An interactive 3D interface for hybrid model specification

Viktor Vasilev

Computer Science and Engineering

London, 02 July 2017
ABSTRACT

To ease development and lower the entry barrier for new adopters many development environments offer visual means to edit complex data. Cyber-physical systems are a perfect candidate for such manipulations since they are usually described in the form of isolated, well defined components that can be manipulated individually. The physical parts of such systems often can be directly translated into real world objects and allowing the developer to interact with those in a familiar manner can greatly increate the usability and agility of the development process.

In this thesis we focus on the exploration of interactive manipulation of hybrid system models. Our research examines a solution based on the Acumen simulation environment. We describe the tight integration between the textual model and 3D visualisation, go into detailed analysis of the implementation and use case-studies to illustrate concrete applications.
It is one of the beautiful compensations in this life that no one can sincerely try to help another without helping himself.

— Ralph Waldo Emerson

ACKNOWLEDGMENTS

My deepest gratitude goes to the people at the Computing and Communication laboratory in the Embedded and Intelligent Systems research environment at Halmstad University. Professor Walid Taha was the main driving force behind the thesis project, and made it possible with his constant feedback, guidance and leadership.

I would like to thank especially Adam Duracz, Jan Duracz and Kevin Atkinson for the numerous discussions and helpful insights in Scala and the inner-workings of Acumen.

I am truly thankful for the friendship of Yuantao Fan (范元涛), Maytheewat Aramrattana, Xu Fei (徐飞) and Carlos Fuentes. They all helped me laugh, be motivated and work harder to pursue the completion of the thesis.
## CONTENTS

1 **INTRODUCTION** 1  
1.1 Problem Statement 1  
1.2 Contributions 3  
1.3 Proposed Solution 4  
1.3.1 Visual Guides 5  
1.3.2 Camera Movement 5  
1.3.3 Shape Manipulation 5  
1.3.4 Textual Model Generation and Synchronisation 6  
1.3.5 Testing and Correctness 6  
1.4 Related Work 6  
1.5 Thesis structure 6  

2 **IMPLEMENTATION** 9  
2.1 jMonkeyEngine vs Java 3D 9  
2.2 jMonkeyEngine abstractions 10  
2.2.1 The Scene Graph 11  
2.2.2 Geometry Primitives, Materials and Lighting 12  
2.2.3 Input Mapping 13  
2.2.4 Camera 14  
2.3 Integration with Acumen 14  
2.3.1 Acumen GUI Structure 14  
2.3.2 SimpleApp 15  
2.3.3 Visual Guides 16  
2.3.4 Lighting 16  
2.3.5 HUD Integration 17  
2.3.6 Views 18  
2.3.7 Information Flow 18  
2.3.8 Shapes, Picking and the Selection Box 18  
2.3.9 Keyboard Shortcuts and Mouse Controls 20  
2.3.10 Scene Transformations 20  
2.3.11 Shape Manipulation 22  
2.4 Working with the AST 24  

3 **CASE STUDIES** 31  

4 **DISCUSSION AND FUTURE WORK** 47  
4.1 Implementation Assessment 47  
4.2 Missing Features and Known Issues 48  
4.3 Suggested Improvements 49  

5 **CONCLUSION** 51  

A **APPENDIX A** 53  
B **APPENDIX B** 55  

**BIBLIOGRAPHY** 56
LIST OF FIGURES

Figure 1  The Acumen interface with the addition of the new visual modeling tab. 3
Figure 2  Interaction between the textual model and interactive interface is bidirectional. Changes in the 3D scene are reflected in to code and vice versa. 4
Figure 3  Scene graph node grouping used to isolate different parts of the application geometries. 12
Figure 4  Primitive meshes supplied by jMonkeyEngine. 12
Figure 5  Acumen’s GUI organises its features in different tabs. In this image the currently selected tab is the Visual Editor. 15
Figure 6  The visual guides as seen in the visual editor for an empty model. 17
Figure 7  The HUD showing left and right mouse button states, the currently selected node, the tool in use and view state. 17
Figure 8  A Node hosting two geometries, one for the real geometric shape and another for the selection box markers. 19
Figure 9  A sphere (real shape) with selection box highlighting it as the currently selected shape by showing red markers in each box corner. 19
Figure 10  Right mouse button context menu popup shown when a sphere was selected. 20
Figure 11  The right-handed jMonkeyEngine (a) and Acumen (b) coordinate systems. 20
Figure 12  A visualisation of the spherical coordinate system used in our implementation. The camera moves on the surface of a sphere with radius r and its position is described by two angles, θ and φ. 21
Figure 13  The dialog for deleting a single box inside `Main` class’ _3D_ section. Since there is no explicit instance of `Main`, the option to delete the create statement is disabled. 23
Figure 14  The context menu and dialog seen when adding a sphere primitive shape to a parent class. 23
Figure 15  The context menu and dialog seen when adding a class instance to a parent class. 24
LISTINGS

Listing 1  A simple Acumen model with statically positioned 3D shapes. The parameters specified for each shape are position, size, color and rotation. 2
Listing 2  Extending the SimpleApplication class. 11
Listing 3  Defining a new analog listener. 13
Listing 4  Defining a new action listener. 13
Listing 5  Method dispatch in the analog listener based on the currently selected tool and left mouse button state. 15
Listing 6  Code used for setting the camera position and orientation from the radius and longitude and latitude angles. 22
Listing 7  Interesting Abstract Syntax Tree (AST) classes. 24
Listing 8  The ClassDesc constructor. 25
Listing 9  The TDShape constructor. 25
Listing 10  The VarParam class hierarchy. 26
Listing 11  The InstanceTree constructor. 26
Listing 12  The method findTDVar of class InstanceTree and its helpers used to resolve variables into AST literals. 27
Listing 13  Class ObjData and the code used to attach and retrieve it from 3D nodes. 28

ACRONYMS

DSL  Domain Specific Language
API  Application Programming Interface
HUD  Head-Up Display
AST  Abstract Syntax Tree
JSON  JavaScript Object Notation
INTRODUCTION

One of the main reasons to conduct software simulation is that system design often involves building concrete proof of concept prototypes, which can be both costly and time-consuming. Simulation tools usually employ a Domain Specific Language (DSL) to describe the systems that are being modeled. The developer’s familiarity with the DSL and the expressiveness of the language are contributing factors to the speed of implementation and the quality of the simulation. To help improve the developer’s productivity, some simulation tools allow visual programming, which can reduce the need for the user to be proficient in the corresponding DSL. Commercial applications such as Simulink [15], Modelica [9], EcosimPro [4], MapleSim [8] and open source alternatives like Scilab/Xcos [14], ASCEND [2], JMCAD [5], and Ptolemy II [11] provide visual modeling capabilities. It is common to offer rich component libraries to further cut down the development time. The ready to use components can hide arbitrary complicated logic and be interconnected and composed to create more complex systems.

Acumen [1][24] is a tool for simulating models of cyber-physical systems. It provides an environment for development and visualisation of hybrid model simulations. Acumen’s DSL represents the systems in the form of differential equations and discrete conditions and actions. Such models describe entities that can correspond to concrete (physical) and virtual (cyber) world objects. Having a modeling tool that bridges the abstraction gap between the real world objects and their corresponding textual model can improve the development process immensely. This thesis aims to evaluate the applicability of interactive 3D design tools for manipulation of textual hybrid model specifications. In essence it tries to explore the possibility to interconnect Acumen’s DSL with a three dimensional partial view of the model that is open to interactive modifications.

1.1 PROBLEM STATEMENT

The Acumen environment uses a DSL to model cyber-physical problems. The user describes the model by applying analytical mathematics and isolating the continuous laws and discrete events that occur in the system. While the core of a model’s description is purely mathematical, some aspects are visual. 3D shapes can be attached to Acumen objects by utilizing a special _3D annotation [26] and the simulation data can be used to control the position, size, rotation and color
of these shapes. After the simulation is performed, the 3D information is used to animate the model’s behavior in space as a function of time.

The goal of this thesis is to explore ways of improving the visualisation and user interaction with the 3D shapes present in the textual model. For readers unfamiliar with the Acumen syntax, consider the following model containing four boxes positioned in 3D space:

Listing 1: A simple Acumen model with statically positioned 3D shapes.
The parameters specified for each shape are position, size, color and rotation.

class Cubes()
private
  _3D := [["Box", [2, 0, 2], [1, 1, 1], [0.1, 0.2, 0.3], [0, 0, 0]],
          ["Box", [-2, 0, 2], [1, 1, 1], [0.2, 0.3, 0.4], [0, 0, 0]],
          ["Box", [2, 0, -2], [1, 1, 1], [0.3, 0.4, 0.5], [0, 0, 0]],
          ["Box", [-2, 0, -2], [1, 1, 1], [0.4, 0.5, 0.6], [0, 0, 0]]]
end
class Cubes()
private
  _3D := [["Box", [2, 0, 2], [1, 1, 1], [0.1, 0.2, 0.3], [0, 0, 0]],
          ["Box", [-2, 0, 2], [1, 1, 1], [0.2, 0.3, 0.4], [0, 0, 0]],
          ["Box", [2, 0, -2], [1, 1, 1], [0.3, 0.4, 0.5], [0, 0, 0]],
          ["Box", [-2, 0, -2], [1, 1, 1], [0.4, 0.5, 0.6], [0, 0, 0]]]
end

The position, scale, color and rotation are encoded as vectors right after the shape name “Box”. Currently the only way to manipulate the properties of the shapes is through a text editor, which makes changes cumbersome and inconvenient. Moving and scaling in 3D space is done by modifying vectors of floating point numbers and the reason for this is that only a single textual representation of the model is available. An interactive interface on the other hand would make the applied transformations directly translate into visual changes in X, Y and Z coordinates or the width, height and depth dimension. Another obvious improvement would be the ability to choose the shape’s color from a palette, instead of modifying a vector of numbers. In contrast to 3D coordinates, where values modify the axis offsets, there is not an obvious correspondence between colors in RGB color space and floating point values.

Acumen models are composed from classes and their instances, which makes the structure inherently hierarchical in nature. There is a lot to be gained if these hierarchies can be exploited to visualize the connections and dependencies between object instances. A perfect example for this would be grouping 3D shapes based on a common origin and facilitating their movement as a group, and not separate entities. The interactive interface, by employing direct visual manipulation, would make this much easier compared to performing textual changes.

To summarize, enabling interactive modeling can benefit the users in multiple ways:

• It lowers the entry barrier and broadens the audience that is able to use Acumen effectively. This is due to the fact that visual
tools are much more user-friendly compared to a dry textual representation. The ability to interact with an object has always been a way to accelerate experimentation and ignite curiosity.

- Providing an interface similar to other commercial modeling tools can benefit long-time users. Especially familiarity with shortcuts, controls, terminology and 3D views can directly translate into productivity and comfortable use.

- It gives experienced users better tools to complete tasks which are not the core of the modeling problem. Manipulating object orientation in 3D and applying scaling and shearing can be done much quicker visually, than by editing text. This gives the user more time to focus on the actual problem that is being modeled, rather than spending time manually adjusting minute details.

- It can provide visual indicators for grouping and connecting objects, which is an aspect of the model that could be obtained easily from the 3D scene visualisation. This approach also allows interactive exploration of class interdependencies.

1.2 CONTRIBUTIONS

We design and implement an extension to Acumen that provides basic 3D modeling capabilities and does automatic synchronisation between the changes performed in through the interactive modeller and the textual model (see Figure 1).

Figure 1: The Acumen interface with the addition of the new visual modeling tab.
• Textual models can be loaded in the 3D editor.
• Keyboard shortcuts and mouse are used for navigating the scene by panning, zooming and orbiting.
• Manipulating the 3D scene is automatically reflected in the code window. The types of transformations available to the user include:
  – Movement, scaling, rotation and color change of basic shapes.
  – Addition and removal of basic shapes and class instances.

1.3 PROPOSED SOLUTION

Before proposing a solution to the stated problem, we will mention some limitations that need to be set in place. Acumen has the notion of time-zero initial conditions, which apply at the exact instant the simulation is started. Our solution considers the placement of 3D shapes in this exact instance, and not during the actual simulation. This is needed, since the shapes parameters could depend on values available only at simulation time. Another limitation is that we do not allow for arbitrary algebraic expressions to be used as 3D shape parameters, that is, the values need to be constant, so that value inference can be performed.

To face the challenges raised in the problem statement, we propose and evaluate the following extension to Acumen as a solution. The textual model serves as the original description for the 3D scene, and contains all the information associated with each and every class. Based on the properties extracted from the textual model, a 3D representation will be generated. This representation follows the Acumen semantics, so that dynamic object instantiations are depicted in the 3D scene. All present Acumen shapes are mapped to 3D meshes which are placed in space according to their position, rotation, scale and color in the textual code.

Our solution will furthermore allow the user to interact with the scene through a simple mouse- and keyboard-driven interface (see Figure 2).

Figure 2: Interaction between the textual model and interactive interface is bidirectional. Changes in the 3D scene are reflected in to code and vice versa.
1.3.1 Visual Guides

Navigation in 3D space is improved by adding visual guides to the scene. The first one is a grid lying on a plane oriented parallel to Y and Z axes and having zero elevation. The second one is a visual representation of the X, Y and Z axes of the 3D world that are centered at the origin and provide visual orientation.

1.3.2 Camera Movement

Navigating the scene is done by allowing the most common camera operations:

- Panning in a plane parallel to the direction to which the camera is facing.
- Zooming in and out.
- Rotating around a central point.

The proposed solution supports freehand camera orientation that gives the user the ability to inspect the scene from any desirable angle. It also features predefined camera angles that lock the camera viewpoint perpendicular to a given plane, for example viewing the scene from a top or bottom perspective.

1.3.3 Shape Manipulation

3D meshes are selectable in the 3D scene, which allows the user to apply interactive operations on them. A visual guide, such as highlighting, is applied to the selected mesh to ease its visual identification. All possible operations are made available through a mouse context menu and keyboard shortcuts. There is support for:

- Adding new class instances, adding primitive 3D shapes to existing classes.
- Deleting existing class instances or 3D primitives.
- Transformation of 3D primitives by applying movement, scaling and rotation.
- Optional snapping functionality. This is especially useful to avoid having to do precise mouse movement to adjust transformations by small amounts.
1.3.4 Textual Model Generation and Synchronisation

All 3D scene manipulations are translated into corresponding changes of the textual model. This means that the user always sees the source code corresponding to the currently visualized model.

1.3.5 Testing and Correctness

To have the highest quality implementation and thus low defect count we apply software testing practices. While having modular and testable code does not disprove the presence of bugs, it can improve the code quality by covering as many use cases as possible. Having a test suite also helps reduce defects introduced by future changes. Regression tests can help avoid interference between new features and those already present in Acumen.

1.4 Related Work

The areas explored in this thesis span over a wide range of topics ranging from 3D computer graphics [3] to structure editors [12]. Other simulation packages like Modelica also feature visual editors for their models. The implementation of such an extension in OpenModelica is discussed in [17] and [18] with a focus on interconnection of components in a 2D plane. Structure editing capabilities for Modelica are proposed in [23] where the focal point is to preserve source code formatting while making changes to the model. There has also been research in the area of 3D interaction in the context of 2D UIs [25]. Creating user interfaces that are comfortable to use are investigated in papers that explore human-computer interaction [19]. The work in [19] is similar to an idea explored early in the conception of this work, which tried interfacing with an already well established modeling package SketchUp [16]. Since SketchUp is extensible via Ruby language bindings, it is entirely possible to communicate with Acumen through a socket interface and exchange model data between the two. This idea was discarded as it introduces external dependencies to Acumen, which is not currently desirable. Finally we mention works that explore the applicability of visual programming paradigms. Many questions related to the viability of such interfaces are asked in [22]. It is yet to be determined how successful the results of our exploration are.

1.5 Thesis Structure

The thesis is organized into five chapters. The first and current one is INTRODUCTION, followed by a discussion of the IMPLEMENTATION details. After that we present a list of CASE STUDIES to illustrate Acu-
men features that were developed to support the thesis. Next follows DISCUSSION AND FUTURE WORK which assesses the implementation and discusses missing features and future improvements. Lastly we present the CONCLUSION.
IMPLEMENTATION

Acumen’s source code base is written mostly in the Scala programming language and is distributed under BSD and GPL licenses. Instead of using Java 3D as the current simulation animation code [26], our implementation brings into the project a new library for 3D graphics called jMonkeyEngine [6][20]. Further in this chapter we describe the advantages and disadvantages of the technologies involved and also discuss the practical implementation details.

2.1 jMonkeyEngine vs Java 3D

Prior to starting the implementation, there were considerations regarding the technologies that can be used to accomplish the work. The main concern across the people involved in the development process was that the official Sun Java 3D Application Programming Interface (API) has become a community source project, without an official maintainer and no particular roadmap for the future. On the other hand, the Sun API was already used in Acumen in the system that visualises the simulation data. The choice between employing the Java 3D technology for future projects and using a completely new solution needed to consider at least the following:

1. What is the availability of documentation and learning material for the new API? Java 3D although unmaintained has a rich set of books covering application development.

2. How maintainable is the code once it is written? Would the API be updated with new features if they are needed, and would bugfixes appear in future versions?

3. How easy is it to deploy applications depending on the API? Does the user have an easy way of installing the dependencies or can they be bundled with Acumen? What platforms and operating systems are supported?

4. How rich are the features provided by the API? Is there expectation that in future versions there will be improvements to already known shortcomings or addition of new bleeding edge features?

5. Is it distributed under an open source license compatible with the rest of Acumen?
Our search revealed that one of the best alternatives to Java 3D is the rendering engine called jMonkeyEngine. While it is popular in particular for game development, nothing stops the developer from using it for any kind of 3D application. Similar to Java 3D it offers a canvas on which the rendering is performed and comes with a scene graph functionality for organizing the 3D scene. The advantages over Java 3D were that it is being actively developed and maintained, while also having multitude of unique features, for example integration with physics engines, special effects, multimedia asset management and is distributed under a permissive BSD license. While there are not as many books available about jMonkeyEngine, it has excellent documentation, a well maintained wiki and a helpful community forum.

One goal when choosing the 3D framework was also to simplify the deployment process. The current dependency on Java 3D requires that the user installs the necessary libraries himself and this complicates the ease of use for new adopters. The ideal solution would be to bundle all dependencies with Acumen and distribute them in a single package, thus making it transparent for the user what kind of dependencies are needed at runtime. jMonkeyEngine, being primary a game development library, offers good modularity and embeddability, thus allowing fine grained library control and easier distribution.

2.2 jMonkeyEngine Abstractions

jMonkeyEngine provides useful abstractions to easily build applications. A base class called SimpleApplication is used as the main building block and users are expected to extend it and customize suitable to their application’s needs. The base class provides many convenient features [7] such as:

- Application settings (settings) containing persistent information such as preferred width and height, color bit-depth, anti-aliasing and update frequency.
- Scene graph (rootNode) maintaining a tree data structure that contains all objects present in the 3D scene.
- Asset manager (assetManager) used to organize models, textures, materials and sounds in a easy to use manner.
- User interface (guiNode) used for Head-Up Display (HUD)-like apps that need to display information on a 2D plane orthogonal to the user point of view.
- Input manager (inputManager) that manages the keyboard presses, mouse movement and clicks, and joystick input.
- Audio manager (audioRenderer) used for sound playback.
• State manager (stateManager) that allows the application to maintain and switch between different behavioral states.

• Camera (cam) which has convenient default view angle and near and far planes.

SimpleApplication also supplies routines used for initialization, starting, restarting and stopping the application, as well as event-loop integration. By extending this class and overriding its methods the user is provided with the skeleton for their application.

As in many games, the nature of our application is very reactive, since an interactive editor mostly reacts to user inputs and no proactive work is being done. This makes the structure of the app fairly simple, as illustrated in the following code snippet:

Listing 2: Extending the SimpleApplication class.

```java
class SimpleApp extends SimpleApplication {
    override def simpleInitApp() {
        // Setup Inputs
        // Setup Scene
        // Setup Camera
    }
    ...
}
```

After starting the application all the processing is done in response to user input either by keyboard presses or mouse movements and clicks. Further down we will visit some of the core components provided by jMonkeyEngine framework to better understand how they are used in our application.

2.2.1 The Scene Graph

All 3D objects drawn on screen are managed in the scene graph. It is a tree-like structure that allows nodes to be attached and detached, and provides efficient storage and traversal. jMonkeyEngine uses the term spatial to refer to the objects in the scene graph, with class Spatial containing the implementation. The main features of spatial are that they can be transformed, are interconnected and maintain parent-child relationships. They are used as a base for other classes which share the same properties, notably the Node and Geometry classes. Nodes are used for the internal nodes in the scene graph and the main purpose they serve is to group other spatial together in a collection. They are a purely structural element that is not visually rendered. A Geometry on the other hand is used as leaf node of the scene graph and contains an actual mesh and associated materials used to visually render the object. The typical jMonkeyEngine scene graphs structure is visualized in Figure 3:
By using Nodes to group spatials in the scene graph, the user can apply 3D transformations to groups of nodes. Any translation, rotation or scaling applied to a node is applied to all of its children. This makes it easy to manipulate only parts of the scene isolated by a common root node, while leaving all other nodes untouched. A useful trait of class Node is the ability to associate user-defined data with it. Our application relies upon that feature to keep a mapping between 3D shapes in the scene graph and the Acumen class instances they belong to.

2.2.2 Geometry Primitives, Materials and Lighting

Each Geometry class instance references a mesh that is used to visually render the node. jMonkeyEngine [7] supplies meshes for the many common geometric primitives such as Box, Cylinder, Sphere, Cone, Dome, Torus and others (see Figure 4).

The engine also provides non-3D meshes such as Line, Quad and Curve used for 2D and 1D objects. More complex shapes can be created by composing multiple primitive ones and attaching them to a common root Node.
To take full advantage of the rendering capabilities of jMonkeyEngine, normally a material is associated with the Geometry instance. Materials carry the properties needed to render the surface of a mesh, that is its shininess, texture, color and transparency. The engine comes with predefined solid color shaded materials, which are used in the implementation of our interactive editor. The scene graph also contains light sources which are used to illuminate the scene. The classes DirectionalLight and AmbientLight are the ones we make use of. Adding lighting to the scene also generates shadows which make navigation and depth perception in 3D space easier. Many material properties interact with the light subsystem, most notably the parameters “Ambient”, “Diffuse”, “Specular” and “Shininess”. Specifying different values for those vary the visual appearance of the meshes.

2.2.3 Input Mapping

The input manager provided by jMonkeyEngine is configured by creating mapping triggers to actions and then creating listeners that react on those actions.

- Triggers are specific events such as mouse button clicks, mouse and joystick movement and key presses.

- Mappings associate triggers with conveniently named actions. For example the left mouse button click can be associated with the action “SELECT”.

- Each action mapping is registered in a listener that executed the appropriate code when the input event is triggered.

There are three types of listeners depending on the kind of event source. The first one is called analog listener and is used when the input events form a continuous stream of data. An example for an analog listener is the one used to react on mouse movement events.

Listing 3: Defining a new analog listener.

```scala
val analogListener = new AnalogListener {
  def onAnalog(name: String, value: Float, tpf: Float) { ... }
}
```

The other type is called action listener and deals with discrete events which signify a state change, such as key is pressed or mouse button is clicked.

Listing 4: Defining a new action listener.

```scala
val actionListener = new ActionListener {
  def onAction(name: String, keyPressed: Boolean, tpf: Float) { ... }
}
```
The third kind is a touch listener which is specially designed for work with touchscreen devices and handles tapping and multi-finger gestures, but is not used in our code.

2.2.4 Camera

jMonkeyEngine has a camera class that allows easy positioning and orientation. The direction the camera is facing is defined by specifying the left axis, up axis and viewing direction. The SimpleApplication class provides a default camera with sensible frustum values for the far and near, left and right and top and bottom planes. The framework also supports chase and fly-by camera modes, but they are not very useful in our application scenario.

jMonkeyEngine provides a useful mechanism to present information in a HUD fashion. Text and images attached to the HUD node are rendered on a plane orthogonal to the user’s viewing direction. The HUD can be used for providing immediate visual feedback to the user or for debugging purposes. jMonkeyEngine integrates with Nifty GUI [10], a framework for rich embeddable 2D interfaces. Nifty uses XML documents to describe the GUI layout, group the elements in panels and layers and map inputs to action handlers. While very powerful, it was not used in our application.

2.3 Integration with Acumen

2.3.1 Acumen GUI Structure

Many Acumen features are presented to the user in the form of tabs in the GUI (see Figure 5). The simulation data is accessible in multiple forms, each one hosted in a separate tab:

- **Plot** A view of instances and variables graphed as continuous time series.
- **Table** Tabular view of instance and variable data.
- **3D** Three-dimensional simulation animation view.

The interactive editor is also hosted in a tab with the contents residing in VisualEditorTab and the jMonkeyEngine application in VisualEditorView. Currently the view comprises simply the 3D canvas, painted by jMonkeyEngine, and an “Update Scene” button used to parse the textual model and creates the 3D scene. This manual scene update approach was the simplest way to implement the functionality without the need to synchronise access to the code area and the 3D scene. Automatic (on-the-fly) updates represent a desirable feature and could be examined in future work.
2.3.2 SimpleApp

The core of our implementation lies in SimpleApp, a class extending jMonkeyEngine’s SimpleApplication. It contains the state of the whole interactive editor and is where most of the logic resides. We override the simpleInitApp method with our initialization code. This method is called by SimpleApplication’s startCanvas method when the tab becomes visible. After the application is started, all events are handled reactively. The initialization function sets up the input handlers, the scene visual guides, the lights and the camera. From this point on the user interactions, in the form of mouse movements, clicks and keyboard presses are handled and delegated to the corresponding callbacks.

The analog listener, used for event streams (for example mouse movement) does very simple dispatch based on the state of the left mouse button and the currently selected tool:

Listing 5: Method dispatch in the analog listener based on the currently selected tool and left mouse button state.

``` scala
val analogListener = new AnalogListener {
  def onAnalog(name: String, value: Float, tpf: Float) {
    if (lmbrPressed) {
      curToolMode match {
        case ToolMode.Pan => handleCameraPan(name, value, tpf)
        case ToolMode.Orbit => handleCameraRotation(name, value, tpf)
        case ToolMode.Zoom => handleCameraZoom(name, value, tpf)
        case ToolMode.Move | ToolMode.Rotate | ToolMode.Scale => handleObjectTransform(name, value, tpf)
      }
    }
  }
}
```
This handler is called repeatedly with the name of the mapping, the value by which the axis changed and the time per frame (which depends on the current frames per second count [7] and is used to achieve smooth animations). ToolMode is an enumeration of all possible modes the editor can be in. These are: Select, Pan, Orbit, Zoom, Move, Rotate and Scale. The ViewMode on the other hand tracks the currently selected view mode, which is one of Isometric, Top, Front, Left, Right and Back.

The action listener, used to keep track of the keyboard events, contains a bit more complex logic. All keyboard shortcuts and mouse clicks are dispatched here. These include:

- Recording the position of left and right mouse button presses.
- Keeping the state of the left and right mouse buttons.
- Creating and showing the right mouse button pop-up menu.
- Keeping the state of the ALT, CMD and CTRL keys.
- Changing the tool state for Select, Pan, Orbit, Move, Rotate and Scale.
- Applying the camera transform for Isometric, Top, Front, Left, Right and Back scene views.

2.3.3 Visual Guides

The editor utilizes a grid and X, Y and Z axis guides to aid the user in navigating 3D space and ease debugging (see Figure 6).

The grid is equidistant with 1 world unit between the lines. The axes are colored in the same manner as the _3D panel: red for X, green for Y and blue for Z. While Grid is a class provided by jMonkeyEngine, the Axes are custom created nodes build form cylinders with cones at the top pointing in the positive direction and placed at the origin of the scene.

2.3.4 Lighting

While lighting is not crucial for our application, it is needed to make improve the user’s perception of multiple dimensions. The world is lit by three static light sources. Two are directional lights with light and dark color oriented opposite to each other and one is a white ambient light. The directional lights are global and have only a direction, but no source. The ambient light affects the whole scene by modifying the brightness and color of the whole scene [7]. The directional lights, in
contrast to the ambient light, affect the shadows cast onto the world and improve the eye depth perception.

2.3.5 **HUD Integration**

We make use of the HUD functionality to provide various state feedback to the user. The following information is relayed on the HUD surface area:

- The state of the mouse buttons.
- The current selected node label.
- The current tool mode of the application.
- The current view mode of the application.

This information is always visible in the lower left corner of the canvas (Figure 7):

![HUD showing left and right mouse button states, the currently selected node, the tool in use and view state.](image)

Figure 7: The HUD showing left and right mouse button states, the currently selected node, the tool in use and view state.
2.3.6 Views

Our application offers several predefined scene views. By utilizing keyboard shortcuts the user can position the camera in one of the following ways:

- Isometric (Cmd+1, ESC) oriented from a 45 degree angle towards X, Y and Z.
- Top (Cmd+2) oriented perpendicular to the X-Y plane looking towards -Z.
- Front (Cmd+3) oriented perpendicular to the Y-Z plane looking towards -X.
- Left (Cmd+4) oriented perpendicular to the X-Z plane looking towards -Y.
- Right (Cmd+5) oriented perpendicular to the X-Y plane looking towards +Y.
- Back (Cmd+6) oriented perpendicular to the X-Y plane looking towards +X.

2.3.7 Information Flow

The visual editor uses a button to synchronise the TextArea used for Acumen code and the 3D canvas, while the reverse synchronisation is done automatically. Any changes performed in the canvas will be immediately visible in the code. The button used to update the 3D scene causes the model code to be parsed, transformed into a tree data structure that contains all shape instances and then visualized on the canvas. Further transformations, additions or deletion of shapes and classes are reflected seamlessly in the Acumen model.

2.3.8 Shapes, Picking and the Selection Box

The visual editor uses Nodes to organize the geometries in the scene. Each shape visualized in the model is represented as a Geometry instance attached to a Node. Whenever the shape is selected or deselected, a selection box Geometry comprising a box with its corners highlighted is also attached or detached (see Figures 8 and 9). The selection box geometry is purely a visual cue to identify the currently selected shape.

Picking is handled entirely by jMonkeyEngine and is done by casting rays into the scene and calculating their intersection with the present shapes. The algorithm is implemented in the handlePicking method and works in the following manner:
Figure 8: A Node hosting two geometries, one for the real geometric shape and another for the selection box markers.

Figure 9: A sphere (real shape) with selection box highlighting it as the currently selected shape by showing red markers in each box corner.

- The 2D mouse screen coordinates at which the user pressed the left mouse button are converted to 3D world coordinates.
- A ray is cast from this 3D point in the direction pointing away from the camera.
- The whole scene model tree is checked for collisions with the ray.
- All resulting collisions are sorted by distance and the one closest to the user is picked.
- In case there was a closest collision, the currently selected node is updated. This is done by clearing the selection box of the currently selected node and adding a new one to the closest colliding shape node.
2.3.9 Keyboard Shortcuts and Mouse Controls

All actions available to the user are accessible through the right mouse button context menu (see Figure 10) and most of them are bound to keyboard shortcuts. The left mouse button is used to actuate the scene and object transformations, that is the user must hold the button to pan, zoom, orbit the scene or scale, zoom or rotate a shape. An exhaustive list of keyboard shortcuts is available in Appendix A.

![Right mouse button context menu popup shown when a sphere was selected.](image)

2.3.10 Scene Transformations

Both jMonkeyEngine and Acumen use right-handed Cartesian coordinate systems [13], but while the former has the Z axis pointing upwards, the latter has it towards the user (see Figure 11). This distinction is important when interpreting the _3D annotations and translating the coordinates to the scene. The axis in both the _3D view and the new interactive editor share the same colors, red for X, green for Y and blue for Z.

![The right-handed jMonkeyEngine (a) and Acumen (b) coordinate systems.](image)
Our implementation uses a spherical coordinate system for the camera positioning. Arbitrary camera positions are identified by two angles, $\theta$ (longitude) and $\phi$ (latitude) and a radius $r$ (see Figure 12). When orbiting the scene, the user mouse input is translated into small increments or decrements of $\theta$ and $\phi$, while the radius is preserved. Zooming in and out on the other hand preserves the angles while modifying the radius.

![Figure 12: A visualisation of the spherical coordinate system used in our implementation. The camera moves on the surface of a sphere with radius $r$ and its position is described by two angles, $\theta$ and $\phi$.](image)

We derive the exact equations needed to convert between spherical and 3D Cartesian coordinates. From the information shown in Figure 12, we know:

$$\sin(\phi) = \frac{d}{r}$$
$$\sin(\theta) = \frac{b}{d}$$
$$\cos(\phi) = \frac{c}{r}$$
$$\cos(\theta) = \frac{a}{d}$$

After simple rearrangements we arrive at:

$$d = r \times \sin(\phi)$$
$$b = d \times \sin(\theta) = r \times \sin(\phi) \times \sin(\theta)$$
$$c = r \times \cos(\phi)$$
$$a = d \times \cos(\theta) = r \times \sin(\phi) \times \cos(\theta)$$
After these transformations we have the three Cartesian coordinates $a$ for $X$, $c$ for $Y$ and $b$ for $Z$ for the position described with angles $\phi$, $\theta$ and radius $r$.

Listing 6: Code used for setting the camera position and orientation from the radius and longitude and latitude angles.

```java
camPos.x = (curRadius * math.sin(FastMath.DEG_TO_RAD * myPhi) * math.cos(FastMath.DEG_TO_RAD * myTheta)).toFloat
camPos.y = (curRadius * math.cos(FastMath.DEG_TO_RAD * myPhi)).toFloat
camPos.z = (curRadius * math.sin(FastMath.DEG_TO_RAD * myPhi) * math.sin(FastMath.DEG_TO_RAD * myTheta)).toFloat

cam.setLocation(camPos)
cam.lookAt(new Vector3f(0f, 0f, 0f), new Vector3f(0f, 1f, 0f))
```

In the SimpleApp class, the functions that encapsulate the camera positioning code are:

- `setCameraFromAngles(myTheta: Float, myPhi: Float)`: Performs the calculations described above to position the camera accordingly and point its direction towards the origin of the scene.

- `rotateCamFromMousePos(lastMousePos: Vector2f, curMousePos: Vector2f)`: Considers the current and last known mouse position to calculate phi and theta deltas and update the camera position by calling `setCameraFromAngles`.

There is also a function `moveCameraPreserveAngles(vec: Vector3f)` that is used by the code handling zooming in the different views (isometric, top, front, left, right and back). It is always called with a vector that has two components zero and one non-zero and thus modifies only the radius of the camera positioning sphere.

2.3.11 Shape Manipulation

Move, rotate and scale transformations can be applied to a shape that has been selected. The analog listener continuously calls the corresponding transformation handler with updated mouse axis information and position delta.

Movement is handled in `handleTransformMove` by modifying the selected node’s translation in a plane that is perpendicular to the direction the camera is looking at. Changes in the Z dimension (mouse wheel) are ignored to achieve the 2D planar movement. Scaling on the other hand is done in `handleTransformScale` by growing or shrinking the selected node with a fixed amount on every call of the handler.

The rotation transformation in `handleTransformRotate` is performed by modifying the rotation angles by fixed amount on every handler call.

A color transformation is not bound to the mouse positioning, so a separate color chooser dialog is presented to the user. The RGB
value returned by the dialog is used to update the selected node’s color information.

The delete parent and delete node transformations also use dialogs (see Figure 13). The user needs to select a node in the scene, after which deleting it can be done by either removing the corresponding shape from the _3D section, or by deleting the create statement that creates an instance of the class containing the _3D section.

Figure 13: The dialog for deleting a single box inside Main class’ _3D section. Since there is no explicit instance of Main, the option to delete the create statement is disabled.

Our application also gives the ability to add either primitive shapes (see Figure 14) or an instance of classes (see Figure 15) present in the model. Shapes are added to the _3D section of a class selected through a dropdown menu. They default to position fixed at world origin, a neutral color and size of 1 world unit. Instances on the other hand present a bigger challenge, since their constructors can receive parameters and automatic instantiation can not guess which variables or constants to pass in.

(a) Add shape  (b) Add shape to parent

Figure 14: The context menu and dialog seen when adding a sphere primitive shape to a parent class.

When instantiating a class, the newly created variable will be named by:

• Taking the first letter of the class name.
- Converting it to lowercase.
- Appending a number, so that the new name does not conflict with variable names already present in this class.

In case that the class constructor has arguments, the new instance will populate the arguments with placeholders in a manner similar to naming the instance variables. For example, an instance of class Test(x,y,z) could be t0 := create Test(x0,y0,z0), given that t0, x0, y0 and z0 are unique.

![Tools](image1.png)

(a) Add instance

![Add to Parent](image2.png)

(b) Add instance to parent

Figure 15: The context menu and dialog seen when adding a class instance to a parent class.

## 2.4 Working with the AST

The first step when converting an Acumen model to its 3D representation is to parse the program into an AST. The AST structure follows closely the language grammar. At a very high level the program consists of a series of class declarations described by class ClassDef in the AST (see Listing 7).

Listing 7: Interesting AST classes.

```scala
case class ClassDef(name: ClassName, fields: List[Name],
                    priv: List[Init], body: List[Action])

case class Init(x: Name, var rhs: InitRhs)

sealed abstract class Expr
case class Lit(gv: GroundValue) extends Expr

sealed abstract class GroundValue
case class GInt(i: Int) extends GroundValue
case class GDouble(d: Double) extends GroundValue
case class GBool(b: Boolean) extends GroundValue
case class GStr(s: String) extends GroundValue
```
Each class has a name, constructor arguments, a list of Init statements in the private section and a list of actions in the body. For our application we only look into the private section that describes the initial conditions, in contrast to the body which is subject to evaluation while the simulation is running. There are two kinds of Init statements. One is the assignment to a variable \( x := 5 \) and the other the instantiation of a class \( t := \text{create Table}() \). The right-hand side of the Init statements can further be reduced to expressions and later to values. We’re especially interested in the Lit class that encapsulates values of different types. A GroundValue can hold an integer, a floating point number, a boolean or a string.

All changes to the 3D visualisation are reflected into the AST by modifying the corresponding Lit object. Converting back to the textual representation is a matter of traversing the AST and printing all elements.

Our application analyzes the AST to create its own class representations, internal to the visual editor. The class containing this information is called ClassDesc:

Listing 8: The ClassDesc constructor.

```scala
class ClassDesc(val name: String,
    val args: Seq[String],
    val threeDs: Seq[TDShape],
    val locals: List[Local],
    val insts: List[String])
```

The constructor parameters of ClassDesc contain all the information needed to infer local variable and instance information:

- **name** The class name.
- **args** The constructor arguments names.
- **threeDs** The contents of the _3D section if any.
- **locals** All variables local to the private section.
- **insts** All variables used as instance names in create statements.

The constructor arguments in **args** are kept in an ordered data structure **Seq**, and the same applies for the _3D elements in **threeDs**. Their positions are important for the instantiation and correspondence to visual elements in the jMonkey canvas. The type used for the _3D elements is **TDShape**, which is an abstract class inherited by **TDBox**, **TDSphere** and **TDCylinder**:

Listing 9: The TDShape constructor.

```scala
sealed abstract class TDShape(val pos: TDPos,
    val size: TDSize,
    val color: TDColor)
```
The arguments TDPos, TDSize and TDColor of TDShape are abstractions to give unified types for the position, size and color of the various shapes and ultimately resolve into VarParams (see Listing 10).

<table>
<thead>
<tr>
<th>sealed abstract class</th>
<th>VarParam</th>
</tr>
</thead>
<tbody>
<tr>
<td>case class Constant(lit: Lit)</td>
<td>extends VarParam</td>
</tr>
<tr>
<td>case class Implied(l: List[VarParam])</td>
<td>extends VarParam</td>
</tr>
<tr>
<td>case class Local(name: Name, l: List[VarParam])</td>
<td>extends VarParam</td>
</tr>
<tr>
<td>case class Argument(name: Name)</td>
<td>extends VarParam</td>
</tr>
</tbody>
</table>

Most notably, the dimensions of different shapes are given by vectors of different sizes. For a box, it is a vector with 3 elements (x, y, z), while a cylinder has 2 (height, radius) and a sphere only one (radius). A VarParam is an abstract class with four case classes, used to describe the possible sources of a variable:

1. **Constant** is an integer or floating point constant value local to an expression.
2. **Local** is a variable with assigned value local to the class.
3. **Implied** is a composite variable used anonymously. For example \([1, v2, z]\) in the expression \(x := \text{create X([1,v2,z])}\) is implied. It can not be referenced by name like a Local variable and also could contain values that could be passed as constructor arguments and need further inference.
4. **Argument** is an argument passed through the constructor.

By using the class description information, we create a tree representation of the Acumen model. This is done by the class called InstanceTree:

| class InstanceTree(val expr: Option[Init], |
| val name: String, |
| val parent: Option[InstanceTree], |
| val cls: ClassDesc, |
| val args: Seq[VarParam]) |

The constructor arguments contain the following data:

- **expr** A reference to the corresponding AST Init instance.
- **name** The instance name.
- **parent** A reference to the parent instance.
- **cls** A reference to the class description information.
args The parameters used to create the instance.

The `extractInstances` method of `InstanceTree` delivers a flattened list of instantiations done by `create` statements from the whole model. Visualisation is done in method `visualizeInstance` of `SimpleApp` by iterating through all `InstanceTree` objects, whose class description indicates that they have a non-empty `_3D` section. The `InstanceTree` class features a series of methods, `findTDColorVars`, `findTDSizes` and `findTDPo` that are used to resolve the position, size and color variables of the 3D shapes into `Lit` object of the `AST`. Most of the work is done by the `findTDVar` function, that takes a `VarParam` as argument and delivers the list of corresponding `AST` literals (see Listing 12). We have mentioned that a `VarParam` has four concrete case classes, each of which is handled as follows:

1. Constant contains a direct reference to the `AST` literal.
2. Local and Implied are resolved by applying `findTDVar` to each of the `VarParams` in their internal parameter list. To visualize this, consider a local declaration `x := [a, 1, 2, p]`. The `member` over which we map contains the `VarParams` for each of `a, 1, 2` and `p` and `findTDVar` will recursively proceed to resolve them and return the corresponding literals.
3. Argument is resolved by looking up the constructor argument index and calling `findArg` recursively on the parent class instance.

Listing 12: The method `findTDVar` of class `InstanceTree` and its helpers used to resolve variables into `AST` literals.

```scala
def findTDVar(vp: VarParam): List[Lit] = {
  vp match {
    case c:Constant => findConstant(c)
    case i:Implied => findImplied(i)
    case l:Local => findLocal(l)
    case a:Argument => findArg(a, cls.args.indexWhere(_ == a.name.x))
  }
}

def findLocal(loc: Local): List[Lit] = {
  loc.l.map(findTDVar).flatten
}

def findImplied(i: Implied): List[Lit] = {
  i.l.map(findTDVar).flatten
}

def findConstant(c: Constant): List[Lit] = {
  List(c.lit)
}
```
def findArg(a: Argument, idx: Int): List[Lit] = {
  args(idx) match {
    case c: Constant => findConstant(c)
    case i: Implied => findImplied(i)
    case l: Local => parent.get.findLocal(l)
    case arg: Argument => {
      val par = parent.get
      val pClass = par.cls
      par.findArg(arg, pClass.args.indexWhere(_ == arg.name.x))
    }
  }
}

The Lits hold the actual values as present in the Acumen model text. Modifications in the 3D scene are carried by altering the Lit objects corresponding to the properties being modified. To facilitate this, each 3D shape has a reference to an object of class ObjData (see Listing 13), which connects the 3D shapes with the corresponding AST Lit objects used as size, position and color. jMonkeyEngine provides an API to attach arbitrary key-value data to Nodes:

Listing 13: Class ObjData and the code used to attach and retrieve it from 3D nodes.

```scala
class ObjData(val name: String,
               val inst: InstanceTree,
               val idx: Int,
               val origin: List[Lit],
               val offset: List[Lit],
               val size: List[Lit],
               val color: List[Lit])

// Attach object data
node.setUserData("ObjData", objData)

// Retrieve object data
objData = selectedNode.getUserData[ObjData]("ObjData")
```

Notice that the ObjData class has an offset parameter. This offset is used to allow classes to use an expression for the position of a 3D shape and thus allow shapes to share a common origin. Consider this example with multiple instances of the same class at different positions:

```scala
class Main(simulator)
  private
    t1 := create Table([-4,0,-4]);
    t2 := create Table([4,0,-4]);
    t3 := create Table([-4,0,4]);
    t4 := create Table([4,0,4]);
  end
end

class Table(pos)
private
  _3D:= ["Box", pos + [0, 0, -0.05], ...],
  ["Box", pos + [-2.86, 1.41, -0.63], ...],
  ["Box", pos + [-2.86, -1.41, -0.63], ...],
  ["Box", pos + [2.86, -1.41, -0.63], ...],
  ["Box", pos + [2.86, 1.41, -0.63], ...],
  ["Box", pos + [0, 0, 0.123], ...],
  ["Box", pos + [0, 0, 0], ...]]
end
end

Modification in the visual editor of the position of any table element will result in change in the origin literals for that object and will leave the position of the other instances unchanged.

After manipulating 3D shapes, the visual editor will use the ObjData instance methods updatePos, updateSize and updateColor to carry out the actual AST modifications. Creating the updated textual Acumen model is done by the pretty printer already present in Acumen.
CASE STUDIES

In this chapter we present Acumen models, their visualisation and transformations, along with user-applied interactions as use-cases for the functionality implemented in our solution. The Acumen code contains highlights on elements which change during the user interaction.

CASE 1

An empty model without _3D annotation demonstrating the visual guides (grid and axes).

```c#
class Main(simulator)
    private
    end
end
```
CASE 2

2a. A model with _3D annotation in class Main. One sphere centered at origin.

```java
class Main(simulator)
  private
    _3D:= ["Sphere", [0, 0, 0], 1, [0, 1, 1], [0, 0, 0]]
  end
end
```

ACTION: apply interactive scaling

2b. Same model as Case 2a with scaled sphere.

```java
class Main(simulator)
  private
    _3D:= ["Sphere", [0, 0, 0], 4.75, [0, 1, 1], [0, 0, 0]]
  end
end
```
CASE 3

3a. A model with _3D annotation in class Main. One sphere centered at origin.

```java
class Main(simulator)
    private
        _3D:= ["Sphere", [0, 0, 0], 1, [0, 1, 1], [0, 0, 0]]
    end
end
```

ACTION: apply interactive move

3b. Same model as Case 3a with moved sphere.

```java
class Main(simulator)
    private
        _3D:= ["Sphere", [-3.11, -6.36, 1.69], 1, [0, 1, 1], [0, 0, 0]]
    end
end
```
CASE 4

4a. A model with 3D annotation in class Main. One sphere centered at origin.

```java
class Main(simulator)
    private
        _3D:= ["Sphere", [0, 0, 0], 1,
               [0, 1, 1], [0, 0, 0]]
    end
end
```

ACTION: apply color change

4b. Same model as Case 4a with changed color.

```java
class Main(simulator)
    private
        _3D:= ["Sphere", [0, 0, 0], 1,
               [0.4, 0.4, 0.6], [0, 0, 0]]
    end
end
```
5a. A model with a class instance created in Main. One class with a _3D annotation holding a box centered at origin. Scale passed as argument through constructor.

```java
class Main(simulator)
    private
        e:= create Example([1,1,1])
    end
end
class Example(s)
    private
        _3D:= ["Box", [0, 0, 0], s, [0.2, 1, 0.2], [0, 0, 0]]
    end
end
```

**ACTION:** apply interactive scaling

5b. Same model as Case 5a with scaled box.

```java
class Main(simulator)
    private
        e:= create Example([3.07, 3.07, 3.07])
    end
end
class Example(s)
    private
        _3D:= ["Box", [0, 0, 0], s, [0.2, 1, 0.2], [0, 0, 0]]
    end
end
```
### Case 6

6a. A model with class instance created in Main. One class with _3D holding a cylinder centered at origin. Position passed as argument through constructor.

```plaintext
class Main(simulator)
  private e := create Example([0, 0, 0])
end

class Example(p)
  private
    _3D := ["Cylinder", p, [1, 4, 0.5], [0.2, 1, 0.2], [0, 0, 0]]
  end
end
```

**ACTION:** apply interactive move

6b. Same model as Case 6a with moved cylinder.

```plaintext
class Main(simulator)
  private
    e := create Example([-6.54, -6.62, -0.14])
  end
end

class Example(p)
  private
    _3D := ["Cylinder", p, [1, 4, 0.5], [0.2, 1, 0.2], [0, 0, 0]]
  end
end
```
CASE 7

7a. A model with class instance created in Main. One class with _3D holding a sphere centered at origin.

```java
class Main(simulator)
    private
e:= create Example([0.2, 0, 0.2])
end
end
class Example(c)
    private
        _3D:=["Sphere", [0, 0, 0], 1, c, [0, 0, 0]]
    end
end
```

ACTION: apply color change

7b. Selecting the sphere and choosing “Change Color” menu item opens the color dialogue.

7c. Same model as Case 7a with changed color.

```java
class Main(simulator)
    private e:= create Example([0.8, 0.4, 0]) end
end
class Example(c)
    private _3D:=["Sphere", [0, 0, 0], 1, c, [0, 0, 0]] end
end
```
CASE 8

8a. An empty model.

```plaintext
class Main(simulator)
    private
    end
end
```

ACTION: add primitive shape

8b. Context menu used to invoke the dialog that adds primitive shape to a class.

8c. Dialog for adding a primitive, featuring a drop-down menu to pick a class to host the primitive.

8d. A model with _3D in Main class containing a Cylinder.

```plaintext
class Main(simulator)
    private
        _3D= [
            "Cylinder", [0, 0, 0], [1, 5], [0.3, 0.7, 0.2], [0, 0, 1]
        end
    end
end
```
**CASE 9**

9a. A model with one empty Example class besides Main.

```java
class Main(simulator)
    private end
end
class Example()
    private end
end
```

**ACTION:** add primitive to Example class

9b. Dialog to choose class to which to add primitive.

9c. A model with Example class with one 3D primitive. The 3D view is unchanged, since there is no instance of the class.

```java
class Main(simulator)
    private end
end
class Example()
    private
        3D := [
            "Box", [0, 0, 0], [1, 1, 1], [0.3, 0.7, 0.2], [0, 0, 0]
        ]
    end
end
```

**ACTION:** add instance of Example class to Main
9d. Menu to chose class to instantiate.

9e. Dialog to choose host to receive class instance.

9f. A model with class with a primitive, instantiated in Main.

```plaintext
class Main(simulator)
    private
        e0:= create Example()
    end
end
class Example()
    private
        _3D:= ["Box", [0, 0, 0], [1, 1, 1], [0.3, 0.7, 0.2], [0, 0, 0]]
    end
end
```

ACTION: add instance of Example class to Main
CASE 10

10a. A model with a class with a primitive, instantiated in Main.

```java
class Main(simulator)
    private e0 := create Example() end
end
class Example()
    private
        _3D := ["Box", [0, 0, 0], [1, 1, 1], [0.3, 0.7, 0.2], [0, 0, 0]]
    end
end
```

ACTION: delete primitive
OR
ACTION: delete instance

10b. A dialog asking to specify what to delete. Options are the object from the _3D section it where belongs, or the instance that creates this object.

10c. Depending on if we choose to delete the primitive or the instance we get:

```java
class Main(simulator)
    private e := create Example() end
end
class Example()
    private end
end
```

OR

```java
class Main(simulator)
    private end
end
class Example()
    private
        _3D := ["Box", [0, 0, 0], [1, 1, 1], [0.3, 0.7, 0.2], [0, 0, 0]]
    end
end
```
CASE 11

11a. A model with a class that has multiple primitives that share a common origin:

```plaintext
class Main(simulator)
private
  b := create Boxes([0, -4, 0]);
end
end
class Boxes(pos)
private
  _3D := [
    ['Box', pos + [2, 0, 2], [1, 1, 1], [0.1, 0.2, 0.3], [0, 0, 0]],
    ['Box', pos + [-2, 0, 2], [1, 1, 1], [0.2, 0.3, 0.4], [0, 0, 0]],
    ['Box', pos + [2, 0, -2], [1, 1, 1], [0.3, 0.4, 0.5], [0, 0, 0]],
    ['Box', pos + [-2, 0, -2], [1, 1, 1], [0.4, 0.5, 0.6], [0, 0, 0]]
  ]
end
end
```

ACTION: interactive move to single primitive

11b. Since the origin is shared, moving one box results in all of them being moved. The change in the position is reflected in the origin and not individual box offsets.

```plaintext
class Main(simulator)
private
  b := create Boxes([3.37, 0.09, 1.22])
end
end
class Boxes(pos)
private
  _3D := [
    ['Box', pos + [2, 0, 2], [1, 1, 1], [0.1, 0.2, 0.3], [0, 0, 0]],
    ['Box', pos + [-2, 0, 2], [1, 1, 1], [0.2, 0.3, 0.4], [0, 0, 0]],
    ['Box', pos + [2, 0, -2], [1, 1, 1], [0.3, 0.4, 0.5], [0, 0, 0]],
    ['Box', pos + [-2, 0, -2], [1, 1, 1], [0.4, 0.5, 0.6], [0, 0, 0]]
  ]
end
end
```
**CASE 12**

12a. A model with a class containing multiple primitives that share a common size.

```java
class Main(simulator)
    private
        b := createBoxes([1,1,1]);
    end
end
class Boxes(scale)
    private
        _3D := ["Box", [2, -4, 2], scale, [0.1, 0.2, 0.3], [0, 0, 0]],
            ["Box", [-2, -4, 2], scale, [0.2, 0.3, 0.4], [0, 0, 0]],
            ["Box", [2, -4, -2], scale, [0.3, 0.4, 0.5], [0, 0, 0]],
            ["Box", [-2, -4, -2], scale, [0.4, 0.5, 0.6], [0, 0, 0]]
    end
end

ACTION: scale single primitive

12b. Since the scale is shared, resizing one box results in all of them being resized.

```java
class Main(simulator)
    private
        b := createBoxes([1.63, 1.63, 1.63]);
    end
end
class Boxes(scale)
    private
        _3D := ["Box", [2, -4, 2], scale, [0.1, 0.2, 0.3], [0, 0, 0]],
            ["Box", [-2, -4, 2], scale, [0.2, 0.3, 0.4], [0, 0, 0]],
            ["Box", [2, -4, -2], scale, [0.3, 0.4, 0.5], [0, 0, 0]],
            ["Box", [-2, -4, -2], scale, [0.4, 0.5, 0.6], [0, 0, 0]]
    end
end
```
13a. A model with a class that has multiple constructor parameters.

```plaintext
class Main(simulator)
  private end
end

class Boxes(pos, scale)
  private
  .3D := ["Box", pos + [2, 0, 2], scale, [0.1, 0.2, 0.3], [0, 0, 0]],
  ["Box", pos + [-2, 0, 2], scale, [0.2, 0.3, 0.4], [0, 0, 0]],
  ["Box", pos + [2, 0, -2], scale, [0.3, 0.4, 0.5], [0, 0, 0]],
  ["Box", pos + [-2, 0, -2], scale, [0.4, 0.5, 0.6], [0, 0, 0]]
end
end
```

**ACTION:** instantiate class

13b. From the “Add Instance” menu choose class Boxes, then add an instance to class Main.

13c. Instantiating the Boxes class results in an error message. The updated model has placeholders passed as arguments to the class instance.

```plaintext
class Main(simulator)
  private b0 := create Boxes(pos0, scale0) end
end

class Boxes(pos, scale)
  private
  .3D := ["Box", pos + [2, 0, 2], scale, [0.1, 0.2, 0.3], [0, 0, 0]],
  ["Box", pos + [-2, 0, 2], scale, [0.2, 0.3, 0.4], [0, 0, 0]],
  ["Box", pos + [2, 0, -2], scale, [0.3, 0.4, 0.5], [0, 0, 0]],
  ["Box", pos + [-2, 0, -2], scale, [0.4, 0.5, 0.6], [0, 0, 0]]
end
end
```
CASE 14

14a. Using a model containing comments results in an error message.

```java
// This is a test
class Main(simulator)
    private
end
end
```

ACTION: load the model in the visual editor

14b. An error message appears.
DISCUSSION AND FUTURE WORK

4.1 IMPLEMENTATION ASSESSMENT

An important question to be answered is how close did the implementation come to the goals set when the thesis was initially formulated. There is definitely a discrepancy in terms of planned features and actually implemented ones. Here we examine the final product in regard of three different criteria.

Were all planned features implemented?

The answer here is that while a lot of the features were implemented, some are still pending or only partially functional. The proposal initially envisioned allowing the user to “zoom” in and out of the 3D representation of classes, virtually enabling the modeller to drill down into finer and finer details. This behavior could be really useful when working on isolated parts of a bigger project with many classes. Another planned task was to have some kind of formal verification for the interactive transformations applied on the AST, so that we can guarantee that the mutations preserve the initial correctness of the textual model. While this sounds as an interesting project, there was not enough time to focus on that part. Another aspect worth mentioning is the handling of arithmetic expressions as parameters for 3D shapes. This functionality proved to be much more difficult to implement right, because on the one hand it would require evaluation capabilities for arbitrary arithmetic expressions, and on the other it is not clear which parts of those expressions need to be adjusted when a transformation in 3D space is done interactively.

What is the overall quality and does the final product feel well polished and feature complete?

The test cases supplied with our implementation can definitely be improved and enriched to cover more cases. Some of the decisions regarding the code structure were taken when first becoming familiar with jMonkeyEngine and might not follow best practices. As a consequence, there is room for changes that modularise the code and make it easier to test.

Certain aspects of the implementation, mostly dealing with usability, are still rough. An example is the manipulation of objects in
3D, which is not comparable with the quality of commercial modeling tools:

- Other tools have much more sophisticated and finer control over the axes of rotation.
- Grouping of objects on a logical level different from the Acumen class abstraction can be potentially useful.
- Better visual guides can be implemented which help the user identify object identity and belonging.

How flexible is the implementation and is it generic enough to allow easy addition of features and extensibility?

Again, many decisions regarding the implementation were taken while still exploring and learning jMonkeyEngine. Also some features were initially implemented for very basic cases and later expanded to cover more sophisticated ones. This organic evolution of the code sometimes makes it difficult to extend and changes have deep impact and require code refactoring or restructuring.

4.2 Missing Features and Known Issues

The implementation of the proposed solution has some known shortcomings. To ease future development, we present a list of known issues which can be addressed to immediately improve the features and behavior of the interactive editor:

- Rotation of basic shapes has a very naive implementation and is not wired to update the AST.
- Comments and indentation are not preserved in the AST and models containing comments are refused to be loaded in the editor. Changes to the Acumen parser and AST types are needed to overcome this.
- Expressions can not be used as variable values. This limits the expressiveness of Acumen for models that can be loaded in the editor, but is not trivial to implement since it requires expression evaluation to be performed prior to simulation.
- Shapes are currently resized by scaling in all directions. Any interactive modeller should support arbitrary resizing of the primitives. Making this work would require an overhaul of the shape selection mechanics.
- Moving shapes when they have a summation passed as position argument is not fully implemented. Currently only a special
One big improvement would be a proper implementation of node rotations. Commercial software uses sophisticated mechanisms to allow many degrees of freedom when transforming meshes, but even a simple implementation which lets the user choose an axis to rotate around will add a lot of value to our editor. Similar criteria applies to performing non-uniform transforms, for example when scaling meshes in arbitrary directions. Giving the users an intuitive way to achieve that will make a big difference in the perceived usability of the editor.

In regards to usability, the feedback provided by the interactive editor can be improved a lot. One option is to enrich the HUD with more state information, for example detailed positional and transformation data for the selected node. Another would be to have a visual indication such as highlighting in the text view reflecting which shapes are being currently manipulated in the 3D view. A search function for arbitrary instances based on class name criteria might improve the navigation of scenes. It can also make it easier to select object not by using the mouse, but by applying textual filters. This
feature also hints at the possibility to group objects which are not related semantically in Acumen. Performing transformations on such object groups might be an interesting side project.

There is also room for improvement in the messages presented to the user. They can contain a detailed explanation when actions can not be performed (for example when the loading of a model is refused due to presence of comments) and will give the user valuable information which will let them work around the shortcomings of the software.

Additional features to be considered might include the copying, pasting and cloning of already existing instances. This needs careful consideration, since pasting needs to preserve the model correctness and the parameters needed to be passed to an instance might not exist at the point where it is pasted. The addition of a library of commonly used visual elements (including the ability to define own custom ones) can save many hours of work and speed up the modeling process.

An interesting venue for future research is experimentation with VR headsets and gesture input to create the 3D model.
CONCLUSION

The thesis examined the question whether a 3D interface can be successfully used for visual interaction with a hybrid modeling language. Further it looked into the benefits of creating a bi-directional mapping between the textual representation of simulations and easily editable 3D objects.

The implementation accompanying the thesis showed the viability of the interactive interface and demonstrated many of the basic capabilities that such a system should implement. It explored the application of 3D manipulation primitives, as are often found in commercial modeling software, to a partial view of hybrid models in Acumen. We showcased a visual interface similar to established commercial applications which provides benefits to new adopters and eases the workflow of long-time users. The ability to directly interact with objects in the 3D scene shows the potential that such technology has to engage non-technical users in the design process of hybrid models.

Some of the bigger challenges encountered while working on the thesis are listed below:

- The lack of a structured source code editor makes applying small local changes to the source code very difficult. The current implementation analyzes the textual code at once and renders it in 3D, and conversely the 3D changes are applied to the AST nodes and synthesized into a full textual Acumen model. An AST-aware code editor could provide seamless bi-directional mapping between source code and 3D objects.

Having a clean interface between the AST and 3D view and well defined serialization protocol can be also beneficial for further studies. One such tasks would be interfacing with 3rd party software like SketchUp [16] to offload the actual 3D manipulation from Acumen.

- Acumen expressions are very powerful and allow the formulation of sophisticated equations. Due to the simplistic nature of our implementation, the 3D view only deals with constant expressions. Adding support for algebraic computations can be the topic of further research.

- The visualization is limited to the discrete laws of the model and there is no support for the continuous ones. The reason for this is the difficulty to capture graphically their ever-changing nature. As with the previous point, future studies in this area can try come up with an approach to perform evaluation and
make it possible to capture values of continuous parameters at certain times.

Many of the limitations of the current implementations can be seen as topics for future studies. Improvements can be made both in the scope of visualization and manipulation, as well as in the quality and number of tools available to the user.
APPENDIX A

KEYBOARD SHOR TCUTS

• Select (Alt+E)
• Pan (Alt+A)
• Orbit (Alt+O)
• Zoom (Alt+Z)
• Rotate (Alt+R)
• Scale (Alt+C)
• Move (Alt+M)
• Delete (Alt+D)
• Change Color (Alt+K)
ACUMEN SOCKET INTERFACE

To enable interoperability with other software simulation packages, we have implemented a socket interface that exposes Acumen’s internal state to the outside world. The implementation adds a socket server listening on a predefined port. Acumen blocks until a client is connected, after which the simulation starts running. On each simulation step the entire internal state is serialized in JavaScript Object Notation (JSON) format and sent over the network. The simulation continues with the state data read and deserialized back from the external source (see Figure 16):

To start Acumen in server mode the user passes as arguments the model to be executed and the port number on which to listen for a connection:

```
java -jar acumen-VERSION.jar <acumen-model-file> listen <port>
```

A concrete example that runs a bouncing ball simulation and listens on port 5000 would be:

```
java -jar acumen-VERSION.jar bouncing_ball_3d.acm listen 5000
```

The Acumen distribution supplies example echo clients written in Java and OCaml that can be used as starting point for further development.

There is a wide range of applications for this kind of interface. For example, much more specialized network protocol and traffic simulation packages can use Acumen’s hybrid models to create co-simulated environments. The blocking nature of the communication allows easy synchronisation between the participants and presents a stepping stone in the direction of further research in the area of distributed computation and cooperation.
BIBLIOGRAPHY


DECLARATION

I declare, that I have made the presented thesis myself and solely with the aid of the means permitted by the examination regulations of Halmstad University. The literature used is indicated in the bibliography. I have indicated literally or correspondingly assumed contents as such.

The text of the above declaration was taken from [21].

London, July 2017

Viktor Vasilev