IDL: A Geometric Interference Detection Language

Pedro Santos
Pedro.Santos@iscte.pt

Manuel Gamito
mag@iscte.pt

José Miguel Salles Dias
Miguel.Dias@iscte.pt

ADETTI/ISCTE, Associação para o Desenvolvimento das Telecomunicações e Técnicas de Informática, Edifício ISCTE, 1600-082 Lisboa, Portugal, www.adetti.iscte.pt

Abstract

This paper describes a novel programming language approach to the problem of automatically verifying the design in an architectural and civil engineering project, fully described in 3D. The language is referred to as IDL, Interference Detection Language, and is used to write geometric-based design verification tests, which are then interpreted and applied to an architectural 3D virtual scene, resulting in geometric interferences if the geometric objects of the scene fail to verify those tests. IDL operators are algorithmically based on simple Geometric Boolean Set and Constructive Solid Geometry operations, which are applied on a Binary Spatial Partitioning organisation of the 3D scene being analysed. This paper also presents Visual IDL, an intuitive graphical editor, which allows a common user to write design verification tests, according to the IDL syntax notation, and still be relatively independent of the language constructs. With Visual IDL, the user is not required to learn the language itself, but is still able to write typical tests quickly and efficiently. Visual IDL can be thought of as a higher-level abstraction of the language, allowing the user to concentrate on exactly what is required, rather than on how to execute it.

Keywords

Computing in AEC-Architecture Engineering and Construction, automatic design verification, geometric interference detection, visual scripting language.

1. INTRODUCTION[1]

Automatic Design Verification (ADV) technology is becoming an important topic in 3D Virtual Prototyping and Digital Mockup (DMU) applications. Some players are operating in this area, with their solutions biased towards the automotive and aerospace markets. This type of technology uses a spatial organisation of the 3D virtual scene based on voxels [Tecoplan]. After a voxel conversion pre-processing stage, end-users can quickly verify any test environments for collision conditions and minimum distance checks between parts. Collisions can also be checked on-line for assembly and disassembly of parts. Static and dynamic swept volumes can be generated for packaging purposes. The technology can be either integrated into existing high-end CAD packages, such as CATIA, or offered as a standalone product, using the VRML data format and a multi-user virtual environment [Blaxxun]. This approach has inspired the authors in developing a new Automatic Design Verification technique for the AEC-Architecture Engineering and Construction sectors, (but that can well be applied in other industrial sectors), with the following characteristics:

- Flexible and upgradeable, through the development of the core of the technology: a simple yet extensible design verification language, referred to as IDL, the Interference Detection Language, the focus of this paper.
- Data format independence by the adoption of the VRML as its native format.
- Support of Object Classification following the recommendations of an ISO standard [ISO 13567].

IDL operators are algorithmically based on basic Geometric Boolean Set and Constructive Solid Geometry operations, which are applied on a Binary Spatial Partitioning (BSP) organisation of the 3D scene being analysed [Requicha80,Foley90]. BSPs provide a much more efficient and general spatial partitioning scheme, for the implementation of Geometric Boolean Set operators, than the voxel organisation scheme [Thibault87] (such as the one used in [Tecoplan]). The IDL scripting language supports different types of geometrical interference tests: minimum

[1] This work is sponsored by the European project IST 26287 M3D "Multi-site cooperative 3D design system for architecture".

IDL layer	Element Class Name	IDL category	R Colour	G Colour	B Colour
AR: Architect	Interior wall	WALL-I	134	134	134
AR: Architect	Exterior wall	WALL-E	187	187	187
SR: Structural Engineer	Concrete Column	C-COL-	0	255	255
SR: Structural Engineer	Concrete Beam	C-BEAM	0	0	255
AC: Air-conditioning Engineer	Air ducts	DUCTS-	255	255	0

Table 1: Example of ISO Codes

distance checks, collisions, intersections between different types of AEC objects and in fact, any user-defined design verification rule that can be expressed by means of a geometric Boolean expression. Our surveys, have shown that architects, designers and engineers require the availability of an Automatic Design Verification tool, such as the one presented in this text, in their regular design process tasks [Dias99]. This ADV tool, in turn, requires programming scripts to be written in a specific technical-oriented language (IDL). This may be a difficult task to the above-mentioned users. To overcome this problem, we have developed Visual IDL, an intuitive graphical editor that allows a common user to write design verification tests, according to the IDL syntax notation, whilst maintaining a relative independency from the language constructs. IDL was conceived, taking in mind its extensibility potential: new objects produced by parts in motion of the assembled design can be used in the future in the context of IDL tests. The IDL language and ADV tool were developed in the framework of a European Union funded project, referred to as M3D (Multi-Site Cooperative 3D Design System for Architecture [M3D]). This project has recently been finalised by all of its developers and partners and approved by a European committee of expert reviewers. The developed M3D system aids architects and engineers, from several specialities, to work concurrently over the Internet, in the design of a full architectural and building construction project, allowing them to be geographically dispersed. M3D provides also the design team, with a multi-user 3D shared virtual environment, especially applicable to the AEC sectors [Luo00]. Each member of such a team can access a joint M3D session from different locations. M3D supports directly the Design Management process, providing a service that enables each specialist to insert 3D design work into a common web database. M3D also allows the aggregation of the several specialised projects, into a final integrated building construction project. In M3D, users are able to navigate in cooperative mode through the 3D virtual scene, pinpointing potential problems, exchanging ideas, comments and suggestions. Users can also communicate by attaching "post-its", linking several documents of different MIME types to 3D scene objects or by using more traditional communication means, such as text chat, audioconference and videoconference. Users are able to activate the ADV tool, whenever needed, that will check for geometric interferences among the different elements of the 3D scene, of different specialities (architectural, structural, water and sewage, etc) that comprise the building construction project. This tool, starts by analysing the whole geometric and topological description of the scene and organises it spatially using a BSP tree scheme. Afterwards, it interprets the IDL script previously selected by the user, which defines the design rules that must be enforced in the integrated project, and executes it over the scene graph, possibly computing the geometric interferences that violate those rules. If geometric interferences are indeed found, ADV adds them to the main 3D scene graph, for viewing purposes.

The paper is structured as follows: the syntax of IDL is synthetically explained in section 2. Visual IDL is then introduced in section 3, as a means to accelerate the development and execution of IDL scripts. This section emphasises in the description of the Visual IDL wizard service and in user interface issues. Section 4 briefly presents some results of interference detection tests, obtained by the execution of the ADV tool over a 3D scene. Section 5, provides a synthesis of the presented architecture and extracts some conclusions.

2. IDL SYNTAX

An IDL test is divided in two parts: the declarative part and the tests specification part.

2.1 The declarative part

In the declarative part, we first define the object we want to test (this is called the target object) and then which objects (the sources) will be tested against it. The objects are all named according to a specific ISO standard [ISO 13567]. The geometry is subdivided into a hierarchy of layers and categories. A layer is a portion of the architectural project that is done independently by one specific specialisation of the design team. As examples, we may have an architectural layer, a structural engineering layer or a water and sewage layer, among others. Each layer is further subdivided into a series of categories, that is, specific specialised project elements (also known as object classes). An architect could define categories like walls, stairs or window openings.

By definition, each object in M3D belongs to a specific layer and, within that layer, to a specific category. This classification of objects is necessary to properly specify the design verification tests. Along with the layer and category definition, an object is also defined by its colour code (in RGB format), which aids in the data exchange process with third party AEC CAD applications.

In Table 1, we show a sample of common ISO 13567 codes (or ISO codes for short) used in M3D. As we can observe in this table, for each layer there are several categories. Taking, as an example, an architectural interior wall: it belongs to the layer of architecture (AR) and category WALL-I, so its full ISO name is ARWALL-I. After declaring the target and the source objects classes in an ordered list, with the target category first, followed by the source categories, all separated by commas and using their corresponding ISO codes, a short string follows it, describing the test itself. This text is merely an aid for the user, since the ADV program will not act on it, as we can see in Table 2.

2.2 The tests specification part

In the tests specification part, it is possible to define rules involving different categories of objects, by means of expressions, written using the IDL syntax. IDL expressions, can be grouped in two types: geometric expressions, which always produce, as a result, a geometric object (that can possibly be a `null` object) and logical expressions, which evaluate to `true` or `false`. Different operators are available in IDL to write expressions. There are Geometric Boolean operators (`union`, `intersection` and `difference`), Unary operators (`scale`, `grow` and `shrink`), Relational operators (“`==`” and “`!=`”) and Compound operators (`subtract` and `compose`). There is also an operator performing attribution. Geometric Boolean algebra is the underlying mathematical theory behind our approach to interference detection. It describes the basic set theory of mathematics but, in our case, applied to three-dimensional geometric objects. According to this algebra, several objects can be combined, with the help of *Geometric Boolean operators*, to produce new objects. With Geometric Boolean algebra it is possible to write an IDL expression like **C** = **A** *op* **B**, where **A** and **B** are two original objects, *op* is a Boolean operator and **C** is the object that results from applying *op* to **A** and **B**. The basic Boolean operators are demonstrated in Figure 1.

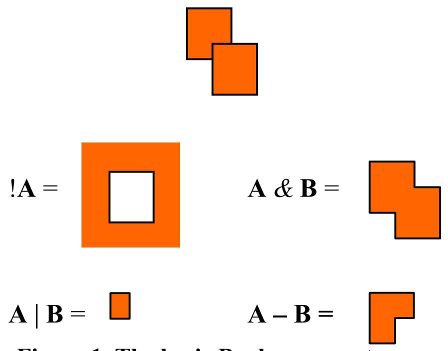


Figure 1. The basic Boolean operators.

The IDL language allows the results of geometric Boolean operations to be stored in new objects and these objects to be used in subsequent computations. In this way, it is equivalent to write:

```
func(object1 & object2), or
new_object = object1 & object2
func(new_object)
```

for some IDL operator `func`. In the remainder of this section, all the arguments to IDL operators that expect geometries use named objects but they can be replaced by any other valid IDL geometric expression, since it always evaluates to an object.

In design verification tasks, it is sometimes necessary to change the volume of an object by enlarging or shrinking it to some amount. This requirement has lead us to the definition of Volume Changing operators, which are unary operators since they take only one argument. As an example, let us suppose that a design rule for a building, expresses the requirement that a person measuring 2.0 meters tall, could climb the stairs without bumping his head on any other AEC element, such as a roof slab. One way to address this requirement is to take the stairs object and enlarge it by 2.0 meters along the vertical direction. This would define a safe volume for the stairs. If we then compute efficiently, a geometric intersection between this safe volume and all other objects in the scene, we will be able to detect if the building obeys this design rule. If an intersection is found, it means that some object of the building is inside the safe volume and the stairs don’t have the necessary space for a 2.0 meters tall person to climb. Volume-changing operators are thus useful to define tolerance volumes around objects and check for minimum distance interferences. There are, at present, three volume-changing operators defined in the IDL language. These are, as mentioned, the `scale`, `grow` and `shrink` operators. They all take one object and produce a new object. The `scale` operator is the simplest one. It takes the form:

scale(object,percent), or:

scale(object,(percent$_x$, percent$_y$, percent$_z$))

The first form scales the object by a global percentage. Scaling by 200% will make the object twice as big, and scaling by 50% will scale it down to half its original size. The second form of the scale operator is similar but applies different scaling factors along the X, Y, and Z directions. Figure 2 gives two examples of the scaling operator.

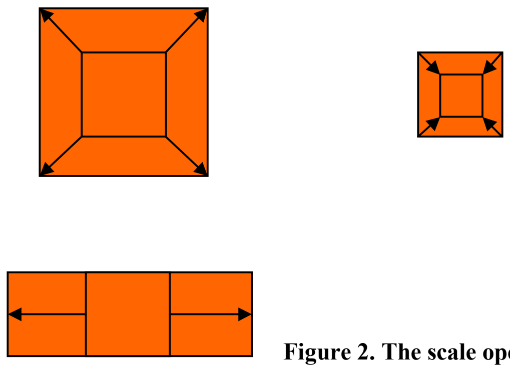


Figure 2. The scale operator.

Next, we have the `grow` and `shrink` operators. They enlarge or shrink, an object by some absolute amount along a chosen direction vector. The object is expanded, or reduced, only along the direction of the vector. They take the form:

```
grow(object,(vx, vy, vz))
shrink(object,(vx, vy, vz))
```

where (v_x, v_y, v_z) is the vector along which the object is enlarged or shrunk. The magnitude of this vector gives the amount of displacement of the object. Figure 3 gives examples of these two operators. In both cases, the object is displaced along the vertical direction by a distance of 1.0 meter. Note that, while the `scale` operator changes the volume of an object by a relative percentage value, the amount of displacement of the `grow` and `shrink` operators is absolute and expressed in meters.

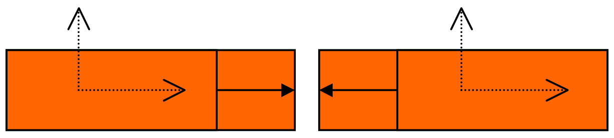


Figure 3. The grow operator for a parallelepiped object. The two dash arrows show two axes of symmetry for the object. The longest arrow is the main axis of symmetry.

These two operators still take another form, which can only be applied to parallelepiped objects. Parallelepipeds are fairly common objects in building projects. Most structural pillars and beams take their shape. Parallelepipeds have a simple geometric structure and it is possible to compute their three axes of symmetry. The `grow` and `shrink` operators, when applied to parallelepiped objects, can take the special form:

```
grow(object, dist)
shrink(object, dist)
```

where it is understood that `object` must return a parallelepiped. The object is enlarged, or shrunk, along its main axis of symmetry by the absolute amount *dist*, in meters. Figure 3 gives an example for the case of the `grow` operator. The `shrink` operator works the same way, with the difference that the volume of the parallelepiped is reduced, rather than expanded. Note that growing or shrinking a parallelepiped by an absolute displacement vector is still a valid operation. So, this special kind of objects can use either type of the volume-changing operators. The displacement amount *dist* can take positive or negative values. If *dist* is positive, the facet of the parallelepiped in the direction of the main axis of symmetry is selected for enlargement or shrinkage. If *dist* is negative, the facet, opposite to the main axis, is selected. Unfortunately, this special form of growing and shrinking operators cannot be used for general objects because these do not always have well defined axes of symmetry.

The Relational operators, used in IDL logical expressions, are the equality and its counterpart, the inequality operator. They are both used to compare two objects. The result of these operators is not another geometric object, but a Boolean value (true or false), depending on the result of the comparison. The equality operator takes the form:

```
object1 == object2
```

and the inequality operator has the form:

```
object1 != object2
```

Often one wishes to compare an object against the empty set. In that case, the special keyword `null` is used to represent the empty set, and the comparison becomes:

```
object == null
```

and equivalently for the inequality operator. These two operators are almost always used in the `return` statement of an IDL test, which will be explained later.

Lastly, the Geometric compound operators operate on whole categories of M3D objects. They are called *compound operators* because they execute, repeatedly, a sequence of basic Boolean operators over all the objects belonging to some given category. At present there are two compound operators defined in the IDL language: the `compose` operator and the `subtract` operator. The compose operator can take two forms:

```
compose(object, category)
```

or:

```
compose(object,(category, category,…,category))
```

This operator computes the intersection of an object of a target category against all the objects belonging to the specified source categories. The first form of the operator above computes the intersection against a single source category, while the second form specifies a list of source categories, enclosed in parenthesis and separated by commas.

To be more specific, if `object` is some geometric object of a target category and `category` is a list of objects $\{v_1, v_2, \ldots, v_n\}$ belonging to some specific source category, the result of the `compose` operator is:

```
compose(object, category)
=(object & v1)|(object & v2)|…|(object & vn)
```

It is the union of all intersections between the first object and all the objects from the source category. This is a useful operator when we wish to check if an object intersects a whole category of objects (interior walls or floor slabs, for instance). It would be tiresome to write explicitly the whole Boolean expression on the right side of the above equality. The `compose` operator conveniently encapsulates such a complicated expression in a simple form.

The `subtract` operator is written is a similar manner:

```
subtract(object, category)
```

or:

```
subtract(object,(category, category,…,categ))
```

This operator takes an object and removes all the portions of its volume that are intersected by objects from the specified categories. Again, this operator can be expanded according to the expression:

```
subtract(object, category) =
  object - v1 - v1 - … - vn
```

where, as before, {v_1, v_2,…, v_n} is the list of objects belonging to a source `category`. The `subtract` operator is useful to remove portions of an object that are common to another category, or categories, of objects.

And finally, at the end of every IDL test there must be a `return` statement. This `return` statement determines if the test failed (a geometric interference was found) or not (the target object does not violate the specified design rule). It contains a Boolean expression featuring one of the two equality operators. The result of this Boolean equality, or inequality, is the final result of the test. The `return` statement takes the form:

```
return error_object if test_object == null;
```

or:

```
return error_object if test_object != null;
```

The `error_object` in the above `return` statements represents the geometric shape of the inconsistency. It is sent to the M3D editor and attached to the scene graph to be visualized, if the Boolean expressions evaluate to `false` for the test object. If we consider again the case of checking if concrete columns are inside of walls, the geometry of the inconsistency will be the portion of a column that is outside of all the walls. This volume of the pillar is flagged in the editor with a blinking colour or a dashed shape for easy visualization. The `test_object` in the `return` statement is what determines the outcome of the test. It should either be equal or different from the empty set, depending on the version of the `return` statement that is used. As an example, in Table 2, we have a typical test performed on the LuisaZ[2] project.

After analysing the IDL script, we can see that it describes a test to verify if air-conditioning ducts (target category) intersect with structural beams (source category). The test simply checks for an interference between these two object categories (using the `compose` operator) and scales the result by a factor of 110%, so that the interference will be enlarged in order to see it more easily in the M3D Editor. Note that the target object category (ACDUCTS-) is referenced in the test specification part by the $$ symbol (which evaluates to all the objects of its category), and all the source objects, of given categories, are identified, respectively, by $1, $2, $3, etc.

[2] Luisa Zambuginho or LuisaZ is a small house project developed by Architect Jorge Silva from Oficina de Arquitectura, Lisbon and used as one of the benchmarks for M3D

```
DEF
v!ACDUCTS-,SRC-BEAM!
Air_ducts_intersects_concrete_beams
Info
{
        string "
        result = compose($$,$1);
        return scale(result, 1.1) if result!=null;
        "
}
```

Table 2: Example test

As we have seen, the IDL language, although powerful and flexible in the description of geometric-based design rules, can be complex for non-technical users. It requires some "a priori" knowledge and understanding of the language constructs, and of geometric Boolean operations, to create the desired design verification tests. We obviously recognized that this was too much of a burden to the users that just wanted to check for interferences between specific objects of their 3D scenes. Users should not be required to learn the language syntax in detail to be able to perform automatic design verification in building construction projects. The Visual IDL tool was created, to support this user requirement.

2.3 Development of the IDL language

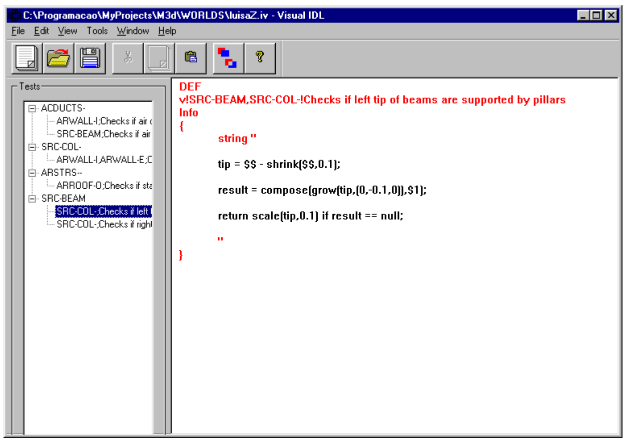
The IDL language was implemented as a Yacc-based grammar file. This file was translated into a code parser by the Yacc tool and this parser was then integrated into the rest of the application.

The IDL scripts are translated internally by the application into a tree of operators. Each IDL operator has one geometric output and one or more inputs. We build the tree by instancing such operators and connecting the outputs of some to the inputs of others. Invoking the operator at the top of the tree will cause the subsequent invocation of all the other operators beneath it. The operators at the bottom of the tree receive the original objects that comprise the geometry of the building construction project. These objects are processed through the chain of operators and the operator at the top will return the final geometry of the IDL test implemented by the tree.

Using this implementation strategy, it is very straightforward to extend the IDL language with new operators. This was a need that had been envisaged from the early stages of design of the language. One could not foresee all the types of geometric tests that architects and engineers would wish to use on their construction projects. Whenever some functionality is desired that is not already supported by the language, one or more appropriate operators can be designed, its syntax coded into the Yacc grammar file and a corresponding instance added to the tree of operators.

3. THE NEED FOR EFFICIENCY IN DESIGN VERIFICATION

Visual IDL was developed to allow the users to create their design verification rules (or tests) rapidly, efficiently and (as this is always desirable) without too much effort. In fact, most of the time a user just needs to make a typical test (like the ones using the `compose` operator) and doesn't require anything too elaborate. So, Visual IDL is a simple and intuitive, yet powerful editor allowing a user to write simple as well as complex design verification rules. With this tool it's possible to create new test files, save them to disk, open existing files, add, modify and remove tests, etc. Its user interface is shown below:


C:\Programacao\MyProjects\M3d\WORLDS\luisaZ.iv - Visual IDL
File Edit View Tools Window Help
Tests

```
ACDUCTS-
    ARWALL-I;Checks if air
    SRC-BEAM;Checks if air
SRC-COL-
    ARWALL-I,ARWALL-E;C
ARSTRS--
    ARROOF-O;Checks if sta
SRC-BEAM
    SRC-COL-;Checks if left
    SRC-COL-;Checks if righ
```

```
DEF
v!SRC-BEAM,SRC-COL-!Checks if left tip of beams are supported by pillars
Info
{
        string "

        tip = $$ - shrink($$,0.1);

        result = compose(grow(tip,(0,-0.1,0)),$1);

        return scale(tip,0.1) if result == null;

        "
}
```

Figure 4: The Visual IDL program

3.1 A Closer Look to Visual IDL

The main Visual IDL window shows the currently active test, which is selected in the tree bar, at the left. Only one test is displayed at a time, as this is easier to understand and to work with. The tree bar displays all the tests and is organized as follows: the main nodes of the tree show the target object category of the test. By opening the node, the source object categories are displayed. By clicking on one of these child nodes, the corresponding test is displayed in the main window, replacing the previous one. Therefore, the tests are organized in the tree bar in a hierarchal mode, giving the user an overall look and an ordered view of the test file. As mentioned, in order to write a test, the user must know the object category to be tested (the target), the objects categories which will be tested against it (the sources) as well as what kind of test he would like. It can be a simple `compose`, a `scale`, `grow` or even `shrink` type of test or a programmed mixture of the above, written in IDL script. To aid this task we have developed a Visual IDL wizard.

3.2 The core service: the Wizard

The Visual IDL wizard directly helps the user to write the test. It will take him in a series of steps, posing several questions about what kind of test the user wants and then creates it, with little effort from the user. To work with the wizard, Visual IDL needs to be launched from the M3D Editor application, which must have a 3D architectural scene already opened. This scene can either be complex or relatively simple like the one of Figure 5.

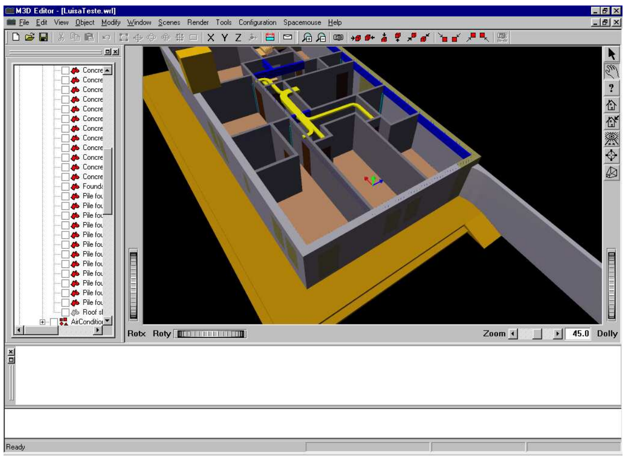
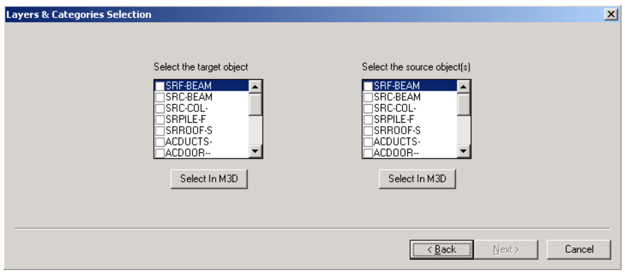

M3D Editor - [LuisaTeste.wrl]
File Edit View Object Modify Window Scenes Render Tools Configuration Spacemouse Help
Rotx Roty
Zoom
45.0
Dolly
Ready

Figure 5: The M3D Editor application with an open scene

This scene is made of several AEC objects, which are all correctly "labelled" with their corresponding ISO codes. For example, the yellow air ducts have an ISO code of ACDUCTS-. A wall has an ARWALL- ISO name attached. Internally, the M3D Editor, the ADV and Visual IDL recognize each object by these eight character names. Since the M3D Editor and Visual IDL are separate applications, M3D must hand-over to Visual IDL all the ISO names that are currently in the scene. Therefore, there's a "two-way communication channel" between the two applications for data transfer and synchronization of tasks.

3.3 A Step By Step Tour

Once the user has the scene loaded, he may want to see, as an example, if the air ducts intersects with any of the interior wall objects. This is an easy and typical task for Visual IDL. After it's launched from M3D, and the wizard is activated, the user will be able to choose which will be the target and the source object categories. These are displayed in a list of checkboxes according to their ISO names (see Figure 6).



Select the target object	Select the source object(s)
☐ SRF-BEAM	☐ SRF-BEAM
☐ SRC-BEAM	☐ SRC-BEAM
☐ SRC-COL-	☐ SRC-COL-
☐ SRPILE-F	☐ SRPILE-F
☐ SRROOF-S	☐ SRROOF-S
☐ ACDUCTS-	☐ ACDUCTS-
☐ ACDOOR--	☐ ACDOOR--

Figure 6: Selecting objects in the Wizard

There are two possible ways of choosing the desired objects. If their exact ISO codes are known, it is possible to choose them directly in the wizard checkboxes. However, users may also select objects by picking operations in the M3D Editor scene. This can be done for the target as well as the source objects. In this case, the Visual IDL window will reshape to a smaller size, put itself to the side of the screen and give the input focus to the M3D Editor (Figure 7).

```
DEF
v!SRC-BEAM,SRC-COL-
!Test1_Tests_if_left_tip_of_beams_are_supported_by_c
olumns
Info
{
        string "
        large = grow($$,(0.0,-0.1,0.0));
        short = shrink(large,0.1);
        tip = large - short;
        error = $$ - shrink($$,0.1);
        return error if compose(tip,$1) == null;
        "
}


DEF
v!SRC-BEAM,SRC-COL-
!Test2_Tests_right_tip_of_beams_are_supported_by_col
umns
Info
{
        string "
        large = grow($$,(0.0,-0.1,0.0));
        short = shrink(large,-0.1);
        tip = large - short;
        error = $$ - shrink($$,-0.1);
        return error if compose(tip,$1) == null;
        "
}


DEF
v!ACDUCTS-,SRC-BEAM
!Test3_Checks_if_air_ducts_intersect_beams
Info
{
        string "
        result = compose($$,$1);
        return scale(result,1.1) if result != null;
        "
}


DEF
v!ARSTRS--,ARROOF-O
!Test4_Tests_stairs_have_enough_space_above_them
Info
{
        string "
        large = grow($$,(0.0,1.0,0.0));
        intersect = compose(large,$1);
        return scale(intersect,1.1)
                if intersect != null;
        "
}
```

Table 3: Test file

In this example, the user has chosen air ducts (the yellow object) and one of the walls surrounding it. The wizard activates the corresponding checkboxes: ACDUCTS- and ARWALL-, which in fact means all objects of those categories. To create more specific tests, that deal only with limited set of objects, and not the entire category, the user may select in the Editor's scene tree, which particular objects he doesn't want to consider for the test. After this stage, the user is prompted for a short text that describes the test.

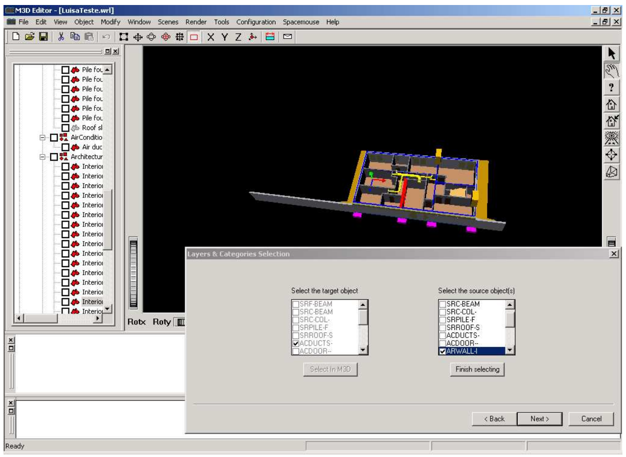


Figure 7: Picking objects interactively in M3D

Finally, the user chooses the type of high-level test to be performed as shown in Figure 8.

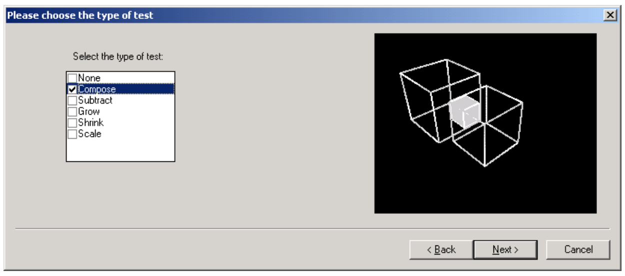
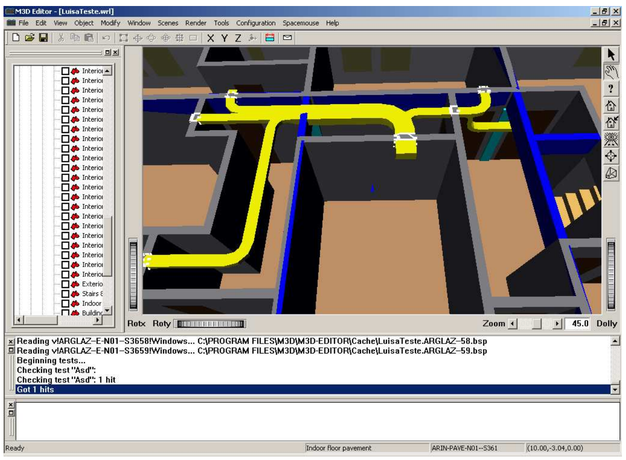


Figure 8: Choosing the type of test (with a graphical rendering aid)

On the left side, the user can choose the desired test. In the middle of the dialog, several optional parameters can be set according to the type of test chosen. And, in the right side, there's an OpenGL frame window graphically displaying in an interactive animation, the type of test and its effect on the geometry. As several values of the parameters are changed, the animation changes accordingly, enabling the user to have full knowledge of what the test will perform, and how. Whenever a simple intersection test is required, the `compose` type of test must be selected, as explained in section 2.2. After this stage, the wizard finishes and writes the IDL code, according to the target and source object categories chosen, the description entered and the type of test specified (and its parameters). After this process, the user can alter what the wizard has produced. The text in black (test specification section) can manually be changed. However, the text in red (declarations section) cannot be changed so easily. This is to prevent inexperienced users to accidentally change the header of the test. In order to change it (the target and source objects as well as the description text) the user can just double-click on the red text and the wizard will pop up. This time, however, it will help him change this information according to his needs. The steps are identical so we will not cover them again. Users with a fairly good knowledge of the IDL language can, consequently, use the wizard to write the basis for a typical design verification test and then, change the body of the test to more complex IDL statements according to their needs. So, while writing typical and more common tests for most of the users, the wizard also provides advanced users with the 'skeleton', or 'building block', of a more powerful and possibly more interesting test.


M3D Editor - [LuisaTeste.wrl]
File Edit View Object Modify Window Scenes Render Tools Configuration Spacemouse Help
X Y Z
Interioi
Exterio
Stairs &
Indoor
Building
Rotx Roty
Zoom 45.0 Dolly

```
Reading v!ARGLAZ–E-N01–S3658!Windows... C:\PROGRAM FILES\M3D\M3D-EDITOR\Cache\LuisaTeste.ARGLAZ–58.bsp
Reading v!ARGLAZ–E-N01–S3659!Windows... C:\PROGRAM FILES\M3D\M3D-EDITOR\Cache\LuisaTeste.ARGLAZ–59.bsp
Beginning tests...
Checking test "Asd":
Checking test "Asd": 1 hit
Got 1 hits
```

```
Ready          Indoor floor pavement          ARIN-PAVE-N01--S361          (10.00,-3.04,0.00)
```

Figure 9: The interferences calculated by ADV

3.4 Performing Automatic Design Verification

After the definition of an IDL script file, the user may invoke the ADV program directly from the M3D editor, as an asynchronous process, which uses, as arguments, this file and a pointer to the 3D scene graph. This way, the editor does´nt block while waiting for the completion of the automatic design verification process. The input data format of the 3D scene is VRML, so ADV requires an initial stage where the proprietary data formats corresponding to the different specialised projects (produced by native CAD applications), must be converted into the VRML format, using the guidelines for layer naming convention described in the ISO 13567 standard. The input shape objects are described according to a BREP format. However, ADV requires a BSP spatial description. Therefore, a filter is executed that converts the VRML format with a BREP organisation, into the same format, but with a BSP tree organisation. Note that, this pre-processing stage is skipped by our algorithm, if the same IDL test is executed for the same 3D scene, a second time, and no shape is altered, since we maintain a cache of the BSP tree of the scene. After this stage, ADV parses the IDL file and executes the geometric-based design verification tests, as stated in the IDL script. It will eventually finish to calculate the geometric interferences, organised as a hierarchy of VRML nodes under a common VRML SoSeparator node. These are the geometries that have originated `false` results in the IDL logical expressions, thus corresponding to the shapes that fail the IDL design verification tests. These interference results will then be added to the M3D Editor's scene graph and visualised, highlighted in white (see Figure 9), under a distributed multi-user collaborative environment [Luo00]. As we can see in this figure, ADV has detected where the air-ducts intersect the walls (in the white areas).

4. RESULTS

Our object classification scheme, the Visual IDL wizard and ADV technique were extensively used in different stages of several building construction projects, between March 1999 and March 2001. Different specialities projects were fully designed in 3D, using proprietary CAD packages and later exported to M3D. An international design team was lead by Architect Jorge Silva from Oficina de Arquitectura, OA in Lisbon. The involved specialities included architecture, water and sewage (OA, Portugal), structural engineering (Betar, Portugal), electricity (IDOM, Spain) and air conditioning (ARQMAQ, Spain). The users, connected by means of a private IP (over ISDN) network, have performed in their offices at their own locations, the design tasks that correspond to different stages of the following projects:

- Luisa Zambujinho/LuisaZ, Almada - March 1999
- Aveiro Pavilion - July 1999
- School Pavilion Tiana, Spain - November 1999
- Mirante Library, Sintra - November 1999 (first phase)
- House in Palma de Mallorca- March 2000
- Industrial Pavilion in Barcelona- September 2000
- Mirante Library, Sintra - November 2000 to March 2001 (second phase)

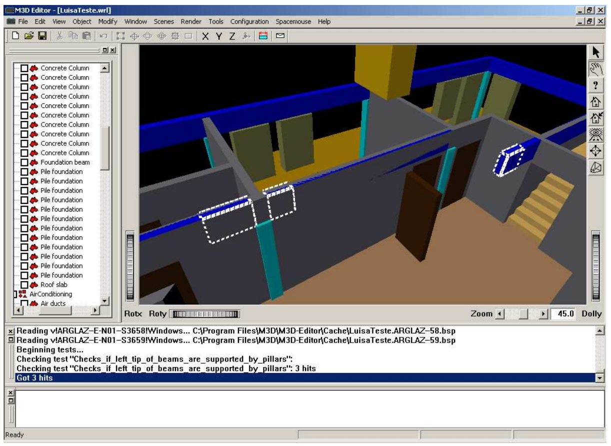
For the specific case of the Mirante Library, starting from November 2000 up to March 2001, the user group made intensive international connectivity tests and on-line synchronous co-operative work sessions. The trials occurred weekly mainly among three locations: Lisbon (OA), Palma (Arqmaq) and Barcelona (Idom). Special user trial scripts where developed to structure the on-line sessions, which were used to test and fine tune the performance of the M3D Editor, ADV algorithm and Visual IDL tool and our AEC project integration technique, within a distributed collaborative work session, served by a central database system. During these real-life user trials, we have performed more than 20 design verification tests with success (that is, we have found a number of design-related issues, that were subsequently solved).

ADV and Visual IDL, were judged by the users has the "most innovative functionality of all the modules" of the M3D system (comprising Multi-user Editor, Database, Conference Management, ADV & Visual IDL) [Fonseca01], enabling operations of intersection, subtraction, reunion, growing, safe volume computation, etc, with AEC shapes. With these tools, users were able to check if the 3D geometries of different specialities fulfilled the rules established by the architects, designers and engineers, or if, within the same speciality, the normative, geometric or common practice rules were verified. The possibility of discussing and checking in group and in real time, the found interference detection results, makes it, according to the users [Fonseca01], "an effective toll for validating the design integration of different specialities within a building construction project".

The following figures (Figures 10 to 13) depict the ADV results, which where embedded in the M3D Editor scene graph, that correspond the IDL test file of table 3 (section 3.4), used for the LuisaZ case. This was a small project with 1760 triangles, including architecture and structural engineering. In table 4 we present the CPU times used by our ADV algorithm, which we have split in the following phases:

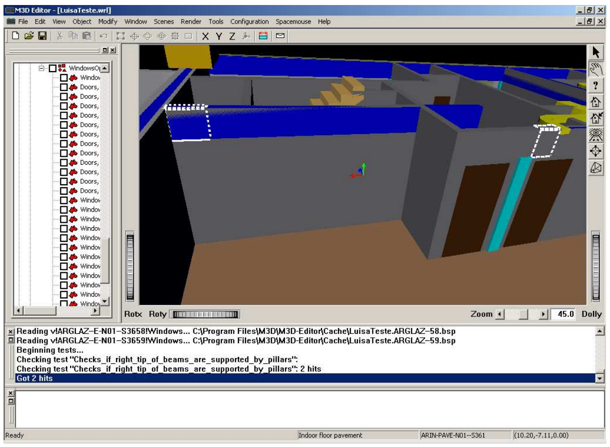
- Phase 1: BREP to BSP conversion and creation of BSP files in the cache.
- Phase 2: Reading the BSPs files from the cache and parsing the IDL script file.
- Phase 3: Performing geometric interference tests.

In table 5, we present the CPU times, of executing the same IDL test in the LuisaZ project, but for the second time this test was executed, that is, when the BSP tree cache was already pre-calculated. The results show an average reduction of 70% in the total CPU time, used by the ADV algorithm, relatively to the cases of Table 4 (with no BSP files yet available in the cache).



```
Reading v!ARGLAZ–E-N01–S3658!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–58.bsp
Reading v!ARGLAZ–E-N01–S3659!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–59.bsp
Beginning tests...
Checking test "Checks_if_left_tip_of_beams_are_supported_by_pillars":
Checking test "Checks_if_left_tip_of_beams_are_supported_by_pillars": 3 hits
Got 3 hits
```

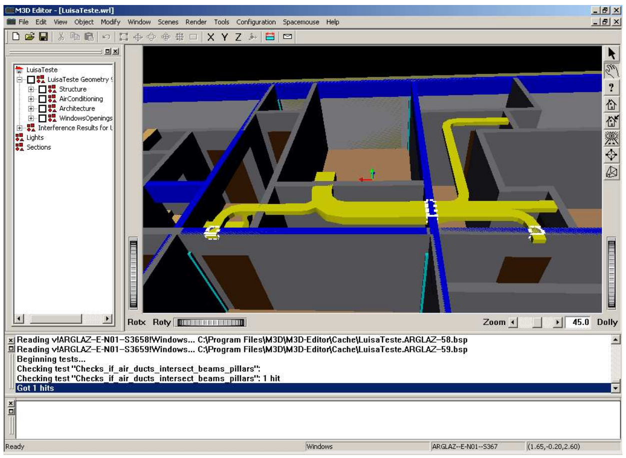
Figure 10: The interference results of test 1. The left tip of these beams are not supported by the columns, but by the walls.



```
Reading v!ARGLAZ–E-N01–S3658!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–58.bsp
Reading v!ARGLAZ–E-N01–S3659!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–59.bsp
Beginning tests...
Checking test "Checks_if_right_tip_of_beams_are_supported_by_pillars":
Checking test "Checks_if_right_tip_of_beams_are_supported_by_pillars": 2 hits
Got 2 hits
```

Ready | Indoor floor pavement | ARIN-PAVE-N01--S361 | (10.20,-7.11,0.00)

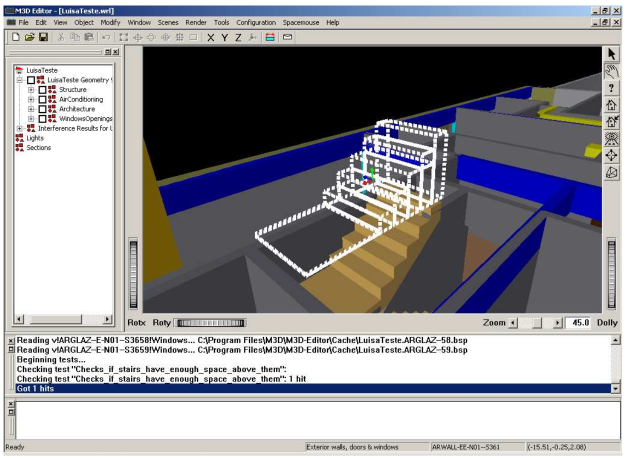
Figure 11: The interference results of test 2. The right tip of these beams are also not properly supported by the columns.



```
Reading v!ARGLAZ–E-N01–S3658!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–58.bsp
Reading v!ARGLAZ–E-N01–S3659!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–59.bsp
Beginning tests...
Checking test "Checks_if_air_ducts_intersect_beams_pillars":
Checking test "Checks_if_air_ducts_intersect_beams_pillars": 1 hit
Got 1 hits
```

Ready | Windows | ARGLAZ--E-N01--S367 | (1.65,-0.20,2.60)

Figure 12: The interference results of test 3. As shown, the air ducts are intersecting the beams.



```
Reading v!ARGLAZ–E-N01–S3658!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–58.bsp
Reading v!ARGLAZ–E-N01–S3659!Windows... C:\Program Files\M3D\M3D-Editor\Cache\LuisaTeste.ARGLAZ–59.bsp
Beginning tests...
Checking test "Checks_if_stairs_have_enough_space_above_them":
Checking test "Checks_if_stairs_have_enough_space_above_them": 1 hit
Got 1 hits
```

Ready | Exterior walls, doors & windows | ARWALL-EE-N01--S361 | (-15.51,-0.25,2.08)

Figure 13: The interference results of test 4. According to the test, the stairs don't have enough space between them and the roof.

CPU time in *ms*	Phase 1	Phase 2	Phase 3	Total
1st Test	1471	171	425	2067
2nd Test	1463	160	418	2041
3rd Test	1451	198	473	2122
4th Test	1493	185	594	2272

Table 4 CPU time used by our ADV algorithm in the LuisaZ project with an initially empty BSP tree cache

CPU time in ms	Phase 1	Phase 2	Phase 3	Total
1st Test	17	154	412	583
2nd Test	17	168	418	603
3rd Test	17	162	431	610
4th Test	18	164	546	728

Table 5 CPU time utilised by our ADV algorithm in the LuisaZ project with an initially complete BSP tree cache

5. CONCLUSIONS

The IDL language has proven to be powerful yet flexible enough to be extensively used by the M3D project architectural and engineering users, such as Oficina de Arquitectura and Betar in Lisbon, or Arqmaq and Idom in Spain. With IDL, users can write tests that perform complex, valuable design verification tasks on geometric data. Great care was taken to ensure that the language remains consistent and yet powerful and efficient. With IDL, the ADV application has a standardized manner of knowing how to act on the geometry. The ADV tool reads an input IDL script file, parses it, analyses the tests according to the IDL syntax and calculates the interferences, attaching these to the M3D Editor's scene graph. Visual IDL was introduced as a bridge between the user and IDL. Geometric interference tests can now be designed, developed and deployed more easily and efficiently. Specifically, the wizard helps common users to write their typical tests with a fairly acceptable degree of complexity and also helps more advanced users with a good knowledge of the language, to write a simpler prototype version of a test first, allowing subsequent developments of the test, perfecting it and improving it until it's finished. Lastly, the wizard itself makes it more easier to choose the target and source objects (by direct picking operations) and the OpenGL frame window provides a clear, clean and simple vision of the types of tests, their operation and their final effects on the objects.

Future activities are planned to address the analysis of temporal-spatial conflicts (and not just spatial, as for the presented results) in design verification and planning tasks [Akinci2000]. These may occur during the time planning of the building construction operations, which involve design elements, construction systems, dedicated machinery, necessary operations, safety regulations, materials and workers. All these entities impose specific and well known time, volumetric and spatial constraints, as well as safety zones, in the construction site time-space continuum. These must be studied concerning its geometric interference, in order to improve safety, for a better humanisation of the construction site workplace, as well as to enhance the quality of the building construction.

ACKNOWLEDGEMENTS

We would like to thank, in no special order, Rui Silvestre, Rafael Bastos and Luís Monteiro for their ideas, comments, suggestions and support, as well as António Lopes and Paulo Lopes for their early work in the project. A special word of gratitude goes to the Oficina de Arquitectura team, namely Arq. Jorge Silva and Zé Manel, as well as for the Eng^{o} Miguel Vilar from Betar, for their ideas and requirements that are in the genesis of the M3D system and of ADV and Visual IDL.

6. REFERENCES

[Akinci2000] Akinci, B., "4D workplanner - A prototype System for Automated Generation of Construction Spaces and Analysis of Time-Space Conflicts", ICCCBE-VIII, Stanford, pp 740-747, August 2000

[Blaxxun] (www.blaxxun.com)

[Dias 99] Dias, J. M. S., Gamito, M., Silva, J., Fonseca, J., Luo, Y., Galli, R., "Interference Detection in Architectural Databases", Short paper Eurographics 99, Milan, September 1999

[Foley90] Foley, van Dam, Feiner, Hughes, "Computer Graphics – Principles and Practice – Second Edition", Addison-Wesley Systems Programming Series, 1990

[Fonseca01] Fonseca, Z, Silva, J., Torres, I., Vilar, M., Castañeda, D., Albiol, M., "Report on the M3D user trial evaluation", ESPRIT 26287, Deliv 3.2, M3D/OA/WP5/T5.3/2001/DL, March 2001 1990

[ISO 13567] ISO standard 13567 "Technical product documentation-Organisation and naming of layers for CAD"

[Luo00] Luo, Y., Galli, R., Sanchez, D., Bennassar, A., Almeida, A. C., Dias, J. M. S., "A Co-operative Architecture Design System via Communication Network", ICCCBE-VIII, Stanford University, USA, August 2000

[M3D] www.m3d.org

[Requicha80] Requicha, A.A.G., "*Representations for Rigid Solids: Theory, Methods and Systems*", Computing Surveys, 12(4), pp. 437-464, December 1980

[Tecoplan] www.tecoplan.com

[Thibault 87] Thibault, W.C., Naylor,B.F., "Set operations on polyhedra using binary space partitioning trees", Computer Graphics (SIGGRAPH '87 Proceedings) (July 1987), M.C. Stone, Ed., vol 21, pp.153-162