Expanding the Functionality of Optical CAD Using the Python Scripting Language

Mikhail Kopylov [1]

[1] *The Keldysh Institute of the Applied Mathematics of RAS, Miusskaya Sq. 4, Moscow, 125047, Russia*

Abstract

Nowadays, scripting is becoming a basic functionality in a very large number of different applications. This paper considers the experience of expanding the program capabilities of the optical modeling system using the Python scripting language. A brief overview of existing solutions is discussed. The approach based on the method of using the unified entity interface is proposed, which makes the process of expansion of the system simple and convenient for both its developers and end users. The new program modules like script interpreter, script editor and built-in parametric object libraries have been designed and integrated into the optical modeling system to work with scenarios are considered in detail. Software extension mechanism by means of adding new script-based object classes is provided. Examples of using Python API for a number of simple operations and examples of work with some simulation and automation modules based on scenarios are considered.

Keywords

Modeling automation, extensibility, scripts, parametric modeling, python, graphical interface

1. Introduction

No matter how advanced an optical modeling system is, sooner or later its capabilities will become insufficient to solve the various problems that arise in practice [1]. Scripting languages may be used to overcome this limitation. Let's note the main benefits that can be obtained using scripting languages:

- Modeling automation. The user can program the necessary sequence of actions that must be performed for a specific modeling task, save it in a script (a file written in a scripting language), and then reuse it.
- Extensibility. Using scripting languages and the corresponding system support, the user can create various types of objects based on scripts, expanding the optical modeling system with new capabilities. Such extensions can be new types and classes, simulation modules, add-ons, graphical interface elements.
- Lowering the entry threshold for ordinary users. To develop extensions, there is no need to thoroughly understand the large and complex internal structure of an optical modeling system, usually written in C++ and requiring advanced qualifications from the user.

Python is often chosen as a scripting language due to a number of its advantages, such as: simplicity, elegant design, disciplining syntax, extensibility, full support for object-oriented programming, cloud and extremely scalable computing [2] availability on various platforms (Windows, Linux, 32/64 bit). It also has over a thousand extension packages for a wide variety of purposes (numpy, scipy, matplotlib, imageio) [3, 4]. It is also important to note that this language is actively supported and continues to evolve.

Into the optical CAD complex, which is being developed by our team, we have integrated support for the Python scripting language. To embed this support, the approach based on the method of using the unified entity interface of complex's objects was applied. Thanks to this approach, it was possible

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia
EMAIL: dvaag@hotmail.com (M. Kopylov)
ORCID: 0000-0002-9526-0766 (M. Kopylov)

to completely solve the problems of automation and subsequent expansion, as well as obtain a number of advantages over alternative solutions. At the same time, special attention was paid to ensuring the friendliness of the application interface (optical CAD's Python API) for the ordinary user and adherence to the principles of object-oriented programming.

2. Existing solutions

Today, there are a big number of CAD systems that include support for various scripting languages. Examples of such systems include Rhino from Rhinoceros, Autodesk Maya, 3DMax, CATIA, and so on.

Upon closer inspection, it becomes clear that all of these systems use very similar approaches to work with scripting languages.

First of all, all these systems assume the presence of a special module called a script interpreter. The main task of the script interpreter is to execute scripts in a supported language with the subsequent translation of constructs dealing with the rest of the application modules.

Another module present in all of the above systems is the script editor. Typically, the script editor is the simplest text editor that allows the user to create and edit scripts, check syntax, and track possible errors in the script code. Some systems have support for integration with external editors that support more functionality, such as debugging scripts at runtime, using breakpoints, etc.

Finally, all the systems provide a dedicated API, which is used for interaction between scripts and system objects. Below, we consider the script support features of some of the above systems in more detail.

The optical modeling system Autodesk Maya has support for several scripting languages at once - MEL (Maya Embedded Language) and Python. Scripts written in the MEL language use the procedural MEL API [5], which has very rich functionality, but is extremely inconvenient for the user due to the lack of object orientation. The Maya Python API [6, 7] is free from this drawback, but this API is very difficult for many tasks due to its low level. It should also be noted that the MEL language uses its own syntax, so certain skills are required from the user to work with it. Scripts can be created and run directly in Autodesk Maya using the built-in script editor. Running scripts in command line mode is also supported. Scripts can be saved as text files in the file system, as well as grouped into special libraries. In addition to all this, there is the possibility of creating plugins based on scripts that extend the capabilities of the system.

The Rhino optical modeling system contains a special Rhinoscript tool used to create scripts based on the Microsoft VBScript language. To work with scripts, COM technology is used, through which access to most of the system objects is provided. Scripts can be run either through the built-in editor or from external files. Rhino has a fairly user-friendly object-oriented Rhinoscript API. This API contains more than half a thousand different methods that allow the user easily automate most of the applied tasks [8].

At the same time, there are some shortcomings in the implementation of scripting support in Rhino. The first is the inability to create new types of scene objects or expand existing scene objects using scripts. The user is limited to only those types and objects that are available through the Rhinoscript API. Thus, adding new functionality to the Rhino system is only possible by developers and not by end users. Second, Rhino lacks support for running scripts from command-line, which can make it difficult to automate some tasks. The last drawback, in our opinion, is that scripts cannot be placed directly into the scene, which means they cannot be saved with it and cannot be automatically executed when the scene is loaded. Therefore, scripting requires explicit user interaction, which can also complicate automation tasks.

Support for scripting languages in CATIA CAD is built in a similar way using COM technology. In CATIA, it is possible to run scripts both from the graphical interface and from the command line interface. One of the interesting features of this CAD is that it can be equipped with an external script interpreter. In this case, users can use any language that supports COM technology, such as Python, Java, or C#, to write scripts. CATIA comes with an advanced script editor, and has a large and well-documented Automation API [9].

Having considered the existing solutions and approaches, we have developed our own approach that eliminates some of the shortcomings that exist in the above systems.

3. Functionality extension method based on unified entity interface

Optical modeling complex we have developed is a modular system built on an object-oriented structure and consisting of various components that can be divided into three main subgroups - kernel modules, graphical interface modules and object libraries [1]. The kernel modules include objects that represent the simulated scene and its components like nodes, light sources, surface attributes, modeling parameters, objects performing modeling, objects-modeling results, etc.

It is important to note that set of kernel components contain all the functionality of the complex, allowing one to perform calculations without using a graphical user interface, for example, by starting calculations from the command line. This is achieved thanks to certain architectural decisions we have use. The same decisions have been used to support the scripting.

3.1. Details of the proposed method

The entity interface is the common interface on which all complex's objects are based. This interface is built from the requirements of the graphical interface - completeness, consistency, sufficiency. The basis for the implementation of the entity interface is a base C++ class called `Entity`, which provides special methods for reading / changing the values of properties of objects, and methods for performing various actions on objects.

The set of objects inherited from the `Entity` class form the C++ API of the optical complex. This interface was naturally transferred to the Python language using the ready-made CPython [10] package via a small number of wrappers over `Entity` and some other kernel's C++ classes. One of the advantages of this method is the ease of subsequent maintenance of the system by developers. So, when adding any new objects or components to the optical modeling complex, the new functionality will be immediately available from scripts. At the same time, there is no need to write any additional code to support new objects in scripts. All that is required is to inherit new objects from the `Entity` class, as well as document the new features in the Python API's User Manual of the optical complex. Another advantage is that the resulting Python API inherits a user-friendly hierarchy of complex's objects, and is also completely object-oriented.

Using the entity interface made it possible to implement a script interpreter that supports the complex's Python API as a separate independent software component, as shown in Figure 1. The script interpreter can be called either from the graphical user interface or from the command line. In the latter case, the script interpreter is explicitly linked with the kernel of the complex.

Since the entity interface of scene's objects is directly translated into Python constructs, the scripts can use the usual dot notation to access sets of properties and methods of various objects as it is shown below:

```
x = object.prop  # Read the scene's object property "prop" to "x" variable
object.prop = x  # Set "x" variable to the scene's object property "prop"
x = object.list[y]  # Read the object's list property (element of array) "list" to "x" variable
…
object.Procedure(param1, param2)  # Invoke method "Procedure" in the scene's object
r = object.Function(param1)  # Invoke method "Function" in the scene's object, and get the result
…
```

This approach allows developing the scripts using the well-established syntax of the Python language. In our case, there is no need to use any additions or separate syntactic constructions to access system's objects from scripts. The end user only needs basic knowledge of the Python language.

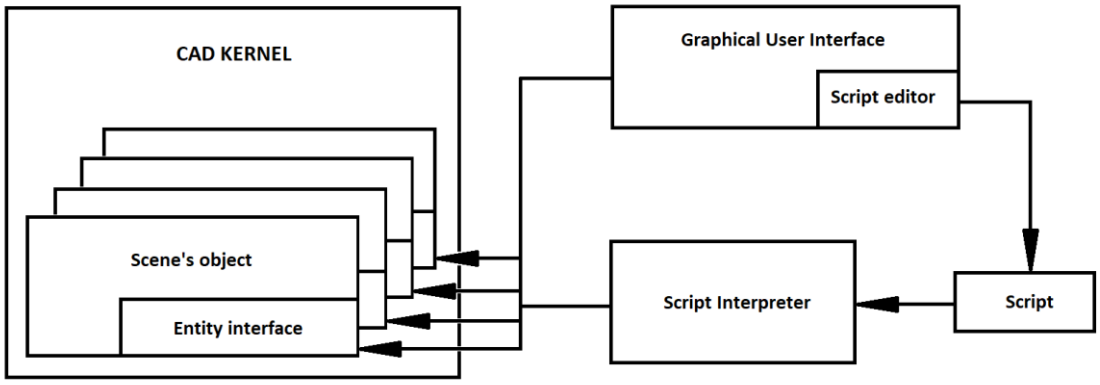

Figure 1: Functional scheme of the approach

For creating or editing scripts, our CAD system has a special module called the script editor. This editor also interacts with scene objects through the `Entity` interface. Due to this, the ability to invoke it is available for any scripted scene object through a special menu item in the graphical user interface. The script editor itself supports some additional features to facilitate script development, such as error checking, loading and saving scripts in the file system. In addition, it supports the simplest templates for the most commonly used extension classes, and provides access to the API documentation. It should also be noted that in our implementation there is no way to auto generate scripts based on user actions, for example, by tracking mouse or keyboard events.

3.2. Extension classes

Modeling automation with scripts is not always convenient. This is because script files containing modeling algorithms are usually stored outside the scene. This makes it difficult to change the modeling parameters, as it requires changing the script itself, which can be large and complex. The same issues are appeared during expanding CAD's functionality via scenarios.

To overcome these problems we use a special approach for expanding the existed functionality, which consists in adding new library objects. The added object can be any scene object. The object placed in the library is saved in the file system, thereby becoming available in subsequent user sessions.

Among other things, objects of new types which are defined by a special class in the Python language can also be added to the library. We call such classes as extension classes. As a rule, such objects are created directly in the scene, then debugged, and finally saved in the library.

Our optical modeling system allows creating of extensions classes for the following categories of objects:

- Scene nodes of arbitrary complexity, including simulators;
- New types of geometric objects;
- Function objects;
- Special types of surface geometry.

All of these classes are parametric. This means that their current state is described by a set of parameter values. From the entity interface point of view, these parameters are properties, thus they can be changed by the user at any time using the graphical interface or using scenarios via Python API.

Figure 2 shows the example of a simple extension class that defines a new type of geometric objects that have the form of a rectangle.

As it is seen from the example, the structure of these classes is quite simple. Any extension class has a header that defines the name of the new type and its category. The category of a class is determined by specifying a base type. Then, the definitions of the class parameters and methods are following. Each extension class has a set of obligatory methods that are used during creation. These methods are:

- Param() This method defines the parameters for the new extension class. Parameter types can be simple, such as scalars and strings, or complex, such as arrays or tables of mixed element types.

In our example, we define three parameters - the size of the rectangle, the coordinates of its center, and the number of line segments used for triangulation.

- Init() This method is a constructor and is executed only once when the instance of object of extension class is being created.
- Eval() This method is automatically invoked by the kernel of optical modeling complex, when one or more parameters is changed from outside via GUI or scenarios. In other words, the purpose of this method is to update the object's representation according to the new parameter values.
- OnNotify() This method allows the object to react on changes emerging in other objects of the optical complex. A special notifications mechanism available via Python API is used for this.

```
class Rectangle(PMesh):
   # Parameters definition
    org      = Param("Origin", VectType(Size))
    size     = Param("Size", VectType(Size(0.001),2), (10, 5))
    step     = Param("Subdivision Step", Int(2, 1000), 10)

    # Object creation
    def Init(self):
        self.AddPart("Rectangle")

    # Calculation of the geometry
    def Eval(self):
        length  = self.size[0], width  = self.size[1]
        step = self.step
        x1, y1, z = self.org
        x2 = x1 + length
        y2 = y1 + width
        xx = []
        yy = []
        for i in range(step):
            xx.append(x1 + i * (x2 - x1) / (step - 1))
            yy.append([y1 + i * (y2 - y1) / (step - 1)])
        xx = [xx]
        self.ClearGeometry() # Resetting previous geometry
        self.AddVerts(0, xx, yy, [[z]])

    # Notification handler
    def OnNotify(self, object, reason):
        pass
```

Figure 2: Extension class example

Newly created parametric objects can be added to the object library for further use. Later, they can be accessed using the graphical user interface. To reuse them, you need to drag and drop the desired object from the library into the scene tree. In this case, a special dialog shown in Figure 3 will be appeared. This dialog will allow you to edit the parameters if necessary. Access to parametric objects from scripts is also possible. To do this, you need to execute certain methods, as shown in the example below:

 # Get the "Rectangle" type from library
Rectangle = GetClass(Shape, "Rectangle")
 # Create new object instance of "Rectangle" type
rect = Rectangle(name = "Rectangle 1", size = (10, 10), origin = (0, 0, 0), step = 10)
 # Add created object to the scene
GetScene().AddNode(MeshNode(rect))
…

In Figure 4 different parametric objects of Rectangle class are shown.

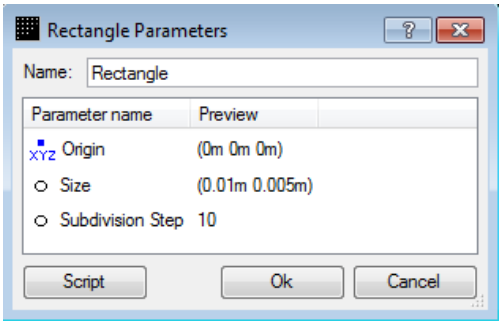

Figure 3: Parameters editor

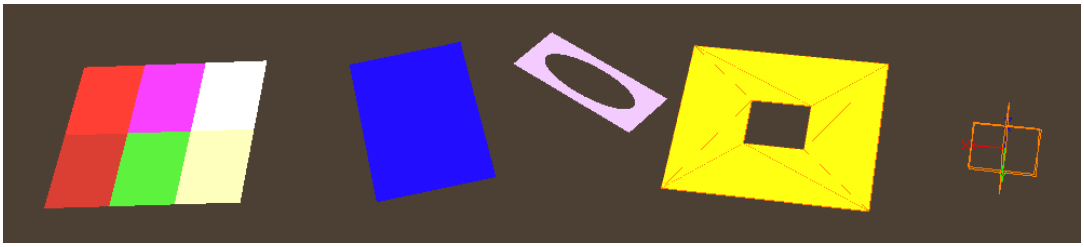

Figure 4: Various objects of Rectangle class

3.3. Advantages and disadvantages of the proposed method

Compared to other implementations of scripting support in other CAD systems, our approach has several advantages, which have already been described above. Despite all this, there are some disadvantages in the proposed method also. First of all, this approach is very difficult to repeat in other optical CAD systems, since this requires at least an adequate internal programming interface using an object-oriented structure. The second disadvantage arises from the first - the strong binding to the optical complex API makes it hardly possible to use the developed script extensions and classes in other optical CADs. At a minimum, this will require a special adaptation of the scripts for the target system's API, which can be very time consuming and therefore meaningless. Another disadvantage is the extremely lacking support for accessing GUI objects from scripts. Limited access to the graphical interface is possible only indirectly through special intermediary classes which complicate the system and do not correspond to the concept of object-oriented programming.

4. Examples of simulators based on the proposed approach

One of the purposes of the extension classes is the creation of various simulators written on Python language that expand the capabilities of the optical modeling complex. The first example shows a simulator that generates a series of images for later analysis, and allows the calculation of the areas with different brightness in the image for various daylight conditions.

Using the parameters, the user can configure the simulator. These parameters include the geographic position of objects in the scene, cloudiness, a list of dates for modeling (year, month, day, time), time step, camera position, and many other parameters that allow you to control various characteristics of image generation from the selection of calculation methods to the format of names generated image files.

As a result of the work of this simulator, two set of image files are generated as it is shown in Figure 5. The first set contains realistic images of the scene, for each time step specified in parameters. The second set contains colored image files presenting the illumination of different areas of the scene. Also these files can be post-processed later via other simulators and tools of our optical complex.

A more complex example of using of the proposed approaches can be a special design optimizer that allows you to optimize any scene parameters to a given target function calculated using FMCRT or Path Tracing simulators. An example of such optimizer is the Plane Light Emitter Device (PLED) optimizer. The specificity of this optimization is a huge number of design parameters. The PLED optimization is

based on modification of two-dimensional microstructure distribution where the each cell in this distribution is a design parameter. Optimization goal is to achieve maximal uniformity of light above output top light group plate.

During the development of this optimizer, a large number of extension classes in Python were written, each of which is aimed for specification of all required input for optimization. These classes are designed to:

- Provide access to optimization parameters, their links to a scene objects and constraints.
- Declare of objective functions.
- Declare of optimization parameters.
- Implement methods of optimizations.
- Implement simulators used in optimization.

In the end, all these classes were aggregated into a special scripted object placed in the library of the optical modeling complex to provide the user a convenient mechanism for working with the optimizer. To start working with it, you just need to place the optimizer object into the scene tree, configure the necessary parameters and start optimization. The results of the optimization can be monitored through the graphical user interface, and also saved to a file. An example of this calculation and its results are shown in Figure 6.

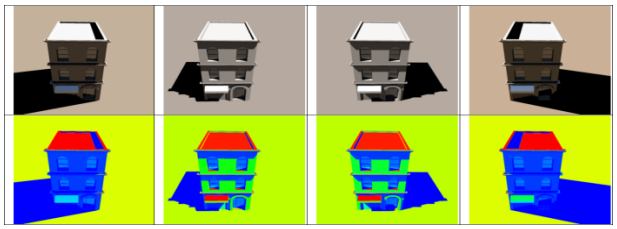

Figure 5: Daylight analysis simulator

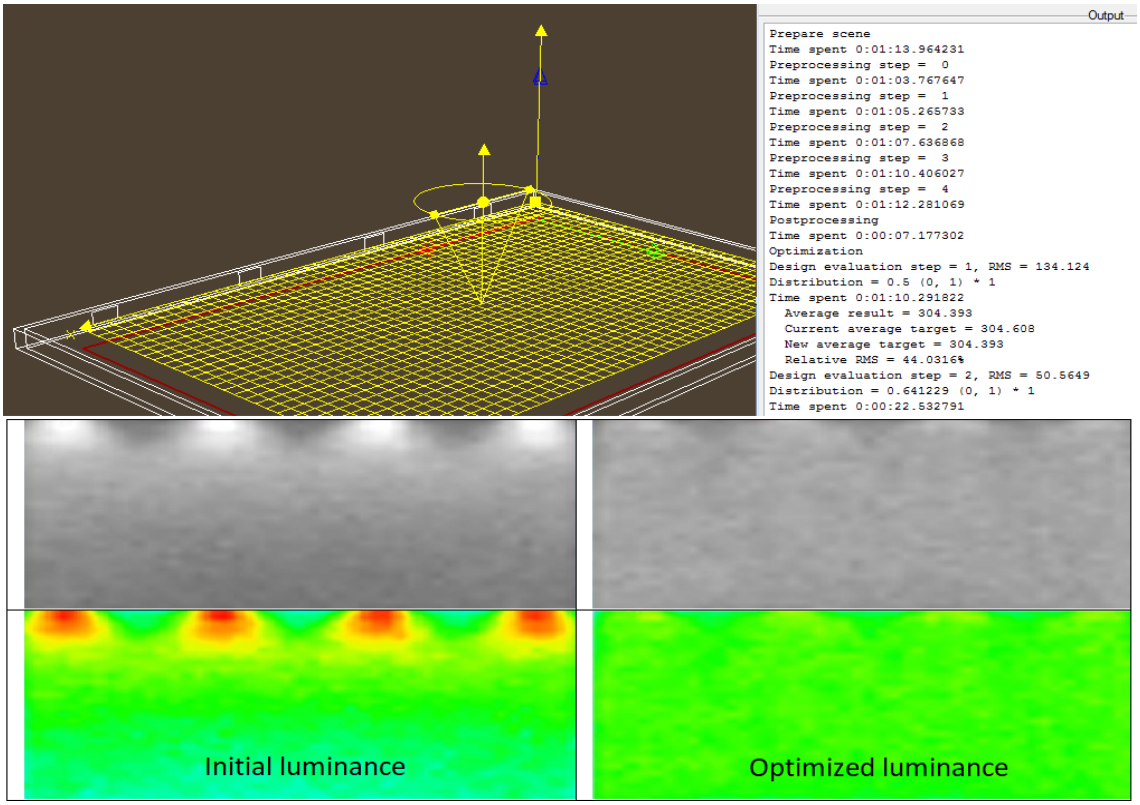

Figure 6: The optimization process and its results obtained via a script-based design optimization tool

5. Conclusion

The presented approaches made it possible to easily expand the capabilities of the optical modeling system. At the moment, within the framework of working with the system, a fairly large number of different scenarios have been written. At the same time, scenarios and extension classes are widely used not only by the end users, but also by the developers themselves. Some of the scenarios are used directly for modeling tasks; the other part is used for service tasks such as testing and diagnosing of the components of the system.

6. References

[1] B. K. Barladian, A. G. Voloboy, V. A. Galaktionov, L. Z. Shapiro, Integration of realistic computer graphics into computer-aided design and product lifecycle management systems, Programming and Computer Software 44(4) (2018) 225-232. doi:10.1134/s0361768818040047.

[2] Ami Marowka, Python accelerators for high-performance computing, The Journal of Supercomputing 74(4) (2018) 1449-1460. doi:10.1007/s11227-017-2213-5.

[3] NumPy The fundamental package for scientific computing with Python, 2020. URL: https://numpy.org.

[4] Stefan Van der Walt, et al, scikit-image: image processing in Python, PeerJ 2 e453 (2014), doi: 10.7717/peerj.453.

[5] PyMEL for Maya, 2018. URL: https://help.autodesk.com/cloudhelp/2018/JPN/Maya-Tech-Docs/PyMel/index.html.

[6] Gilenn Emille G. Collado, Modeling and Python Scripting in Maya for the Animation Short Style, Department of Computer Engineering California Polytechnic State University, San Luis Obispo, CA, 2016.

[7] A. Mechtley, R. Trowbridge, Maya Python for Games and Film: A Complete Reference for the Maya Python API, CRC Press, 2011. doi:10.1201/9780123785794.

[8] L. Kera, J. Niemasz, C.F. Reinhart, Animated building performance simulation (ABPS)–linking Rhinoceros/Grasshopper with Radiance/Daysim, Conference proceedings of 4-th National Conference of IBPSA-USA, New York City, New York, 2010, pp. 321-327.

[9] Yihui Li. Research of Integration Technology between CATIA and TOOLMANAGER Based on CAA, International Journal of Advanced Network, Monitoring and Controls 1(1) (2018). doi:10.21307/ijanmc-2016-010.

[10] Extending and Embedding the Python Interpreter, 2021. URL: http://docs.python.org/3/extending/index.html.