMLContextNode contains its own version of the MaterialIndices structure (the material library itself, however, is shared between the contexts). The GMLRenderAction traverses each node and triggers a render call. In this call the aforementioned method is used to render the mesh, as well as the remaining Applets which do not necessarily use the MaterialIndices structure for their render process. The problem is that for each node the list of materials is sequentially activated and the corresponding faces are rendered. This leads to an unnecessary amount of OpenGL state changes, which reduces the render performance, especially if many models are visible in a scene.

The rendering of the node-tree is lacking another feature typically implemented by scene graph systems: the culling of nodes outside the viewing frustum. To enable this feature the GMLRenderAction has to be extended. The lack of a culling mechanism is moderated by the fact that the combined B-Rep data structure integrates a culling mechanism. However, the computation for the mechanism is done for each GMLContextNode, even if it is known to be outside the viewing frustum.

Another issue is that temporary Applets are currently not context-aware. For instance, the `iogetkey` or the `iocapturemouse` Applets can not be registered with a distinct model. It is only possible to register them once as global keyboard or mouse handler, respectively. To change this behavior temporary Applets do have to be stored with the context they are created in. For the usage of the `iogetkey` Applet that means that before keyboard events are delegated to a model it has to be set as the focus context to activate the correct temporary Applet set.

7.1.2 OpenSG Integration

The section above discusses the unoptimized rendering process of the Grimaldo framework concerning the GMLContextNode. The same problem occurs with the OpenSG integration of GML contexts. The rendering functor of the GMLGeometry NodeCore is calling the render methods of the GMLContextNode to draw the model onto the screen. As there is currently no possibility for a GMLContexNode to separately render faces with a specific material the OpenSG support for this feature can not be used.

The current integration of GML contexts into OpenSG is targeted to be used on a single host system. OpenSG provides sophisticated possibilities to render a scene on more than one host, which is called *cluster rendering*. At this stage of development there is no cluster support for the GMLGeometry NodeCore. To make the current implementation of the GMLGeometry NodeCore capable of being rendered on multiple servers a way has to be found initialize a GMLApplet instance on all used render servers. The OpenSG system supports to synchronize data types to multiple servers in implementing a special data type for the GMLApplet. With a GMLApplet on all servers the GML commands issued on the input host of the cluster environment - the *client* - can be distributed to the servers. This results in a equally configured GML environment on all servers, which allows to display GML models in a cluster environment. The problem with this approach is the integration of the GML picking mechanism. If the user picks an visual element of a GML object with an attached callback the callback gets executed and may change the appearance of the model. To trigger this change on all servers the callback has to be distributed to the servers, too. This issue is still open and has to be solved.

Another issue is the storing of a GMLGeometry NodeCore to a binary file. OpenSG provides the possibility to store the content of the scene graph into a file and restore it from there later on. For the GMLGeometry NodeCore this means to identify the parts of the implementation that have to be stored to a file to enable a reconstruction of the GML model. One possibility is to store the content of the FakeDictProperties assigned to a GMLContextNode to this file, as well as the initial command to create the GML model. If the GML creation code solely uses FakeDictProperties as parameter for the model creation it is possible to restore the model. However, if the model's appearance is changed in picking and dragging a visual slider and the change is not reflected in the FakeDictProperty parameter the change would not be saved.

7.2 Future Work

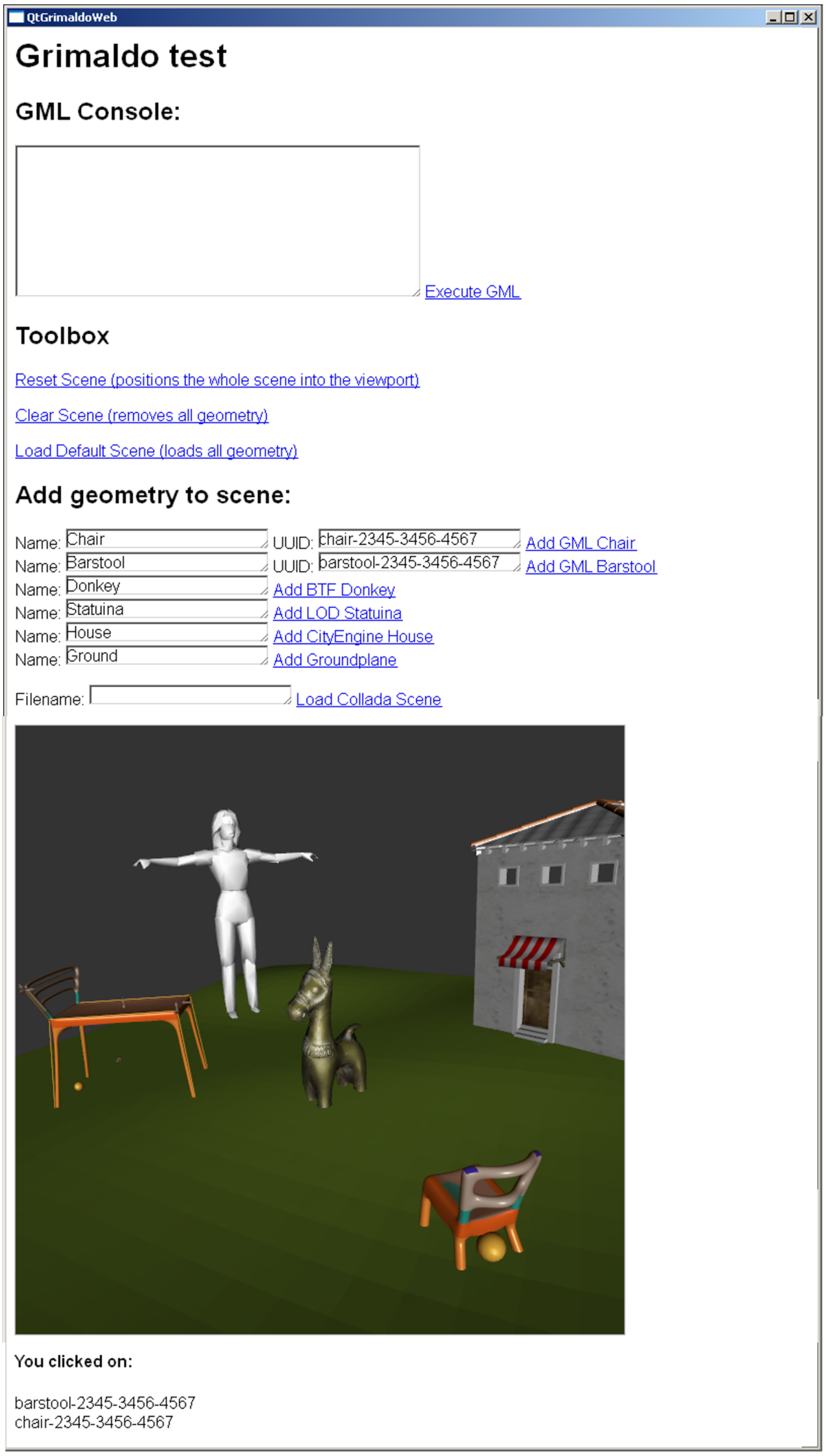
Multiple instancing The Grimaldo framework currently does not natively support multiple instancing of GMLContextNodes. This feature would be beneficial for the efficient rendering of multiple nodes referencing the same model. To render multiple nodes with the same geometry currently two separate nodes have to be created and the same set of GML commands has to be called on both nodes, which results in a double allocation of memory for the set of needed resources. The resources are stored with a node in using Properties. To represent two nodes with the same geometry it is sufficient to create one node with the resource set contained in Properties. Then the second node is created. This time it is not equipped with a new set of resources but the resources are *linked* from the first node. The result is that two nodes are stored within the node-tree and can be treated as independent models. However, their resource set is shared and exists only once in memory. To enable multiple instancing two steps are necessary. The first step is to extend the PropertyGroup object with the functionality to link Properties between PropertyGroup instances. To link one Property into another group a Property with the same name is created in the second group object. However, the content object contained within the first Property does not get copied, only a pointer to the object is stored within the second Property. The second step is to ensure the correct memory management of linked Properties. In the example before two nodes where sharing the same set of resource Properties. What happens if the first created node gets removed? In this case its resources are deleted and the second node's Properties are containing pointers that do not exist anymore. To circumvent this a reference counting system for Properties would be necessary. Each time a Property gets linked to another group the reference count of content object is increased. If a Property gets deleted and the reference count of the content is higher than one only the counter is diminished by one. The object does not get deleted. The reference counting ensures that the object only gets deleted if no Property is referencing it anymore. To implement the multiple instancing feature on the level of PropertyGroups has the advantage that not only nodes can use it, but objects derived from the PropertyGroup interface in general.

Optimized GMLRenderAction In section 7.1 the problem of the unoptimized rendering process for combined B-Rep meshes is described, which results in more OpenGL state changes than necessary during one rendering call. The solution to this problem is the extension of the GMLRenderAction to be capable of registering rendering functors to a specific material, the same technique that is used by the OpenSG rendering system. Each GMLContextNode has a single instance of the MaterialIndices data structure that assigns each face of the mesh to a material. If the GMLRenderAction traverses a GMLContextNode the node has to register a material with the Action and a functor responsible for rendering the faces assigned to this material. This has to be repeated for each material used by the model. Although each node contains its own version of the MaterialIndices object the used material library is the same for all nodes. Therefore the models are using the same materials internally. After the render action finishes the node-tree traversal it contains a list of materials, each with functors responsible for rendering a set of faces. Each material is now activated and the functors are used to render the actual faces. This method optimizes the rendering performance and can also be used with the OpenSG integration. The work in [Ger05] shows a similar approach for efficiently render the combined B-Rep mesh. However, Applets that do not utilize the MaterialIndices data structure for the rendering have to be considered explicitly.

A QtGrimaldoWeb

The Grimaldo framework is used in the EU-project 3D-COFORM. The project has the aim to "establish 3D documentation as an affordable, practical and effective mechanism for long term documentation of tangible cultural heritage." [DC] Grimaldo contributes to the *Visualization Support Library (VSL)*, which is using the OpenSG scene graph library to display cultural heritage artefacts. The VSL library consists of multiple plugins responsible for displaying the 3D-COFORM datasets. One of these plugins is the GML integration into OpenSG described in this thesis. The plugin is used to render GML models as well as to script the scene graph for dynamic creation of virtual scenes.

Figure A.1 shows the prototype of a web-browser plugin named *QtGrimaldoWeb*, which is capable of displaying 3D content via the VSL library. The browser directly communicates with the Grimaldo framework with a Javascript interface. A user is provided with links to load different artefacts which are then displayed in the 3D window. GML commands can also be directly entered to control the OpenSG scene. The screenshot shows a set of loaded artefacts, amongst them two chair GML models. The models have identification IDs attached which are revealed when clicking on them. The output is shown at the bottom of the page. The IDs are used to connect the object to a database with relevant information to the object. The communication channel for the identification daa between Grimaldo and the Javascript interface is established in using Properties.



Grimaldo test

GML Console:

Execute GML

Toolbox

Reset Scene (positions the whole scene into the viewport)

Clear Scene (removes all geometry)

Load Default Scene (loads all geometry)

Add geometry to scene:

Name: Chair UUID: chair-2345-3456-4567 Add GML Chair
Name: Barstool UUID: barstool-2345-3456-4567 Add GML Barstool
Name: Donkey Add BTF Donkey
Name: Statuina Add LOD Statuina
Name: House Add CityEngine House
Name: Ground Add Groundplane

Filename: Load Collada Scene

You clicked on:

barstool-2345-3456-4567
chair-2345-3456-4567

Figure A.1: The QtGrimaldoWeb website

B Notification Interface

The *notification interface* is a tool to add an *invoker - receiver* mechanism to an object. With so called *signals* the object defines a set of invoker objects to which an external application can bind receiver objects.

This technique is popular for years in windowing toolkits for graphical user interface (GUI) programming. In this context it provides a generic way to transmit user action to the host-application. Such user actions are mouse clicks, keystrokes or similar events. Whenever a user clicks a button or uses other tools of the windowing system the underlying application is notified and can trigger actions according to the notification origin. This mechanism is an effective way to decouple the windowing system from the host application logic. In [GHJV94] the technique is described as the *Command* design pattern.

The *NotificationInterface* object allows a client to define custom signals. Callback functions - called slots - can subscribe to a signal. With the `setObject` method the notification interface allows to connect an object that can then emit signals. The object itself does not know anything about the surrounding event-system as the system is simply *attached* to the object. When a signal is emitted via the*emitSignal*-method the subscribing slots are executed with the object given as argument.

The notification interface technique offers a straightforward mechanism to add a signals to an arbitrary object in using the template mechanism of the C++ programming language. Using this generic programming technique results in the declaration of the `NotificationInterface` object shown in listing B.1.

```
class NotificationInterface
{
public:
  template<class T>
  void setObject(T* object);

  void addSignal(const std::string& name);
  bool emitSignal(const std::string& name);

#ifdef WITH_BOOST
  typedef boost::function<void (T*)> SlotCallback;
  bool subscribe(const std::string& signalname,
                 Slot callback)
#else
  typedef SlotCallback<T> Slot;
  bool subscribe(const std::string& signalname,
                 Slot* callback)
#endif

  private:
    T* m_object;
    std::map<std::string, Signal*> m_signalMap;
}
```

Figure B.1: NotificationInterface (truncated)

A slot can subscribe to a signal in two ways:

- In using the Boost library's `boost::bind` functionality.
- In deriving from a `SlotCallback` base object that provides an pure virtual *onExecute* method that has to be implemented.

The usage of Boost [Booa] for binding callbacks to signals allows a flexible subscription mechanism that supports pointer to simple functions, pointer to member functions and functor objects in using `boost::bind` to connect a callback. More information on the capabilities of the Boost-way of callback handling can be found in the online documentation of the Boost library (see [Boob]).

The dependency to the Boost library can be disabled, as it might not always be possible in a project to use this library. To disable it the SDK has to be compiled without the `WITH_BOOST` preprocessor flag.

Bibliography

[Ado99] ADOBE SYSTEMS INC.: *Postscript Language Reference Manual.* Addison-Wesley, 1999. 5

[Ale01] ALEXANDRESCU A.: *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Ben03] BENDER M.; BRILL M.: *Computergrafik*, 1 ed. 2003. 3

[BHSF09] BEIN M., HAVEMANN S., STORK A., FELLNER D.: Sketching subdivision surfaces. In *EUROGRAPHICS Symposium on Sketch-Based Interface and Modeling* (2009), Grimm C. J., (Ed.). 10

[Ble] BLENDER3D:. An open source 3D content creation suite, www.blender.org. 39

[Booa] Boost; www.boost.org (accessed November 11, 2010). 44, 75

[Boob] Boost::bind Documentation; http://www.boost.org/doc/libs/1_45_0/libs/bind/bind.html (accessed October 5, 2010). 75

[DC] 3D-COFORM:. www.3d-coform.eu. 72

[FHH03] FELLNER D., HAVEMANN S., HOPP A.: *DAVE - Eine neue Technologie zur preiswerten und hochqualitativen immersiven 3D-Darstellung.* Technical Paper, Technical University of Braunschweig, 2003. 3

[GBHF05] GERTH B., BERND R., HAVEMANN S., FELLNER D.: 3d modeling for non-expert users with the castle construction kit v0.5. In *The 6th International Symposium on Virtual Reality, Archeology and Cultural Heritage* (2005). 3

[Ger05] GERTH B.: *Generative Scene Manipulation in OpenSG.* Master's thesis, 2005. 3, 6, 53, 58, 59, 65, 66, 67, 71

[GHJV94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, November 1994. 39, 42, 74

[GLU] GLUT - The OpenGL Utility Toolkit; http://www.opengl.org/resources/libraries/glut (accessed October 23, 2010). 48, 68

[Han03] HANISCH F.: Wsi/gris, 2003. http://www.gris.uni-tuebingen.de/edu/projects/vis/coursebook/rendering/engine/ (accessed November 26, 2010). 57

[Hav03a] HAVEMANN S.; FELLNER D.: *Progressive Combined BReps - Multi-Resolution Meshes for Incremental Real-time Shape Manipulation.* Paper submitted to COMPUTER GRAPHICS Forum December, 2003. 3

[Hav03b] HAVEMANN S.: *Introduction to the Generative Modeling Language.* Version 1.31, Insitute of Computer Graphics, Technical University of Braunschweig, 2003. 5, 67

[Hav05a] HAVEMANN S.; FELLNER D.: *Epoch Future Research Directions.* Position Paper, Technical University of Braunschweig, 2005. 3

[Hav05b] HAVEMANN S.: *Generative Mesh Modeling.* PhD thesis, Institut of Computer Graphics, Braunschweig Technical University Germany, 2005. 1, 5, 6, 8, 20

[Hav10] HAVEMANN S.: *Towards a Lingua Franca for Computer Graphics.* unpublished technical report; Insitute of Computer Graphics and Knowledge Visualization, Graz University of Technology, 2010. 5

[Kried] KRIEGER M.: *Generatives Modellieren in Maya.* Master's thesis, Unpublished. 51

[LGP] LGPL:. GNU Lesser General Public License, version 2, www.gnu.de/documents/lgpl-2.1.en.html (accessed November 11, 2010). 15

[May] Maya, a 3D modeling environment; usa.autodesk.com. 51

[Opea] OpenSG, a 3D scene graph system for virtual reality applications; www.opensg.org. 3, 39, 45, 57

[Opeb] OpenGL Overview, http://www.opengl.org/about/overview (accessed November 11, 2010). 57

[OSGa] OpenSG Features: Performance; http://www.opensg.org/wiki/FeaturesPerformance (accessed November 26, 2010). 57

[OSGb] OpenSG Features: Extensibility; http://www.opensg.org/wiki/FeaturesPerformance (accessed November 26, 2010).

[RVB02] REINERS D., VOSS G., BEHR J.: *OpenSG: Basic Concepts*. Paper submitted to 1. OpenSG Symposium, 2002. 58

[VBRR02] VOSS G., BEHR J., REINERS D., ROTH M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization* (2002), Bartz D., Pueyo X.,, Reinhard E., (Eds.). 58

[WBKS05] WILLIAMS A., BARRUS S., KEITH MORLEY P., SHIRLEY P.: *An efficient and robust ray-box intersection algorithm*. Proceeding, SIGGRAPH '05 ACM SIGGRAPH 2005 Courses, 2005. 19

[WNDD99] WOO M., NEIDER J., DAVIS T., D. S.: *OpenGL Programming Guide*. Addison-Wesley, 1999. 26

List of Tables

List of Figures