An Object-Oriented Approach to Model Scenes of Buildings

Thierry Matthey Hanspeter Bieri

Research Group on Computational Geometry and Graphics
Institute of Computer Science and Applied Mathematics
University of Berne, Neubrückstrasse 10, 3012 Berne, Switzerland
E-mail: matthey,bieri@iam.unibe.ch

Abstract

This paper presents an object-oriented approach for describing and modelling buildings by means of computer graphics. The approach is based on an object- and component-oriented graphics framework. Scenes of buildings are conventionally modelled by a set of primitives and some aggregates. The present approach, however, uses highly abstract objects which represent certain parts of real buildings. These objects form together a high-level hierarchical construction set and are responsible for providing decompositions into primitives where necessary. Dependencies and side effects are handled by the abstract objects themselves. First results show that the new approach leads to simple and flexible representations without losing the broadness necessary to describe real buildings. However, as its main application domain is urban planning, it does not attempt complete photorealism.

1. Introduction

Modelling and rendering the appearance and dynamics of real-world scenes seems to be largely solved nowadays. In particular, architectural scenes are easy to model with current geometry-based methods like CAD. Many efforts to model architectural scenes have produced impressive pictures, walk-throughs and fly-throughs.
Since at the time increased computer power allows to represent scenes not only as simple wireframes, geometric modelling by computer has become more and more important for all kinds of visualization purposes. Urban planning and archaeology require similar possibilities of computer modelling and visualization as architecture. Entertainment and simulators have increased the interest in realistic renderings in real time.
Modelling realistic scenes depends on a detailed geometry model and realistic textures. Without any effective tools, it is an extremely labour-intensive process. The next three subsections refer to the three main approaches conventionally used to simplify the modelling process.

1.1. Geometry-based systems

Cf. [3, 10]. The user has to position the elements of the scene manually and to survey the accuracy of the model, especially when only photographs are available. Partially, the geometric input can be obtained by converting CAD data or by digitizing architectural plans, if available. But this converting and digitizing process leads to a considerable loss of abstraction, e.g. after digitizing plans the user has to transform pixels to lines and polygons. Especially in the case of large scenes, the lack of structure has several disadvantages. When a scene is updated, e.g. a whole wall of a building is moved, the user himself has to eliminate side effects. Further the representation of scenes is rather memory consuming.
Another problem is that realistic output is only achieved by modelling most geometric details and by suitable textures. After modelling, the scene has to be rendered by techniques like radiosity or raytracing, which are time consuming. For real time applications preprocessing is indispensable.

1.2. Image-based systems

Creating models directly from photographs has received increased interest, since real images as input have the advantage of yielding photorealistic renderings as output. Such image-based systems rely on artificial intelligence and computer vision techniques. The most promising systems, e.g. [8, 6, 11], determine the scene from a set of photographs using stereo algorithms.
But state-of-the-art stereo algorithms have a number of significant weaknesses. The photographs need to be very similar for good results. Therefore, a large set of photographs is needed to reconstruct a whole building, which increases the computing costs. To get acceptable results in reasonable time, interactions by the user are indispensable.

1.3. A hybrid approach

The interesting approach from P.E. Debevec et al. [4] allows to model and render existing architectural scenes starting from a small set of still photographs. This modelling approach combines both geometry-based and image-based techniques.
In a first step a simple model is generated by photogrammetric modelling. The user identifies corresponding lines in the given set of photographs and in the model. Additionally, it is possible to define constraints and dependencies for the primitives, which facilitates the recovery of the basic geometry of the scene. In a second step a stereo algorithm detects how the real scene deviates from the model and also allows to recover additional details. Finally, the rendering process uses the updated model and the textures of the given photographs to generate the output.
This approach produces good results with a minimum of interaction and without the weaknesses of a pure image-based system. However, it is time consuming, and the extraction of textures from the photographs for the view-dependent texture-mapping is a difficult task. The level of abstraction of the scene representation is still low.

Given a suitable input, the three main approaches produce good results in a reasonable amount of time. But they do not really consider the properties and constraints of real buildings, which would lead to more comprehensive, compact and intuitive models. Their biggest weakness is that, after the scene of buildings has been modelled, it is normally difficult to make alterations that exceed local changes.

Below we present a new approach, which leads to simple scene descriptions and easy interactive updating of scenes. Urban planning is the main application domain we have in mind. On the one hand this means that modifications of a scene are frequent. Further they should be accomplished easily and interactively. On the other hand the generated images do not have to be completely photorealistic.

The section below explains the basic philosophy of our approach. Then a simple example is given to illustrate the modelling process. Next, we present our construction set, i.e. the main classes needed to work with abstract objects. Finally, we discuss briefly a number of extensions which could transfer our approach into a applicable working modelling system. For a more detailed exposition we refer to [7].

2. An object-oriented modelling approach

We propose an object-oriented approach based on an object- and component-oriented graphics framework, i.e. an approach that is more than just an implementation using an object-oriented programming language. The geometry-based methods mentioned above use a small set of primitives and some aggregates which are directly handled by the active components or applications themselves. Contrary to these and some other approaches, our object-oriented approach uses highly abstract objects to represent "building blocks" of real buildings and aggregates to model whole scenes. It will be shown that the resulting models are flexible, compact and intuitively appealing.

Thus the main philosophy of our approach consists in working with *abstract objects* that are indeed much more abstract than those conventionally used in geometric modelling. Abstract objects are used generically, i.e. in an application they represent modules or "building blocks" of real buildings. An important part of such a module is normally its geometry, i.e. a set of points in $\mathbf{R}^2$ or $\mathbf{R}^3$ (e.g. a polygon). Abstract objects have properties and a behaviour similar to the real objects they represent. Additionally, they take care of logical dependencies and constraints within the model. They are even responsible for their own concrete representation based on a set of primitives and aggregates. For instance, an abstract object NURBS may be asked to represent itself analytically or by a mesh of triangles, according to the application.

Our abstract objects form together a hierarchic *construction set*. There are two main types of abstract objects. One object, implemented by the class `Building`, has the role of a root and represents the (initial) *empty building*. It defines only the basic geometry, i.e. base and height, and serves as a *container* for the other objects (class `BuildingObject`), which refine the building. The base of a building is always given by a polygon, possibly with holes that define inner courts. For a given height a default building is defined. It consist only of the walls of an extruded polygon (Fig.1).

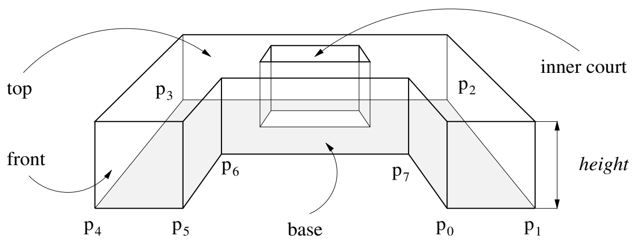

Figure 1. Definition of a default building.

Using an *object-oriented graphics framework* has been essential for our approach, for we need a design and mechanisms which allow a user to introduce and use new abstract objects which cannot directly be handled by an application itself. A typical example is the inclusion of an abstract ob-

ject whose actual representation (e.g. by `Building`) is not understood by the application. In such a case the new abstract object will provide an alternative representation like a decomposition with primitives. The mechanisms which allow such inclusions have been implemented by means of an object-oriented design without any tags and code duplication.
All abstract objects have been implemented starting from a common base class (Fig.2) which supports a small number of well designed protocols. The different levels of hierarchy avoid code duplication and group the abstract objects into logical groups, e.g. aggregates or primitives. Thereby, this design allows the integration of new object types without any influence on other parts.

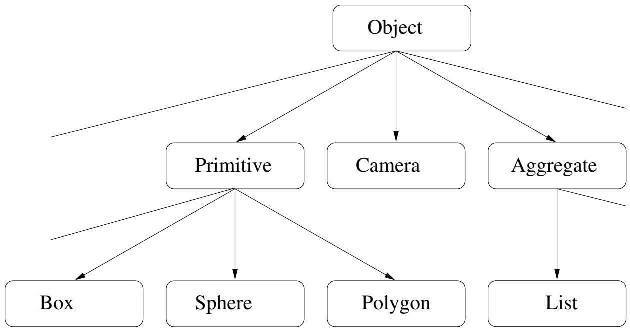

Figure 2. A hierarchy of abstract objects.

2.1. BOOGA

Actually, our approach has been based on the object- and component-oriented graphics framework BOOGA[1], developed by St. Amann and Ch. Streit [1, 2, 12]. BOOGA

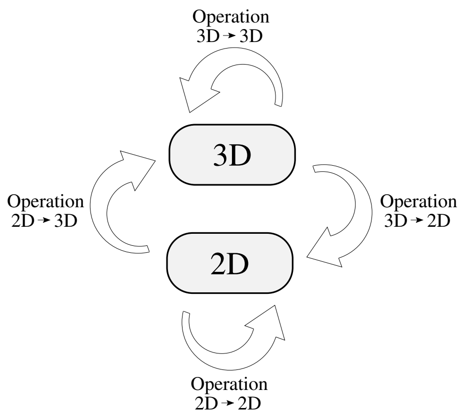

Figure 3. BOOGA's concept of abstraction.

attempts to support all main fields of computer graphics, including image processing and computer vision. It starts with distinguishing between the *2D-world* and the *3D-world* and introduces transitions between them (Fig.3). It provides a library with well-designed base classes (e.g. a general 3D object) and a small set of protocols. The use of design patterns [5] makes the design comprehensible and easy to expand. This framework represents scenes by means of directed acyclic graphs. Aggregates are the structural objects (implemented using the Composite Pattern [5]) of the scene graph. They are responsible for their subobjects. Our abstract objects have only one parent object because of the high dependence on the parent object.
Individual *components* as well as whole applications (i.e. sequences of components) have to visit the nodes of the scene graph (Fig.4) to process every single object. In order to obtain more flexibility, the traversal process has been isolated from the further processing. When an object is visited it is passed to the active component for processing. A component may reject an unknown object, and it then expects this object to react appropriately. It will by default ask the rejected object for an alternative representation. As we have already explained, this philosophy is fundamental for our approach.

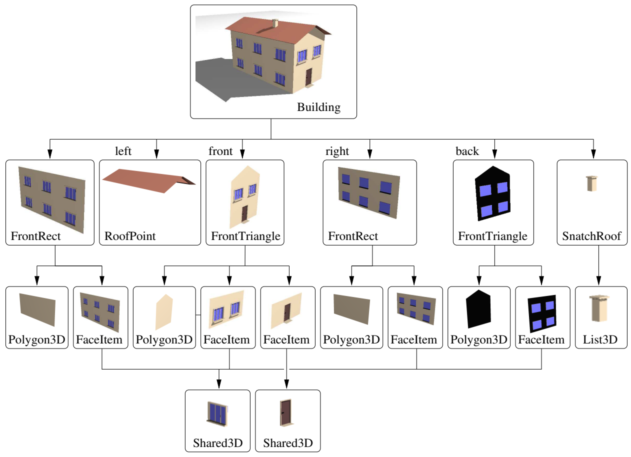

Figure 4. A pruned representation of a house.

Figure 4 shows a typical scene graph of a building and illustrates the construction set. At the top `Building` defines the base and the height. By means of the abstract objects `FrontTriangle` and `FrontRect` the geometry of each front is modelled. The details of the fronts are modelled by means of `FaceItem` and a compound object. `SnatchRoof` is responsible for placing the chimney onto the roof which is modelled by means of `RoofPoint`.

The next section illustrates the modelling process corresponding to our approach by means of a simple example.

[1]**B**erne's **O**bject–**O**riented **G**raphics **A**rchitecture.

3. An example

The following short example shows the construction of a simple house. It is meant to illustrate how an implementation based on our approach can be used to model buildings in practice. In particular, it should indicate the simplicity as well as some of the potential of our approach. The scene, i.e. the house, is specified in BSDL[2] which is similar to the specification language of *RayShade*[9].
In a first step we define the base and the height of the desired house by means of the abstract object `Building`, which is the base and container of the whole building. By default the object (with no subobjects) creates only the walls (Fig.5). Then the geometry of the fronts is defined, i.e. we add front objects as subobjects. We define a flattened ridge with `FrontTriangle` by additional points $[0.3, 1.25]$ and $[0.7, 1.25]$, which are relative to the actual default front (Fig.6). For the sides we use rectangular fronts with the same size as the one of the actual default front.

```
// Geometry of the house
// (height, vertex, vertex,...)
building (20,[-10,-20,0],[10,-20,0],
             [10,20,0],[-10,20,0]){

  // Default texture of the house
  rock;
  // Back front
  fronttri(0,0,[0.3,1.25],[0.7,1.25]);
  // Left front
  frontrect(1,0,0,1);
  // Front front
  fronttri(2,0,[0.3,1.25],[0.7,1.25]);
  // Right front
  frontrect(3,0,0,1);
}
```

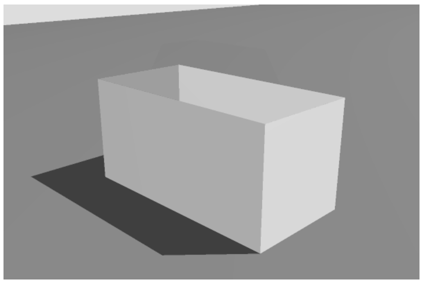

Figure 5. Empty (default-)building.

In a second step, the roof and a simple chimney are defined. For the roof we choose one which covers the whole base of the building and allows additional roof points. To get a more realistic result, a roof is modelled with a horizontal distance $ledge$ by which the roof overhangs the base. The chimney is defined by two boxes and placed properly in the z-direction onto the roof using `SnatchRoof` (Fig.7).

```
building (20,[-10,-20,0],[10,-20,0],
             [10,20,0],[-10,20,0]){
  ...
  // (ledge, point, point,...)
  roofpoint(2,[0, 17,7.5],[0, 17,7.5])
    {redroof;}
  // The chimney at [5,6,z]
  snatch([5,6]){chimney;}
}
```

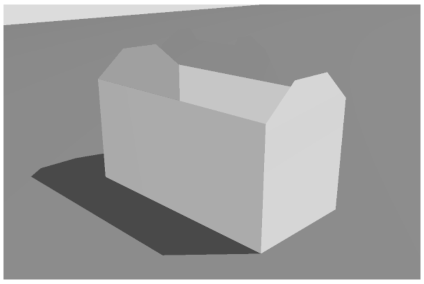

Figure 6. House with fronts.

In a last step, we model a door and the windows, which depend on their parent fronts using `Face` objects (Fig.8). To place them onto the fronts, we use `FaceItem` which allows a to generate copies of a given object and aling them. Additionally, we displace the copies relatively and cut out a suitable hole. The following description models the front of the house with a door and three windows. The other sides are modelled similarly.

```
  // Front front
  fronttri(2,0,[.3,1.25],[.7,1.25]){
    // (from, to, column, row)
    // 2. storey with two windows
    faceitem([0,0.5],[1,1],2,1){
      window;  // Window
      hole;    // Hole for the window
      // Relative displacement
      displacement[0.5,0.4];
    }
    // 1. storey with a door
    faceitem([0,0],[1,0.5],1,1){
      hole;
      door;
      displacement[0.5,0];
    }
    // Top window
    faceitem([0.3,1],[.7,1.3],1,1){
```

[2] **B**OOGA **S**cene **D**escription **L**anguage.

```
    windowSmall;
    hole;
    displacement[0.5,0.3];
  }
}
```

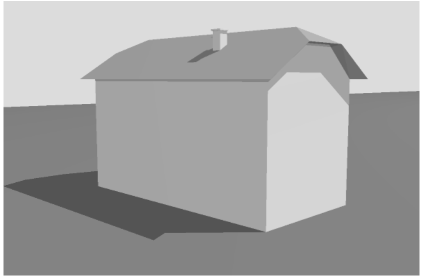

Figure 7. House with roof and chimney.

To refine the house, we attach a border between the two stories using `BottomBorder`. Figure 8 shows the final result. The representation of this scene is similar to that in figure 4.

```
// (ledge, borderwidth, borderheight,
     borderdepth)
bottomborder(0.5,0.5,0.5,0.1){
  height(9); // Elevation from the
             // base of the house
}
```

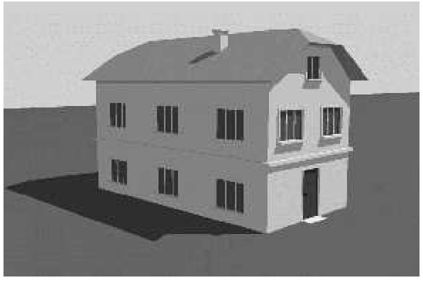

Figure 8. The final house.

4. The construction set

The design of the objects of our construction set is based on the object hierarchy of figure 9. The base is the class `Building` which represents the initial empty building (Fig.1). `Building` is derived from the base class `Object3D` of 3D objects. Its main responsibility is to propagate certain services (e.g. intersection or decomposition) for a client and to provide for the building objects information they asked for.

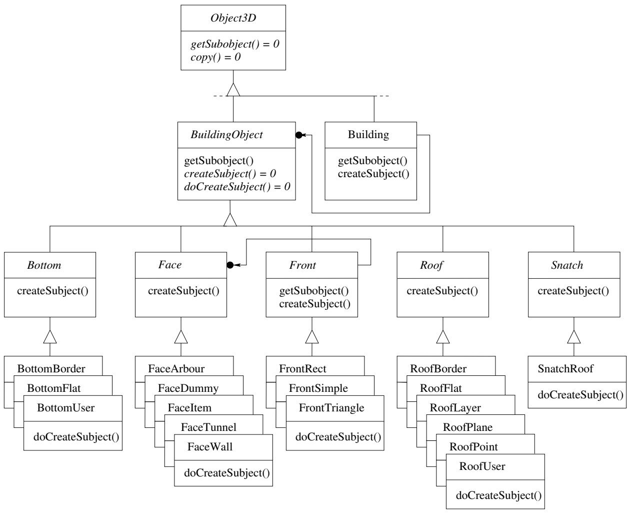

Figure 9. The object hierarchy.

The building objects (`BuildingObject`) are either subobjects of `Building` or `BuildingObject`. They represent parts of real buildings and refine them. `BuildingObject` is the base class of a 3-level object hierarchy and is derived from `Object3D`. It supports protocols to propagate general services and queries and provides a *cache* (cf. proxy pattern in [5]) to avoid redundant recalculations of the concrete representation. The second level groups the abstract objects into logical units and provides for these units a number of services, e.g. the computation of the actual location or to store geometric data. The third level contains the concrete classes which will provide the concrete representations by primitives and aggregates and take care of specific parameters and properties.
These three levels make it possible to reuse code and to introduce new abstract objects easily. The developer has to implement a small number of methods which define the specific properties and their behaviour. For a new object type a general abstract object must be found, which can be a difficult task. In this case it is appropriate to split the abstract objects into several groups. In the following, we present all abstract objects forming our construction set.

4.1. Building

`Building` is the base element of the construction set and serves as a container for the subsequently used building objects. It defines the overall geometry, as shown in figure 1. Optional holes define inner courts. The fronts of the initial building are identified by two indexes. The first index ($frontindex$) defines the front of the base or inner court. The second index ($polygonindex$) defines the base or inner court itself.

`building` $(\mathbf{h}, \mathbf{p_0}, \mathbf{p_1}, \ldots, \mathbf{p_{n-1}})\{$
`hole` $(\mathbf{h_0}, \mathbf{h_1}, \cdots, \mathbf{h_{n-1}})$;*buildingobjects*;}
This specification defines the general geometry of the building with of the vertices $\mathbf{p_i} \in \mathbf{R}^3$, height h and additional inner courts `hole[...]`. All vertices must have the same z-coordinates. If the building does not contain any front objects, it creates by default the appropriate walls (Fig.10).

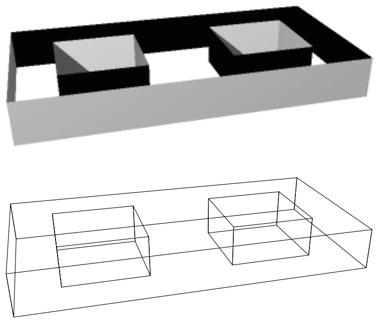

Figure 10. Building with no subobjects.

4.2. Bottom

`Bottom` is the base class and the abstraction of abstract objects which model the base or a floor of the building. They depend on the geometry of the base and the inner courts. The three concrete classes `BottomBorder`, `BottomFlat` and `BottomUser` are subobjects of `building`.

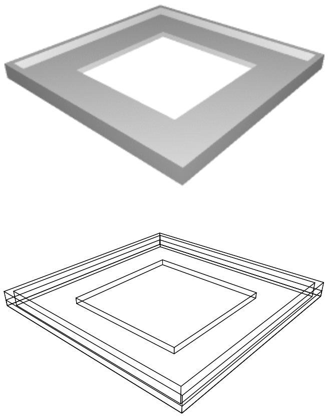

Figure 11. Example of `BottomBorder`.

`bottomborder`
$(ledge, borderwidth, borderheight, borderdepth)$
Creates a border representing a balcony (Fig.11) around the base and takes care of all inner courts.

`bottomflat`
Creates a simple polygon as the base of the building.

`bottomuser` {*anObject*;}
Is a simple container for a general object and allows to model a user-defined base or floor.

4.3. Roof

`Roof` is the base class and the abstraction of abstract objects which model several kinds of roofs and depend on the geometry of the building. Additionally, `Roof` provides the service to define a roof ledge which overhangs the base with a horizontal distance $ledge$. The six concrete objects `RoofBorder`, `RoofFlat`, `RoofLayer`, `RoofPlane`, `RoofPoint` and `RoofUser` are subobjects of `building`.

`roofborder`
$(ledge, borderwidth, borderheight, borderdepth)$
Creates a border as `BottomBorder` does, but adjusted to the top of the building.

`roofflat` $(ledge)$
Creates a flat roof strictly based on the given geometry of `Building` (Fig.12).

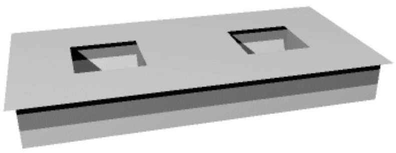

Figure 12. `RoofFlat`.

`roofplane` $(ledge, \alpha)$
Creates a roof with constant inclination $5° \leq \alpha \leq 85°$ and a given roof ledge (Fig.13). The roof does not consider the geometry defined by front objects.

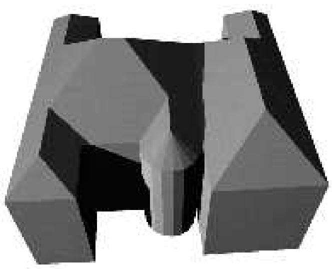

Figure 13. `RoofPlane`.

`rooflayer` $(\mathbf{l_0}, \mathbf{l_1}, \mathbf{l_2}, \ldots, \mathbf{l_{n-1}})$

Creates a roof defined by layers which form the skeleton (as the contours of a terrain) of the roof (Fig.14). A layer is defined by a vector $\mathbf{l_i} = (ledge, height, flatness)$. $ledge$ defines the horizontal distance overhanging the base. $flatness \in [0,1]$. With $flatness = 1$ the layer depends only on the geometry of the building and with $flatness = 0$ it depends fully on the geometry of the front objects, if defined.

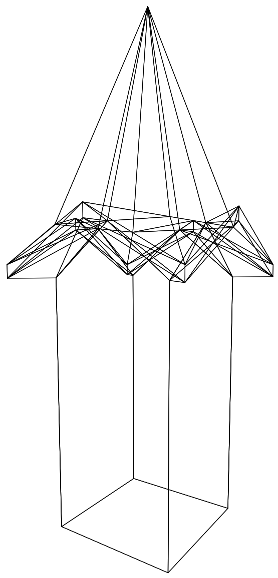

Figure 14. `RoofLayer`**.**

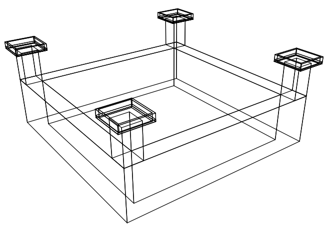

Figure 15. `RoofUser`**.**

`roofpoint` $(ledge, \mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{n-1}})$

Creates a roof depending fully on the geometry of the front objects and the building geometry with a given roof ledge. $\mathbf{p_i} \in \mathbf{R}^3$ defines an additional roof point pulling out the roof from the (top) plane.

`roofuser` $\{$*anObject*; $\}$

Is a simple container for a general object and allows to model a user-defined roof (Fig.15), e.g. a dormer modelled by another building.

4.4. Front

`Front` is the base class and the abstraction of abstract objects which define the geometry of the fronts and replace the default front geometry of the building. `Front` computes the transformation to position the subobjects i.e. the face objects. `Front` introduces a relative coordinate system in which the unit square corresponds to the actual wall defined by `Building`. The three concrete classes `FrontSimple`, `FrontRect` and `FrontTriangle` are subobjects of `building`.

`front` $(frontindex, polygonindex)$ $\{$*faces*;$\}$

Is a simple container for face objects. It positions them only.

`frontrect` $(frontindex, polygonindex, bottom, top)$ $\{$*faces*;$\}$

Creates a rectangular polygon $((0, bottom), (1, top))$, which can overlap the default front at the top and the bottom (Fig.16). For a face object a suitable hole is cut out of its front.

Figure 16. `FrontRect`**.**

`fronttri` $(frontindex, polygonindex, \mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{n-1}})$ $\{$*faces*;$\}$

Creates a user-defined front with the additional vertices $\mathbf{p_i} \in [0,1] \times [-\infty, -1] \cup [1, +\infty]$, which can overlap the default front at the top and the bottom (Fig.17). For each face objects an appropriate hole is cut out of its front.

Figure 17. `FrontTriangle`**.**

4.5. Face

`Face` is the base class and the abstraction of abstract objects which allow to model details on the front without knowledge of absolute position. A face object is defined by a rectangle $(\mathbf{from}, \mathbf{to}) \in [0,1] \times [0,1]$ (if not defined otherwise by the parent front) relative to the parent front object. `FaceArbour`, `FaceDummy`, `FaceItem`, `FaceTunnel` and `FaceWall` are subobjects of a front object. `FaceDummy` is a face object which is passed to other front objects to provide specific information. E.g. a tunnel will influence two or more front objects, but can only be subobject of one front object.

`facearbour` $(\mathbf{from}, \mathbf{to}, n)$
: Creates n arbours inside the rectangle $(\mathbf{from}, \mathbf{to})$. These are modelled using Bézier curves (Fig.18). It is possible to change the default values of the Bézier curve by attributes.

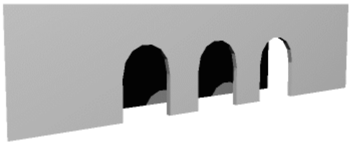

Figure 18. `FaceArbour`.

`faceitem` $(\mathbf{from}, \mathbf{to}, column, row)$ {*anObject*;}
: Alings a general object *anObject* in two dimensions $(column, row)$. It is possible to define a hole in the face object by means of the attribute `hole` and vertices $\mathbf{h_i} \in \mathbf{R}^3$. A hole with no vertices is by default the bounding box of *anObject* (Fig.19).

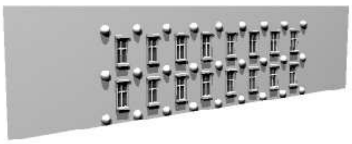

Figure 19. `FaceItem`.

`facetunnel` $(\mathbf{from}, \mathbf{to}, \mathbf{otherfrom}, \mathbf{otherto}, frontindex, polygonindex)$
: Creates a tunnel between the parent front and the front $(frontindex, polygonindex)$, which is modelled using a Bézier curve (Fig.20). It is possible to change the default values of the Bézier curve by attributes.

`facewall` $(\mathbf{from}, \mathbf{to})$
: Creates a rectangular polygon $(from, to)$.

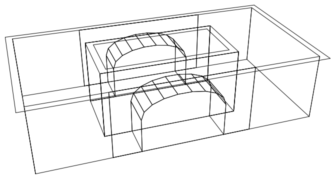

Figure 20. `FaceTunnel`.

4.6. Snatch

`Snatch` is the base class and the abstraction of abstract objects which allow to locate and position a general object with respect to other objects. The only implemented concrete class `SnatchRoof` is a subobject of `building`.

`snatch` $(\mathbf{position})$ {*anObject*;}
: Places a general object *anObject* onto the roof by $\mathbf{position} \in \mathbf{R}^2$ with respect to the given x- and y-coordinates (Fig.21).

Figure 21. `SnatchRoof`.

The presented abstract objects are the kernel for our implementation and modelling system, but do not claim to be complete.

5. Conclusions

We have presented a new approach for modelling and describing buildings. The user constructs the buildings by means of a few highly abstract objects which represent certain parts of real buildings and form together a construction set. The user starts with an initial empty building and makes refinements as long as necessary, i.e. he uses top-down modelling. The construction process is simple and comprehensible. The nearly non-redundant definition avoids the most redundant and meaningless specifications. Fairly realistic looking buildings (Fig.8) and even scenes with many buildings (Fig.22) can be generated interactively and efficiently. Due to the well designed object hierarchy (Fig.9), the construction set can easily be extended with new abstract objects. A natural extension would be to include set operations, i.e. some facilities of CSG. Especially, the cut operation would be useful, as, without it, some logical dependencies between abstract objects lead to rather complicated protocols.

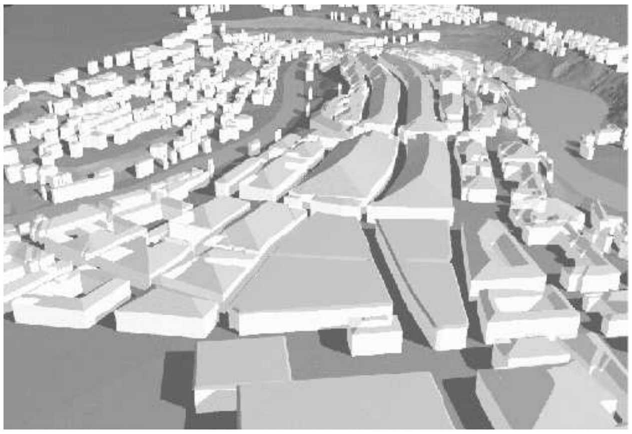

Figure 22. The old part of Berne.

Our construction set has also been developed to model larger scenes, i.e. complexes of buildings, or whole parts of a city (Fig.22). An important application is urban planning, where, as already mentioned, interactivity and efficiency are typically more important than photorealism. To generate a whole scene of buildings, we have developed a simple application. It generates the buildings from a specified set of bases. After a scene has been modelled or generated, it is possible to change the position of the buildings by simple tools or applications. Further variants can be generated easily and quickly. In a future step, animation could be included also. This would allow us e.g. to show the development of a city through time.

To further accelerate the modelling process, we need some additional high-level modelling tools. One such tool would allow to sketch a simple scene in 2D and then automatically generate a corresponding 3D scene.

The time complexity of our implementation depends mostly on the generation of the concrete representation of the abstract objects. Because of this, a cache is used to store the concrete representation to avoid unnecessery computations. The representation will only be generated on demand. This allows to save computing time and memory space. In case of very large scenes, i.e. with more than 250 buildings, the main problem is the time needed to visualize the concrete representation of all the buildings. However, switching off whole branches of the scene graph is not quite satisfactory. A new approach has to be found to simplify geometrically the concrete scene representation.

References

[1] St. Amann, *Komponentenorientierte Entwicklung von Grafikapplikationen mit BOOGA*, PhD thesis, Institute of Informatics, University of Berne, 1997.

[2] St. Amann, Ch. Streit, H. Bieri, *BOOGA: A Component-Oriented Framework for Computer Graphics*, GraphiCon '97 Conference Proceedings, 193-200, Moscow, 1997.

[3] A. Bowyer, J. Woodwark, *Introduction to Computing with Geometry*, Information Geometers Ltd, 1993.

[4] P.E. Debevec, C.J. Taylor, J. Malik, *Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach*, SIGGRAPH '96 Conference Proceedings, 11-20, 1996.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object–Oriented Software*, Addison Wesley, 1995.

[6] A. Hildebrand, *Von der Photographie zum 3D-Modell*, Springer-Verlag, 1997.

[7] T. Matthey, *Objektorientierte Gebäudemodellierung*, Master's thesis, Institute of Informatics, University of Berne, 1997.

[8] L. McMillan, G. Bishop, *Plenoptic modeling: An image-based rendering system*, SIGGRAPH '95, 1995.

[9] `http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html`, *RayShade* Web Site.

[10] J. Rooney, P. Steadman (Eds.), *Principles of Computer-Aided Design*, The Open University, 1987.

[11] A. Streilein, S. A. Mason, *Photogrammetric reconstruction of buildings for 3D city models*, South African Journal of Surveying and Mapping, Vol. 23, Part 5, August 1996.

[12] Ch. Streit, *BOOGA, ein Komponentenframework für Grafikanwendungen*, PhD thesis, Institute of Informatics, University of Berne, 1997.