Lumière: a Novel Framework for Rendering 3D graphics in Smalltalk

Fernando Olivero, Michele Lanza, Romain Robbes
REVEAL@ Faculty of Informatics - University of Lugano, Switzerland
{fernando.olivero, michele.lanza, romain.robbes}@usi.ch

ABSTRACT

To render 3D graphics there is a number of different frameworks written in Smalltalk. While most of them provide powerful facilities, many of them are outdated, abandoned, undocumented or heavyweight.

In this paper we present Lumière, a novel lightweight framework for rendering 3D graphics using OpenGL based on a stage metaphor. Lumière is implemented using the Pharo IDE. In its current state it supports basic and composite shapes to populate 3D scenes, features a camera, and a lighting model.

We illustrate the usage of Lumière with Gaucho, an environment for visual programming we are currently building.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Graphical environments

Keywords

3D, Smalltalk, OpenGL

1. INTRODUCTION

Over the years, the Smalltalk language and its many dialects have featured many frameworks for rendering graphics in 3D. Well-known examples include Alice[6] and Croquet[8] for Squeak, ST3D for Dolphin Smalltalk, and the Jun[1] framework for VisualWorks.

However many of the existing frameworks are affected by problems such obselence, complexity, and licensing. Squeak Alice is no longer maintained and therefore lacks support for features present in most modern renderers (for example multi-texturing, Vertex buffer objects, vertex shaders, etc.). Croquet, while it is still maintained, has become a complex full-fledged collaborative environment that runs on top of Squeak, and has thus long left the status of being a framework that is easy to get into. Jun is less complex than Croquet, but is affected by the licensing policy of VisualWorks, despite the fact that Jun itself is free and open.

Indeed, the Smalltalk community is missing a lightweight and open source 3D framework. To fulfill this need, using Pharo[2], we are developing a novel framework for rendering 3D graphics in Smalltalk called **Lumière** .

One of the cornerstones of Lumière is to hinge on the metaphor of a stage. We believe this helps to make the framework and its usage more intuitive, as metaphors are powerful tools to assist the usage of abstract concepts: If a framework is built around an intuitive metaphor which maps abstracts concepts to real-world objects, programmers can obtain an immediate mental model of the framework's domain model, thus easing the understanding and usage of the framework.

To apply this concept to 3D graphics, Lumière's stage metaphor implies that all graphics are produced by cameras taking pictures of 3D shapes lit by the lights of the stage. A stage provides the setting for taking pictures of a composition of visual objects we call micro-worlds. With Lumière a Pharo programmer can produce 3D graphics using high level abstractions (cameras, lights, stages and shapes) instead of low-level graphic instructions.

We want Lumière to be fully integrated in the Pharo environment to provide a seamless user experience. Thus Lumière's rendered stages are integrated with the windows and browsers of the Pharo environment, and makes interactions with a stage possible through the use of the mouse and the keyboard.

In this paper we describe Lumière, the reasons that led to its implementation, the metaphor behind it, its key characteristics and capabilities. To illustrate its usage and potential we describe *Gaucho*, a 3D environment for visual programming we are currently building.

Structure of the paper.

In Section 2 we describe existing 3D frameworks developed in Smalltalk. In Section 3 we explain the motivation for developing Lumière instead of using an existing framework. In Section 4 we detail the design and the implementation of this novel framework, while in Section 5 we demonstrate the extensibility of Lumière and in Section 6 we briefly discuss the performance of Lumière. Section 7 describes our 3D visual programming environment named Gaucho, built on top of Lumière. Finally, in Section 8 we conclude this paper.

[1]`http://www.cc.kyoto-su.ac.jp/~atsushi/Jun/`

[2]`http://www.pharo-project.org/home`

2. RELATED WORK

Smalltalk has several frameworks and tools for producing 3D graphics; these have distinct approaches to the problem of rendering graphics. In this section we categorize the existing frameworks and compare them according to relevant properties.

The frameworks or tools for producing 3D graphics in Smalltalk can be categorized into two groups, grouped by the level of abstraction they provide over graphics instructions. The first group contains low level library interfaces solely providing communication with a 3D library written in another language, while the higher level frameworks abstract over the graphic libraries in order to ease graphic programming.

2.1 Low level library interfaces

Several Smalltalk dialects have interfaces to hardware accelerated libraries such as OpenGL and DirectX.

Examples are the OpenGL interfaces provided by Smalltalk X[3] and Squeak[4] (see Figure 1, B), and Smalltalk MT[5] (which also features a DirectX interface).

While these interfaces allow the programmers to produce 3D graphics in Smalltalk, they only provide glue code for calling graphics primitives. Programmers are forced to think in terms of primitive elements such as vertices, normals and polygons. Worse, data structures are underused. For instance, the primitives to use a vertex has three floating point parameters instead of a dedicated vertex data structure. The data structure used for more complex shapes is a simple array of floating-point values. This violates the basic principles of object-oriented programming and forces developers to deal with painstaking low-level details.

The decision to have a low level interface is a sound decision for efficiency purposes, in order to render large numbers of polygons. However, abstractions are needed for modeling purposes, which these libraries do not provide. It is up to the programmer to implement them.

2.2 High level frameworks

These frameworks, built on top of graphic libraries, provide high level abstractions for modeling 3D objects and producing graphics. They provide facilities for describing a 3D scene in terms of objects, promoting separation of concerns (and reuse) for the lighting model, the transformations that define spatial relationships between shapes, the camera, and the viewing volume of the scene. Examples are Balloon3D, Squeak Alice[6], ST3D, Croquet[8] and Jun.

We compare the frameworks using several properties listed in Table 1. The relevant properties we distinguish are *abstractions*, *size*, *loadable*, *free commercial usage*, and *maintained.*

- Abstractions refers to the amount of high level concepts that the framework provides, promoting reusability and separation of concerns.

- The size is considered because smaller frameworks are easier to understand and maintain.

- Loadable refers to the modularity of the framework, i.e.., whether it is easily loadable (in the best case with just one click) into a Smalltalk image.

- Free commercial usage: this property describes if the framework can be used in a commercial context without any license problems. In Smalltalk this is far from being an irrelevant issue, since deployed applications are inseparable from the virtual machine and the image containing the language library.

- Finally, the framework status is considered, whether it is still being maintained or has become obsolete.

[3]`http://live.exept.de/doc/`
[4]`http://www.cosmocows.com/OpenGL`
[5]`http://www.huv.com/smalltalk/opengl.html`

Balloon3D.

This is a renderer for Squeak written entirely in Smalltalk. It provides lighting, shading, texturing, meshes and matrix transformations in an object oriented framework. Balloon3D does not provide animations, only rendering a single frame at a time. The framework is not maintained anymore and has become outdated, thus it lacks several modern features (such as pixel and vertex shaders; multi texturing, etc.).

We considered maintaining and enhancing Balloon3D instead of building Lumière, but found several problems with this approach. For example Balloon3D was designed as renderer written entirely in Smalltalk, and the hardware acceleration using OpenGL was added later, so we preferred to build a renderer designed from scratch to take full advantage of the performance and scalability of OpenGL. Moreover the lack of documentation, usage and maintenance of Balloon3D made us discard the idea of maintaing and enhancing it.

Squeak Alice.

It is an implementation of Alice[2] for Squeak. It is basically a scene graph based renderer. A scene graph is a hierarchical structure for modeling the relationships between objects in the scene. It also features a scripting environment for 3D objects that allows for the creation of animated worlds through time (see Figure 1, C). Squeak Alice uses Balloon3D for rendering the scenes[3]. This framework is also not maintained, and because it uses Ballon3D it also lacks modern features.

ST3D.

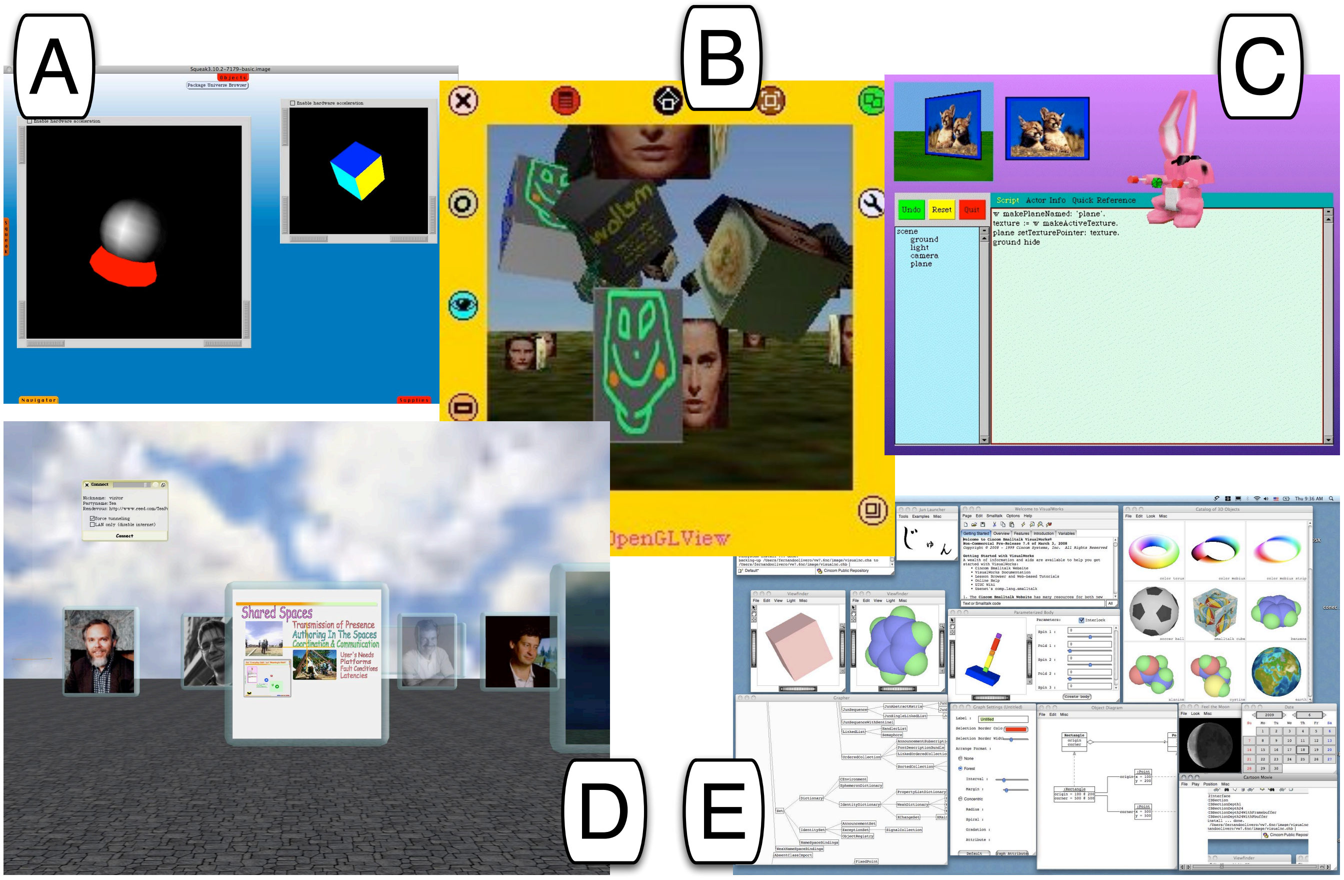
This is a framework for creating 3D applications such as games, simulations and modelers, with an OpenGL interface. It runs on the Dolphin Smalltalk dialect. This commercial framework was abandoned by the company that created it and it cannot be loaded into the latest Dolphin version.

Jun.

This is a framework for handling multimedia, 2D and 3D graphics (see Figure 1, E). Jun is object-oriented and provides good abstractions for modeling 2D and 3D graphics. For example the CodeCity[**?**, 9] tool has been built using Jun. It runs on the VisualWorks Smalltalk dialect and uses OpenGL as a base renderer. Using it in a commercial context for free is not possible because Jun is affected by the licensing policy of Visual Works.

Croquet.

This is a full-fledged open source Smalltalk environment that allows for the creation of collaborative 3D applications



```
w makePlaneNamed: 'plane'.
texture := w makeActiveTexture.
plane setTexturePointer: texture.
ground hide
```

Figure 1: Screen captures of existing 3D frameworks. A: Balloon 3D; B: Squeak OpenGL wrapper; C: Squeak Alice; D: Croquet; E: Jun

(see Figure 1, D). For the actual rendering it uses the TeaPot framework, which is based on OpenGL. Croquet provides a rich set of abstractions for modeling hierarchical scenes, and for defining the material, texture and geometry of its elements. Croquet is a large and complex system with good abstractions, but its usage is far from being immediate, and due to its complexity it fails to load into a standard Squeak image. To make up for this, the Croquet developers offer ready-made images.

3. MOTIVATION

As we see from the brief, and certainly not exhaustive, survey we conducted in the previous section there seems to be no ideal solution for a Smalltalker who wants to to delve into the domain of 3D graphics.

Our goal is to develop a modern, efficient and lightweight open source 3D framework for Smalltalk, that provides an interface to OpenGL, high level abstractions for modeling the scenes, features detailed documentation and supports the latest features of modern renderers. The framework should also be one-click loadable into a standard Pharo image.

For rendering the graphics we could either build a new renderer using Balloon3D, DirectX or OpenGL. Building a new engine implies providing maintenance and keeping up with all the new features and extensions of the others. For example since 2004 Balloon3D has not been maintained and presently lacks several modern features. For convenience, scalability and efficiency we chose to use an existing, modern, powerful and hardware accelerated renderer. Because we require the framework to be open source and cross platform we settled on OpenGL (instead of Microsoft DirectX, which is commercial and only available on Windows).

None of the existing frameworks satisfy our needs. Squeak Alice and ST3D are high level 3D frameworks that are no longer maintained and have become obsolete. Croquet is overly complex and does not even cleanly load into a standard Squeak or Pharo image. Jun is affected by the licensing policy of VisualWorks, despite the fact that Jun itself is free and open, it is also poorly documented and does not support many of the latest OpenGL features.

In short, it is time to reinvent the wheel with Pharo and Lumière.

4. Lumière

Lumière within Pharo is depicted in Figure 2. In this section we describe Lumière's characteristics, features and implementation.

4.1 Characteristics

Lumière is an object oriented framework that provides

Name	Dialect	Abstractions	Size	Loadable	Free Commercial Usage	Maintained
Balloon3D	Squeak	Low	Small	Yes	Yes	No
OpenGLMorph	Squeak	None	Small	Yes	Yes	No
SqueakAlice	Squeak	Medium	Small	Yes	Yes	No
ST3D	Dolphin	N/A	N/A	Yes	No	No
Jun	VisualWorks	High	Medium	Yes	No	Yes
Croquet	Squeak	High	Large	No	Yes	Yes

Table 1: Features and characteristics of existing frameworks and tools

programmers with facilities to model micro-worlds and produce graphics using high level abstractions such as cameras, lights, stages and shapes. It provides a layer of abstraction over graphical primitives and low level rendering. Lumière hinges on a stage metaphor, while rendering is accomplished by a camera taking pictures of the micro-worlds, illuminated by the stage lights. For populating the worlds programmers deal with shapes and their visual properties (like scale, materials and textures), instead of low-level constructs such as vertices and normals that specify polygons and lighting models.

Lumière is currently composed of 71 classes, including tests and examples. Its reduced size makes it easier to understand than more complex frameworks such as Croquet.

The framework is one-click loadable into the latest Pharo image, i.e., we developed a loader that downloads the packages and dependencies from an http repository and installs them into the image. To load Lumière one can evaluate the following script:

```
ScriptLoader
loadLatestPackage: 'LumiereLoader'
from: 'http://www.squeaksource.com/Lumiere'.
LumiereLoader  load.
```

The framework uses OpenGL as the base renderer, the industry standard system for doing high performance graphics. Scalability is achieved because the objects in the scenes and their relationships are modeled with a scene graph, an approach that is based on one of the most renowned frameworks for producing 3D graphics in the industry called Open Scene Graph[6].

Lumière has support for modern OpenGL features, such as Vertex Arrays and Vertex Buffer Objects[7] (efficient techniques to specify polygons and their visual properties, allowing for improved performance when rendering a large number of polygons). Many modern features are still missing (vertex and fragment shaders, procedural texture mapping), but the framework has the required infrastructure for easily adding them, as we detail in Section 5.

4.2 Features

Lumière currently features a stage metaphor, the concept of a micro-world, a set of basic shapes, and provides the means to view and manipulate the stages. In Figure 3 we display an UML diagram with the core classes of the stage metaphor. In Figure 4 we present the model entity diagram of a stage and the collaborators.

[6]`http://www.openscenegraph.org/projects/osg`
[7]`http://www.opengl.org/wiki/GL_ARB_vertex_buffer_object`

The stage metaphor.

Lumière produces 3D graphics by using cameras taking pictures of micro-worlds illuminated by the lights of the stage. With the stage metaphor Lumière attempts to simulate the real world, providing a layer of abstraction over OpenGL.

The facade to the rendering framework is the stage, an instance of `LStage`. This object contains a micro-world, cameras and lights. Besides serving as an interface to the rendering system, the stage also models environmental properties such as ambient lights and fog. A stage is not a visible object, it is an object that provides the setting for taking pictures of a micro-world that can be visualized as a regular morph in the environment.

The lights are objects with the responsibility of illuminating the scenes, there are several types of lights in Lumière, based on OpenGL lighting model (see Section 4.3). These objects are located anywhere in the stage, and directly influence the final appearance of the shapes in the scenes.

The cameras are objects with the responsibility of taking pictures of micro-worlds on a canvas. They dictate the distance, orientation and angle of sight from which the picture is taken, and the visible portion of the 3D graphic that will appear on the rendered image (see Section 4.3).

Micro-worlds and shapes.

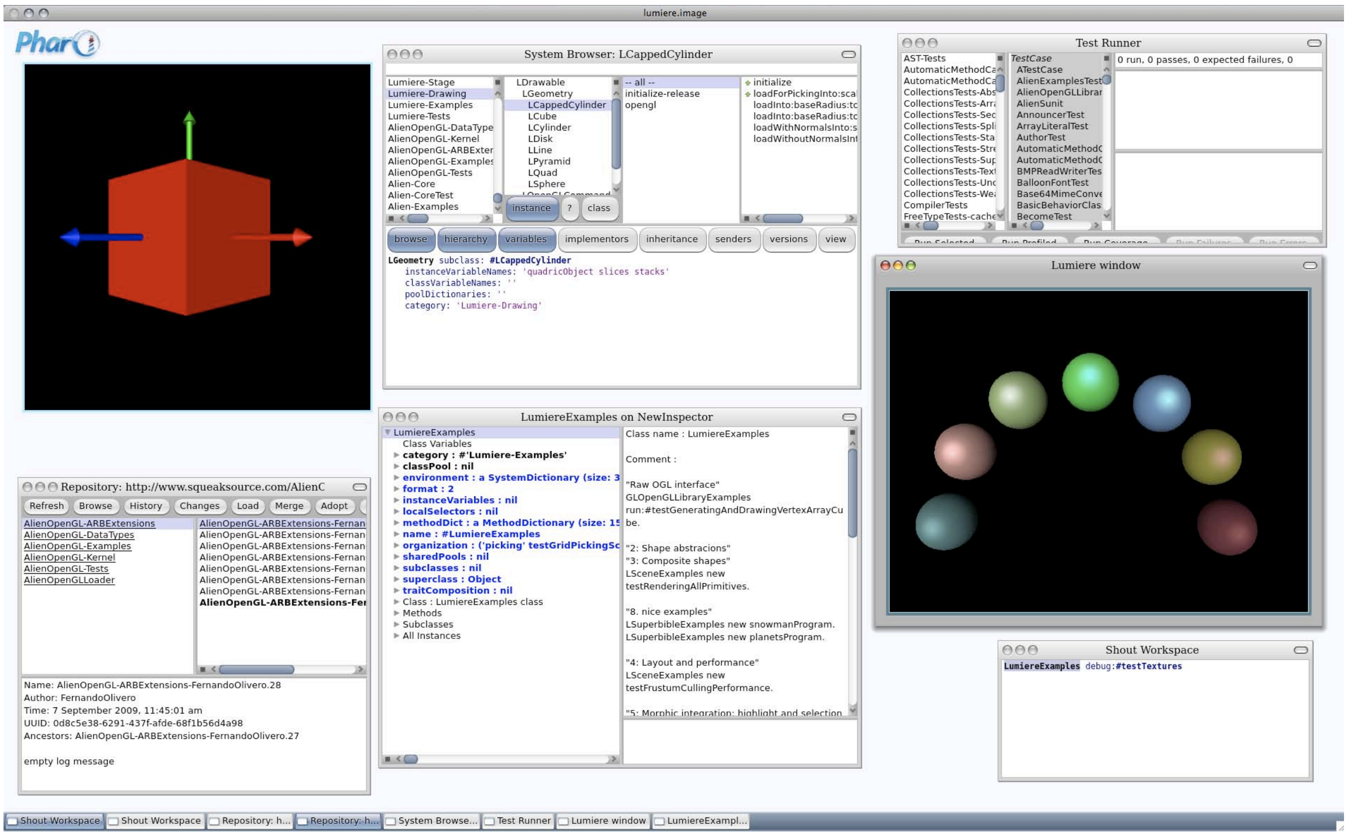
Shapes are the visual entities of the framework, and they populate Lumière micro-worlds. In Figure 5 we present a Lumière stage displaying some shapes. Lumière currently provides primitive shapes, such as spheres, cubes, cylinders and disks, that are instances of `LShape`. Shapes have geometric and other visual properties, like color, scale, material and textures.

Shapes can also be composed (see Figure 6), created programatically from other shapes by specifying the shapes that conform the composition and the spatial relationships between them.

Micro-worlds are modeled internally as a scene graph (see Section 4.3) enabling the construction of worlds using high level language. The user can describe the spatial relationships between shapes in a world (and in composite shapes) with a high level protocol instead of issuing commands defining translation, rotation or scaling nodes of the scene graph to achieve the same goal. In Appendix A we show the code to generate Figure 6.

Viewing and manipulating the stages.

Our goal is to provide a seamless integration between Lumière and the underlying environment, unifying Lumière and Pharo GUI applications. A morph is the graphical element in a Pharo GUI, part of the Morphic framework (we



```
LGeometry subclass: #LCappedCylinder
    instanceVariableNames: 'quadricObject slices stacks'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Lumiere-Drawing'
```

```
0 run, 0 passes, 0 expected failures, 0
```

```
Name: AlienOpenGL-ARBExtensions-FernandoOlivero.28
Author: FernandoOlivero
Time: 7 September 2009, 11:45:01 am
UUID: 0d8c5e38-6291-437f-afde-68f1b56d4a98
Ancestors: AlienOpenGL-ARBExtensions-FernandoOlivero.27

empty log message
```

```
Class name : LumiereExamples

Comment :

"Raw OGL interface"
GLOpenGLLibraryExamples
run:#testGeneratingAndDrawingVertexArrayCube.

"2: Shape abstracions"
"3: Composite shapes"
LSceneExamples new
testRenderingAllPrimitives.

"8. nice examples"
LSuperbibleExamples new snowmanProgram.
LSuperbibleExamples new planetsProgram.

"4: Layout and performance"
LSceneExamples new
testFrustumCullingPerformance.
```

```
LumiereExamples debug:#testTextures
```

Figure 2: Pharo and Lumière integration

provide more details in Section 4.3).

For visualizing the stages and their micro-worlds we created a new morph called `LStageMorph`. A stage morph can be integrated into a Pharo system window, supporting common operations such as minimizing and moving the window containing the stage morph (see Figure 2).

Stage interaction is provided through the keyboard and mouse. Clicking on the mouse anywhere on the stage selects the shape underneath the cursor. Hovering around the scene with the cursor updates a floating view, that displays information about the shape beneath it. Different event handler policies can be applied to a stage, customizing, adding or removing interactions, e.g., the camera position and orientation can be modified using the keyboard.

4.3 Implementation

In this section we explain how communicating from Smalltalk to OpenGL is achieved and explain how Lumière renders scenes.

OpenGL interface.

OpenGL is defined as a graphics system that is a software interface to graphics hardware. For calling this foreign library from Pharo we use the ALIEN FFI[8] framework, developed by Cadence systems for their Newspeak enviroment, also available in Pharo. We extended the framework with a complete OpenGL interface, composed of a singleton instance of the `LOpenGlLibrary` and related classes like `GLEnum` for reifying OpenGL data types.

Drawing primitive figures.

In Lumière the primitive figures are rendered to an OpenGl canvas, that contains an OpenGL context which does the actual rendering. A canvas, instance of `LOpenGLCanvas`, is an abstraction of a 3D surface where primitive figures can be rendered. For example it knows how to answer the messages `#drawSphereScaled:`, `#drawCubeScaled:` and `#loadColor:`. The canvas forwards the messages requesting the rendering of figures to the proper instance of `LGeometry`, and the messages that modify OpenGL state to its opengl context. The `LGeometry` subclasses model the Lumière primitive shapes, which understand the message `#loadInto:`, thus knowing how to render a figure in the OpenGL context passed as argument.

Lighting model.

The scene of the stage be can lit by different kinds of lights. The stage has an ambient light, an omnipresent light that hits all the shapes in the scene independently of their location and orientation. It can also have up to eight additional lights (This constraint is imposed by OpenGL), spotlight or directional, and each light can have different values for the ambient, diffuse and specular components. The final color of the rendered figure is computed, by OpenGL, from the position, color and material properties of each shape and the

[8]`http://www.squeaksource.com/Alien.html`

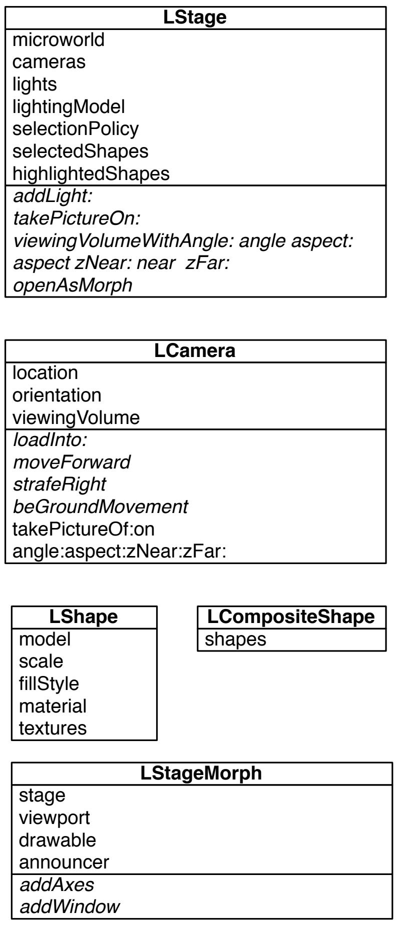


Figure 3: Lumiere Stage UML Diagram

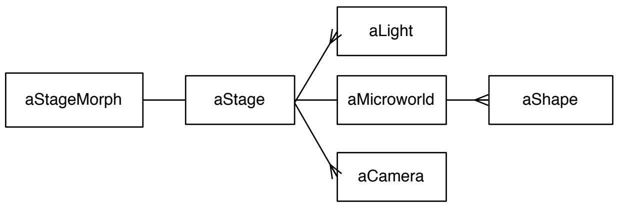


Figure 4: Lumiere Stage Model Entity Diagram

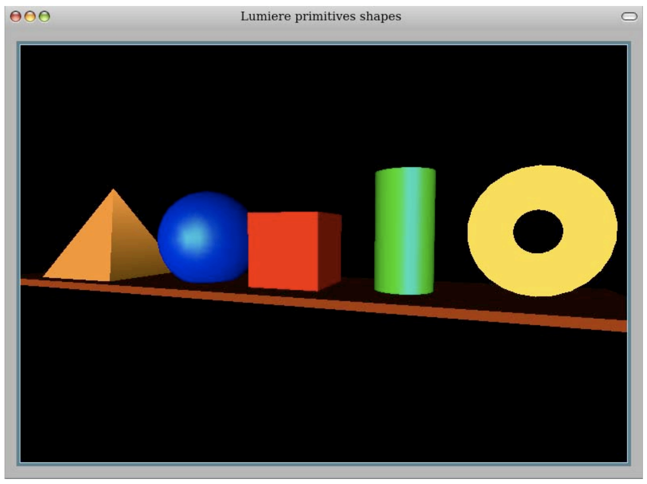


Figure 5: Lumiere primitive shapes

positions and values of each light of the stage (see Figure 6). Using Lumière it is possible to take pictures of the scenes without the lighting model (see Figure 5). In that case the color of a rendered figure is equal to the color of the shape that produced it. Figure 7 shows the same scene rendered with and without the lighting model.

Clipping and projection of 3D scenes.

A viewing volume (instance of `LViewingVolume`) dictates the portion of the scene that is visible and the kind of projection to be used by the camera, intervening directly on the appearance of the final picture rendered. The clipping is done when traversing the scene, testing bounding volumes of the nodes against the viewing volume values. For a detailed description of 3D projections see [10] and [7].

Scene graph.

A scene graph is the underlying implementation of how objects are disposed on a micro-world. A scene graph is a directed graph that holds the visual objects that form a a world and their spatial relationships. Several different types of nodes can be added to the graph. There are nodes for performing a translation, rotation or scaling. Other nodes group nodes together, and the remaining are drawable nodes (nodes that when loaded produce a figure to be rendered). These nodes can contain shapes or other renderable constructs of OpenGL like display lists, vertex arrays or vertex buffer objects (constructs for optimized rendering of OpenGL commands and polygons).

A scene graph is a convenient structure for traversing the micro-world to perform the culling and rendering. Traversing the scene graph is performed by different visitors, each one of which traverses the scene with a particular goal, implemented using the Visitor design pattern [4]. This approach eases extending the scene with additional functionality. Currently Lumière features rendering, culling and picking visitors. Culling is the process of identifying and removing all the shapes in the scene that do not appear in the final image, like for example shapes that are outside the viewing volume or occluded behind visible objects. A rendering visitor traverses the tree with a given canvas, and loads the culled nodes as they are visited. The nodes know how to load themselves into the canvas answering the polymorphic message #`loadInto:`.

Morph and Window integration.

The low level rendering is done in an opengl drawable surface, that is created and managed by the operating system. Lumière uses the Morphic framework[5] for integrating the opengl drawables into Pharo. An instance of `LDrawable` creates and manages the communication with the low level drawable, and provides access to it. A Lumière stage is visualized with a morph, an instance of `LStageMorph`. Fig-

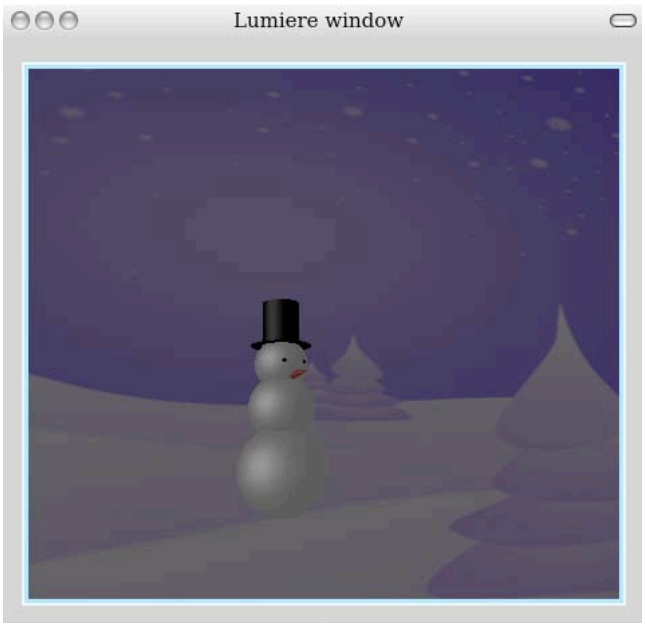


Figure 6: Lumiere composite shape

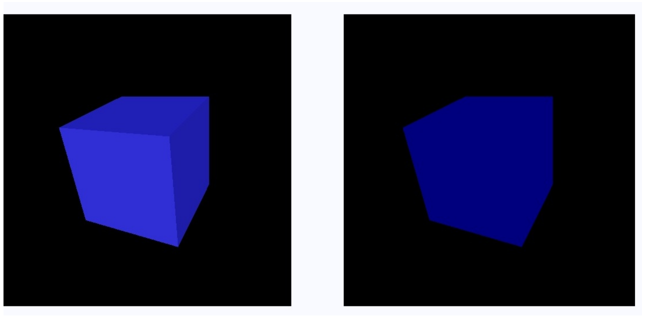


Figure 7: The effect of using a lighting model

ure 7 shows two stage morphs. A stage morph can be integrated into a Pharo system window. Common operations such as minimizing and moving the window containing the stage morph are supported.

4.4 Framework comparison

In the related work we briefly described other high level 3D frameworks in Smalltalk, then we presented Lumière. In this section we compare Lumière with Croquet and Jun, two of the most powerful and modern frameworks in Smalltalk.

Croquet is a complex 3D collaborative framework that allows the creation of rich, replicated and synchronized worlds. Croquet replaces the underlying system with an interactive 3D world. This makes Croquet hard to use it only for producing 3D graphics integrated with the rest of the system, which is the main feature that Lumière provides to programmers. Lumière is lightweight compared to Croquet, but has less features due to the different objectives of both frameworks. For example Lumière lacks the portal navigation between worlds that Croquet provides, but using Lumière is straightforward to dynamically create and visualize a world with some shapes, tasks that requires subclassing a new world in Croquet and coding an initialization method. Several abstractions present in Croquet are incorporated into Lumière, for example texture and material reification.

Jun is a powerful framework for producing 2D and 3D graphics, available in Visual Works. Jun supports an older version of OpenGL (version 1.0), and is lacking some modern features of the latest versions (which brings performance issues). Jun has licensing problems for using it in a commercial context, because it runs on Visual Works. However Jun provides good abstractions and a rich set of features to the programmer, that Lumière should incorporate. For example 2D text rendering in 3D graphics. Currently Lumière incorporates the feature of quickly opening any visual shape (called OpenGLCompoundObject in Jun and Shape in Lumière) by sending the message `#open`.

5. EXTENDING THE FRAMEWORK

Extending the framework can be easily be done because of the modular design of Lumière, the stage metaphor and the reification of several concepts such as textures, cameras and lights.

For example adding multi-textures capabilities –applying more than one texture to a shape– to Lumière shapes, would involve extending only one class, `LTexture`, creating a new polymorphic class whose instances can be associated to any shape, and know how to load themselves into the canvas during the rendering process.

Adding offscreen rendering support, such as OpenGL Frame Buffer Object, could easily be accomplished because of the rendering process of Lumière is performed by taking pictures of micro-worlds onto a canvas, and this canvas could have different destinations, like for example a Frame Buffer Object instead of normal OpenGL buffers.

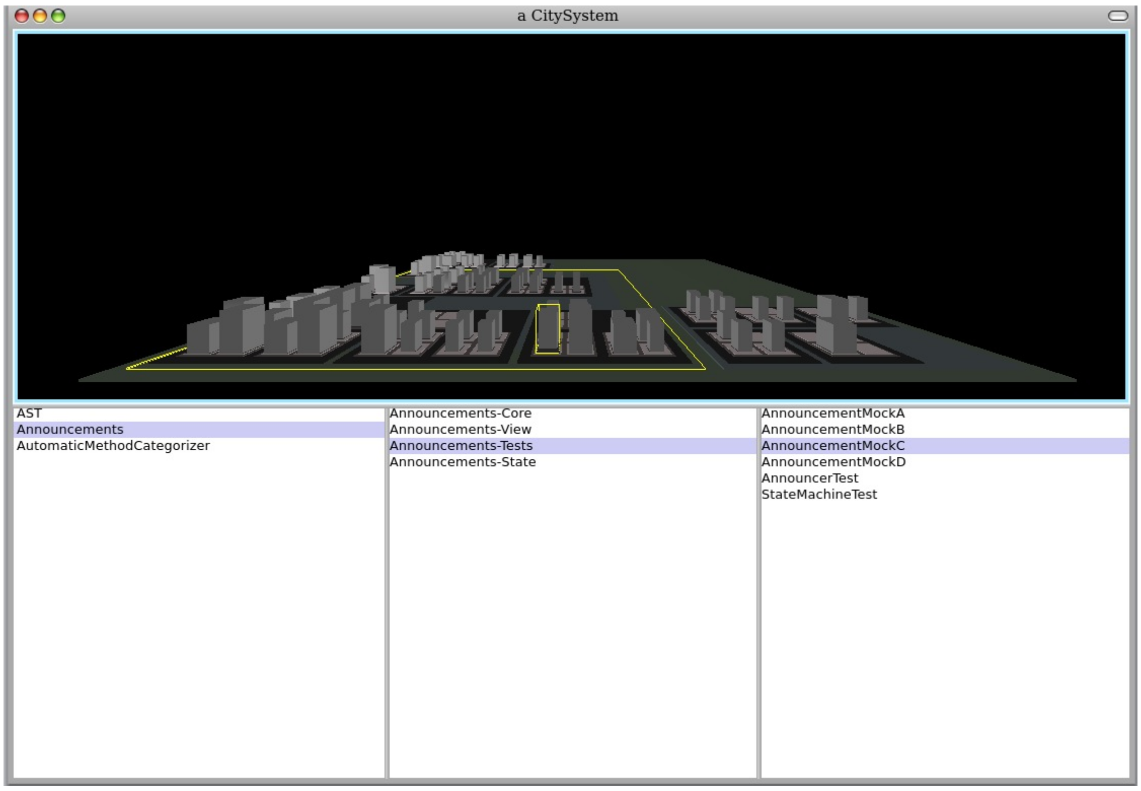
A further example of the extensibility of Lumière is the visitor infrastructure through which one can easily traverse the scene graph, as shown in the previous section.

6. PERFORMANCE

Lumière uses efficient techniques, such as Display List, Vertex Array and Vertex Buffer Object, for describing and loading into OpenGL the vertices, normals and colors that define the primitives shapes. The rendering process achieves good performance because the usage of this techniques minimizes the uploading and exchange of data between Lumière and the hardware.

Lumière also performs frustum culling -culling is the process of identifying and removing all the shapes in the scene that do not appear in the final image- thus the base renderer, OpenGL, is relieved from performing futile computations on invisible polygons.

In Figure 9 a stage with a micro-world containing 10000 cubes is displayed. Lumière does not yet provide animations, so we cannot compare the performance against other framework measured in terms of FPS (frames per second). However we scripted a camera zooming behavior, from the camera original location moving closer to the grid of cubes 8 times, and it took 11 seconds to run in average. Though performance measurements of 3D renderers heavily depends on the kind of graphics the programmer desires to produce, we believe that in its current state Lumière allows producing interactive complex micro-worlds. Moreover, the per-



AST
Announcements
AutomaticMethodCategorizer

Announcements-Core
Announcements-View
Announcements-Tests
Announcements-State

AnnouncementMockA
AnnouncementMockB
AnnouncementMockC
AnnouncementMockD
AnnouncerTest
StateMachineTest

Figure 8: A Gaucho-Glamour browser

formance can be further improved, for example the test we performed could be improved by implementing different software culling techniques such as quad tree[9] for modeling and traversing the micro-world.

7. APPLICATION: GAUCHO

In this section we present *Gaucho*, a visual programing environment we are currently building. Gaucho uses Lumière for producing an interactive 3D integrated development environment. We devised it to research alternative ways of browsing and writing programs, through direct manipulation of 3D visual objects that represent programming language constructs like packages, classes and methods.

7.1 The City Metaphor

Gaucho employs a city metaphor, where packages represent districts, classes represent buildings, and methods appartments. This metaphor and the implementation are based on CodeCity[9], a tool for visual exploration of large scale evolving software built by Wettel and Lanza, using Jun and Visual Works.

Gaucho creates a city that represents a collection of packages in the image. This city is converted to a Lumière scene for 3D visualization.

To visualize the city we created particular shapes for buildings, neighborhoods and districts by subclassing `LCompositeShape`, creating classes named `BuildingShape`, `DistrictShape`, and `NeighbourhoodShape`. The newly created shapes represent language constructs (such as packages, classes and messages), and programatically generate the composite figures to render according to certain metrics of the concrete language construct they represent. For example the height and width of a building are proportional to the number of methods and instance variables of its model, a class. The depth and width of the districts, representing packages, depend on the number and size of the classes they contain. This approach is identical to the one of CodeCity.

[9]A quad tree is a structure for implementing a scene graph that allows discarding bigger portions of the scene on each intersection test

7.2 Navigating through the city and browsing the code

Providing 3D visualizations of the models is only one of the goals of Gaucho, the other one is allowing to browse and modify programs by direct manipulation of its visualization.

Using Lumière mouse hovering event handling and floating information morph, Gaucho is able to present the properties of the language construct that is under the cursor, providing detailed information of every shape in the city whenever the user moves the mouse. Using Lumière mouse down event handling, we can select shapes in Gaucho scenes.

To produce interactive cities for browsing the programs, we integrated Gaucho cities (and therefore Lumière scenes) into a scriptable browser framework called *Glamour*[1]. Using Glamour we created a customized browser, similar to the standard browsers of Pharo, that provides full duplex synchronization between Gaucho scenes of cities and Glamour code browsers. When a shape is selected in Gaucho, the corresponding object is selected in the Glamour code

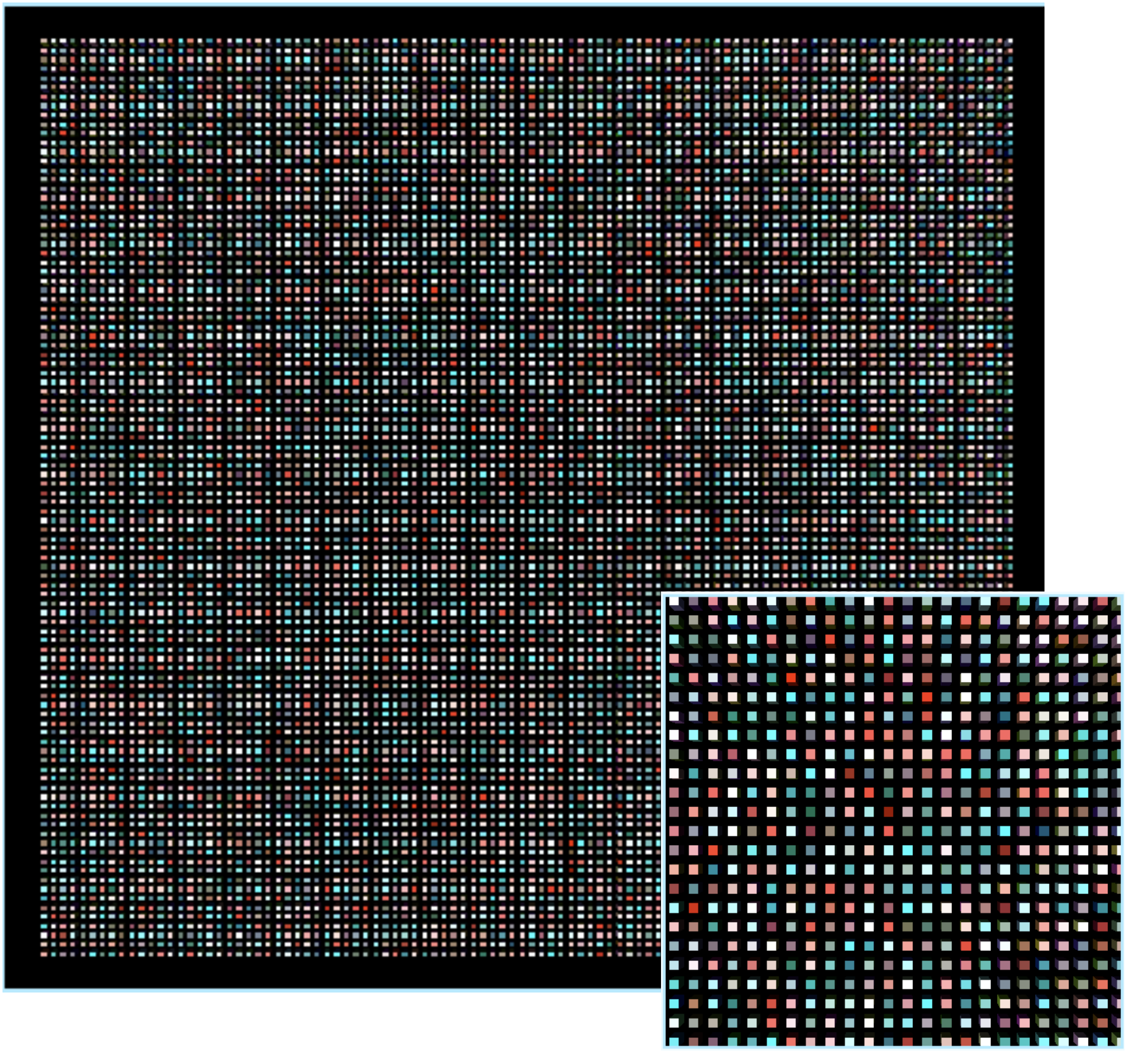


Figure 9: 10000 cubes rendered with Lumière

browser, and the same occurs while selecting an object in the browser: the corresponding city element is highlighted. In Figure 8 we show a Gaucho-Glamour browser visualizing three packages in the system.

Building a Gaucho prototype environment using Lumière, for performing the rendering and providing interaction with shapes was straightforward. The framework is simple to use and extensible enough to create applications such as Gaucho, and for integrating Lumière scenes into the Pharo standard windows, or the new scriptable Glamour browsers.

The work on Gaucho has just started, but the ease with which we could build it in this still raw form is not only a proof-of-concept for Lumière, but also serves another purpose: understanding what functionality we must implement in Lumière.

8. CONCLUSIONS

We surveyed several frameworks and tools for producing 3D graphics in Smalltalk, and showed that there is no lightweight, open source, freely usable in a commercial context and most of all modern alternative to existing frameworks. To fulfill this need we developed Lumière, a 3D framework with a stage metaphor and a scene graph implementation, built it on top of Pharo and OpenGL. We described its main features and characteristics, and detailed its implementation.

To illustrate the usage and capabilities of Lumière, we presented Gaucho, a visual programming environment we are building, that uses Lumière in order to produce interactive 3D cities for manipulating programs.

In its current state Lumière lacks several features such as texturing and multi texturing, transparency, camera animations for flying around the scene, and better implementations of the scene graph for efficient culling algorithms such as quad trees. All these features are part of the future work we plan to implement.

In this sense, we are just at the beginning.

9. REFERENCES

[1] P. Bunge. Scripting browsers with glamour. Master's thesis, University of Bern, Apr. 2009.

[2] M. J. Conway. Alice: Interactive 3d scripting for novices. Ph.d. dissertation, University of Virginia, may 1998.

[3] G. Dickie. A computer-aided music composition application using 3d graphics - research and initial design. Degree of master of science in computer science, Department of Computer Science Montana State University.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.

[5] J. H. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. In *In Proceedings of User Interface and Software Technology (UIST 95) ACM*, pages 21–28. ACM Press, 1995.

[6] K. R. Mark Guzdial. *Squeak: Open Personal Computing and Multimedia.* Prentice Hall, 2001.

[7] T. McReynolds and D. Blythe. *Advanced Graphics Programming Using OpenGL (The Morgan Kaufmann Series in Computer Graphics).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[8] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet - a collaboration system architecture, Jan. 2003.

[9] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.

[10] R. Wright, B. Lipchak, and N. Haemel. *Opengl® superbible: comprehensive tutorial and reference, fourth edition.* Addison-Wesley Professional, 2007.

APPENDIX

A. EXAMPLES OF Lumière SCENE CREATION

To further illustrate how to create Lumière scenes and populate them with shapes we present two scripts of code.

The following script creates a minimal world similar to the unlit scene cube of Figure 7. The world contains only one node, which is a drawable node that will render a blue cube.

```
cube := LShape cube.
cube color: Color blue.
cube openAsMorph.
```

The following script creates the scene in the Figure 5. The scene contains only one composite shape, that has a translation-drawable node pair for each primitive, except for the cylinder which is also rotated.

The composite is displayed as a morph sending the message #`openAsMorph`.

```
base := LShape cube.
base
color: Color brown;
scale: {  18.0. 0.2. 4.0  }.

cube := LShape cube.
cube
color: Color red lighter;
scale: {2.0. 2.0. 2.0}.

sphere := LShape sphere.
sphere
color: Color blue lighter;
scale: 3.0.

cappedCylinder := LShape cappedCylinder.
cappedCylinder
color: Color green;
scale: {  1.5. 1.5. 6.0  }.

pyramid := LShape pyramid.
pyramid
color: Color orange;
scale: {  3.0. 3.0. 3.0  }.

donut := LShape donut.
donut
color: Color yellow;
scale: {1.0. 3.0}.

primitives := LCompositeShape new.
primitives
addChild: base;
addChild: pyramid at: {  -12.0. 3.2. 0.0 };
addChild: sphere  at: {  -4.0. 2.7. 0.0 };
addChild: cube  at: {  2.0. 2.2. 0.0 };
addChild: cappedCylinder at:{8.0. 3.2. 0.0}
          rotatedBy:{90.0. 1.0. 0.0. 0.0 };
addChild: donut at: { 14.0. 3.2. 0.0 }.
primitives openAsMorph.
```