Abstract

We present a framework for formulating domain-dependent 3D search queries suitable for content-based 3D search over the web. Users are typically not willing to spend much time to create a 3D query object. They expect to quickly see a result set in which they can navigate by further differentiating the query object. Our system innovates by using a streamlined parametric 3D modeling engine on both client and server side. Parametric tools have greater expressiveness, they allow shape manipulation through a few high-level parameters, as well as incremental assembly of query objects. Short command strings are sent from client to server to keep the query objects on both sides in sync. This reduces turnaround times and allows asynchronous updates of live result sets.


Keywords: 3D search interfaces, rapid shape design, domain-dependent modeling tools, generative modeling, GML.

1 Introduction

The great success of content-based search and retrieval has become obvious with textual documents: Google & Co. allow looking for texts simply by specifying words that appear in these texts. This is a great paradigm change compared to the long-standing practice in libraries. Traditionally a librarian ingests a book who understands the text and assigns appropriate subjects and keywords typically chosen from a controlled standard vocabulary. In every library the subject index permits retrieving relevant text documents efficiently if, and only if, also the researcher is acquainted with the controlled vocabulary, or can use a thesaurus containing keyword synonyms. So neither assigning keywords nor retrieving texts is trivial. Content-based retrieval is easier for casual users, and it is also very transparent: Only documents that contain the query string are returned. The difficulty is the ranking, e.g., when thousands of documents match a query. Sophisticated ranking algorithms are used, e.g., to infer the intended context from the word combination in a multi-word query. Relevance feedback improves the precision by logging the results that are selected most often.

Such levels of sophistication can only be achieved when content-based search and retrieval are available in the first place. But this is difficult for multimedia documents: Images, videos, music, drawings, and 3D models can be understood as generalized documents. Very large digital libraries of such multimedia documents are manageable only if services for markup, indexing, and retrieval [Endres and Fellner 2000] are available. However, to formulate a content-based multimedia query users have to create an image, a video, or some music etc. For 3D-documents, the user must provide a 3D model to formulate, e.g., a query-by-example. It turns out that “formulating” such a query can be a difficult problem.

1.1 Formulating a 3D-query

Query-by-example requires a query object: When looking for chairs, the user has to provide a 3D model of a chair. Unfortunately, users typically look for models that they do not have. In this case, a new query model needs to be created, which leads to the difficult 3D modeling problem: High-end modeling toolkits such as Blender, AutoCAD, 3D-Studio Max, Maya etc. are out of scope for average or casual users. 3D modeling tools for non-expert users, e.g., Google SketchUp [Google 2008], provide a small but well-defined set of modeling tools and are easier to learn. But even with those tools creating a query model takes significantly longer than, e.g., typing in a text-based query.

1.2 Domain-dependent Parametric 3D Query Objects

3D search technology is generally better in discerning classes of objects (furniture, cars, animals) than in discriminating the objects within a class (Regency vs. Art Deco style, dog vs. cat, car model), as pointed out by [Bustos et al. 2007]. Shape subtleties are more difficult to express and take longer to sketch, so designing a query object requires more advanced skills. This problem can be solved using class- or domain-dependent parametric modeling tools:

- Complex query objects are easier to create and to control using only a small number of high-level parameters,
- Subtleties can be expressed if there are specific parameters or modeling tools for them,
- Search is more specific, e.g., for the number of floors of a building, the slope of a stairway.
Domain-dependent query objects can vary in flexibility (see Fig. 1) to reflect the trade-off between flexibility and ease-of-use:

- **Configurable objects** are parametric objects with a fixed number of parameters,
- **Hierarchical configurations** have a fixed structure with configurable sub-features
- **Free design** has the fewest restrictions but may require a certain level of modeling skills.

1.3 Immediate Feedback

It can be frustrating for users investing time to sketch a detailed query object when the query yields a bad or un-intended result set. Immediate feedback is desirable to guide the creation process. The desired search result might show up before the query object is thought to be finished (cf. Google suggest), or the result set indicates that a query model is developing towards an unintended class of shapes. Both can reduce user frustration.

Maintaining a “live result set” is technically demanding. The query object is typically a mesh that is sent to the server whenever it is changed. This can occur several times per second (slider ticks) if a high level of responsiveness is desired. One way to reduce traffic is to transmit only changed model parts, ideally just the change commands. The server creates its own copy of the query object that is kept in sync with the client object.

Updates are easy in the object configuration scenario, as the parameter set is sufficient to specify the query. In the free design scenario it is important that all object modifications are faithfully transmitted, otherwise the query objects on client and server get out of sync.

1.4 Integration with an Existing 3D Search Framework

Assuming a technology for domain-dependent parametric shape queries exists, and that it is possible to do give immediate feedback with a live result set, the next challenge is the integration of the query front-end with the actual shape retrieval back-end (search engine). The shape search engine typically has technical requirements in terms of, e.g., the web service framework or communication standard to use. So the query interface must be very flexible.

1.5 Feature Wish List

To summarize, the system for formulating 3D shape queries has the following requirements:

- 3D user interface on the client, integrated with web browser
- Parametric query objects, to reduce network load
- Possibility to use domain-dependent 3D objects for queries
- Immediate feedback through live result sets
- Easy front-end integration with existing 3D search back-ends

We are describing such a system in this paper, and we argue that no existing system fulfills all of these requirements.

2 Related Work

Although our system does not directly process any free-hand sketches, it is related to sketch-based 3D interfaces in the spirit of Teddy [Igarashi et al. 2006] and its sophisticated successors, e.g., published in the Eurographics Sketch-Based Interfaces and Modeling (SBIM) workshop series. Sketch-based interfaces work well for skilled non-experts and are therefore also used for searching. The interesting approach from [Lee and Funkhouser 2008] follows a *Modeling By Example* approach, which is primarily useful for refining search results, e.g., to search for an object composed of parts from other models. Our framework differs in that we aim at domain-dependent modeling tools and parametric objects; it would be difficult to search for a car rim with a certain number of spikes in their system.

An approach for sketching of procedural models was presented by [Schmidt and Singh 2008], who store direct surface manipulations in a surface tree where nodes contain re-usable modeling operations such as sharpen contour, make bump etc. Another approach from [Yang et al. 2005] is to use sketch recognition to infer from the sketch the type and parameters of a parametric 3D modeling tool; they use primarily sweeps and surfaces of revolution. An-
other approach, which could also be integrated with our system, is the sketch and retrieval of LEGO models from [Santos et al. 2008]. This is an example of a domain-dependent search tool that is nevertheless broadly applicable due to the "blockiness" of many 3D models. However, the difference is that we aim at providing a general framework that can incorporate many of such domain-dependent techniques for rapid creation of parametric models, instead of focusing on one particular technique.

Our work is also related to Web 3D since our models to be transferred over the internet, in fact fast enough for live updates of the result set. In principle, each of the sketching techniques mentioned above could be used in a web 3D context. Unfortunately, web-based parametric modeling has not received very much research attention. VRML/X3D for example provides only limited 3D modeling capabilities, mainly just sweeping (X3D::Extrusion) [Brutzman and Daly 2007]. Unfortunately, this limitation rules out also Ajax3D from www.ajax3d.org, which provides asynchronous communication that is very useful for live result sets. Another alternative is Papervision3D, an open source 3D engine for Adobe Flash. Flash benefits from a large user base, but it also suffers from very limited 3D modeling capabilities.

In summary, there are very few alternatives for 3D modeling toolkits suitable for an integration into a web browser.

3 System Overview

The 3D query formulation system has two main components, a parametric modeling engine running on the client, and a client-server communication layer to connect the client to an existing 3D search back-end. Both components are introduced in this section, and explained in greater detail in the following two sections.

3.1 GML-based Parametric Modeling Engine

The Generative Modeling Language (GML) is a general shape description language for procedural models [Havemann 2005]. It provides modeling operations for polygonal and free-form shapes, which are adaptively tesselated on-the-fly. GML allows for very compact model descriptions, which is especially useful in a web context [Berndt et al. 2005]. Its suitability for 3D modeling by non-expert users was also assessed [Gerth et al. 2005].

The central concept is the stream of tokens known from PostScript: A GML program consists of individual tokens that are executed one after another. Tokens either contain data (literal token), which are put on the stack, or processing instructions (name, operator, function). Names can be literal (\(\text{xyz}\)) or executable (\(\text{xyz}\)). Executable names are looked up in the dictionary stack, where each dictionary contains a list of (name, Token) pairs. A name lookup can lead to a built-in operator (\(\text{add}, \text{sub}, \text{mul}\) etc.), which is executed. It can also lead to an array of tokens. Arrays are either literal or executable. A literal array is just put on the stack like other literal tokens. An executable array is the same as a functions. If a name lookup leads to an executable array, it is executed token by token (function call).

3.2 The PROBADO Search Engine Framework

PROBADO is a cooperative digital library project funded by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG). The main goal is to integrate generalized (multimedia) documents, in particular music and 3D models of architectural buildings, into the workflow of existing libraries [Krottmaier et al. 2007]. The rationale behind PROBADO is that academic, technical, and even public libraries will soon have to deal with masses of multimedia documents. As explained before, multimedia-documents do not fit well into the existing library workflow: No librarian can assign keywords to all the documents to ingest, so automatic approaches such as content-based retrieval are indispensable. However, also future libraries will be in charge of quality control: Users expect to find high quality documents in a library.

The layered PROBADO architecture and the integration of the GMLSearchService are shown in Fig. 2. PROBADO provides a special SOAP message (DoContentSearch) for query-by-example requests. Even though OBJ (Alias/Wavefront) is an ASCII format for polygonal 3D models, SOAP requires the model to be base64 encoded, which increases the data size by 25%. The SOAP message is created by a web service with integrated GML interpreter that tesselates the GML model, converts it to OBJ, and retrieves the search results from PROBADO. The results are returned to the client for presentation with Silverlight and embedded GML applet.

Obviously the web service containing the server side GML interpreter should be located on the same machine as the 3D search system, since this is the communication bottleneck. Unfortunately, this is not always possible: PROBADO does not only allow decentralized data repositories but also de-centralized search engines. Not all search engine providers would allow the GMLSearchService to be installed on their machines. – This underpins again the necessity to develop more space-efficient encodings for shape.
### 3D Modeling in a Web Browser

This section demonstrates using a set of parametric modeling tools, the SketchLite toolkit, for the rapid creation of 3D query models. The toolkit falls into the free design category, as the user can decide where to apply which tool. It is mode-based, one modeling tool is always chosen as the current tool; if no modeling tool is active, the user can navigate in the scene. The toolkit is scripted, the script is interpreted by the GML plugin on the client side. Running as a web browser plugin it displays an OpenGL context embedded in a HTML page. The 2D GUI needed, e.g., for choosing the current modeling tool and for entering numerical values, is part of the surrounding HTML page.

The plugin receives all commands through JavaScript. The challenge is to keep the interface between plugin and JavaScript very generic; it must not depend on any particular set of modeling tools. Furthermore it must allow bi-directional communication since model changes, e.g., dragging on a 3D slider, might have to be displayed in the 2D GUI, and parameter changes in the 2D GUI must have an effect in 3D.

The following section shall convey a thorough technical understanding on how the modeling functionality is realized on the client side. The goal is an enabling technology for a new generation of domain-dependent 3D query formulation interfaces.

#### 4.1 The ActiveGML Plugin

On Microsoft Windows, the whole GML modeling functionality is contained in a single ActiveX control called ActiveGML. ActiveX controls can be directly used as browser plugins in Microsoft Internet Explorer (IE). However, since the Gecko Plugin API uses a very similar communication approach, ActiveGML can be recompiled into a plugin for all non-IE browsers, i.e., Firefox, Opera, Safari, Chrome, basically by changing some compiler flags. GML is implemented in C++ and uses only standard OpenGL, so this works also on Linux. A port to MacOS X is under development.

The advantage is that ActiveGML is compatible with standard HTML/JavaScript toolkits like DOJO [Dojo 2008] or the Yahoo! User Interface Library (YUI) [YUI 2008], which provide more appealing 2D GUI elements. Furthermore, ActiveGML can also be used with frameworks for Rich Internet Applications such as Microsoft Silverlight [Silverlight 2008], Adobe Flex [Adobe 2009], or JavaFX. The plugin is simply embedded using the object tag (or embed for non-IE). The following HTML/JavaScript code tells ActiveGML to load a GML script and to execute its main method:

```html
<object width="600" height="600" id="ActiveGML" classid="clsid:A7BB...3E29E"
    type="application/generic−modeling">
</object>
<script>
    var theControl =
        document.getElementById("ActiveGML");
    theControl.Call("LoadLibrary("SketchLite.xgml")");
    document.getElementById("ActiveGML").Call("SketchLite.main");
</script>
```

The first version of ActiveGML presented in [Berndt et al. 2005] supported only two methods, LoadLibrary and Call. The bi-directional communication with JavaScript required adding two new methods (GetStackSize and GetStackElementAsString) and one GML operator (call-javascript). They are described next:

- **LoadLibrary(string)** – Loads a GML file from a web resource or a local file.
- **Call(string)** – Executes GML commands. The next example shows how GML code is executed through a JavaScript event attached to a HTML image acting as button:

```html
<img src="static−resources/icon Extrude" alt="Extrude" onclick="javascript :ActiveGML "
    Call(\"SketchLite.modeling.extrude\")">
```

GML supports path expressions. SketchLite and SketchLite.Modeling are dictionaries, the latter contains the extrude function to select the extrude tool as current modeling tool.

- **GetStackSize()** – Returns the number of tokens that are currently on the GML operand stack.
- **GetStackElementAsString(index)** – Returns the string representation of a stack element. The following JavaScript code displays two alert boxes, the first displaying “1”, the second “3”, which is the result of executing 1.2 2.1 add in GML.

```javascript
function callSomeGMLCode() {
    var theControl =
        document.getElementById("ActiveGML");
    theControl.Call("clear,
                    .Call("MyFoo", 1, 2.1, 3);"
                );
    alert(theControl.GetStackSize());
    alert(theControl.GetStackElementAsString(0));
}
```

The new GML operator call-javascript is the necessary hook back from the plugin to the surrounding web browser. It is also realized in a very generic way, as it expects just two tokens on the stack:

- Data to be passed to the javascript function (string converted)
- Name of a javascript function (name)

The following GML code calls the JavaScript function MyFoo with the string 

```
[ (2,4,6) /hallo ]
```

as argument. Note that the MyFoo function must be defined in the HTML page, otherwise an error results.

```
[ (1,2,3) 2 mul /hallo ] /MyFoo call−javascript
```

#### 4.2 Structure of the Modeling Toolkit

The SketchLite toolkit is inspired by the push/pull approach used, e.g., in Google SketchUp [Google 2008]. It presents itself as a library of GML functions to activate the respective modeling tools. Every tool essentially modifies the mouse callback functions of the GML 3D window, so that faces can be pulled or extrusions applied.

Fig. 3 shows the GML code to insert a new box into the existing scene. It is activated when the new box icon is clicked; the next click on the grid produces the new box. The icapturemouse operator (line 22) expects as input on the stack a GML function (in curly brackets) that is to be executed at every mouse event. Just before this callback function is executed, the GML runtime system sets the following variables to describe the mouse event:

- **mouseEvent** – integer: 1 - push, 2 - drag, 3 - release
- **mouseButton** – integer: 1 - left, 2 - right, 3 - middle button
- **pNear** – 3D point: mouse position on the near plane
- **pFar** – 3D point: mouse position on the far plane

The pick ray is the line segment between the 3D points pNear and pFar. The intersection point of the ray with the ground plane is calculated using the intersectLinePlane operator (line 10). Then a polygon (array of 3D points) is created for the bottom side of the box, and it is translated to the pick point (map operator applying the function in 13-16). The polygon is converted to a mesh face which is extruded (17-18). Finally, the registered mouse callback is removed, to keep the next mouse click from creating another box.
Other modeling operations (extrude, move face, etc.) require intersecting the pick ray with existing geometry. The rayintersect operator returns the halfedge of the mesh face that is hit. Internally, all mesh modeling operations use halfedges. GML provides also low-level mesh operators, the Euler operators such as makeEF to split a face or makeEV to split a vertex [Mäntylä 1988]. Euler operators are typically not directly exposed in a user interface, but are used to create modeling operations that require fine-grained mesh access.

```plaintext
userereg
ioremoveall % remove any active mouse callbacks
dict
{
  userereg
  mouseButton 1 eq
  SketchLite.UI.History.newMacro % for undo/redo
  % compute intersection with grid plane
  pNear pFar % line pNear to pFar
  (0,0,1) 0 % (n,d) of the plane
  intersectPlane pop !intersectPoint
  % box bottom polygon: square of (0,0,1)×5 units
  [ (0,0,0) (5,0,0) (5,5,0) (0,5,0) ]
  {
    % bottom center to (0,0,0)
    intersectPoint add % move to intersection
    map % apply to each polygon point
    3 poly2doubleface % polygon to double-sided quad
    (0,5,3) extrude pop % extrude quad, discard edge
    ioremoveall % remove this mouse callback
  } if
locapturemouse pop
```

**Figure 3:** GML code to register a callback function that handles a mouse event for creating a new box.

### 4.3 Unlimited Linear Undo / Redo

The GML modeling engine internally keeps track of all Euler operators. In fact, all higher-level modeling tools implemented in C++ are also based on Euler operators. The resulting Euler sequence can be split in sub-sequences called Euler macros. Since Euler operators are invertible, undo/redo of Euler macros is possible. The advantage is that the cost for undo is independent from the length of the modeling history, which is not the case with history-based undo (i.e., replay all recorded modeling operations except the last one). Whenever the user uses a modeling tool, SketchLite starts a new Euler macro (using SketchLite.UI.History.newMacro, line 6) and stores it so that it can be un- and re-done. Linear undo/redo is especially useful in combination with live result set, since the user sometimes needs to backtrack when results become “wrong”.

### 4.4 Modeling Tools of the SketchLite Toolkit

Available modeling tools are represented as small icons to the right of the 3D widget. In case no tool is active, the user can rotate / pan / zoom the model with the mouse. To activate a tool a small string of GML commands is sent to the ActiveGML plugin through a simple JavaScript function:

```javascript
function ActivateTool(command)
  { document.getElementById("ActiveGML").Call(command); }
```

The following list shows the modeling tools available for the rapid design of 3D query objects, and the respective GML commands. Most tools can be deactivated using a right-click to switch back to camera navigation mode.

- **New Block** – SketchLite.UI.newBox
  After clicking on the icon the user can place a new block. The next left-click on the ground grid specifies the box position.
- **Delete Block** – SketchLite.UI.deleteBox
  After selecting the tool, the user can delete a box (including all operations applied to it, i.e., extrude, move face, etc.).
- **Extrude** – SketchLite.modeling.extrude
  The next selected face is extruded in face normal direction.
- **Move Face** – SketchLite.modeling.moveF
  The next selected face is moved along in normal direction. In contrast to extrude this does not add any new geometry.
- **Move Edge** – SketchLite.modeling.moveE
  A halfedge identifies one face, one edge, and one vertex of the mesh. The next selected edge is moved in normal direction of the selected face.
- **Move Vertex** – SketchLite.modeling.moveV
  The next selected vertex is moved in normal direction of the selected face.
- **Smooth Edge** – SketchLite.edge.toggleSharpness
  The next selected edge is smoothed (see Fig. 5b). The edge color reflects the sharpness status (red: sharp, green: smooth).
- **Smooth Face** – SketchLite.face.sharpness_0
  All edges of the next selected face are set to smooth (Fig. 5c).
- **Sharp Face** – SketchLite.face.sharpness_1
  All edges of the next selected face are set to sharp.
- **Undo** – SketchLite.UI.History.undo
- **Redo** – SketchLite.UI.History.redo
  As explained in the text, full undo/redo is supported for all modeling operations.

**Figure 5:** (a) box with only sharp edges (b) box with one smooth edge (c) box where one face has only smooth edges

Figure 4 shows the step-by-step creation of a simple armchair with only eleven tool applications. Each operation is essentially one mouse click to activate the tool, one to apply it, and possibly a bit of dragging. Sometimes a tool can remain activated because all faces to be selected are visible (e.g., in step 4d,e).

This model has been submitted as a query-by-example into the PROBADO 3D Search Engine. The results returned for this particular query are shown in Fig. 6. With techniques for asynchronous partial updates of web pages, like for instance Ajax, the result-set can be updated after each modeling step. This way the user gets immediate feedback on how the shape modifications influence the result set. This is in fact one way of navigating or browsing through an extensive database of 3D objects.
5 Client-Server Communication for 3D Searching with Live Updates

The 3D modeling session described so far takes place only on the client computer. How is the query model transmitted to the 3D search service, and how are the search results returned? The GMLSearchService uses a service-oriented architecture (SOA) to transfer the 3D model, which consists only of a character string, a stream of GML tokens, from client to server. Currently SOAP 1.1 [Box et al. 2000] is used as communication protocol, but other protocols such as REST would be equally suitable.

The communication is initiated on the client side. A Silverlight control implements the SOAP client as well as the presentation logic for the result set. On the server side a web service implemented in ASP.NET acts as SOAP server. It is hosted on an Internet Information Services (IIS 7) running on Windows Server 2008. Both client and server are only loosely coupled and can – of course – be replaced by components that use other platforms or SOAP APIs.

5.1 Asynchronous Updates: Silverlight, AJAX & Co.

The Silverlight control on the client side is responsible for transmitting the GML code from the GML plugin to the web service. It is initiated by the GML operator call-javascript to call a JavaScript function. Here the JavaScript engine acts as a communication bridge between the ActiveGML plugin and the Silverlight control.

One outstanding feature of GML is the ability to assemble new code at runtime. As explained in section 3.1, a GML program is only an (executable) array of tokens. In fact the only difference between a literal array (which is pushed on the stack when executed) and an executable array (which is executed token by token) is the literal array and an ex-

The load operator retrieves a named object without executing it, the array operator creates an array, and cvx stands for convert to executable and simply changes one flag. – Even functions that are generated this way can be converted to strings of GML code. So whenever the user modifies the 3D model within the ActiveGML control, a small GML function describing the modification is assembled, containing typically only one line of code:

```
polygon 1 array { SketchLite.modeling.polybox } arrayappend
"ModelUpdateCallback" call javascript
```

The assembled function is sent via call-javascript as character string to some Javascript function ModelUpdateCallback. This function is supposed to pass the code string on to a Silverlight control running on the same HTML page. It might be as short as this:

```
function ModelUpdateCallback(updateData) {
    document.getElementById("GMLClientSearch").Content.GMLSearch.ModelUpdate(updateData);
}
```

5.2 Adapting Existing Parametric GML Models

Parametric GML models are typically not developed for 3D queries. Given a suitable parametric model, how can it be used as domain-dependent 3D query frontend? The key observation is that interactive GML models obey a typical structure (Fig. 7). As explained earlier, GML dictionaries can serve as name spaces. But dictionaries can also be seen as objects in the object-oriented sense, i.e., as a container for data and functions. It is even possible to switch dynamically: with either definition, \( x \) evaluates to 2.2:

```
/x 2.2 def
/x { 1.0 1.2 add } def
```

In Fig.7, the dict operator creates a new empty dictionary and begin puts it on top of the dictionary stack so that it receives all subsequent definitions, e.g., the function definitions in the “class body”.

```
1   dict dup begin
2   /CalcParams { ... } def
3   /UpdateModel { ... } def
4   /OnMouseButton1_Push { ... } def
5   /OnMouseButton1_Drag { ... } def
6   /OnMouseButton1_Release { ... } def
7   CalcParams UpdateModel
8   } def
9   end
10  register - controller
```

The end operator (line 9) pops it off the dictionary stack; but due to the dup in line 1 it remains on the operand stack. register-controller
stands for a typical function that takes the dictionary and registers its respective functions as mouse callback functions. In this example the parametric model is regenerated whenever MouseButton1 is released: First the user input is converted to model parameters (e.g., by processing the pick ray), and then the model is updated to reflect the parameter change. The domain-dependent modeling functionality is encapsulated in the UpdateModel function.

```javascript
1  /OnMouseButtonDown_Release {
2     CalcParams UpdateModel
3     ParamsToArray { UpdateModel } arrayappend
4     "ModelUpdateCallback" call–javascript
5     def
```

Figure 8: Modification of the mouse callback function to trigger asynchronous updates of the live result set.

By adding only two lines of code (and one function) this example can be adapted to the client-server framework (Fig. 8). Line 3 calls a new function ParamsToArray that simply puts the model parameters into an array, and appends the function call UpdateModel to the newly generated function. Line 4 sends it as string to the JavaScript function ModelUpdateCallback. In this example only the release event triggers a model update. In general, the GML application is responsible for the choice of events that are sent to the server, for instance release, pick-release, or pick-drag-release. The latter will result in very frequent model updates on the server, and it will also considerably increase the data throughput between client and server, and between server and 3D query engine (transmission of tessellated model).

5.3 Serverside GML

SOAP requests issued by the client are handled by a web service. The services it offers and their input- and output parameters are accessible through a description in Web Service Description Language (WSDL) format. Our query framework requires only four functions to realize 3D searching in a very generic way:

- **StartNewSession** – Starts a new modeling session and returns a Universally Unique Identifier (UUID, a 128 bit number) as session identifier. It is used for the other methods to identify the correct session. One user can have several search sessions in parallel, e.g., when using tabbed browsing.

- **EndSession** – Ends an active modeling session. The only parameter is the session ID. Another way to close a session is by a timeout due to a long period of inactivity, typically five minutes. This is important to release server resources.

- **ProcessGML** – Receives and processes the transmitted GML token stream. Parameters are the session ID and a character string containing GML commands, i.e., tokens separated by whitespaces. There is no return value.

- **FetchResults** – This method is called by the client to retrieve the current result set. Parameter is the session ID, the return value consists of an array of 3-tuples (id, title, url) where id is a unique result identifier (string, URL, or UUID), title is the displayed name, and url points to a thumbnail of the 3D model. To obtain this result set, the current 3D model is sent to a 3D search engine by exporting a tessellation of the GML model to a file or a stream, which is quite fast.

For each modeling session the web service creates one server-side GML interpreter to execute the received GML code and create a mesh that is identical to the one in the client. Fig. 9 shows two intermediate steps from the armchair sketching example.

6 Results and Discussion

The client side of the resulting system is shown in action in Fig. 10. The search results during a modeling session are frequently updated, almost every ProcessGML call is followed by a call to FetchResults. The problem that the result list is still waiting for the response while the user continues modifying the model is solved by using a queue of ProcessGML messages sent to the server. The next update of the results (from the last request in the queue) is not requested before the pending FetchResults has returned. But all model change message are of course faithfully applied to the server-side GML model to keep it in sync.

As we expected we found that the latency between the client and the GMLSearchService is greatly reduced by sending instead of a mesh only a parametric model, and to send this even just incrementally. The amount of data sent between the client and the server is drastically reduced. GML allows generating any desired number of triangles on the server with a very short sequence of tokens when using complex domain-dependent models.

A comparison of SOAP message sizes between the GMLSearch client/GMLSearch server and the GMLSearch Service/PROBADO framework is shown in Fig. 11. The size of the response is not listed in the table, since it depends on the length of the IDs, titles, and thumbnail URLs. The size of messages sent to PROBADO can suddenly explode, especially with modeling operations that produce larger number of polygons, such as the smoothing of the armrest in step (e). However, as expected the size of the messages containing the GML token stream remains more or less constant.

The interface of the GMLSearchService is very generic: The ProcessGML method will accept basically any GML code as input for the server side GML interpreter. This loose binding between the client and server provides great flexibility. However, in a production environment it is for security reasons not acceptable to let clients send GML code to the server for execution. For this reason a strongly typed message interface with a static set of fields may become necessary. This can be achieved by sending the GML calls...
of all modeling tools, such as SketchUpLite.NewBox(position), as individual SOAP messages. This can help to avoid GML-injection attacks, a similar threat as SQL-injection.

The main bottleneck in the current implementation is the SOAP communication between the GMLSearchService and PROBADO. Integrating the service directly into the PROBADO repository layer would eliminate the extra traffic for sending the OBJ file and, thus, drastically speed up the response time. In the optimal case the tessellation is created directly within the search engine.

### 7 Conclusion and Future Work

The presented framework opens many possibilities for future research. The most promising perspective is that we can now develop and test new approaches for domain-dependent 3D search engines. As mentioned before, classical 3D-retrieval is very good at discriminating between shape classes. Domain-dependent tools obviously work best when used within a class of similar models. So both approaches ideally complement each other.

We envisage a system where very generic 3D search formulation tools are used in the beginning of a query, for instance also incorporating approaches like [Lee and Funkhouser 2008] or [Yang et al. 2005]. Once the intended domain becomes clear, the user interface switches to the appropriate set of domain-dependent modeling tools. It should be clear that technically this is not a problem with our framework. This would be the combination of the best of the two approaches.

Another interesting avenue of research is to evaluate the modeling sessions for the query objects. They give valuable insight for improving the modeling tools, and might also help solving the question of how average people deal with shape creation in general.

### 8 Acknowledgements

We gratefully acknowledge the support from the German Science Foundation. The PROBADO project started in February 2006 with a tentative duration of five years. Partners are the University of Bonn, Darmstadt Technical University, the German Library of Science and Technology in Hannover, and the Bavarian State Library in Munich. Cooperation partners are Graz University of Technology and Braunschweig Technical University. For further information, please refer to the PROBADO website at [http://www.probado.de](http://www.probado.de). PROBADO is founded by the German Research Foundation under grant no. (554975 (1) Oldenburg, BIB48 OLof 01-02)

### References

<table>
<thead>
<tr>
<th>Step</th>
<th>request (GML)</th>
<th>request (PROBADO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session start</td>
<td>268</td>
<td>-</td>
</tr>
<tr>
<td>(a) new box</td>
<td>405</td>
<td>1158</td>
</tr>
<tr>
<td>(b) flatten</td>
<td>405</td>
<td>1158</td>
</tr>
<tr>
<td>(c) extrude 1</td>
<td>382</td>
<td>1443</td>
</tr>
<tr>
<td>(c) extrude 2</td>
<td>383</td>
<td>1728</td>
</tr>
<tr>
<td>(c) extrude 3</td>
<td>382</td>
<td>2014</td>
</tr>
<tr>
<td>(d) extrude 1</td>
<td>382</td>
<td>2311</td>
</tr>
<tr>
<td>(d) extrude 2</td>
<td>383</td>
<td>2608</td>
</tr>
<tr>
<td>(d) extrude 3</td>
<td>382</td>
<td>2014</td>
</tr>
<tr>
<td>(e) smooth 1</td>
<td>376</td>
<td>9808</td>
</tr>
<tr>
<td>(e) smooth 2</td>
<td>375</td>
<td>16710</td>
</tr>
<tr>
<td>Total</td>
<td>4123</td>
<td>41359</td>
</tr>
</tbody>
</table>

Figure 11: Size of the SOAP requests (in bytes) for the armchair modeling example.


Brutzman, D., and Daly, L. 2007. X3D: Extensible 3D Graphics for Web Authors. Morgan Kaufmann. 3


Google, 2008. Google sketchup. 1, 4


Lee, J., and Funkhouser, T. 2008. Sketch-based search and composition of 3D models. In Eurographics Workshop on Sketch-Based Interfaces and Modeling. 2, 8


Santos, T., Ferreira, A., Dias, F., and Fonseca, M. 2008. Using sketches and retrieval to create lego models. In Eurographics Workshop on Sketch-Based Interfaces and Modeling. 3


Yang, C., Sharon, D., and van de Panne, M. 2005. Sketch-based modeling of parameterized objects. In Eurographics Workshop on Sketch-Based Interfaces and Modeling. 2, 8

YUI, 2008. The yahoo! user interface library (yui) [http://developer.yahoo.com/yui/]. 4